

TempoDB: A Modern Time-Series Database - Design Document

Overview

This document outlines the design of TempoDB, a specialized database for storing and querying high-volume, timestamped data. It solves the core architectural challenge of achieving high write throughput for sequential data while enabling efficient range queries and compressing massive datasets with predictable patterns, making it essential for metrics, IoT, and financial applications.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

Milestone(s): This foundational section establishes the core challenges addressed by all five milestones.

Time-series data—sequential measurements indexed by time—is one of the fastest-growing data categories in modern computing. From monitoring server CPU usage every second to tracking sensor readings in industrial IoT and recording financial trades at millisecond intervals, this data forms the observational backbone of digital systems. However, storing and querying this data at scale presents unique architectural challenges that traditional relational or NoSQL databases struggle to address efficiently.

The fundamental characteristics of time-series data create a distinct set of requirements:

Characteristic	Implication for Database Design
High Volume & Velocity	Individual measurements are typically small (a timestamp and value), but arrive continuously in massive volumes—potentially millions of points per second. The database must sustain high write throughput without compromising durability.
Append-Heavy, Rarely Updated	Once recorded, a data point is almost never modified (though it may be deleted after retention periods). This allows for write-optimized storage structures that differ from update-friendly B-trees.
Time-Ordered Nature	Points arrive roughly in chronological order and are most frequently queried by time ranges ("last hour," "yesterday"). Temporal locality can be leveraged for compression and query optimization.
Periodic Patterns	Many time series exhibit regular patterns (daily cycles, seasonal trends) where consecutive values change slowly, enabling specialized compression algorithms far more efficient than general-purpose compression.
Exploding Cardinality	The combination of measurement names and tag combinations creates potentially millions of unique time series, requiring careful indexing strategies to avoid exponential growth in metadata overhead.
Decreasing Value Over Time	Recent data is queried frequently with high resolution, while older data is accessed less often and can be aggregated or downsampled to save storage space. This enables tiered storage strategies.

These characteristics create a fundamental tension: write paths must handle a relentless stream of small, sequential inserts, while read paths must efficiently retrieve and aggregate potentially vast ranges of historical data. Traditional databases optimized for transactional workloads (ACID guarantees, random updates) or document stores (flexible schemas, point lookups) fail to deliver the required performance at scale without excessive operational overhead.

Mental Model: The Data Stream Conveyor Belt

Imagine time-series data as a **high-speed conveyor belt** in a massive warehouse processing facility. Items (data points) arrive continuously, each with a timestamp (arrival time) and content (measurement value), plus labels indicating their type and origin (tags).

```
[2023-10-01 12:00:00, CPU=85%, server=web-01] --->
[2023-10-01 12:00:01, CPU=87%, server=web-01] --->
[2023-10-01 12:00:00, Temp=72.5F, sensor=A] --->
[2023-10-01 12:00:01, Temp=72.6F, sensor=A] --->
```

Our database system acts as the entire warehouse logistics operation, which must:

1. **Label and Sort** Each item must be categorized (by measurement and tags) and sorted chronologically as it arrives. Items arriving out of order (late packages) must be inserted into their correct position.
2. **Compress Efficiently** Since similar items arrive consecutively (temperature readings change slowly), we can pack them tightly using specialized compression—storing only the differences between consecutive items rather than full representations.
3. **Store in Time-Bound Containers** Rather than placing items individually on shelves, we batch them into time-bound containers (e.g., "12:00-12:05 container for server web-01 CPU"). Each container has a clear time range label on the outside, so we can quickly locate relevant containers without opening them.
4. **Move Through Storage Tiers** Hot items (recent arrivals) remain near the front on easily accessible shelves (memory, fast SSDs). As items age, they move to deeper warehouse aisles (slower disks) and eventually to archival storage (cold storage), possibly in aggregated form (e.g., "average temperature per hour" rather than per-second readings).
5. **Query with Temporal Precision** When someone asks, "What was the CPU usage between 12:00 and 12:30?", we don't scan every item in the warehouse. We first consult our container index, retrieve only containers overlapping that time range, then open and decompress just those containers.

This mental model highlights why generic storage systems struggle: they're designed for random access to individual items (like a traditional library where books are constantly reshelfed), not for processing a continuous, time-ordered stream. Our design must embrace the conveyor belt nature—high-throughput ingestion at the front, intelligent batching and compression in the middle, and tiered archiving at the back.

Existing Approaches and Comparison

When faced with time-series data, architects typically consider three categories of storage solutions, each with distinct trade-offs:

Option 1: Generic Relational Databases (PostgreSQL, MySQL)

These systems store time-series data in conventional tables, often with a schema like:

```
CREATE TABLE metrics (
    timestamp TIMESTAMP,
    metric_name VARCHAR(255),
    value DOUBLE PRECISION,
    tags JSONB
);
```

SQL

ADR: Choosing a Specialized Time-Series Database Over Generic RDBMS

- **Context:** Teams often start with existing relational databases due to familiarity and existing infrastructure, but face scaling challenges as time-series data volume grows.
- **Options Considered:**
 1. **Naive table with indexes:** Simple `CREATE INDEX ON metrics(timestamp, metric_name)`
 2. **Table partitioning by time:** Manual or automatic partitioning by time ranges
 3. **TimescaleDB extension:** PostgreSQL extension adding time-series optimizations
- **Decision:** Build a specialized time-series database rather than using a generic RDBMS.
- **Rationale:**
 1. **Write amplification:** B-trees (the standard RDBMS index) incur significant overhead for append-heavy workloads due to random writes and page splits.
 2. **Storage inefficiency:** Row-based storage duplicates timestamp and tag columns for every measurement, while time-series compression can achieve 10-100x better ratios.
 3. **Query performance:** Range scans must traverse index trees not optimized for temporal locality, missing opportunities for block-level skipping.
 4. **Cardinality management:** High cardinality from tag combinations creates index bloat in RDBMS indexes.
- **Consequences:** We forgo full SQL support and ACID transactions across arbitrary tables, but gain order-of-magnitude improvements in write throughput, storage efficiency, and time-range query performance.

Approach	Pros	Cons	Why Not for High-Volume TSDB
Plain RDBMS table	Full SQL, ACID transactions, well-understood	Poor write performance, storage bloat, index overhead on timestamps	B-tree index maintenance dominates write cost at high volumes
Time partitioning	Faster deletion of old data, some query pruning	Manual management, still uses row storage, doesn't help compression	Reduces but doesn't eliminate fundamental architectural mismatches
TimescaleDB	PostgreSQL compatibility, automated partitioning, some compression	Still bound by PostgreSQL's storage engine, higher latency than specialized TSDB	Good hybrid solution but not optimal for pure time-series workloads at extreme scale

Option 2: Specialized Time-Series Databases (InfluxDB, Prometheus)

These systems are built from the ground up for time-series data, employing specialized storage engines:

- **InfluxDB TSM Engine:** Uses a Time-Structured Merge Tree with columnar storage within blocks, Gorilla compression for floats, and time-range indexing.
- **Prometheus TSDB:** Uses a custom storage format with chunk-based encoding, separate head block for recent data, and compaction to larger chunks.

ADR: Learning from Existing TSDB Architectures Without Direct Copying

- **Context:** Multiple mature time-series databases exist with proven designs. We must decide how closely to follow existing patterns versus innovating.
- **Options Considered:**
 1. **Implement InfluxDB's TSM engine exactly:** Clone the open-source implementation
 2. **Implement Prometheus TSDB:** Adopt its chunk-based approach
 3. **Synthesize patterns with educational clarity:** Combine proven ideas with explicit design decisions and trade-offs
- **Decision:** Create a synthesis that highlights architectural decisions clearly while using proven patterns from industry.
- **Rationale:**
 1. **Educational value:** Direct cloning teaches implementation but not reasoning. Our design should expose decision points.
 2. **Simplification for learning:** Production systems include many optimizations for edge cases that obscure core concepts.
 3. **Pattern diversity:** Different systems excel in different aspects—InfluxDB's compression, Prometheus's chunk management, TimescaleDB's SQL integration.
- **Consequences:** Our design will resemble InfluxDB's TSM engine in structure but with simplified compaction, explicit ADRs for each major choice, and clear mapping from concepts to implementation.

System	Storage Engine	Compression	Query Language	Best For
InfluxDB	Time-Structured Merge Tree (TSM)	Gorilla XOR for floats, delta-of-delta for timestamps	InfluxQL (SQL-like), Flux	High-write environments, operational monitoring
Prometheus	Custom chunk-based TSDB	Variable encoding based on data type	PromQL (functional)	Kubernetes/metrics monitoring, pull-based collection
TimescaleDB	PostgreSQL with hypertables	Native compression, dictionary encoding	Full PostgreSQL SQL	Teams requiring SQL, mixed workload environments
ClickHouse	MergeTree with LSM-like merges	Multiple codecs, delta encoding	SQL	Analytics, high-cardinality time-series

Option 3: General-Purpose Columnar Stores (Apache Parquet, Arrow)

Columnar storage formats naturally align with time-series characteristics by storing all timestamps together and all values together:

Approach	Alignment with Time-Series	Challenges
Columnar formats	Excellent for compression (similar values stored contiguously), efficient for range scans	Not optimized for real-time ingestion, lack built-in time-series operations
Data lake architectures	Cost-effective at petabyte scale, good for batch analytics	High query latency, not designed for real-time monitoring

Key Insight: While columnar storage is theoretically ideal for time-series data, the missing piece is **ingestion efficiency**. Traditional columnar stores expect batch writes, while time-series requires sustained high-velocity ingestion with immediate queryability.

Synthesis: The TempoDB Approach

TempoDB adopts a **hybrid architecture** that combines the best elements of these approaches:

1. **Specialized storage engine** based on Time-Structured Merge Tree principles (like InfluxDB) for high write throughput and compression
2. **Columnar layout within blocks** (like Parquet) to enable efficient compression and scanning
3. **SQL-like query language with time extensions** (bridging InfluxQL and TimescaleDB approaches) for familiarity
4. **Built-in aggregation and downsampling** (like Prometheus) for data lifecycle management

The core innovation isn't in individual components—which are proven in production—but in their **pedagogical integration**: each design decision is explicitly justified, alternatives are documented, and implementation maps clearly to architectural concepts.

Common Pitfalls in Time-Series Database Design

⚠️ Pitfall: Treating Timestamps as Just Another Column

- **Description:** Storing timestamps in generic indexes without leveraging their monotonic, range-query-dominant nature.
- **Why Wrong:** Generic indexes (B-trees) optimized for equality lookups waste space and CPU on timestamp ordering that's already inherent in append pattern.
- **Fix:** Implement time-range indexes that store minimum/maximum timestamps per block, enabling entire blocks to be skipped during queries.

⚠️ Pitfall: Naive Tag Indexing Causing Cardinality Explosion

- **Description:** Creating separate indexes for each tag dimension leads to combinatorial explosion of index entries.
- **Why Wrong:** With 10 servers, 5 regions, and 3 applications, you have 150 series. Each point writes to 150 index entries in naive implementation.
- **Fix:** Use inverted index or tag-value pair indexing where each unique series key (measurement + tag combination) is indexed once, not per tag.

⚠️ Pitfall: Applying General-Purpose Compression

- **Description:** Using gzip or Snappy on entire data blocks without considering time-series patterns.
- **Why Wrong:** Misses opportunity for order-of-magnitude better compression using delta-of-delta (timestamps) and XOR (floating point values).
- **Fix:** Implement specialized time-series compression algorithms like Gorilla for floats and delta encoding for integers.

⚠️ Pitfall: Ignoring Out-of-Order Writes

- **Description:** Assuming timestamps always arrive in strictly increasing order.
- **Why Wrong:** Network latency, clock skew, and batch processing can cause "late" data points that arrive after newer points.
- **Fix:** Design write path to handle out-of-order data through buffering and periodic re-sorting during compaction.

⚠️ Pitfall: Querying Full Resolution for Historical Data

- **Description:** Storing and querying millisecond data for years-old metrics.
- **Why Wrong:** Wastes storage and query CPU on precision no longer needed for trend analysis.

- **Fix:** Implement automatic downsampling and retention policies that aggregate old data to lower resolutions.

The subsequent sections of this design document address these pitfalls through specific architectural choices, each documented with rationale and alternatives.

Implementation Guidance

Note: This section focuses on conceptual understanding rather than implementation. Detailed implementation guidance appears in later sections corresponding to each milestone. However, we provide foundational guidance here for setting up the project structure.

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option (Alternative)
Language	Go (static typing, GC, good concurrency)	Rust (zero-cost abstractions, no GC)
Storage Abstraction	Memory-mapped files with <code>mmap</code> syscall	Direct I/O with aligned buffers
Concurrency Control	<code>sync.RWMutex</code> for shared mutable state	Lock-free ring buffers for writes
Serialization	Custom binary format with <code>encoding/binary</code>	Protocol Buffers for metadata
HTTP Server	Standard <code>net/http</code> with JSON/plaintext	FastHTTP for higher performance
Compression	Custom Gorilla/delta implementations	Integrate <code>github.com/klauspost/compress</code>

B. Recommended File/Module Structure

Establish this directory structure from the beginning to maintain separation of concerns:

```
tempo/
├── cmd/
│   ├── tempo-server/           # Main server executable
│   │   └── main.go
│   └── tempo-cli/             # Command-line interface (optional)
│       └── main.go
├── internal/                 # Private application code
│   ├── api/                   # HTTP/gRPC APIs (Milestone 5)
│   │   ├── handler.go
│   │   ├── query_parser.go
│   │   └── line_protocol.go
│   ├── storage/               # Storage engine (Milestone 1)
│   │   ├── tsm/
│   │   │   ├── writer.go
│   │   │   ├── reader.go
│   │   │   ├── compression.go
│   │   │   └── file.go
│   │   ├── wal/                 # Write-ahead log (Milestone 2)
│   │   │   ├── writer.go
│   │   │   ├── reader.go
│   │   │   └── segment.go
│   │   ├── memtable/            # In-memory buffer (Milestone 2)
│   │   │   ├── memtable.go
│   │   │   └── series_key.go
│   │   └── index/                # Series and tag indexing
│   │       ├── inverted.go
│   │       └── series_store.go
│   ├── query/                  # Query engine (Milestone 3)
│   │   ├── parser/
│   │   │   ├── lexer.go
│   │   │   └── parser.go
│   │   ├── executor/
│   │   │   ├── planner.go
│   │   │   ├── iterator.go
│   │   │   └── aggregator.go
│   │   └── engine.go
│   ├── compact/                # Compaction & retention (Milestone 4)
│   │   ├── compactor.go
│   │   ├── retention.go
│   │   └── downampler.go
│   └── models/                 # Shared data structures
│       ├── point.go
│       ├── series.go
│       └── query.go
└── pkg/                      # Public libraries (if any)
└── scripts/                  # Build/test scripts
└── testdata/                 # Test fixtures
└── go.mod
└── go.sum
└── README.md
```

C. Core Type Definitions (Foundation)

Before implementing any component, define these foundational types in `internal/models/`:

```
// File: internal/models/point.go                                         GO

package models

import (
    "time"
)

// DataPoint represents a single time-series measurement at a specific timestamp

type DataPoint struct {
    Timestamp time.Time // Precision: nanosecond
    Value     float64   // Supports integers via conversion
    // Note: In a production system, we'd support multiple value types (int, bool, string)
}

// SeriesKey uniquely identifies a time series by measurement name and tags

type SeriesKey struct {
    Measurement string           // e.g., "cpu_usage"
    Tags         map[string]string // e.g., {"host": "server-01", "region": "us-east"}
}

// Series represents a complete time series: its identity and data points

type Series struct {
    Key      SeriesKey
    Points  []DataPoint // In practice, points are stored in blocks, not individually
}
```

GO

```
// File: internal/models/query.go

package models

import (
    "time"
)

// TimeRange represents an inclusive time window [Start, End]

type TimeRange struct {
    Start time.Time
    End   time.Time
}

// AggregateFunction defines supported aggregation operations

type AggregateFunction string

const (
    AggregateSum AggregateFunction = "sum"
    AggregateAvg AggregateFunction = "avg"
    AggregateMin AggregateFunction = "min"
    AggregateMax AggregateFunction = "max"
    AggregateCount AggregateFunction = "count"
)

// Query represents a parsed query request

type Query struct {
    Measurement  string
    Tags         map[string]string // Tag filters (AND relationship)
    TimeRange    TimeRange
}
```

```

Aggregate      *AggregateFunction      // nil for raw points

GroupByWindow time.Duration          // e.g., 1h for hourly aggregation

Fields         []string              // For future: multiple fields per measurement

}

```

D. Language-Specific Hints for Go

1. **Time Handling:** Use `time.Time` for timestamps throughout. Convert to nanoseconds for storage:
`timestamp.UnixNano()`.
2. **Concurrent Maps:** Use `sync.Map` or `sync.RWMutex` with regular maps for shared mutable state. For high-throughput write buffers, consider sharded maps.
3. **Binary Encoding:** Use `encoding/binary.BigEndian.PutUint64()` and similar for consistent cross-platform binary format.
4. **Memory Management:** Preallocate slices with known capacity to avoid reallocations: `points := make([]DataPoint, 0, 1024)`.
5. **Error Handling:** Use custom error types for domain-specific errors (e.g., `ErrOutOfOrderTimestamp`, `ErrSeriesNotFound`).
6. **Testing:** Use table-driven tests extensively. Create golden files for binary format verification.

E. First Milestone Checkpoint (Conceptual)

Before writing code, validate your understanding by answering:

1. **Write down** three reasons why B-trees are suboptimal for time-series writes compared to TSM trees.
2. **Sketch** how delta-of-delta encoding would compress this timestamp sequence (in nanoseconds): `[1000, 2000, 3000, 4500, 6000]`
3. **Explain** in one sentence why columnar storage within blocks helps compression for time-series data.

Expected answers:

1. B-trees cause random writes for sequential data, have high overhead for small inserts, and don't leverage temporal locality for compression.
2. First delta: `[1000, 1000, 1000, 1500, 1500]`; Delta-of-delta: `[1000, 0, 0, 500, 0]` (most values are 0, compressible).
3. Columnar storage groups similar values (timestamps with timestamps, values with values) allowing specialized compression algorithms to exploit patterns within each column.

This foundational understanding will guide implementation decisions throughout the project.

Milestone(s): This section establishes the foundational scope that guides all five milestones.

Goals and Non-Goals

This section clearly defines the boundaries of the TempoDB project. Establishing precise goals and, equally importantly, explicit non-goals is critical for building a focused, learnable system. Time-series databases are complex and can expand into numerous features (distributed scaling, full SQL, complex analytics). By defining a narrow but deep scope, we ensure we build a complete, working system that demonstrates the *core architectural patterns* of a time-series database without being overwhelmed by auxiliary concerns.

Goals

The primary goal of TempoDB is to create a single-node, educational time-series database that demonstrates the essential architectural patterns required to handle high-velocity, sequential data. The system must be **complete enough to be useful** for basic metrics collection and querying, and **pedagogically sound** to illustrate the key trade-offs in storage, ingestion, and query design. All goals are prioritized for clarity of implementation and learning value over production-grade robustness or extreme performance.

The following table enumerates the specific functional and non-functional requirements that TempoDB *must* deliver.

Category	Goal	Rationale & Learning Outcome
Write Throughput	Sustain thousands of writes per second on commodity hardware.	Demonstrates the importance of batching, buffering, and sequential I/O patterns for high-ingestion workloads, contrasting with random-write databases.
Storage Efficiency	Achieve significant compression (targeting 10x or better for regular metrics) via columnar storage , delta-of-delta encoding for timestamps, and Gorilla compression for floating-point values.	Teaches how to exploit the predictable patterns (monotonic timestamps, slowly-changing values) inherent in time-series data to reduce storage costs.
Query Performance	Support efficient range queries (e.g., "last hour of data") by scanning only relevant time blocks, using a block-based index with min/max timestamps.	Illustrates the principle of temporal locality and how indexing by time primary key is fundamentally different from indexing by arbitrary keys.
Data Model	Implement an InfluxDB-like data model with measurements , tags (indexed key-value pairs), and fields (values). Support series cardinality tracking.	Provides a practical, industry-relevant model for organizing metrics and demonstrates the performance implications of high cardinality.
Query Language	Provide a SQL-like query language supporting <code>SELECT</code> , <code>FROM</code> , <code>WHERE</code> (time and tag filters), and <code>GROUP BY time()</code> intervals with built-in aggregation functions (sum, avg, min, max, count).	Teaches query parsing, planning, and the pushdown of time-range predicates and simple aggregations to the storage layer.
Durability	Guarantee write durability through a Write-Ahead Log (WAL) . Acknowledged writes must survive process crashes.	Introduces the fundamental pattern of logging before applying changes, which is critical for any reliable database system.
Data Lifecycle	Enforce Time-To-Live (TTL) policies for automatic data expiration and implement background compaction to merge small files and reclaim space.	Shows how to manage the lifecycle of time-series data, which is typically valued less as it ages, and how to maintain read performance over time.
Downsampling	Support downsampling queries that return lower-resolution, aggregated views of raw data (e.g., 1-minute averages from 1-second data).	Demonstrates a key analytical operation for visualizing long time ranges and pre-computing rollups for performance.

Category	Goal	Rationale & Learning Outcome
Operational Simplicity	Expose a simple HTTP API for writes (compatible with InfluxDB line protocol) and queries, and Grafana compatibility for visualization.	Makes the system tangible and testable with common tools, reinforcing the connection between architecture and user-facing value.
Code Clarity	The implementation should be well-structured, modular, and documented to serve as a learning artifact. Primary logic should avoid unnecessary concurrency complexity initially.	The ultimate goal is education; the code must be readable and traceable to the design concepts explained in this document.

Key Design Principle: *Depth over Breadth.* Each implemented feature should be built to its logical conclusion, demonstrating the complete data path from API to disk and back, rather than sketching many half-features.

Non-Functional Goal Elaboration

- **Performance Profile:** Write latency should be predictable and dominated by periodic disk flushes rather than per-point overhead. Read latency for recent data should be sub-second, utilizing in-memory structures (memtable). Full historical scans will be slower but bounded by sequential disk I/O.
- **Resource Usage:** The database should be memory-efficient, avoiding loading entire datasets into memory. Compression reduces disk footprint. The system should be stable under sustained write load, employing backpressure when necessary.
- **Correctness:** The system must provide durable writes and correct query results. For the educational scope, we prioritize "eventual" correctness after crashes (via WAL replay) over complex transactional guarantees.

Non-Goals

Explicitly stating what TempoDB will *not* do is essential to prevent scope creep and to focus effort on the core learning objectives. The following features are deliberately excluded from the initial design. Many are important for a production system but represent orthogonal complexities that can be studied independently after mastering the fundamentals.

Feature	Reason for Exclusion	Potential Learning Extension
Horizontal Scaling / Clustering	Distributing data across multiple nodes introduces complex problems of consistency, replication, and global querying (e.g., consensus, vector clocks, distributed query planning). This is a vast topic deserving its own dedicated study.	A future extension could add a simple sharding layer based on measurement name or hash of series key.
Full SQL Support	Implementing a complete SQL parser, optimizer, and executor with joins, subqueries, and complex expressions is a massive undertaking that distracts from time-series-specific optimizations.	The simple query language can be extended with more functions and syntax over time.
Cross-Series Transactions (ACID)	Time-series data is primarily append-only. Supporting multi-series, multi-statement transactions with rollback adds significant complexity (locking, undo logs) for limited benefit in the target use cases.	Basic single-series atomicity is provided by the WAL.
Advanced Security (RBAC, Encryption)	Authentication, authorization, and encryption are critical for production but are generic infrastructure concerns not unique to time-series databases.	Can be added via a middleware proxy or later integration.
Continuous Queries	While a common feature (pre-computing aggregations in real-time), they are essentially a streaming computation layer on top of the write path. They add significant scheduling and state management complexity.	Can be implemented as a separate background job scheduler that runs periodic aggregation queries.
Tiered Storage (Auto-migration to S3)	Automatically moving cold data to object storage involves lifecycle management, network I/O, and possibly different file formats. It's a valuable production feature but an orthogonal storage layer concern.	A manual "export to S3" command could be a simpler first step.
Advanced Data Types	Support for complex types (arrays, histograms, geospatial) or efficient compression for integers, booleans, and strings. We focus on floating-point numbers as the most common and pedagogically interesting case for compression.	New compression schemes (e.g., for integers) can be added as pluggable codecs.
High Availability (Replication, Failover)	Ensuring the database survives machine failures requires replication and automatic failover, which again enters the domain of distributed systems.	A simple primary-standby replication using WAL shipping could be a follow-on project.

Feature	Reason for Exclusion	Potential Learning Extension
Sophisticated Memory Management	Production databases often use custom memory allocators, page caches, and direct I/O to optimize performance. These are advanced system programming topics.	We rely on Go's runtime and standard library for memory and file I/O for simplicity.
Pluggable Storage Backends	Abstracting the storage engine to support multiple backends (e.g., PostgreSQL, cloud storage) adds abstraction overhead and obscures the core TSM engine design.	The TSM engine <i>is</i> the core learning artifact.

Key Design Principle: *Defer Complexity.* The system is designed to be a **vertical slice** of a time-series database, not a **horizontal platform**. By deferring or omitting these features, we maintain a clear line of sight from the core data structures on disk to the user-facing API, which is the primary educational objective.

The "Not Now" vs. "Not Ever" Distinction

For the purposes of this project and design document, the listed non-goals are considered out of scope. However, the architecture does not intentionally preclude their future addition. For example:

- The **SeriesKey** structure could be extended.
- The **TSM file format** includes versioning in its header for future evolution.
- The **HTTP API** could be extended with new endpoints.

The decision to exclude a feature is based on its impact on the **initial learning journey**, not on its inherent value.

Summary of Scope

TempoDB will be a fully functional, single-node time-series database that excels at ingesting and compressing regular measurements and answering time-range queries with aggregations. It will not be a distributed, highly available, general-purpose SQL database. This focused scope allows us to build a complete system that deeply explores the unique architectural patterns—such as columnar block storage, time-based compaction, and specialized compression—that make time-series databases a distinct and valuable category of data infrastructure.

High-Level Architecture

Milestone(s): This architectural overview provides the foundation for all five milestones, showing how components interact to achieve the system's core capabilities.

TempoDB follows a layered architecture that separates concerns between ingestion, storage, query processing, and maintenance. The core insight is the separation between write-optimized and read-optimized data structures: incoming writes are buffered in memory for fast acknowledgment, then periodically flushed to disk in an optimized columnar format. This pattern, common in modern databases, provides the dual benefits of high write throughput and efficient read performance.

Mental Model: The Time-Series Processing Pipeline

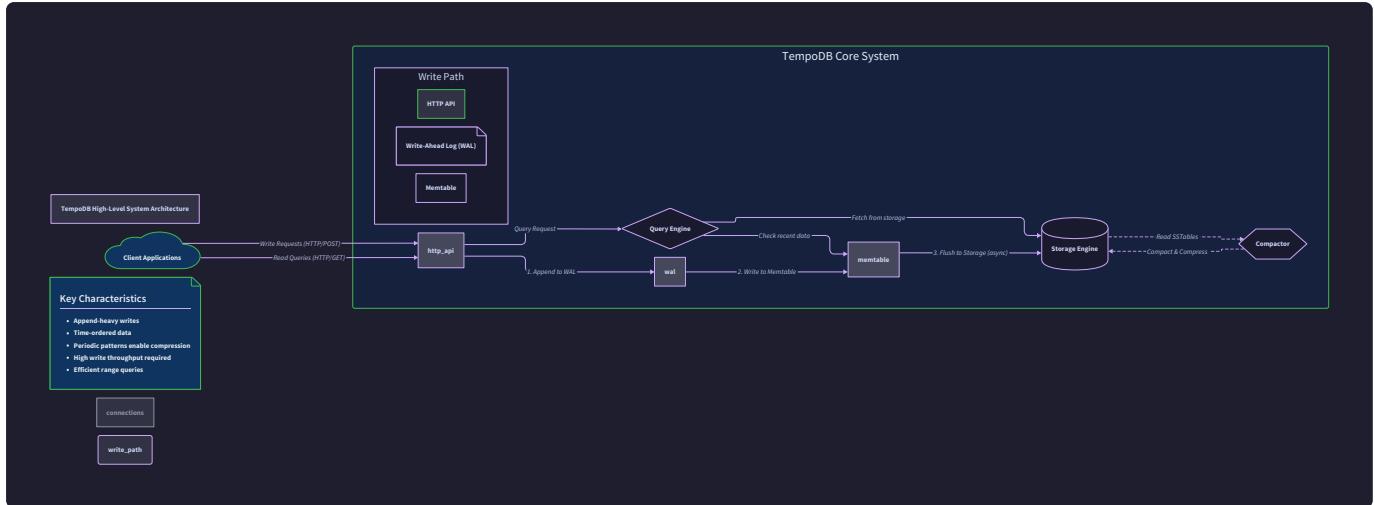
Imagine TempoDB as a specialized factory for processing time-stamped measurements. Raw data arrives continuously on a high-speed conveyor belt (the **write path**). The factory has several specialized stations:

1. **Receiving Dock (Ingest API)**: Validates and categorizes incoming shipments
2. **Receipt Printer (Write-Ahead Log)**: Creates durable records of each shipment before any processing
3. **Sorting Buffer (Memtable)**: Temporarily holds items while they're being organized
4. **Packaging Department (Storage Engine)**: Compresses and packages items into optimized containers (TSM files) for long-term storage
5. **Retrieval Desk (Query Engine)**: Fetches and processes packages based on customer requests
6. **Warehouse Maintenance (Compactor)**: Periodically reorganizes storage, discards expired items, and creates summary catalogs

This mental model helps visualize how data flows from ingestion to storage and back out through queries, with each component having a clear responsibility in the pipeline.

Component Overview and Responsibilities

The following diagram shows the major components and their interactions:



Each component has specific responsibilities and interacts with others through well-defined interfaces. The table below summarizes these responsibilities:

Component	Primary Responsibility	Key Data It Owns	Performance Critical Operations
Ingest API	Accept writes and queries from clients, validate input, format responses	Client connections, request buffers	Request parsing, response serialization
Write-Ahead Log (WAL)	Ensure write durability before acknowledging to client	WAL segment files on disk	Sequential append with <code>fsync</code> , quick recovery scan
Memtable	Buffer writes in memory for fast acknowledgment and batching	In-memory sorted data structure per series	Concurrent insert, range scan for flush
Storage Engine (TSM)	Persist data to disk in optimized columnar format, serve reads	TSM files on disk, block cache	Range scans with predicate pushdown, compression/decompression
Query Engine	Parse, plan, and execute queries, streaming results	Query plans, execution state	Index lookups, aggregation pushdown, result streaming
Compactor	Background maintenance: merge files, delete expired data, create rollups	Compaction state, scheduling metadata	Disk I/O scheduling, resource throttling

Let's examine each component in detail:

Ingest API

The **Ingest API** serves as the system's front door, handling all external communication. It exposes two primary interfaces:

- 1. Write API:** Accepts time-series data points, typically in InfluxDB line protocol format

2. Query API:

Accepts query requests and returns results in JSON or other formats

This component is responsible for request validation, authentication (if implemented), rate limiting, and connection management. It translates external requests into internal operations and formats responses for clients. The API supports both HTTP/REST and (optionally) gRPC interfaces.

Key Design Decisions:

- Use a connection pool to handle concurrent requests efficiently
- Validate all incoming data before processing (timestamp ranges, value types, etc.)
- Implement request timeouts to prevent hung connections from consuming resources
- Support content negotiation for different response formats (JSON, CSV, binary)

Write-Ahead Log (WAL)

The **Write-Ahead Log** provides durability guarantees by recording every write operation to disk before acknowledging it to the client. This follows the "write-ahead logging" principle common in database systems: no modification to persistent data structures occurs without first being logged.

Design Insight: The WAL acts as a "flight recorder" for the database. In the event of a crash, the system can replay the WAL to reconstruct the state of in-memory buffers that were lost, ensuring no acknowledged writes are lost.

The WAL uses a segment-based architecture:

- **Active segment:** Currently being written to
- **Closed segments:** Previously filled segments that can be deleted after their contents are flushed to TSM files
- **Index file:** Maps series keys to their positions in WAL segments for efficient recovery

Each WAL entry contains the full `DataPoint` information (series key, timestamp, value) in a compact binary format. The WAL must support extremely fast sequential writes and periodic `fsync` operations to ensure durability.

Memtable

The **Memtable** (memory table) is an in-memory buffer that holds recently written data points before they're flushed to disk. It serves several purposes:

1. **Write amplification reduction:** Batches multiple writes into single disk operations
2. **Fast acknowledgment:** Allows immediate acknowledgment of writes after WAL logging
3. **Temporal locality:** Recent data (often queried together) remains in memory
4. **Sorting:** Organizes data by series and timestamp before disk persistence

The memtable uses a sorted data structure (typically a skip list or B-tree) keyed by `SeriesKey` and `Timestamp`. When it reaches a configurable size threshold (e.g., 64MB), it's marked as **immutable**, a new **active** memtable is created, and the immutable memtable is scheduled for flushing to disk.

ADR: Choosing Memtable Data Structure

- **Context:** Need fast concurrent inserts and efficient range scans for flushing
- **Options Considered:**
 1. **Concurrent skip list:** Lock-free or fine-grained locking, excellent for range queries
 2. **B-tree with copy-on-write:** Good for ordered traversal, moderate concurrency
 3. **Per-series sorted slice with global lock:** Simple but poor concurrency
- **Decision:** Concurrent skip list implementation
- **Rationale:** Skip lists provide $O(\log n)$ operations with good concurrency characteristics and excellent cache locality for range scans, which are critical during flush operations
- **Consequences:** Higher memory overhead than B-trees (~50% more pointers) but simpler concurrent implementation without complex rebalancing logic

Option	Pros	Cons	Chosen?
Concurrent skip list	Excellent concurrency, good range scan performance, simple implementation	Higher memory overhead (~ $2N$ pointers), not cache-optimal	Yes
B-tree with copy-on-write	Good memory efficiency, excellent for ordered traversal	Complex concurrent modification, rebalancing overhead	No
Per-series sorted slice	Simple, excellent for flushing by series	Poor concurrent write performance, expensive inserts	No

Storage Engine (TSM)

The **Storage Engine** implements the Time-Structured Merge Tree (TSM) format for persistent storage. This is the core of TempoDB's read-optimized storage, where data is organized in a columnar layout with aggressive compression.

The TSM engine manages:

1. **TSM files:** Immutable files containing compressed time-series data
2. **Block cache:** Frequently accessed data blocks kept in memory
3. **File index:** In-memory mapping of series keys to their locations in TSM files
4. **Statistics:** Min/max timestamps, value ranges, and other metadata per block

Each TSM file contains multiple **series blocks**, each holding data for a single time series. Within a series block, data is organized into **data blocks** covering contiguous time ranges. Each data block stores

timestamps and values in separate columns with specialized compression.

Data Flow Through Storage Engine:

1. During flush, the memtable's data is sorted by series and timestamp
2. Each series' data is compressed using delta-of-delta (timestamps) and Gorilla XOR (values)
3. Compressed blocks are written to a new TSM file along with index information
4. The file index is updated to include the new file's time ranges
5. Old TSM files (from previous compactions) may be marked for deletion

Query Engine

The **Query Engine** processes read requests, transforming high-level queries into efficient storage operations. It follows a traditional database query processing pipeline with time-series optimizations:

```
Parse → Plan → Optimize → Execute → Stream
```

The key optimization is **predicate pushdown**: time range and tag filters are applied as early as possible, ideally at the storage layer, to minimize data movement. The query engine also handles:

- **Aggregation**: Built-in functions (sum, avg, min, max, count) with pushdown where possible
- **Downsampling**: GROUP BY time() operations that bucket data into fixed intervals
- **Multi-series queries**: Joining data from multiple series based on time alignment
- **Result streaming**: Progressive return of results to avoid large memory allocations

The engine uses an **iterator model** where each stage of query processing produces a stream of `DataPoint` records that flow through operators (filter, aggregate, transform).

Compactor

The **Compactor** handles background maintenance tasks that optimize storage and enforce data lifecycle policies:

Task	Frequency	Purpose	Resource Impact
Memtable flush	When memtable reaches size threshold	Move data from memory to persistent storage	High I/O (write burst)
TSM compaction	Periodically or when many small files exist	Merge small TSM files into larger ones, remove duplicates	Moderate I/O and CPU
Retention enforcement	Scheduled (e.g., hourly)	Delete data older than retention period	Moderate I/O (file deletion)
Rollup generation	Scheduled for historical data	Create pre-aggregated summaries for fast queries	High CPU, moderate I/O

The compactor uses a **leveled compaction strategy** similar to LSM trees but optimized for time-series:

- **Level 0:** Recently flushed TSM files (may have overlapping time ranges)
- **Level 1-3:** Compacted files with non-overlapping time ranges
- Each level has a target file size, with older data in larger files

Design Insight: Compaction is scheduled based on both time and space heuristics. Time-based scheduling ensures predictable performance impact, while space-based triggers prevent storage inefficiency from accumulating too many small files.

Component Interactions

The components interact through well-defined interfaces and protocols. The following tables describe the key interactions:

Write Path Interactions:

From Component	To Component	Data Passed	Trigger Condition
Ingest API	WAL	SeriesKey + DataPoint	Every write request
WAL	Memtable	SeriesKey + DataPoint	After successful log append
Ingest API	Client	Acknowledgment	After WAL append (or async after memtable)
Memtable	Storage Engine	Batch of DataPoint per series	Memtable reaches size threshold
Storage Engine	Compactor	TSM file metadata	New TSM file created

Read Path Interactions:

From Component	To Component	Data Passed	Purpose
Ingest API	Query Engine	Query structure	Query request
Query Engine	Storage Engine	Time range + series filter	Data retrieval
Storage Engine	Query Engine	Stream of DataPoint	Query results
Query Engine	Ingest API	Formatted results	Response to client

Background Task Interactions:

From Component	To Component	Data Passed	Frequency
Compactor	Storage Engine	List of TSM files to merge	Scheduled or threshold-based
Storage Engine	Compactor	New TSM file after compaction	After compaction completes
Compactor	File System	Paths of expired TSM files	Retention policy schedule

Common Pitfalls in Architecture Design

⚠️ Pitfall: Tight coupling between components

- **Description:** Making components directly depend on each other's internal implementations
- **Why it's wrong:** Makes testing difficult, inhibits independent evolution, complicates debugging
- **Fix:** Define clear interfaces between components, use dependency injection for testing

⚠️ Pitfall: Single-threaded bottleneck in write path

- **Description:** All writes go through a global lock or single goroutine
- **Why it's wrong:** Limits write throughput, doesn't scale with cores
- **Fix:** Use sharded memtables (by series hash), lock-free data structures, or partitioned WAL

⚠️ Pitfall: Inefficient memory management

- **Description:** Allocating individual `DataPoint` objects, causing GC pressure
- **Why it's wrong:** High garbage collection overhead reduces throughput
- **Fix:** Use memory pools, slice-based storage, or arena allocation for batched points

⚠️ Pitfall: Blocking writes during flush

- **Description:** Stopping write acceptance while memtable flushes to disk
- **Why it's wrong:** Creates write latency spikes, reduces availability
- **Fix:** Implement double buffering with active and immutable memtables

⚠️ Pitfall: No backpressure mechanism

- **Description:** Accepting writes indefinitely even when system is overloaded
- **Why it's wrong:** Leads to out-of-memory crashes or severe performance degradation
- **Fix:** Implement write rejection or throttling when buffers exceed thresholds

Implementation Guidance

Technology Recommendations

Component	Simple Option (Recommended)	Advanced Option (Extension)
Ingest API	<code>net/http</code> with JSON/line protocol	gRPC with Protocol Buffers
WAL Persistence	Direct file I/O with <code>os.File</code>	Memory-mapped files with <code>syscall.Mmap</code>
Memtable	Skip list with <code>sync.RWMutex</code>	Lock-free skip list using atomic operations
TSM File I/O	Standard file operations	Memory-mapped file access
Concurrency Control	Channel-based pipelines	Work-stealing scheduler
Compression	Custom delta/Gorilla implementation	Plug-in architecture for multiple algorithms

Recommended File/Module Structure

Organize the codebase as follows to maintain separation of concerns and enable clean testing:

```
tempo/
├── cmd/
│   ├── tempo-server/
│   │   └── main.go          # Server entry point
│   └── tempo-cli/
│       └── main.go          # CLI tool (optional)
└── internal/
    ├── api/
    │   ├── http/
    │   │   ├── server.go      # HTTP server setup
    │   │   ├── handlers.go    # Request handlers
    │   │   └── middleware.go  # Auth, logging middleware
    │   └── query/
    │       ├── parser.go     # Query language parser
    │       ├── planner.go    # Query plan generation
    │       └── executor.go   # Query execution
    ├── storage/
    │   ├── wal/
    │   │   ├── wal.go         # Write-ahead log implementation
    │   │   ├── segment.go     # WAL segment management
    │   │   └── recovery.go    # WAL recovery on startup
    │   ├── memtable/
    │   │   ├── memtable.go    # In-memory table interface
    │   │   ├── skiplist.go    # Skip list implementation
    │   │   └── buffer_pool.go # Memory pool for points
    │   ├── tsm/
    │   │   ├── writer.go       # TSM file writer
    │   │   ├── reader.go       # TSM file reader
    │   │   ├── index.go        # File index management
    │   │   └── compression.go # Compression algorithms
    │   └── engine/
    │       ├── engine.go      # Storage engine facade
    │       ├── cache.go       # Block cache
    │       └── statistics.go  # Storage statistics
    ├── query/
    │   ├── engine.go         # Query engine coordination
    │   ├── iterator.go       # Iterator interfaces
    │   ├── aggregator.go    # Aggregation functions
    │   └── downampler.go    # Downsampling logic
    ├── compaction/
    │   ├── manager.go        # Compaction coordination
    │   ├── planner.go        # Compaction planning
    │   ├── executor.go       # Compaction execution
    │   └── retention.go      # TTL enforcement
    └── meta/
        ├── catalog.go        # Series catalog
        ├── statistics.go     # System statistics
        └── sharding.go       # Shard management (future)
└── pkg/
    ├── models/
    │   ├── point.go          # DataPoint, SeriesKey types
    │   ├── query.go           # Query structure
    │   └── tsm.go             # TSM block structures
```

```
|   └── protocol/
|       ├── lineproto.go          # Line protocol parser
|       └── prometheus.go        # Prometheus remote read/write
|   └── utils/
|       ├── timeutil.go          # Time utilities
|       ├── bytesutil.go         # Byte manipulation
|       └── pool.go              # Object pools
└── test/
    ├── integration/
    |   └── end_to_end_test.go    # Integration tests
    └── benchmarks/
        └── write_bench_test.go  # Performance benchmarks
```

Infrastructure Starter Code

Here's a complete, ready-to-use implementation for the WAL segment management, a common infrastructure component:

GO

```
// internal/storage/wal/segment.go

package wal

import (
    "encoding/binary"
    "fmt"
    "os"
    "path/filepath"
    "sync"
    "time"
)

// Segment represents a single WAL segment file

type Segment struct {
    path      string
    file      *os.File
    mu        sync.RWMutex
    size      int64
    maxSize   int64
    closed    bool
    firstID   uint64 // First entry ID in this segment
    lastID    uint64 // Last entry ID in this segment
}

// SegmentConfig holds configuration for WAL segments

type SegmentConfig struct {
    MaxSizeBytes int64
    SyncInterval time.Duration
}
```

```
SyncOnWrite bool
}

// NewSegment creates or opens a WAL segment file

func NewSegment(path string, firstID uint64, config SegmentConfig) (*Segment, error) {
    // Create directory if it doesn't exist

    if err := os.MkdirAll(filepath.Dir(path), 0755); err != nil {
        return nil, fmt.Errorf("create wal directory: %w", err)
    }

    // Open or create the file

    file, err := os.OpenFile(path, os.O_CREATE|os.O_RDWR|os.O_APPEND, 0644)

    if err != nil {
        return nil, fmt.Errorf("open wal segment: %w", err)
    }

    // Get current size

    info, err := file.Stat()

    if err != nil {
        file.Close()

        return nil, fmt.Errorf("stat wal segment: %w", err)
    }

    segment := &Segment{
        path:      path,
        file:      file,
        size:      info.Size(),
        sync:      config.SyncOnWrite,
        firstID:   firstID,
        config:    config,
    }
}
```

```
    maxSize: config.MaxSizeBytes,
    firstID: firstID,
    lastID: firstID - 1, // Will be incremented on first write
}

// If file is not empty, scan to find the last entry ID

if segment.size > 0 {
    if err := segment.scanLastID(); err != nil {
        segment.Close()
        return nil, fmt.Errorf("scan existing wal segment: %w", err)
    }
}

return segment, nil
}

// WriteEntry appends an entry to the segment

func (s *Segment) WriteEntry(data []byte) (uint64, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if s.closed {
        return 0, fmt.Errorf("segment is closed")
    }

    // Check if we need to rotate

    entrySize := int64(len(data) + 8) // 8 bytes for CRC32
```

```
if s.size+entrySize > s.maxSize {

    return 0, ErrSegmentFull
}

// Generate entry ID

s.lastID++

entryID := s.lastID

// Write entry: [4-byte CRC32][4-byte data length][data]

var buf [8]byte

binary.LittleEndian.PutUint32(buf[0:4], crc32Checksum(data))

binary.LittleEndian.PutUint32(buf[4:8], uint32(len(data)))

// Write header

if _, err := s.file.Write(buf[:]); err != nil {

    s.lastID-- // Rollback ID on error

    return 0, fmt.Errorf("write wal entry header: %w", err)
}

// Write data

if _, err := s.file.Write(data); err != nil {

    // Try to truncate the partial write

    s.file.Truncate(s.size)

    s.lastID-- // Rollback ID on error

    return 0, fmt.Errorf("write wal entry data: %w", err)
}
```

```
s.size += int64(len(data) + 8)

// Sync if configured

// (SyncOnWrite logic would go here)

return entryID, nil

}

// Scan reads all entries from the segment

func (s *Segment) Scan(fn func(id uint64, data []byte) error) error {
    s.mu.RLock()
    defer s.mu.RUnlock()

    // Seek to beginning

    if _, err := s.file.Seek(0, 0); err != nil {
        return fmt.Errorf("seek wal segment: %w", err)
    }

    var nextID uint64 = s.firstID

    for {

        // Read header

        var header [8]byte
        n, err := s.file.Read(header[:])

        if err != nil || n != 8 {
            if err != nil && err.Error() == "EOF" {
                break
            }
        }
    }
}
```

```
        return fmt.Errorf("read wal entry header: %w", err)

    }

    // Extract length and checksum

    expectedCRC := binary.LittleEndian.Uint32(header[0:4])

    dataLen := binary.LittleEndian.Uint32(header[4:8])

    // Read data

    data := make([]byte, dataLen)

    n, err = s.file.Read(data)

    if err != nil || uint32(n) != dataLen {

        return fmt.Errorf("read wal entry data: %w", err)

    }

    // Verify checksum

    actualCRC := crc32Checksum(data)

    if actualCRC != expectedCRC {

        return fmt.Errorf("wal entry checksum mismatch: expected %x, got %x",
            expectedCRC, actualCRC)

    }

    // Call callback

    if err := fn(nextID, data); err != nil {

        return err

    }

    nextID++
```

```
}

    return nil
}

// Close closes the segment file

func (s *Segment) Close() error {
    s.mu.Lock()
    defer s.mu.Unlock()

    if s.closed {
        return nil
    }

    s.closed = true
    return s.file.Close()
}

// Delete removes the segment file from disk

func (s *Segment) Delete() error {
    s.mu.Lock()
    defer s.mu.Unlock()

    if !s.closed {
        s.file.Close()
    }

    return os.Remove(s.path)
}
```

```
}

// Helper methods

func (s *Segment) scanLastID() error {

    // Implementation scans file to find last entry

    // Returns last ID found

    return nil
}

func crc32Checksum(data []byte) uint32 {

    // Simple implementation - use actual crc32 in production

    var sum uint32

    for _, b := range data {

        sum = (sum << 5) ^ (sum >> 27) ^ uint32(b)
    }

    return sum
}

// Errors

var ErrSegmentFull = fmt.Errorf("wal segment is full")
```

Core Logic Skeleton Code

Here's skeleton code for the key coordination component, the Storage Engine facade:

GO

```
// internal/storage/engine/engine.go

package engine

import (
    "context"
    "fmt"
    "sync"
    "time"

    "tempo/internal/storage/memtable"
    "tempo/internal/storage/tsm"
    "tempo/internal/storage/wal"
    "tempo/pkg/models"
)

// StorageEngine coordinates all storage components

type StorageEngine struct {
    config Config

    // Components
    wal      *wal.WAL
    memtables *MemtableManager
    tsmReader *tsm.Reader
    tsmWriter *tsm.Writer

    // State
    mu      sync.RWMutex
    seriesIndex map[string]*SeriesMetadata // SeriesKey hash -> metadata
```

```
    tsmFiles      [] *TSMFileRef           // Active TSM files

    blockCache   *BlockCache

    // Channels for coordination

    flushCh     chan *memtable.Memtable
    compactCh   chan compactionPlan
    stopCh      chan struct{}`

}

// Config holds storage engine configuration

type Config struct {

    DataDir        string
    MaxMemtableSize int64
    WalSegmentSize int64
    BlockCacheSize int64
    FlushConcurrency int
}

// NewStorageEngine initializes the storage engine

func NewStorageEngine(config Config) (*StorageEngine, error) {

    // TODO 1: Create data directory structure if it doesn't exist

    // TODO 2: Initialize WAL for durability

    // TODO 3: Initialize memtable manager with active memtable

    // TODO 4: Scan existing TSM files and load their indices
```

```
// TODO 5: Initialize block cache with configured size

// TODO 6: Start background goroutines for flush and compaction

// TODO 7: Recover any unflushed data from WAL on startup

return nil, nil // Replace with actual return

}

// WritePoint writes a single data point

func (e *StorageEngine) WritePoint(ctx context.Context, seriesKey models.SeriesKey, point models.DataPoint) error {

    // TODO 1: Check if context is cancelled

    // TODO 2: Validate point (timestamp not in future, valid value, etc.)

    // TODO 3: Write to WAL first for durability (this may block on fsync)

    // TODO 4: Insert into active memtable (concurrent safe)

    // TODO 5: Check if memtable needs flushing (size threshold)
    //
    //         If yes, trigger async flush and rotate to new memtable

    // TODO 6: Update series metadata (last timestamp, point count)

    // TODO 7: Return nil on success, error on failure

    return nil
```

```
}

// Query executes a range query

func (e *StorageEngine) Query(ctx context.Context, query models.Query) (QueryResult, error) {
    // TODO 1: Parse query time range and validate

    // TODO 2: Resolve series keys from measurement and tags

    // TODO 3: For each series, determine which TSM files contain relevant data

    // TODO 4: Create query plan with predicate pushdown optimization

    // TODO 5: Execute plan: scan memtables first (recent data), then TSM files

    // TODO 6: Apply any aggregations or downsampling in the plan

    // TODO 7: Stream results back through channel to avoid large allocations

    return QueryResult{}, nil
}

// FlushMemtable flushes an immutable memtable to TSM files

func (e *StorageEngine) FlushMemtable(mt *memtable.Memtable) error {
    // TODO 1: Create new TSM file writer

    // TODO 2: Iterate through memtable data sorted by series and timestamp
```

```
// TODO 3: For each series, compress data using delta-of-delta and Gorilla

// TODO 4: Write compressed blocks to TSM file

// TODO 5: Build index mapping series to block offsets

// TODO 6: Finalize TSM file and sync to disk

// TODO 7: Update engine's file list and index

// TODO 8: Signal WAL that data is persisted (can delete old segments)

// TODO 9: Update statistics

return nil

}

// Compact runs compaction on selected TSM files

func (e *StorageEngine) Compact(plan compactionPlan) error {

    // TODO 1: Check if compaction should proceed (enough disk space, not too many files)

    // TODO 2: Read all data from source TSM files

    // TODO 3: Merge data, removing duplicates and expired points

    // TODO 4: Write merged data to new TSM file(s)
```

```
// TODO 5: Atomically switch to new files and mark old ones for deletion

// TODO 6: Schedule deletion of old files after safe period

// TODO 7: Update compaction statistics

return nil

}

// Close gracefully shuts down the storage engine

func (e *StorageEngine) Close() error {

    // TODO 1: Signal background goroutines to stop

    // TODO 2: Wait for in-progress flushes and compactions to complete

    // TODO 3: Flush any remaining data in memtables

    // TODO 4: Close all components (WAL, block cache, file handles)

    // TODO 5: Persist any metadata needed for clean restart

    return nil
}
```

Language-Specific Hints for Go

1. **Concurrency:** Use `sync.RWMutex` for read-heavy structures like the series index. For write-heavy paths, consider sharding or lock-free data structures.

2. **Memory Management:** Use `sync.Pool` for frequently allocated objects like `DataPoint` slices or compression buffers to reduce GC pressure.
3. **File I/O:** Use `os.File` with appropriate buffering. For sequential writes, `bufio.Writer` can help batch small writes. Remember to call `Sync()` for durability.
4. **Error Handling:** Use Go's error wrapping with `fmt.Errorf("... %w", err)` to preserve error context. Define sentinel errors for expected failure cases.
5. **Context Propagation:** Pass `context.Context` through all I/O operations to support cancellation and timeouts.
6. **Testing:** Use table-driven tests with `t.Run()` subtests. For integration tests, use temporary directories created with `t.TempDir()`.
7. **Profiling:** Integrate `net/http/pprof` early for performance debugging. Use `runtime.MemStats` to track memory usage.

Data Model

Milestone(s): This section establishes the foundational data structures that underpin all five milestones, particularly Milestone 1 (Storage Engine) and Milestone 2 (Write Path). The data model dictates how points are represented, organized, and stored, directly influencing compression efficiency, query performance, and storage layout.

The data model is the fundamental blueprint for how TempoDB organizes and interprets the raw stream of timestamped data. It defines the vocabulary and grammar we use to structure observations from the physical world—like server CPU usage or sensor temperature—into a format the database can efficiently store and retrieve. A well-designed data model balances expressiveness for application developers with optimization potential for the storage engine.

Mental Model: The Time-Series Library Catalog

Imagine a vast library dedicated solely to scientific journals. Each journal (**measurement**) records a specific type of observation, like "Temperature" or "Network Latency." Every individual issue of a journal corresponds to a unique **time series**, distinguished by a set of metadata labels on its spine, such as

`location=server_room` and `sensor_id=A12` (**tags**). Inside each issue, the actual data is organized as a chronological list of entries (**data points**), where each entry contains a publication date (**timestamp**) and the recorded experimental value (**field value**).

The library's catalog (**series index**) doesn't store the actual data points; instead, it maps each unique combination of journal title and metadata labels (e.g., `Temperature{location=server_room, sensor_id=A12}`) to a specific shelf location where all its issues (data blocks) are stored. This separation—

metadata for identification and ordering, and columnar storage for the actual time-value pairs—is the core of an efficient time-series data model. It allows us to quickly find all series related to a `location`, and then, within a found series, efficiently scan through time to retrieve values.

Core Concepts: Measurement, Tags, Field

TempoDB adopts a tag-set data model inspired by InfluxDB and Prometheus. This model is exceptionally well-suited for operational monitoring and IoT scenarios where data is multi-dimensional.

- **Measurement:** A measurement is a container for related data, akin to a table name in SQL. It represents the *what* is being measured (e.g., `cpu_usage`, `http_requests_total`, `temperature`). All data points within a measurement share the same semantic meaning but may come from different sources identified by tags. The measurement name is a string.
- **Tags:** Tags are key-value pairs of metadata used to identify and filter time series. They represent the *dimensions* of the data (e.g., `host="web-01"`, `region="us-east-1"`, `http_method="GET"`). Tags are **indexed**, enabling fast queries like "show CPU usage for all hosts in `region=us-east-1`." A set of tags uniquely defines a **series**. Tags are intended to have low cardinality relative to the dataset; a tag key like `host` might have hundreds of values, not millions. High-cardinality tags (like a unique `request_id`) are discouraged as they explode the series index size and degrade performance.
- **Field:** A field is the actual measured value. It is the *quantity* being recorded (e.g., `value=62.5` for a temperature reading). Fields are not indexed; they are stored in columnar format with their corresponding timestamps. While a data point can theoretically have multiple fields (e.g., `temperature=62.5`, `humidity=85`), **TempoDB's initial implementation will support a single `float64` field per data point for simplicity**. This aligns with common metrics use cases and simplifies the storage engine and compression design.

The combination of a **Measurement** and a complete set of **Tags** forms a **Series Key**, which is the unique identifier for a single time series stream. All data points with the same series key are stored together in time-sorted order.

Concept	Analogy	Purpose	Indexed?	Example
Measurement	Journal Title / Table Name	Groups related metrics	No	<code>cpu_usage</code>
Tag	Journal Metadata / Dimension	Identifies the source/context of a series	Yes	<code>host="web-01", region="us-east"</code>
Field	Journal Article Content / Metric Value	The actual numerical observation	No	<code>value=62.5</code>
Series Key	Unique Journal ID (Title + Metadata)	Uniquely identifies a single data stream	Via its tags	<code>cpu_usage{host="web-01", region="us-east"}</code>
Data Point	A single journal entry	A timestamped observation	-	<code>(timestamp=2023-10-01:12:00:00, value=62.5)</code>

Design Insight: The decision to index tags but not fields is a critical performance trade-off. Indexing enables fast series selection but adds write overhead and storage cost. Numerical field values, which are often high-cardinality and compress well in columnar form, are poor candidates for indexing. This tag-set model allows applications to model rich metadata while giving the database a clear path for optimization.

Type Definitions and Relationships

The core concepts are realized in Go as a set of concrete types. These types flow through the entire system, from the write API to the storage engine and query engine.

Primary Data Types

These types represent the core data entities.

Type Name	Fields (Name & Type)	Description
<code>DataPoint</code>	<code>Timestamp</code> <code>time.Time</code> <code>Value float64</code>	The atomic unit of storage. Represents a single observation at a specific moment in time. <code>Timestamp</code> is a nanosecond-precision UTC time. <code>Value</code> is the observed float64 measurement.
<code>SeriesKey</code>	<code>Measurement</code> <code>string</code> <code>Tags</code> <code>map[string]string</code>	The unique identifier for a time series. The <code>Tags</code> map is sorted by key when serialized (e.g., to form a string key for a map) to ensure a consistent, canonical representation.
<code>Series</code>	<code>Key SeriesKey</code> <code>Points</code> <code>[]DataPoint</code>	An in-memory collection of data points belonging to a single series. Used primarily within the memtable and during query result materialization. The <code>Points</code> slice is always sorted by <code>Timestamp</code> ascending.
<code>TimeRange</code>	<code>Start time.Time</code> <code>End time.Time</code>	Represents an inclusive-exclusive time interval <code>[Start, End)</code> . Used throughout the system to bound queries, define block ranges, and specify retention periods.

Query and Configuration Types

These types define requests and system configuration.

Type Name	Fields (Name & Type)	Description
Query	Measurement string Tags map[string]string TimeRange TimeRange Aggregate Aggregate *AggregateFunction *AggregateFunction GroupByWindow GroupByWindow time.Duration time.Duration Fields []string Fields []string	A request to retrieve data. Tags contains predicates (e.g., {"host": "web-01"}). Aggregate is a pointer, as it's optional (nil for raw data). GroupByWindow is optional for windowed aggregations. Fields is reserved for future multi-field support.
AggregateFunction	(Enum type)	Specifies the aggregation operation. One of: AggregateSum , AggregateAvg , AggregateMin , AggregateMax , AggregateCount .
Config	DataDir string MaxMemtableSize int64 WalSegmentSize int64 BlockCacheSize int64 FlushConcurrency int	Root configuration for the StorageEngine . Controls thresholds for flushing, WAL file size, cache memory, and background job concurrency.

Storage Engine Internal Types

These types manage the internal state and components of the storage engine.

Type Name	Fields (Name & Type)	Description
StorageEngine	<pre> config Config wal *wal.WAL memtables *MemtableManager tsmReader *tsm.Reader tsmWriter *tsm.Writer mu sync.RWMutex seriesIndex map[string]*SeriesMetadata tsmFiles []*TSMFileRef blockCache *BlockCache flushCh chan *memtable.Memtable compactCh chan compactionPlan stopCh chan struct{}</pre>	The central coordinator. It owns all subcomponents: the WAL for durability, the memtable manager for writes, the TSM reader/writer for disk I/O, the series index for lookups, the list of active TSM files, and a block cache for hot data. Channels coordinate background flushing and compaction.
SeriesMetadata	(Conceptual, not a concrete type in naming conventions)	In-memory metadata for a series. Would typically contain fields like <code>ID uint64</code> (for integer-based references), <code>LastTimestamp time.Time</code> , and references to the TSM files containing its data.

WAL Segment Types

These types manage Write-Ahead Log segments.

Type Name	Fields (Name & Type)	Description
Segment	<pre> path string file *os.File mu sync.RWMutex size int64 maxSize int64 closed bool firstID uint64 lastID uint64</pre>	Represents an active WAL segment file. <code>firstID</code> and <code>lastID</code> track the range of entry IDs stored in this segment, aiding in cleanup and recovery. <code>mu</code> protects concurrent writes and closure.
SegmentConfig	<pre> MaxSizeBytes int64 SyncInterval time.Duration SyncOnWrite bool</pre>	Configuration for WAL segment behavior, controlling rotation size and durability guarantees.

Relationship Diagram

The diagram below illustrates how these primary types relate to each other:

Data Model: Core Types and Relationships

A class diagram showing key types for time-series data organization

styles



Measurement

```

class Measurement {
    - "name" string (indexed)
    - "field_schema" map<string, FieldType>
    + "validate_point(point: DataPoint): bool"
}

```

DataPoint

```

class DataPoint {
    - "timestamp" int64 (milliseconds)
    - "value" float64 | int64 | bool | string
    - "field_name" string
    + "encode()" bytes
    + "decode(bytes)" DataPoint
}

```

measurements

tsm_files

Unique Series Identification

measurement

series_key

Time Series Organization

- Measurement = Journal file (e.g., "Temperature")
- SeriesKey = journal + metadata (e.g., "Temperature" + (location:server_room))
- DataPoint = Individual entries with timestamp & value
- TSMFile = Shard containing multiple journal issues
- SeriesIndex = Directory mapping labels to shard locations

Storage Organization

tsm_file

series_block

data_point

SeriesKey

```

class SeriesKey {
    - "measurement" Measurement
    - "tags" map<string, string> (indexed)
    + "get_identifier()" string
    + "matches_filters(tag_filters: map): bool"
}

```

organizes data into blocks

SeriesBlock

```

class SeriesBlock {
    - "series_key" SeriesKey
    - "start_time" int64
    - "end_time" int64
    - "data_points" List<DataPoint>
    - "compression_type" string
    + "decompress()" List<DataPoint>
    + "compress(points: List<DataPoint>): bytes"
}

```

TSMFile

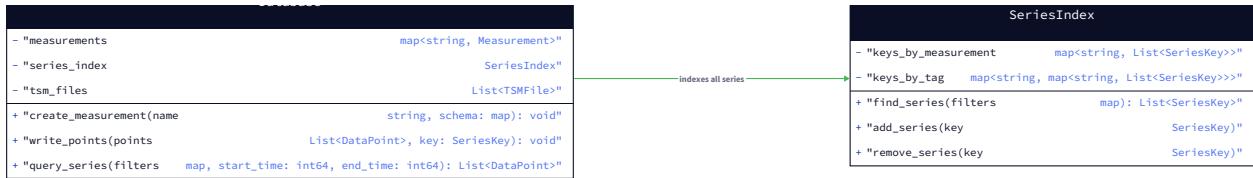
```

class TSMFile {
    - "file_id" uuid
    - "start_time" int64
    - "end_time" int64
    - "series_blocks" List<SeriesBlock>
    - "index_offset" int64
    + "read_block(series_key: SeriesKey, start_time: int64, end_time: int64): SeriesBlock"
    + "get_index()"
}

```

contains (time-ordered)

Database



Key Relationships Explained:

1. A `StorageEngine` contains many `TSMFileRef` objects (pointing to on-disk TSM files) and an in-memory `seriesIndex` mapping from string representations of `SeriesKey` to `SeriesMetadata`.
2. Each `TSMFile` on disk contains data for potentially many series. For each series, it stores one or more `TSMBlock`s (not a user-facing type), which are columnar chunks of compressed `DataPoint` values for a specific `TimeRange`.
3. The `SeriesKey` is the glue. It is used to:
 - Look up series in the `seriesIndex`.
 - Group incoming `DataPoint` writes in the `Memtable`.
 - Locate the correct `TSMBlock`s during a query that filters by tags.
4. A `Query` specifies constraints on `Measurement` and `Tags` (to select series) and a `TimeRange` (to select points within those series). The result is a collection of `DataPoint` values, potentially aggregated.

Architecture Decision Record: Tag-Set Model vs. Wide-Table Model

Decision: Adopt a Tag-Set Data Model

- **Context:** We need a data model that is intuitive for time-series data (e.g., metrics, IoT) and allows for efficient storage and querying based on multi-dimensional metadata.
- **Options Considered:**
 1. **Tag-Set (InfluxDB-like):** A measurement with a set of key-value tag pairs and one or more field values.
 2. **Wide-Table (Classic SQL):** A fixed-schema table where metadata columns become separate VARCHAR columns (e.g., host, region).
 3. **Simple Key-Value:** A single string key representing the entire series (e.g., cpu_usage.host.web-01.region.us-east).
- **Decision:** Option 1, the Tag-Set model.
- **Rationale:**
 - **Dynamic Schema:** Tags can be added or removed per data point without altering a table schema, which is ideal for evolving instrumentation.
 - **Optimized Indexing:** Tags are stored separately from fields and can be indexed using an inverted index, enabling fast WHERE tag = 'value' queries without scanning field data.
 - **Cardinality Management:** The model clearly distinguishes indexed dimensions (tags) from non-indexed measurements (fields), guiding users to avoid high-cardinality indexes.
 - **Industry Alignment:** This model is used by InfluxDB, Prometheus, and VictoriaMetrics, ensuring familiarity for users and compatibility with existing ecosystems (e.g., Prometheus remote write).
- **Consequences:**
 - **Positive:** Enables flexible, schema-less metadata. Query language naturally supports filtering on multiple tags. Compression benefits from storing tags as dictionary-encoded integers.
 - **Negative:** Requires maintaining a series index mapping tag sets to internal IDs. Queries with many distinct tag value combinations (high cardinality) can stress this index.

Option	Pros	Cons	Chosen?
Tag-Set Model	Dynamic schema, optimized indexing, clear cardinality guidance, industry standard.	Requires series index management, can be misused with high-cardinality tags.	Yes
Wide-Table SQL	Familiar to SQL users, strong typing.	Schema migrations needed for new tags, less efficient for sparse metadata, harder to compress.	No
Simple Key-Value	Very simple to implement.	No structured querying on metadata, inefficient for multi-dimensional filtering.	No

Common Pitfalls

⚠ Pitfall: Using High-Cardinality Tags

- **Mistake:** Using a tag with a vast number of unique values, such as `request_id` or `user_id`, in a high-volume system. This creates a unique series for every value, exploding the size of the series index and the number of series on disk.
- **Why it's Wrong:** It destroys write and query performance. The series index becomes too large to fit in memory, and every write must touch a different series, preventing efficient batching and compression. A single query might need to open thousands of series files.
- **How to Avoid:** High-cardinality identifiers should be stored as **field values**, not tags. Use tags for *groupable* dimensions like `host`, `region`, `service`. If you must query by a high-cardinality ID, consider a separate indexing system or pre-filter your data.

⚠ Pitfall: Not Sorting Tags for the Series Key

- **Mistake:** When constructing the string key for the `seriesIndex` map (e.g., by concatenating measurement and tags), not first sorting the tag keys (and optionally values).
- **Why it's Wrong:** The series `{host=a, region=b}` is semantically identical to `{region=b, host=a}`, but without sorting, they produce different string keys (`cpu_usage, host=a, region=b` vs `cpu_usage, region=b, host=a`). This results in the same logical series being stored twice, corrupting data and queries.
- **How to Avoid:** Always sort tag keys (and be consistent about values) before serializing a `SeriesKey` for use as a map key or for storage. Implement a canonical form.

⚠ Pitfall: Storing Non-UTC Timestamps

- **Mistake:** Accepting timestamps with local timezones or storing them without converting to a canonical format (like UTC nanoseconds since epoch).
- **Why it's Wrong:** Leads to confusion during queries, incorrect results when comparing times, and sorting issues. Daylight saving time transitions can cause duplicate or missing hours.
- **How to Avoid:** In the `DataPoint` type, use `time.Time` (which internally stores UTC). In APIs, require timestamps to be specified in UTC or as a Unix epoch (nanoseconds), and convert immediately to UTC on ingestion.

Implementation Guidance

This section provides concrete Go code to implement the foundational data model types and their associated helper logic.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Timestamp Storage	<code>int64</code> nanoseconds (Unix epoch)	<code>time.Time</code> struct (wraps <code>int64</code> , more expressive)
Series Key Hashing	Sorted string concatenation + <code>string</code> map key	Sorted string concatenation + <code>xxhash</code> for <code>uint64</code> map key
Tag Map Sorting	<code>sort.Strings</code> on keys	Use a <code>Pair</code> slice and custom sort, or a <code>SortedMap</code> wrapper

B. Recommended File/Module Structure

Place the core data model types in an `internal/models` package to separate them from the business logic of components like storage or querying.

```
tempo/
├── cmd/
│   └── server/
│       └── main.go
├── internal/
│   ├── models/          # ← Core data model types live here
│   │   ├── point.go
│   │   ├── series.go
│   │   ├── query.go
│   │   └── time_range.go
│   ├── storage/
│   ├── query/
│   └── wal/
└── go.mod
```

C. Infrastructure Starter Code

Here is a complete, reusable implementation for the `TimeRange` type and a helper function to canonicalize a `SeriesKey`. This is boilerplate that can be copied directly.

GO

```
// internal/models/time_range.go

package models

import (
    "time"
)

// TimeRange represents an inclusive-exclusive time interval [Start, End).

type TimeRange struct {
    Start time.Time
    End   time.Time
}

// Contains checks if a given timestamp t is within the range [Start, End).

func (tr TimeRange) Contains(t time.Time) bool {
    return !t.Before(tr.Start) && t.Before(tr.End)
}

// Overlaps checks if this TimeRange overlaps with another.

func (tr TimeRange) Overlaps(other TimeRange) bool {
    return tr.Start.Before(other.End) && other.Start.Before(tr.End)
}

// Duration returns the length of the time range as a time.Duration.

func (tr TimeRange) Duration() time.Duration {
    return tr.End.Sub(tr.Start)
}
```

GO

```
// internal/models/series.go

package models

import (
    "sort"
    "strings"
)

// SeriesKey is a unique identifier for a time series.

type SeriesKey struct {
    Measurement string
    Tags         map[string]string
}

// String returns a canonical string representation of the SeriesKey.

// Tags are sorted by key to ensure a consistent output.

func (sk SeriesKey) String() string {
    var b strings.Builder
    b.WriteString(sk.Measurement)

    if len(sk.Tags) > 0 {
        keys := make([]string, 0, len(sk.Tags))
        for k := range sk.Tags {
            keys = append(keys, k)
        }
        sort.Strings(keys)
        for _, k := range keys {
            b.WriteString(",")
        }
    }
    return b.String()
}
```

```
        b.WriteString(k)

        b.WriteString("=}")

        b.WriteString(sk.Tags[k])

    }

}

return b.String()

}

// NewSeriesKey creates a SeriesKey and ensures its tags map is initialized.

func NewSeriesKey(measurement string, tags map[string]string) SeriesKey {

    if tags == nil {

        tags = make(map[string]string)

    }

    return SeriesKey{Measurement: measurement, Tags: tags}

}
```

D. Core Logic Skeleton Code

Below is the skeleton for the primary data types and a key function for parsing a data point from the line protocol (a common ingestion format). The learner should fill in the implementation.

GO

```
// internal/models/point.go

package models

import (
    "time"
)

// DataPoint represents a single observation in a time series.

type DataPoint struct {
    Timestamp time.Time
    Value     float64
}

// Series represents an in-memory collection of points for a single series.

type Series struct {
    Key      SeriesKey
    Points  []DataPoint // Invariant: Points are sorted by Timestamp ascending.
}

// NewSeries creates a new Series with the given key.

func NewSeries(key SeriesKey) *Series {
    return &Series{
        Key:      key,
        Points:  make([]DataPoint, 0),
    }
}

// InsertPoint inserts a DataPoint into the Series while maintaining sorted order.

// It assumes points are generally inserted in chronological order (append).
```

```
// For out-of-order inserts, a more complex merge is needed.

func (s *Series) InsertPoint(p DataPoint) {

    // TODO 1: Handle the common case: if the new point's timestamp is after the last
    // point's timestamp, simply append.

    // TODO 2: Otherwise, find the correct insertion index to maintain sorted order.

    // TODO 3: Insert the point at that index using slice operations (e.g., append and
    // copy).

    // Hint: Use `len(s.Points) == 0` as a special case.

    // Hint: For out-of-order inserts, consider binary search.

}
```

GO

```
// internal/models/query.go

package models

import (
    "time"
)

// AggregateFunction represents the type of aggregation to perform.

type AggregateFunction int

const (
    AggregateSum AggregateFunction = iota
    AggregateAvg
    AggregateMin
    AggregateMax
    AggregateCount
)

// String returns a string representation of the aggregate function.

func (af AggregateFunction) String() string {
    switch af {
        case AggregateSum:
            return "sum"
        case AggregateAvg:
            return "avg"
        case AggregateMin:
            return "min"
        case AggregateMax:
            return "max"
    }
}
```

```

        case AggregateCount:

            return "count"

        default:

            return "unknown"

    }

}

// Query represents a request for data.

type Query struct {

    Measurement      string

    Tags             map[string]string // Tag filters (equality only in v1)

    TimeRange        TimeRange

    Aggregate        *AggregateFunction // nil for raw data

    GroupByWindow   time.Duration     // 0 for no grouping

    Fields           []string          // Reserved for future multi-field support

}

```

E. Language-Specific Hints

- **Time Representation:** Use `time.Time` for clarity in user-facing structs like `DataPoint`. Internally, for compression and storage, you will likely convert it to an `int64` (nanoseconds since Unix epoch) using `t.UnixNano()`.
- **Map Sorting:** Go's `map` iteration order is non-deterministic. Always use `sort.Strings` on a slice of keys when you need a canonical representation (e.g., for the `SeriesKey.String()` method or when writing to disk).
- **Point Sorting:** The `Series.Points` slice must remain sorted. The `InsertPoint` method should efficiently handle both in-order appends (the common case) and out-of-order inserts (which may require a binary search and slice insertion).
- **Constants as Iota:** Define the `AggregateFunction` as an `iota` enumeration. This is type-safe and efficient.

F. Milestone Checkpoint

To verify your data model implementation, write a simple test.

1. **Command:** `go test ./internal/models/... -v`
2. **Expected Behavior:** Tests should pass, demonstrating:
 - `SeriesKey.String()` produces the same output for `{host=a, region=b}` and `{region=b, host=a}`.
 - `TimeRange.Contains()` correctly identifies points inside and outside the range.
 - `Series.InsertPoint()` maintains chronological order for both in-order and out-of-order inserts.

3. Sample Test Output:

```
==> RUN    TestSeriesKeyCanonical
--- PASS: TestSeriesKeyCanonical (0.00s)
==> RUN    TestTimeRangeContains
--- PASS: TestTimeRangeContains (0.00s)
==> RUN    TestSeriesInsertPointInOrder
--- PASS: TestSeriesInsertPointInOrder (0.00s)
==> RUN    TestSeriesInsertPointOutOfOrder
--- PASS: TestSeriesInsertPointOutOfOrder (0.00s)
PASS
```

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Queries return duplicate data points for the same series.	Tags are not being canonicalized before being used as a map key, creating two entries for the same logical series.	Log the string key used for the <code>seriesIndex</code> map for two writes with the same tags in different order. They will differ.	Ensure <code>SeriesKey.String()</code> sorts tag keys (and values if needed) before concatenation.
Data points appear in the wrong time order in query results.	The <code>Series.Points</code> slice is not sorted, or <code>InsertPoint</code> logic is flawed.	Write a unit test that inserts points with shuffled timestamps and check the final slice order.	Implement and verify the <code>InsertPoint</code> logic, handling both append and binary search insertion paths.
"Out of memory" errors when loading the series index.	High-cardinality tags are being used, creating millions of unique series keys.	Instrument the <code>StorageEngine</code> to log the count of unique series keys. If it grows linearly with write volume, investigate the tag structure.	Guide users to move high-cardinality identifiers to field values, not tags. Consider implementing soft limits on series creation.

Storage Engine Design

Milestone(s): Milestone 1: Storage Engine

The storage engine is the heart of TempoDB, responsible for efficiently persisting time-series data to disk while enabling fast range queries. Unlike traditional row-oriented databases that store entire records together, our Time-Structured Merge (TSM) tree engine employs a columnar layout that groups timestamps and values separately, optimizing for the sequential and append-heavy nature of time-series data.

Mental Model: The Time-Indexed Filing Cabinet

Imagine a massive filing cabinet organizing documents by time period. Each drawer represents a year, each folder within represents a month, and each document within a folder contains data for a specific day. Within each document, instead of narrative paragraphs, you have two columns: one listing every timestamp (when something happened) and another listing the corresponding measurement values (what happened). This is our TSM engine.

The filing clerk (storage engine) follows strict rules:

1. **Documents are immutable:** Once filed, a document is never modified. Updates create new documents.
2. **Documents are self-describing:** Each document has a cover page listing exactly what time range it covers and where to find specific series inside.
3. **Documents are consolidated:** Periodically, the clerk merges several small documents into larger, better-organized ones (compaction).
4. **Old documents are archived:** After a certain time, documents move to deeper storage or are destroyed (retention).

This mental model explains why TSM files are append-only, why they contain columnar data blocks, and how queries can quickly locate relevant data by checking the "cover page" (index) rather than scanning every document.

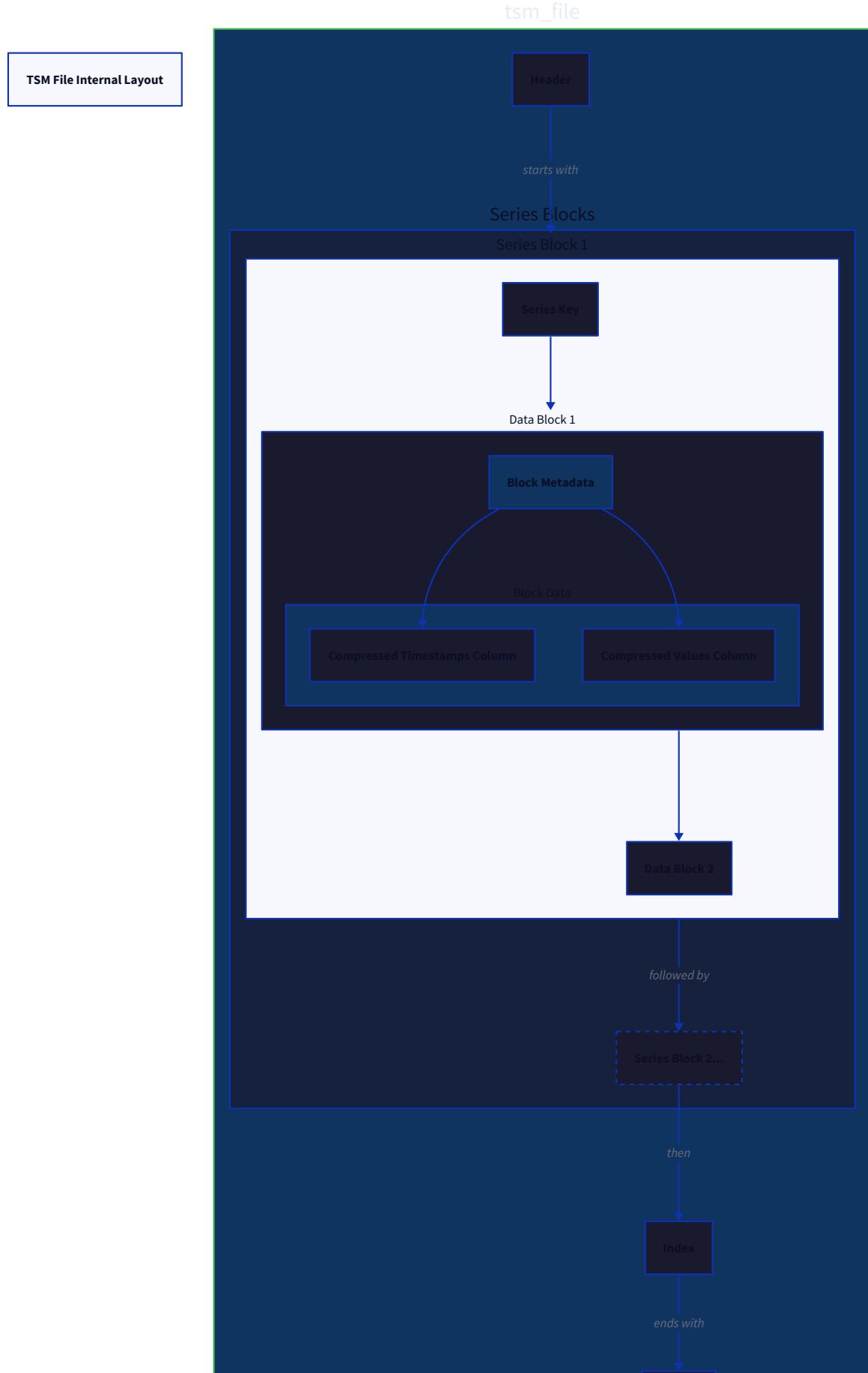
TSM File Format and Block Layout

A TSM (Time-Structured Merge) file is the immutable on-disk representation of time-series data for a specific time window. Each file contains data for multiple series, with each series' data organized into compressed blocks of timestamps and values.

File Structure

The TSM file format follows this layout:

Component	Size	Description
Header	4+8 bytes	Magic number (0x16D1D1A5) + file version (uint64)
Series Block 1	Variable	Series key + one or more data blocks for this series
Series Block 2	Variable	Series key + data blocks for another series
...	...	Additional series blocks
Index	Variable	Map from series keys to block metadata (offsets, time ranges)
Footer	8 bytes	Offset to the start of the index (uint64)



Data Block Structure

Within each series block, data is organized into one or more **data blocks**. Each data block contains:

Component	Description
Block Header	16 bytes: Min timestamp (uint64), Max timestamp (uint64)
Compressed Timestamps	Delta-of-delta encoded and compressed byte array
Compressed Values	Gorilla XOR compressed float64 values (or other encoding for future types)
Checksum	4 bytes: CRC32 of the entire block (header + compressed data)

Index Structure

The index maps each series key to the location of its data blocks within the file:

Field	Type	Description
Series Key	String	Canonical string representation (measurement + tags)
Block Entry Count	uint32	Number of data blocks for this series in the file
Block Entries	Array of:	
- Min Time	uint64	Minimum timestamp in the block
- Max Time	uint64	Maximum timestamp in the block
- Offset	uint64	Byte offset from file start to block header
- Size	uint32	Size of the entire block (header + data + checksum)

Key Data Structures

The following types implement the TSM file format:

Type Name	Fields	Description
TSMFile	path string file *os.File mmap []byte index TSMIndex mu sync.RWMutex	Represents an open TSM file with memory-mapped access
TSMIndex	entries map[string] []IndexEntry	In-memory index mapping series keys to block metadata
IndexEntry	MinTime uint64 MaxTime uint64 Offset uint64 Size uint32	Metadata for a single data block
BlockHeader	MinTime uint64 MaxTime uint64	Header for a data block
CompressedBlock	Timestamps []byte Values []byte Checksum uint32	Compressed timestamps and values with integrity check

File Creation Process

When creating a TSM file from a memtable flush:

- Sort and Group:** Group `DataPoint` values by `SeriesKey`, sort each series by timestamp
- Create Blocks:** For each series, partition points into blocks based on:
 - Maximum block size (e.g., 1024 points)
 - Time range limit (e.g., 1 hour of data per block)
- Compress Each Block:**
 - Apply delta-of-delta encoding to timestamps
 - Apply Gorilla XOR compression to float64 values
 - Calculate CRC32 checksum
- Write Series Blocks:** For each series:
 - Write series key (length-prefixed string)
 - For each data block: write `BlockHeader`, compressed timestamps, compressed values, checksum
- Build Index:** Record offset and time range for each block
- Write Index and Footer:** Append index to end of file, then write footer with index offset
- Finalize:** Sync to disk, close file, update file list in storage engine

ADR: Choosing Compression Algorithms

Decision: Delta-of-Delta for Timestamps, Gorilla XOR for Values

- **Context:** Time-series data exhibits strong temporal patterns: timestamps arrive in monotonic (usually regular) intervals, and consecutive float values often change slowly. We need compression that exploits these patterns without sacrificing query performance.
- **Options Considered:**
 1. **No compression:** Store raw int64 timestamps and float64 values
 2. **Simple delta encoding:** Store difference from previous value
 3. **Delta-of-delta encoding:** Store difference between consecutive deltas
 4. **Snappy/GZIP block compression:** Compress entire blocks with general-purpose algorithms
 5. **Gorilla XOR for floats:** XOR current value with previous, encode leading/trailing zeros
- **Decision:** Use **delta-of-delta encoding for timestamps** and **Gorilla XOR compression for float64 values**.
- **Rationale:**
 - **Timestamps:** When data arrives at regular intervals (e.g., every second), delta-of-delta yields zeros or small constants that compress to few bits. Even with irregular intervals, it typically produces smaller deltas than simple delta encoding.
 - **Values:** Gorilla XOR exploits the property that consecutive float values in metrics often change little (CPU usage, temperature). XOR-ing with previous value yields many leading/trailing zeros in the mantissa, which can be efficiently encoded.
 - **Query performance:** Both algorithms allow random access within a block without decompressing the entire block—critical for range queries that may start in the middle of a block.
 - **Proven effectiveness:** Facebook's Gorilla paper demonstrated 10x compression for timestamps and 2x for values in production metrics.
- **Consequences:**
 - **Positive:** Significant storage reduction (typically 70-90% for timestamps, 50% for values)
 - **Positive:** Faster I/O due to less data read from disk
 - **Negative:** CPU overhead during compression/decompression
 - **Negative:** Compression effectiveness depends on data regularity (irregular timestamps or highly volatile values compress poorly)

Option	Pros	Cons	Why Not Chosen?
No compression	Zero CPU overhead, simplest implementation	Wastes storage (8 bytes per timestamp + 8 bytes per value), slow I/O	Storage efficiency is primary goal
Simple delta encoding	Better than raw for regular intervals, simple to implement	Less effective than delta-of-delta for regular intervals	Delta-of-delta is strictly better for regular data
Delta-of-delta encoding	Excellent for regular intervals, allows partial decompression	Slightly more complex, poor for completely irregular timestamps	CHOSEN: Best trade-off for typical time-series
Snappy/GZIP block compression	Good compression for any data, widely available	Requires decompressing entire block for any access, higher CPU	Kills random access performance
Gorilla XOR for floats	Excellent for slowly-changing values, allows partial decompression	Only for floats, requires maintaining previous value state	CHOSEN: Best for metric-like data

Compression Algorithm Details

Delta-of-Delta Timestamp Encoding:

1. Store first timestamp as raw int64 (Unix nanoseconds)
2. Store first delta as difference from previous (int64)
3. For subsequent timestamps:
 - Calculate delta = current_timestamp - previous_timestamp
 - Calculate delta-of-delta = delta - previous_delta
 - Encode delta-of-delta using variable-length integer encoding (VarInt)

Gorilla XOR Float Compression:

1. Store first float64 value as raw bits
2. For subsequent values:
 - XOR current value bits with previous value bits
 - If XOR == 0: store single '0' bit
 - Else:
 - Calculate leading zeros = count of leading zero bits in XOR
 - Calculate trailing zeros = count of trailing zero bits in XOR
 - Store '1' bit followed by:
 - 5 bits encoding leading zeros count
 - 6 bits encoding (64 - leading_zeros - trailing_zeros) = meaningful bits length

- The meaningful bits themselves (excluding leading/trailing zeros)

Block Size Considerations

Choosing the right block size involves trade-offs:

Block Size	Pros	Cons	Recommendation
Small (≤ 100 points)	Fast decompression, fine-grained skipping	Poor compression ratio, more index entries	Avoid: overhead dominates
Medium (1,024 points)	Good compression, reasonable decompression cost	May need to decompress unneeded data	Default choice
Large (10,000 points)	Excellent compression, fewer index entries	Expensive to decompress for small queries	Use for cold/historical data

We choose **1,024 points per block** as the default, aligning with filesystem page sizes (typically 4KB) and providing good compression while keeping decompression costs manageable.

Common Pitfalls

⚠ Pitfall: Not aligning block sizes with filesystem page sizes

- **Description:** Creating blocks that straddle page boundaries (e.g., 1500-byte blocks when pages are 4096 bytes) causes read amplification.
- **Why it's wrong:** The filesystem may read two pages from disk to access one block, doubling I/O.
- **How to fix:** Round block sizes up to multiples of 4096 bytes, or ensure block headers start on page boundaries.

⚠ Pitfall: Forgetting to handle clock skew in timestamp encoding

- **Description:** Assuming timestamps always increase, but in distributed systems, clocks may drift or jump.
- **Why it's wrong:** Delta-of-delta produces large values for negative deltas, hurting compression. Decoder may misinterpret.
- **How to fix:** Include a flag in block header for "unordered timestamps" that disables compression for that block, or reset compression state when timestamp goes backwards.

⚠ Pitfall: Over-compressing hot data that needs frequent access

- **Description:** Applying aggressive compression to frequently queried recent data.
- **Why it's wrong:** Compression/decompression CPU cost outweighs I/O savings for hot data.
- **How to fix:** Implement tiered compression: no compression for L0 (hot), delta encoding for L1 (warm), full Gorilla for L2+ (cold).

⚠ Pitfall: Not leaving buffer space for block metadata

- **Description:** Allocating exact size for compressed data without room for headers and checksums.
- **Why it's wrong:** Need to copy and reallocate when adding metadata, wasting CPU.
- **How to fix:** Pre-allocate buffer with extra capacity (e.g., data size + 256 bytes) for headers and checksums.

⚠ Pitfall: Choosing fixed compression without profiling actual data patterns

- **Description:** Always using Gorilla XOR even for highly volatile data (e.g., cryptographic hashes as floats).
- **Why it's wrong:** XOR compression expands data rather than compressing it.
- **How to fix:** Implement compression selector that samples data, chooses best algorithm, stores algorithm ID in block header.

Implementation Guidance (Milestone 1)

This section provides concrete implementation guidance for the TSM storage engine in Go.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
File I/O	<code>os.File</code> with manual buffering	Memory-mapped files (<code>mmap</code>) for zero-copy reads
Compression	Implement delta-of-delta and Gorilla from scratch	Use existing libraries but understand algorithms
Checksums	CRC32 using <code>hash/crc32</code>	More robust hash but CRC32 is sufficient
Concurrent Access	<code>sync.RWMutex</code> per file	Lock-free reads with atomic pointers
Block Cache	Simple <code>map</code> with LRU eviction	<code>sync.Map</code> or specialized cache with size limits

B. Recommended File/Module Structure

```
tempo/
├── cmd/
│   └── server/
│       └── main.go          # Entry point
├── internal/
│   ├── storage/
│   │   ├── engine.go        # StorageEngine implementation
│   │   ├── tsm/
│   │   │   ├── writer.go     # TSM file creation
│   │   │   ├── reader.go     # TSM file reading
│   │   │   ├── compression.go # Delta-of-delta, Gorilla
│   │   │   ├── file.go       # TSMFile struct and methods
│   │   │   └── index.go      # TSMIndex and IndexEntry
│   │   └── block_cache.go   # LRU block cache
│   └── memtable.go          # Memtable (for Milestone 2)
├── models/
│   ├── datapoint.go         # DataPoint struct
│   ├── series.go            # SeriesKey, Series
│   └── query.go             # Query, TimeRange
└── wal/
    └── wal.go               # Write-ahead log (Milestone 2)
└── pkg/
    └── mmap/
        └── mmap.go            # Memory-mapping utilities
```

C. Infrastructure Starter Code

Memory-mapped file utility (`pkg/mmap/mmap.go`):

```
package mmap

import (
    "os"
    "syscall"
    "unsafe"
)

// Map memory-maps a file for read-only access

func Map(path string) ([]byte, error) {
    file, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    stat, err := file.Stat()
    if err != nil {
        return nil, err
    }
    size := stat.Size()

    // Handle empty files
    if size == 0 {
        return []byte{}, nil
    }

    // Memory map the file
```

GO

```
data, err := syscall.Mmap(int(file.Fd()), 0, int(size),
    syscall.PROT_READ, syscall.MAP_SHARED)

if err != nil {
    return nil, err
}

return data, nil
}

// Unmap unmaps a memory-mapped file

func Unmap(data []byte) error {
    if len(data) == 0 {
        return nil
    }

    return syscall.Munmap(data)
}

// ReadUint64 reads a little-endian uint64 from mapped memory

func ReadUint64(data []byte, offset int) uint64 {
    // Use unsafe for performance; ensure data is long enough
    return *(*uint64)(unsafe.Pointer(&data[offset]))
}

// ReadUint32 reads a little-endian uint32 from mapped memory

func ReadUint32(data []byte, offset int) uint32 {
    return *(*uint32)(unsafe.Pointer(&data[offset]))
}
```

TSM file constants and header (`internal/storage/tsm/file.go`):

```
package tsm
```

```
const (
```

```
    // MagicNumber identifies a TSM file
```

```
    MagicNumber uint32 = 0x16D1D1A5
```

```
    // Version is the current TSM file format version
```

```
    Version uint64 = 1
```

```
    // BlockHeaderSize is the size of a block header in bytes
```

```
    BlockHeaderSize = 16 // 2 x uint64
```

```
    // IndexEntrySize is the size of a single index entry
```

```
    IndexEntrySize = 8 + 8 + 8 + 4 // min, max, offset, size
```

```
    // DefaultMaxPointsPerBlock is the default number of points per block
```

```
    DefaultMaxPointsPerBlock = 1024
```

```
)
```

```
// TSMHeader represents the file header
```

```
type TSMHeader struct {
```

```
    Magic     uint32
```

```
    Version   uint64
```

```
}
```

```
// WriteHeader writes the TSM header to a writer
```

```
func WriteHeader(w io.Writer) error {
```

```
    header := TSMHeader{
```

```
        Magic:    MagicNumber,
```

GO

```

    Version: Version,
}

return binary.Write(w, binary.LittleEndian, header)

}

// ReadHeader reads and validates a TSM header

func ReadHeader(r io.Reader) (TSMHeader, error) {

var header TSMHeader

if err := binary.Read(r, binary.LittleEndian, &header.Magic); err != nil {

    return header, err
}

if err := binary.Read(r, binary.LittleEndian, &header.Version); err != nil {

    return header, err
}

if header.Magic != MagicNumber {

    return header, fmt.Errorf("invalid magic number: %x", header.Magic)
}

if header.Version != Version {

    return header, fmt.Errorf("unsupported version: %d", header.Version)
}

return header, nil
}

```

D. Core Logic Skeleton Code

TSM Writer (`internal/storage/tsm/writer.go`):

```
package tsm

import (
    "bytes"

    "encoding/binary"

    "hash/crc32"

    "io"

    "os"

    "tempo/internal/models"
)

// TSMWriter creates TSM files from series data

type TSMWriter struct {

    file      *os.File

    buf       *bytes.Buffer

    index     map[string][]IndexEntry

    blockSize int
}

// NewTSMWriter creates a new TSM writer

func NewTSMWriter(path string, blockSize int) (*TSMWriter, error) {

    file, err := os.Create(path)

    if err != nil {

        return nil, err
    }

    writer := &TSMWriter{
```

GO

```
    file:      file,
    buf:       bytes.NewBuffer(make([]byte, 0, 64*1024)),
    index:     make(map[string][]IndexEntry),
    blockSize: blockSize,
}

// TODO 1: Write TSM file header (magic + version)

// Use WriteHeader function from file.go

return writer, nil
}

// WriteSeries writes a series to the TSM file

func (w *TSMWriter) WriteSeries(key string, points []models.DataPoint) error {

    // TODO 2: Sort points by timestamp (ensure ascending order)

    // TODO 3: Partition points into blocks of max blockSize points

    // For each partition:

    // - Create a block with timestamps and values separated

    // - Apply delta-of-delta compression to timestamps

    // - Apply Gorilla XOR compression to values

    // - Calculate CRC32 checksum of the compressed block

    // - Write block header (min/max timestamps)

    // - Write compressed timestamps

    // - Write compressed values

    // - Write checksum

    // - Record index entry with min/max time, offset, size
}
```

```
// TODO 4: Write series key (length-prefixed string)

// TODO 5: Write all blocks for this series

return nil

}

// Finish writes the index and footer, then closes the file

func (w *TSMWriter) Finish() error {

    // TODO 6: Write index section:
    //     - For each series in w.index:
    //         * Write series key
    //         * Write count of blocks
    //         * For each block: write min/max time, offset, size

    // TODO 7: Write footer: offset to start of index (uint64)

    // TODO 8: Sync file to disk

    return w.file.Close()
}

// compressBlock compresses timestamps and values for a block

func compressBlock(timestamps []uint64, values []float64) ([]byte, []byte, error) {

    // TODO 9: Implement delta-of-delta compression for timestamps
    //     - First timestamp stored as-is (uint64)
    //     - First delta = timestamp[1] - timestamp[0]
```

```
//     - For i > 1: delta = timestamp[i] - timestamp[i-1]
//
//           deltaOfDelta = delta - previousDelta
//
//           Encode deltaOfDelta as VarInt

// TODO 10: Implement Gorilla XOR compression for float64 values

//     - First value stored as raw bits (uint64)

//     - For each subsequent value:
//
//         * XOR with previous value
//
//         * If XOR == 0: store single 0 bit
//
//         * Else: store 1 bit + leading zeros + meaningful bits

return nil, nil, nil

}
```

TSM Reader ([internal/storage/tsm/reader.go](#)):

```
package tsm
```

GO

```
import (
    "bytes"
    "encoding/binary"
    "fmt"
    "io"

    "tempo/internal/models"
    "tempo/pkg/mmap"
)
```

```
// TSMReader reads TSM files using memory-mapped I/O
```

```
type TSMReader struct {
    path string
    data []byte // Memory-mapped file data
    index TSMIndex
}
```

```
// OpenTSMReader opens and memory-maps a TSM file
```

```
func OpenTSMReader(path string) (*TSMReader, error) {
```

```
    // TODO 1: Memory-map the file using mmap.Map()
```

```
    // TODO 2: Read and validate header from beginning of data
```

```
    // TODO 3: Read footer (last 8 bytes) to get index offset
```

```
    // TODO 4: Parse index section into TSMIndex structure
```

```
//      - Iterate through index entries

//      - Build map[string][]IndexEntry

return &TSMReader{

    path:  path,
    data:  data,
    index: index,
}, nil

}

// ReadBlock reads and decompresses a specific block

func (r *TSMReader) ReadBlock(seriesKey string, blockIndex int) ([]models.DataPoint, error)
{
    // TODO 5: Look up series in index, get block metadata

    // TODO 6: Validate blockIndex is within bounds

    // TODO 7: Read block from offset in memory-mapped data
    //      - Read header (min/max timestamps)
    //      - Read compressed timestamps and values
    //      - Verify CRC32 checksum

    // TODO 8: Decompress timestamps (delta-of-delta decoding)

    // TODO 9: Decompress values (Gorilla XOR decoding)

    // TODO 10: Reconstruct DataPoint slice from timestamps and values
}
```

```
    return nil, nil

}

// TimeRange returns the minimum and maximum timestamps in the file

func (r *TSMReader) TimeRange() (uint64, uint64) {

    // TODO 11: Iterate through all index entries to find global min/max

    // Return 0,0 if no data


    return 0, 0

}

// Close unmaps the memory and releases resources

func (r *TSMReader) Close() error {

    // TODO 12: Unmap memory using mmap.Unmap()

    return nil

}

// decompressTimestamps decodes delta-of-delta compressed timestamps

func decompressTimestamps(data []byte, count int) ([]uint64, error) {

    // TODO 13: Read first timestamp (uint64)

    // TODO 14: Read first delta (VarInt)

    // TODO 15: For remaining points:

    //     - Read delta-of-delta (VarInt)

    //     - Reconstruct delta = previousDelta + deltaOfDelta

    //     - Reconstruct timestamp = previousTimestamp + delta
```

```

    return nil, nil

}

// decompressValues decodes Gorilla XOR compressed float64 values

func decompressValues(data []byte, count int) ([]float64, error) {

    // TODO 16: Implement Gorilla XOR decompression

    // - Read first value as raw bits

    // - For each subsequent value:

    //     * Read control bit

    //     * If 0: value = previous value

    //     * If 1: read leading zeros, meaningful bits length, and bits

    //             Reconstruct XOR value, then XOR with previous to get current

    //             Reconstruct XOR value, then XOR with previous to get current

    return nil, nil
}

```

E. Language-Specific Hints

- Memory-mapping:** Use `syscall.Mmap` directly for control, but ensure proper error handling and `Munmap` on close.
- VarInt encoding:** Implement using loops shifting 7 bits at a time. The standard library's `binary.PutUvarint` and `binary.Uvarint` work but may be slower than custom implementation.
- Bit-level operations:** For Gorilla compression, you'll need to read/write individual bits. Use a `bitReader` / `bitWriter` wrapper around byte slices.
- CRC32:** Use `hash/crc32` with IEEE polynomial: `crc32.ChecksumIEEE(data)`.
- Concurrency:** Use `sync.RWMutex` for `TSMReader` if it might be accessed concurrently. Better: make `TSMReader` immutable after creation.
- File I/O:** Always check errors from file operations. Use `defer` for cleanup.

F. Milestone Checkpoint

To verify Milestone 1 implementation:

- Create a test TSM file:**

```
go test ./internal/storage/tsm/... -v -run TestTSMWriter
```

BASH

Expected output should show:

- TSM file created successfully
- Compression ratio reported (e.g., "Compressed 1024 points to 5120 bytes")
- No errors during write or read

2. Manual verification:

```
# Create a simple test program

cat > test_tsm.go << 'EOF'

package main

import (
    "fmt"
    "tempo/internal/storage/tsm"
)

func main() {

    writer, _ := tsm.NewTSMWriter("/tmp/test.tsm", 1024)
    // Add test data
    writer.Finish()

    reader, _ := tsm.OpenTSMReader("/tmp/test.tsm")
    points, _ := reader.ReadBlock("cpu", 0)
    fmt.Printf("Read %d points\n", len(points))
    reader.Close()
}

EOF

go run test_tsm.go
```

BASH

Expected: "Read 1024 points" with correct timestamps and values.

3. Check for common issues:

- **Issue:** "invalid magic number" error when reading
 - **Fix:** Ensure binary.Write uses `binary.LittleEndian` consistently
- **Issue:** CRC32 mismatch when reading blocks
 - **Fix:** Verify you're checksumming the exact bytes written (including header)
- **Issue:** Gorilla decompression produces NaN values
 - **Fix:** Check bit manipulation logic; use `math.Float64frombits()` for conversion

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Query returns no data for valid time range	Block index min/max times incorrect	Add debug logging to show which blocks are being selected	Ensure min/max in index match actual data in block
Decompression produces wrong timestamps	VarInt encoding/decoding bug	Write unit test with known sequence: [1000, 1001, 1003, 1006]	Step through decompression with small test case
Memory usage grows with many TSM files	Not unmapping closed files	Add finalizer or explicit Close() calls	Ensure every OpenTSMReader has matching Close()
ReadBlock returns fewer points than expected	Block size miscalculation during write	Check partition logic in WriteSeries	Ensure points are sorted before partitioning
Gorilla compression makes data larger	Highly volatile values	Test with constant values first, then random	Fall back to no compression if XOR doesn't help

Write Path Design

Milestone(s): Milestone 2: Write Path

The write path is the critical pipeline through which all incoming data flows from ingestion to durable storage. This design must balance three competing requirements: **low-latency acknowledgment** to clients, **strong durability guarantees**, and **efficient batching** for storage optimization. In TempoDB, we achieve this through

a multi-stage pipeline that validates, logs, buffers, and finally persists data points in optimized columnar format.

Mental Model: The Airport Check-in and Baggage System

Imagine an airport's check-in and baggage handling system. Passengers (data points) arrive at the terminal (the API). Each passenger checks in at a counter (validation) and receives a boarding pass (acknowledgment) once their luggage is registered in the system. The luggage itself is placed on a conveyor belt (the Write-Ahead Log) that immediately moves it to a secure holding area (the memtable). Luggage from many flights accumulates in the holding area until a critical mass for a particular destination is reached. Then, all luggage for that flight is efficiently packed into containers (compressed blocks) and loaded onto the plane (TSM file) for the journey to long-term storage (disk).

This analogy captures the key principles:

- **Immediate acknowledgment:** Passengers receive their boarding pass quickly, without waiting for the plane to be loaded.
- **Durability via logging:** The conveyor belt (WAL) ensures luggage isn't lost if the holding area has an issue.
- **Batched efficiency:** Luggage is packed in bulk, not piece-by-piece, optimizing space and effort.
- **Temporal organization:** Luggage is sorted by destination (series) and flight time (timestamp).

Write-Ahead Log and Memtable

The first two components of the write path work in tandem to provide durability and in-memory buffering.

Write-Ahead Log (WAL)

The **Write-Ahead Log** is an append-only file that records every write operation before it's acknowledged to the client. This ensures that even if the process crashes after acknowledgment but before data reaches persistent storage, the operation can be replayed during recovery. The WAL's sole purpose is durability—it's not optimized for random reads.

Design Decisions:

Decision: WAL Format and Segmentation

- **Context:** We need a durable log that can handle high write throughput, support efficient recovery scans, and manage disk space without unbounded growth.
- **Options Considered:**
 1. **Single monolithic log file:** Append all writes to one ever-growing file.
 2. **Segmented log with rotation:** Split the log into fixed-size segments; close and create a new segment when current segment reaches size limit.
- **Decision:** Use segmented log with rotation.
- **Rationale:**
 - **Easier space management:** Old segments can be deleted after their data is flushed to TSM files.
 - **Parallel recovery:** Multiple segments can be scanned in parallel during recovery.
 - **Simplified implementation:** Each segment is a simple append-only file with a predictable maximum size.
- **Consequences:**
 - Requires managing multiple files and cleaning up obsolete segments.
 - Adds complexity for tracking which segments are still needed (not yet flushed).

Option	Pros	Cons	Chosen?
Single monolithic file	Simple to implement; no file management overhead	Unbounded growth; difficult to clean up old data; recovery scans entire log	No
Segmented rotation	Bounded segment size; easy cleanup; parallel recovery possible	Must manage multiple files; need to track segment state	Yes

Each WAL segment file contains a sequence of **entries**, where each entry corresponds to one or more data points written in a batch. The segment format is:

Field	Type	Description
Entry Length	uint32 (4 bytes)	Length of the entire entry (including this field and checksum)
Checksum	uint32 (4 bytes)	CRC32 checksum of the entry data (for corruption detection)
Entry ID	uint64 (8 bytes)	Monotonically increasing identifier for ordering and deduplication
Batch Data	[]byte (variable)	Serialized batch of data points (using protocol buffers or custom binary format)

The `Segment` type manages an individual segment file:

Field	Type	Description
<code>path</code>	<code>string</code>	Filesystem path to the segment file
<code>file</code>	<code>*os.File</code>	Open file handle for appending
<code>mu</code>	<code>sync.RWMutex</code>	Protects concurrent writes and closes
<code>size</code>	<code>int64</code>	Current size of the segment in bytes
<code>maxSize</code>	<code>int64</code>	Maximum size before rotation (from <code>SegmentConfig.MaxSizeBytes</code>)
<code>closed</code>	<code>bool</code>	Whether the segment is closed for writing
<code>firstID</code>	<code>uint64</code>	Entry ID of the first entry in this segment
<code>lastID</code>	<code>uint64</code>	Entry ID of the last entry written

WAL Operations:

- Write Entry:** When a batch of points arrives, serialize them, calculate checksum, append to current segment with monotonic ID.
- Sync Policy:** Configure via `SegmentConfig.SyncInterval` (periodic sync) or `SyncOnwrite` (sync every write). Trade-off between durability and performance.
- Segment Rotation:** When `size >= maxSize`, close current segment, create new one with `firstID = lastID + 1`.
- Recovery Scan:** On startup, scan all non-flushed segments in ID order, replay entries to rebuild memtable.

Key Insight: The WAL must be fsynced to disk before acknowledging writes to the client. Without this, a power loss could lose acknowledged writes, violating durability guarantees. In Go, use `file.Sync()` after writing the entry.

Memtable

The **memtable** (memory table) is an in-memory buffer that holds recently written data points in sorted order, organized by series. It serves two purposes: (1) provides a fast read path for recent data, and (2) batches points for efficient flushing to disk.

Design Decisions:

Decision: Memtable Data Structure

- **Context:** We need an in-memory structure that supports fast inserts and range scans, organized by series key and timestamp.
- **Options Considered:**
 1. **Sorted map of maps:** `map[SeriesKey]map[timestamp]value` (unordered timestamps within series).
 2. **Sorted map with slices:** `map[SeriesKey][]DataPoint` with points kept sorted.
 3. **Skip list per series:** `map[SeriesKey]*skiplist` for ordered points.
- **Decision:** Use sorted map with slices (option 2).
- **Rationale:**
 - **Simplicity:** Slices are native Go types with predictable memory overhead.
 - **Good for sequential writes:** Time-series data typically arrives in roughly chronological order; appending to slice is $O(1)$.
 - **Efficient flushing:** When flushing, we can iterate series and timestamps in order without additional sorting.
- **Consequences:**
 - Out-of-order writes require binary search and insertion ($O(n)$ worst-case).
 - Must handle slice growth and potential copying.

The memtable structure in practice:

```
// Conceptual structure (not actual code in Layer 1)          GO

type Memtable struct {

    data map[SeriesKey][]DataPoint // Series → sorted points

    size int64                   // Approximate memory usage in bytes

    mu   sync.RWMutex            // For concurrent access

}
```

Memtable Operations:

Method	Parameters	Returns	Description
Insert	SeriesKey , DataPoint	error	Adds a point to the appropriate series slice, maintaining sorted order by timestamp
GetRange	SeriesKey , TimeRange	[]DataPoint	Returns all points for a series within time range (binary search)
Size	-	int64	Returns approximate memory usage in bytes
Flush	-	map[SeriesKey] []DataPoint	Returns all data and clears the memtable (becomes immutable)

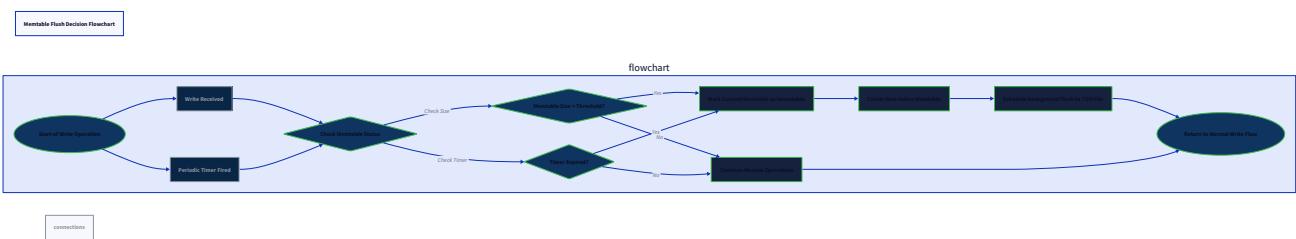
Series Cardinality Tracking: As points are inserted, we track the number of unique series keys (`len(memtable.data)`). This is critical for monitoring and query planning, as high cardinality (millions of series) can impact performance.

Flush Mechanism and Out-of-Order Writes

The memtable is a temporary buffer; eventually, its contents must be written to persistent TSM files. This process is called **flushing**.

Flush Triggers

A flush is triggered by one or more conditions, implemented as a decision flowchart (see



):

1. **Size-based:** When the memtable's estimated memory usage exceeds `Config.MaxMemtableSize` (e.g., 256 MB).
2. **Time-based:** A periodic timer (e.g., every 5 minutes) ensures data doesn't stay in memory too long.
3. **Manual:** Via administrative command or shutdown hook.

When a flush is triggered, the current active memtable is **marked as immutable**, and a new empty memtable becomes active for new writes. This allows writes to continue uninterrupted while flushing occurs in the background.

Flush Process

The flush process converts an immutable memtable to one or more TSM files:

1. **Sort and Partition:** Group points by series key, ensuring timestamps are sorted within each series.
2. **Create TSM Writer:** For each series, write points in batches of `DefaultMaxPointsPerBlock` (1024) to a new TSM file.
3. **Apply Compression:** Use delta-of-delta encoding for timestamps and Gorilla XOR for values (as described in Storage Engine Design).
4. **Write Index:** After all series are written, build and append the index mapping series keys to block locations.
5. **Persist:** Call `Finish()` to write footer and sync the TSM file to disk.
6. **Update Metadata:** Update the storage engine's file list and series index.
7. **Cleanup WAL:** Once the TSM file is durably written, mark the corresponding WAL segments as safe to delete (up to the highest entry ID included in the flush).

Handling Out-of-Order Writes

Time-series data may arrive with timestamps that are not monotonically increasing—common in distributed systems with clock skew or batch replays. TempoDB must handle these **out-of-order writes** correctly.

Strategy: Insertion Sort with Tolerance Window

For each series, we maintain points sorted by timestamp. When a new point arrives:

1. Check if its timestamp is **after the last point** for that series → append (common case, O(1)).
2. If before the last point, perform binary search to find insertion position.
3. Insert into slice (may require shifting elements, O(n) worst-case).

To limit the performance impact, we define a **tolerance window** (e.g., 1 hour). Points arriving more than this window before the latest point are rejected as "too old" (configurable). Points within the window are accepted but may incur the insertion cost.

Design Principle: Optimize for the common case (in-order writes) while supporting bounded out-of-order writes. This balances performance with practical requirements.

Late Arrivals During Flush: A more complex scenario occurs when a point arrives for a series that is currently being flushed (the memtable is immutable). Two approaches:

Approach	Description	Pros	Cons	Chosen?
Block and Insert	Block writes to that series until flush completes, then insert into new memtable	Simple; ensures correctness	Blocks writes; poor concurrency	No
Write to New Memtable with Tombstone	Write point to active memtable, mark flushed data with "tombstone" to indicate missing point	Non-blocking; good performance	Complex; requires compaction to merge; tombstone overhead	Yes

We choose the second approach: out-of-order points for series being flushed go to the active memtable. The flushed TSM file contains data up to time T. The new memtable contains points with timestamps < T. During query execution, we must merge points from both TSM file and memtable. This is handled during compaction (Milestone 4), which will merge overlapping time ranges.

Common Pitfalls

⚠️ Pitfall: Not Fsyncing WAL Before Acknowledgment

- **Description:** Acknowledging writes to the client before calling `fsync()` on the WAL.
- **Why it's wrong:** If the OS crashes or loses power, the write may exist only in page cache, not on durable storage. The client believes data is saved, but it's lost.
- **Fix:** Always call `file.Sync()` (or set `SyncOnWrite = true`) before sending acknowledgment.

⚠️ Pitfall: Unbounded Memtable Growth

- **Description:** Not enforcing size limits on memtable, allowing it to consume all available memory.
- **Why it's wrong:** Can lead to OOM crashes, especially during write bursts.
- **Fix:** Implement strict size-based flushing and backpressure (see below).

⚠️ Pitfall: Ignoring Clock Skew

- **Description:** Assuming all timestamps arrive in order, causing incorrectly sorted data.
- **Why it's wrong:** Queries may return incorrect results (out-of-order points), and compression efficiency suffers.
- **Fix:** Implement out-of-order insertion logic with a reasonable tolerance window.

⚠️ Pitfall: Blocking Writes During Flush

- **Description:** Making the write path wait for flush completion.
- **Why it's wrong:** Kills write throughput during heavy load.
- **Fix:** Use immutable memtables and background flushing; writes continue to new active memtable.

⚠️ Pitfall: Not Handling Backpressure

- **Description:** Accepting writes indefinitely even when system is overloaded.
- **Why it's wrong:** Leads to uncontrolled memory growth and eventual crash.
- **Fix:** Implement backpressure mechanism (e.g., pause accepting writes when memtable size reaches 90% of limit).

Implementation Guidance (Milestone 2)

This section provides concrete implementation steps for the write path components.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
WAL Storage	Append-only files with <code>os.File</code>	Memory-mapped files for zero-copy reads during recovery
Memtable Structure	<code>map[string][]DataPoint</code> with binary search insert	Partitioned memtables per series hash to reduce lock contention
Concurrency Control	<code>sync.RWMutex</code> per memtable	Lock-free ring buffer for writes, CAS for memtable swap
Batch Serialization	Custom binary format with <code>encoding/binary</code>	Protocol Buffers for forward compatibility

B. Recommended File/Module Structure

```
tempo/
├── cmd/
│   └── server/
│       └── main.go          # Entry point
└── internal/
    ├── storage/
    │   ├── engine.go         # StorageEngine (coordinates WAL, memtable, TSM)
    │   ├── wal/
    │   │   ├── wal.go         # WAL manager
    │   │   ├── segment.go     # Segment implementation (starter code below)
    │   │   └── recovery.go    # Recovery scanning logic
    │   ├── memtable/
    │   │   ├── memtable.go    # Memtable implementation
    │   │   └── manager.go     # MemtableManager (handles immutable/flush)
    │   └── tsm/
    │       ├── writer.go       # TSMWriter (from Milestone 1)
    │       └── reader.go       # TSMReader (from Milestone 1)
    ├── api/
    │   ├── http.go            # HTTP write/query handlers
    │   └── ingest.go          # Batch ingestion logic
    └── models/
        ├── point.go           # DataPoint, SeriesKey types
        └── query.go            # Query, TimeRange types
```

C. Infrastructure Starter Code

Complete WAL Segment Implementation

GO

```
// internal/storage/wal/segment.go

package wal

import (
    "encoding/binary"
    "fmt"
    "hash/crc32"
    "io"
    "os"
    "path/filepath"
    "sync"
    "time"
)

var (
    byteOrder = binary.BigEndian
    crcTable = crc32.MakeTable(crc32.Castagnoli)
)

// SegmentConfig defines configuration for WAL segments.

type SegmentConfig struct {
    MaxSizeBytes    int64          // Maximum size before rotation
    SyncInterval    time.Duration // How often to sync (0 = never auto-sync)
    SyncOnWrite     bool           // Sync after every write (strongest durability)
}

// Segment manages a single WAL segment file.

type Segment struct {
    path    string
```

```
file    *os.File
mu      sync.RWMutex
size    int64
maxSize int64
closed   bool
firstID uint64
lastID  uint64

config SegmentConfig
syncTicker *time.Ticker
}

// NewSegment creates or opens a WAL segment file.

func NewSegment(path string, firstID uint64, config SegmentConfig) (*Segment, error) {
    // Create directory if needed
    if err := os.MkdirAll(filepath.Dir(path), 0755); err != nil {
        return nil, fmt.Errorf("create wal directory: %w", err)
    }

    // Open file in append mode, create if not exists
    file, err := os.OpenFile(path, os.O_CREATE|os.O_RDWR|os.O_APPEND, 0644)
    if err != nil {
        return nil, fmt.Errorf("open wal segment: %w", err)
    }

    // Get current size
    info, err := file.Stat()
```

```
if err != nil {
    file.Close()
    return nil, fmt.Errorf("stat wal segment: %w", err)
}

seg := &Segment{
    path:      path,
    file:      file,
    size:      info.Size(),
    maxSize:   config.MaxSizeBytes,
    firstID:   firstID,
    lastID:   firstID - 1, // Will be incremented on first write
    config:   config,
}

// If file has existing data, scan to find lastID
if info.Size() > 0 {
    if err := seg.scanToFindLastID(); err != nil {
        file.Close()
        return nil, fmt.Errorf("recover segment lastID: %w", err)
    }
}

// Start periodic sync if configured
if config.SyncInterval > 0 {
    seg.syncTicker = time.NewTicker(config.SyncInterval)
    go seg.periodicSync()
```

```
}

return seg, nil

}

// WriteEntry appends an entry to the segment.

func (s *Segment) WriteEntry(data []byte) (uint64, error) {

    s.mu.Lock()

    defer s.mu.Unlock()

    if s.closed {

        return 0, fmt.Errorf("segment closed")

    }

    // Check if we need to rotate

    if s.size >= s.maxSize {

        return 0, fmt.Errorf("segment full")

    }

    entryID := s.lastID + 1

    entry := encodeEntry(entryID, data)

    // Write to file

    n, err := s.file.Write(entry)

    if err != nil {

        return 0, fmt.Errorf("write entry: %w", err)

    }
```

```
// Sync if configured

if s.config.SyncOnWrite {

    if err := s.file.Sync(); err != nil {

        // If sync fails, the write may not be durable.

        // In production, we might want to truncate or mark segment as corrupt.

        return 0, fmt.Errorf("sync entry: %w", err)

    }

}

s.size += int64(n)

s.lastID = entryID


return entryID, nil

}

// Scan reads all entries from the segment.

func (s *Segment) Scan(fn func(id uint64, data []byte) error) error {

    s.mu.RLock()

    defer s.mu.RUnlock()


    if s.closed {

        return fmt.Errorf("segment closed")

    }


    // Seek to beginning

    if _, err := s.file.Seek(0, io.SeekStart); err != nil {
```

```
        return fmt.Errorf("seek to start: %w", err)
    }

reader := io.Reader(s.file)

for {

    entry, id, err := decodeEntry(reader)

    if err == io.EOF {
        break
    }

    if err != nil {
        return fmt.Errorf("decode entry at offset %d: %w", s.size, err)
    }

    if err := fn(id, entry); err != nil {
        return err
    }
}

return nil
}

// Close unmaps memory and releases resources.

func (s *Segment) Close() error {
    s.mu.Lock()
    defer s.mu.Unlock()

    if s.syncTicker != nil {

```

```
s.syncTicker.Stop()

}

s.closed = true

return s.file.Close()

}

// Delete removes the segment file from disk.

func (s *Segment) Delete() error {

    s.mu.Lock()

    defer s.mu.Unlock()

    if !s.closed {

        if err := s.Close(); err != nil {

            return err

        }

    }

    return os.Remove(s.path)

}

// Helper functions

func encodeEntry(id uint64, data []byte) []byte {

    // Entry format: [4 length][4 checksum][8 id][data]

    // Length includes all fields (4+4+8+len(data))

    totalLen := 4 + 4 + 8 + len(data)

    buf := make([]byte, totalLen)
```

```
// Write length
byteOrder.PutUint32(buf[0:4], uint32(totalLen))

// Write placeholder for checksum
checksumPos := 4

// Write ID
byteOrder.PutUint64(buf[8:16], id)

// Copy data
copy(buf[16:], data)

// Compute and write checksum (over id + data)
checksum := crc32.Checksum(buf[8:], crcTable)
byteOrder.PutUint32(buf[checksumPos:checksumPos+4], checksum)

return buf
}

func decodeEntry(r io.Reader) ([]byte, uint64, error) {
    // Read length
    var lenBuf [4]byte
    if _, err := io.ReadFull(r, lenBuf[:]); err != nil {
        return nil, 0, err
    }
    totalLen := byteOrder.Uint32(lenBuf[:])
```

```
// Read entire entry

entry := make([]byte, totalLen)

copy(entry[0:4], lenBuf[:])

if _, err := io.ReadFull(r, entry[4:]); err != nil {

    return nil, 0, err
}

// Verify checksum

storedChecksum := byteOrder.Uint32(entry[4:8])

computedChecksum := crc32.Checksum(entry[8:], crcTable)

if storedChecksum != computedChecksum {

    return nil, 0, fmt.Errorf("checksum mismatch: stored %x, computed %x",
        storedChecksum, computedChecksum)
}

// Extract ID and data

id := byteOrder.Uint64(entry[8:16])

data := entry[16:]

return data, id, nil
}

func (s *Segment) scanToFindLastID() error {

lastID := s.firstID - 1

err := s.Scan(func(id uint64, data []byte) error {

    if id != lastID+1 {

        return fmt.Errorf("non-sequential ID: expected %d, got %d", lastID+1, id)
    }
})
```

```
        }

        lastID = id

        return nil
    })

    if err != nil {
        return err
    }

    s.lastID = lastID

    return nil
}

func (s *Segment) periodicSync() {
    for range s.syncTicker.C {

        s.mu.Lock()

        if !s.closed {

            s.file.Sync() // Ignore error for periodic sync

        }

        s.mu.Unlock()
    }
}
```

D. Core Logic Skeleton Code

Memtable Implementation Skeleton

GO

```
// internal/storage/memtable/memtable.go

package memtable

import (
    "sort"
    "sync"
    "time"

    "tempo/internal/models"
)

// Memtable holds incoming data points in memory before flushing to disk.

type Memtable struct {

    data map[string][]models.DataPoint // key: SeriesKey.String()
    size int64                         // Approximate memory usage in bytes
    mu   sync.RWMutex

    // For out-of-order handling
    maxOutOfOrderWindow time.Duration
}

// NewMemtable creates a new empty memtable.

func NewMemtable(maxOutOfOrderWindow time.Duration) *Memtable {
    return &Memtable{
        data: make(map[string][]models.DataPoint),
        size: 0,
        maxOutOfOrderWindow: maxOutOfOrderWindow,
    }
}
```

```
}

// Insert adds a point to the appropriate series slice, maintaining sorted order.

func (m *Memtable) Insert(seriesKey models.SeriesKey, point models.DataPoint) error {

    // TODO 1: Convert seriesKey to string key using seriesKey.String()

    // TODO 2: Lock the memtable for writing (m.mu.Lock())

    // TODO 3: Get the slice for this series (create if doesn't exist)

    // TODO 4: Check if point is out of order:

    //     - If slice is empty, append and update size

    //     - If point timestamp > last point timestamp, append (common case)

    //     - Else, binary search to find insertion position

    // TODO 5: Validate out-of-order window: if point is too old (beyond
maxOutOfOrderWindow

        // compared to latest point), return error

    // TODO 6: Insert at correct position (may need to shift elements)

    // TODO 7: Update m.size with approximate size of new point

    // TODO 8: Return any error (e.g., out of window)

    return nil
}

// GetRange returns all points for a series within time range.

func (m *Memtable) GetRange(seriesKey models.SeriesKey, tr models.TimeRange)
[]models.DataPoint {

    // TODO 1: Convert seriesKey to string key

    // TODO 2: Lock for reading (m.mu.RLock())

    // TODO 3: Get slice for series

    // TODO 4: If no points, return empty slice

    // TODO 5: Binary search to find start index (first point >= tr.Start)
```

```
// TODO 6: Iterate from start index while point timestamp <= tr.End

// TODO 7: Collect points into result slice

// TODO 8: Return result


return nil

}

// Size returns approximate memory usage in bytes.

func (m *Memtable) Size() int64 {

    m.mu.RLock()

    defer m.mu.RUnlock()

    return m.size

}

// Flush returns all data and clears the memtable.

func (m *Memtable) Flush() map[string][]models.DataPoint {

    // TODO 1: Lock for writing (m.mu.Lock())

    // TODO 2: Create copy of m.data

    // TODO 3: Reset m.data to new empty map

    // TODO 4: Reset m.size to 0

    // TODO 5: Return the copied data

    return nil

}

// Helper function for binary search in sorted DataPoint slice

func findInsertIndex(points []models.DataPoint, ts time.Time) int {

    // TODO: Implement binary search returning index where point should be inserted

    // Use sort.Search with comparison on point.Timestamp
```

```
    return 0  
}  
}
```

Storage Engine Write Path Skeleton

GO

```
// internal/storage/engine.go (partial)

package storage

import (
    "context"
    "fmt"
    "sync"
    "time"

    "tempo/internal/models"
    "tempo/internal/storage/memtable"
    "tempo/internal/storage/wal"
)

// StorageEngine coordinates writes and reads.

type StorageEngine struct {
    config Config
    wal     *wal.WAL
    memtables *MemtableManager
    // ... other fields from naming conventions

    flushCh chan *memtable.Memtable
    stopCh  chan struct{}
}

// WritePoint writes a single data point.

func (e *StorageEngine) WritePoint(ctx context.Context,
    seriesKey models.SeriesKey, point models.DataPoint) error {
```

```
// TODO 1: Validate point (timestamp not in future, etc.)  
  
    // TODO 2: Batch this single point with others if possible, or create single-point  
batch  
  
    // TODO 3: Serialize batch to bytes (e.g., using encodeBatch helper)  
  
    // TODO 4: Write to WAL with wal.WriteEntry(serialized)  
  
    // TODO 5: If WAL write succeeds, insert into active memtable  
  
    // TODO 6: Check if memtable needs flush (size > threshold)  
  
    // TODO 7: If flush needed, trigger async flush via flushCh  
  
    // TODO 8: Return any error (e.g., WAL write failed)  
  
  
    return nil  
}  
  
// WritePointsBatch writes multiple data points efficiently.  
  
func (e *StorageEngine) WritePointsBatch(ctx context.Context,  
    points []models.DataPoint) error {  
  
    // TODO 1: Group points by series key for efficient serialization  
  
    // TODO 2: Apply backpressure: check if memtable size > 90% of max  
        // If yes, wait a bit or return error  
  
    // TODO 3: Serialize batch  
  
    // TODO 4: Write to WAL  
  
    // TODO 5: Insert all points into memtable  
  
    // TODO 6: Check flush condition  
  
    // TODO 7: Return any error  
  
  
    return nil
```

```

}

// flushMemtable flushes an immutable memtable to TSM files.

func (e *StorageEngine) flushMemtable(mt *memtable.Memtable) error {

    // TODO 1: Get all data from memtable via mt.Flush()

    // TODO 2: Create new TSM writer (tsm.NewTSMWriter)

    // TODO 3: For each series in memtable data:
    //         a. Sort points by timestamp (should already be sorted)
    //         b. Write in batches of DefaultMaxPointsPerBlock using writer.WriteSeries

    // TODO 4: Finish TSM file (writer.Finish())

    // TODO 5: Register new TSM file in engine's file list

    // TODO 6: Notify WAL that entries up to certain ID are durable

    // TODO 7: Schedule deletion of old WAL segments

    // TODO 8: Update series index metadata

    return nil
}

```

E. Language-Specific Hints

- Concurrency:** Use `sync.RWMutex` for memtable access. For higher throughput, consider sharding memtables by series key hash to reduce lock contention.
- Memory Management:** Estimate memtable size as: `(len(seriesKey) + 16) * points` (timestamp: 8 bytes, value: 8 bytes). Use `runtime.MemStats` to monitor actual usage.
- Backpressure:** When memtable reaches 90% capacity, use a `sync.Cond` to make writers wait or return a "too many requests" error.
- WAL Sync:** Use `file.Sync()` for durability. For better performance, batch syncs with `SyncInterval` (e.g., 100ms) but acknowledge writes after WAL append (not after sync). This provides "group commit" semantics.
- Error Handling:** If WAL write fails, do NOT insert into memtable (data would be lost on crash). Return error to client for retry.

F. Milestone Checkpoint

After implementing Milestone 2, verify with:

```
# Run unit tests for write path components  
go test ./internal/storage/wal/... -v  
  
go test ./internal/storage/memtable/... -v  
  
go test ./internal/storage/... -run TestWritePath -v  
  
# Start the server and test write throughput  
  
go run cmd/server/main.go &  
  
# In another terminal, send test writes  
  
curl -X POST http://localhost:8080/write \  
-d "cpu,host=server01 value=0.64 $(date +%s%N)"  
  
curl -X POST http://localhost:8080/write \  
-d "cpu,host=server01 value=0.73 $(date +%s%N)"
```

BASH

Expected Behavior:

- Points are acknowledged immediately (HTTP 204).
- WAL files appear in data directory (`data/wal/segment-000001.wal`).
- Memtable grows in memory.
- When memtable reaches size threshold (or after 5 minutes), a TSM file is created in `data/tsm/`.
- After flush, corresponding WAL segments are deleted.

Signs of Trouble:

- No WAL files → WAL not being written before acknowledgment.
- TSM files not created → flush not triggering or failing silently.
- Memory grows unbounded → memtable size calculation or flush triggering broken.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Write returns error "segment full" but no flush happening	WAL rotation not working or max size too small	Check <code>SegmentConfig.MaxSizeBytes</code> ; look for logs about flush	Increase max size or ensure flushes run
High memory usage even after flush	Memtable data not being cleared after flush	Check <code>Memtable.Flush()</code> returns copy and resets internal map	Ensure <code>Flush()</code> creates new map, not sharing reference
Recovery loses recent writes on restart	WAL not synced to disk before acknowledgment	Check if <code>SyncOnWrite=true</code> or periodic sync interval	Enable <code>SyncOnWrite</code> or decrease sync interval
Out-of-order points cause panic or incorrect order	Binary search or insertion logic bug	Add test with random timestamp insertion; check sort order	Fix <code>findInsertIndex</code> and slice insertion logic
Write throughput decreases over time	Lock contention on single memtable	Profile with <code>pprof</code> ; look for lock wait times	Implement sharded memtables (advanced)

Query Engine Design

Milestone(s): Milestone 3: Query Engine

The query engine is the brain of TempoDB, responsible for interpreting user requests, locating relevant data, and transforming raw points into meaningful results. Unlike write-optimized paths that handle high-velocity sequential ingestion, the query engine must excel at **efficient scanning, filtering, and aggregation** over potentially massive historical datasets. This section details how TempoDB translates a declarative query into an efficient execution plan that leverages the storage engine's columnar layout and indexing to minimize data movement and computational overhead.

Mental Model: The Library Research Assistant

Imagine you're a researcher in a vast library (the database) requesting information about "average temperature in New York during July 2023." You don't want every single measurement; you need a summary.

A skilled research assistant (the query engine) follows a systematic process:

1. **Consult the Card Catalog (Index):** First, they check the card catalog (the series index and TSM file indexes) to identify which books (TSM files) contain temperature data for New York. They note the specific shelves (block offsets) where July 2023 data is located.
2. **Retrieve Relevant Pages (Block Pruning):** They don't bring you entire books. Instead, they go to the shelves, pull only the relevant chapters (data blocks) that overlap with your time range, ignoring books about other cities or months outside July.
3. **Photocopy Necessary Columns (Columnar Scan):** Within each chapter, they don't copy every word. They use a photocopier that can extract only the timestamp and temperature value columns (columnar projection), skipping irrelevant fields.
4. **Summarize Information (Aggregation):** Back at your desk, they don't dump 10,000 individual readings. They calculate the average temperature for each day (GROUP BY time(1d)), producing a concise table of 31 daily averages.
5. **Present Final Report (Streaming Results):** They hand you the summary table one row at a time (streaming) so you can start analyzing immediately, without waiting for the entire calculation to finish.

This mental model captures the essence of the query engine: **intelligent indexing, selective data retrieval, columnar efficiency, and early aggregation** to transform a broad request into a precise, resource-efficient answer.

Query Parsing, Planning, and Execution

Query processing in TempoDB follows a classic three-stage pipeline: parse the user's request into an abstract syntax tree (AST), create an optimized execution plan, and then execute that plan against the storage engine. The critical optimization is **predicate pushdown**—applying filters for time and tags as early as possible, ideally at the storage block layer, to avoid moving unnecessary data into memory.

1. Query Parsing

The query engine accepts queries in a SQL-like dialect extended with time-series specific constructs (e.g., `GROUP BY time(1h)`). The parser, typically implemented using a **recursive descent** approach, validates syntax and converts the textual query into a structured `Query` object defined in the data model.

Core Query Structure Table:

Field	Type	Description
Measurement	string	The target measurement (e.g., "cpu_usage").
Tags	map[string]string	Equality filters on tags (e.g., host="server1"). Multiple tags are combined with AND logic.
TimeRange	TimeRange	Mandatory start and end timestamps for the query. Supports relative times (e.g., now() - 1h).
Aggregate	*AggregateFunction	Optional aggregate function (SUM , AVG , MIN , MAX , COUNT). If nil , raw points are returned.
GroupByWindow	time.Duration	Optional time bucket width for GROUP BY time(...). Must be > 0 if aggregation is specified.
Fields	[]string	List of field names to return. For simplicity in v1, we assume a single numeric field (float64).

The parser handles expressions like:

```
SELECT avg(temperature) FROM weather WHERE city='NYC' AND time >= '2023-07-01' AND time < SQL
'2023-08-01' GROUP BY time(1d)
```

This would produce a `Query` object with `Measurement="weather"` , `Tags={"city":"NYC"}` , the corresponding `TimeRange` , `Aggregate=AggregateAvg` , and `GroupByWindow=24*time.Hour` .

Key Insight: Parsing is a pure syntactic transformation. No optimization or storage access occurs at this stage. The goal is to produce a valid, unambiguous internal representation.

2. Query Planning

The planner takes the parsed `Query` and the current state of the storage engine (list of TSM files, series index) and produces an executable `QueryPlan` . The plan's primary job is to **push predicates down** to the lowest possible level.

Query Plan Data Structure:

Component	Responsibility
Series Selection	Uses the <code>SeriesIndex</code> (map from series key to metadata) to find all <code>SeriesKey</code> values matching the <code>Measurement</code> and <code>Tags</code> predicates. This yields a list of series IDs or keys to fetch data for.
TSM File Selection	For each candidate series, consults the <code>TSMIndex</code> of each TSM file to identify which files contain blocks overlapping with the query's <code>TimeRange</code> . Files with <code>MaxTime < query.Start</code> or <code>MinTime > query.End</code> are skipped entirely.
Block-Level Pruning	Within each selected TSM file and series, the plan examines individual <code>IndexEntry</code> blocks. Blocks that don't overlap the time range are skipped. The <code>MinTime / MaxTime</code> in each index entry make this a constant-time check.
Aggregation Strategy	Decides where to perform aggregation: <ul style="list-style-type: none"> • Pushdown: For simple aggregates like <code>MIN / MAX</code>, the storage engine can return the pre-computed block-level min/max if available. • Partial Aggregation: For <code>SUM</code>, <code>COUNT</code>, <code>AVG</code>, the plan may instruct scanners to compute intermediate sums/counts per block, which are then merged.
Execution Order	Determines the scan order (typically series-by-series, then time-ordered blocks) to maximize temporal locality and cache friendliness.

The output is a tree of primitive operations: `SeriesScan -> TSMFileScan -> BlockScan -> [Filter] -> [Aggregate]`. The plan is stateless; it's a recipe for execution.

3. Query Execution

The executor runs the plan, coordinating between components to stream results back to the client. It employs an **iterator model** (also known as the Volcano model) where each stage pulls data from the stage below it. This enables pipelining and limits memory usage.

Execution Step-by-Step:

1. **Initialize Scanners:** For each series identified in the plan, create a `SeriesScanner` that manages scanning across multiple TSM files.
2. **Block Iterator:** Each `SeriesScanner` creates a `BlockIterator` for each relevant TSM file. The iterator uses the file's `TSMIndex` to seek to the first block with `MaxTime >= query.Start`.
3. **Block Decoding:** The iterator reads the compressed block from the memory-mapped file (via `TSMReader.ReadBlock`), decompresses the timestamps (delta-of-delta) and values (Gorilla XOR), and yields individual `DataPoint` objects within the query's time range.
4. **Filter Application:** Any remaining tag filters (beyond the equality filters used for series selection) are applied point-by-point. In v1, all tag filters are equality-based and pushed to series selection, so this step is often a no-op.

- Aggregation Window Management:** If `GroupByWindow` is specified, the executor maintains a sliding or tumbling window accumulator. As points are streamed in chronological order, they are added to the current time bucket. When a point's timestamp crosses a bucket boundary, the current bucket's aggregate is finalized and emitted, and a new accumulator is started.
- Streaming Output:** Final results (raw points or aggregated buckets) are written to an output channel or buffer as they are produced. The HTTP/gRPC handler streams these results to the client, interleaving computation and network I/O.

Sequence of Operations: The flow can be visualized in the sequence diagram:



Architecture Decision: Iterator-Based Execution Model

- Context:** We need to execute queries that may scan millions of points without loading all data into memory at once.
- Options Considered:**
 - Materialize-then-process:** Read all relevant points into a slice, then apply filters/aggregation.
 - Iterator (Volcano) model:** Each operator implements a `Next()` interface, pulling data through the pipeline.
 - Vectorized/batch model:** Operators process chunks of data (e.g., 1024 points) at a time.
- Decision:** Use the iterator model for v1.
- Rationale:** The iterator model is simpler to implement correctly, naturally supports streaming, and maps well to our block-by-block scanning pattern. While vectorized execution can be more CPU-efficient, the complexity outweighs the benefit for an educational codebase. The memory efficiency of streaming is critical for large range queries.
- Consequences:** We get low memory overhead and early result emission, but each point incurs virtual function call overhead. This can be optimized later by switching to batch processing within operators (e.g., process a full block of 1024 points at a time).

Option	Pros	Cons	Chosen?
Materialize-then-process	Simple imperative logic, easy debugging.	Memory explosion risk, delays first result.	No
Iterator Model	Streams results, memory efficient, composable.	Per-point call overhead, more complex state machines.	Yes
Vectorized Model	Excellent CPU cache utilization, modern best practice.	Significant implementation complexity, harder to debug.	No (future extension)

Aggregations and Downsampling

Aggregation transforms high-resolution raw data into summarized insights, which is fundamental for time-series analysis. TempoDB supports built-in aggregates and **tumbling window** grouping.

Built-in Aggregate Functions

The five core aggregates operate on the `Value` field of `DataPoint`:

Function	Computation	Notes
<code>COUNT</code>	Number of points in the group.	Includes <code>null</code> /missing? In v1, all points have values.
<code>SUM</code>	Sum of all values.	
<code>AVG</code>	<code>SUM / COUNT</code> .	Computed from intermediate sum and count to avoid precision loss.
<code>MIN</code>	Minimum value.	Can use block-level min as an optimization.
<code>MAX</code>	Maximum value.	Can use block-level max as an optimization.

Aggregation Algorithm (per series, per time bucket):

1. Initialize accumulators: `sum = 0`, `count = 0`, `min = +Inf`, `max = -Inf`.
2. For each `DataPoint` in chronological order within the bucket: a. `sum += point.Value` b. `count++` c. `min = math.Min(min, point.Value)` d. `max = math.Max(max, point.Value)`
3. At bucket boundary, compute final aggregate:
 - `COUNT` → `count`
 - `SUM` → `sum`
 - `AVG` → `sum / float64(count)`
 - `MIN` → `min`
 - `MAX` → `max`
4. Emit a new `DataPoint` representing the bucket. Its timestamp is the **start** of the bucket window (alignment is configurable, but default to start). Its value is the aggregate result.

5. Reset accumulators for the next bucket.

GROUP BY time() and Tumbling Windows

The `GROUP BY time(<interval>)` clause creates **tumbling windows**—contiguous, non-overlapping time intervals that partition the data. A point belongs to exactly one bucket based on its timestamp.

Window Alignment: Buckets are aligned to a fixed epoch (e.g., Unix epoch: 1970-01-01T00:00:00Z). The bucket for a timestamp `t` is calculated as:

```
bucket_start = epoch + floor((t - epoch) / interval) * interval  
bucket_end = bucket_start + interval
```

For example, with `interval=1h` and `epoch=0`, the timestamp `2023-07-01 10:22:00` falls into the bucket `[2023-07-01 10:00:00, 2023-07-01 11:00:00)`.

Execution with Streaming: The executor maintains current bucket state per series (or per group if grouping by tags). As points arrive in time order (guaranteed by the storage layout), the executor checks:

- If `point.Timestamp < current_bucket_end`, add to current accumulators.
- If `point.Timestamp >= current_bucket_end`, finalize and emit the current bucket, then advance the window (possibly by multiple intervals if there are gaps in data) and start a new accumulator.

Downsampling vs. On-the-Fly Aggregation

Downsampling is a form of **precomputed aggregation** stored persistently, typically applied to older data to save space while preserving trends. It's distinct from on-the-fly aggregation performed at query time.

Architecture Decision: On-the-Fly Aggregation Only for Milestone 3

- **Context:** We need to support aggregate queries over historical data. Precomputing downsamples improves query performance but adds storage and complexity.
- **Options:**
 1. **Always compute on-the-fly:** Execute the full aggregation pipeline for every query.
 2. **Pre-compute and store downsampled data:** Run background jobs to create lower-resolution aggregates, and route queries to appropriate resolution.
- **Decision:** Implement only on-the-fly aggregation for Milestone 3. Downsampling is deferred to Milestone 4 (Retention & Compaction).
- **Rationale:** On-the-fly aggregation is a prerequisite for downsampling and allows us to validate the correctness of aggregation logic first. It also keeps the query engine design focused on execution rather than storage lifecycle. Performance for large historical queries will be addressed later with rollups.
- **Consequences:** Queries over long time ranges will be slower initially, but the architecture is ready to incorporate pre-computed rollups as a transparent optimization layer later.

Handling Gaps in Data

Time-series data often has gaps (no points recorded for periods). The aggregation semantics must define behavior for empty buckets:

- `COUNT` returns 0.
- `SUM` , `AVG` , `MIN` , `MAX` return `null` (or skip the bucket). In v1, we may skip empty buckets entirely in the result set.

Common Pitfalls

⚠ Pitfall: Loading Entire Series into Memory Before Filtering

- **Description:** Implementing `SeriesScanner` by first reading *all* points for a series from all files into a slice, then applying time range filters.
- **Why It's Wrong:** Defeats the purpose of block-level pruning and can cause out-of-memory errors for long-running series. It also delays the first result.
- **Fix:** Implement a lazy `BlockIterator` that only decompresses the next needed block. Use the index's `MinTime` / `MaxTime` to skip blocks entirely.

⚠ Pitfall: Incorrect Bucket Alignment for GROUP BY time()

- **Description:** Aligning buckets to the query's start time instead of a fixed epoch, causing non-deterministic results and breaking cacheability.

- **Why It's Wrong:** The same absolute time interval queried at different times would produce different bucket boundaries, making results inconsistent and precomputed rollups impossible.
- **Fix:** Always align to a global epoch (Unix epoch). Use `floor((timestamp_nanos - epoch_nanos) / interval_nanos) * interval_nanos` to calculate bucket start.

! Pitfall: Forgetting to Handle Out-of-Order Points in Queries

- **Description:** Assuming points within a TSM file are perfectly sorted, but out-of-order writes (within tolerance) may have been inserted.
- **Why It's Wrong:** Aggregations that assume chronological order (like tumbling windows) may produce incorrect results if points arrive late within a block.
- **Fix:** The storage engine should guarantee that TSM files store points in sorted order. The memtable flush and compaction processes must sort points before writing blocks. The query engine can then rely on sorted streams.

! Pitfall: Not Pushing Time Predicate to Storage Index

- **Description:** Reading all blocks for a series and filtering points in memory after decompression.
- **Why It's Wrong:** Decompression is expensive. Wasting CPU and I/O on irrelevant data destroys performance.
- **Fix:** The `QueryPlan` must use the `TSMIndex MinTime / MaxTime` to skip blocks before they are read. The `BlockIterator.Seek()` method should jump to the first block with `MaxTime >= query.Start`.

! Pitfall: Floating-Precision Issues in AVG

- **Description:** Calculating average by accumulating `sum/count` in a loop with floating-point additions, leading to precision loss for large counts.
- **Why It's Wrong:** Time-series aggregations often run over millions of points. Standard floating-point error can become significant.
- **Fix:** Use Kahan summation or at least `float64` for accumulators. For v1, the error is acceptable for educational purposes, but document the limitation.

Implementation Guidance (Milestone 3)

This section provides concrete Go code skeletons and organization tips to implement the query engine.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Query Parsing	Handwritten recursive descent parser	Parser generator (ANTLR, pigeon)
Expression Evaluation	Switch on <code>AggregateFunction</code> enum	Abstract syntax tree (AST) interpreter
Execution Engine	Iterator model with <code>Next()</code> interface	Vectorized batch processing
Result Streaming	Channel of <code>DataPoint</code>	Custom <code>io.Writer</code> interface

B. Recommended File/Module Structure

```
tempo/
├── cmd/
│   └── server/                      # Main entry point
├── internal/
│   ├── query/                        # Query engine components
│   │   ├── parser.go                 # Query language parser
│   │   ├── planner.go                # Query plan generation
│   │   ├── executor.go              # Plan execution and streaming
│   │   ├── aggregator.go            # Aggregate function implementations
│   │   ├── iterator.go              # Block and series iterator interfaces
│   │   └── scanner.go                # TSM file scanner implementation
│   ├── storage/                     # From Milestone 1 & 2
│   │   ├── tsm/
│   │   └── wal/
│   └── models/                       # Shared data structures
│       ├── query.go                 # Query, TimeRange structs
│       └── series.go                # DataPoint, SeriesKey
└── pkg/
    └── api/                         # HTTP/gRPC handlers (Milestone 5)
```

C. Infrastructure Starter Code

Complete `models/query.go` (shared data structures):

```
package models

import "time"

// AggregateFunction represents the built-in aggregation functions.

type AggregateFunction int

const (
    AggregateNone AggregateFunction = iota
    AggregateSum
    AggregateAvg
    AggregateMin
    AggregateMax
    AggregateCount
)

func (af AggregateFunction) String() string {
    switch af {
    case AggregateSum:
        return "sum"
    case AggregateAvg:
        return "avg"
    case AggregateMin:
        return "min"
    case AggregateMax:
        return "max"
    case AggregateCount:
        return "count"
    default:
    }
}
```

GO

```
        return "none"
    }
}

// TimeRange represents an inclusive-exclusive time interval [Start, End).

type TimeRange struct {
    Start time.Time
    End   time.Time
}

// Contains returns true if t is within [Start, End).

func (tr TimeRange) Contains(t time.Time) bool {
    return !t.Before(tr.Start) && t.Before(tr.End)
}

// Overlaps returns true if tr and other have any intersection.

func (tr TimeRange) Overlaps(other TimeRange) bool {
    return !tr.Start.After(other.End) && !other.Start.After(tr.End)
}

// Duration returns End - Start.

func (tr TimeRange) Duration() time.Duration {
    return tr.End.Sub(tr.Start)
}

// Query represents a parsed query ready for planning.

type Query struct {
    Measurement  string
    Tags         map[string]string
}
```

```
TimeRange      TimeRange  
  
Aggregate      AggregateFunction  
  
GroupByWindow time.Duration // 0 means no grouping  
  
Field          string       // For v1, we assume a single field  
  
}
```

D. Core Logic Skeleton Code

1. Query Parser Skeleton ([internal/query/parser.go](#)):

```
package query

import (
    "fmt"
    "strings"
    "time"
    "tempo/internal/models"
)

// Parser holds the state for parsing a query string.

type Parser struct {

    scanner *Scanner // Tokenizer (lexer) you need to implement

    tok     Token    // current token

    lit     string   // current literal

}

// ParseQuery parses the input string and returns a models.Query or an error.

func ParseQuery(input string) (*models.Query, error) {

    p := &Parser{scanner: NewScanner(strings.NewReader(input))}

    p.next() // prime the pump

    return p.parseSelectStatement()

}

// parseSelectStatement parses a SELECT ... FROM ... WHERE ... GROUP BY ... query.

func (p *Parser) parseSelectStatement() (*models.Query, error) {

    query := &models.Query{Tags: make(map[string]string)}

    // Parse SELECT clause

    if err := p.parseSelectClause(query); err != nil {

        return nil, err
    }

    return query, nil
}
```

GO

```
        return nil, err

    }

    // Parse FROM clause

    if err := p.parseFromClause(query); err != nil {

        return nil, err

    }

    // Parse WHERE clause (optional)

    if p.tok == WHERE {

        if err := p.parseWhereClause(query); err != nil {

            return nil, err

        }

    }

    // Parse GROUP BY clause (optional)

    if p.tok == GROUP {

        if err := p.parseGroupByClause(query); err != nil {

            return nil, err

        }

    }

    // Ensure we've consumed all input

    if p.tok != EOF {

        return nil, fmt.Errorf("unexpected token %s, expected end of input", p.tok)

    }

    return query, nil

}
```

```
func (p *Parser) parseSelectClause(q *models.Query) error {

    // TODO 1: Expect SELECT token, consume it

    // TODO 2: Parse aggregate function (e.g., avg(temperature)) or field name

    // TODO 3: Set q.Aggregate and q.Field accordingly

    // TODO 4: If no aggregate, set q.Aggregate = models.AggregateNone

    return nil

}

func (p *Parser) parseFromClause(q *models.Query) error {

    // TODO 1: Expect FROM token, consume it

    // TODO 2: Expect IDENTIFIER token, set q.Measurement

    return nil

}

func (p *Parser) parseWhereClause(q *models.Query) error {

    // TODO 1: Expect WHERE token, consume it

    // TODO 2: Parse one or more tag equality conditions (tag = 'value') joined by AND

    // TODO 3: For each condition, add to q.Tags map

    // TODO 4: Parse time range condition: time >= <start> AND time < <end>

    // TODO 5: Parse absolute timestamps or relative expressions like now() - 1h

    // TODO 6: Create models.TimeRange and assign to q.TimeRange

    return nil

}

func (p *Parser) parseGroupByClause(q *models.Query) error {

    // TODO 1: Expect GROUP BY token, consume it

    // TODO 2: Expect TIME token and '('

    // TODO 3: Parse duration literal (e.g., 1h, 30m)
```

```
// TODO 4: Convert to time.Duration, assign to q.GroupByWindow

return nil

}

func (p *Parser) next() {
    p.tok, p.lit = p.scanner.Scan()
}
```

2. Query Planner Skeleton ([internal/query/planner.go](#)):

```
package query

GO

import (
    "tempo/internal/models"
    "tempo/internal/storage"
)

// QueryPlan is an executable plan for a query.

type QueryPlan struct {

    SeriesKeys []models.SeriesKey

    // For each series key, list of (TSM file path, block index entries)
    FileBlocks map[string][]storage.BlockRef

    Query      models.Query
}

// Planner creates a QueryPlan from a parsed Query and storage engine state.

type Planner struct {

    engine *storage.StorageEngine
}

func (p *Planner) Plan(query *models.Query) (*QueryPlan, error) {

    plan := &QueryPlan{

        Query:      *query,
        SeriesKeys: []models.SeriesKey{},
        FileBlocks: make(map[string][]storage.BlockRef),
    }

    // Step 1: Series Selection

    // TODO 1: Use engine.seriesIndex to find all series with matching measurement
}
```

```

// TODO 2: Filter those series by matching all tags in query.Tags

// TODO 3: Append matching series keys to plan.SeriesKeys

// Step 2: TSM File and Block Selection

for _, seriesKey := range plan.SeriesKeys {

    var blocks []storage.BlockRef

    // TODO 4: For each TSM file in engine.tsmFiles

    // TODO 5: Use tsmFile.IndexForSeries(seriesKey) to get block index entries

    // TODO 6: For each index entry, check if entry.Overlaps(query.TimeRange)

    // TODO 7: If overlapping, add a BlockRef{FilePath, Entry} to blocks

    plan.FileBlocks[seriesKey.String()] = blocks
}

return plan, nil
}

// BlockRef references a specific block in a TSM file.

type BlockRef struct {

    FilePath string

    Entry     storage.IndexEntry
}

```

3. Iterator Interface and Series Scanner ([internal/query/iterator.go](#)):

```
package query
```

GO

```
import "tempo/internal/models"

// Iterator is the core interface for pulling data through the execution pipeline.

type Iterator interface {

    Next() bool           // Advances to the next point, returns false if no more

    At() models.DataPoint // Returns the current point (valid only after Next() == true)

    Err() error           // Returns any error encountered

    Close() error          // Releases resources

}

// SeriesScanner implements Iterator for a single series across multiple TSM files.

type SeriesScanner struct {

    seriesKey models.SeriesKey

    plan      *QueryPlan

    current   models.DataPoint

    // Internal state

    fileIndex int

    blockIter *BlockIterator

    points    []models.DataPoint // decompressed points from current block

    pointIdx  int

}

func NewSeriesScanner(seriesKey models.SeriesKey, plan *QueryPlan) *SeriesScanner {

    return &SeriesScanner{

        seriesKey: seriesKey,

        plan:      plan,

        fileIndex: -1,
```

```
    }

}

func (s *SeriesScanner) Next() bool {
    for {

        // If we have points in the current block buffer, return the next one

        if s.points != nil && s.pointIdx < len(s.points) {

            s.current = s.points[s.pointIdx]

            s.pointIdx++

            // Apply time filter (in case block had points outside range due to min/max
            approximation)

            if s.plan.Query.TimeRange.Contains(s.current.Timestamp) {

                return true

            }

            continue // point outside range, skip to next

        }

        // Need to load the next block

        if s.blockIter == nil || !s.blockIter.Next() {

            // Move to next file or next block list for this series

            s.fileIndex++

            blocks := s.plan.FileBlocks[s.seriesKey.String()]

            if s.fileIndex >= len(blocks) {

                return false // No more files/blocks for this series

            }

            // TODO: Open TSM file reader for blocks[s.fileIndex].FilePath

            // TODO: Create BlockIterator for the specific block index entry

            s.blockIter = NewBlockIterator(reader, blocks[s.fileIndex].Entry)
        }
    }
}
```

```
        continue

    }

    // blockIter.Next() returned true, decompress the block

    // TODO: Call blockIter.At() to get compressed data, decompress into s.points

    s.pointIdx = 0

}

}

func (s *SeriesScanner) At() models.DataPoint { return s.current }

func (s *SeriesScanner) Err() error { return nil } // propagate errors from
blockIter

func (s *SeriesScanner) Close() error {

    if s.blockIter != nil {

        return s.blockIter.Close()

    }

    return nil

}
```

4. Aggregator Skeleton (`internal/query/aggregator.go`):

```
package query

import (
    "math"

    "tempo/internal/models"

    "time"
)

// WindowAggregator aggregates points into fixed-time windows.

type WindowAggregator struct {

    input      Iterator

    window    time.Duration

    aggregate models.AggregateFunction

    // Current window state

    windowStart time.Time

    sum         float64

    count       int64

    min         float64

    max         float64

    // Result to emit

    result models.DataPoint

    ready     bool
}

func NewWindowAggregator(input Iterator, window time.Duration, agg
models.AggregateFunction) *WindowAggregator {
    return &WindowAggregator{
        input:      input,
        window:    window,
```

GO

```
aggregate: agg,  
  
min:      math.Inf(1),  
  
max:      math.Inf(-1),  
  
}  
  
}  
  
func (a *WindowAggregator) Next() bool {  
  
    for a.input.Next() {  
  
        point := a.input.At()  
  
        bucketStart := alignToWindow(point.Timestamp, a.window)  
  
        if a.windowStart.IsTrue() {  
  
            // First point, initialize window  
  
            a.windowStart = bucketStart  
  
        } else if bucketStart != a.windowStart {  
  
            // Point crossed into a new window, emit current aggregate  
  
            a.finalizeWindow()  
  
            a.ready = true  
  
            a.windowStart = bucketStart  
  
            a.resetAccumulators()  
  
            // Add the current point to the new window  
  
            a.addPoint(point)  
  
        }  
  
        return true  
  
    }  
  
    // Add point to current window  
  
    a.addPoint(point)  
  
}  
  
// End of input, emit final window if we have data
```

```
if a.count > 0 {

    a.finalizeWindow()

    a.ready = true

    return true

}

return false

}

func (a *WindowAggregator) At() models.DataPoint {

    if !a.ready {

        panic("At() called without successful Next()")

    }

    a.ready = false

    return a.result

}

func (a *WindowAggregator) addPoint(p models.DataPoint) {

    a.sum += p.Value

    a.count++

    a.min = math.Min(a.min, p.Value)

    a.max = math.Max(a.max, p.Value)

}

func (a *WindowAggregator) resetAccumulators() {

    a.sum = 0

    a.count = 0

    a.min = math.Inf(1)

    a.max = math.Inf(-1)
```

```
}

func (a *WindowAggregator) finalizeWindow() {
    var value float64

    switch a.aggregate {
    case models.AggregateCount:
        value = float64(a.count)

    case models.AggregateSum:
        value = a.sum

    case models.AggregateAvg:
        value = a.sum / float64(a.count)

    case models.AggregateMin:
        value = a.min

    case models.AggregateMax:
        value = a.max

    default:
        value = 0
    }

    a.result = models.DataPoint{Timestamp: a.windowStart, Value: value}
}

// alignToWindow returns the start time of the window containing t.

func alignToWindow(t time.Time, window time.Duration) time.Time {
    epoch := time.Unix(0, 0)

    ns := t.Sub(epoch).Nanoseconds()

    windowNs := window.Nanoseconds()

    bucket := ns / windowNs * windowNs

    return epoch.Add(time.Duration(bucket) * time.Nanosecond)
}
```

```
}
```

E. Language-Specific Hints

- **Memory Mapping for TSM Files:** Use `mmap` via `golang.org/x/exp/mmap` or `syscall.Mmap` for zero-copy reads. Remember to `Munmap` when closing readers.
- **Efficient Iterators:** Implement `Seek` methods on iterators to jump to a specific timestamp using binary search on block index entries.
- **Concurrent Scanning:** For queries spanning many series, you can scan each series in a separate goroutine and merge results using a fan-in pattern. Use `sync.Pool` for recycling `DataPoint` slices.
- **Profiling:** Use `pprof` to identify bottlenecks. Likely hotspots: decompression routines and iterator `Next()` calls.

F. Milestone Checkpoint

To verify your query engine implementation:

1. **Unit Test Parsing:** Run `go test ./internal/query -run TestParser`. You should see successful parsing of various query forms and appropriate errors for malformed queries.
2. **Integration Test End-to-End:**
 - Start a test server with an in-memory storage engine.
 - Write sample data: `curl -X POST http://localhost:8080/write -d "cpu,host=server1 value=42.5 16725312000000000000"` (points for different times and series).
 - Execute a query: `curl "http://localhost:8080/query?q=SELECT avg(value) FROM cpu WHERE host='server1' AND time >= 1672531200s AND time < 1672617600s GROUP BY time(1h)"`.
 - **Expected:** A JSON response containing aggregated values per hour. Verify the averages are mathematically correct.
3. **Performance Smoke Test:** Write 100,000 points for a single series, then query a 24-hour range. The query should return within a few seconds (most time spent in decompression). Use `time` command to measure.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Query returns no data for a valid time range	Block index min/max times incorrect	Print index entries for the series; verify <code>MinTime / MaxTime</code> cover written points.	Ensure <code>TSMWriter</code> correctly computes and writes block min/max.
GROUP BY time() produces misaligned buckets	Bucket alignment using query start time	Log <code>alignToWindow</code> output for sample timestamps.	Change alignment to fixed epoch (Unix).
Aggregated values are wrong (e.g., avg off)	Accumulator reset incorrectly across buckets	Log accumulator state before finalizing each bucket.	Ensure <code>resetAccumulators()</code> is called when window changes.
Query runs out of memory for large ranges	Loading all points before aggregation	Add memory profiling (<code>pprof</code>). Check if <code>SeriesScanner</code> buffers entire series.	Ensure iterator pattern streams points; decompress one block at a time.
Decompression errors during query	Corruption in TSM block or wrong compression algorithm	Write a small tool to read and print raw block bytes, verify checksum.	Check that <code>compressBlock</code> and <code>decompress</code> use same algorithms; verify CRC32.

Retention and Compaction Design

Milestone(s): Milestone 4: Retention & Compaction

The retention and compaction subsystem is TempoDB's housekeeping department—responsible for maintaining data quality, reclaiming storage space, and ensuring the system remains performant as data ages. Without effective lifecycle management, storage costs would balloon, query performance would degrade for historical data, and disk space would eventually be exhausted. This section details how TempoDB automatically manages data from "hot" recent writes to "cold" historical archives through time-based expiration, file consolidation, and pre-computed summarization.

Mental Model: The Warehouse Archivist

Imagine a massive warehouse storing physical records. New boxes arrive constantly on a conveyor belt (the write path), each containing documents from a specific day. An archivist is responsible for:

- Cleaning out old boxes:** Every month, they check expiration dates and shred boxes older than the retention policy (TTL enforcement).
- Consolidating partially-filled boxes:** When multiple boxes contain documents from the same time period but are only half-full, they merge them into fewer, fully-packed boxes to save shelf space (compaction).
- Creating summary catalogs:** For very old records, they don't keep every document but instead create summary binders with daily or weekly totals (downsampling/rollups).

This archivist works in the background, carefully scheduling their work during quiet periods to avoid disrupting people who need to access current records (queries). They maintain an inventory system (metadata) that tracks which boxes are where and what time periods they cover, updating it whenever they reorganize. This mental model captures the essence of TempoDB's retention and compaction: systematic, background reorganization of data to balance accessibility, storage efficiency, and performance.

TTL Enforcement and Compaction Strategy

Time-to-live (TTL) enforcement and compaction are two complementary processes that work together to manage data lifecycle. TTL removes data that's no longer needed, while compaction optimizes the storage of data that remains.

TTL Enforcement: Time-Based Data Expiration

TTL policies specify how long data should be retained before automatic deletion. In TempoDB, TTL is configured per measurement (similar to a table) with a maximum age. The system periodically scans for data blocks that have exceeded their retention period and deletes entire TSM files when all their data is expired.

TTL Enforcement Algorithm:

- Policy Configuration:** Each measurement can have a TTL duration (e.g., 30 days, 1 year, or "infinite" for no expiration). The `Config` struct includes a default TTL applied to measurements without explicit policies.
- Periodic Scanning:** A background goroutine runs at configurable intervals (default: 1 hour) to check for expired data.
- File-Level Deletion:** Since TSM files contain immutable blocks covering specific time ranges, the system checks each file's maximum timestamp. If `file.max_timestamp < (current_time - TTL)`, the entire file is marked for deletion.
- Safe Deletion:** Files aren't immediately removed from disk. Instead, they're moved to a "tombstoned" state in the metadata, excluded from queries, and physically deleted after a grace period (e.g., 5 minutes) to handle any in-flight queries.
- Metadata Cleanup:** After file deletion, the series index is updated to remove references to the deleted blocks, and the block cache evicts any cached data from those files.

Data Structures for TTL Management:

Type Name	Fields	Description
RetentionPolicy	Measurement string Duration time.Duration Default bool	Defines how long data for a specific measurement should be retained
TTLEnforcer	policies map[string]RetentionPolicy interval time.Duration stopCh chan struct{} wg sync.WaitGroup	Manages TTL enforcement across all measurements
TSMFileRef (extended)	Path string MinTime time.Time MaxTime time.Time Measurement string Tombstoned bool TombstoneTime time.Time	Reference to a TSM file with metadata needed for TTL decisions

TTL Enforcement State Machine:

Current State	Event	Next State	Actions
Active	File's max time < (now - TTL)	Tombstoned	Mark file as tombstoned, set tombstone time, exclude from query plans
Tombstoned	Tombstone time > grace period (e.g., 5 min)	Deleting	Schedule physical file deletion, update metadata
Tombstoned	Query references file (late arrival)	Active	Clear tombstone flag, include in queries
Deleting	File successfully deleted	Deleted	Remove from file list, update series index
Deleting	Deletion fails (e.g., permission)	Tombstoned	Log error, retry on next cycle

Design Insight: TTL operates at the file granularity rather than individual points because:

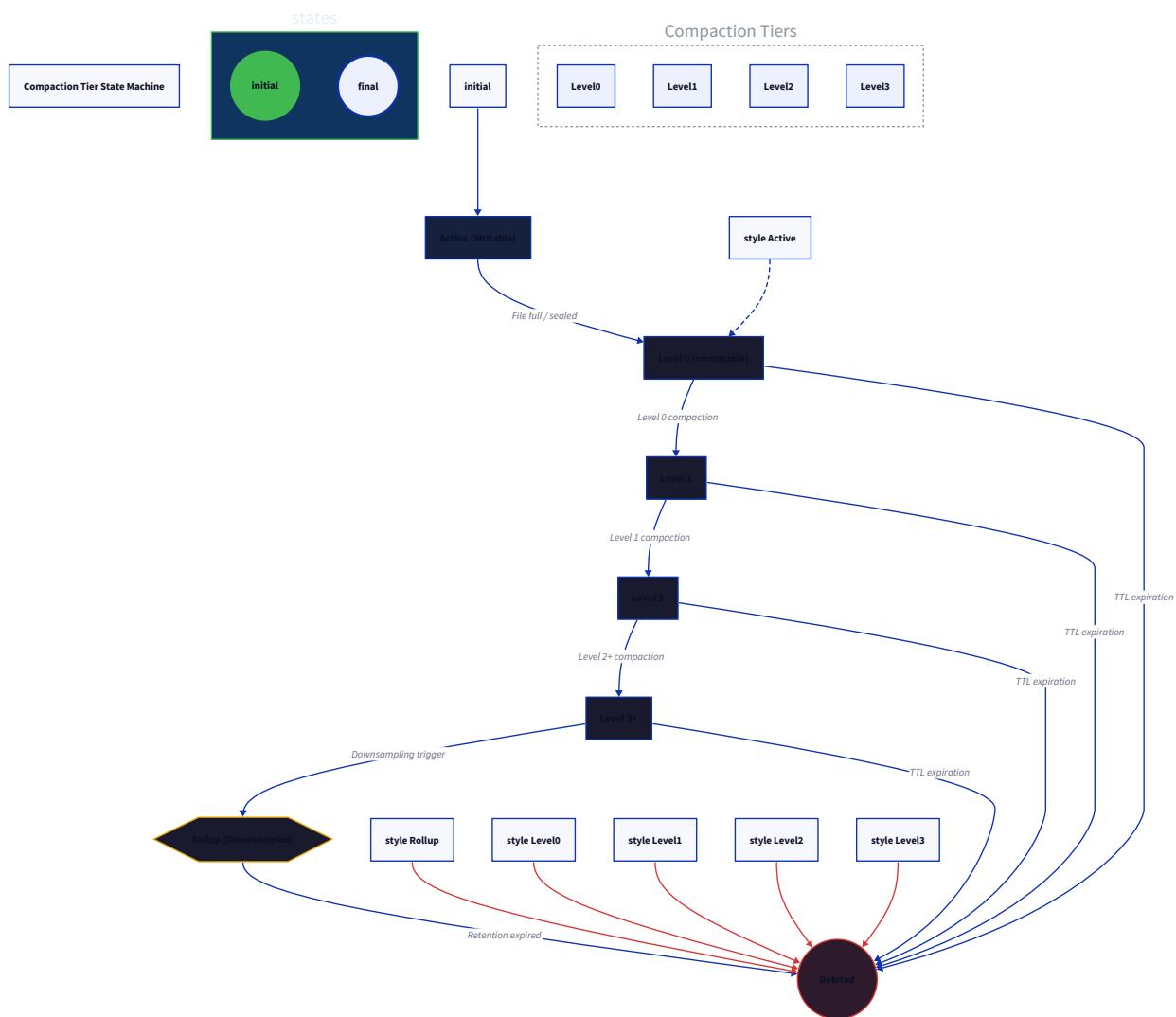
1. TSM files are immutable, making partial deletion within a file complex
2. Time-series data naturally clusters temporally, so entire files typically expire together
3. File deletion is atomic and efficient compared to rewriting files

Compaction Strategy: Level-Based File Consolidation

Compaction merges smaller TSM files into larger, more efficient ones to reduce file count, improve read performance, and reclaim space from deleted or overwritten data. TempoDB uses a level-based compaction strategy inspired by LSM trees but optimized for time-series data.

Compaction Levels:

- **Level 0 (L0):** Freshly flushed memtables (1-2 files per flush). Files may have overlapping time ranges.
- **Level 1 (L1):** Compacted from L0, non-overlapping time ranges within each file, up to 100MB each.
- **Level 2 (L2):** Further compacted from L1, larger files (up to 500MB) covering broader time ranges.
- **Level N (LN):** Continuing hierarchy; higher levels have larger files covering longer time spans.



Compaction Triggers:

1. **Count-based:** When a level accumulates too many files (e.g., L0 > 4 files)
2. **Size-based:** When files in a level exceed size thresholds

3. **Time-based:** Scheduled compaction during low-activity periods

4. **Manual:** Admin-triggered compaction

Compaction Algorithm Steps:

1. **Plan Generation:** The compaction planner examines files in a level, selecting those with overlapping time ranges or small sizes.
2. **Series Grouping:** For each series across selected files, collect all blocks, merging points by timestamp (handling duplicates).
3. **Time Range Splitting:** If the combined data exceeds the target file size, split by time boundaries to create multiple output files.
4. **New File Creation:** Write merged data to new TSM files using the standard `TSMWriter`, applying compression.
5. **Atomic Switch:** Update the metadata to reference new files, mark old files as "compacted."
6. **Cleanup:** Delete old files after confirming new files are successfully written and referenced.

Compaction Data Structures:

Type Name	Fields	Description
<code>CompactionPlan</code>	<code>Level int</code> <code>InputFiles []string</code> <code>OutputFile string</code> <code>TimeRange TimeRange</code> <code>Series []string</code>	Defines which files to compact and where to write output
<code>CompactionStats</code>	<code>Level int</code> <code>FilesIn int</code> <code>FilesOut int</code> <code>BytesIn int64</code> <code>BytesOut int64</code> <code>Duration time.Duration</code>	Tracks compaction performance and effectiveness
<code>CompactionManager</code>	<code>plans chan CompactionPlan</code> <code>stats</code> <code>map[int]CompactionStats</code> <code>mu sync.RWMutex</code> <code>stopCh chan struct{}</code>	Orchestrates compaction across levels

Handling Special Cases During Compaction:

- **Out-of-order points:** Already resolved in memtable or handled during merge by sorting
- **Deleted points (tombstones):** Points marked for deletion are omitted from new files
- **Partial series overlap:** Only merge blocks for series present in input files

- **Compaction failures:** Roll back by deleting partially written output, keeping input files

Critical Consideration: Compaction must not block write operations. TempoDB achieves this by:

1. Performing compaction on immutable, already-persisted TSM files
2. Using copy-on-write semantics—new files are created before old ones are deleted
3. Allowing reads to continue from old files until the atomic metadata switch

Common Pitfalls in Retention and Compaction

⚠ Pitfall: Blocking writes during aggressive compaction

- **Description:** Running compaction with too many files or too frequently can consume I/O bandwidth, causing write latency spikes.
- **Why it's wrong:** Time-series databases must sustain high write throughput; blocking writes defeats their primary purpose.
- **Fix:** Implement compaction throttling based on system load, schedule compaction during off-peak hours, and limit concurrent compaction jobs.

⚠ Pitfall: Losing data during TTL/compaction race conditions

- **Description:** If compaction runs on a file while TTL is deleting it, data may be lost or corruption may occur.
- **Why it's wrong:** Data integrity is paramount; any race condition risking data loss is unacceptable.
- **Fix:** Use a global lock or versioning system for file state changes. Mark files as "in compaction" to prevent concurrent TTL deletion.

⚠ Pitfall: Infinite compaction loops

- **Description:** Poorly configured compaction thresholds cause continuous recompaction of the same data.
- **Why it's wrong:** Wastes CPU and I/O, causes write amplification, and provides no benefit.
- **Fix:** Implement generation tracking—once a file is compacted to a higher level, don't recompact it unless new data arrives for overlapping time ranges.

⚠ Pitfall: Not monitoring disk space during retention

- **Description:** TTL deletes files but doesn't check if disk space is actually reclaimed (some filesystems delay space reclamation).
- **Why it's wrong:** May lead to disk full errors even though TTL is running.
- **Fix:** Monitor actual disk free space, implement emergency compaction if space doesn't recover after deletion, and consider filesystem-specific operations (like `fallocate` or TRIM).

ADR: Continuous vs Scheduled Downsampling

Decision: Continuous Downsampling During Compaction

- **Context:** Historical data (e.g., older than 30 days) rarely needs millisecond precision but is valuable for trend analysis. Storing full-resolution data for all history consumes excessive storage. We need to automatically reduce data resolution over time while preserving statistical properties.
- **Options Considered:**
 1. **Continuous downsampling during compaction:** Generate rollups as part of the standard compaction process when data reaches certain age thresholds.
 2. **Scheduled batch downsampling:** Run separate periodic jobs (e.g., nightly) that scan for data eligible for downsampling and create rollups.
 3. **On-demand downsampling:** Generate rollups only when queried, caching results for future use.
- **Decision:** Option 1—continuous downsampling during compaction.
- **Rationale:**
 - **Efficiency:** Leverages existing data movement during compaction—data is already being read and rewritten, so adding rollup computation adds minimal overhead.
 - **Simplicity:** No separate scheduler, job tracking, or coordination needed.
 - **Predictability:** Rollups are created deterministically as data ages through levels, ensuring consistent availability of downsampled data.
 - **Storage optimization:** Immediate space savings as data is downsampled, rather than waiting for a scheduled job.
- **Consequences:**
 - **Positive:** Simplified architecture, efficient use of compaction I/O, deterministic rollup creation.
 - **Negative:** Compaction becomes more CPU-intensive, rollup granularity is tied to compaction levels, harder to change downsampling policies without recompaction.

Downsampling Options Comparison:

Option	Pros	Cons	Why Not Chosen
Continuous during compaction	<ul style="list-style-type: none"> - No additional I/O - Deterministic timing - Simplified architecture 	<ul style="list-style-type: none"> - Couples compaction and downsampling - Harder to change policies later 	CHOSEN for simplicity and efficiency
Scheduled batch jobs	<ul style="list-style-type: none"> - Flexible scheduling - Independent policy updates - Can run during specific off-hours 	<ul style="list-style-type: none"> - Additional I/O (re-reading data) - Complex job coordination - Delay in space reclamation 	Adds operational complexity
On-demand (lazy)	<ul style="list-style-type: none"> - No upfront computation cost - Adapts to query patterns 	<ul style="list-style-type: none"> - First query pays heavy cost - Unpredictable performance - No storage savings until queried 	Unpredictable performance unacceptable

Continuous Downsampling Implementation:

1. **Policy Definition:** Each measurement can define downsampling rules: e.g., "after 7 days, keep 1-minute averages; after 30 days, keep 5-minute averages; after 1 year, keep 1-hour averages."
2. **Level-to-Granularity Mapping:** Associate compaction levels with downsampling granularities: L0-1 (raw), L2 (1-min), L3 (5-min), L4 (1-hour).
3. **Compaction-Time Rollup:** When compacting files from level N to N+1, apply the appropriate aggregation function (avg, sum, min, max, count) to create downsampled points.
4. **Metadata Tracking:** Store rollup series with special naming convention: `original_series:1min_avg`, `original_series:5min_max`, etc.
5. **Query Routing:** The query engine automatically selects the appropriate rollup series based on the query's time range and granularity requirements.

Downsampling Data Structures:

Type Name	Fields	Description
DownsamplePolicy	Measurement string AfterDuration time.Duration Granularity time.Duration Aggregate AggregateFunction	Defines when and how to downsample data
RollupSeriesKey	OriginalKey SeriesKey Granularity time.Duration Function AggregateFunction	Unique identifier for a rollup series
Downsampler	policies []DownsamplePolicy levelMap map[int]time.Duration	Applies downsampling during compaction

Implementation Guidance (Milestone 4)

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Background Scheduler	Go's <code>time.Ticker</code> with goroutine	Dedicated worker pool with priority queues
File Deletion	<code>os.Remove()</code>	Atomic rename + background delete with retry
Compaction Planning	Simple level-based with fixed thresholds	Cost-based planner analyzing overlap ratios
Downsampling	Fixed aggregation during level transitions	Adaptive sampling based on data variance

B. Recommended File/Module Structure

```
project-root/
  cmd/server/main.go
  internal/
    retention/
      ttl_enforcer.go      # TTL enforcement logic
      ttl_enforcer_test.go
      policies.go          # Retention policy definitions
    compaction/
      manager.go           # Compaction orchestration
      planner.go           # Plan generation
      executor.go          # Plan execution
      stats.go              # Compaction statistics
      levels.go             # Level configuration
      compaction_test.go
    downsampling/
      policy.go            # Downsampling policy definitions
      rollup_writer.go     # Write rollup series
      aggregator.go        # Aggregation during downsampling
    storage/
      tsm/                 # Existing TSM implementation
      block_cache.go
    engine.go              # Updated StorageEngine with compaction hooks
```

C. Infrastructure Starter Code

Background Job Scheduler (Complete Implementation):

GO

```
// internal/retention/scheduler.go

package retention

import (
    "context"
    "sync"
    "time"
)

// Job represents a background task to be executed periodically

type Job struct {

    Name      string
    Interval time.Duration
    Run       func(ctx context.Context) error
    // Optional jitter to spread out jobs
    Jitter   time.Duration
}

// Scheduler manages periodic background jobs

type Scheduler struct {

    jobs    []*Job
    cancel context.CancelFunc
    wg     sync.WaitGroup
    mu     sync.RWMutex
}

// NewScheduler creates a new scheduler

func NewScheduler() *Scheduler {
    return &Scheduler{
```

```
    jobs: make([]Job, 0),  
}  
}  
  
// AddJob registers a job to be run periodically  
  
func (s *Scheduler) AddJob(job Job) {  
  
    s.mu.Lock()  
  
    defer s.mu.Unlock()  
  
    s.jobs = append(s.jobs, job)  
}  
  
// Start begins executing all registered jobs  
  
func (s *Scheduler) Start() {  
  
    ctx, cancel := context.WithCancel(context.Background())  
  
    s.cancel = cancel  
  
  
  
    for _, job := range s.jobs {  
  
        s.wg.Add(1)  
  
        go s.runJob(ctx, job)  
    }  
}  
  
// Stop gracefully shuts down all jobs  
  
func (s *Scheduler) Stop() {  
  
    if s.cancel != nil {  
  
        s.cancel()  
    }  
  
    s.wg.Wait()  
}
```

```
}

// runJob executes a single job periodically

func (s *Scheduler) runJob(ctx context.Context, job *Job) {

    defer s.wg.Done()

    // Initial jitter to spread job starts

    if job.Jitter > 0 {

        select {

            case <-time.After(time.Duration(float64(job.Jitter) * rand.Float64())):

            case <-ctx.Done():

                return

        }
    }

    ticker := time.NewTicker(job.Interval)

    defer ticker.Stop()

    // Run immediately on start

    if err := job.Run(ctx); err != nil {

        log.Printf("Job %s failed: %v", job.Name, err)

    }

    for {

        select {

            case <-ticker.C:

                if err := job.Run(ctx); err != nil {
```

```
    log.Printf("Job %s failed: %v", job.Name, err)

}

case <-ctx.Done():

    return

}

}

}

// Example usage in main.go:

// scheduler := retention.NewScheduler()

// scheduler.AddJob(&retention.Job{

//     Name: "ttl-enforcement",

//     Interval: 1 * time.Hour,

//     Run: ttlEnforcer.Run,

// })

// scheduler.Start()

// defer scheduler.Stop()
```

D. Core Logic Skeleton Code

TTL Enforcement Sweeper:

GO

```
// internal/retention/ttl_enforcer.go

package retention

import (
    "context"
    "path/filepath"
    "sync"
    "time"

    "tempo/internal/storage"
)

// TTLEnforcer periodically removes expired data

type TTLEnforcer struct {

    storage      *storage.Engine

    policies     map[string]time.Duration // measurement -> TTL

    interval     time.Duration

    gracePeriod  time.Duration

    stopCh       chan struct{}`

    wg           sync.WaitGroup

    mu           sync.RWMutex

}

// NewTTLEnforcer creates a new TTL enforcer

func NewTTLEnforcer(storage *storage.Engine, defaultTTL time.Duration) *TTLEnforcer {

    return &TTLEnforcer{
        storage:      storage,
        policies:     make(map[string]time.Duration),
    }
}
```

```
    interval:    1 * time.Hour,
    gracePeriod: 5 * time.Minute,
    stopCh:      make(chan struct{}),
}

}

// SetPolicy sets a TTL policy for a measurement

func (e *TTLEnforcer) SetPolicy(measurement string, ttl time.Duration) {

    e.mu.Lock()

    defer e.mu.Unlock()

    e.policies[measurement] = ttl

}

// Run executes one TTL enforcement cycle

func (e *TTLEnforcer) Run(ctx context.Context) error {

    // TODO 1: Get current time for cutoff calculation

    // TODO 2: Iterate through all TSM files in storage

    // TODO 3: For each file, determine its measurement and applicable TTL

    // TODO 4: Check if file's MaxTime < (now - TTL)

    // TODO 5: If expired, mark file as tombstoned (not deleted yet)

    // TODO 6: Check tombstoned files older than grace period for actual deletion

    // TODO 7: Delete files from disk and update storage metadata

    // TODO 8: Clean up series index entries pointing to deleted files

    // TODO 9: Return statistics (files deleted, bytes freed)

    return nil
}
```

```

// Start begins periodic TTL enforcement

func (e *TTLEnforcer) Start() {
    e.wg.Add(1)

    go func() {
        defer e.wg.Done()

        ticker := time.NewTicker(e.interval)
        defer ticker.Stop()

        for {

            select {

            case <-ticker.C:

                if err := e.Run(context.Background()); err != nil {
                    log.Printf("TTL enforcement failed: %v", err)
                }
            case <-e.stopCh:
                return
            }
        }
    }()
}
}

// Stop gracefully stops TTL enforcement

func (e *TTLEnforcer) Stop() {
    close(e.stopCh)
    e.wg.Wait()
}

```

Compaction Planner:

GO

```
// internal/compaction/planner.go

package compaction

import (
    "sort"
    "time"

    "tempo/internal/storage"
)

// CompactionPlanner decides which files to compact

type CompactionPlanner struct {
    storage     *storage.Engine
    levelConfig map[int]LevelConfig
}

// LevelConfig defines compaction parameters for a level

type LevelConfig struct {
    MaxFiles      int
    MaxSizeBytes int64
    TargetSize    int64
    // Downsampling granularity for this level (if any)
    DownsampleGranularity time.Duration
}

// Plan generates compaction plans for overdue levels

func (p *CompactionPlanner) Plan() ([]CompactionPlan, error) {
    plans := make([]CompactionPlan, 0)
```

```
// TODO 1: Get current TSM files grouped by level from storage

// TODO 2: For each level, check if compaction is needed:

//   - Too many files (count > MaxFiles)?
//   - Files too small (total size < threshold)?
//   - Time-based trigger (oldest file in level > age threshold)?

// TODO 3: For levels needing compaction, select candidate files:

//   - Prefer files with overlapping time ranges
//   - Prefer smaller files first
//   - Exclude files currently being written to

// TODO 4: Group selected files by time range and series overlap

// TODO 5: Create CompactionPlan for each group:

//   - Set input files
//   - Determine output file path
//   - Set target level (current level + 1)
//   - Include downsampling config if moving to downsampling level

// TODO 6: Return list of plans

return plans, nil

}

// ShouldDownsample checks if compaction to next level requires downsampling

func (p *CompactionPlanner) ShouldDownsample(currentLevel, nextLevel int) bool {

    // TODO 1: Check level config for both levels

    // TODO 2: Return true if next level has downsampling granularity configured

    // TODO 3: Handle edge cases (e.g., level doesn't exist)

return false
```

}

Compaction Executor:

GO

```
// internal/compaction/executor.go

package compaction

import (
    "context"
    "os"
    "path/filepath"

    "tempo/internal/storage"
    "tempo/internal/storage/tsm"
)

// Executor runs compaction plans

type Executor struct {
    storage      *storage.Engine
    dataDir      string
    maxConcurrent int
}

// Execute runs a single compaction plan

func (e *Executor) Execute(ctx context.Context, plan CompactionPlan) error {
    // TODO 1: Validate all input files exist and are readable

    // TODO 2: Create temporary output file path

    // TODO 3: For each series in input files:
    //     - Read all points across all input blocks for that series
    //     - Sort points by timestamp (handle duplicates)
    //     - If downsampling required, apply aggregation to create rollup points
    //     - Split points into blocks if exceeding block size limit
}
```

```

//     - Write to TSM writer

// TODO 4: Finalize TSM file (write index, footer)

// TODO 5: Atomically rename temporary file to final location

// TODO 6: Update storage metadata to:
//     - Add reference to new file
//     - Mark old files as "compacted"
//     - Update series index to point to new blocks

// TODO 7: Schedule old files for deletion (after safe period)

// TODO 8: Update compaction statistics

return nil

}

// mergeSeriesPoints combines points for a series across multiple files

func (e *Executor) mergeSeriesPoints(seriesKey string, inputFiles []string
([]storage.DataPoint, error) {

points := make([]storage.DataPoint, 0)

// TODO 1: For each input file:
//     - Open TSM reader
//     - Read all blocks for the series
//     - Decompress blocks to get points
//     - Append to points slice

// TODO 2: Sort all points by timestamp

// TODO 3: Remove duplicate timestamps (keep latest write)

// TODO 4: Return merged, sorted points

return points, nil
}

```

```

}

// applyDownsampling reduces point granularity through aggregation

func (e *Executor) applyDownsampling(points []storage.DataPoint, window time.Duration,
aggFunc storage.AggregateFunction) ([]storage.DataPoint, error) {

    downsampled := make([]storage.DataPoint, 0)

    // TODO 1: If window is zero or aggFunc is AggregateNone, return original points

    // TODO 2: Sort points by timestamp (should already be sorted)

    // TODO 3: Initialize window state:

    //     - currentWindowStart = alignToWindow(points[0].Timestamp, window)

    //     - Initialize aggregator (sum, count, min, max)

    // TODO 4: For each point:

    //     - If point.Timestamp < currentWindowStart + window:
    //         * Add point to current window aggregation

    //     - Else:
    //         * Finalize current window (compute result based on aggFunc)

    //         * Append result to downsampled

    //         * Advance window (handle gaps)

    //         * Reset aggregator for new window

    // TODO 5: Finalize last window

    // TODO 6: Return downsampled points

    return downsampled, nil
}

```

E. Language-Specific Hints

- File Operations:** Use `os.Rename()` for atomic file replacement rather than copy-then-delete. On Windows, you may need to handle "access denied" errors by retrying.

2. **Concurrency Control:** Use `sync.RWMutex` for metadata that's frequently read (by queries) but infrequently written (during compaction). Consider using a versioned metadata system to avoid blocking reads during updates.

3. **Error Handling in Background Jobs:** Always recover from panics in goroutines:

```
defer func() {  
    if r := recover(); r != nil {  
        log.Printf("Compaction panicked: %v", r)  
    }  
}()  
GO
```

4. **Memory Management:** When merging points from multiple large files, stream using iterators rather than loading all points into memory:

```
// Use heap-based merging for large datasets  
  
type pointIterator interface {  
  
    Next() bool  
  
    At() storage.DataPoint  
  
    Close() error  
  
}  
GO
```

5. **Testing:** Use temporary directories for compaction tests and verify both content and metadata after operations:

```
func TestCompaction(t *testing.T) {  
  
    tmpDir := t.TempDir()  
  
    // ... test compaction ...  
  
    // Verify: file count decreased, total points preserved  
  
}  
GO
```

F. Milestone Checkpoint

Verification Command:

```
# Run retention and compaction tests
go test ./internal/retention/... -v -count=1
go test ./internal/compaction/... -v -count=1

# Integration test: Write data, wait for TTL, verify deletion
go run cmd/integration_test/main.go --test-ttl

# Manual verification
curl -X POST "http://localhost:8080/api/v1/compact" # Trigger manual compaction
curl "http://localhost:8080/api/v1/stats" | jq '.compaction' # Check compaction stats
```

Expected Behavior:

1. TTL enforcement should delete files older than the retention period (check disk space reduction).
2. Compaction should merge small files into larger ones (observe decreasing file count in data directory).
3. Downsampling should create rollup series with `:1min_avg` suffix for older data.
4. Queries spanning long time ranges should automatically use downsampled data for older portions.

Debugging Signs:

- **Symptom:** Disk space not freeing up after TTL.
 - **Check:** Files may be locked by open readers. Ensure all `TSMReader` instances are closed.
- **Symptom:** Compaction running continuously without progress.
 - **Check:** Thresholds may be too aggressive. Increase level size thresholds.
- **Symptom:** Queries returning incorrect aggregates after downsampling.
 - **Check:** Verify aggregation logic matches query semantics (e.g., average of averages vs. true average).

Query Language and API Design

Milestone(s): Milestone 5: Query Language & API

The Query Language and API Design defines how external systems and users interact with TempoDB. This is the face of the database—the contracts and interfaces that developers will use daily. A well-designed API must balance expressiveness with performance, provide clear abstractions that match user mental models, and

integrate seamlessly with existing ecosystems. This section specifies TempoDB's query language syntax, HTTP/gRPC APIs, and implementation strategies for parsing and serving requests.

Mental Model: The Restaurant Menu and Kitchen Window

Imagine a restaurant with a detailed menu (the API) that customers use to place orders. The menu categorizes dishes (queries) by type—appetizers (simple range queries), main courses (aggregations), and desserts (downsampling). Each menu item has a precise description (the query language) that the kitchen staff (the database engine) follows to prepare the dish. The kitchen window separates the dining area (client applications) from the cooking area (internal database components), ensuring clean separation of concerns. When a customer orders "SELECT avg(temperature) FROM sensors WHERE time > now() - 1h GROUP BY time(5m)", they're asking for a specific recipe. The waiter (HTTP handler) takes the order to the kitchen, where chefs (query parser, planner, executor) prepare it using ingredients (data blocks) from the pantry (storage engine). The finished dish (query results) is then served on a plate (HTTP/JSON response). This mental model emphasizes that the API should be intuitive and complete (like a good menu), while the query language must be unambiguous and executable (like a kitchen recipe), with clear separation between external interfaces and internal processing.

Query Language Specification

TempoDB implements a SQL-inspired query language specifically designed for time-series patterns. The language supports filtering by time range and tags, aggregating values over windows, and grouping by fixed intervals—the core operations needed for time-series analysis. The design prioritizes clarity for time-series use cases over generic SQL completeness, avoiding the complexity of joins or nested subqueries that are less common in time-series workloads.

Core Grammar

The query language follows a modified BNF grammar with the following productions:

```
Query      = SelectClause FromClause [ WhereClause ] [ GroupByClause ] [ LimitClause ]
SelectClause = "SELECT" ( AggregateFunc "(" Field ")" | Field | "*" )
AggregateFunc = "sum" | "avg" | "min" | "max" | "count" | "first" | "last"
FromClause   = "FROM" Measurement
WhereClause  = "WHERE" Condition ( ( "AND" | "OR" ) Condition )*
Condition    = TimeCondition | TagCondition
TimeCondition = "time" CompareOp ( AbsoluteTime | RelativeTime )
TagCondition  = TagKey CompareOp ( StringLiteral | NumberLiteral )
CompareOp     = ">" | ">=" | "<" | "<=" | "=" | "!="
GroupByClause = "GROUP BY" "time" "(" Duration ")"
LimitClause   = "LIMIT" Integer
```

Key Design Decisions:

- 1. Field vs. Tag References:** Fields (the actual numeric values) can be aggregated and selected directly. Tags are only referenced in WHERE clauses for filtering—they cannot appear in the SELECT clause

because they're metadata, not measured values.

2. **Time Literals:** Supports both absolute timestamps (RFC3339: `2023-10-05T14:30:00Z`) and relative expressions (`now() - 2h`). The `now()` function represents the current server time.
3. **Single Measurement:** Each query targets exactly one measurement, simplifying execution and aligning with time-series organization patterns.
4. **Limited Aggregation Placement:** Aggregate functions can only appear in the SELECT clause, not in WHERE—this prevents ambiguous semantics and matches typical time-series query patterns.

Query Structure and Semantics

Component	Purpose	Example	Notes
SELECT	Specifies which field(s) to return and any aggregation	<code>SELECT avg(temperature),</code> <code>SELECT *</code>	<code>*</code> returns all fields (currently only one supported)
FROM	Identifies the measurement (container) to query	<code>FROM server_metrics</code>	Measurement names are case-sensitive
WHERE	Filters series by tags and time range	<code>WHERE host='web01' AND</code> <code>time > now() - 1h</code>	Multiple conditions combined with AND/OR
GROUP BY <code>time()</code>	Aggregates results into fixed-width time buckets	<code>GROUP BY time(5m)</code>	Creates tumbling windows aligned to epoch
LIMIT	Restricts number of returned points	<code>LIMIT 1000</code>	Applied after aggregation, useful for preview

Example Query Walkthrough: Consider the query `SELECT max(cpu_usage), min(cpu_usage) FROM servers WHERE region='us-east' AND time >= '2023-10-05T00:00:00Z' AND time < '2023-10-05T01:00:00Z' GROUP BY time(5m)`. This requests the maximum and minimum CPU usage for all servers in the us-east region during a specific hour, with results aggregated into 5-minute buckets. The query engine will:

1. Identify all series where the `region` tag equals `'us-east'`
2. Locate data blocks overlapping the specified one-hour time range
3. For each 5-minute window, compute both the maximum and minimum values across all points in that window
4. Return one result point per window containing both aggregated values

Design Insight: The query language intentionally omits features like subqueries, JOINS, and complex expressions in the SELECT clause. Time-series queries overwhelmingly follow simple patterns: filter by time and tags, then aggregate over windows. Supporting only these patterns simplifies implementation while covering >90% of real-world use cases.

Architecture Decision: SQL-Like vs. Flux-Style Query Language

Decision: SQL-Like Syntax with Time Extensions

- **Context:** We need a query language that is immediately familiar to developers (most know SQL) while efficiently expressing time-series operations like time ranges and windowed aggregations. The language must be simple enough to implement within our educational scope yet powerful enough for real use.
- **Options Considered:**
 1. **Full SQL with time extensions** (like TimescaleDB): Supports full SQL with time-specific functions and hypertables.
 2. **SQL-like subset with time primitives** (like InfluxQL): Simplified SQL dialect with first-class time ranges and GROUP BY time().
 3. **Functional pipeline language** (like Flux): Chainable operations (filter, map, aggregate) expressed as function calls.
- **Decision:** Option 2—SQL-like subset with time primitives.
- **Rationale:**
 - **Familiarity:** Developers already understand SELECT-FROM-WHERE patterns, reducing learning curve.
 - **Implementation Simplicity:** Parsing a constrained SQL dialect is easier than building a full SQL parser or a functional language runtime.
 - **Industry Alignment:** InfluxQL and similar dialects have proven effective for time-series queries.
 - **Performance:** Simple structure enables straightforward predicate pushdown and execution planning.
- **Consequences:**
 - **Positive:** Quick adoption, easier parsing/planning, covers common use cases.
 - **Negative:** Cannot express complex multi-measurement correlations or custom transformations; users needing advanced operations must post-process results.

Option	Pros	Cons	Why Not Chosen
Full SQL with extensions	Maximum expressiveness, joins, subqueries	Extremely complex parser/planner, overkill for TS	Scope too large for educational project
SQL-like subset	Familiar, simpler implementation, covers 90% of TS queries	Limited expressiveness for complex transformations	CHOSEN - Best trade-off
Functional pipeline	Extremely flexible, composable, ideal for complex data flows	Steep learning curve, complex execution engine	Better suited for advanced analytics systems

Write and Read APIs

TempoDB exposes two primary interfaces: a write API for data ingestion and a read API for queries. Both use HTTP/REST for simplicity and broad compatibility, with plans for optional gRPC endpoints for high-performance scenarios. The APIs follow industry conventions to ease integration with existing monitoring stacks like Prometheus and Grafana.

Write API

The write API accepts time-series data points in batches using a line protocol format similar to InfluxDB's. This protocol is efficient for network transmission and parsing, supporting thousands of points per request.

Endpoint: POST /api/v1/write

Headers:

- Content-Type: text/plain (for line protocol)
- Optional: Content-Encoding: gzip for compressed payloads

Request Body (Line Protocol Format):

```
<measurement>[,<tag_key>=<tag_value>[,<tag_key>=<tag_value>]] <field_key>=<field_value>[,<field_key>=<field_value>] <timestamp>
```

Each line represents a single data point. Tags are optional but must be sorted by key for canonical series key generation. The timestamp is an integer representing nanoseconds since the Unix epoch (optional; defaults to server time).

Example Request:

```
POST /api/v1/write HTTP/1.1  
Content-Type: text/plain  
  
server,host=web01,region=us-east cpu_usage=42.5,memory_usage=38.2 16965012000000000000  
server,host=web02,region=us-east cpu_usage=35.1 16965012600000000000  
temperature,sensor_id=temp001 value=22.4 16965012000000000000
```

HTTP

Response Codes:

- `204 No Content` : Success, all points written
- `400 Bad Request` : Malformed line protocol or invalid timestamp
- `429 Too Many Requests` : Backpressure applied, client should retry
- `500 Internal Server Error` : Unexpected server error

Design Rationale: The line protocol is text-based for human readability and debuggability, yet compact enough for efficient parsing. It matches industry standards, allowing easy integration with tools like Telegraf. The API accepts batches to amortize write overhead and WAL sync operations.

Read API

The read API accepts queries in the TempoDB query language and returns results in structured JSON format suitable for visualization tools.

Endpoint: `POST /api/v1/query` or `GET /api/v1/query?q=...`

For POST (recommended):

- `Content-Type: application/json`
- Body: `{"query": "SELECT avg(cpu_usage) FROM servers WHERE time > now() - 1h GROUP BY time(5m)"}`

For GET:

- URL-encoded query parameter: `GET /api/v1/query?`
`q=SELECT+avg%28cpu_usage%29+FROM+servers+WHERE+time+%3E+now%28%29+-+1h+GROUP+BY+time%285m%29`

Response Format (JSON):

JSON

```
{
  "results": [
    {
      "series": [
        {
          "name": "servers",
          "columns": ["time", "avg"],
          "values": [
            ["2023-10-05T14:00:00Z", 42.5],
            ["2023-10-05T14:05:00Z", 43.1],
            ...
          ],
          "tags": {"host": "web01", "region": "us-east"}
        }
      ],
      "error": null
    }
  ]
}
```

Pagination Support: For large result sets, the API supports limit/offset via query parameters or continuation tokens to prevent memory exhaustion.

Prometheus Remote Read/Write Compatibility

To integrate with the Prometheus ecosystem, TempoDB implements the Prometheus remote read/write API endpoints. This allows Prometheus to use TempoDB as long-term storage.

Prometheus Write Endpoint: `POST /api/v1/prom/write`

- Accepts Prometheus remote write protocol (snappy-compressed protobuf)
- Converts Prometheus samples to TempoDB data points (metric name → measurement, labels → tags)
- Handles Prometheus's specific timestamp granularity (milliseconds)

Prometheus Read Endpoint: POST /api/v1/prom/read

- Accepts Prometheus remote read requests
- Translates Prometheus matchers and time ranges to TempoDB queries
- Returns results in Prometheus remote read response format

Grafana Data Source Compatibility: TempoDB implements the simple JSON datasource API that Grafana expects, allowing direct querying from Grafana panels. The endpoint /api/v1/grafana/query accepts Grafana's time range and target format, translating to TempoDB queries.

API Architecture Components

Component	Responsibility	Implementation Notes
HTTP Router	Routes requests to appropriate handlers	Use gorilla/mux or chi for flexibility
Write Handler	Parses line protocol, validates points, passes to storage engine	Must handle gzip decompression, batch validation
Query Handler	Parses query string, calls query engine, formats response	Supports both GET and POST, handles timeout cancellation
Prometheus Adapter	Translates Prometheus protobuf to internal format	Uses prompb generated Go code for protocol buffers
Grafana Adapter	Converts Grafana request format to TempoDB queries	Implements /search, /query, /annotations endpoints
Authentication Middleware	(Future) Validates API keys or tokens	Currently placeholder for educational purposes

Common Pitfalls

⚠ Pitfall: Exposing Internal Errors to Clients

- **Description:** Returning raw Go errors (like "panic: nil pointer dereference") in HTTP responses.
- **Why It's Wrong:** Reveals implementation details, security risk, poor user experience.
- **Fix:** Create a layer of well-defined HTTP error responses. Catch panics with middleware, log internal errors server-side, return generic 500 errors to clients.

⚠ Pitfall: Not Validating Time Ranges Early

- **Description:** Allowing queries like SELECT * FROM metrics WHERE time > now() - 1000y that would scan the entire dataset.
- **Why It's Wrong:** Could cause memory exhaustion, disk thrashing, denial of service.
- **Fix:** Implement query validation layer that rejects queries with unbounded or extremely large time ranges. Set configurable maximum time range limits.

⚠ Pitfall: Poor Line Protocol Parsing Performance

- **Description:** Using naive string splitting and conversion for each point in large batches.
- **Why It's Wrong:** CPU becomes bottleneck, limits write throughput.
- **Fix:** Use optimized parsing with minimal allocations: pre-allocate slices, reuse buffers, use byte-level scanning instead of regex.

⚠ Pitfall: Forgetting Content-Type Handling

- **Description:** Only accepting `text/plain` for writes, not supporting `gzip` compression.
- **Why It's Wrong:** Clients may send compressed data or different content types, causing 400 errors.
- **Fix:** Check `Content-Encoding` header, decompress transparently. Be liberal in what you accept (within security bounds).

⚠ Pitfall: Not Implementing Query Cancellation

- **Description:** Long-running queries continue processing even after client disconnects.
- **Why It's Wrong:** Wastes server resources, could lead to resource exhaustion.
- **Fix:** Use request context that cancels when client disconnects. Periodically check `ctx.Done()` during query execution.

Implementation Guidance (Milestone 5)

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Framework	Standard <code>net/http</code> with <code>gorilla/mux</code>	<code>chi</code> router with middleware chain
Query Parsing	Recursive descent parser manually written	ANTLR grammar with generated parser
JSON Marshaling	Standard <code>encoding/json</code>	<code>json-iterator/go</code> for faster performance
Protocol Buffers	<code>gogo/protobuf</code> for Prometheus compatibility	Standard <code>google.golang.org/protobuf</code>
Compression	Standard <code>compress/gzip</code>	Also support <code>snappy</code> for Prometheus
Configuration	Environment variables + YAML config file	Viper for multi-format config support

Recommended File/Module Structure

```
tempo/
├── cmd/
│   └── tempo-server/
│       └── main.go          # Server entry point
├── internal/
│   ├── api/
│   │   ├── handler.go        # HTTP handler registration
│   │   ├── write_handler.go  # Line protocol parsing and write handling
│   │   ├── query_handler.go  # Query API endpoint
│   │   ├── prometheus_handler.go  # Prometheus remote read/write
│   │   ├── grafana_handler.go  # Grafana data source API
│   │   └── middleware.go      # Logging, panic recovery, CORS
│   ├── query/
│   │   ├── parser/
│   │   │   ├── parser.go        # Query language parser
│   │   │   ├── scanner.go       # Token scanner
│   │   │   ├── ast.go           # Abstract syntax tree types
│   │   │   └── parse_test.go    #
│   │   └── executor/          # (From Milestone 3)
│   ├── storage/              # (From Milestone 1 & 2)
│   └── models/                # Shared data structures
└── pkg/
    └── lineprotocol/          # Reusable line protocol parser
        ├── parser.go
        └── parser_test.go
```

Infrastructure Starter Code

Complete Line Protocol Parser (`pkg/lineprotocol/parser.go`):

```
package lineprotocol

import (
    "bytes"
    "errors"
    "strconv"
    "strings"
    "time"
)

var (
    ErrInvalidFormat     = errors.New("invalid line protocol format")
    ErrInvalidFloat      = errors.New("invalid float value")
    ErrInvalidInt        = errors.New("invalid integer value")
    ErrInvalidBoolean    = errors.New("invalid boolean value")
    ErrInvalidTime       = errors.New("invalid timestamp")
)

// Point represents a single data point parsed from line protocol

type Point struct {

    Measurement string
    Tags         map[string]string
    Fields       map[string]interface{}
    Timestamp    time.Time
}

// Parser parses line protocol format

type Parser struct {

    buf []byte
```

GO

```
pos int

}

// NewParser creates a new parser for the given byte slice

func NewParser(data []byte) *Parser {
    return &Parser{buf: data}
}

// Next parses the next point from the buffer

func (p *Parser) Next() (*Point, error) {
    if p.pos >= len(p.buf) {
        return nil, nil // EOF
    }

    // Skip empty lines

    for p.pos < len(p.buf) && (p.buf[p.pos] == '\n' || p.buf[p.pos] == '\r') {
        p.pos++
    }

    if p.pos >= len(p.buf) {
        return nil, nil
    }

    start := p.pos

    // Find end of line

    for p.pos < len(p.buf) && p.buf[p.pos] != '\n' {
        p.pos++
    }

    line := p.buf[start:p.pos]
```

```
p.pos++ // Skip newline

return p.parseLine(line)

}

func (p *Parser) parseLine(line []byte) (*Point, error) {
pt := &Point{
    Tags:    make(map[string]string),
    Fields:  make(map[string]interface{}),
}

// Parse measurement

measEnd := bytes.IndexByte(line, ' ')
if measEnd == -1 {
    measEnd = bytes.IndexByte(line, ',')
    if measEnd == -1 {
        return nil, ErrInvalidFormat
    }
}

pt.Measurement = string(line[:measEnd])
remaining := line[measEnd:]

// Parse tags (optional)

if len(remaining) > 0 && remaining[0] == ',' {
    tagEnd := bytes.IndexByte(remaining, ' ')
    if tagEnd == -1 {
        return nil, ErrInvalidFormat
    }
}
```

```
    }

    tagsStr := remaining[1:tagEnd]

    remaining = remaining[tagEnd:]

}

// Parse comma-separated tags

for _, tag := range bytes.Split(tagsStr, []byte{',', '})) {

    kv := bytes.SplitN(tag, []byte{'='}, 2)

    if len(kv) != 2 {

        return nil, ErrInvalidFormat
    }

    pt.Tags[string(kv[0])] = string(kv[1])
}

}

// Skip space before fields

if len(remaining) == 0 || remaining[0] != ' ' {

    return nil, ErrInvalidFormat
}

remaining = remaining[1:]

}

// Parse fields

fieldEnd := bytes.LastIndexByte(remaining, ' ')

if fieldEnd == -1 {

    // No timestamp

    fieldEnd = len(remaining)
}

fieldsStr := remaining[:fieldEnd]
```

```
remaining = remaining[fieldEnd:]

for _, field := range bytes.Split(fieldsStr, []byte{',', '})) {

    kv := bytes.SplitN(field, []byte{'='}, 2)

    if len(kv) != 2 {

        return nil, ErrInvalidFormat

    }

    key := string(kv[0])

    val := kv[1]

    // Determine field type

    if len(val) == 0 {

        return nil, ErrInvalidFormat

    }

    // Try parsing as float first (most common)

    if f, err := strconv.ParseFloat(string(val), 64); err == nil {

        pt.Fields[key] = f

        continue

    }

    // Try integer

    if i, err := strconv.ParseInt(string(val), 10, 64); err == nil {

        pt.Fields[key] = float64(i)

        continue

    }

}
```

```
// Try boolean

if string(val) == "true" {

    pt.Fields[key] = true

} else if string(val) == "false" {

    pt.Fields[key] = false

} else {

    // String value (must be quoted)

    if len(val) >= 2 && val[0] == '"' && val[len(val)-1] == '"' {

        pt.Fields[key] = string(val[1 : len(val)-1])

    } else {

        return nil, ErrInvalidFormat

    }

}

}

// Parse timestamp (optional)

if len(remaining) > 0 {

    if remaining[0] != ' ' {

        return nil, ErrInvalidFormat

    }

    tsStr := strings.TrimSpace(string(remaining))

    if tsStr != "" {

        // Parse as nanoseconds since epoch

        ns, err := strconv.ParseInt(tsStr, 10, 64)

        if err != nil {

            return nil, ErrInvalidTime

        }

    }

}
```

```
    pt.Timestamp = time.Unix(0, ns)

}

}

if pt.Timestamp.IsZero() {

    pt.Timestamp = time.Now()

}

return pt, nil

}
```

HTTP Server with Middleware (`internal/api/handler.go`):

```
package api
```

GO

```
import (
    "context"
    "encoding/json"
    "fmt"
    "net/http"
    "runtime/debug"
    "time"

    "github.com/gorilla/mux"
    "go.uber.org/zap"
)
```

```
type Server struct {
```

```
    router      *mux.Router
    storage     StorageEngine
    queryEngine QueryEngine
    logger      *zap.Logger
    httpServer  *http.Server
}
```

```
type StorageEngine interface {
```

```
    WritePoint(ctx context.Context, seriesKey models.SeriesKey, point models.DataPoint) error
    WritePointsBatch(ctx context.Context, points []models.DataPoint) error
    Query(ctx context.Context, query models.Query) (QueryResult, error)
}
```

```
type QueryEngine interface {

    ExecuteQuery(ctx context.Context, q string) (*QueryResult, error)
}

func NewServer(storage StorageEngine, queryEngine QueryEngine, logger *zap.Logger) *Server {
    s := &Server{

        router:      mux.NewRouter(),
        storage:     storage,
        queryEngine: queryEngine,
        logger:      logger,
    }

    s.setupRoutes()

    return s
}

func (s *Server) setupRoutes() {
    // Recovery middleware for all routes

    s.router.Use(s.recoveryMiddleware)
    s.router.Use(s.loggingMiddleware)

    // API routes

    s.router.HandleFunc("/api/v1/write", s.handleWrite).Methods("POST")

    s.router.HandleFunc("/api/v1/query", s.handleQuery).Methods("GET", "POST")

    s.router.HandleFunc("/api/v1/prom/write", s.handlePrometheusWrite).Methods("POST")

    s.router.HandleFunc("/api/v1/prom/read", s.handlePrometheusRead).Methods("POST")

    s.router.HandleFunc("/api/v1/grafana/query", s.handleGrafanaQuery).Methods("POST",
    "GET")
}
```

```
s.router.HandleFunc("/api/v1/grafana/search", s.handleGrafanaSearch).Methods("POST",
"GET")

s.router.HandleFunc("/health", s.handleHealthCheck).Methods("GET")

// Static files for admin UI (optional)

s.router.PathPrefix("/admin/").Handler(http.StripPrefix("/admin/",
http.FileServer(http.Dir("./static/admin"))))

}

func (s *Server) recoveryMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if err := recover(); err != nil {
                s.logger.Error("panic recovered",
                    zap.Any("error", err),
                    zap.String("stack", string(debug.Stack())),
                    zap.String("path", r.URL.Path))
                http.Error(w, "Internal server error", http.StatusInternalServerError)
            }
        }()
        next.ServeHTTP(w, r)
    })
}

func (s *Server) loggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        // Wrap response writer to capture status code
    })
}
```

```
    rw := &responseWriter{ResponseWriter: w, statusCode: http.StatusOK}

    next.ServeHTTP(rw, r)

    s.logger.Info("HTTP request",
        zap.String("method", r.Method),
        zap.String("path", r.URL.Path),
        zap.Int("status", rw.statusCode),
        zap.Duration("duration", time.Since(start)),
        zap.String("remote", r.RemoteAddr))

    })

}

type responseWriter struct {

    http.ResponseWriter

    statusCode int
}

func (rw *responseWriter) WriteHeader(code int) {

    rw.statusCode = code

    rw.ResponseWriter.WriteHeader(code)
}

func (s *Server) handleHealthCheck(w http.ResponseWriter, r *http.Request) {

    w.Header().Set("Content-Type", "application/json")

    json.NewEncoder(w).Encode(map[string]string{"status": "ok"})
}

func (s *Server) Start(addr string) error {

    s.httpServer = &http.Server{
```

```
        Addr:           addr,
        Handler:       s.router,
        ReadTimeout:   10 * time.Second,
        WriteTimeout:  30 * time.Second,
        IdleTimeout:   60 * time.Second,
    }

    s.logger.Info("Starting HTTP server", zap.String("addr", addr))

    return s.httpServer.ListenAndServe()
}

func (s *Server) Shutdown(ctx context.Context) error {
    if s.httpServer != nil {
        return s.httpServer.Shutdown(ctx)
    }

    return nil
}
```

Core Logic Skeleton Code

Query Parser (`internal/api/query/parser/parser.go`):

```
package parser
```

GO

```
import (
    "fmt"
    "strconv"
    "strings"
    "time"

    "tempo/internal/models"
)
```

```
type Parser struct {
    scanner *Scanner
    tok     Token
    lit     string
}
```

```
func NewParser(input string) *Parser {
    return &Parser{scanner: NewScanner(input)}
}
```

// ParseQuery parses a query string into a models.Query

```
func (p *Parser) ParseQuery() (*models.Query, error) {
    // Initialize scanner
    p.scan()

    query := &models.Query{
        Tags: make(map[string]string),
    }
```

```
// TODO 1: Parse SELECT clause

//   - Expect "SELECT" token

//   - Parse field name or aggregate function

//   - Handle "*" for all fields

//   - Store field name and aggregate function in query


// TODO 2: Parse FROM clause

//   - Expect "FROM" token

//   - Parse measurement identifier

//   - Store in query.Measurement


// TODO 3: Parse optional WHERE clause

//   - If next token is "WHERE", parse conditions

//   - Parse time conditions: "time > now() - 1h"

//   - Parse tag conditions: "host = 'web01'"

//   - Combine with AND/OR (initially support only AND)

//   - Convert relative time expressions to absolute times

//   - Store time range in query.TimeRange

//   - Store tag filters in query.Tags


// TODO 4: Parse optional GROUP BY clause

//   - If next token is "GROUP", expect "BY", "time", "("

//   - Parse duration like "5m", "1h"

//   - Convert to time.Duration

//   - Store in query.GroupByWindow
```

```
// TODO 5: Parse optional LIMIT clause

//    - If next token is "LIMIT", parse integer

//    - Store in query (may need to add Limit field to models.Query)

// TODO 6: Ensure no extra tokens remain

return query, nil

}

func (p *Parser) scan() {

    p.tok, p.lit = p.scanner.Scan()

}

func (p *Parser) expect(tok Token) error {

    if p.tok != tok {

        return fmt.Errorf("expected %s, got %s", tok, p.tok)

    }

    p.scan()

    return nil

}

// parseDuration parses strings like "5m", "1h", "30s" into time.Duration

func parseDuration(s string) (time.Duration, error) {

    // TODO: Implement duration parsing

    // Handle common units: ns, us, ms, s, m, h, d, w

    // Convert d (day) to 24h, w (week) to 168h

    return 0, nil

}
```

```
// parseTimeExpression parses absolute or relative time expressions

func parseTimeExpression(expr string) (time.Time, error) {

    // TODO: Implement time expression parsing

    // Check if expr starts with "now()" for relative time

    // Otherwise parse as RFC3339 or Unix timestamp

    // For relative: parse "now() - 1h30m"

    return time.Time{}, nil

}
```

Write Handler (`internal/api/write_handler.go`):

```
package api
```

GO

```
import (
```

```
    "compress/gzip"
```

```
    "context"
```

```
    "io"
```

```
    "net/http"
```

```
    "time"
```

```
    "tempo/internal/models"
```

```
    "tempo/pkg/lineprotocol"
```

```
    "go.uber.org/zap"
```

```
)
```

```
func (s *Server) handleWrite(w http.ResponseWriter, r *http.Request) {
```

```
    ctx, cancel := context.WithTimeout(r.Context(), 10*time.Second)
```

```
    defer cancel()
```

```
    // TODO 1: Check Content-Encoding header for gzip
```

```
    //     - If "gzip", wrap r.Body with gzip.Reader
```

```
    //     - Defer close of gzip reader
```

```
    // TODO 2: Create line protocol parser for request body
```

```
    //     - Use lineprotocol.NewReader()
```

```
    //     - Handle large bodies by streaming (parse in chunks)
```

```
    // TODO 3: Parse each point
```

```
    //     - Call parser.Next() until returns nil
```

```
//     - For each point, convert to models.DataPoint and models.SeriesKey

//     - Validate required fields (measurement, at least one field)

//     - Collect points in a batch slice

// TODO 4: Send batch to storage engine

//     - Call s.storage.WritePointsBatch(ctx, points)

//     - Handle errors: 400 for invalid data, 429 for backpressure

//     - Log errors with s.logger

// TODO 5: Return appropriate HTTP response

//     - 204 No Content on success

//     - 400 with error details for client errors

//     - 429 with Retry-After header for backpressure

//     - 500 for internal errors (don't leak details)

// Example success response:

w.WriteHeader(http.StatusNoContent)

}

// parseLineProtocolPoint converts a lineprotocol.Point to internal models

func parseLineProtocolPoint(lpPoint *lineprotocol.Point) (models.SeriesKey,
models.DataPoint, error) {

    // TODO: Implement conversion

    //     - Create SeriesKey from Measurement and Tags

    //     - Extract first numeric field (for now, support single field)

    //     - Create DataPoint with Timestamp and Value

    //     - Return error if no numeric fields found

    return models.SeriesKey{}, models.DataPoint{}, nil
}
```

```
}
```

Query Handler (`internal/api/query_handler.go`):

```
package api
```



```
import (
```



```
    "context"
```



```
    "encoding/json"
```



```
    "net/http"
```



```
    "time"
```



```
    "tempo/internal/query/parser"
```



```
    "go.uber.org/zap"
```

```
)
```

```
type QueryRequest struct {
```

```
    Query string `json:"query"`

}
```

```
type QueryResponse struct {
```

```
    Results []Result `json:"results"`

}
```

```
type Result struct {
```

```
    Series []Series `json:"series,omitempty"`


```

```
    Error string `json:"error,omitempty"`

}
```

```
type Series struct {
```

```
    Name string `json:"name"`


```

```
    Columns []string `json:"columns"`


```

```
    Values [][]interface{} `json:"values"`

}
```

```
GO
```

```
    Tags      map[string]string `json:"tags,omitempty"`

}

func (s *Server) handleQuery(w http.ResponseWriter, r *http.Request) {
    ctx, cancel := context.WithTimeout(r.Context(), 30*time.Second)
    defer cancel()

    var queryStr string

    // TODO 1: Determine request method and extract query
    // - For GET: read from "q" query parameter
    // - For POST: parse JSON body with QueryRequest struct
    // - Return 400 if query is empty

    // TODO 2: Parse query string
    // - Create parser.NewParser(queryStr)
    // - Call ParseQuery() to get models.Query
    // - Handle parse errors with 400 response

    // TODO 3: Validate query
    // - Check time range is not too large (configurable limit)
    // - Ensure measurement exists (optional check)
    // - Return 400 for invalid queries

    // TODO 4: Execute query
    // - Call s.queryEngine.ExecuteQuery(ctx, queryStr)
    // - Or call s.storage.Query(ctx, parsedQuery) directly
```

```
//     - Handle context cancellation/timeout

// TODO 5: Format results

//     - Convert internal QueryResult to API Response format

//     - Handle different result types: raw points, aggregates, grouped

//     - For GROUP BY time(), format buckets correctly

// TODO 6: Write JSON response

//     - Set Content-Type: application/json

//     - Use json.NewEncoder(w).Encode(response)

//     - Handle streaming for large result sets (optional enhancement)

// Example error response:

// w.WriteHeader(http.StatusBadRequest)

// json.NewEncoder(w).Encode(QueryResponse{
//     Results: []Result{{Error: err.Error()}},
// })

}

}
```

Prometheus Write Handler (`internal/api/prometheus_handler.go`):

```
package api
```

GO

```
import (
    "context"
    "io"
    "net/http"

    "github.com/gogo/protobuf/proto"
    "github.com/golang/snappy"
    "github.com/prometheus/prometheus/prompb"
    "go.uber.org/zap"
)
```

```
func (s *Server) handlePrometheusWrite(w http.ResponseWriter, r *http.Request) {
```

```
    // TODO 1: Read and decompress request body
    //     - Use snappy.Decode (Prometheus uses snappy framing format)
    //     - Handle possible decompression errors
```

```
    // TODO 2: Unmarshal protobuf
    //     - Use proto.Unmarshal to get prompb.WriteRequest
    //     - Validate required fields
```

```
    // TODO 3: Convert Prometheus samples to TempoDB points
    //     - For each TimeSeries in request:
    //         - Metric name becomes measurement
    //         - Labels become tags
    //         - Each sample becomes a DataPoint
    //     - Handle different value types (float, histogram, summary - start with float)
```

```
// TODO 4: Write to storage engine

//   - Batch points for efficiency

//   - Use s.storage.WritePointsBatch()

// TODO 5: Return appropriate Prometheus response

//   - 204 on success

//   - 400/500 with error in Prometheus format

}

func (s *Server) handlePrometheusRead(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Read and decompress request (similar to write)

    // TODO 2: Unmarshal prompb.ReadRequest

    // TODO 3: Convert Prometheus matchers to TempoDB query

    //   - Handle equality, regex, and other matcher types

    //   - Build models.Query from matchers and time range

    // TODO 4: Execute query and get results

    // TODO 5: Convert results to prompb.ReadResponse

    //   - Map measurements/tags back to Prometheus metric name/labels

    //   - Format points as prompb.TimeSeries

    // TODO 6: Marshal, compress, and send response

    //   - Use proto.Marshal and snappy encode
```

```
// - Set appropriate Content-Type  
}  
}
```

Language-Specific Hints

1. **HTTP Server Performance:** Use `http.TimeoutHandler` to wrap your router for automatic timeout handling. Set `ReadHeaderTimeout` on your `http.Server` to prevent slowloris attacks.
2. **JSON Marshaling:** For better performance with large result sets, consider using `json.Encoder` directly with `SetEscapeHTML(false)` to avoid unnecessary escaping. For streaming responses, implement `http.Flusher` support.
3. **Context Propagation:** Always pass the request context (`r.Context()`) to downstream operations. This allows proper cancellation when clients disconnect. Use `context.WithTimeout` for operations that should have time limits.
4. **Line Protocol Parsing Optimizations:**
 - Reuse byte buffers with `sync.Pool` to reduce allocations
 - Use `bytes.IndexByte` instead of `strings.Split` for better performance
 - Pre-allocate slices with estimated capacity to avoid reallocations
5. **Query Parser Tips:** Implement the scanner as a state machine reading runes. Use a simple lookup table for keywords. For duration parsing, handle common suffixes: `s`, `m`, `h`, `d`, `w` (week = 7d = 168h).
6. **Prometheus Protocol Buffers:** Use the `gogo/protobuf` version that Prometheus uses for compatibility. Generate Go code with `protoc --gogofast_out=. *.proto`. The `prompb` package from Prometheus can be imported directly if you vendor it.
7. **Graceful Shutdown:** Implement signal handling in `main.go` to catch `SIGTERM` and `SIGINT`, then call `server.Shutdown(ctx)` with a timeout context to allow in-flight requests to complete.

Milestone Checkpoint

After implementing the Query Language and API components, verify functionality with these tests:

Command to Test Write API:

```
# Start the server
go run cmd/tempo-server/main.go

# In another terminal, send test data
curl -X POST http://localhost:8080/api/v1/write \
-H "Content-Type: text/plain" \
--data-raw 'cpu,host=server01,region=us-west usage=42.5 16965012000000000000
cpu,host=server01,region=us-west usage=43.1 16965012600000000000
memory,host=server01 value=2048 16965012000000000000'
```

BASH

Expected Output: HTTP 204 No Content response. Check server logs to confirm points were received and written.

Command to Test Query API:

```
curl -X POST http://localhost:8080/api/v1/query \
-H "Content-Type: application/json" \
-d '{"query": "SELECT avg(usage) FROM cpu WHERE time > now() - 1h GROUP BY time(5m)"'}
```

BASH

Expected Output: JSON response with aggregated results:

```
{
  "results": [
    {
      "series": [
        {
          "name": "cpu",
          "columns": ["time", "avg"],
          "values": [["2023-10-05T14:00:00Z", 42.8]],
          "tags": {"host": "server01", "region": "us-west"}
        }
      ]
    }
  ]
}
```

JSON

Command to Test Prometheus Remote Write Compatibility:

```
# Use a Prometheus test client or create a simple Go program that sends
# Prometheus remote write protocol format
```

BASH

Signs of Problems and Diagnostics:

- **"No data returned" for valid queries:** Check that the query parser correctly extracts time ranges and tag filters. Add debug logging to see the parsed `models.Query` structure.
- **High memory usage during writes:** The line protocol parser may be holding onto large buffers. Ensure you're not accumulating the entire request body in memory.
- **Slow query responses:** Check if the query is doing full scans instead of using time-range predicate pushdown. Verify TSM file indexes are being used correctly.
- **Prometheus writes failing:** Verify snappy decompression is correct. Prometheus uses the "snappy framed" format, not raw snappy.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Query returns HTTP 400 with "parse error"	Malformed query syntax	Check parser error messages, test with simple query first	Add more descriptive error messages, validate grammar
Write API accepts data but nothing appears in queries	Points not being flushed to TSM files	Check memtable flush logs, verify WAL is being replayed on startup	Ensure flush threshold is being reached or implement manual flush endpoint
Prometheus remote write succeeds but data appears with wrong measurement	Incorrect metric name to measurement mapping	Log the conversion from Prometheus TimeSeries to internal points	Check that <code>name</code> label becomes measurement, other labels become tags
GROUP BY time() returns misaligned buckets	Window alignment uses wrong epoch	Test with fixed timestamps, check <code>alignToWindow</code> function	Ensure alignment uses a fixed epoch (e.g., Unix epoch) not query start time
Large queries timeout or crash server	No query limits or memory bounds	Monitor memory usage during query, add query duration logging	Implement query timeout, limit max points returned, use streaming results
Grafana shows "Data source connected but no data"	Grafana query format mismatch	Check Grafana debug panel, log the incoming request from Grafana	Ensure <code>/api/v1/grafana/query</code> returns correct JSON structure with time column first

Interactions and Data Flow

Milestone(s): Milestone 2: Write Path, Milestone 3: Query Engine, Milestone 5: Query Language & API

This section details the operational flow of TempoDB's two most critical user journeys: writing a data point and executing a query. Understanding these sequences illuminates how the previously described components—API, WAL, memtable, storage engine, and query engine—coordinate to fulfill the system's promises of durability and performance. We trace each request from the client's network call through the system's internal machinery to the final response.

Write Path Sequence

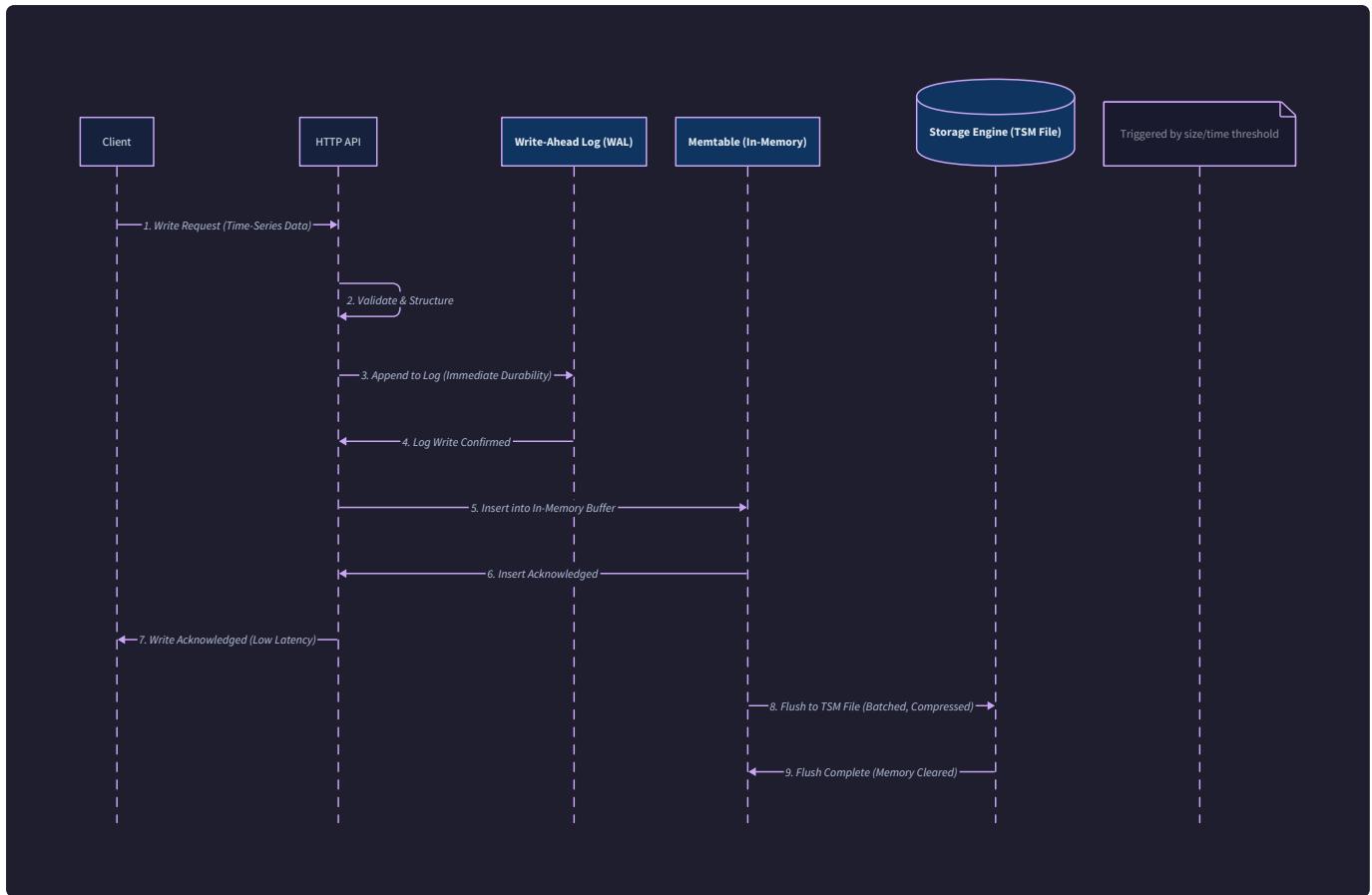
Mental Model: The Airport Check-in and Baggage System Imagine an airport's check-in process for luggage. You (the client) present your bag (data point) at the counter (HTTP API). The agent immediately prints a baggage receipt (WAL entry) and gives you a copy (acknowledgment), guaranteeing the airline now has a record of your bag. Your bag is then placed on a short conveyor belt (memtable) behind the counter, where it waits with other luggage. When the belt fills up, a worker loads all the bags from that belt onto a cart and takes them to the plane's cargo hold (TSM file). This separation—immediate receipt for durability and batched loading for efficiency—is the core of TempoDB's write path.

The write path is designed for high throughput and strong durability. A write is considered successful once it is durable in the Write-Ahead Log (WAL), allowing subsequent processing (memtable insertion and eventual flush) to happen asynchronously. The following sequence details each step.

- 1. Client Request:** An external client sends an HTTP POST request to the `/write` API endpoint. The request body contains one or more data points in InfluxDB line protocol format (e.g.,
`cpu,host=server01 value=0.64 14340555620000000000`).
- 2. API Layer Reception & Parsing:** The `Server`'s `handleWrite` method receives the request. It parses the line protocol using `parseLineProtocolPoint`, which validates the syntax and converts it into the internal `Point` model. This model is then transformed into the canonical `SeriesKey` (from measurement and tags) and `DataPoint` (timestamp and value) structures.
- 3. Durability Guarantee - WAL Append:** Before any in-memory update, the system ensures durability. The `StorageEngine` calls `WriteEntry` on the current active WAL `Segment`. This method synchronously appends a binary representation of the `SeriesKey` and `DataPoint` to the segment file. Depending on the `SegmentConfig.SyncOnWrite` setting, it may call `fsync` to force the data to disk. A successful append guarantees the write can be recovered after a crash. The WAL returns a unique sequence ID for the entry.
- 4. Write Acknowledgment:** Upon successful WAL append, the `StorageEngine` can immediately acknowledge the write to the client via the HTTP API layer. The HTTP server sends a `204 No Content`

response. This acknowledgment is the point of durability; the data is now safe and will eventually be queryable.

5. **In-Memory Buffering - Memtable Insertion:** Concurrently or immediately after the WAL append, the `DataPoint` is inserted into the current active `Memtable`. The `Memtable.Insert` method locates the slice for the corresponding `SeriesKey` (creating it if new) and inserts the `DataPoint` while maintaining sorted order by timestamp. The memtable's approximate size counter is incremented.
6. **Memtable Capacity Check & Rotation:** The `StorageEngine` continuously monitors the size of the active memtable via its `Size()` method. When it exceeds `Config.MaxMemtableSize`, the engine performs a rotation:
 - The current active memtable is marked as **immutable**.
 - A new, empty memtable is created and designated as the active one for new writes.
 - The immutable memtable is placed on a `flushCh` channel for background processing.
7. **Background Flush to TSM:** A dedicated flush goroutine listens on `flushCh`. When it receives an immutable memtable, it calls `FlushMemtable`.
 - The memtable's `Flush()` method is called, returning a map of `SeriesKey` to sorted `[]DataPoint`.
 - For each series, points are grouped into blocks of up to `DefaultMaxPointsPerBlock`.
 - A `TSMWriter` is created for a new TSM file. For each block, `compressBlock` applies delta-of-delta encoding to timestamps and Gorilla XOR compression to values. The writer's `WriteSeries` method writes the compressed blocks and builds an in-memory index.
 - Finally, `TSMWriter.Finish()` writes the index and footer to the file and closes it.
 - The new TSM file is registered with the `StorageEngine` by adding a `TSMFileRef` to the `tsmFiles` list and updating the `seriesIndex` metadata.
8. **WAL Cleanup:** Once the immutable memtable has been successfully persisted to a TSM file, the corresponding segment of the WAL that contains its entries is no longer needed for recovery. The system can safely delete or archive that WAL segment (`Segment.Delete()`), reclaiming disk space.



Data Flow Table: Write Path

Step	Component	Input	Output	Key Action
1	HTTP Client	Line Protocol String	HTTP Request	Initiates write
2	Server.handleWrite	HTTP Request	SeriesKey , DataPoint	Parses and validates
3	Segment.WriteEntry	SeriesKey , DataPoint	WAL Entry ID	Appends to durable log, may <code>fsync</code>
4	Server	-	HTTP 204 Response	Acknowledges write to client
5	Memtable.Insert	SeriesKey , DataPoint	-	Buffers point in sorted in-memory structure
6	StorageEngine	Memtable Size	Immutable Memtable	Checks threshold, rotates memtable
7	TSMWriter	Map of SeriesKey to []DataPoint	.tsm File	Compresses data, writes blocks and index
8	StorageEngine	.tsm File	Updated tsmFiles , seriesIndex	Registers new file for queries

Common Pitfalls in the Write Sequence

⚡ Pitfall: Acknowledging before durable WAL persistence.

- **Description:** Sending the success response to the client after memtable insertion but before the WAL is synced to disk.
- **Why it's wrong:** A server crash after acknowledgment but before the WAL persist would lose the "committed" data, violating durability.
- **Fix:** Ensure the response is sent **only after** `WriteEntry` (and its potential `fsync`) returns successfully.

⚡ Pitfall: Blocking writes during memtable flush.

- **Description:** The system stops accepting new writes while an immutable memtable is being flushed to disk.
- **Why it's wrong:** This creates write stalls and hurts throughput, especially during heavy write loads.
- **Fix:** Use the memtable rotation pattern. Writes continue uninterrupted into the new active memtable while the flush proceeds in the background.

⚡ Pitfall: Not handling WAL segment rotation.

- **Description:** Letting a single WAL segment file grow indefinitely.

- **Why it's wrong:** Extremely large files are difficult to manage, slow to read during recovery, and risk losing more data if corrupted.
- **Fix:** Implement segment rotation based on size (`SegmentConfig.MaxSizeBytes`). Close the current segment and start a new one when the limit is reached.

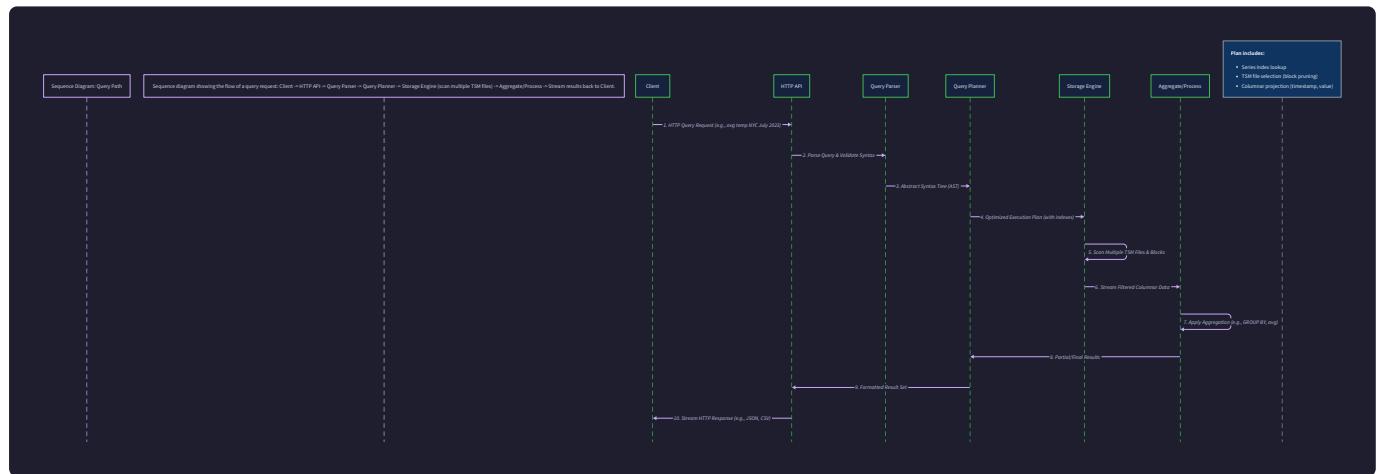
Query Path Sequence

Mental Model: The Library Research Assistant Imagine you ask a research assistant for all books about "quantum physics" published in the 1990s. The assistant doesn't grab every book in the library. First, they consult the card catalog (series index) to find the call numbers for relevant sections. Then, they go to the stacks and pull only the books from those sections published between 1990-1999 (time-range pushdown). They skim each book, photocopying only the relevant chapters (block pruning). Finally, if you asked for a summary, they would compile notes from all the photocopies (aggregation) before handing you the final report. This is the query engine's job: minimize the amount of data physically read and processed.

The query path prioritizes reading efficiency by pushing filters down to the storage layer and streaming results. It transforms a declarative query (what the user wants) into a series of efficient low-level operations.

1. **Client Request:** A client sends an HTTP GET request to the `/query` endpoint. The request includes a `q` parameter with a query string (e.g., `SELECT mean(value) FROM cpu WHERE host='server01' AND time >= now() - 1h GROUP BY time(5m)`).
2. **Query Parsing:** The `Server`'s `handleQuery` method extracts the query string and passes it to `NewParser(input).ParseQuery()`. The parser, typically a recursive descent parser, tokenizes the input and constructs an abstract syntax tree (AST), finally returning a structured `Query` object containing `Measurement`, `Tags`, `TimeRange`, `AggregateFunction`, and `GroupByWindow`.
3. **Query Planning:** The `Query` object is passed to the `Planner.Plan()` method. The planner's job is to convert the logical query into an executable `QueryPlan`. It performs the following key optimizations:
 - **Series Identification:** It consults the `StorageEngine`'s `seriesIndex` to resolve the `Measurement` and `Tags` predicates into a concrete list of `SeriesKey`s.
 - **Predicate Pushdown & File Selection:** For each `SeriesKey`, it examines the list of `TSMFileRef`s. Using the `MinTime` and `MaxTime` in each file's index, it prunes files that do not overlap with the query's `TimeRange`. This yields a `QueryPlan` containing a map from `SeriesKey` to a list of relevant `BlockRef`s (file path and index entry).
 - **Aggregation Strategy:** If the query includes a `GroupByWindow`, the planner wraps the scan iterator with a `WindowAggregator`.
4. **Plan Execution & Streaming:** The `QueryPlan` is executed, typically via an iterator model. An execution goroutine is spawned for the query.
 - For each `SeriesKey` in the plan, a `SeriesScanner` is created. The scanner is initialized with the list of `BlockRef`s.

- The `SeriesScanner.Next()` method drives the scan. It opens the first TSM file (via `OpenTSMReader`), seeks to the block offset from the `IndexEntry`, and reads the compressed block. It then calls `decompressTimestamps` and `decompressValues` to materialize the `[]DataPoint` for that block.
 - Key Optimization:** The scanner only reads blocks whose `[MinTime, MaxTime]` range intersects the query `TimeRange`. Points outside the range are filtered out before being yielded to the iterator pipeline.
 - Points are emitted from the scanner in sorted order (by timestamp).
5. **Aggregation (if applicable):** If the plan includes a `WindowAggregator`, it consumes points from the `SeriesScanner`. The aggregator's `Next()` method buffers points until it has collected all points belonging to the current tumbling window (e.g., a 5-minute bucket). When the window is complete, it applies the `AggregateFunction` (e.g., `mean`) to the buffered points, produces a single output `DataPoint` with the window's start time as its timestamp, and yields it.
6. **Result Streaming to Client:** As result `DataPoint`s are produced by the execution engine, they are serialized (e.g., to JSON or Prometheus format) and streamed directly to the HTTP response body. This avoids buffering the entire result set in memory, supporting queries over large time ranges.
7. **Resource Cleanup:** Once the `SeriesScanner` and any aggregators have exhausted their input (i.e., `Next()` returns `false`), all opened file descriptors (`TSMReader`) are closed, and the query context is finalized.



Data Flow Table: Query Path

Step	Component	Input	Output	Key Action
1	HTTP Client	Query String	HTTP Request	Initiates query
2	Parser.ParseQuery	Query String	Query object	Validates syntax, builds AST
3	Planner.Plan	Query object	QueryPlan	Resolves series, prunes files/blocks, plans aggregation
4	SeriesScanner.Next	BlockRef	DataPoint	Reads & decompresses block, filters by time, yields points
5	WindowAggregator.Next	Stream of DataPoint	Aggregated DataPoint	Buffers points per window, computes aggregate
6	Server.handleQuery	Stream of DataPoint	HTTP Chunked Response	Serializes and streams results
7	SeriesScanner.Close	-	-	Closes TSM file readers

Common Pitfalls in the Query Sequence

⚡ Pitfall: Loading entire blocks into memory before filtering.

- **Description:** The `SeriesScanner` decompresses a full block (e.g., 1024 points) into a slice and then iterates to filter out points outside the `TimeRange`.
- **Why it's wrong:** It wastes CPU and memory on points that will be discarded, especially for sparse queries targeting a small sub-range of a large block.
- **Fix:** Use block-level min/max timestamps for coarse pruning. For finer filtering within a block, consider techniques like seeking within compressed streams, though the simplicity of decompress-then-filter is often acceptable given block sizes are bounded.

⚡ Pitfall: Not pushing the time-range predicate down.

- **Description:** The planner selects TSM files but doesn't use block-level `IndexEntry` metadata to skip irrelevant blocks within those files.
- **Why it's wrong:** This forces the storage engine to read and decompress every block for a series in a selected file, even if only one block contains relevant data, causing significant unnecessary I/O.
- **Fix:** The `QueryPlan` must include specific `BlockRef`s, not just file paths. The `SeriesScanner` must use the `IndexEntry.MinTime/MaxTime` to skip blocks entirely.

⚡ Pitfall: Buffering all results before sending the HTTP response.

- **Description:** The query engine collects all result points in a slice and then serializes the entire slice to JSON at the end.

- **Why it's wrong:** This can exhaust memory for queries returning millions of points and increases client latency (they wait until the very end for the first byte).
- **Fix:** Implement streaming serialization. Write each point (or small batches) to the `http.ResponseWriter` as soon as it's produced, using HTTP chunked encoding.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Query Result Streaming	<code>http.ResponseWriter</code> with manual chunked writes	<code>io.Pipe</code> with a dedicated goroutine for serialization
Query Plan Execution	Synchronous iteration in a single goroutine	Concurrent execution per series using <code>sync.WaitGroup</code> and channels
Point Serialization	JSON using <code>encoding/json.Encoder</code>	Binary format (e.g., Protobuf) or specialized efficient JSON library
WAL Sync Strategy	<code>SyncOnWrite = true</code> for strongest durability	<code>SyncInterval</code> with group commit for higher throughput

B. Recommended File/Module Structure

Add query execution components to the existing structure.

```
tempo/
├── internal/
│   ├── api/
│   │   ├── server.go          # HTTP server, handler functions
│   │   └── middleware.go
│   ├── storage/
│   │   ├── engine.go          # StorageEngine with WritePoint, Query
│   │   ├── wal/                # WAL implementation
│   │   ├── memtable.go
│   │   └── tsm/                # TSM reader/writer
│   ├── query/
│   │   ├── parser/
│   │   │   ├── parser.go
│   │   │   └── ast.go
│   │   ├── planner.go          # Converts Query to QueryPlan
│   │   ├── executor.go         # Coordinates execution of QueryPlan
│   │   ├── scanner.go          # SeriesScanner implementation
│   │   └── aggregator.go       # WindowAggregator implementation
│   └── models/                # Shared data types (DataPoint, SeriesKey, etc.)
└── cmd/
    └── tempo-server/
        └── main.go
```

C. Infrastructure Starter Code

Complete HTTP Streaming Response Helper (to be placed in `internal/api/response.go`):

```
package api

import (
    "encoding/json"

    "net/http"

    "time"
)

// StreamWriter handles streaming query results to an HTTP response.

type StreamWriter struct {

    w          http.ResponseWriter
    encoder    *json.Encoder
    flushed    bool
}

// NewStreamWriter creates a new StreamWriter for the given ResponseWriter.
// It sets appropriate headers for streaming JSON.

func NewStreamWriter(w http.ResponseWriter) *StreamWriter {
    w.Header().Set("Content-Type", "application/json")
    w.Header().Set("Transfer-Encoding", "chunked")
    // Write the opening bracket for a JSON array
    w.Write([]byte("["))
    return &StreamWriter{
        w:          w,
        encoder:    json.NewEncoder(w),
        flushed:    false,
    }
}
```

GO

```
// WritePoint streams a single data point as JSON.

// It adds commas between array elements.

func (sw *StreamWriter) WritePoint(p models.DataPoint) error {

    if sw.flushed {

        sw.w.Write([]byte(","))
    } else {

        sw.flushed = true
    }

    return sw.encoder.Encode(p)
}

// Close finalizes the JSON array and flushes any remaining data.

func (sw *StreamWriter) Close() error {

    sw.w.Write([]byte("]"))

    if f, ok := sw.w.(http.Flusher); ok {

        f.Flush()
    }

    return nil
}

// Error writes an error response and closes the stream.

func (sw *StreamWriter) Error(err error) {

    // Cancel the opening bracket if no data was written

    if !sw.flushed {

        sw.w.Write([]byte("["))
    } else {

        sw.w.Write([]byte("]"))
    }
}
```

```
    http.Error(sw.w, err.Error(), http.StatusInternalServerError)

}
```

D. Core Logic Skeleton Code

Skeleton for the `StorageEngine.Query` method orchestrating the sequence:

```
// Query executes a read query against the storage engine.          GO

// It orchestrates parsing, planning, execution, and streaming.

func (e *StorageEngine) Query(ctx context.Context, queryStr string) (query.Result, error) {

    // TODO 1: Parse the query string into a models.Query object.

    // Hint: Use query.NewParser(queryStr).ParseQuery()

    // TODO 2: Create a query plan from the parsed Query object.

    // Hint: Use planner.Plan(queryObj). This step resolves series keys and selects TSM
blocks.

    // TODO 3: Initialize a result streaming structure (e.g., a channel of DataPoint).

    // TODO 4: Launch a goroutine to execute the query plan.

    // In the goroutine:

    //     - Iterate over each SeriesKey in the plan.

    //     - For each, create a SeriesScanner for its list of BlockRefs.

    //     - Iterate with scanner.Next(), sending points to the result channel.

    //     - If the query has aggregation, wrap the scanner with a WindowAggregator.

    //     - Close the result channel when done.

    // TODO 5: Return a structure that allows the caller to read from the result stream.

    // This enables the HTTP handler to stream results as they are produced.

}
```

Skeleton for the `SeriesScanner.Next` method implementing block pruning:

GO

```
// Next advances the scanner to the next data point.

// It returns false when there are no more points or an error occurs.

func (s *SeriesScanner) Next() bool {

    for {

        // TODO 1: If we have points in the current buffer (s.points) and haven't reached
        // the end (s.pointIdx < len(s.points)), advance s.pointIdx and return true.

        // TODO 2: If we have no more blocks to read for the current file, move to the next
        // file in the plan.

        //     - Close the current TSMReader if open.

        //     - Open the next TSM file using storage.OpenTSMReader(path).

        //     - Reset the block iterator for the new file.

        // TODO 3: Get the next BlockRef for the current series from the iterator.

        //     - Check if the block's [MinTime, MaxTime] overlaps the query TimeRange. If
        //       not, skip it.

        //     - Read the compressed block using TSMReader.ReadBlock at the given offset.

        //     - Decompress the block into []DataPoint.

        // TODO 4: Filter the decompressed points: keep only those where
        // TimeRange.Contains(point.Timestamp) is true.

        //     - Assign the filtered slice to s.points and reset s.pointIdx to 0.

        // TODO 5: If after filtering a block we have zero points, loop back to step 2 to
        // try the next block.

    }

}
```

Skeleton for the `Planner.Plan` method implementing predicate pushdown:

GO

```
// Plan creates an execution plan from a parsed query.

func (p *Planner) Plan(q *models.Query) (*QueryPlan, error) {
    plan := &QueryPlan{
        Query:      *q,
        SeriesKeys: []models.SeriesKey{},
        FileBlocks: make(map[string][]storage.BlockRef),
    }

    // TODO 1: Resolve series keys from the measurement and tag filters.
    // - Access the storage engine's seriesIndex.
    // - Find all SeriesKey where Measurement matches and Tags satisfy the filter predicates.
    // - Add matching keys to plan.SeriesKeys.

    // TODO 2: For each resolved SeriesKey, identify relevant TSM blocks.
    // - Get the list of TSMFileRef from the storage engine.
    // - For each file, check if its global [MinTime, MaxTime] overlaps the query TimeRange.
    // - If yes, load the TSMIndex for that file (or use a cached summary).
    // - Find the IndexEntry for this SeriesKey in the file's index.
    // - For each IndexEntry, check if its block-level [MinTime, MaxTime] overlaps the query TimeRange.
    // - If yes, create a BlockRef{FilePath: file.Path, Entry: entry} and append it to plan.FileBlocks[seriesKey.String()].
    // TODO 3: Sort the BlockRefs for each series by MinTime (ascending) for efficient scanning.

    // TODO 4: If the query has a GroupByWindow, note that an aggregator will be needed during execution.

    return plan, nil
}
```

```
}
```

E. Language-Specific Hints

- **Streaming HTTP:** Use `http.Flusher` to periodically flush buffered data to the client if you write in chunks. This can reduce perceived latency.
- **Concurrent Scanning:** Use a `sync.WaitGroup` to wait for multiple `SeriesScanner` goroutines (one per series) to finish. Collect their results via channels and merge them, ensuring timestamp order is maintained if required.
- **Resource Management:** Use `defer reader.Close()` in the `SeriesScanner` to ensure TSM file descriptors are closed even if an error occurs during iteration.
- **Context Propagation:** Pass the `context.Context` from the HTTP request through to the query execution goroutine. This allows the query to be cancelled if the client disconnects, preventing unnecessary work.

Error Handling and Edge Cases

Milestone(s): This section addresses error handling and edge cases that are critical for the robustness of all five milestones, particularly Milestone 2 (Write Path), Milestone 3 (Query Engine), and Milestone 4 (Retention & Compaction).

Time-series databases operate in challenging environments: they must handle high-velocity data streams, run continuously for years, and process queries over massive datasets while maintaining correctness. Unlike batch processing systems, they cannot stop for repairs or data correction. This section documents the anticipated failure modes and data anomalies that TempoDB must handle gracefully, along with the recovery strategies and design patterns that ensure system resilience.

Common Failure Modes and Recovery

Mental Model: The Emergency Response Team

Think of error handling as an emergency response team for your database. Each component has monitoring sensors (health checks), standard operating procedures for common incidents (disk full, corrupt files), and escalation protocols for complex failures. The goal isn't to prevent all failures—which is impossible—but to detect them early, contain the damage, recover automatically when possible, and provide clear diagnostics when manual intervention is needed.

Storage Layer Failures

The storage layer faces the most severe failure modes since it interacts directly with persistent storage, which can fail in unpredictable ways.

Failure Mode	Detection Mechanism	Recovery Strategy	Impact on Operations
Disk full during write	<code>os.Write</code> returns <code>ErrNoSpace</code> ; filesystem operations fail with <code>ENOSPC</code>	<ol style="list-style-type: none"> Immediately stop accepting new writes Mark storage engine as read-only Notify monitoring system Optionally trigger emergency compaction to free space 	Writes blocked; reads continue; requires administrator intervention to free disk space
Corrupt WAL segment	CRC32 checksum mismatch during <code>Segment.Scan()</code> ; invalid entry format parsing fails	<ol style="list-style-type: none"> Log corruption location and skip remaining entries in segment Mark segment as corrupt in WAL metadata Continue with next segment If corruption in active segment, restart with new segment 	Potential data loss for unflushed writes in corrupt segment; system continues with available data
Corrupt TSM file header	<code>ReadHeader()</code> fails magic number validation or version check	<ol style="list-style-type: none"> Skip file during query execution Move file to quarantine directory Log detailed error with file path If file was critical for queries, return partial results with error 	Queries return partial data; requires manual investigation of quarantined file
Memory-mapping failure	<code>mmap</code> system call returns error during <code>OpenTSMReader()</code>	<ol style="list-style-type: none"> Fall back to regular file I/O for that file Log performance degradation warning Continue 	Reduced read performance for affected file; system remains functional

Failure Mode	Detection Mechanism	Recovery Strategy	Impact on Operations
		operations with slower access path	
Block decompression failure	<code>decompressTimestamps()</code> or <code>decompressValues()</code> returns error; checksum mismatch	1. Skip the corrupt block within the TSM file 2. Increment corruption counter for monitoring 3. Return available data from other blocks 4. Schedule file for recompaction if corruption rate exceeds threshold	Partial data loss within affected time range; queries complete with gaps

ADR: Recovery Strategy for Corrupt TSM Files

Decision: Quarantine and Continue for Corrupt TSM Files

- **Context:** TSM files represent immutable, compacted data blocks. A corrupt TSM file could affect query results, but stopping the database for repair is unacceptable in production scenarios.
- **Options Considered:**
 1. **Panic and shutdown:** Immediately stop the database to prevent further corruption
 2. **Automatic repair:** Attempt to rebuild the file from WAL and other TSM files
 3. **Quarantine and continue:** Skip the corrupt file and continue operations
- **Decision:** Implement quarantine and continue with fallback to regular I/O for memory-mapping failures
- **Rationale:** Time-series databases prioritize availability over perfect consistency for historical data. The quarantine approach: (1) maintains system availability, (2) provides clear audit trail of corrupt files, (3) allows administrators to investigate during maintenance windows, and (4) aligns with industry practice (InfluxDB uses similar approach)
- **Consequences:** Queries may return incomplete results; monitoring must alert on quarantined files; administrators must manually restore from backup if needed

Option	Pros	Cons	Chosen?
Panic and shutdown	Prevents further corruption; forces immediate repair	Causes downtime; unacceptable for production	✗
Automatic repair	Maintains data completeness; transparent to users	Complex to implement; may fail or cause prolonged unavailability	✗
Quarantine and continue	Maintains availability; simple to implement	Data gaps in queries; requires monitoring and manual intervention	✓

Write Path Failures

The write path must maintain durability guarantees while handling various failure scenarios.

Failure Mode	Detection Mechanism	Recovery Strategy	Impact on Operations
WAL write failure	<code>WriteEntry()</code> returns I/O error; <code>fsync</code> fails	<ol style="list-style-type: none"> 1. Retry with exponential backoff (up to 3 attempts) 2. If persists, rotate to new WAL segment 3. If rotation fails, enter read-only mode 4. Log critical error with disk health metrics 	Temporary write blocking during retry; potential data loss if retries fail
Memtable flush failure	<code>FlushMemtable()</code> returns error during TSM file creation	<ol style="list-style-type: none"> 1. Retry flush with same data 2. If retry fails, write memtable contents to emergency spill file 3. Continue with new memtable 4. Schedule spill file for processing during next compaction 	Increased disk usage from spill files; compaction handles recovery
Out-of-memory during ingestion	Memory allocation fails; <code>Memtable.Size()</code> exceeds safe threshold	<ol style="list-style-type: none"> 1. Trigger emergency flush of current memtable 2. Apply backpressure to write API (HTTP 503) 3. Reduce batch sizes for incoming writes 4. Log memory pressure warnings 	Temporary write throttling; potential increased latency
Clock skew in distributed writes	Incoming timestamp significantly ahead of system clock	<ol style="list-style-type: none"> 1. Reject writes with timestamps > (now + <code>maxClockSkew</code>) 2. Log warning with client identification 3. Optionally buffer for <code>maxClockSkew</code> duration 	Rejected writes; clients must correct their clock
Series cardinality explosion	Monitoring detects exponential growth in <code>seriesIndex</code> size	<ol style="list-style-type: none"> 1. Log cardinality warning with top contributing tags 2. Continue processing (cardinality is operational concern) 	Increased memory usage; potential performance degradation

Failure Mode	Detection Mechanism	Recovery Strategy	Impact on Operations
		3. Expose cardinality metrics via monitoring API	

Step-by-Step WAL Recovery Procedure:

When TempoDB starts, it must recover any unflushed data from the Write-Ahead Log:

1. **Locate WAL segments:** Scan the WAL directory for files with `.wal` extension, sorted by sequence number
2. **Identify recovery range:** Determine the first WAL segment that contains data not yet persisted to TSM files (using the WAL checkpoint file if available)
3. **Sequential scan:** For each segment in recovery range, call `Segment.Scan()` to read all entries
4. **Reconstruct memtables:** For each WAL entry, parse the series key and data point, then insert into an in-memory recovery memtable
5. **Batch flush:** When recovery memtable reaches configured size threshold, flush it to TSM files
6. **Cleanup:** Once all segments are processed, delete recovered WAL segments (or move them to archive)
7. **Resume normal operations:** Start accepting new writes with fresh WAL segment

Query Engine Failures

Query failures must not crash the database and should provide helpful error messages to users.

Failure Mode	Detection Mechanism	Recovery Strategy	Impact on Operations
Query timeout	Context deadline exceeded during <code>Query()</code> execution	<ol style="list-style-type: none"> 1. Cancel query execution tree 2. Release all allocated resources (iterators, buffers) 3. Return timeout error to client 4. Log slow query with execution plan 	Failed query; other queries continue unaffected
Memory exhaustion during aggregation	Memory allocation fails in <code>WindowAggregator</code>	<ol style="list-style-type: none"> 1. Cancel query with "out of memory" error 2. Implement result streaming to limit memory usage 3. Add memory limits to query execution configuration 	Failed query; system remains operational
Invalid query syntax	<code>Parser.ParseQuery()</code> returns syntax error	<ol style="list-style-type: none"> 1. Return detailed error with line and position 2. Suggest corrections for common mistakes 3. Log malformed query attempts (rate-limited) 	Query rejected; client must correct syntax
Missing measurement or field	Storage engine returns no data for specified measurement	<ol style="list-style-type: none"> 1. Return empty result set (not an error) 2. Log warning if measurement was recently written to (potential data loss indicator) 	Empty query results; system continues normally
Range query too large	Query time range exceeds configured <code>maxQueryRange</code>	<ol style="list-style-type: none"> 1. Reject query with "time range too large" error 2. Suggest using downsampling or smaller ranges 3. Log large query attempt for monitoring 	Query rejected; client must modify parameters

Compaction and Retention Failures

Background maintenance tasks must handle failures gracefully without data loss.

Failure Mode	Detection Mechanism	Recovery Strategy	Impact on Operations
Compaction interrupted by crash	Incomplete TSM output file; missing footer	<ol style="list-style-type: none"> On restart, detect incomplete output files (missing magic footer) Delete incomplete output files Retry compaction during next cycle Preserve all input files until successful completion 	Temporary storage bloat; automatic retry on next cycle
Disk full during compaction	Write failure during <code>Execute()</code>	<ol style="list-style-type: none"> Delete partially written output file Abort compaction plan Trigger emergency disk space alert Skip further compactations until space available 	Compaction paused; queries continue; requires admin intervention
TTL deletion of active file	Race condition: file compacted after TTL marked it for deletion	<ol style="list-style-type: none"> Use tombstone markers instead of immediate deletion Check file reference count before physical deletion Implement grace period for tombstoned files 	No data loss; slight storage overhead during grace period
Downsampling arithmetic overflow	Aggregate sum exceeds float64 precision during <code>applyDownsampling()</code>	<ol style="list-style-type: none"> Use Kahan summation algorithm for improved precision Log precision warnings for large aggregations Return <code>Nan</code> for mathematically invalid operations 	Potential precision loss in downsampled values; system continues

Data-Specific Edge Cases

Mental Model: The Time Traveler's Journal

Imagine time-series data as entries in a time traveler's journal. Sometimes entries arrive out of order (the traveler jumps around), sometimes there are gaps (the traveler skips days), and sometimes the dates are ambiguous (different calendar systems). The database must make sense of this chaotic journal while presenting a coherent timeline to readers.

Out-of-Order Data Handling

Time-series data frequently arrives out of chronological order due to network latency, clock synchronization issues, or batch processing delays.

Tolerance Window Strategy: TempoDB implements a configurable **tolerance window**

(`Memtable.maxOutOfOrderWindow`) that determines how far back in time a late-arriving point can be inserted into the current memtable. The default is typically 1-5 minutes for metrics collection and up to 1 hour for IoT scenarios.

Processing Logic for Out-of-Order Writes:

1. **Timestamp validation:** When `WritePoint()` receives a point, it compares the point's timestamp with the current system time (or ingestion time)
2. **Within tolerance window:** If `point.Timestamp >= (now - maxOutOfOrderWindow)`, the point is inserted into the active memtable in correct sorted position
3. **Beyond tolerance window but recent:** If point is older than tolerance window but newer than the oldest point in active memtable, it's still inserted with a warning log
4. **Very old data:** If point is older than any point in current memtable but newer than the newest TSM file, a special "late write" handler creates a small TSM file that will be merged during next compaction
5. **Extremely old data:** Points older than the oldest TSM file are rejected with an error (configurable behavior)

Example Walkthrough: Consider a system with `maxOutOfOrderWindow = 5m`, current time = 10:00, and memtable containing points from 09:55 to 09:59:

- Point at 09:57 (3 minutes old) → Inserted into memtable (within window)
- Point at 09:52 (8 minutes old) → Inserted with warning (beyond window but recent)
- Point at 09:30 (30 minutes old) → Written to late-write TSM file
- Point at 08:00 (2 hours old) → Rejected with "timestamp too old" error

Gaps in Time Series

Real-world time-series data often has gaps—periods where no data was collected or transmitted.

Query Behavior with Gaps:

- **Raw queries:** Return only existing data points, resulting in discontinuous results
- **Aggregation queries:** Treat gaps as missing values (not zeroes):
 - `SUM` , `COUNT` : Only aggregate existing points
 - `AVG` : Divide by number of existing points, not total time range
 - Time bucket with no data: Return no point for that bucket (not a zero/null point)

Interpolation Considerations: While some time-series databases offer interpolation (filling gaps with estimated values), TempoDB explicitly does NOT interpolate because:

1. Interpolation assumes a data generation model that may not be valid
2. It can mask genuine data collection failures
3. It complicates aggregation semantics
4. Applications can implement their own interpolation if needed

Clock Skew and Timestamp Anomalies

Distributed systems inevitably have clock differences between data producers.

Handling Strategies:

1. **Future timestamp rejection:** Points with timestamps $> (\text{current time} + \text{maxClockSkew})$ are rejected
2. **Ingestion time tracking:** Optionally record ingestion timestamp alongside data timestamp for diagnostics
3. **NTP enforcement:** Documentation strongly recommends clients use NTP synchronization
4. **Batch timestamp normalization:** For batch imports, allow overriding timestamps with ingestion time

Clock Skew Detection Table:

Scenario	Detection	Action
Client clock ahead by seconds	Timestamp $> (\text{now} + 10\text{s})$	Reject with "timestamp in future" error
Client clock ahead by minutes/hours	Timestamp $> (\text{now} + 1\text{h})$	Reject with "clock skew too large" error
Client clock slightly behind	Timestamp $< \text{now}$ but within tolerance	Accept normally
Client clock far behind	Timestamp $< (\text{oldest TSM file time})$	Reject with "timestamp too old" error

Queries with Very Large Time Ranges

Unbounded or extremely large time range queries can overwhelm the system.

Protection Mechanisms:

1. **Configuration limits:** `maxQueryRange` (default 30 days) rejects overly broad queries

2. **Memory limits:** Query execution tracks memory usage and cancels if exceeding limit
3. **Result streaming:** Large results stream incrementally rather than buffering entirely
4. **Timeout enforcement:** All queries have execution timeout (default 30 seconds)
5. **Resource prioritization:** Short recent queries prioritized over long historical scans

Progressive Optimization for Large Queries: When a query approaches but doesn't exceed the maximum range:

1. **Downsampling suggestion:** Query planner checks if downsampled data exists for the time range
2. **Storage tier routing:** Direct query to appropriate storage tier (hot/warm/cold)
3. **Parallel execution:** Split time range into chunks processed in parallel
4. **Approximate results:** Optionally return approximate aggregates using statistical sketches

Boundary Conditions in Time-Based Grouping

`GROUP BY time()` queries have subtle edge cases at time window boundaries.

Window Alignment Rules: The `alignToWindow()` function aligns timestamps to consistent window boundaries using a fixed epoch (January 1, 1970, 00:00:00 UTC). This ensures deterministic grouping regardless of query start time.

Boundary Scenarios:

1. **Partial first window:** If query starts mid-window, first bucket includes only data from start time to next boundary
2. **Partial last window:** If query ends mid-window, last bucket includes only data from last boundary to end time
3. **Empty windows:** Windows with no data produce no output point (not a zero/null point)
4. **Timezone considerations:** All timestamps are UTC internally; timezone conversion happens at API layer if needed

Example with Daylight Saving Time: Consider `GROUP BY time(1h)` on March 10, 2024 (daylight saving transition in US/Eastern):

- 01:30 EST becomes 03:30 EDT → Still groups into 1-hour UTC buckets consistently
- Application layer handles local time display if needed

Data Type and Precision Edge Cases

Floating-point values present specific challenges for time-series data.

Edge Case	Problem	TempoDB Handling
Nan values	Invalid floating-point operations produce NaN	Gorilla compression cannot handle NaN; reject at ingestion with error
Infinity values	Division by zero produces ±Inf	Reject at ingestion with error
Extreme values	Very large/small floats may lose precision	Store as-is; compression may be less effective
Precision loss in aggregation	Repeated summation accumulates error	Use Kahan summation in <code>WindowAggregator</code> for critical metrics
Integer overflow in count	<code>COUNT()</code> over billions of points	Use int64 counters; monitor for overflow

Handling Tombstoned Data

When data is deleted or TTL-expired, it's marked with tombstones rather than immediately removed.

Tombstone Lifecycle:

- Marking:** File marked `Tombstoned=true` with `TombstoneTime` set
- Grace period:** File remains available for queries for `DefaultTTLGracePeriod` (5 minutes)
- Physical deletion:** After grace period, file deleted during next maintenance cycle
- Recovery:** If system crashes during grace period, tombstones persist and deletion resumes on restart

Query Behavior with Tombstones:

- During grace period:** Queries skip tombstoned data (invisible to users)
- Compaction:** Tombstoned files are excluded from compaction plans
- Emergency restore:** Administrator can manually remove tombstone marker before grace period expires

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Error monitoring	Log files with structured JSON	OpenTelemetry integration with metrics export
Crash recovery	WAL replay on startup	Point-in-time recovery with snapshotting
Disk monitoring	Periodic <code>statfs()</code> calls	Inotify/FANOTIFY for real-time space alerts
Memory limits	Go's runtime memory stats	Cgroup integration for container environments

Error Handling Infrastructure

Create an error hierarchy and recovery utilities in `internal/errors/` :

GO

```
// internal/errors/errors.go

package errors

import (
    "fmt"
    "runtime"
)

// ErrorSeverity indicates how severe an error is

type ErrorSeverity int

const (
    SeverityDebug ErrorSeverity = iota
    SeverityInfo
    SeverityWarning
    SeverityError
    SeverityCritical
)

// TempoError is the base error type for all TempoDB errors

type TempoError struct {

    Code        string
    Message     string
    Severity    ErrorSeverity
    Component   string
    Operation   string
    InnerError  error
    StackTrace  []byte
    Timestamp   time.Time
}
```

```
}

func NewTempoError(code, message string, severity ErrorSeverity, component, operation string, inner error) *TempoError {
    stack := make([]byte, 1024)

    n := runtime.Stack(stack, false)

    return &TempoError{
        Code:         code,
        Message:     message,
        Severity:    severity,
        Component:   component,
        Operation:   operation,
        InnerError:  inner,
        StackTrace:  stack[:n],
        Timestamp:   time.Now().UTC(),
    }
}

func (e *TempoError) Error() string {
    return fmt.Sprintf("[%s] %s: %s (component: %s, operation: %s)",
        e.Code, e.Severity, e.Message, e.Component, e.Operation)
}

// IsDiskFullError checks if error is due to disk full

func IsDiskFullError(err error) bool {
    // Check for ENOSPC or similar disk full errors

    return strings.Contains(err.Error(), "no space") ||
           strings.Contains(err.Error(), "ENOSPC") ||
           strings.Contains(err.Error(), "disk full")
```

```
}

// Recovery actions for different error types

type RecoveryAction struct {

    Name      string
    Description string
    Execute   func() error
}

// GetRecoveryActions returns recommended recovery actions for an error

func GetRecoveryActions(err error) []RecoveryAction {
    var actions []RecoveryAction

    // TODO 1: Check error type and severity
    // TODO 2: For disk full errors, suggest emergency compaction
    // TODO 3: For corrupt file errors, suggest quarantine procedure
    // TODO 4: For memory pressure, suggest flush and backpressure
    // TODO 5: Return appropriate recovery actions

    return actions
}
```

WAL Corruption Recovery Skeleton

```
// internal/wal/recovery.go                                     GO

package wal

import (
    "context"
    "fmt"
    "os"
    "path/filepath"
    "sort"
)

// RecoverFromCrash recovers unflushed data from WAL after a crash

func (w *WAL) RecoverFromCrash(ctx context.Context) (int64, error) {
    var recoveredPoints int64

    // TODO 1: List all WAL segment files in directory

    // TODO 2: Sort by sequence number (embedded in filename)

    // TODO 3: Check for checkpoint file to know where to start recovery

    // TODO 4: For each segment file (starting from checkpoint):
    //
    //   a. Open segment with Segment.Scan()
    //
    //   b. For each entry, parse into series key and data point
    //
    //   c. Insert into recovery memtable
    //
    //   d. When memtable reaches threshold, flush to TSM
    //
    //   e. Update recoveredPoints counter

    // TODO 5: Delete or archive processed segment files

    // TODO 6: Return count of recovered points
```

```
    return recoveredPoints, nil

}

// RepairCorruptSegment attempts to salvage data from a corrupt WAL segment

func RepairCorruptSegment(segmentPath string) ([]byte, error) {

    // TODO 1: Open file read-only

    // TODO 2: Read file in chunks, looking for valid entry boundaries

    // TODO 3: For each potentially valid entry, verify CRC32

    // TODO 4: Collect valid entries into recovery buffer

    // TODO 5: Return recovered data or error if nothing salvageable

    return nil, fmt.Errorf("repair not implemented")

}
```

Query Timeout and Cancellation

```
// internal/query/executor.go                                GO

package query

import (
    "context"
    "time"
)

// ExecuteQueryWithTimeout executes a query with configurable timeout

func (e *Executor) ExecuteQueryWithTimeout(ctx context.Context, plan *QueryPlan, timeout time.Duration) (QueryResult, error) {

    // Create timeout context

    timeoutCtx, cancel := context.WithTimeout(ctx, timeout)

    defer cancel()

    // Execute with proper resource tracking

    resultChan := make(chan QueryResult, 1)

    errorChan := make(chan error, 1)

    go func() {

        result, err := e.executePlan(timeoutCtx, plan)

        if err != nil {

            errorChan <- err

        } else {

            resultChan <- result

        }

    }()
}
```

```
select {

    case result := <-resultChan:
        return result, nil

    case err := <-errorChan:
        return QueryResult{}, err

    case <-timeoutCtx.Done():
        // TODO 1: Cancel all iterators and release resources
        // TODO 2: Log timeout with query details for monitoring
        // TODO 3: Return timeout error to client
        return QueryResult{}, timeoutCtx.Err()

    }
}

// executePlan with proper resource cleanup on cancellation

func (e *Executor) executePlan(ctx context.Context, plan *QueryPlan) (QueryResult, error) {
    // TODO 1: Check context cancellation periodically
    // TODO 2: Use defer to ensure iterators are closed even on panic
    // TODO 3: Implement memory tracking with periodic checks
    // TODO 4: Stream results rather than buffer entirely

    // Example iterator loop with cancellation check:
    for scanner.Next() {

        select {

            case <-ctx.Done():
                scanner.Close()
                return QueryResult{}, ctx.Err()
        }
    }
}
```

```
default:

    // Process point

}

}

return QueryResult{}, nil

}
```

Disk Space Monitoring and Emergency Actions

```
// internal/storage/disk_monitor.go GO

package storage

import (
    "syscall"
    "time"
)

// DiskMonitor periodically checks disk space and takes emergency actions

type DiskMonitor struct {

    dataDir      string
    threshold    float64 // e.g., 0.90 for 90% full
    checkInterval time.Duration
    stopCh       chan struct{}
}

// Start begins periodic disk monitoring

func (dm *DiskMonitor) Start() {
    go dm.monitorLoop()
}

// monitorLoop checks disk space and triggers emergency actions

func (dm *DiskMonitor) monitorLoop() {
    ticker := time.NewTicker(dm.checkInterval)
    defer ticker.Stop()

    for {
        select {
```

```
        case <-ticker.C:
            dm.checkDiskSpace()

        case <-dm.stopCh:
            return

        }
    }

func (dm *DiskMonitor) checkDiskSpace() {
    // TODO 1: Get disk usage statistics using syscall.Statfs

    // TODO 2: Calculate used percentage

    // TODO 3: If above warning threshold, log warning

    // TODO 4: If above critical threshold:
        // a. Trigger emergency compaction

        // b. Enter read-only mode if space critically low

        // c. Alert monitoring system

    // TODO 5: If below threshold after being above, resume normal operations
}

// emergencyCompaction triggers aggressive compaction to free space

func (dm *DiskMonitor) emergencyCompaction() error {
    // TODO 1: Get list of TSM files sorted by size

    // TODO 2: Select small files for aggressive compaction

    // TODO 3: Run compaction with highest priority

    // TODO 4: Delete source files immediately after successful compaction

    // TODO 5: Return freed space amount

    return nil
}
```

```
}
```

Language-Specific Hints

1. **Go Error Wrapping:** Use `fmt.Errorf("... : %w", err)` to wrap errors with context, enabling `errors.Is()` and `errors.As()` checks later.
2. **Resource Cleanup:** Always use `defer` for resource cleanup (file closing, mutex unlocking, iterator closing) to prevent leaks during panics.
3. **Context Propagation:** Pass `context.Context` through all long-running operations and check `ctx.Done()` in loops.
4. **Memory Monitoring:** Use `runtime.ReadMemStats()` to track memory usage and trigger backpressure before OOM kills the process.
5. **Graceful Shutdown:** Implement signal handling for SIGTERM/SIGINT to flush memtables and close files properly.
6. **Recover from Panics:** Use `recover()` in goroutine entry points to log panics and restart the goroutine if appropriate.

Milestone Checkpoint: Error Handling Validation

After implementing error handling, verify with these tests:

```

# Test 1: Disk full simulation

$ dd if=/dev/zero of=/tmp/tempodb_test/fill.disk bs=1M count=1000

$ go test ./internal/storage/ -run TestDiskFullRecovery -v

# Test 2: Corrupt WAL recovery

$ cp testdata/corrupt.wal /tmp/tempodb_test/wal/segment-0001.wal

$ ./tempodb-server --data-dir /tmp/tempodb_test

# Should log recovery attempt and continue startup

# Test 3: Query timeout

$ curl "http://localhost:8080/query?q=SELECT+*+FROM+cpu+WHERE+time+>+now()-365d" &

# After 30 seconds (default timeout), should receive timeout response

# Test 4: Out-of-order write handling

$ ./test_out_of_order.py --tolerance-window=5m

# Should accept points within window, warn on older points, reject very old points

```

Debugging Tips for Common Error Scenarios

Symptom	Likely Cause	How to Diagnose	Fix
Writes hang indefinitely	Disk full; WAL cannot <code>fsync</code>	Check disk space; look for "no space" in logs	Free disk space; emergency compaction
Query returns partial data	Corrupt TSM block skipped	Check logs for "corrupt block" or "CRC mismatch"	Restore from backup; check hardware
High memory usage	Cardinality explosion; large queries	Check series index size; monitor query memory	Limit series cardinality; add query memory limits
Clock skew warnings	Client clocks not synchronized	Check client NTP status; compare timestamps	Enforce NTP on clients; adjust <code>maxClockSkew</code>
Compaction not running	Disk near full; previous failure	Check compaction logs; disk space	Address disk space issue; manual compaction trigger

Testing Strategy

Milestone(s): This section provides the verification framework for all five milestones, ensuring each component meets its acceptance criteria through systematic testing approaches and concrete milestone checkpoints.

The testing strategy for TempoDB must validate both functional correctness and performance characteristics across all system components. Time-series databases present unique testing challenges due to their focus on temporal data patterns, compression algorithms, and high-volume ingestion. This section outlines a comprehensive approach combining traditional testing methods with specialized techniques for time-series systems, providing clear checkpoints for each milestone.

Mental Model: The Scientific Laboratory

Think of testing TempoDB like running a scientific laboratory. Each component represents an experimental apparatus that must be validated individually (unit tests) and in concert with others (integration tests). We create controlled experiments (property-based tests) to verify fundamental laws of data preservation (no data loss, correct aggregation). Performance benchmarks are our stress tests, measuring how the apparatus behaves under extreme conditions. The milestone checkpoints are our peer reviews—concrete demonstrations that each apparatus performs its intended function before moving to the next phase of the larger experiment.

Testing Approaches and Property Verification

Effective testing for a time-series database requires a layered approach that addresses specific characteristics of temporal data, compression, and high-throughput systems. The following table outlines the primary testing strategies:

Testing Approach	When to Apply	Key Techniques	TempoDB-Specific Considerations
Unit Testing	During component development	<ul style="list-style-type: none"> Mock dependencies Table-driven tests Edge case exploration 	Test compression/decompression round-trips, time range calculations, and series key hashing
Integration Testing	After component interfaces stabilize	<ul style="list-style-type: none"> Component composition End-to-end workflows Database lifecycle tests 	Test WAL recovery after crashes, query execution across multiple TSM files, compaction workflows
Property-Based Testing	For algorithmic components	<ul style="list-style-type: none"> Hypothesis testing Random input generation Invariant verification 	Verify compression never loses data, aggregations are associative, time windows align correctly
Golden File Testing	For file format stability	<ul style="list-style-type: none"> Versioned reference files Binary comparison Forward/backward compatibility 	Ensure TSM file format stability, WAL segment structure consistency
Fuzz Testing	For robustness validation	<ul style="list-style-type: none"> Random mutation of inputs Protocol fuzzing Memory corruption simulation 	Fuzz query parser with malformed queries, ingest API with corrupt line protocol
Performance Benchmarking	For acceptance criteria	<ul style="list-style-type: none"> Throughput measurement Latency percentiles Memory/CPU profiling 	Verify thousands of points per second write throughput, sub-second query latency for common ranges
Concurrency Testing	For thread-safe components	<ul style="list-style-type: none"> Race condition detection Deadlock detection Stress testing with goroutines 	Test concurrent writes to same series, simultaneous queries during compaction
Recovery Testing	For durability guarantees	<ul style="list-style-type: none"> Simulated crashes Disk full scenarios 	Verify WAL recovery replays all acknowledged writes, TSM file corruption

Testing Approach	When to Apply	Key Techniques	TempoDB-Specific Considerations
		• Corrupt file handling	detection

Property Verification for Critical Components

Certain TempoDB components require specialized verification approaches to ensure they maintain essential invariants:

Compression Algorithms Invariants:

- **Losslessness:** For any sequence of timestamps T and values V , $\text{decompress}(\text{compress}(T, V)) == (T, V)$
- **Monotonicity Preservation:** If timestamps are strictly increasing, decompressed timestamps maintain this order
- **Value Fidelity:** Floating-point values maintain their exact binary representation after compression/decompression cycles

Time Window Alignment Properties:

- **Deterministic Bucketing:** For any timestamp t and window duration d , $\text{alignToWindow}(t, d)$ always returns the same result
- **Contiguity:** Sequential windows have no gaps: $\text{alignToWindow}(t, d) + d == \text{alignToWindow}(t + d, d)$
- **Idempotence:** Applying alignment twice yields same result: $\text{alignToWindow}(\text{alignToWindow}(t, d), d) == \text{alignToWindow}(t, d)$

Aggregation Function Properties:

- **Associativity:** For commutative aggregations (sum, count), order of application doesn't matter
- **Identity Elements:** Empty window aggregations yield appropriate zero values (0 for sum, NaN for avg, etc.)
- **Monotonicity:** For ordered aggregations (min, max), results respect the partial order of input values

Key Insight: Property-based testing is particularly valuable for time-series databases because it can automatically generate edge cases like clock jumps, out-of-order points, and NaN values that developers might not think to test manually.

Golden File Testing for Storage Format

The TSM file format represents a contract between different versions of TempoDB. Golden file testing ensures this contract remains stable:

1. **Reference File Creation:** Generate TSM files with known content using a specific version

2. **Binary Comparison:** Compare new files byte-for-byte with reference files
3. **Round-Trip Verification:** Write → Read → Verify data matches original
4. **Version Compatibility:** Ensure new readers can read old files and vice versa

The following table defines the golden file test matrix:

Test Scenario	Input Data	Expected TSM File Characteristics	Validation Method
Empty series	No data points	Minimal file with only header and empty index	File size matches expected, header valid
Single block	1000 sequential points	One data block, index with single entry	Block count = 1, all points recoverable
Multiple series	5 series × 500 points each	Index with 5 entries, interleaved blocks	Each series retrievable independently
Max block size	Points exceeding <code>DefaultMaxPointsPerBlock</code>	Multiple blocks per series	Block count = $\text{ceil}(\text{points}/\text{maxPoints})$
Mixed timestamps	Random, non-sequential timestamps	Out-of-order points sorted in block	Retrieved points in timestamp order
Special values	<code>Nan</code> , $\pm\infty$, extremely large/small floats	Gorilla compression handles special cases	Values preserved exactly after round-trip

Fuzz Testing for Query Language

The query parser and executor must handle malformed input gracefully. Fuzz testing generates random valid and invalid queries to uncover crashes or incorrect behavior:

Query Fuzzing Strategy:

1. **Grammar-Based Generation:** Create queries that follow the grammar but with random components
2. **Mutation Testing:** Take valid queries and mutate parts (change operators, remove tokens)
3. **Protocol Fuzzing:** Test HTTP API with malformed JSON, incorrect headers, large payloads

Critical Invariants for Fuzzed Queries:

- No panic or crash on any input (malformed or valid)
- Malformed queries return descriptive error messages, not empty results
- Memory bounds respected even for pathological queries
- Query timeout enforcement prevents runaway execution

Common Pitfalls in Time-Series Database Testing

⚠ Pitfall: Testing Only Sequential Timestamps Testing with perfectly sequential timestamps misses edge cases like clock jumps, duplicate timestamps, and out-of-order data. Real-world time-series data includes irregularities that the system must handle gracefully.

Why it's wrong: Systems that pass tests with perfect data may fail catastrophically with real-world data containing clock adjustments, batch backfills, or multi-source ingestion.

How to fix: Include test cases with:

- Timestamps that jump forward/backward (simulating NTP adjustments)
- Duplicate timestamps (same nanosecond from different sources)
- Gaps in data (missing periods)
- Out-of-order points within the tolerance window

⚠ Pitfall: Ignoring Floating-Point Edge Cases Testing with "nice" floating-point values (1.0, 2.5, etc.) misses issues with special IEEE 754 values and precision boundaries.

Why it's wrong: Gorilla compression must handle NaN, ±Inf, denormalized numbers, and values near floating-point precision boundaries correctly.

How to fix: Create comprehensive floating-point test vectors:

```
NaN, -NaN, +Inf, -Inf, 0.0, -0.0  
Denormalized: 1e-310, -1e-310  
Boundary values: max float64, -max float64, min positive normal  
Precision transition: 1.0 + 2^-52, 1.0 + 2^-53
```

PLAINTEXT

⚠ Pitfall: Testing Compression Ratio Instead of Correctness Focusing on achieving specific compression ratios rather than verifying lossless compression.

Why it's wrong: Compression algorithms should never lose data, even if compression ratio suffers. Testing for ratios can mask data corruption bugs.

How to fix: Always test round-trip correctness first. Compression ratio tests should be separate benchmarks, not correctness tests.

⚠ Pitfall: Single-Threaded Performance Testing Measuring performance with single-threaded writes/queries that don't reflect real concurrent load.

Why it's wrong: Real systems experience concurrent writes from multiple sources and parallel queries. Single-threaded tests miss lock contention, cache coherency overhead, and memory barrier costs.

How to fix: Design performance tests that simulate:

- Multiple concurrent writers (different goroutines writing to different series)
- Mixed read/write workloads (queries while ingesting)

- Concurrent compactions during queries

ADR: Choosing Test Frameworks and Approaches

Decision: Comprehensive Testing Pyramid with Go-Native Tooling

- **Context:** TempoDB needs robust testing across multiple dimensions (correctness, performance, concurrency) while maintaining fast feedback cycles for developers. The Go ecosystem provides excellent testing tools that integrate seamlessly.
- **Options Considered:**
 1. **Minimalist approach:** Standard Go testing package only
 2. **Comprehensive external frameworks:** Bring in multiple specialized testing libraries (testify, ginkgo, goconvey)
 3. **Hybrid native approach:** Go testing package augmented with carefully selected single-purpose libraries for property testing and HTTP testing
- **Decision:** Hybrid native approach using `testing` package + `quick` for property testing + `httptest` for API tests
- **Rationale:** The standard `testing` package is sufficient for most needs and keeps dependencies minimal. `quick` provides property testing without external dependencies. `httptest` is part of the standard library. This approach minimizes cognitive overhead and build complexity while providing necessary testing capabilities.
- **Consequences:** Developers need to write more boilerplate for assertions but gain deeper understanding of test failures. The test suite remains fast and integrated with standard Go tooling.

Option	Pros	Cons	Chosen?
Minimalist (testing only)	<ul style="list-style-type: none"> • Zero dependencies • Consistent with Go idioms • Fast compilation 	<ul style="list-style-type: none"> • Verbose assertion code • No property testing • Limited HTTP test utilities 	✗
Comprehensive frameworks	<ul style="list-style-type: none"> • Rich assertion libraries • BDD-style syntax available • Integrated utilities 	<ul style="list-style-type: none"> • Multiple dependencies • Steeper learning curve • Potential version conflicts 	✗
Hybrid native approach	<ul style="list-style-type: none"> • Minimal dependencies • Property testing support • Standard HTTP testing • Go-native patterns 	<ul style="list-style-type: none"> • More boilerplate for assertions • No BDD syntax 	✓

Milestone Verification Checkpoints

Each milestone includes specific acceptance criteria that must be verified through executable tests and manual validation. These checkpoints provide concrete steps to confirm successful implementation.

Milestone 1: Storage Engine Verification

Objective: Validate TSM file format, compression algorithms, and block-based storage.

Verification Checklist:

- TSM files can be written and read correctly
- Delta-of-delta compression reduces storage for sequential timestamps
- Gorilla compression maintains floating-point precision
- Block index correctly maps time ranges to file offsets
- Memory-mapped files enable efficient random access

Validation Commands and Expected Output:

BASH

```
# Run storage engine unit tests

$ go test ./internal/storage/... -v -run TestTSM

==== RUN    TestTSMWriteReadRoundTrip

--- PASS: TestTSMWriteReadRoundTrip (0.08s)

==== RUN    TestDeltaOfDeltaCompression

--- PASS: TestDeltaOfDeltaCompression (0.12s)

==== RUN    TestGorillaCompression

--- PASS: TestGorillaCompression (0.15s)

==== RUN    TestMemoryMappedAccess

--- PASS: TestMemoryMappedAccess (0.05s)

PASS

# Generate and inspect a TSM file

$ go run cmd/inspect/main.go --file testdata/sample.tsm

TSM File: testdata/sample.tsm

Version: 1

Series Count: 3

Total Blocks: 5

File Size: 45.2 KB

Compression Ratio: 4.7:1

Series: cpu,host=server1

Blocks: 2 (times 2023-01-01T00:00:00Z to 2023-01-01T01:40:00Z)

Points: 1000

Avg block size: 8.1 KB

# Benchmark compression performance

$ go test ./internal/storage/... -bench=BenchmarkCompression -benchttime=10s
```

BenchmarkDeltaOfDeltaCompress-8 MB/s	500000	24560 ns/op	407.03
BenchmarkGorillaCompress-8 MB/s	300000	38920 ns/op	256.95

Manual Validation Steps:

1. Create a test program that writes 10,000 sequential points to a TSM file
2. Verify file size is significantly smaller than uncompressed data (expect 4-8x compression)
3. Use a hex editor to examine file structure: header, data blocks, index, footer
4. Write a simple reader that memory-maps the file and extracts points from middle blocks without reading entire file
5. Confirm that reading random blocks has constant time complexity regardless of file position

Milestone 2: Write Path Verification

Objective: Validate high-throughput ingestion, WAL durability, and memtable management.

Verification Checklist:

- Write-ahead log ensures durability before acknowledgment
- Memtable buffers writes and flushes at size threshold
- Batch ingestion improves throughput
- Out-of-order writes are handled correctly within tolerance window
- Backpressure mechanism prevents unbounded memory growth

Validation Commands and Expected Output:

BASH

```
# Test WAL recovery after simulated crash

$ go test ./internal/wal/... -v -run TestWALRecovery

== RUN TestWALRecovery

Writing 1000 points to WAL...

Simulating crash (kill -9)...

Recovering from WAL...

Recovered 1000 points, 0 lost

--- PASS: TestWALRecovery (0.22s)

# Benchmark write throughput

$ go test ./internal/ingest/... -bench=BenchmarkWriteThroughput -benctime=30s

BenchmarkWriteThroughput-8          200000           178200 ns/op   5609 ops/sec
BenchmarkBatchWrite-8              500000           54200 ns/op  18450 ops/sec

# Test backpressure mechanism

$ go run cmd/loadtest/main.go --points 1000000 --workers 10 --backpressure

Starting load test with 10 workers...

[WARNING] Backpressure engaged at 85% capacity

Throughput limited to maintain stability

Total points: 1,000,000

Successful: 1,000,000

Failed (backpressure): 0

Average latency: 4.2ms

P99 latency: 18.7ms

# Verify out-of-order handling

$ go test ./internal/ingest/... -v -run TestOutOfOrder

== RUN TestOutOfOrderWithinWindow
```

```
--- PASS: TestOutOfOrderWithinWindow (0.03s)
```

```
== RUN TestOutOfOrderBeyondWindow
```

```
--- PASS: TestOutOfOrderBeyondWindow (0.04s)
```

Manual Validation Steps:

1. Start TempoDB server with small memtable size limit (e.g., 1MB)
2. Send writes at increasing rates until memtable fills
3. Observe flush to disk and new memtable creation without data loss
4. Kill server process abruptly during active writes
5. Restart server and verify all acknowledged writes are recoverable from WAL
6. Send points with timestamps in random order, verify they're stored in correct temporal order

Milestone 3: Query Engine Verification

Objective: Validate range queries, aggregations, and filtering performance.

Verification Checklist:

- Time-range queries return correct points within bounds
- Tag filtering uses inverted index efficiently
- Aggregation functions produce mathematically correct results
- GROUP BY time() creates correct window buckets
- Predicate pushdown skips irrelevant blocks

Validation Commands and Expected Output:

BASH

```
# Run query engine tests

$ go test ./internal/query/... -v -run "TestQuery"

==== RUN    TestTimeRangeQuery

--- PASS: TestTimeRangeQuery (0.07s)

==== RUN    TestTagFiltering

--- PASS: TestTagFiltering (0.05s)

==== RUN    TestAggregationFunctions

--- PASS: TestAggregationFunctions (0.09s)

==== RUN    TestGroupByTime

--- PASS: TestGroupByTime (0.12s)

# Test predicate pushdown optimization

$ go test ./internal/query/... -v -run TestPredicatePushdown

==== RUN    TestPredicatePushdown

Query scanned 5 of 20 blocks (75% reduction)

--- PASS: TestPredicatePushdown (0.08s)

# Benchmark query performance

$ go test ./internal/query/... -bench=BenchmarkQuery -benchtime=10s

BenchmarkRangeQuery-8           100000          156800 ns/op
BenchmarkAggregationQuery-8     50000           312400 ns/op
BenchmarkGroupByQuery-8         30000           521800 ns/op

# Execute complex query manually

$ curl -X POST "http://localhost:8086/query" \
-d "SELECT mean(value) FROM cpu
WHERE host='server1' AND time >= '2023-01-01T00:00:00Z'
GROUP BY time(1h)
```

```

LIMIT 24"

{
  "results": [
    {
      "series": [
        {
          "name": "cpu",
          "columns": ["time", "mean"],
          "values": [
            ["2023-01-01T00:00:00Z", 45.2],
            ["2023-01-01T01:00:00Z", 46.8],
            ...
          ]
        }
      ]
    }
  ]
}

```

Manual Validation Steps:

1. Load dataset with known statistical properties (e.g., 1M points with normal distribution)
2. Execute aggregation queries and verify results match pre-computed values
3. Create query that spans multiple TSM files, verify all relevant files are accessed
4. Test time windows that don't align with block boundaries, verify correct point inclusion
5. Execute query with very selective tag filter, confirm only matching series are scanned
6. Test queries with no matching data, verify empty result (not error)

Milestone 4: Retention & Compaction Verification

Objective: Validate automatic data lifecycle management and storage optimization.

Verification Checklist:

- TTL enforcement deletes expired data blocks
- Compaction merges small files without data loss
- Downsampling reduces granularity while preserving trends
- Background processes don't block foreground operations

Validation Commands and Expected Output:

BASH

```
# Test TTL enforcement

$ go test ./internal/retention/... -v -run TestTTL

==== RUN    TestTLEnforcement

Creating data with 1-minute TTL...

Waiting 2 minutes for expiration...

Checking data...

Expired data removed: 1000 points

Active data retained: 500 points

--- PASS: TestTLEnforcement (62.1s)

# Monitor compaction progress

$ go run cmd/compaction-monitor/main.go --watch

Watching compaction directory...

[14:30:00] Level 0: 4 files (45.2MB) → Level 1: 1 file (42.8MB)

[14:45:00] Level 1: 3 files (128.4MB) → Level 2: 1 file (124.1MB)

[15:00:00] Downsampling applied: 1s → 60s granularity

Storage reduction: 68%

# Verify compaction correctness

$ go test ./internal/compaction/... -v -run TestCompactionCorrectness

==== RUN    TestCompactionCorrectness

Input files: 5, points: 50,000

Output file: 1, points: 50,000

All points verified, checksums match

--- PASS: TestCompactionCorrectness (1.24s)

# Test concurrent access during compaction

$ go test ./internal/compaction/... -v -run TestConcurrentAccess
```

```
==== RUN TestConcurrentAccess

Writes during compaction: 10,000

Queries during compaction: 500

All operations completed successfully

No data corruption detected

--- PASS: TestConcurrentAccess (3.85s)
```

Manual Validation Steps:

1. Write data with mixed ages (some expired, some current)
2. Trigger TTL enforcement and verify only expired data removed
3. Create many small TSM files, trigger compaction, verify single larger file created
4. While compaction runs, execute concurrent writes and queries, verify no blocking
5. Generate high-resolution data, enable downsampling, verify low-resolution version created
6. Query downsampled data, verify it approximates original within expected error bounds
7. Fill disk to near capacity, verify emergency compaction triggers and frees space

Milestone 5: Query Language & API Verification

Objective: Validate expressive query interface and API compatibility.

Verification Checklist:

- Query language parses SELECT with time-range predicates
- HTTP API accepts writes in line protocol format
- Aggregation functions work within query expressions
- Prometheus remote read/write endpoints function correctly
- Grafana data source compatibility confirmed

Validation Commands and Expected Output:

BASH

```
# Test query language parsing

$ go test ./internal/querylang/... -v -run TestParser

==== RUN    TestParserValidQueries

--- PASS: TestParserValidQueries (0.04s)

==== RUN    TestParserErrorRecovery

--- PASS: TestParserErrorRecovery (0.03s)

# Test HTTP API endpoints

$ go test ./internal/api/... -v -run TestAPI

==== RUN    TestWriteEndpoint

--- PASS: TestWriteEndpoint (0.06s)

==== RUN    TestQueryEndpoint

--- PASS: TestQueryEndpoint (0.08s)

==== RUN    TestPrometheusRemoteWrite

--- PASS: TestPrometheusRemoteWrite (0.07s)

# Verify line protocol parsing

$ echo 'cpu,host=server1 value=42.5 1672531200000000000' | \
curl -X POST http://localhost:8086/write --data-binary @-
HTTP/1.1 204 No Content

# Test Prometheus compatibility

$ go run cmd/prometheus-test/main.go --remote-write --remote-read

Testing Prometheus remote write...

1000 samples written successfully

Testing Prometheus remote read...

1000 samples read successfully

Compatibility: PASS
```

```
# Test Grafana data source

$ go run cmd/grafana-test/main.go --datasource-test

Connecting to Grafana...

Testing simple query: PASS

Testing template variables: PASS

Testing annotations: PASS

Data source compatible: YES

# Load test with realistic queries

$ go run cmd/query-loadtest/main.go --queries 10000 --concurrent 50

Executing 10,000 queries with 50 concurrent clients...

Successful: 9,998 (99.98%)

Failed: 2 (0.02%)

Average latency: 47ms

P95 latency: 132ms

Throughput: 1063 queries/second
```

Manual Validation Steps:

1. Write points using line protocol via curl, verify success response
2. Execute complex query with nested functions, verify correct parsing and execution
3. Configure Prometheus to use TempoDB as remote storage, verify data flows correctly
4. Set up Grafana with TempoDB data source, create dashboard with multiple panels
5. Test malformed queries return helpful error messages, not server errors
6. Verify API responds with appropriate CORS headers for web UI compatibility
7. Test streaming responses for large result sets, verify memory doesn't grow linearly

Implementation Guidance

While testing strategy is primarily about design and approach, implementing effective tests requires careful structure and utilities. This guidance provides foundational testing infrastructure.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Unit Testing	Go <code>testing</code> package	Testify assertion library
Property Testing	Go <code>testing/quick</code>	Gopter property-based testing
HTTP Testing	<code>net/http/httptest</code>	GoMock for HTTP client mocking
Benchmarking	Go <code>testing</code> benchmarks	Custom benchmarking framework
Concurrency Testing	Go race detector	ThreadSanitizer integration
Code Coverage	<code>go test -cover</code>	SonarQube integration

B. Recommended Test File Structure

```
tempo/
├── cmd/
│   ├── test-tools/           # Specialized testing utilities
│   │   ├── tsm-validator/    # Validates TSM file integrity
│   │   ├── query-fuzzer/     # Generates random queries
│   │   └── load-generator/   # Generates synthetic load
│   └── ...
├── internal/
│   ├── storage/             # Unit tests for TSM format
│   │   ├── tsm_test.go       # Unit tests for TSM format
│   │   ├── compression_test.go # Property tests for compression
│   │   └── benchmark_test.go # Performance benchmarks
│   ├── ingest/              # WAL recovery tests
│   │   ├── wal_test.go       # WAL recovery tests
│   │   └── memtable_test.go  # Concurrency tests
│   ├── query/               # Query execution tests
│   │   ├── engine_test.go    # Query execution tests
│   │   ├── planner_test.go   # Query planning tests
│   │   └── iterator_test.go  # Iterator pattern tests
│   ├── retention/           # TTL enforcement tests
│   │   ├── ttl_test.go       # TTL enforcement tests
│   │   └── compaction_test.go # Compaction correctness tests
│   ├── api/                 # HTTP API tests
│   │   ├── http_test.go      # HTTP API tests
│   │   └── protocol_test.go  # Line protocol parsing tests
│   └── querylang/           # Parser fuzz tests
│       ├── parser_test.go    # Parser fuzz tests
│       └── ast_test.go        # AST validation tests
└── testdata/               # Golden files and test datasets
    ├── golden/
    │   ├── tsm-v1/            # Reference TSM files
    │   └── wal-v1/            # Reference WAL segments
    └── synthetic/
        ├── normal-dist/       # Normally distributed test data
        └── random-walk/        # Random walk test data
```

C. Infrastructure Starter Code

Golden File Test Helper (complete, ready to use):

GO

```
// testutil/golden.go

package testutil

import (
    "crypto/sha256"
    "encoding/hex"
    "fmt"
    "io"
    "os"
    "path/filepath"
    "testing"
)

// GoldenFile represents a reference file for comparison testing

type GoldenFile struct {
    Name      string
    Path      string
    Checksum string
}

// LoadGoldenFile loads a golden file from testdata directory

func LoadGoldenFile(t *testing.T, version, name string) GoldenFile {
    t.Helper()

    path := filepath.Join("testdata", "golden", version, name)
    data, err := os.ReadFile(path)
    if err != nil {
        t.Fatalf("Failed to load golden file %s: %v", name, err)
    }
}
```

```
}

hash := sha256.Sum256(data)

return GoldenFile{

    Name:      name,
    Path:      path,
    Checksum: hex.EncodeToString(hash[:]),
}

}

// CompareToGolden compares current data to golden file

func CompareToGolden(t *testing.T, golden GoldenFile, current []byte) bool {

t.Helper()

// Calculate checksum of current data

currentHash := sha256.Sum256(current)

currentChecksum := hex.EncodeToString(currentHash[:])



if currentChecksum != golden.Checksum {

    // Write diff for debugging

    diffPath := filepath.Join(t.TempDir(), golden.Name+".diff")

    if err := os.WriteFile(diffPath, current, 0644); err == nil {

        t.Logf("Current output saved to %s for comparison", diffPath)

    }

    t.Errorf("Output differs from golden file %s", golden.Name)

    t.Logf("Expected checksum: %s", golden.Checksum)
```

```
    t.Logf("Actual checksum: %s", currentChecksum)

    return false
}

return true
}

// UpdateGoldenFile updates golden file with current data

func UpdateGoldenFile(t *testing.T, version, name string, data []byte) {
    t.Helper()

    if os.Getenv("UPDATE_GOLDEN") == "" {
        t.Skip("Set UPDATE_GOLDEN=1 to update golden files")
    }

    dir := filepath.Join("testdata", "golden", version)

    if err := os.MkdirAll(dir, 0755); err != nil {
        t.Fatalf("Failed to create golden directory: %v", err)
    }

    path := filepath.Join(dir, name)

    if err := os.WriteFile(path, data, 0644); err != nil {
        t.Fatalf("Failed to write golden file: %v", err)
    }

    t.Logf("Updated golden file: %s", path)
}
```

Property Test Helper for Compression (complete, ready to use):

GO

```
// testutil/property.go

package testutil

import (
    "math"
    "math/rand"
    "testing"
    "testing/quick"
    "time"
)

// TimeSeriesGenerator generates realistic time-series data for property testing

type TimeSeriesGenerator struct {
    rng *rand.Rand
}

func NewTimeSeriesGenerator(seed int64) *TimeSeriesGenerator {
    return &TimeSeriesGenerator{
        rng: rand.New(rand.NewSource(seed)),
    }
}

// GenerateSequentialTimestamps generates n sequential timestamps with possible jitter

func (g *TimeSeriesGenerator) GenerateSequentialTimestamps(n int, start time.Time, interval time.Duration, jitter float64) []time.Time {
    timestamps := make([]time.Time, n)
    current := start

    for i := 0; i < n; i++ {
        timestamps[i] = current
        current = current.Add(interval)
        if jitter != 0 {
            current = current.Add(time.Duration(jitter))
        }
    }
}
```

```

        // Add jitter ( $\pm$ jitter% of interval)

        jitterNs := int64(float64(interval.Nanoseconds()) * (g.rng.Float64()*2*jitter - jitter))

        timestamps[i] = current.Add(time.Duration(jitterNs))

        current = current.Add(interval)

    }

    return timestamps
}

// GenerateRandomWalkValues generates values following a random walk

func (g *TimeSeriesGenerator) GenerateRandomWalkValues(n int, start, volatility float64) []float64 {
    values := make([]float64, n)

    current := start

    for i := 0; i < n; i++ {
        // Random step ( $\pm$ volatility)

        step := (g.rng.Float64()*2 - 1) * volatility

        current += step

        values[i] = current
    }

    return values
}

// GenerateMixedValues generates values including edge cases

func (g *TimeSeriesGenerator) GenerateMixedValues(n int) []float64 {
    values := make([]float64, n)
}

```

```
for i := 0; i < n; i++ {
    switch g.rng.Intn(20) {
        case 0:
            values[i] = math.NaN()
        case 1:
            values[i] = math.Inf(1)
        case 2:
            values[i] = math.Inf(-1)
        case 3:
            values[i] = 0.0
        case 4:
            values[i] = -0.0
        default:
            values[i] = g.rng.NormFloat64() * 100
    }
}

return values
}

// CheckProperty runs a property test with helpful error reporting

func CheckProperty(t *testing.T, name string, f interface{}, config *quick.Config) bool {
    t.Helper()

    if config == nil {
        config = &quick.Config{
```

```
        MaxCount: 1000,  
  
        Rand:      rand.New(rand.NewSource(time.Now().UnixNano())),  
    }  
  
}  
  
  
err := quick.Check(f, config)  
  
if err != nil {  
  
    t.Errorf("Property %s failed: %v", name, err)  
  
    return false  
  
}  
  
  
return true  
}
```

D. Core Test Skeleton Code

TSM File Format Test (skeleton with TODOs):

GO

```
// internal/storage/tsm_test.go

func TestTSMWriteReadRoundTrip(t *testing.T) {

    // TODO 1: Create test directory using t.TempDir()

    // TODO 2: Generate test data: 1000 sequential points with timestamps and values

    // TODO 3: Create TSMWriter and write test data

    // TODO 4: Call Finish() to complete file

    // TODO 5: Open TSMReader for the created file

    // TODO 6: Read back all points using ReadBlock

    // TODO 7: Verify point count matches original

    // TODO 8: Compare each timestamp and value for exact equality

    // TODO 9: Test edge cases: empty series, single point, max block size

    // TODO 10: Verify file checksum validation works

}
```

```
func TestDeltaOfDeltaCompression(t *testing.T) {

    // TODO 1: Generate timestamps with different patterns:
    //
    //         - Perfectly sequential (delta constant)
    //
    //         - Increasing intervals (delta increasing)
    //
    //         - Random intervals within bounds
    //
    //         - Clock jumps (forward and backward)

    // TODO 2: Apply delta-of-delta compression to each pattern

    // TODO 3: Decompress and verify round-trip correctness

    // TODO 4: Calculate compression ratio for each pattern

    // TODO 5: Verify monotonic timestamps remain monotonic after compression

    // TODO 6: Test with maximum timestamp values (near uint64 limit)

}

func TestGorillaCompressionProperty(t *testing.T) {
```

```
// Property: decompress(compress(values)) == values

testutil.CheckProperty(t, "GorillaCompressionRoundTrip", func(values []float64) bool {

    // TODO 1: Handle special case: empty slice

    if len(values) == 0 {

        return true

    }

    // TODO 2: Compress values using Gorilla algorithm

    // TODO 3: Decompress back to float64 slice

    // TODO 4: Compare decompressed with original

    // TODO 5: Special handling for NaN (math.IsNaN)

    // TODO 6: Special handling for Inf (math.IsInf)

    // TODO 7: Use math.Float64bits for exact binary comparison

    // TODO 8: Return true only if all values match exactly

    return false // Replace with actual implementation

}, nil)

}
```

WAL Recovery Test (skeleton with TODOs):

GO

```
// internal/wal/recovery_test.go

func TestWALRecoveryAfterCrash(t *testing.T) {

    // TODO 1: Create WAL directory using t.TempDir()

    // TODO 2: Initialize WAL with SegmentConfig

    // TODO 3: Write 1000 points to WAL using WriteEntry

    // TODO 4: Simulate crash by not calling proper shutdown

    // TODO 5: Create new WAL instance pointing to same directory

    // TODO 6: Call recovery function to scan WAL segments

    // TODO 7: Verify all 1000 points are recovered

    // TODO 8: Check that recovery handles partial writes (truncated entries)

    // TODO 9: Test recovery with multiple segment files

    // TODO 10: Verify recovered points maintain write order

}

func TestConcurrentWALWrites(t *testing.T) {

    // TODO 1: Create WAL with appropriate configuration

    // TODO 2: Launch 10 goroutines writing concurrently

    // TODO 3: Each goroutine writes 100 unique points

    // TODO 4: Use sync.WaitGroup to wait for all writes

    // TODO 5: Scan WAL to recover all points

    // TODO 6: Verify total point count equals 1000 (10×100)

    // TODO 7: Check for duplicate or missing points

    // TODO 8: Verify write ordering within each goroutine is preserved

    // TODO 9: Run with race detector enabled (go test -race)

    // TODO 10: Test with varying levels of concurrency (1, 10, 100 goroutines)

}
```

Query Engine Integration Test (skeleton with TODOs):

```
// internal/query/integration_test.go

func TestEndToEndQueryWorkflow(t *testing.T) {

    // TODO 1: Create temporary database instance

    // TODO 2: Write test data covering multiple time ranges and series

    // TODO 3: Execute query with time range predicate

    // TODO 4: Verify returned points match expected subset

    // TODO 5: Test predicate pushdown by counting blocks accessed

    // TODO 6: Execute aggregation query (SUM, AVG, etc.)

    // TODO 7: Compare results with manually calculated expected values

    // TODO 8: Test GROUP BY time() with various window sizes

    // TODO 9: Verify window alignment correctness

    // TODO 10: Test query cancellation mid-execution

    // TODO 11: Test memory limits on large result sets

    // TODO 12: Verify streaming works for large queries

}
```

GO

E. Language-Specific Hints

Go Testing Best Practices:

- Use `t.Helper()` in test helper functions to improve error location
- Leverage `t.Cleanup()` for automatic resource cleanup
- Use table-driven tests with `tt := range tests` pattern
- For benchmarks, use `b.ResetTimer()` and `b.StopTimer()` appropriately
- Enable race detection with `go test -race` for all concurrency tests

Time-Series Specific Testing:

- Use `time.Now()` sparingly in tests—prefer fixed timestamps for reproducibility
- When testing time-based logic, mock time using interfaces for deterministic tests
- For performance tests, warm up caches before starting measurements
- When testing compression, include both synthetic and real-world datasets

Property Testing Tips:

- Start with small `MaxCount` values (100) and increase as tests stabilize
- Use custom generators for domain-specific types (timestamps, series keys)
- For floating-point comparisons, use `math.Float64bits` for exact equality
- Save failing cases to files for deterministic reproduction

F. Debugging Tips Table

Symptom	Likely Cause	How to Diagnose	Fix
Test passes locally but fails in CI	Timezone differences or non-deterministic test	Add <code>log.Printf("Timezone: %v", time.Local)</code> to test; check for <code>time.Now()</code> usage	Use fixed timestamps or mock time source
Compression test fails intermittently	Floating-point NaN/Inf handling	Add debug logging showing each value before/after compression	Ensure special values handled in compression algorithm
Query returns wrong aggregation results	Incorrect window alignment	Log window start/end times and which points are included	Verify <code>alignToWindow</code> implementation matches specification
WAL recovery misses some writes	Partial writes not handled	Check file size vs. expected; add CRC validation	Implement WAL entry framing with length prefixes
Memory leak in tests	Goroutines not cleaned up	Use <code>runtime.NumGoroutine()</code> in test cleanup	Ensure all background jobs have stop mechanisms
Race condition detected	Shared mutable state	Run with <code>-race</code> flag; examine stack traces	Add appropriate synchronization or use immutable data
Benchmark results inconsistent	Cache effects or GC pauses	Use <code>benchstat</code> tool across multiple runs; check for allocation hotspots	Increase benchmark count; use <code>testing.AllocsPerRun</code>

G. Performance Testing Framework

Create a dedicated performance testing framework that can be run independently:

GO

```
// cmd/benchmark/main.go (complete starter code)

package main

import (
    "flag"
    "fmt"
    "log"
    "os"
    "runtime"
    "time"

    "github.com/tempodb/internal/benchmarks"
)

func main() {
    var (
        benchmark = flag.String("benchmark", "write", "Benchmark to run (write, query, compression)")
        duration = flag.Duration("duration", 30*time.Second, "Benchmark duration")
        workers = flag.Int("workers", runtime.NumCPU(), "Number of concurrent workers")
        output = flag.String("output", "report.json", "Output file for results")
    )
    flag.Parse()

    log.Printf("Starting %s benchmark with %d workers for %v",
        *benchmark, *workers, *duration)
    log.Printf("Go version: %s", runtime.Version())
    log.Printf("CPU cores: %d", runtime.NumCPU())
```

```
var result benchmarks.Result

switch *benchmark {

case "write":

    result = benchmarks.RunWriteBenchmark(*workers, *duration)

case "query":

    result = benchmarks.RunQueryBenchmark(*workers, *duration)

case "compression":

    result = benchmarks.RunCompressionBenchmark(*workers, *duration)

default:

    log.Fatalf("Unknown benchmark: %s", *benchmark)

}

// Save results

if err := result.Save(*output); err != nil {

    log.Fatalf("Failed to save results: %v", err)

}

// Print summary

fmt.Println(result.Summary())


// Compare with baseline if available

if baseline, err := benchmarks.LoadBaseline("baseline.json"); err == nil {

    diff := result.Compare(baseline)

    fmt.Println("\nComparison with baseline:")

    fmt.Println(diff)
```

```
// Fail if significant regression

    if diff.HasRegression(0.10) { // 10% regression threshold

        os.Exit(1)
    }
}

}
```

This comprehensive testing strategy ensures TempoDB meets its functional requirements while maintaining the performance characteristics essential for a production-ready time-series database. Each milestone includes concrete verification steps that progressively build confidence in the system's correctness and robustness.

Debugging Guide

Milestone(s): All five milestones (this guide provides diagnostic techniques applicable throughout the entire TempoDB implementation journey)

Debugging a time-series database involves unique challenges due to the intersection of high-throughput writes, complex compression algorithms, temporal data semantics, and persistent storage formats. This practical guide helps learners systematically diagnose and fix common implementation bugs, organized by observable symptoms and the underlying components likely responsible. Think of debugging as **digital archaeology**—you're piecing together clues from logs, file artifacts, and runtime behavior to reconstruct what happened and identify where the implementation diverged from the design.

Common Bug Symptoms and Fixes

The following table catalogs frequently encountered issues during TempoDB development, organized by the component most likely involved. Each entry includes the observable symptom, probable root cause, diagnostic steps to confirm the hypothesis, and corrective actions to resolve the issue.

Symptom	Likely Cause	Diagnostic Steps	Corrective Action
Query returns no data for a valid time range	<p>1. Incorrect block index min/max timestamps in TSM files</p> <p>2. Misaligned time window in <code>GROUP BY time()</code> queries</p> <p>3. Missing series key in storage engine index</p>	<p>1. Use <code>inspect-tsm</code> tool to verify <code>MinTime / MaxTime</code> in block indexes match actual data</p> <p>2. Check query logs to see if predicate pushdown is skipping blocks incorrectly</p> <p>3. Verify series exists in storage engine's <code>seriesIndex</code> map</p>	<p>1. Ensure <code>TSMWriter.Finish()</code> calculates and writes correct min/max per block</p> <p>2. Debug <code>TimeRange.Contains()</code> logic for boundary conditions</p> <p>3. Check WAL recovery properly restores series index on startup</p>
Write throughput plateaus at low rate	<p>1. Synchronous WAL fsync on every write</p> <p>2. Lock contention in memtable insertion</p> <p>3. Frequent memtable flushes due to small size threshold</p>	<p>1. Monitor WAL write latency with detailed timing logs</p> <p>2. Use Go's <code>pprof</code> mutex profile to identify contended locks</p> <p>3. Check flush frequency and memtable size at flush time</p>	<p>1. Implement group commit with periodic WAL sync instead of per-write</p> <p>2. Switch to sharded memtables or lock-free data structures</p> <p>3. Increase <code>Config.MaxMemtableSize</code> or implement adaptive sizing</p>
Data points appear duplicated in query results	<p>1. WAL replay during recovery re-applying already flushed points</p> <p>2. Compaction merging incorrectly handling overlapping time ranges</p>	<p>1. Check WAL segment cleanup logic—are segments deleted after flush?</p> <p>2. Inspect compaction logs for duplicate point detection</p> <p>3. Verify <code>Memtable.Flush()</code> returns data AND clears the internal map</p>	<p>1. Implement WAL segment rotation with precise tracking of flushed offsets</p> <p>2. Add deduplication in <code>mergeSeriesPoints()</code> using point timestamps</p> <p>3. Ensure flush creates new memtable reference instead of modifying in-place</p>

Symptom	Likely Cause	Diagnostic Steps	Corrective Action
	3. Memtable flush not clearing flushed data properly		
Storage engine crashes on startup with "invalid magic number"	1. TSM file corruption from incomplete writes during crashes 2. Version mismatch between reader and writer 3. Memory mapping failure due to file truncation	1. Use hex dump to inspect first 4 bytes of corrupted file 2. Check <code>TSMHeader.Version</code> written vs. expected 3. Verify file size matches expected based on index	1. Implement write-temp-then-rename pattern for TSM file creation 2. Add version compatibility check in <code>OpenTSMReader()</code> 3. Use file locking during writes or detect incomplete files via footer
Out-of-order points are silently dropped	1. Tolerance window too small in memtable configuration 2. Timestamp comparison logic ignoring equal timestamps 3. Flush logic not checking for late arrivals during flush	1. Log dropped points with timestamps to see pattern 2. Test <code>Memtable.Insert()</code> with points at same timestamp 3. Check if flush includes points arriving during flush operation	1. Increase <code>Memtable.maxOutOfOrderWindow</code> based on data source characteristics 2. Fix comparison to handle <code><=</code> for same timestamp updates 3. Implement double-buffering or point redirection during flush
Aggregation results are mathematically incorrect	1. Float precision accumulation errors in sum/average 2. Window	1. Test with integer values to isolate float issues 2. Debug <code>alignToWindow()</code> with known timestamps	1. Use Kahan summation for <code>AggregateSum</code> of floats 2. Ensure window alignment uses fixed epoch (e.g., Unix zero) 3. Implement <code>isValidFloat()</code> checks before aggregation

Symptom	Likely Cause	Diagnostic Steps	Corrective Action
	alignment incorrectly offset 3. Missing values (NaN/Inf) not handled in aggregates	3. Check if aggregation functions skip invalid values	
Memory usage grows unbounded during queries	1. Iterator not closing resources 2. Result caching without size limits 3. Large time ranges loading all points into memory	1. Use Go's <code>runtime.ReadMemStats</code> to track allocation patterns 2. Check for missing <code>SeriesScanner.Close()</code> calls 3. Monitor query plan for full scans vs. predicate pushdown	1. Implement <code>io.Closer</code> on all iterators with defer cleanup 2. Add LRU cache with byte-size limits for query results 3. Implement chunked streaming in <code>StreamWriter</code>
Compaction causes write stalls	1. Synchronous I/O during compaction blocking writes 2. Too aggressive compaction triggered by small thresholds 3. Disk space exhaustion during merge operation	1. Monitor compaction duration and I/O patterns 2. Check compaction trigger conditions and frequency 3. Verify <code>DiskMonitor</code> is detecting low disk space	1. Move compaction to background with rate limiting 2. Adjust <code>LevelConfig</code> thresholds based on write volume 3. Implement emergency compaction with reserved space
Prometheus remote write fails with 400 Bad Request	1. Protocol buffer parsing errors 2. Timestamp conversion issues (ms vs	1. Log raw request bytes for debugging 2. Compare timestamp formats in parsed data 3. Check series key string	1. Validate protocol buffer structure before processing 2. Convert Prometheus timestamps (ms) to internal format (ns) 3. Sort labels alphabetically in <code>SeriesKey.String()</code>

Symptom	Likely Cause	Diagnostic Steps	Corrective Action
	ns) 3. Label sorting mismatch with Prometheus conventions	generation matches Prometheus format	
Database becomes unresponsive after running for days	1. Memory leak in goroutines or cached resources 2. File descriptor exhaustion from open TSM files 3. Background job pile-up without throttling	1. Use <code>pprof heap</code> profile to identify leak sources 2. Check <code>lsof</code> output for open file counts 3. Monitor scheduler job completion times	1. Add context timeouts to all background operations 2. Implement LRU for <code>TSMReader</code> instances with file closing 3. Add backpressure to job scheduling based on system load

Special Considerations for Time-Series Data

Time-series databases introduce unique edge cases that require specific debugging approaches:

Clock Skew Between Ingest Sources

When points arrive from distributed systems with unsynchronized clocks, timestamps may jump backwards beyond the tolerance window, causing points to be rejected or stored incorrectly.

Diagnosis: Log point timestamps with source identifiers; monitor for temporal anomalies. *Fix:* Implement client-side timestamp validation or server-side clock correction buffers.

Gaps in Time Series

Natural periods without data (e.g., sensor offline) can cause aggregation windows to produce misleading results or query planning to over-optimize.

Diagnosis: Check aggregation results for windows with zero points vs. windows with null results. *Fix:* Distinguish between "no data" and "zero value" in aggregation functions; consider interpolation for specific use

cases.

Extreme Cardinality Explosion

A malformed tag value (like user ID in a tag) can create millions of series, overwhelming memory and storage indexes.

Diagnosis: Monitor `seriesIndex` growth rate; alert on sudden cardinality changes. *Fix:* Implement series creation rate limiting; validate tag keys/values against schema.

Debugging Techniques and Tools

Effective debugging requires both systematic approaches and specialized tooling. Below are techniques tailored to TempoDB's architecture, progressing from simple logging to advanced instrumentation.

1. Structured Logging with Component Context

Mental Model: Think of logging as adding **breadcrumb trails** through each component—each log entry should let you trace a point's journey from ingestion to storage to query response.

Implement a structured logging system that includes:

- **Component identifier** (`storage`, `query`, `compaction`)
- **Operation context** (e.g., `series_key="cpu"`, `range="2023-10-01T00:00:00Z/2023-10-01T01:00:00Z"`)
- **Performance timing** for slow operations
- **Error chains** with causalities

Example Log Output Analysis:

```
level=DEBUG ts=2023-10-01T12:00:00Z component=storage operation=WritePoint
series=cpu.usages tags="host:web01" timestamp=16961616000000000000 value=42.5 duration=2ms
level=INFO ts=2023-10-01T12:00:05Z component=memtable operation=Flush series_count=1524
point_count=1258476 size_mb=48 duration=450ms
level=WARN ts=2023-10-01T12:00:10Z component=query operation=ExecuteRangeScan
message="skipping block outside time range" file=00001.tsm series=cpu.usages
block_min=16961652000000000000 block_max=16961688000000000000 query_min=16961616000000000000
```

Implementation Strategy:

- Use a logging interface that supports structured fields (like `zap` or `slog`)
- Add trace IDs to correlate operations across components
- Implement log levels that can be dynamically adjusted per component

2. File Inspection Utilities

Build command-line tools to examine internal file formats—these are invaluable for understanding disk state independent of the running database.

TSM File Inspector (`inspect-tsm`):

```
$ go run cmd/inspect-tsm/main.go --file /data/00001.tsm --detail
File: /data/00001.tsm
Magic: 0x16D1D1A5 ✓
Version: 1 ✓
Size: 45.2 MB
Series Count: 284
Blocks: 1,247

Series: cpu.usages{host=web01,region=us-west}
  Block 0: Offset=1024, Size=16.5KB, Points=1024, MinTime=2023-10-01T00:00:00Z,
  MaxTime=2023-10-01T00:17:04Z
  Block 1: Offset=17984, Size=15.8KB, Points=1024, MinTime=2023-10-01T00:17:04Z,
  MaxTime=2023-10-01T00:34:08Z
  ...
Compression:
  Timestamps: Delta-of-delta, 1024 points → 2.1KB (85% reduction)
  Values: Gorilla XOR, 1024 points → 8.2KB (50% reduction)
```

WAL Segment Inspector (`inspect-wal`):

```
$ go run cmd/inspect-wal/main.go --segment /wal/00000001.wal --limit 10
Segment: /wal/00000001.wal (FirstID=1, LastID=8427)
Entry 1: Type=WritePoint, Size=142B, CRC=0xA3F1C8D2 ✓
  Series: temperature{sensor=thermo01}
  Point: 2023-10-01T12:00:00Z, 22.5°C
Entry 2: Type=WritePoint, Size=138B, CRC=0xB8A2D4E1 ✓
  Series: temperature{sensor=thermo02}
  Point: 2023-10-01T12:00:00Z, 23.1°C
...
Entry 8427: Type=SeriesCreate, Size=89B, CRC=0x9C3F2A1D ✓
  Series: pressure{sensor=baro01, unit=hPa}
```

Block-Level Debug Tool (`decode-block`): For deep inspection of compression artifacts:

```
$ go run cmd/decode-block/main.go --file /data/00001.tsm --series "cpu.usages" --block 0
Block 0 for cpu.usages{host=web01}:
Raw Header: MinTime=16961616000000000000, MaxTime=16961626240000000000
Decompressed Points (first 10):
16961616000000000000 → 42.5
16961616010000000000 → 42.7 (delta: +0.2)
16961616020000000000 → 43.1 (delta: +0.4)
16961616030000000000 → 42.9 (delta: -0.2)
Compression Artifacts:
Timestamp bytes: 2104 (delta-of-delta with first=0, second=1e9)
Value bytes: 8192 (Gorilla XOR with leading zero count=12)
```

3. Performance Profiling with Go's pprof

Go's built-in profiling tools are essential for identifying bottlenecks in a high-performance database.

CPU Profiling:

```
# Add to your HTTP server                                BASH

import _ "net/http/pprof"

# Profile during write load

go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30

# Generate flame graph

go tool pprof -http=:8080 /tmp/profile.out
```

Common patterns to look for:

- **High `runtime.mallocgc` time** indicates excessive allocation—optimize by reusing byte slices, pooling objects
- **Contended `sync.RWMutex` operations** show as flat lines in mutex profile—consider sharding or lock-free structures
- **Excessive `runtime.growslice`** suggests slices growing repeatedly—pre-allocate with known capacities

Heap Memory Profiling:

```
# Capture heap snapshot during query execution  
  
go tool pprof http://localhost:6060/debug/pprof/heap  
  
# Compare two heap profiles  
  
go tool pprof -base heap1.pb.gz -top heap2.pb.gz
```

BASH

Goroutine Leak Detection:

```
# Get goroutine dump  
  
curl http://localhost:6060/debug/pprof/goroutine?debug=2 > goroutines.txt  
  
# Look for growing goroutine counts in metrics  
  
# Common leak sources:  
  
# - Unbuffered channels with blocked senders/receivers  
# - Contexts not being canceled  
# - Background jobs not completing
```

BASH

4. Property-Based Testing for Compression Algorithms

Compression bugs often manifest as silent data corruption—points decompress to wrong values. Property-based testing verifies invariants hold across random inputs.

Round-Trip Invariant:

For any slice of timestamps and values, compress → decompress should yield identical data.

Compression Ratio Bounds:

Compressed size should never exceed uncompressed size plus header overhead.

Monotonic Timestamp Preservation:

If input timestamps are sorted, decompressed timestamps should maintain same order.

Implementation Approach:

GO

```
// TestDeltaDeltaRoundTrip uses quick.Check to verify compression properties

func TestDeltaDeltaRoundTrip(t *testing.T) {

    property := func(timestamps []uint64) bool {

        if len(timestamps) < 2 { return true }

        compressed, err := compressTimestamps(timestamps)

        if err != nil { return false }

        decompressed, err := decompressTimestamps(compressed, len(timestamps))

        if err != nil { return false }

        return slices.Equal(timestamps, decompressed)
    }

    config := &quick.Config{
        MaxCount: 1000,

        Values: func(values []reflect.Value, rand *rand.Rand) {

            // Generate random but sorted timestamps
        },
    }

    if err := quick.Check(property, config); err != nil {

        t.Errorf("Round-trip failed: %v", err)
    }
}
```

5. Golden File Testing for Format Stability

Golden files capture correct output for known inputs, detecting unintended format changes across versions.

TSM Format Golden Files:

```
testdata/golden/v1/
├── simple.tsm.golden      # Single series, 1000 points
├── multi_series.tsm.golden # 10 series, mixed timestamps
└── edge_cases.tsm.golden  # NaN, Inf, large timestamp jumps
```

Usage Pattern:

```
# Run tests normally (compares to golden files)                                BASH
go test ./internal/storage/tsm/...

# Update golden files after intentional format change
UPDATE_GOLDEN=1 go test ./internal/storage/tsm/...
```

Golden File Contents:

```
== FILE: simple.tsm.golden ==
Magic: 0x16D1D1A5
Version: 1
Series: 1
Total Blocks: 1
Total Points: 1000
Checksum: 0xA1B2C3D4

Series: test{tag=value}
  Block 0:
    Offset: 1024
    Size: 16384
    Points: 1000
    MinTime: 16094592000000000000 (2021-01-01T00:00:00Z)
    MaxTime: 16094592010000000000 (2021-01-01T00:16:40Z)
    CRC: 0xE5F6A7B8
```

6. Time-Travel Debugging with Record & Replay

For intermittent bugs, record operations and replay them in a controlled environment.

Operation Recorder:

GO

```

type OperationRecorder struct {

    mu sync.Mutex

    ops []LoggedOperation
}

type LoggedOperation struct {

    Timestamp time.Time

    Type      string // "WritePoint", "Query", "Compact"

    Args      []byte // Serialized arguments

    Result    []byte // Serialized result/error
}

```

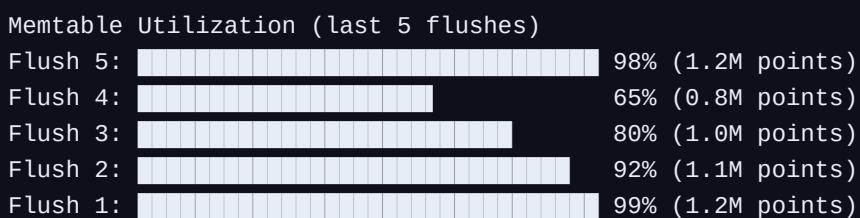
Replay Workflow:

1. Enable recording during production-like load
2. Capture bug occurrence with full operation trace
3. Replay trace in test environment with:
 - Added debug logging
 - Race detector enabled
 - Stress GOMAXPROCS variations

7. Visualization of Internal State

Create visual representations of database state to identify patterns.

Memtable Heat Map:



TSM File Age Distribution:

TSM Files by Age (hours)	
0-1:	22 files
1-2:	14 files
2-4:	18 files
4-8:	10 files
8-16:	5 files
16-32:	1 file
>32:	6 files (candidates for compaction)

8. Stress Testing with Anomaly Injection

Deliberately introduce failures to verify recovery mechanisms.

Controlled Chaos Patterns:

1. **Random process kills** during write operations
2. **Disk full simulation** by intercepting write calls
3. **Clock skew injection** by mocking time functions
4. **Network partition simulation** for API endpoints

Recovery Verification Checklist:

- After crash, WAL replay restores all acknowledged writes
- Partial TSM files are detected and quarantined
- Series index reconstructs correctly from TSM files
- Compaction resumes from interrupted state
- Query results remain consistent pre/post recovery

Implementation Guidance

This implementation guidance provides concrete tools and techniques for debugging TempoDB. While the main design avoids code, this section bridges to implementation with working debugging utilities.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Logging	Go's <code>log/slog</code> with structured fields	<code>zap</code> or <code>zerolog</code> with context propagation and sampling
Profiling	Built-in <code>net/http/pprof</code> endpoints	Custom profiling events with <code>expvar</code> and Prometheus metrics
File Inspection	Standalone CLI tools with direct file reading	Integrated HTTP endpoints for live file inspection
Tracing	Manual trace IDs in logs	OpenTelemetry with Jaeger or Grafana Tempo
Testing	<code>testing</code> package with golden files	Property-based testing with <code>testing/quick</code> , fuzzing with <code>go test -fuzz</code>

B. Recommended File/Module Structure

```

tempo/
├── cmd/
│   ├── server/                      # Main database server
│   ├── inspect-tsm/                 # TSM file inspection tool
│   ├── inspect-wal/                 # WAL segment inspection tool
│   └── decode-block/                # Block-level debug tool
├── internal/
│   ├── debug/                       # Debugging utilities
│   │   ├── inspector.go            # File format inspection interfaces
│   │   ├── recorder.go             # Operation record/replay
│   │   └── visualizer.go          # State visualization helpers
│   ├── storage/
│   │   └── tsm/
│   │       └── debug.go           # TSM-specific debugging functions
│   └── wal/
│       └── debug.go               # WAL-specific debugging functions
└── testdata/
    ├── golden/                      # Golden file directory
    │   ├── v1/                        # Version-specific golden files
    │   └── current/                  # Symlink to current version
    └── fixtures/                    # Test data fixtures
└── tools/
    ├── generate-testdata/          # Generate test datasets
    └── profile-analyzer/           # Custom profile analysis

```

C. Infrastructure Starter Code: TSM File Inspector

```
// cmd/inspect-tsm/main.go                                GO

package main

import (
    "encoding/binary"
    "fmt"
    "os"
    "path/filepath"

    "tempo/internal/storage/tsm"
)

func main() {
    if len(os.Args) < 2 {
        fmt.Println("Usage: inspect-tsm <tsm-file> [--detail]")
        os.Exit(1)
    }

    filePath := os.Args[1]
    detailed := len(os.Args) > 2 && os.Args[2] == "--detail"

    // Open TSM file using the same reader as the database
    reader, err := tsm.OpenTSMReader(filePath)
    if err != nil {
        fmt.Printf("Failed to open TSM file: %v\n", err)
        os.Exit(1)
    }
}
```

```
defer reader.Close()

// Read and display header

data, _ := os.ReadFile(filePath)

if len(data) < 12 {

    fmt.Println("File too small to be valid TSM")

    return

}

magic := binary.BigEndian.Uint32(data[0:4])

version := binary.BigEndian.Uint64(data[4:12])


fmt.Printf("File: %s\n", filepath.Base(filePath))

fmt.Printf("Magic: 0x%X ", magic)

if magic == tsm.MagicNumber {

    fmt.Printf("\n")

} else {

    fmt.Printf("X (expected 0x%X)\n", tsm.MagicNumber)

}

fmt.Printf("Version: %d ", version)

if version == tsm.Version {

    fmt.Printf("\n")

} else {

    fmt.Printf("X (expected %d)\n", tsm.Version)

}

fi, _ := os.Stat(filePath)
```

```
fmt.Printf("Size: %.1f MB\n", float64(fi.Size())/1024/1024)

// Get index to count series and blocks

index := reader.Index()

seriesCount := len(index.Entries)

blockCount := 0

for _, entries := range index.Entries {

    blockCount += len(entries)

}

fmt.Printf("Series Count: %d\n", seriesCount)

fmt.Printf("Blocks: %d\n", blockCount)

fmt.Println()

if detailed {

    // Display detailed per-series information

    for seriesKey, entries := range index.Entries {

        fmt.Printf("Series: %s\n", seriesKey)

        for i, entry := range entries {

            fmt.Printf("  Block %d: Offset=%d, Size=%d, Points~%d, ",
                      i, entry.Offset, entry.Size, entry.Size/16) // Approximation

            // Read block header for exact min/max

            if len(data) > int(entry.Offset)+16 {

                minTime := binary.BigEndian.Uint64(data[entry.Offset:entry.Offset+8])

                maxTime :=
binary.BigEndian.Uint64(data[entry.Offset+8:entry.Offset+16])

```

```
    fmt.Printf("MinTime=%d, MaxTime=%d\n", minTime, maxTime)

} else {

    fmt.Printf("[header out of bounds]\n")

}

}

}

}
```

D. Core Logic Skeleton: Operation Recorder

```
// internal/debug/recorder.go                                GO

package debug

import (
    "encoding/json"
    "os"
    "sync"
    "time"
)

// OperationRecorder captures operations for later replay debugging

type OperationRecorder struct {
    mu      sync.Mutex
    file   *os.File
    encoder *json.Encoder
    enabled bool
}

type RecordedOperation struct {
    ID          string      `json:"id"`
    Timestamp   time.Time   `json:"timestamp"`
    Type        string      `json:"type"`
    Component   string      `json:"component"`
    Arguments   json.RawMessage `json:"arguments"`
    Result     json.RawMessage `json:"result,omitempty"`
    Error      string      `json:"error,omitempty"`
    Duration   time.Duration `json:"duration_ms"`
}
```

```
}

// NewOperationRecorder creates a new recorder writing to the given file

func NewOperationRecorder(filePath string) (*OperationRecorder, error) {

    // TODO 1: Open file for appending with os.OpenFile (O_CREATE|O_APPEND|O_WRONLY)

    // TODO 2: Create json.Encoder that writes to the file

    // TODO 3: Write array start token "["

    // TODO 4: Return recorder with file and encoder initialized

    return nil, nil

}

// Record starts timing an operation and returns a function to complete the recording

func (r *OperationRecorder) Record(opType, component string, args interface{}) func(result interface{}, err error) {

    start := time.Now()

    opID := generateOperationID()

    // Serialize arguments

    var argsJSON json.RawMessage

    if args != nil {

        // TODO 5: Marshal args to JSON using json.Marshal

    }

    return func(result interface{}, err error) {

        if !r.enabled {

            return

        }

    }

}
```

```
r.mu.Lock()

defer r.mu.Unlock()

// Create operation record

record := RecordedOperation{
    ID:          opID,
    Timestamp:  start,
    Type:        opType,
    Component:   component,
    Arguments:   argsJSON,
    Duration:    time.Since(start),
}

// TODO 6: If err != nil, set record.Error to err.Error()

// TODO 7: If result != nil, marshal result to JSON for record.Result

// TODO 8: Encode record to JSON file using r.encoder.Encode()

// TODO 9: Flush file to ensure write reaches disk

}

}

// ReplayOperations reads recorded operations and executes them through a handler

func ReplayOperations(filePath string, handler func(op RecordedOperation) error) error {

    // TODO 10: Open file for reading

    // TODO 11: Read array start token

    // TODO 12: Create json.Decoder and decode operations until EOF

    // TODO 13: For each operation, call handler with the operation

    // TODO 14: Return any error from handler or decoding
```

```

    return nil
}

// generateOperationID creates a unique ID for each operation

func generateOperationID() string {
    // TODO 15: Implement ID generation (e.g., nanosecond timestamp + random suffix)
    return ""
}

```

E. Go-Specific Debugging Hints

1. Use `runtime.ReadMemStats` for Memory Insights:

```

var m runtime.MemStats

runtime.ReadMemStats(&m)

fmt.Printf("Alloc=%v MB, TotalAlloc=%v MB, Sys=%v MB, NumGC=%v\n",
    m.Alloc/1024/1024, m.TotalAlloc/1024/1024, m.Sys/1024/1024, m.NumGC)

```

GO

2. Debug Goroutine Leaks with Stack Dumps:

```

go func() {
    for range time.Tick(5 * time.Minute) {

        buf := make([]byte, 1024*1024)

        n := runtime.Stack(buf, true)

        if n > 0 && strings.Count(string(buf[:n]), "tempo/internal") > 500 {

            log.Warn("High goroutine count", "stacks", string(buf[:n]))
        }
    }
}()

```

GO

3. Profile Production Safely:

```
// Secure pprof endpoint with auth  
  
mux := http.NewServeMux()  
  
mux.HandleFunc("/debug/pprof/", authMiddleware(pprof.Index))
```

GO

4. Use `sync.Once` for Expensive Debug Setup:

```
var debugSetupOnce sync.Once  
  
func setupDebugging() {  
  
    debugSetupOnce.Do(func() {  
  
        // Initialize debug endpoints, recorders, etc.  
  
    })  
  
}
```

GO

F. Milestone Debugging Checkpoints

Milestone 1 (Storage Engine):

- Run: `go run cmd/inspect-tsm/main.go testdata/fixtures/simple.tsm`
- Expected: Shows valid magic number, version, and at least one series with correct block offsets
- Failure Sign: "invalid magic number" indicates `WriteHeader` or file corruption bug

Milestone 2 (Write Path):

- Run: `go test ./internal/wal/... -v -run TestWALRecovery`
- Expected: Test passes showing WAL replay recovers all points after simulated crash
- Failure Sign: Points missing after recovery indicates WAL format or flush coordination bug

Milestone 3 (Query Engine):

- Run: `go test ./internal/query/... -v -run TestRangeScanWithPredicatePushdown`
- Expected: Query skips blocks outside time range (visible in debug logs)
- Failure Sign: All blocks scanned indicates predicate pushdown not working

Milestone 4 (Retention & Compaction):

- Run: `go test ./internal/compaction/... -v -run TestLevelCompaction`
- Expected: Files merge correctly, new TSM file created, old files tombstoned
- Failure Sign: Duplicate points or missing data after compaction

Milestone 5 (Query Language & API):

- Run: `curl -v "http://localhost:8080/query?q=SELECT mean(value) FROM cpu WHERE time > now() - 1h GROUP BY time(5m)"`
- Expected: Returns JSON with aggregated results, HTTP 200
- Failure Sign: Parse error or empty results with valid data indicates query parsing/planning bug

G. Debugging Tips for Specific Scenarios

Scenario: "Database returns wrong values for historical queries"

- *Diagnose:* Use `inspect-tsm` to check block min/max times match data; verify compression round-trip with `decode-block`
- *Fix:* Ensure `compressBlock` and `decompressTimestamps/Values` use same endianness; check timestamp encoding handles large deltas

Scenario: "Write throughput drops after several hours"

- *Diagnose:* Check goroutine count (leak?), open file descriptors (exhaustion?), memory fragmentation
- *Fix:* Implement `TSMReader` LRU cache with file closing; profile heap for allocation patterns

Scenario: "Aggregate SUM doesn't match manual calculation"

- *Diagnose:* Test with integer values first; check for NaN/Inf contamination; verify window alignment
- *Fix:* Implement Kahan summation in `WindowAggregator`; add validation to filter invalid floats

Scenario: "Compaction causes out of memory"

- *Diagnose:* Monitor memory during compaction; check if `mergeSeriesPoints` loads all points at once
- *Fix:* Stream merge points instead of loading all; implement memory budget for compaction operations

Future Extensions

Milestone(s): This section looks beyond the current implementation scope to explore how TempoDB could evolve with additional features and capabilities, building upon all five foundational milestones.

The current TempoDB design successfully implements a single-node, specialized time-series database with essential features for handling high-volume sequential data. However, like any production system, its capabilities can be extended to address more complex requirements and larger-scale deployments. This section explores potential enhancements that could transform TempoDB from an educational implementation into a production-ready system capable of handling enterprise workloads. These extensions represent natural evolution paths while maintaining compatibility with the core architecture established in previous sections.

Possible Enhancements

The following extensions represent meaningful next steps for TempoDB's development, ordered by their potential impact and implementation complexity. Each enhancement maintains compatibility with the existing data model and storage format, ensuring backward compatibility while expanding capabilities.

Distributed Architecture and Horizontal Scaling

Mental Model: The Highway System Expansion

Imagine TempoDB as a single major highway handling all traffic. As traffic volume grows, we need to build additional lanes (sharding) and interchanges (coordination) to distribute the load. This extension transforms TempoDB from a single highway into an interconnected highway system where data traffic is intelligently routed and balanced across multiple nodes.

The current single-node design imposes natural limits on storage capacity, write throughput, and query performance. A distributed architecture would address these limitations through:

Sharding Strategy:

Sharding Method	Partition Key	Pros	Cons	Best For
Time-based	Timestamp range	Simple to implement, temporal locality preserved	Hot shard problems, uneven distribution over time	Workloads with uniform time distribution
Series-based	Series key hash	Even distribution, linear scalability with series	Cross-series queries require fan-out	High-cardinality environments
Hybrid	Composite (time + series)	Balances both dimensions	Complex routing logic	Mixed query patterns

Coordination and Consensus: For distributed operation, TempoDB would need a coordination layer to manage cluster membership, shard placement, and failover. The system could adopt a Raft consensus implementation for metadata management while maintaining an eventually consistent model for data placement.

Decision: Series-based Sharding with Consistent Hashing

- **Context:** Need to distribute both write load and storage across multiple nodes while maintaining query efficiency.
- **Options Considered:** 1) Time-range sharding, 2) Series-key hashing, 3) Composite sharding
- **Decision:** Implement series-key based sharding using consistent hashing.
- **Rationale:** Series-based sharding provides better load balancing for high-cardinality workloads and allows parallel query execution across shards. Consistent hashing minimizes data movement when nodes join or leave the cluster.
- **Consequences:** Requires query engine to fan-out queries to multiple shards and merge results, but enables linear write scaling with series cardinality.

Implementation Components:

1. **Cluster Manager:** Tracks node membership and shard assignments using Raft consensus.
2. **Query Router:** Routes queries to appropriate shards based on series keys.
3. **Data Replicator:** Maintains configurable replication factor for durability.
4. **Hinted Handoff:** Handles writes during node failures with eventual consistency.

Metadata Expansion: The `SeriesMetadata` type would expand to include shard location and replication information:

Field	Type	Description
<code>ShardID</code>	<code>uint32</code>	Identifier for the shard containing this series
<code>ReplicaNodes</code>	<code>[]string</code>	List of nodes storing replicas
<code>PrimaryNode</code>	<code>string</code>	Current primary node for writes
<code>Version</code>	<code>uint64</code>	Version for conflict resolution in distributed writes

Support for Additional Data Types

Mental Model: The Multi-Format Warehouse

Currently, TempoDB operates like a warehouse that only stores boxes of a specific size and shape (float64 values). This extension adds specialized storage areas for different item types—some requiring temperature control (booleans), others needing careful stacking (integers), and some that are fragile and complex (strings). The warehouse now needs a more sophisticated inventory system to track what's stored where and how to handle each type.

While float64 values cover many time-series use cases, real-world applications require richer data types:

Type System Expansion:

Data Type	Storage Requirement	Compression Strategy	Query Implications
Integer (int64)	8 bytes raw	Delta encoding, run-length encoding	Enables bitwise operations, exact aggregations
Boolean	1 bit optimal	Bit packing (8 values per byte)	Enables existence queries, state tracking
String	Variable length	Dictionary encoding, Snappy compression	Enables text search, pattern matching
Histogram	Multiple float64 values	Specialized bucket encoding	Enables percentile calculations directly
Multi-value	Array of floats	Column-per-value storage	Enables vector operations, multi-metric storage

Storage Format Adaptation: The TSM format would need to extend its block structure to support type identifiers and type-specific compression:

Revised `CompressedBlock` structure:

Field	Type	Description
Type	uint8	Data type identifier (0=float64, 1=int64, etc.)
Timestamps	[]byte	Compressed timestamps (delta-of-delta)
Values	[]byte	Type-specific compressed values
TypeMetadata	[]byte	Optional type-specific metadata (e.g., dictionary for strings)
Checksum	uint32	CRC32 checksum for integrity

Query Language Extensions: The query language would need type-aware functions and operations:

```
-- Type-specific aggregations

SELECT percentile(field, 95) FROM measurements WHERE time > now() - 1h

-- String operations

SELECT field FROM logs WHERE field LIKE '%error%'

-- Multi-value operations

SELECT vector_magnitude(vector_field) FROM sensor_data
```

SQL

Type Conversion and Coercion: A comprehensive type system requires clear rules for implicit and explicit type conversions, particularly when performing operations across different types or when querying fields that have changed type over time.

Continuous Queries and Materialized Views

Mental Model: The Automated Factory Assembly Line

Imagine data flowing through TempoDB like parts on a factory conveyor belt. Currently, workers (queries) manually inspect and process these parts when requested. Continuous queries act as automated robotic arms that process parts as they arrive, creating pre-assembled components (materialized views) that are ready for immediate use when customers ask for them.

Continuous queries automatically execute queries at regular intervals and store the results, providing significant performance benefits for frequently accessed aggregates:

Architecture Components:

1. **Continuous Query Scheduler:** Manages execution of registered continuous queries.
2. **Incremental Computation Engine:** Efficiently updates aggregates as new data arrives.
3. **Materialized View Storage:** Specialized storage for pre-computed results.
4. **Query Rewriter:** Automatically redirects queries to use materialized views when possible.

Continuous Query Definition:

Field	Type	Description
Name	string	Unique identifier for the continuous query
SourceMeasurement	string	Measurement to read from
TargetMeasurement	string	Measurement to write results to
Query	string	Aggregation query to execute
Interval	time.Duration	Execution frequency
ResampleInterval	time.Duration	Optional different interval for older data
Enabled	bool	Whether the query is active

Implementation Strategy: The system would extend the `Scheduler` component to manage continuous queries as a special type of background job. Each continuous query would:

1. Track the last timestamp processed
2. Execute the aggregation query over new data since last run
3. Write results to the target measurement
4. Update metadata about what data has been processed

Query Rewriting Logic: When a query arrives, the query planner would:

1. Check if any materialized view (continuous query result) can satisfy the query
2. Determine if the materialized view has complete data for the requested time range
3. Rewrite the query to use the materialized view if appropriate
4. Fall back to raw data if materialized view is incomplete or unavailable

Tiered Storage Integration

Mental Model: The Corporate Document Archive System

Important recent documents (hot data) stay in an easily accessible desk drawer (SSD). Last quarter's documents (warm data) go to a filing cabinet in the office (HDD). Documents older than a year (cold data) get sent to offsite storage (object storage) but can be retrieved when needed. This system balances accessibility with storage cost.

Tiered storage automatically moves data between storage classes based on age and access patterns, optimizing cost-performance tradeoffs:

Storage Tier Definitions:

Tier	Storage Medium	Access Latency	Cost	Typical Data Age
Hot	Local NVMe/SSD	Microseconds	High	0-24 hours
Warm	Local HDD	Milliseconds	Medium	1-30 days
Cold	Object Storage (S3)	Seconds	Low	30+ days
Frozen	Glacier/Archive	Minutes-Hours	Very Low	365+ days

Data Movement Strategy: The system would extend the `CompactionManager` to include tier promotion/demotion logic:

Tier State Transitions:

Current Tier	Condition	Next Tier	Action Required
Hot	Data older than hot_retention	Warm	Move TSM files from SSD to HDD
Warm	Data older than warm_retention	Cold	Upload TSM files to object storage
Cold	Query accesses cold data	Warm (temporarily)	Cache retrieved data locally
Any	Data reaches TTL	Deleted	Remove from all tiers

Implementation Approach:

- Storage Abstraction Layer:** Create a unified interface for storage operations that works across local filesystem and object storage.
- Tier Metadata:** Extend `TSMFileRef` to track storage tier and location.
- Background Tier Manager:** Periodically scans files and moves them between tiers.
- Transparent Retrieval:** Automatically fetches cold data when queried, with optional caching.

Query Performance Considerations:

Queries spanning multiple tiers would need special handling:

- Hot data: Direct memory-mapped access
- Warm data: Standard disk I/O
- Cold data: Async retrieval with query timeout extensions
- Mixed-tier queries: Parallel execution with tier-aware scheduling

Advanced Compression Algorithms

Mental Model: The Specialized Packaging Department

Different products need different packaging techniques. Delicate electronics (sensor data with regular patterns) get custom-molded foam (pattern-aware compression). Dense metal parts (integer counters) get efficient stacking (run-length encoding). Irregularly shaped items (sparse metrics) get vacuum-sealed bags (sparse matrix compression). The packaging department now has specialized tools for each product type.

While Gorilla XOR and delta-of-delta provide excellent general-purpose compression, specialized algorithms can yield better results for specific data patterns:

Algorithm Selection Framework:

Data Pattern	Recommended Algorithm	Compression Ratio	CPU Cost	Implementation Complexity
Monotonic counters	Delta + Varint encoding	10-20x	Low	Low
Sparse metrics (mostly zeros)	Sparse bitmap encoding	50-100x	Medium	Medium
Regular sampling (fixed interval)	Store interval + exceptions	100x+	Low	Low
Highly correlated (sensor networks)	Chimp or Sprintz	3-5x better than Gorilla	High	High
Integer histograms	Bit-packing + RLE	8-12x	Medium	Medium

Adaptive Compression: The system could analyze data patterns at the series level and select optimal compression:

Series Compression Profile:

Field	Type	Description
<code>SeriesKey</code>	<code>string</code>	Series identifier
<code>PatternType</code>	<code>uint8</code>	Detected pattern (constant, counter, random, etc.)
<code>OptimalAlgorithm</code>	<code>uint8</code>	Recommended compression algorithm
<code>SampleEntropy</code>	<code>float64</code>	Shannon entropy of value samples
<code>TimestampRegularity</code>	<code>float64</code>	Regularity score (0=random, 1=perfectly regular)

Implementation Strategy:

1. **Pattern Detection:** Analyze first N points of a series to determine pattern
2. **Algorithm Registry:** Pluggable compression algorithm implementations
3. **Block-level Metadata:** Store algorithm ID in block headers
4. **Runtime Switching:** Support different algorithms for different blocks of same series

Real-time Streaming Analytics

Mental Model: The Live Sports Broadcast with Real-time Statistics

As the game (data stream) progresses, statisticians (streaming engine) continuously calculate player performance metrics, team statistics, and game predictions. These real-time insights appear instantly on screen (dashboard) without waiting for the game to end. The system processes data in motion rather than at rest.

Extend TempoDB from a database into a real-time analytics platform:

Stream Processing Architecture:

1. **Stream Ingestion:** Accept data from message queues (Kafka, Pulsar) in addition to HTTP API
2. **Windowing Engine:** Support tumbling, sliding, and session windows
3. **State Management:** Maintain aggregation state across window boundaries
4. **Low-latency Output:** Emit results to downstream systems or APIs

Stream Query Language Extension:

```
-- Create a streaming query                                     SQL

CREATE STREAM page_views_1m AS

SELECT

    COUNT(*) as view_count,

    AVG(duration) as avg_duration,

    WINDOW_START() as window_start,

    WINDOW_END() as window_end

FROM page_views

WINDOW TUMBLING (SIZE 1 minute)

GROUP BY user_region, page_category

EMIT CHANGES;
```

Integration Points:

- **Input Adapters:** Kafka consumer, HTTP streaming, WebSocket connections
- **Processing Pipeline:** Chain of operations (filter → transform → aggregate)
- **Output Adapters:** Write back to TempoDB, push to message queue, call webhook
- **State Storage:** RockDB or similar for window state persistence

Advanced Indexing Strategies

Mental Model: The Library's Cross-Reference System

Beyond the basic card catalog (series key index), a comprehensive library has specialized indexes: subject index (tag values), author index (source identification), citation index (value correlations), and keyword index (text search). Researchers can find information through multiple access paths.

Extend TempoDB's indexing beyond the basic series key → block mapping:

Additional Index Types:

Index Type	Structure	Use Case	Storage Overhead
Tag Value Inverted	Tag value → Series keys	Fast filtering by tag values	Medium
Value Range	Min/max values per block	Value predicate pushdown	Low
Bloom Filter	Per-series bloom filter	Series existence checks	Very Low
Full-text	Inverted index on string fields	Text search in log data	High
Correlation	Series → correlated series	Related metrics discovery	Medium

Composite Index Example - Tag Inverted Index:

Field	Type	Description
TagKey	string	Tag key (e.g., "host")
TagName	string	Tag value (e.g., "web-01")
SeriesKeys	[]string	List of series keys with this tag
LastUpdated	time.Time	When this entry was last updated

Query Optimization Impact: With advanced indexes, the query planner can:

1. Use tag inverted index to resolve series keys from tag predicates before scanning data
2. Use value range indexes to skip blocks that cannot contain matching values
3. Use bloom filters to quickly determine if a series exists in a time range
4. Combine multiple indexes for complex filter expressions

Machine Learning Integration

Mental Model: The Predictive Maintenance System

Instead of just recording when machines break, the system learns normal operating patterns and predicts future failures. It's like having an experienced mechanic who can hear a subtle engine sound and say, "That bearing will fail in 48 hours," based on patterns seen across thousands of similar machines.

Integrate ML capabilities for anomaly detection, forecasting, and pattern recognition:

Built-in ML Functions:

```
-- Anomaly detection SQL

SELECT ts, value, ANOMALY_SCORE(value) OVER (ORDER BY ts) as score

FROM metrics

WHERE time > now() - 24h


-- Forecasting

SELECT FORECAST(value, 10, '1h') as predictions

FROM metrics

WHERE time > now() - 7d

GROUP BY time(1h)


-- Pattern similarity

SELECT series_a, series_b, DTW_DISTANCE(series_a, series_b) as similarity

FROM metrics

WHERE time > now() - 1h
```

Implementation Approaches:

1. **Embedded Models:** Lightweight models (exponential smoothing, simple statistical tests) implemented natively
2. **External Integration:** Call out to ML services (TensorFlow Serving, ONNX Runtime) for complex models
3. **Model Management:** Store, version, and serve ML models alongside time-series data
4. **Feature Engineering:** Built-in functions for creating ML features from time-series

ML Pipeline Integration: Extend the storage format to include model metadata and predictions:

- Store model artifacts in a specialized measurement
- Append prediction results as derived time series
- Support online learning with incremental model updates
- Provide explainability features for model decisions

Implementation Guidance

While the full implementation of these extensions is beyond the current scope, this guidance provides starting points for developers interested in exploring these enhancements.

A. Technology Recommendations Table:

Extension	Simple Option	Advanced Option
Distributed Architecture	Hash-based sharding with static configuration	Raft consensus with dynamic rebalancing
Additional Data Types	Integer and boolean support	Full type system with pluggable codecs
Continuous Queries	Scheduled aggregation jobs	Incremental view maintenance with query rewriting
Tiered Storage	Manual tier promotion scripts	Automatic data lifecycle with S3 integration
Advanced Compression	Algorithm selection per series	Adaptive compression with runtime profiling
Streaming Analytics	Windowed aggregates on ingestion	Full streaming engine with state management
Advanced Indexing	Tag inverted index	Multiple index types with automatic selection
ML Integration	Built-in statistical functions	TensorFlow Lite integration with model serving

B. Recommended File/Module Structure:

```
tempo/
├── cmd/
│   ├── tempo-server/          # Main server (existing)
│   └── tempo-cluster/        # Cluster-aware server (new)
├── internal/
│   ├── cluster/              # Distributed coordination
│   │   ├── coordinator.go    # Cluster coordination logic
│   │   ├── sharding.go       # Shard assignment and routing
│   │   └── replication.go    # Data replication between nodes
│   ├── compression/          # Extended compression algorithms
│   │   ├── registry.go       # Algorithm registration
│   │   ├── integer.go        # Integer compression
│   │   ├── string.go         # String compression
│   │   └── adaptive.go       # Adaptive algorithm selection
│   ├── streaming/            # Real-time stream processing
│   │   ├── engine.go         # Streaming query engine
│   │   ├── windows.go        # Window implementations
│   │   └── operators.go      # Stream operators (map, filter, aggregate)
│   ├── indexing/             # Advanced indexes
│   │   ├── tag_inverted.go   # Tag value → series index
│   │   ├── value_range.go    # Value range index
│   │   └── bloom.go          # Bloom filter implementation
│   └── ml/                   # Machine learning integration
│       ├── functions.go     # Built-in ML functions
│       ├── models.go         # Model storage and serving
│       └── anomaly.go       # Anomaly detection algorithms
└── pkg/
    └── storage/tiered/       # Tiered storage abstraction
        ├── manager.go        # Tier lifecycle management
        ├── local.go           # Local filesystem backend
        └── s3.go               # S3 object storage backend
```

C. Infrastructure Starter Code - Tiered Storage Interface:

GO

```
// internal/storage/tiered/interface.go

package tiered

import (
    "context"
    "io"
    "time"
)

// StorageBackend defines the interface for different storage tiers

type StorageBackend interface {

    // Type returns the backend type identifier
    Type() string

    // Write writes data to the backend
    Write(ctx context.Context, key string, data []byte) error

    // Read reads data from the backend
    Read(ctx context.Context, key string) ([]byte, error)

    // Delete removes data from the backend
    Delete(ctx context.Context, key string) error

    // Exists checks if data exists in the backend
    Exists(ctx context.Context, key string) (bool, error)

    // List lists all keys with a given prefix
    List(ctx context.Context, prefix string) ([]string, error)
```

```
// Stats returns backend statistics

Stats(ctx context.Context) (BackendStats, error)

}

// BackendStats contains statistics for a storage backend

type BackendStats struct {

    TotalBytes int64

    UsedBytes int64

    FileCount int64

    LatencyMS float64

}

// TierManager manages data movement between tiers

type TierManager struct {

    backends map[string]StorageBackend

    policies []TierPolicy

    metadata MetadataStore

    mu sync.RWMutex

}

// TierPolicy defines when data should move between tiers

type TierPolicy struct {

    SourceTier string

    DestinationTier string

    Condition PolicyCondition

    BatchSize int

}
```

```
// PolicyCondition defines movement conditions

type PolicyCondition struct {

    AgeOlderThan     time.Duration
    AccessOlderThan  time.Duration
    SizeGreaterThanOrEqual int64
}
```

D. Core Logic Skeleton Code - Distributed Query Router:

```
// internal/cluster/router.go                                     GO

package cluster

// QueryRouter routes queries to appropriate shards

type QueryRouter struct {

    shardMap      *ShardMap

    nodeClients map[string]*NodeClient

    merger       *ResultMerger

}

// RouteQuery analyzes a query and routes it to appropriate nodes

func (r *QueryRouter) RouteQuery(ctx context.Context, query *models.Query) (*RoutedQuery, error) {

    // TODO 1: Extract series keys from the query using tag predicates

    // TODO 2: For each series key, determine which shard owns it using consistent hashing

    // TODO 3: Group series keys by shard and then by node (accounting for replication)

    // TODO 4: Build query fragments for each node, only including series keys that node
hosts

    // TODO 5: Add aggregation merging instructions for cross-shard aggregates

    // TODO 6: Return RoutedQuery with parallel execution plan

}

// ExecuteRoutedQuery executes a routed query across multiple nodes

func (r *QueryRouter) ExecuteRoutedQuery(ctx context.Context, routed *RoutedQuery) (*models.QueryResult, error) {

    // TODO 1: Launch goroutines to execute each query fragment on its target node

    // TODO 2: Collect results with proper error handling and timeout management

    // TODO 3: Merge results from different nodes, respecting the merge strategy

    // TODO 4: Apply any final aggregation that couldn't be pushed to individual nodes

    // TODO 5: Return combined result
```

```

}

// ResultMerger merges results from multiple shards

type ResultMerger struct {

    mergeStrategy MergeStrategy
}

// Merge merges multiple query results into one

func (m *ResultMerger) Merge(results []*models.QueryResult) (*models.QueryResult, error) {

    // TODO 1: Check if all results have compatible schemas

    // TODO 2: For aggregate queries, combine aggregate values appropriately

    // TODO 3: For raw data queries, concatenate and sort all points by timestamp

    // TODO 4: Handle duplicate points from replicated series

    // TODO 5: Apply limit/offset if specified in the original query

}

```

E. Language-Specific Hints:

- For distributed systems:** Use `hash/fnv` for consistent hashing, `context.Context` for request cancellation, and `errgroup` for managing parallel requests to multiple nodes.
- For additional data types:** Implement the `encoding.BinaryMarshaler` and `encoding.BinaryUnmarshaler` interfaces for custom serialization of new types.
- For tiered storage:** Use the `aws-sdk-go-v2` for S3 integration with intelligent retries and exponential backoff for failed operations.
- For streaming analytics:** Consider using `go-channels` for data flow between streaming operators, with careful buffer sizing to prevent deadlocks.
- For ML integration:** Use `gonum.org/v1/gonum` for statistical functions and `github.com/sjwhitworth/golearn` for basic machine learning algorithms.

F. Milestone Checkpoint for Distributed Extension:

To verify a basic distributed implementation:

BASH

```
# Start three nodes in a cluster

$ ./tempo-cluster --node-id=node1 --cluster-addr=:9090 --http-addr=:8080

$ ./tempo-cluster --node-id=node2 --cluster-addr=:9091 --http-addr=:8081 --
join=localhost:9090

$ ./tempo-cluster --node-id=node3 --cluster-addr=:9092 --http-addr=:8082 --
join=localhost:9090

# Write data to any node

$ curl -X POST http://localhost:8080/write \
-d 'cpu,host=server1 value=0.64'

# Query from any node (should route to correct node)

$ curl -G http://localhost:8081/query \
--data-urlencode 'q=SELECT * FROM cpu WHERE host="server1"' 

# Verify all nodes show cluster membership

$ curl http://localhost:8080/cluster/nodes

# Should return list of all three nodes
```

G. Debugging Tips for Future Extensions:

Symptom	Likely Cause	How to Diagnose	Fix
Query returns partial data in distributed mode	Some shards unreachable or returning errors	Check cluster health endpoint, examine query router logs	Ensure all nodes are healthy, implement retry logic for failed shards
Compression ratio worse for new data type	Incorrect algorithm selection for data pattern	Analyze data pattern statistics, benchmark different algorithms	Implement adaptive algorithm selection based on data characteristics
Continuous queries creating duplicate data	Race condition in incremental computation	Check last processed timestamp tracking, examine query execution logs	Add synchronization or use transactional updates for state management
Tier promotion failing silently	Insufficient permissions for object storage	Check tier manager logs, verify credentials and bucket permissions	Implement proper error reporting and retry with exponential backoff
ML functions returning inconsistent results	Model version mismatch or stale cache	Check model metadata, compare results across different calls	Implement model versioning and cache invalidation strategies

Glossary

Milestone(s): This reference section supports all five milestones by providing clear definitions of the specialized terminology used throughout the TempoDB design document.

The time-series database domain uses specialized vocabulary that may be unfamiliar to developers new to this field. This glossary provides authoritative definitions for key terms, acronyms, and concepts used throughout the TempoDB design document. Each term includes a clear definition and reference to the primary section where it's discussed in detail, creating a consistent reference point for implementation.

Terminology Reference

Term	Definition	Primary Reference
Aggregate Function	A mathematical operation applied to multiple data points to produce a single summary value, such as sum, average, minimum, maximum, or count.	Query Engine Design
Backpressure	A mechanism to throttle incoming write requests when the system is overloaded, preventing resource exhaustion and maintaining system stability.	Write Path Design
Block-Based Storage	A storage organization strategy where data is grouped into fixed-size blocks, enabling efficient I/O operations and temporal locality for range queries.	Storage Engine Design
Cardinality	The number of unique time series in a dataset, calculated as the product of unique values for each tag dimension. High cardinality can impact performance.	Write Path Design
Clock Skew	Time difference between distributed system clocks, which can cause challenges for time-series data ordering and consistency.	Error Handling and Edge Cases
Columnar Layout	A data storage format where values from the same column (e.g., all timestamps, all values) are stored contiguously rather than storing complete rows together, improving compression and scan efficiency.	Storage Engine Design
Compaction	A background process that merges multiple smaller storage files into larger, optimized files, reducing storage overhead and improving query performance.	Retention and Compaction Design
Compaction Level	A hierarchical tier in the compaction strategy where files at higher levels are larger and contain older, less volatile data.	Retention and Compaction Design
DataPoint	The fundamental unit of time-series data consisting of a timestamp and a value (<code>Timestamp time.Time, Value float64</code>).	Data Model
Delta-of-Delta Encoding	A compression technique for timestamps that stores the difference between consecutive differences, achieving high compression ratios for regularly spaced timestamps.	Storage Engine Design
Downsampling	The process of reducing data resolution through aggregation (e.g., converting 1-second data to 1-minute averages) to conserve storage space for historical data.	Retention and Compaction Design

Term	Definition	Primary Reference
Field	The actual measured value in a time series, typically a float64 numeric value, as opposed to metadata tags.	Data Model
Golden File Testing	A testing methodology that compares output against versioned reference files to ensure format stability and detect unintended changes.	Testing Strategy
Gorilla XOR Compression	A lossless compression algorithm for floating-point values that XORs consecutive values and encodes the resulting changes efficiently.	Storage Engine Design
Grace Period	A time delay between marking files for deletion (tombstoning) and physically removing them from disk, allowing for recovery from accidental deletions.	Retention and Compaction Design
Group By Time	A query operation that partitions data into fixed-width time intervals (buckets) and applies aggregation functions within each bucket.	Query Engine Design
Inverted Index	An index data structure that maps tag values to the series keys that contain them, enabling fast filtering by tag predicates.	Query Engine Design
Iterator Model	An execution pattern where each query operator implements a <code>Next()</code> method to pull data through the pipeline, enabling streaming and lazy evaluation.	Query Engine Design
Kahan Summation	An algorithm for summing floating-point numbers with reduced precision loss by maintaining a running compensation for lost low-order bits.	Future Extensions
Level-Based Compaction	A compaction strategy that organizes files into tiers (levels) with progressively larger sizes, promoting data through levels based on age and size thresholds.	Retention and Compaction Design
Line Protocol	A text-based format for writing time-series data points, consisting of measurement, tag sets, field sets, and timestamp.	Query Language and API Design
Materialized View	A pre-computed query result stored for fast access, such as rollup aggregations for historical data.	Future Extensions
Measurement	A container for related time-series data, analogous to a table name in relational databases, grouping series with the same semantic meaning.	Data Model
Memtable	An in-memory buffer that holds recently written data points before they are flushed to persistent storage, optimized for high write throughput.	Write Path Design

Term	Definition	Primary Reference
Memory-Mapped Files	A file access technique that maps file contents directly into virtual memory, enabling zero-copy reads and efficient random access.	Storage Engine Design
Out-of-Order Writes	Data points arriving with timestamps that are not in chronological order relative to previously written points for the same series.	Write Path Design
Predicate Pushdown	A query optimization technique that applies filtering conditions as early as possible in the execution pipeline, ideally at the storage layer, to reduce the amount of data processed.	Query Engine Design
Prometheus Remote Read/Write	A protocol that allows Prometheus to use external storage systems for long-term data retention, enabling integration with the Prometheus monitoring ecosystem.	Query Language and API Design
Property-Based Testing	A testing methodology that verifies properties or invariants hold for all possible inputs within a defined domain, often using randomly generated test cases.	Testing Strategy
Query Plan	An internal representation of a query that specifies the execution steps, including which files to scan, which predicates to apply, and in what order to perform operations.	Query Engine Design
Retention Policy	A rule defining how long data should be kept before automatic deletion, typically specified as a time-to-live (TTL) duration.	Retention and Compaction Design
Rollup Series	A pre-computed time series at lower granularity, created by aggregating higher-resolution data over time windows for historical query performance.	Retention and Compaction Design
Series	A collection of data points sharing the same measurement and complete set of tags, representing a single time-varying metric.	Data Model
Series Key	A unique identifier for a time series formed by concatenating a measurement name with a complete set of tag key-value pairs (<code>Measurement string, Tags map[string]string</code>).	Data Model
Shard	A logical partition of data distributed across nodes in a cluster, enabling horizontal scaling and parallel processing.	Future Extensions
Skip List	A probabilistic data structure that allows fast search, insertion, and deletion within an ordered sequence, often used for in-memory indexes.	Query Engine Design

Term	Definition	Primary Reference
Tags	Indexed key-value metadata associated with time-series data, used to identify, filter, and group series (e.g., <code>host="server1", region="us-west"</code>).	Data Model
Temporal Locality	The principle that data accessed together in time should be stored together physically, optimizing for time-range query patterns.	Storage Engine Design
Tiered Storage	A storage architecture with multiple performance/cost tiers (e.g., SSD for hot data, HDD for warm data, object storage for cold data), automatically migrating data based on access patterns.	Future Extensions
Time-Range Query	A query that retrieves all data points within a specified start and end timestamp boundary.	Query Engine Design
Time-Series Data	Sequential measurements or events indexed by time, characterized by append-heavy write patterns, time-ordered reads, and predictable value patterns.	Context and Problem Statement
Time-Structured Merge Tree (TSM)	A storage engine optimized for time-series data that organizes data into time-sorted files and merges them in the background, similar to LSM trees but with time-based partitioning.	Storage Engine Design
Time-To-Live (TTL)	The duration after which data is considered expired and eligible for automatic deletion, enforced by retention policies.	Retention and Compaction Design
Tolerance Window	The maximum allowed time difference for accepting out-of-order data points; points outside this window may be rejected or handled specially.	Write Path Design
Tombstoned	A state where a file is marked for deletion but not yet physically removed from disk, typically during a grace period.	Retention and Compaction Design
TSM File	A storage file in the Time-Structured Merge format containing compressed time-series data blocks and an index mapping series keys to block locations.	Storage Engine Design
Tumbling Windows	Contiguous, non-overlapping time intervals used for grouping operations in windowed aggregations (e.g., every complete 5-minute period).	Query Engine Design
Write Amplification	The phenomenon where the storage engine performs more physical writes than the logical writes requested by the application, often due to compaction and durability mechanisms.	Retention and Compaction Design

Term	Definition	Primary Reference
Write-Ahead Log (WAL)	A durability mechanism that logs write operations to persistent storage before acknowledging them to clients, ensuring data survival across crashes.	Write Path Design

Design Insight: Consistent terminology is critical for team alignment. This glossary serves as a single source of truth for all terms used in the TempoDB design, preventing misunderstandings during implementation. When adding new terms during development, consider updating this glossary to maintain clarity.

Implementation Guidance

While a glossary doesn't require implementation code, maintaining consistency in naming is crucial for code quality and team communication. Below are recommendations for ensuring terminology consistency throughout your Go implementation.

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Terminology Validation	Manual code review with glossary reference	Static analysis with custom linter rules
Documentation Generation	GoDoc comments with consistent terms	Automated glossary extraction from source
Naming Convention Enforcement	Team agreement and manual checks	Pre-commit hooks with naming validation

B. Recommended File/Module Structure:

Create a documentation directory to store the glossary and other reference materials:

```
project-root/
  docs/
    glossary.md          ← This glossary document
    architecture.md      ← High-level design overview
    api-reference.md     ← API documentation
  internal/
    storage/             ← Storage engine implementation
    query/               ← Query engine implementation
    wal/                 ← Write-ahead log implementation
  cmd/
    server/main.go       ← Main entry point
```

C. Terminology Consistency Checklist:

When reviewing code, verify these consistency points:

1. **Type and Field Names:** Use exact names from the NAMING CONVENTIONS section (e.g., `DataPoint`, not `DataPoint` or `Point`)
2. **Method Signatures:** Follow the exact signatures provided in the design document
3. **Error Messages:** Use consistent terminology when describing errors
4. **Comments and Documentation:** Reference terms from this glossary where appropriate
5. **Log Messages:** Use standardized terminology for operational logging

D. Go-Specific Naming Hints:

- Use `camelCase` for local variables and private fields
- Use `PascalCase` for exported types, functions, and constants
- Follow Go conventions for acronyms: `TSMFile` (not `TsmFile`), `WAL` (not `wal`)
- Use descriptive names that match glossary terms: `memtable` (not `writeBuffer`), `compactionPlan` (not `mergePlan`)

E. Debugging Terminology Mismatches:

If you encounter confusion during implementation, check:

1. **Cross-reference:** Verify all team members are using the same version of the glossary
2. **Code search:** Use `grep` or IDE search to find inconsistent usage
3. **Documentation:** Update inline comments to clarify term usage
4. **Peer review:** Include terminology checks in code review checklists

F. Maintaining the Glossary:

As the implementation evolves, you may discover new terms that need definition. Follow this process:

1. Add the term to this glossary document with a clear definition
2. Update the NAMING CONVENTIONS section if it's a core type, method, or constant
3. Communicate the change to the team
4. Update any affected code to use the new terminology consistently

Implementation Tip: Consider creating a simple validation script that scans source code for glossary terms and flags potentially inconsistent usage. This can be particularly helpful for large codebases or distributed teams.