Go Time-Series Database (TSDB) — Complete Source Code and Commentary

```
package main


import (
	"bytes"
	"context"
	"encoding/gob"
	"encoding/json"
	"fmt"
	"log"
	"math/rand"
	"os"
	"os/signal"
	"path/filepath"
	"strconv"
	"strings"
	"sync"
	"syscall"
	"time"
)


// ----------------------------
// Core Types and Data Structures
```

```
// ----------------------------

// A single data point in time-series
type Point struct {
	Timestamp time.Time
	Value     float64
}

// Series represents a labeled time-series (e.g., CPU usage of a host)
type Series struct {
	Label string
	Data  []Point
	mu    sync.RWMutex
}

// AddPoint safely adds a point to the series
func (s *Series) AddPoint(p Point) {
	s.mu.Lock()
	defer s.mu.Unlock()
	s.Data = append(s.Data, p)
}

// Query returns points in a time range
func (s *Series) Query(start, end time.Time) []Point {
	s.mu.RLock()
```

```
	defer s.mu.RUnlock()
	var result []Point
	for _, p := range s.Data {
		if !p.Timestamp.Before(start) && !p.Timestamp.After(end) {
			result = append(result, p)
		}
	}
	return result
}

// TSDB is our in-memory time-series DB
type TSDB struct {
	series map[string]*Series
	mu     sync.RWMutex
}

// NewTSDB initializes the DB
func NewTSDB() *TSDB {
	return &TSDB{
		series: make(map[string]*Series),
	}
}

// GetOrCreateSeries gets or creates a new series
func (db *TSDB) GetOrCreateSeries(label string) *Series {
```

```
	db.mu.Lock()
	defer db.mu.Unlock()
	if s, ok := db.series[label]; ok {
		return s
	}
	s := &Series{Label: label}
	db.series[label] = s
	return s
}

// QuerySeries queries a label in a time range
func (db *TSDB) QuerySeries(label string, start, end time.Time) ([]Point, error) {
	db.mu.RLock()
	defer db.mu.RUnlock()
	s, ok := db.series[label]
	if !ok {
		return nil, fmt.Errorf("series not found: %s", label)
	}
	return s.Query(start, end), nil
}

// ----------------------------
// Query Engine (Simple DSL)
// ----------------------------
```

```
// ParseQuery parses a DSL like "cpu.usage:2023-01-01T00:00:00Z~2023-01-01T23:59:59Z"
func ParseQuery(input string) (label string, start, end time.Time, err error) {
	parts := strings.Split(input, ":")
	if len(parts) != 2 {
		return "", time.Time{}, time.Time{}, fmt.Errorf("invalid format")
	}
	label = parts[0]
	times := strings.Split(parts[1], "~")
	if len(times) != 2 {
		return "", time.Time{}, time.Time{}, fmt.Errorf("invalid time range")
	}
	start, err = time.Parse(time.RFC3339, times[0])
	if err != nil {
		return "", time.Time{}, time.Time{}, err
	}
	end, err = time.Parse(time.RFC3339, times[1])
	if err != nil {
		return "", time.Time{}, time.Time{}, err
	}
	return label, start, end, nil
}

// ----------------------------
// Snapshotting and Persistence
// ----------------------------
```

```
func (db *TSDB) SnapshotToDisk(path string) error {
	db.mu.RLock()
	defer db.mu.RUnlock()

	var buf bytes.Buffer
	enc := gob.NewEncoder(&buf)
	if err := enc.Encode(db.series); err != nil {
		return err
	}
	return os.WriteFile(path, buf.Bytes(), 0644)
}

func (db *TSDB) LoadFromDisk(path string) error {
	data, err := os.ReadFile(path)
	if err != nil {
		return err
	}
	dec := gob.NewDecoder(bytes.NewReader(data))
	var s map[string]*Series
	if err := dec.Decode(&s); err != nil {
		return err
	}
	db.mu.Lock()
	db.series = s
```

```
	db.mu.Unlock()
	return nil
}

// ----------------------------
// Background Tasks: Writer, Snapshotter
// ----------------------------

func startBackgroundWriter(ctx context.Context, db *TSDB) {
	go func() {
		ticker := time.NewTicker(500 * time.Millisecond)
		defer ticker.Stop()

		for {
			select {
			case <-ctx.Done():
				log.Println("Writer stopped")
				return
			case <-ticker.C:
				now := time.Now().UTC()
				labels := []string{"cpu.usage", "mem.usage", "disk.io"}
				for _, label := range labels {
					s := db.GetOrCreateSeries(label)
					p := Point{
						Timestamp: now,
```

```
                                        Value:     rand.Float64() * 100,
                                    }
                                    s.AddPoint(p)
                                }
                            }
                        }
                    }()
}

func startSnapshotter(ctx context.Context, db *TSDB, path string) {
        go func() {
                ticker := time.NewTicker(30 * time.Second)
                defer ticker.Stop()

                for {
                        select {
                        case <-ctx.Done():
                                log.Println("Snapshotter stopped")
                                return
                        case <-ticker.C:
                                if err := db.SnapshotToDisk(path); err != nil {
                                        log.Println("snapshot failed:", err)
                                } else {
                                        log.Println("snapshot saved")
                                }
```

```
                }
        }
    }()
}

// ----------------------------
// API Layer (Simple CLI-based)
// ----------------------------

func serveCLI(ctx context.Context, db *TSDB) {
    fmt.Println("Enter queries (e.g., cpu.usage:2025-05-01T00:00:00Z~2025-05-07T00:00:00Z), or type 'exit':")
    for {
        fmt.Print("> ")
        var input string
        if _, err := fmt.Scanln(&input); err != nil {
            continue
        }
        if input == "exit" {
            return
        }
        label, start, end, err := ParseQuery(input)
        if err != nil {
            fmt.Println("error:", err)
            continue
```

```
                }
                points, err := db.QuerySeries(label, start, end)
                if err != nil {
                        fmt.Println("query error:", err)
                        continue
                }
                out, _ := json.MarshalIndent(points, "", "  ")
                fmt.Println(string(out))
        }
}

// ----------------------------
// Graceful Shutdown
// ----------------------------

func setupSignalHandler(cancelFunc context.CancelFunc, db *TSDB, snapshotPath string) {
        c := make(chan os.Signal, 1)
        signal.Notify(c, os.Interrupt, syscall.SIGTERM)
        go func() {
                <-c
                log.Println("Shutdown initiated...")
                if err := db.SnapshotToDisk(snapshotPath); err != nil {
                        log.Println("Failed to save snapshot:", err)
                }
                cancelFunc()
```

```
        }()
}

// ----------------------------
// Main Entry Point
// ----------------------------

func main() {
        snapshotPath := filepath.Join(".", "tsdb_snapshot.gob")
        db := NewTSDB()
        if err := db.LoadFromDisk(snapshotPath); err == nil {
                log.Println("snapshot restored")
        }

        ctx, cancel := context.WithCancel(context.Background())
        defer cancel()

        setupSignalHandler(cancel, db, snapshotPath)
        startBackgroundWriter(ctx, db)
        startSnapshotter(ctx, db, snapshotPath)
        serveCLI(ctx, db)

        log.Println("Goodbye!")
}
```