SIMPLIFYING BY IMPLEMENTING DISTRIBUTED SYSTEMS

Distributed Systems Simplified

Database Sharding:

Scaling writes by breaking your data into pieces.

AUTHOR:

Niv Rave

Github: https://github.com/NivRave/Distributed-Systems-Simplified

The Limits of a Single Leader

The Write-Heavy Bottleneck

The Problem: Your database hits 100% CPU. Write speed is crawling because a single machine, no matter how big - has physical limits.

While Read Replication scales your reads, every single Write still has to go through one Leader, which, as your app grows, becomes the bottleneck for all updates, inserts, and deletes.

The Scaling Wall:

- Vertical Scaling: Buying a $50,000 server just for more disk I/O is a temporary fix with a very expensive ceiling.
- Blast Radius: If your one-and-only Write node fails, your entire application becomes "Read-Only."

The Fix: **Sharding.**

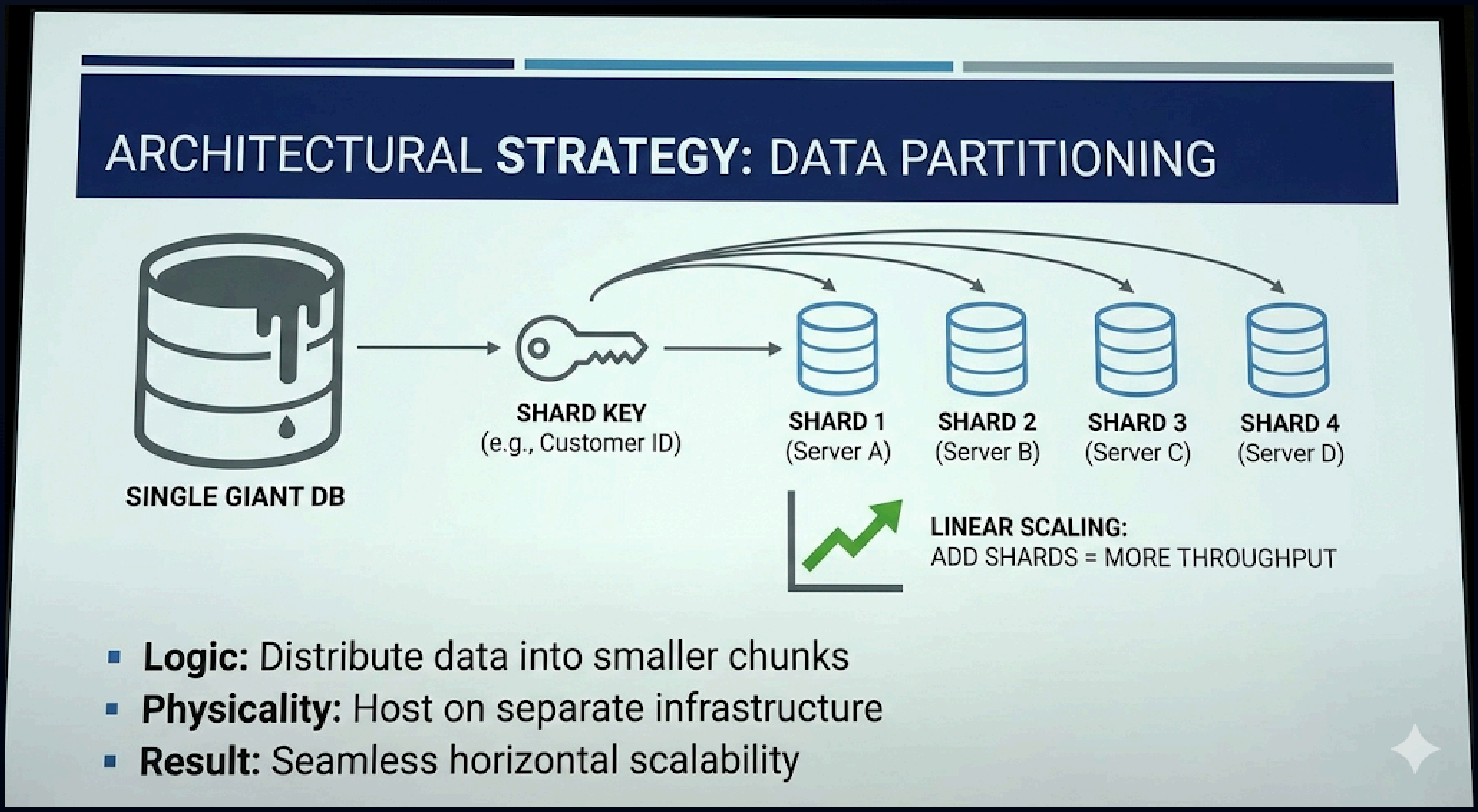
The Architectural Solution

Divide and Conquer: Sharding

Instead of one massive server, we build a Cluster:

- Instead of one massive DB, we build a **Sharded Cluster**.
- **The Logic**: Slice the dataset into chunks based on a **Shard Key**.
- **The Physicality**: Place those chunks on entirely separate, independent servers.

The Result: Your write capacity scales linearly. Need more throughput? Just add more shards.



ARCHITECTURAL **STRATEGY:** DATA PARTITIONING

- **Logic:** Distribute data into smaller chunks
- **Physicality:** Host on separate infrastructure
- **Result:** Seamless horizontal scalability

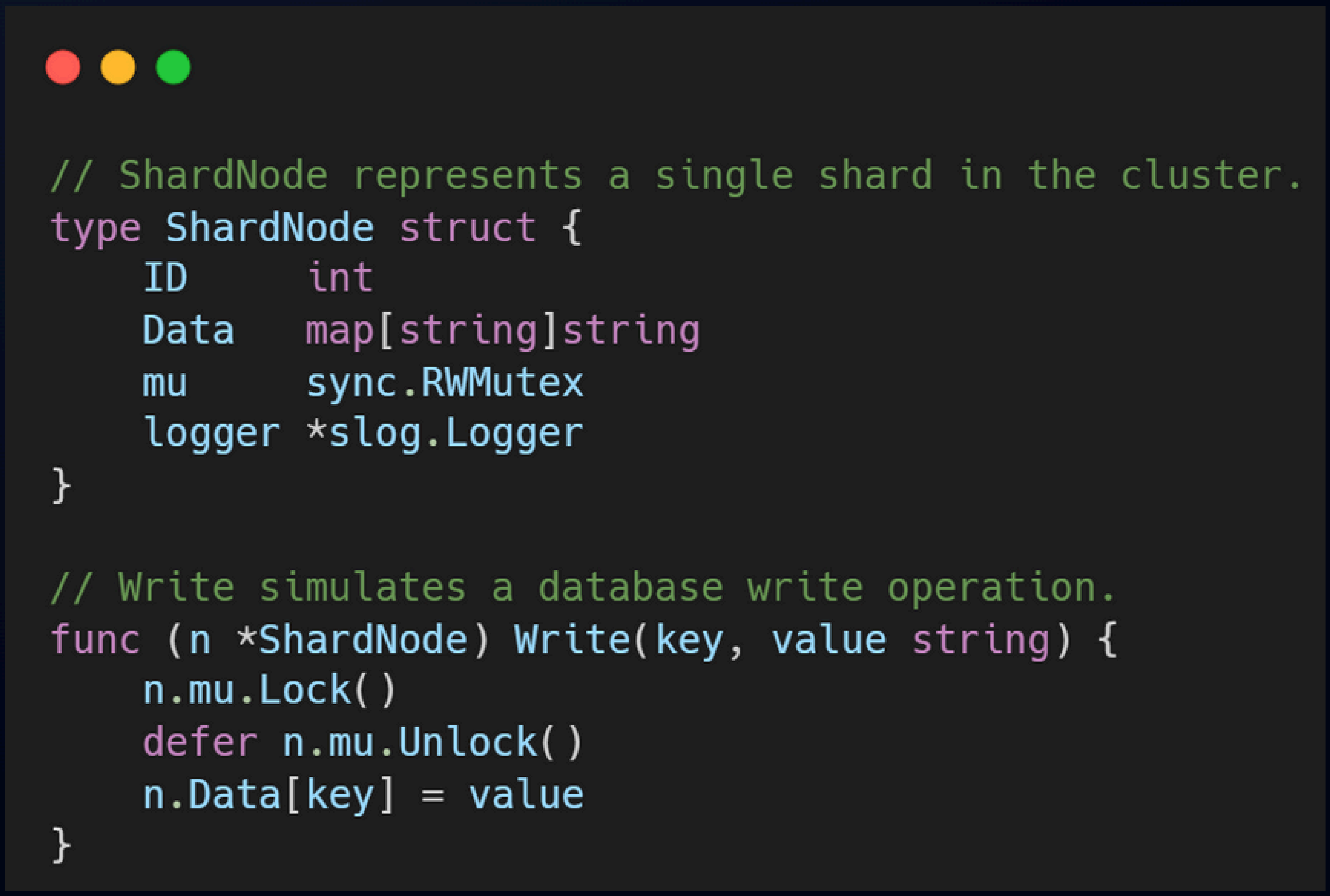
Code: The Shard

Modeling the Shard Node

We represent an individual physical shard using a Go struct.

- *sync.RWMutex*: Even within a shard, we need thread-safe access to prevent race conditions during high-volume writes.
- Independent State: Each shard has its own isolated map - it knows nothing about the data on other shards.

Below is the implementation under *node.go*:



```go
// ShardNode represents a single shard in the cluster.
type ShardNode struct {
    ID     int
    Data   map[string]string
    mu     sync.RWMutex
    logger *slog.Logger
}

// Write simulates a database write operation.
func (n *ShardNode) Write(key, value string) {
    n.mu.Lock()
    defer n.mu.Unlock()
    n.Data[key] = value
}
```

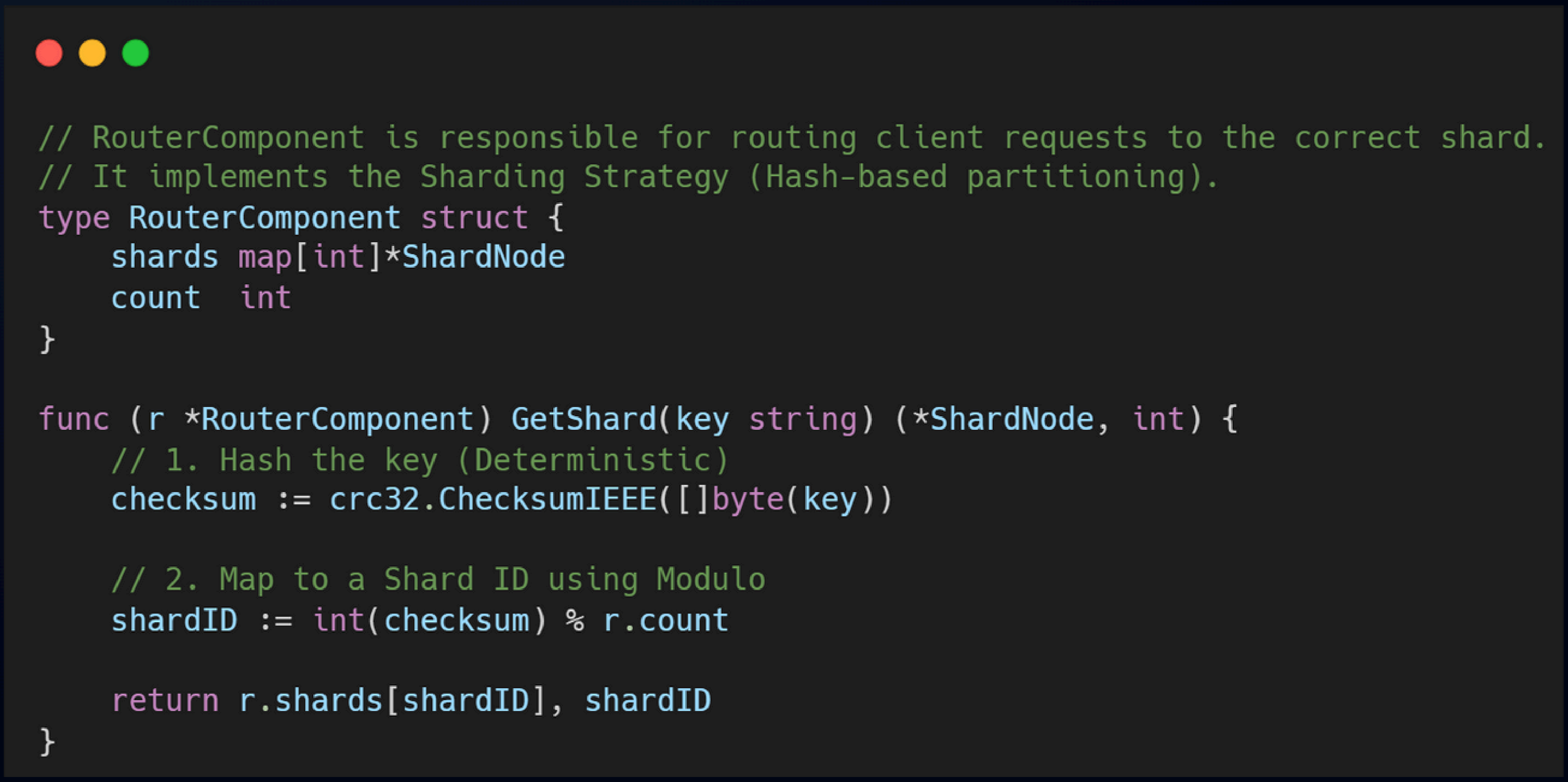
Code: The Router

The "Brain" of the Cluster

The Router is the gateway. It implements the partitioning logic to decide which shard owns a specific key.

- **Deterministic**: *user:100* must always map to the same Shard ID.
- **Hash Function**: We use *crc32* to transform a string key into a numeric hash.

Below is the implementation under *router.go*:



```go
// RouterComponent is responsible for routing client requests to the correct shard.
// It implements the Sharding Strategy (Hash-based partitioning).
type RouterComponent struct {
    shards map[int]*ShardNode
    count  int
}

func (r *RouterComponent) GetShard(key string) (*ShardNode, int) {
    // 1. Hash the key (Deterministic)
    checksum := crc32.ChecksumIEEE([]byte(key))

    // 2. Map to a Shard ID using Modulo
    shardID := int(checksum) % r.count

    return r.shards[shardID], shardID
}
```

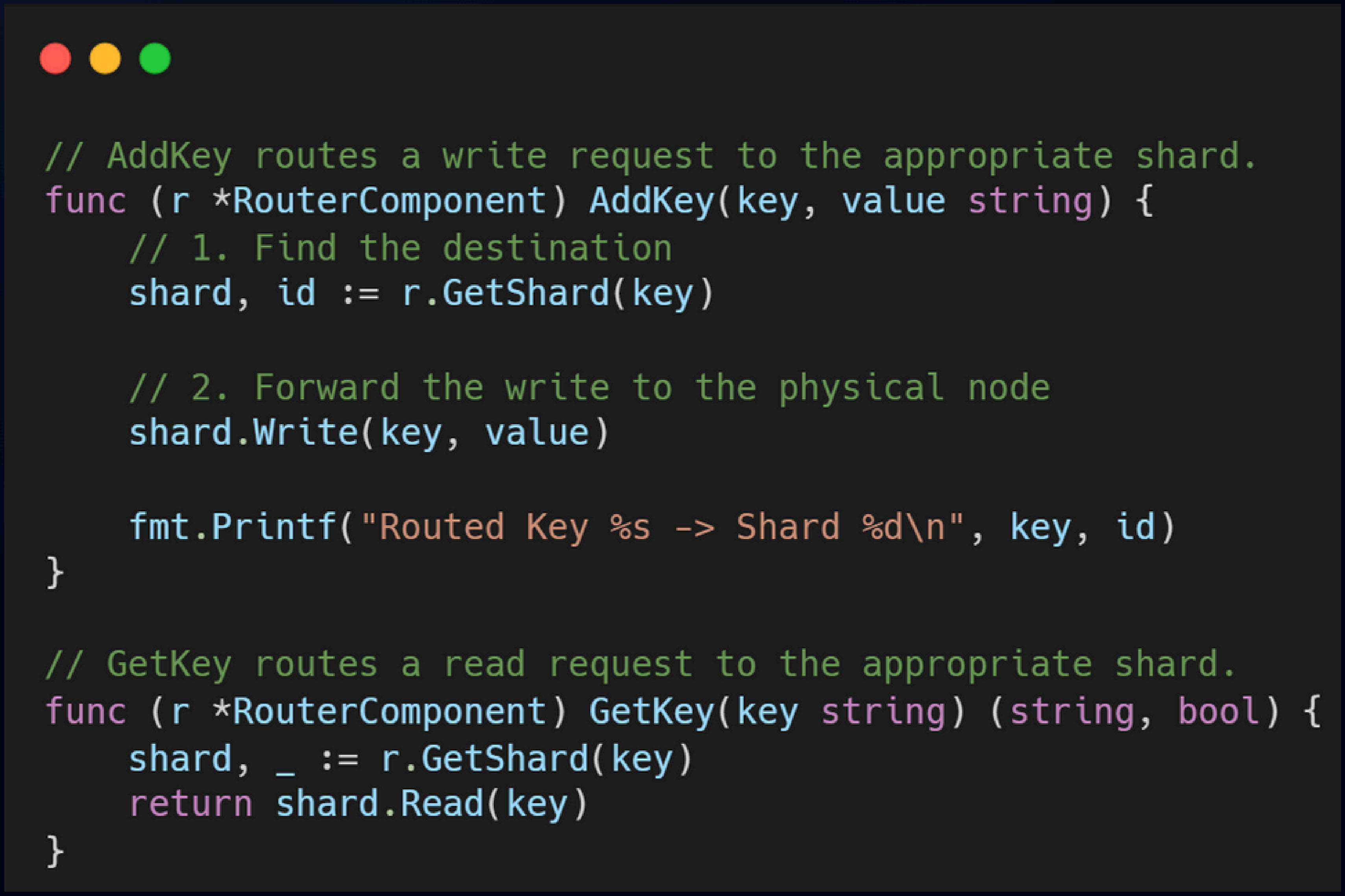
Code: Just-In-Time Routing

Transparent Scaling

The client doesn't need to know which shard holds their data. They talk to the Router, and the Router forwards the request.

- **Proxy Pattern**: The router calculates the shard ID and immediately calls that shard's *Write* method.
- **Linear Growth**: As you add more shards, the router simply updates its *count* to distribute the load across more nodes.

Below is the implementation under *router.go*:



```go
// AddKey routes a write request to the appropriate shard.
func (r *RouterComponent) AddKey(key, value string) {
    // 1. Find the destination
    shard, id := r.GetShard(key)

    // 2. Forward the write to the physical node
    shard.Write(key, value)

    fmt.Printf("Routed Key %s -> Shard %d\n", key, id)
}

// GetKey routes a read request to the appropriate shard.
func (r *RouterComponent) GetKey(key string) (string, bool) {
    shard, _ := r.GetShard(key)
    return shard.Read(key)
}
```

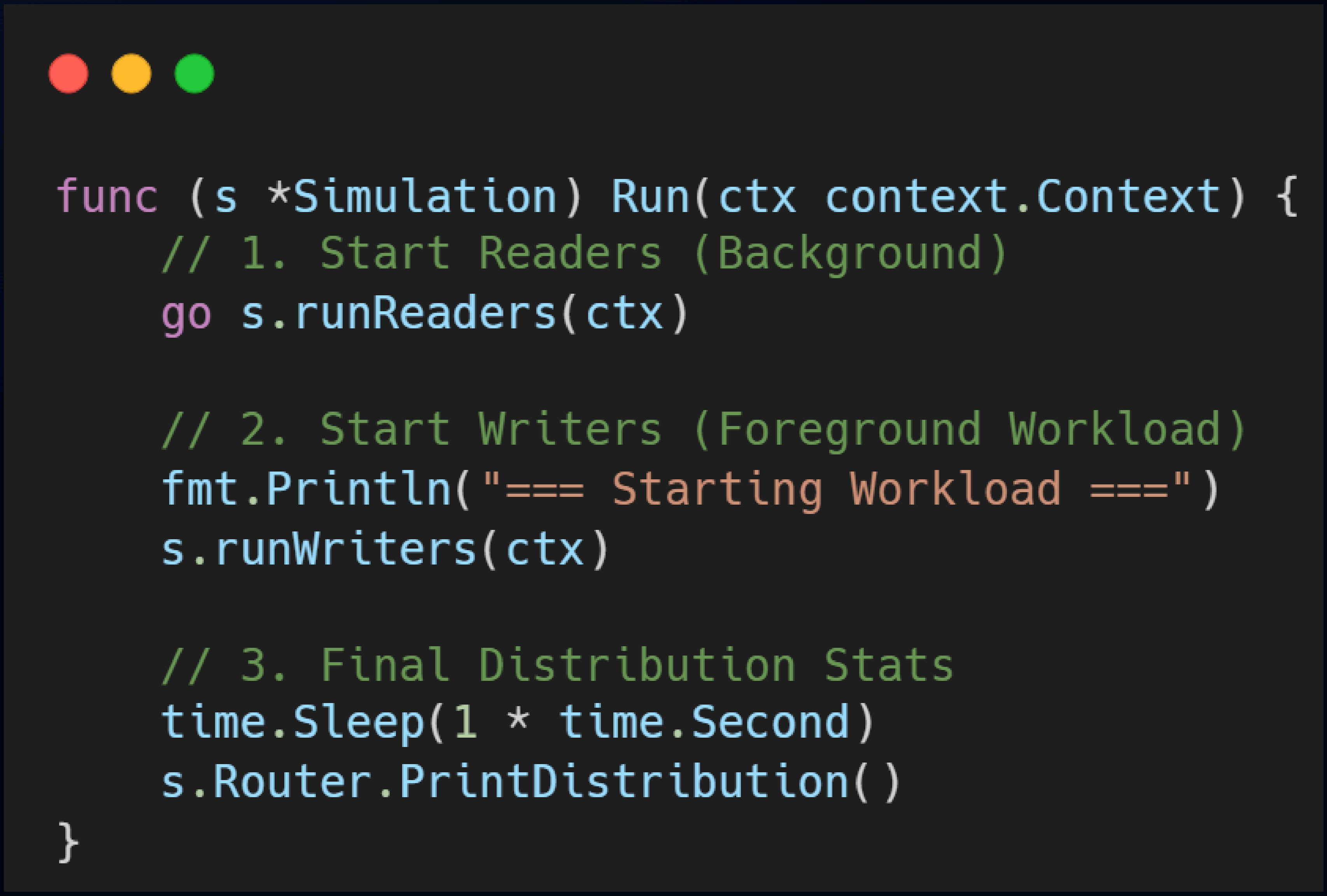
Code: Visualizing the Execution

The Simulation in Action - try it out!

To see Sharding in action, we need an orchestrator that manages the lifecycle of our cluster.

- **Orchestration**: We initialize 4 physical Shards and 1 Router.
- **Graceful Shutdown**: We use *context.WithCancel* and *os/signal* to handle Ctrl+C. This ensures all background goroutines exit cleanly - a critical skill for any Go backend engineer.
- **Background Readers**: Just like in the real world, clients are trying to read while we are still writing!

Below is the implementation under *simulation.go*:



```go
func (s *Simulation) Run(ctx context.Context) {
    // 1. Start Readers (Background)
    go s.runReaders(ctx)

    // 2. Start Writers (Foreground Workload)
    fmt.Println("=== Starting Workload ===")
    s.runWriters(ctx)

    // 3. Final Distribution Stats
    time.Sleep(1 * time.Second)
    s.Router.PrintDistribution()
}
```

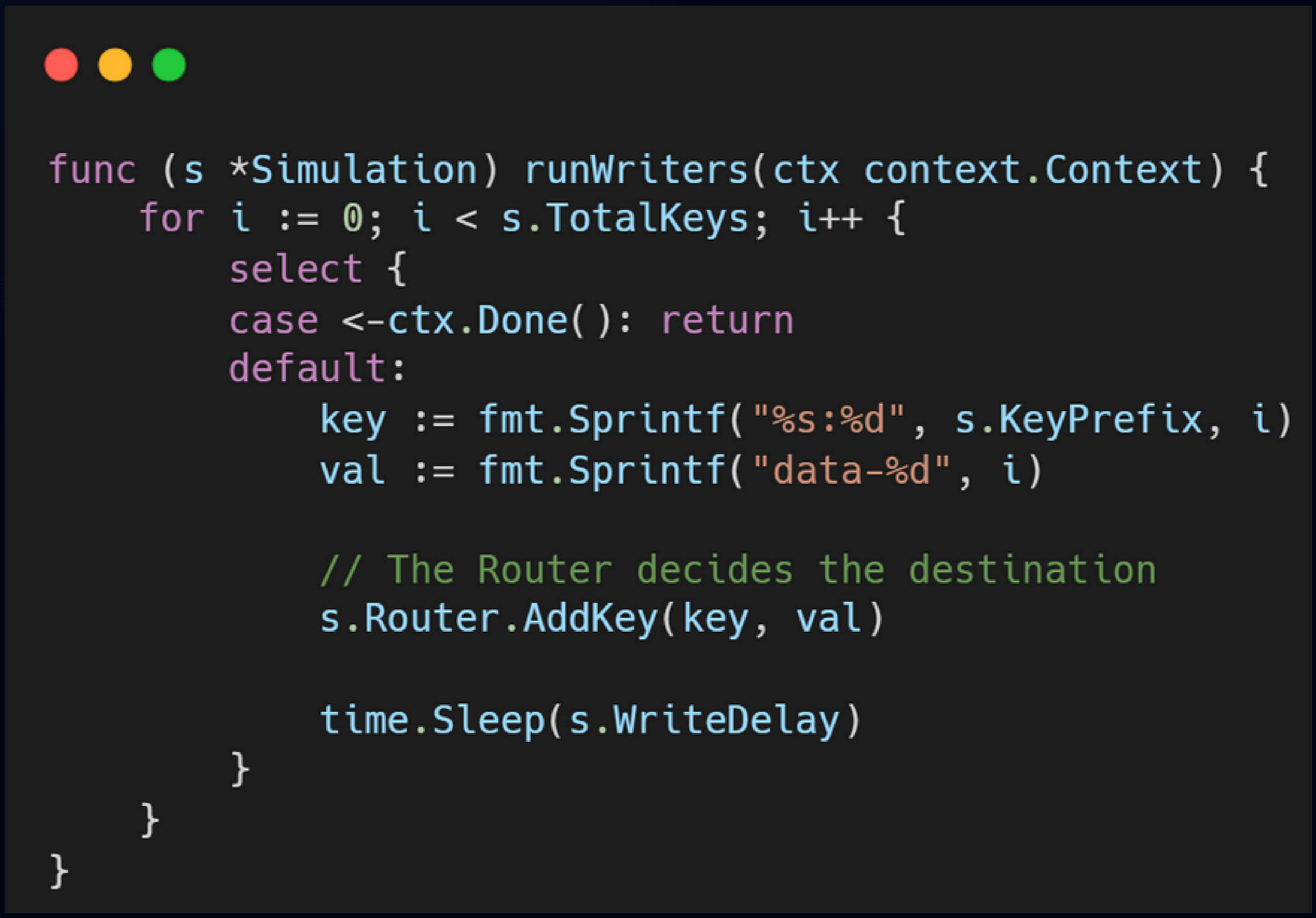
Code: Generating the Load

Stress Testing the System

How do we "prove" the data is balanced? We simulate a burst of traffic.

- **Deterministic Keys**: We write keys like *user:1*, *user:2* to see how the hashing logic maps sequential data to different nodes.
- **Concurrency**: While *runWriters* is pushing data, *runReaders* is randomly querying shards.
- **The "Hit" vs "Miss"**: Because readers run in parallel, you'll see "Read Hits" only after the writer has successfully routed that specific key to its shard.

Below is the implementation under *simulation.go*:



```go
func (s *Simulation) runWriters(ctx context.Context) {
    for i := 0; i < s.TotalKeys; i++ {
        select {
        case <-ctx.Done(): return
        default:
            key := fmt.Sprintf("%s:%d", s.KeyPrefix, i)
            val := fmt.Sprintf("data-%d", i)

            // The Router decides the destination
            s.Router.AddKey(key, val)

            time.Sleep(s.WriteDelay)
        }
    }
}
```

Summary & Takeaways

☑ HORIZONTAL SCALABILITY

Sharding solves the Write-Heavy problem. By adding more shards, you gain more CPU and Disk I/O bandwidth.

☑ DETERMINISTIC ROUTING

The Router is your map. It ensures that user:1 is always found on the same physical shard.

☑ FAULT ISOLATION

If one shard crashes, only 25% of your data is affected. The rest of your users stay online.

☑ THE COMPLEXITY TRADE-OFF:

You gain scale, but you lose easy Range Queries. If you need to find "all users," you must now query every single shard.

Want to dive deeper?

Explore the full source code and deep-dive documentation on my GitHub repository or connect via LinkedIn.

GitHub LinkedIn

LIKE • COMMENT • SHARE