Scaling Strategies

Horizontal vs Vertical Scaling

Vertical scaling: Add more resources to single instance (more CPU cores, more RAM)
Horizontal scaling: Add more instances (more servers, more goroutines)

When to Scale Vertically

Good for:

- Shared state (single process easier)
- Low latency requirements (no network)
- Simple deployment
- Memory-bound workloads

```
// Vertical: Increase GOMAXPROCS
runtime.GOMAXPROCS(runtime.NumCPU() * 2)

// More workers in single process
pool := NewWorkerPool(runtime.NumCPU() * 10)
```

Limits:

- Single machine capacity
- Single point of failure
- Cost increases exponentially

When to Scale Horizontally

Good for:

- Stateless workloads
- High availability
- Cost-effective at scale
- Geographic distribution

```
// Horizontal: Multiple instances behind load balancer
// Each instance runs independently

// Scale by adding servers, not by changing code
```

Challenges:

- State synchronization
- Network latency
- Deployment complexity

Partitioning Work

Strategy 1: Data Partitioning

Divide data into independent chunks.

```
// Partition by key range
func processUsers(users []User, workers int) {
    chunkSize := len(users) / workers

    var wg sync.WaitGroup
    for i := 0; i < workers; i++ {
        wg.Add(1)

        start := i * chunkSize
        end := start + chunkSize
        if i == workers-1 {
            end = len(users)  // Last worker handles remainder
        }

        go func(chunk []User) {
            defer wg.Done()
            for _, user := range chunk {
                process(user)
            }
        }(users[start:end])
    }

    wg.Wait()
}
```

Use when:

- Data can be split independently
- No cross-partition dependencies
- Equal work per partition

Strategy 2: Sharding

Partition by hash/key to specific workers.

```
type ShardedProcessor struct {
    shards []*Shard
}

type Shard struct {
    id    int
    tasks chan Task
    wg    sync.WaitGroup
}

func NewShardedProcessor(numShards int) *ShardedProcessor {
    sp := &ShardedProcessor{
        shards: make([]*Shard, numShards),
    }
```

```
    for i := 0; i < numShards; i++ {
        sp.shards[i] = &Shard{
            id:    i,
            tasks: make(chan Task, 100),
        }
        sp.shards[i].wg.Add(1)
        go sp.shards[i].run()
    }

    return sp
}

func (sp *ShardedProcessor) Submit(task Task) {
    shard := sp.getShard(task.Key)
    shard.tasks <- task
}

func (sp *ShardedProcessor) getShard(key string) *Shard {
    hash := fnv32(key)
    return sp.shards[hash%uint32(len(sp.shards))]
}

func (s *Shard) run() {
    defer s.wg.Done()
    for task := range s.tasks {
        process(task)
    }
}
```

Benefits:

- Consistent routing (same key → same shard)
- Independent shard state
- Easy to scale (add shards)

Use when:

- Natural key-based distribution
- Need consistent routing
- Want to scale incrementally

Strategy 3: Round-Robin

Distribute work evenly across workers.

```
type LoadBalancer struct {
    workers []Worker
    index   atomic.Uint32
}

func (lb *LoadBalancer) Submit(task Task) {
    // Round-robin selection
    idx := lb.index.Add(1) % uint32(len(lb.workers))
```

```
    lb.workers[idx].Process(task)
}
```

Use when:

- Work uniform
- No affinity needed
- Simple distribution

Strategy 4: Dynamic Work Stealing

Workers steal from each other when idle.

```
type WorkStealingPool struct {
    workers []*StealingWorker
}

type StealingWorker struct {
    id    int
    queue deque.Deque[Task]
    pool  *WorkStealingPool
}

func (w *StealingWorker) run() {
    for {
        // Try own queue first
        if task, ok := w.queue.PopBottom(); ok {
            task.Execute()
            continue
        }

        // Steal from others
        if task, ok := w.steal(); ok {
            task.Execute()
            continue
        }

        // No work
        time.Sleep(time.Millisecond)
    }
}

func (w *StealingWorker) steal() (Task, bool) {
    for _, victim := range w.pool.workers {
        if victim.id == w.id {
            continue
        }

        if task, ok := victim.queue.PopTop(); ok {
            return task, true
        }
    }
```

```
    return Task{}, false
}
```

Use when:

- Work distribution uneven
- Want automatic load balancing
- Can tolerate complexity

Dynamic Scaling

Auto-scaling Worker Pool

```
type DynamicPool struct {
    minWorkers int
    maxWorkers int
    tasks      chan Task

    mu         sync.Mutex
    workers    int

    scaleUp    chan struct{}
    scaleDown  chan struct{}
}

func (p *DynamicPool) monitor() {
    ticker := time.NewTicker(10 * time.Second)
    defer ticker.Stop()

    for range ticker.C {
        queueLen := len(p.tasks)

        p.mu.Lock()
        currentWorkers := p.workers
        p.mu.Unlock()

        // Scale up if queue building
        if queueLen > currentWorkers*10 && currentWorkers < p.maxWorkers {
            p.addWorker()
        }

        // Scale down if mostly idle
        if queueLen < currentWorkers*2 && currentWorkers > p.minWorkers {
            p.removeWorker()
        }
    }
}

func (p *DynamicPool) addWorker() {
    p.mu.Lock()
    p.workers++
```

```
    workerID := p.workers
    p.mu.Unlock()

    go p.worker(workerID)
}

func (p *DynamicPool) removeWorker() {
    select {
    case p.scaleDown <- struct{}{}:
        p.mu.Lock()
        p.workers--
        p.mu.Unlock()
    default:
    }
}

func (p *DynamicPool) worker(id int) {
    for {
        select {
        case task := <-p.tasks:
            task.Execute()
        case <-p.scaleDown:
            return  // Shutdown this worker
        }
    }
}
```

Metrics for scaling:

- Queue depth (backlog)
- CPU utilization
- Response latency
- Error rate

Adaptive Concurrency

Adjust concurrency based on performance.

```
type AdaptiveSystem struct {
    concurrency atomic.Int32

    // Metrics
    successRate  atomic.Int32  // Per 1000 requests
    avgLatency   atomic.Int64  // Nanoseconds
}

func (as *AdaptiveSystem) adjust() {
    ticker := time.NewTicker(time.Second)
    defer ticker.Stop()

    for range ticker.C {
        success := as.successRate.Load()
```

```
        latency := as.avgLatency.Load()
        current := as.concurrency.Load()

        // High success, low latency → increase
        if success > 990 && latency < 100_000_000 {  // < 100ms
            as.concurrency.Store(current + 1)
        }

        // Low success or high latency → decrease
        if success < 950 || latency > 500_000_000 {  // > 500ms
            if current > 1 {
                as.concurrency.Store(current - 1)
            }
        }

        // Reset metrics
        as.successRate.Store(0)
        as.avgLatency.Store(0)
    }
}

func (as *AdaptiveSystem) process(task Task) {
    // Limit concurrent operations
    sem := make(chan struct{}, as.concurrency.Load())

    sem <- struct{}{}
    defer func() { <-sem }()

    start := time.Now()
    err := task.Execute()
    duration := time.Since(start)

    // Record metrics
    if err == nil {
        as.successRate.Add(1000 / 1000)  // Success
    }
    as.avgLatency.Add(duration.Nanoseconds() / 1000)
}
```

Load Balancing Patterns

Pattern 1: Least-Loaded

Send work to least busy worker.

```
type LeastLoadedBalancer struct {
    workers []*LoadedWorker
    mu      sync.RWMutex
}

type LoadedWorker struct {
```

```
    id      int
    tasks   chan Task
    pending atomic.Int32
}

func (lb *LeastLoadedBalancer) Submit(task Task) {
    worker := lb.findLeastLoaded()
    worker.pending.Add(1)
    worker.tasks <- task
}

func (lb *LeastLoadedBalancer) findLeastLoaded() *LoadedWorker {
    lb.mu.RLock()
    defer lb.mu.RUnlock()

    var best *LoadedWorker
    minLoad := int32(math.MaxInt32)

    for _, worker := range lb.workers {
        load := worker.pending.Load()
        if load < minLoad {
            minLoad = load
            best = worker
        }
    }

    return best
}

func (w *LoadedWorker) run() {
    for task := range w.tasks {
        task.Execute()
        w.pending.Add(-1)
    }
}
```

Pattern 2: Weighted Round-Robin

Consider worker capacity.

```
type WeightedBalancer struct {
    workers []WeightedWorker
    index   atomic.Uint32
}

type WeightedWorker struct {
    worker Worker
    weight int  // Capacity multiplier
}

func (wb *WeightedBalancer) Submit(task Task) {
```

```
    // Skip based on weight
    idx := int(wb.index.Add(1))

    for {
        worker := wb.workers[idx%len(wb.workers)]
        if idx%worker.weight == 0 {
            worker.worker.Process(task)
            return
        }
        idx++
    }
}
```

Pattern 3: Consistent Hashing

Minimize redistribution when adding/removing workers.

```
type ConsistentHash struct {
    ring     map[uint32]Worker
    sortedKeys []uint32
    mu       sync.RWMutex
}

func (ch *ConsistentHash) AddWorker(worker Worker, replicas int) {
    ch.mu.Lock()
    defer ch.mu.Unlock()

    for i := 0; i < replicas; i++ {
        hash := fnv32(fmt.Sprintf("%s:%d", worker.ID, i))
        ch.ring[hash] = worker
        ch.sortedKeys = append(ch.sortedKeys, hash)
    }

    sort.Slice(ch.sortedKeys, func(i, j int) bool {
        return ch.sortedKeys[i] < ch.sortedKeys[j]
    })
}

func (ch *ConsistentHash) Get(key string) Worker {
    ch.mu.RLock()
    defer ch.mu.RUnlock()

    if len(ch.ring) == 0 {
        return nil
    }

    hash := fnv32(key)

    // Binary search for closest worker
    idx := sort.Search(len(ch.sortedKeys), func(i int) bool {
        return ch.sortedKeys[i] >= hash
```

```
    })

    if idx == len(ch.sortedKeys) {
        idx = 0
    }

    return ch.ring[ch.sortedKeys[idx]]
}
```

Performance Patterns

Pattern 1: Batching

Process multiple items together.

```
type Batcher struct {
    batchSize int
    buffer    []Task
    mu        sync.Mutex
    cond      *sync.Cond
}

func NewBatcher(size int) *Batcher {
    b := &Batcher{batchSize: size, buffer: make([]Task, 0, size)}
    b.cond = sync.NewCond(&b.mu)
    go b.flusher()
    return b
}

func (b *Batcher) Submit(task Task) {
    b.mu.Lock()
    b.buffer = append(b.buffer, task)

    if len(b.buffer) >= b.batchSize {
        b.cond.Signal()
    }
    b.mu.Unlock()
}

func (b *Batcher) flusher() {
    ticker := time.NewTicker(100 * time.Millisecond)
    defer ticker.Stop()

    for {
        select {
        case <-ticker.C:
            b.flush()
        }
    }
}
```

```
func (b *Batcher) flush() {
    b.mu.Lock()
    if len(b.buffer) == 0 {
        b.mu.Unlock()
        return
    }

    batch := b.buffer
    b.buffer = make([]Task, 0, b.batchSize)
    b.mu.Unlock()

    processBatch(batch)  // Single operation for whole batch
}
```

Use for:

- Database inserts (batch INSERT)
- Network requests (batch RPC)
- File writes (buffer before flush)

Pattern 2: Pipelining

Overlap stages for throughput.

```
// 3-stage pipeline with parallelism at each stage
type Pipeline struct {
    stage1Workers int
    stage2Workers int
    stage3Workers int

    stage1Out chan Stage1Result
    stage2Out chan Stage2Result
}

func (p *Pipeline) Start() {
    // Stage 1: Parallel
    for i := 0; i < p.stage1Workers; i++ {
        go p.stage1()
    }

    // Stage 2: Parallel
    for i := 0; i < p.stage2Workers; i++ {
        go p.stage2()
    }

    // Stage 3: Parallel
    for i := 0; i < p.stage3Workers; i++ {
        go p.stage3()
    }
}
```

Real Example: Image Processing Service

```
type ImageService struct {
    // Horizontal scaling components
    fetchPool   *WorkerPool      // Download images
    processPool *WorkerPool      // Transform images
    uploadPool  *WorkerPool      // Upload results

    // Sharded state
    cache       *ShardedCache    // Recently processed

    // Load balancing
    balancer    *LeastLoadedBalancer

    // Metrics
    processed   atomic.Int64
    errors      atomic.Int64
}

func NewImageService() *ImageService {
    cpus := runtime.NumCPU()

    return &ImageService{
        // I/O-bound: More workers than CPUs
        fetchPool:  NewWorkerPool(cpus * 10),

        // CPU-bound: Workers = CPUs
        processPool: NewWorkerPool(cpus),

        // I/O-bound: More workers than CPUs
        uploadPool: NewWorkerPool(cpus * 10),

        // 16 shards to reduce contention
        cache: NewShardedCache(16),
    }
}

func (is *ImageService) Process(imageURL string) error {
    // Check cache (sharded for scalability)
    if result, ok := is.cache.Get(imageURL); ok {
        return result
    }

    // Pipeline stages

    // Stage 1: Fetch (I/O-bound, high concurrency)
    image, err := is.fetchPool.Submit(FetchTask{URL: imageURL}).Get()
    if err != nil {
        is.errors.Add(1)
        return err
```

```
    }

    // Stage 2: Process (CPU-bound, limited concurrency)
    processed, err := is.processPool.Submit(ProcessTask{Image: image}).Get()
    if err != nil {
        is.errors.Add(1)
        return err
    }

    // Stage 3: Upload (I/O-bound, high concurrency)
    result, err := is.uploadPool.Submit(UploadTask{Image: processed}).Get()
    if err != nil {
        is.errors.Add(1)
        return err
    }

    // Cache result
    is.cache.Set(imageURL, result)
    is.processed.Add(1)

    return nil
}
```

Scaling Checklist

- [ ] Identify bottleneck (CPU, I/O, memory, network)
- [ ] Choose scaling strategy (vertical vs horizontal)
- [ ] Partition work appropriately
- [ ] Match concurrency to workload (CPU-bound vs I/O-bound)
- [ ] Implement load balancing
- [ ] Add monitoring and metrics
- [ ] Test under realistic load
- [ ] Consider auto-scaling
- [ ] Plan for failure scenarios
- [ ] Document scaling limits

Interview Questions

Q: "How do you choose between horizontal and vertical scaling?"

"Vertical: Shared state (single process easier), low latency (no network), simple deployment. Horizontal: Stateless workloads, high availability, cost-effective at scale. Example: In-memory cache (vertical), stateless HTTP API (horizontal). Benchmark: If adding cores helps proportionally, scale vertically. If bottleneck is single-machine limits or need HA, scale horizontally."

Q: "How many workers for CPU-bound vs I/O-bound work?"

"CPU-bound: Workers = NumCPU (more doesn't help, causes context switching). I/O-bound: Workers = NumCPU × 10-100 (waiting for I/O, parallel requests help). Example: Image processing (CPU-bound) = 8 workers on 8-core. HTTP fetching (I/O-bound) = 80 workers on 8-core. Benchmark to find optimal."

Q: "What is sharding and when to use it?"

"Sharding: Partition data/work by key, route to specific worker/instance. Reduces lock contention (16 shards = 1/16 contention), enables consistent routing (same key → same shard). Use when: Natural key distribution (user ID, session ID), need independent state, want to scale incrementally. Example: Cache sharded by key hash, ensures same key always hits same shard (better for cache locality)."

Q: "How to implement auto-scaling?"

"Monitor metrics: Queue depth, CPU utilization, latency, error rate. Scale up if: Queue growing (backlog), high CPU (>80%), high latency (>SLA). Scale down if: Queue small, low CPU (<20%), low latency. Example: If queue depth > workers × 10, add worker. If < workers × 2 for 5 minutes, remove worker. Gradual scaling (add/remove one at a time), avoid thrashing (cooldown period between changes)."

Key Takeaways

1. **Vertical scaling: Add resources. Horizontal: Add instances**
2. **CPU-bound work: Workers = NumCPU**
3. **I/O-bound work: Workers = NumCPU × 10-100**
4. **Sharding reduces contention, enables scaling**
5. **Load balancing distributes work evenly**
6. **Batching improves throughput**
7. **Monitor metrics for auto-scaling**
8. **Test under realistic load**
9. **Match concurrency to workload type**
10. **Simple is better, add complexity only when needed**

Exercises

1. Benchmark worker pool with 1, 2, 4, 8, 16, 32 workers for CPU-bound and I/O-bound work.
2. Implement sharded cache with 1, 4, 16, 64 shards, measure lock contention.
3. Build auto-scaling worker pool based on queue depth and latency.
4. Compare round-robin vs least-loaded vs consistent hashing for load balancing.
5. Design scalable image processing system handling 10,000 requests/second.

Next: failure-modes.md - Designing for failure and graceful degradation.