# Concurrent, High-Performance Data-Access With Go

Sunil Sayyaparaju
Engineer
Aerospike

Khosrow Afroozeh
Engineer
Aerospike

# What does High Performance mean?

# North American RTB speeds & feeds

- 100 millisecond or 150 millisecond ad delivery
  - De-facto standard set in 2004 by Washington Post and others
- North America is 70 to 90 milliseconds wide
  - Two or three data centers
- Newyork-Amsterdam is 150 milliseconds
- Auction is limited to 30 milliseconds
  - Typically closes in 5 milliseconds
- Winners have more data, better models – in 5 milliseconds

# North American RTB speeds & feeds

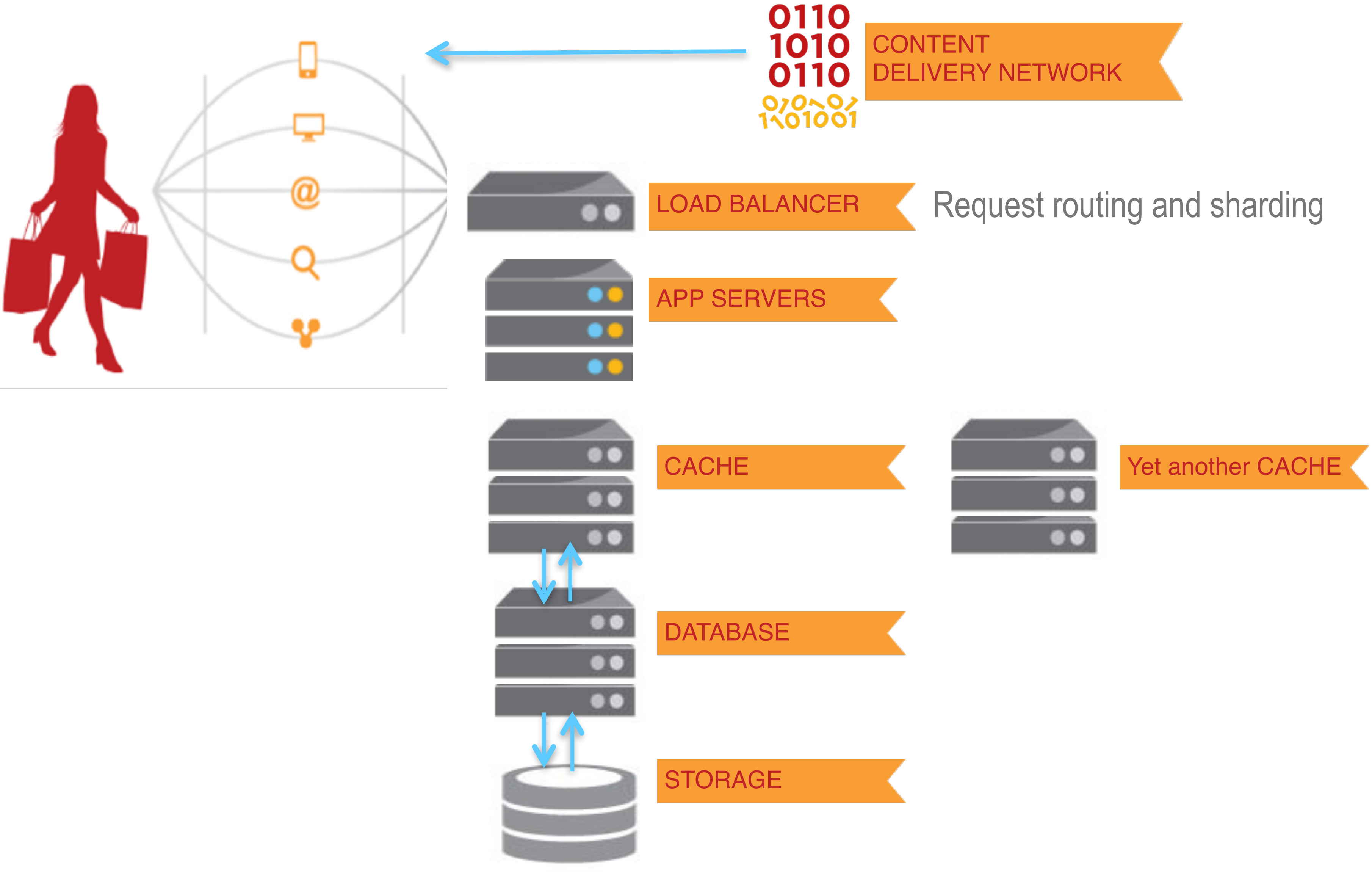
- 1 to 6 billion cookies tracked
  - Some companies track 200M, some track 20B

- Each bidder has their own data pool
  - Data is your weapon
  - Recent searches, behavior, IP addresses
  - Audience clusters (K-cluster, K-means) from offline Batch Processing

- “Remnant” from Google, Yahoo is about 0.6 million / sec
- Facebook exchange: about 0.6 million / sec
- “others” are 0.5 million / sec

Currently around 2.0M / sec in North America

# Layer-Driven Architecture



CONTENT DELIVERY NETWORK

LOAD BALANCER

Request routing and sharding

APP SERVERS

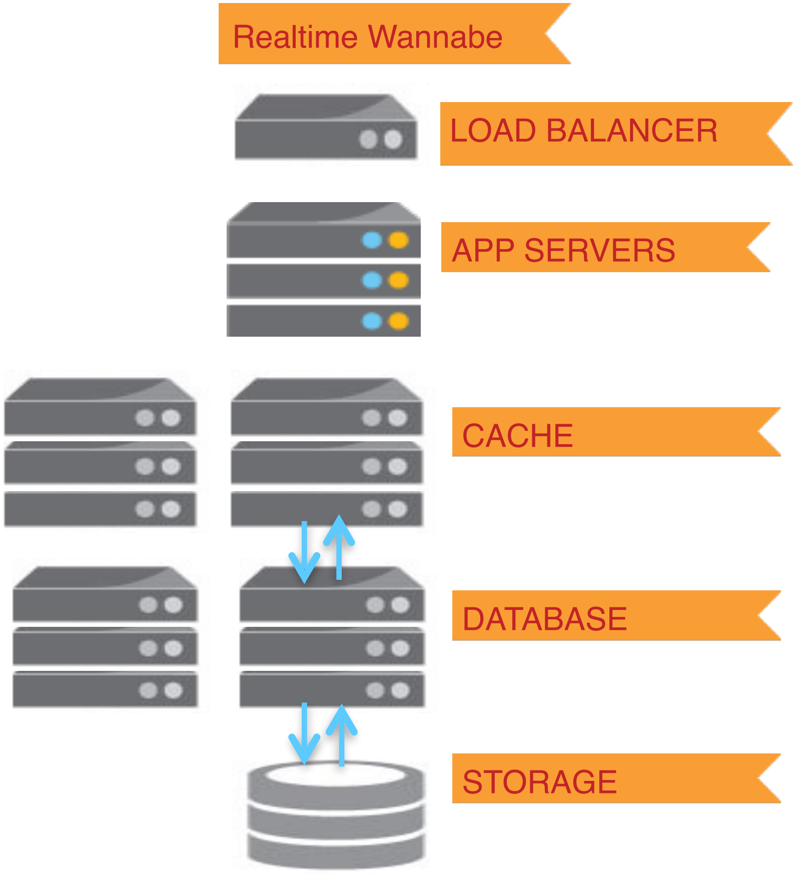
CACHE

Yet another CACHE

DATABASE

STORAGE

# Minimalism Makes a Comeback



Realtime Wannabe

LOAD BALANCER

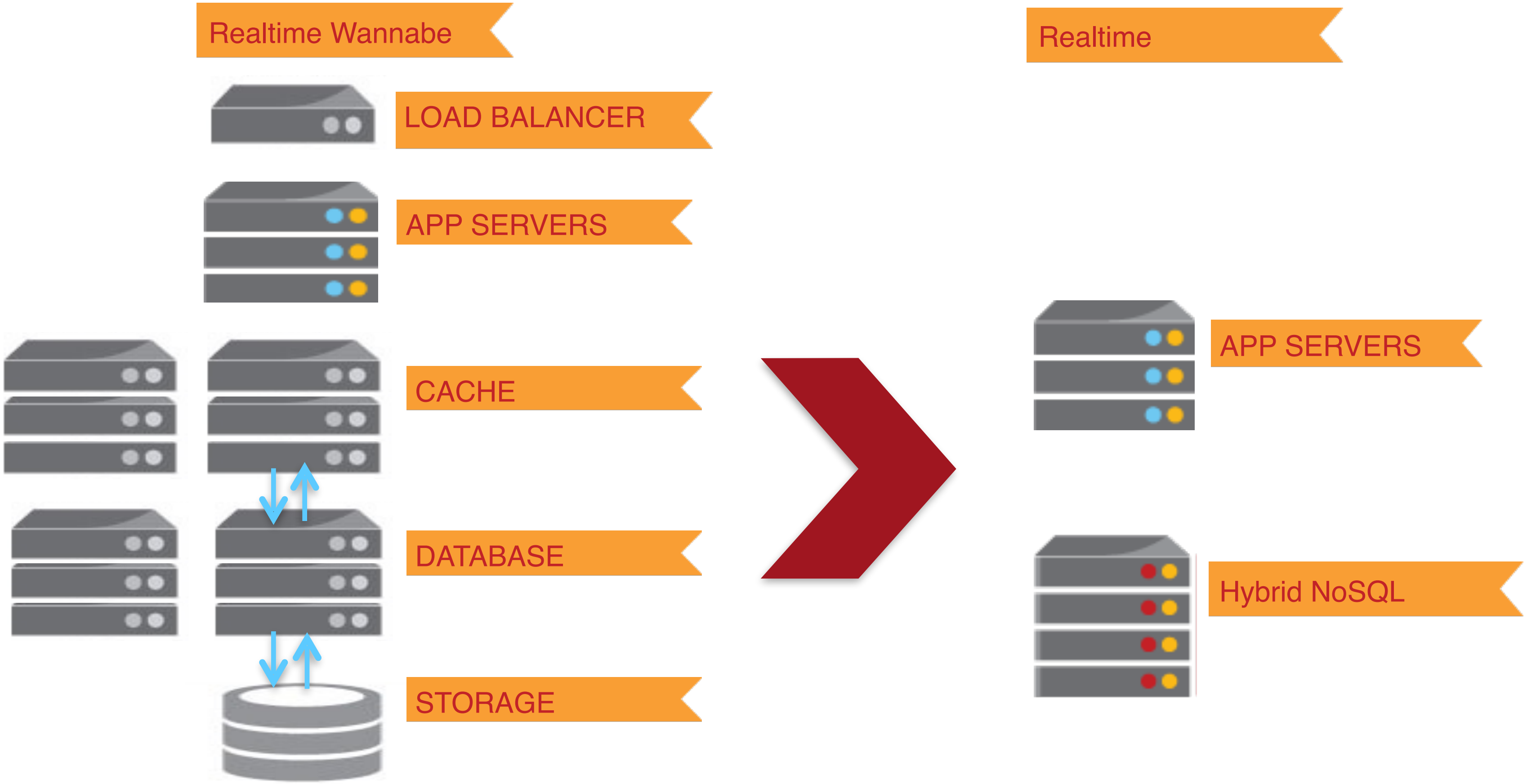
APP SERVERS

CACHE

DATABASE

STORAGE

# Minimalism Makes a Comeback


Realtime Wannabe

LOAD BALANCER

APP SERVERS

CACHE

DATABASE

STORAGE

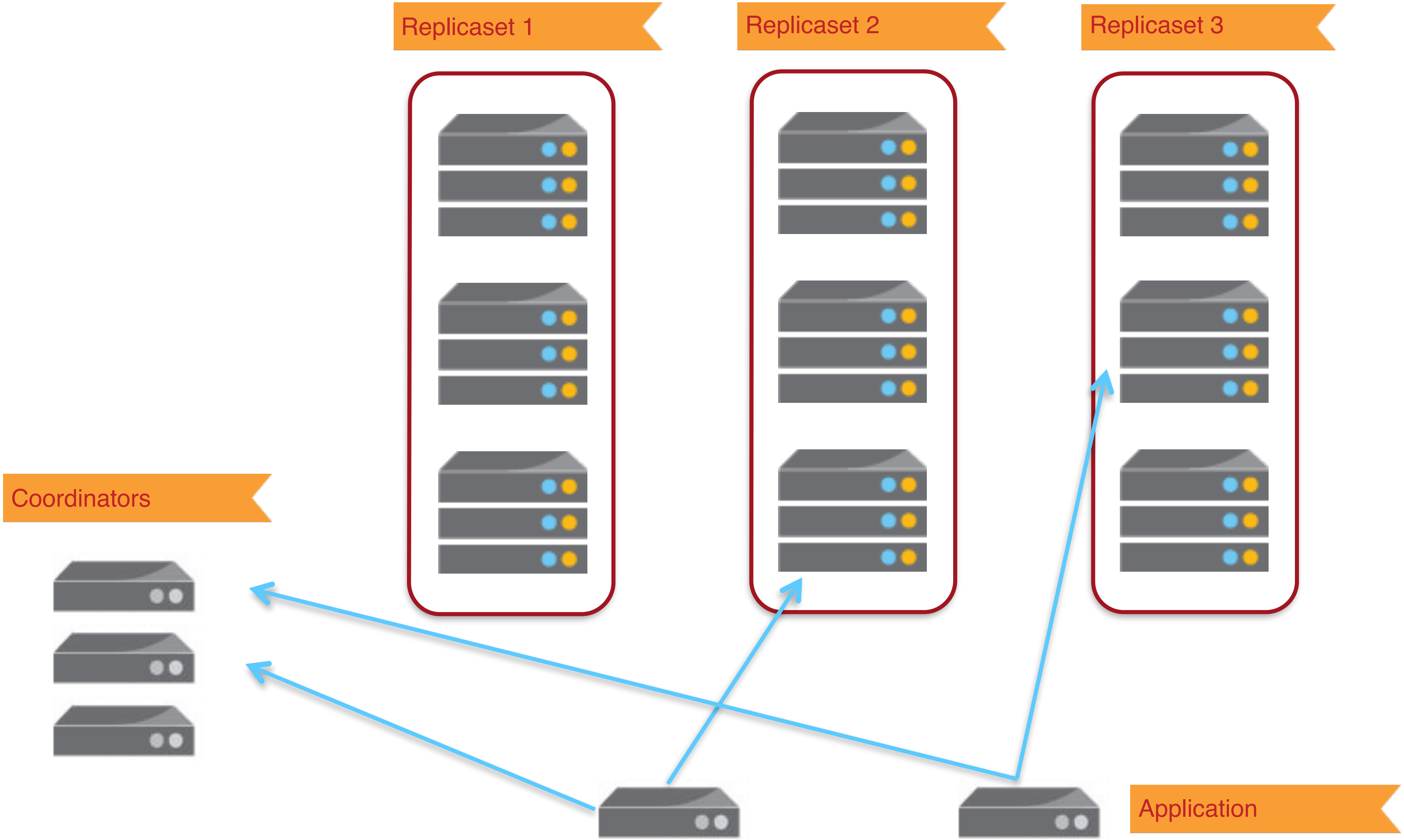
Realtime

APP SERVERS

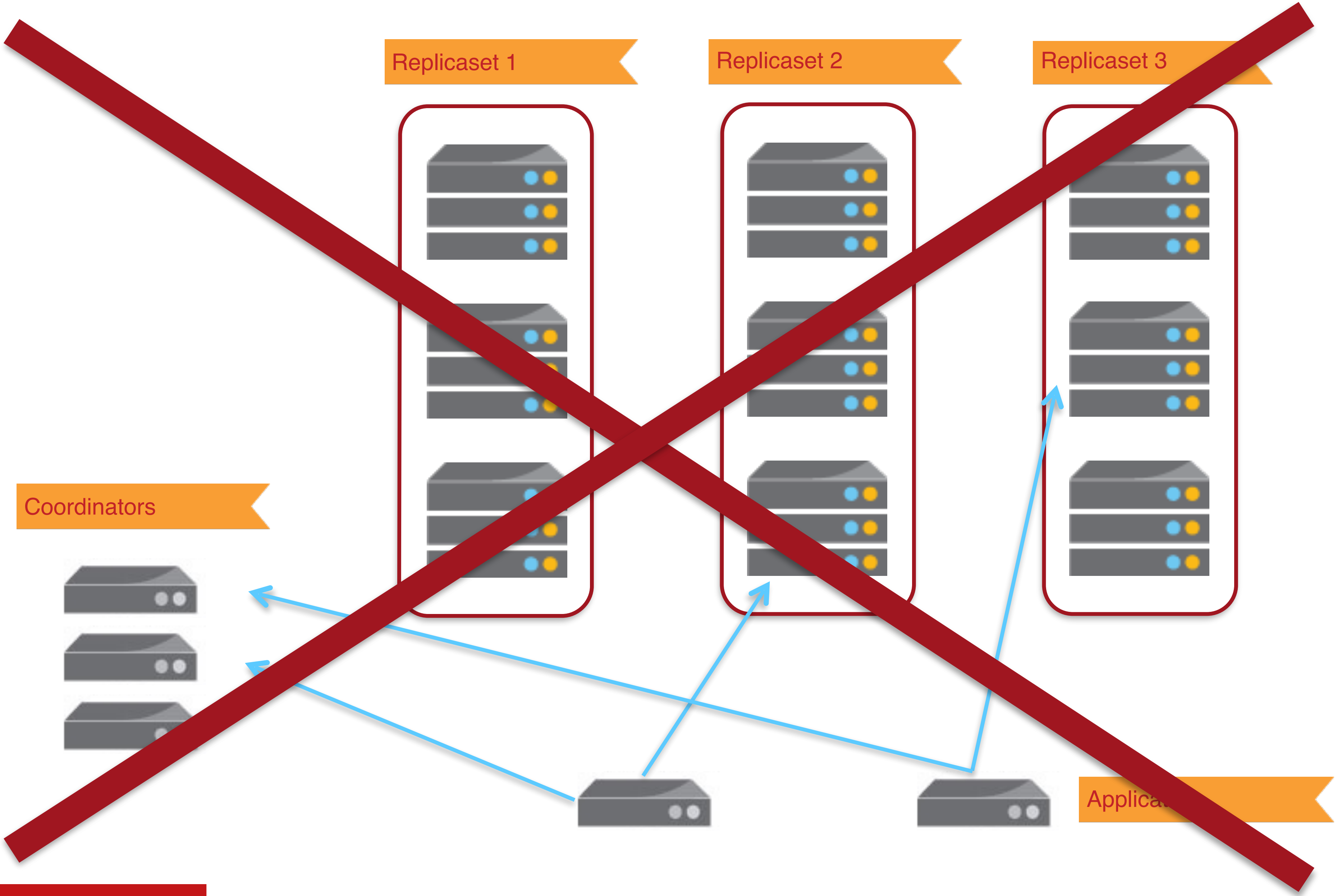
Hybrid NoSQL

# Aerospike is a distributed database

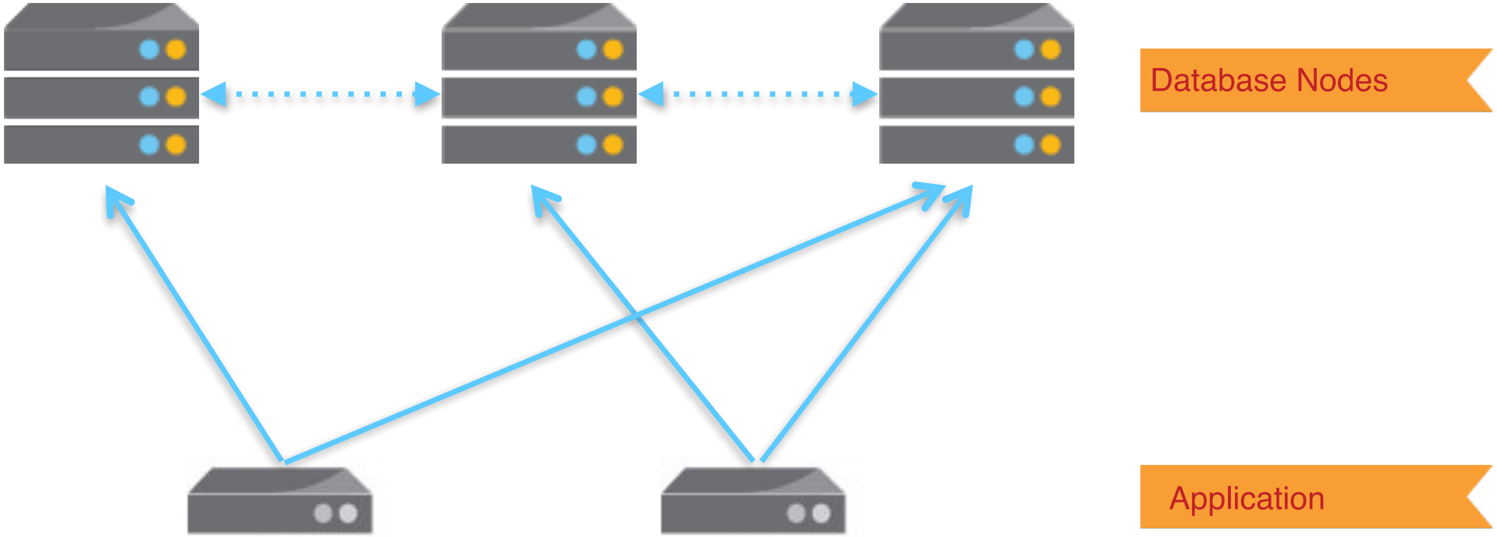
# Hint: It is *NOT* like this!



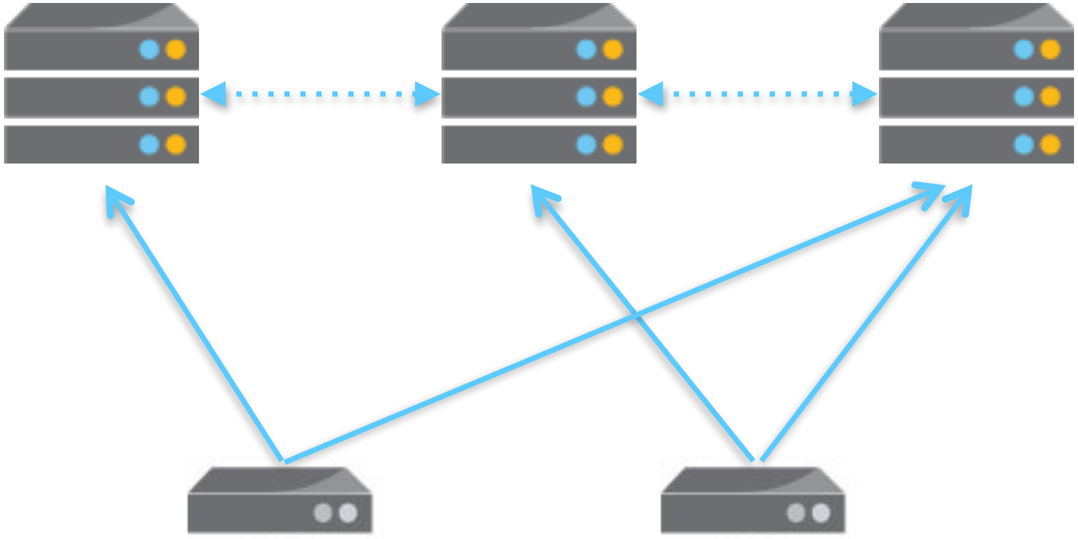
# Hint: It is *NOT* like this!



# Aerospike



# Aerospike



Benefits of Design:

1) **No Hotspots** – DHT simplifies data partitioning

2) Smart Client – **1 hop** to data, *no load balancers needed*

3) **Shared Nothing** Architecture, every node identical - *no coordinators needed*

4) Single row **ACID** – sync'd replication in cluster

5) Smart Cluster, **Zero Touch** – auto-failover, rebalancing, rack aware, rolling upgrades...

6) Transactions and long running tasks prioritized real-time

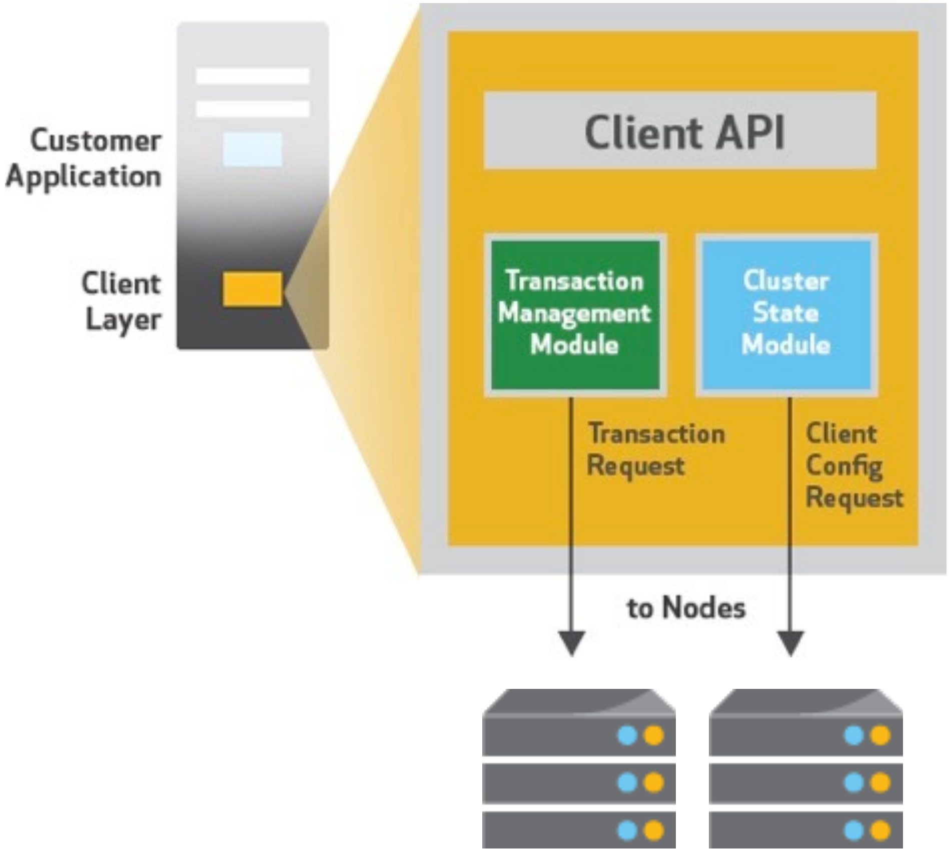
7) **XDR** – sync'd replication across data centers ensures **Zero Downtime**

8) **Scale linearly** as data-sizes and workloads increase

9) Add capacity with **no service interruption**

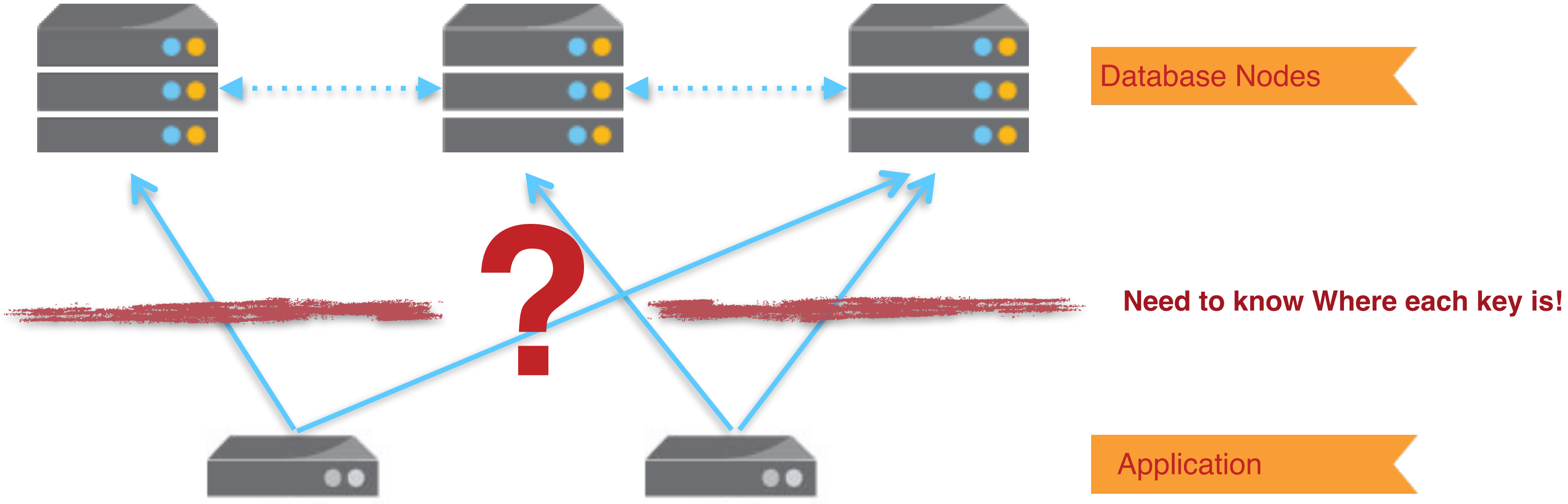
# Client Libraries Need To Do More - TRANSPARENTLY

- **Implements Aerospike API**
  - Optimistic row locking
  - Optimized binary protocol
- **Cluster Tracking**
  - Learns about cluster changes, Partition Map
- **Transaction Semantics**
  - Global Transaction ID
  - Retransmit and timeout
- **Linearly Scales**
  - No extra hop to data
  - No load balancers in the way



# Tracking Cluster Changes

# Cluster Tracking



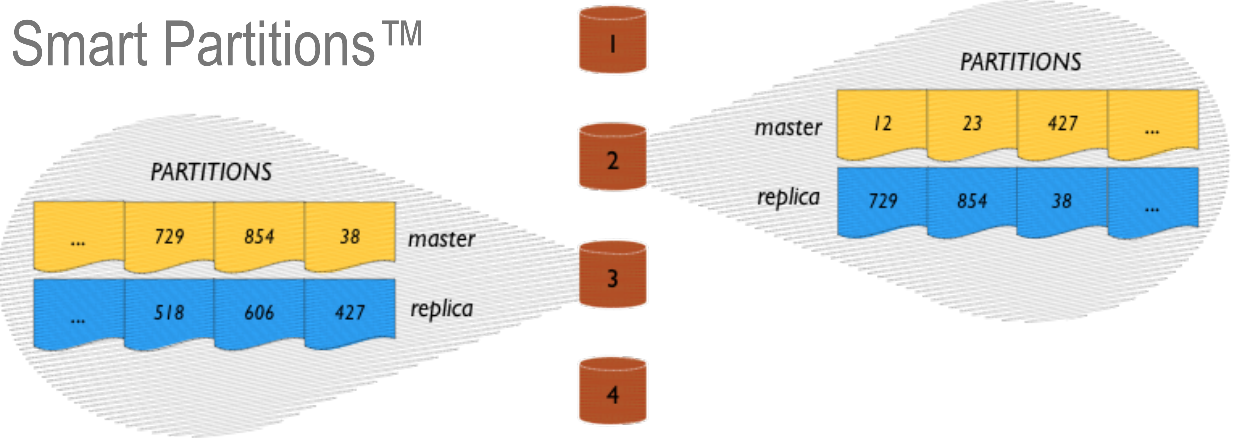
Database Nodes

Need to know Where each key is!

Application

Data is **distributed evenly** across nodes in a cluster using the Aerospike Smart Partitions™ algorithm.

- Even distribution of
  - Partitions across nodes
  - Records across Partitions
  - Data across Flash device



PARTITIONS

| | | | | |
|---|---|---|---|---|
| ... | 729 | 854 | 38 | master |
| ... | 518 | 606 | 427 | replica |

PARTITIONS

| | | | | |
|---|---|---|---|---|
| master | 12 | 23 | 427 | ... |
| replica | 729 | 854 | 38 | ... |

# Cluster Tracking

Spawn Goroutine To Track Cluster Changes

```
func NewCluster(policy *ClientPolicy, hosts []*Host) (*Cluster, error) {
        newCluster := &Cluster{...}

        // start up cluster maintenance go routine
        newCluster.wgTend.Add(1)
        go newCluster.clusterBoss(policy)

        return newCluster, nil
}

func (clstr *Cluster) clusterBoss(policy *ClientPolicy) {
        defer clstr.wgTend.Done()

Loop:
        for {
                select {
                case <-clstr.tendChannel:
                        // tend channel closed
                        break Loop
                case <-time.After(tendInterval):
                        if err := clstr.tend(); err != nil {
                                Logger.Warn(err.Error())
                        }
                }
        }

        // cleanup code goes here
        // close the nodes, ...
}

func (clstr *Cluster) Close() {
        if !clstr.closed.Get() {
                // send close signal to maintenance channel
                close(clstr.tendChannel)

                // wait until tend is over
                clstr.wgTend.Wait()
        }
}
```

# Cluster Tracking

```
func NewCluster(policy *ClientPolicy, hosts []*Host) (*Cluster, error) {
        newCluster := &Cluster{...}

        // start up cluster maintenance go routine
        newCluster.wgTend.Add(1)
        go newCluster.clusterBoss(policy)

        return newCluster, nil
}

func (clstr *Cluster) clusterBoss(policy *ClientPolicy) {
        defer clstr.wgTend.Done()

Loop:
        for {
                select {
                case <-clstr.tendChannel:
                        // tend channel closed
                        break Loop
                case <-time.After(tendInterval):
                        if err := clstr.tend(); err != nil {
                                Logger.Warn(err.Error())
                        }
                }
        }

        // cleanup code goes here
        // close the nodes, ...
}

func (clstr *Cluster) Close() {
        if !clstr.closed.Get() {
                // send close signal to maintenance channel
                close(clstr.tendChannel)

                // wait until tend is over
                clstr.wgTend.Wait()
        }
}
```

On Intervals, Update
Cluster Status

# Cluster Tracking

```
func NewCluster(policy *ClientPolicy, hosts []*Host) (*Cluster, error) {
        newCluster := &Cluster{...}

        // start up cluster maintenance go routine
        newCluster.wgTend.Add(1)
        go newCluster.clusterBoss(policy)

        return newCluster, nil
}

func (clstr *Cluster) clusterBoss(policy *ClientPolicy) {
        defer clstr.wgTend.Done()

Loop:
        for {
                select {
                case <-clstr.tendChannel:
                        // tend channel closed
                        break Loop
                case <-time.After(tendInterval):
                        if err := clstr.tend(); err != nil {
                                Logger.Warn(err.Error())
                        }
                }
        }

        // cleanup code goes here
        // close the nodes, ...
}

func (clstr *Cluster) Close() {
        if !clstr.closed.Get() {
                // send close signal to maintenance channel
                close(clstr.tendChannel)

                // wait until tend is over
                clstr.wgTend.Wait()
        }
}
```

Broadcast Closing Of Cluster

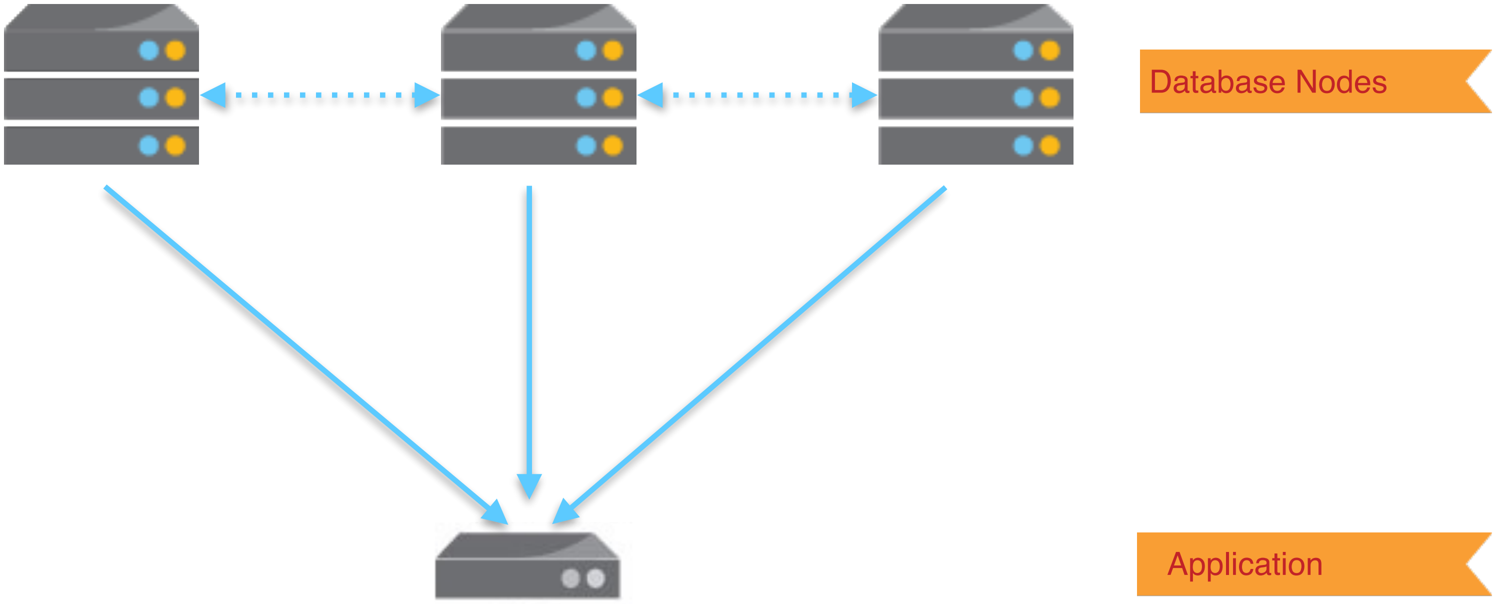
# Cluster Tracking

Break the loop to clean up

```go
func NewCluster(policy *ClientPolicy, hosts []*Host) (*Cluster, error) {
        newCluster := &Cluster{...}

        // start up cluster maintenance go routine
        newCluster.wgTend.Add(1)
        go newCluster.clusterBoss(policy)

        return newCluster, nil
}

func (clstr *Cluster) clusterBoss(policy *ClientPolicy) {
        defer clstr.wgTend.Done()

Loop:
        for {
                select {
                case <-clstr.tendChannel:
                        // tend channel closed
                        break Loop
                case <-time.After(tendInterval):
                        if err := clstr.tend(); err != nil {
                                Logger.Warn(err.Error())
                        }
                }
        }

        // cleanup code goes here
        // close the nodes, ...
}

func (clstr *Cluster) Close() {
        if !clstr.closed.Get() {
                // send close signal to maintenance channel
                close(clstr.tendChannel)

                // wait until tend is over
                clstr.wgTend.Wait()
        }
}
```

# How To Stream Back Data From Multiple Nodes

# Queries



## Queries are Scatter And Gather

1 Goroutine Per Node

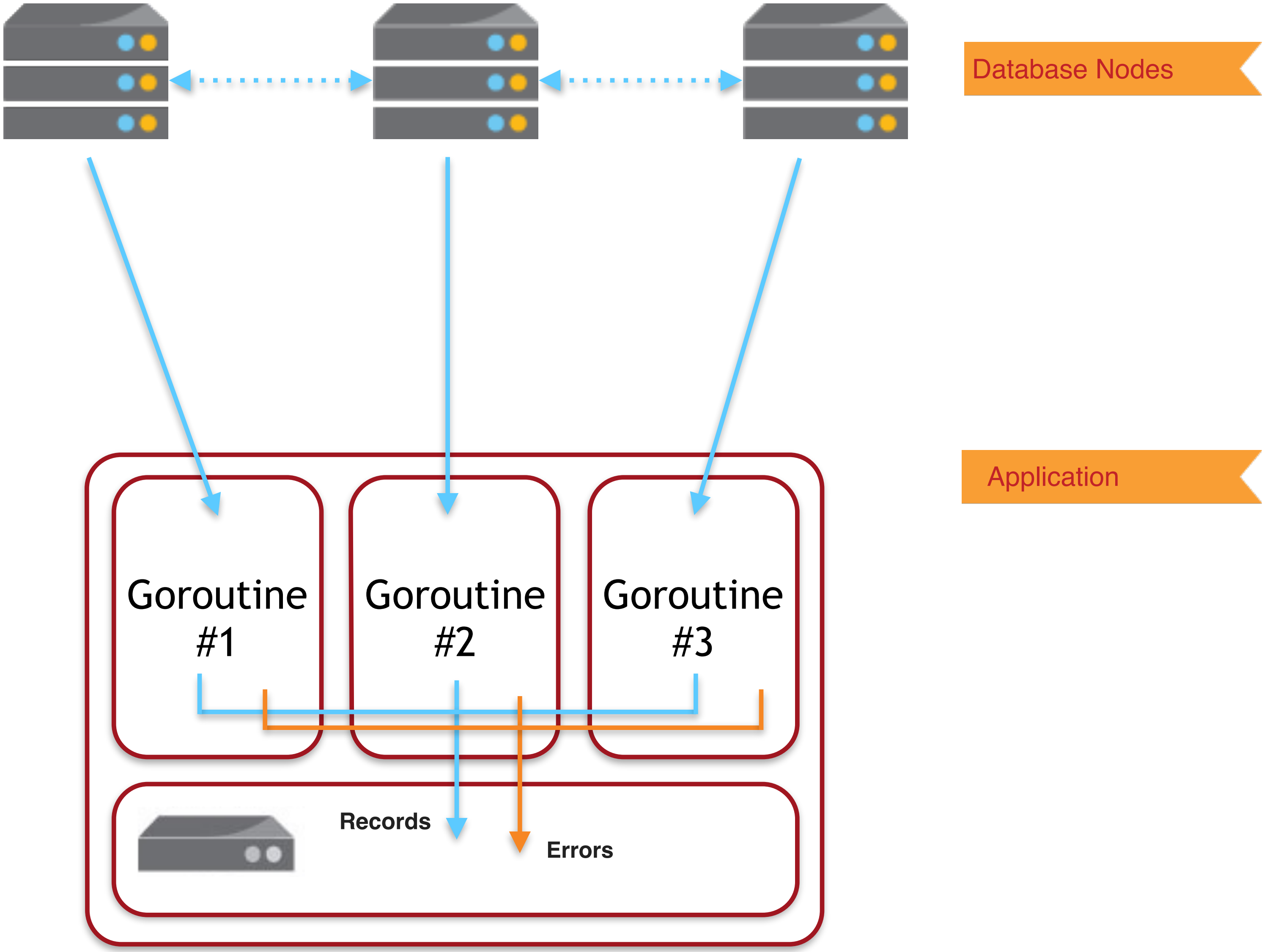
## Data Is Streamed Back

No Cursors

## Errors Are Per Node

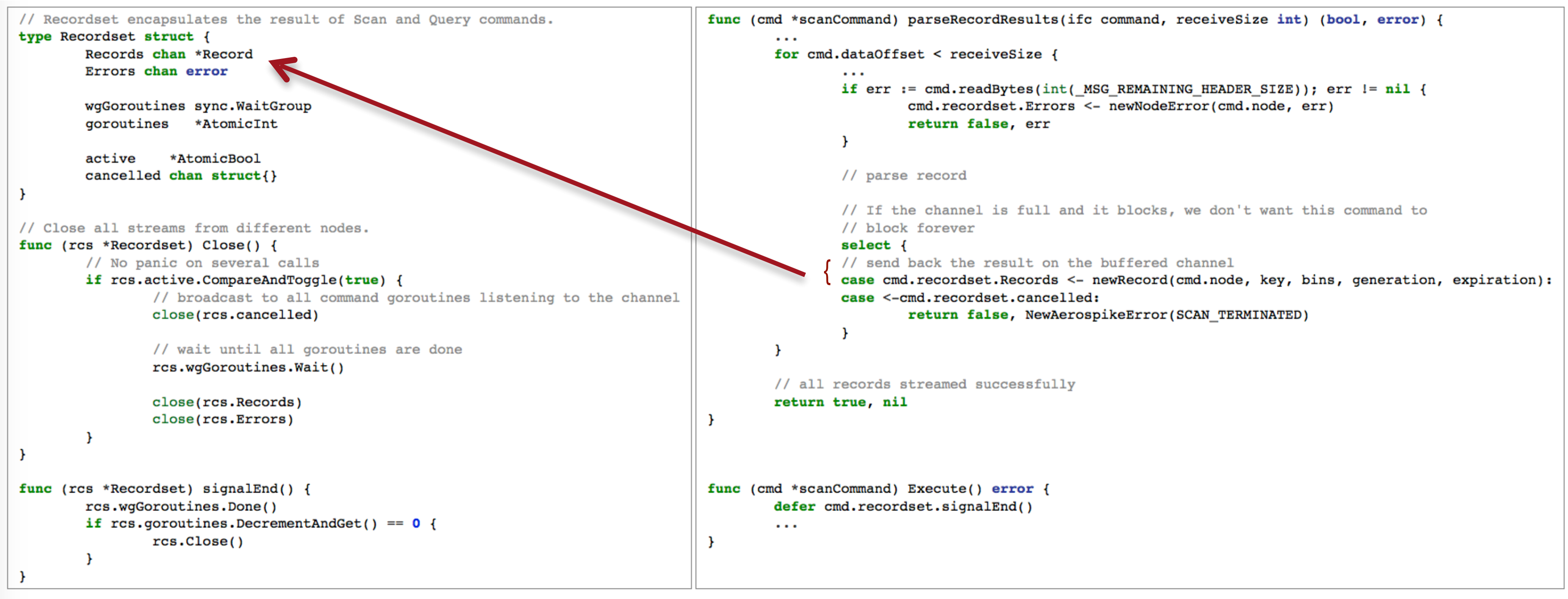
Might Want To Continue With Other Nodes

# Queries



AEROSPIKE

# Queries

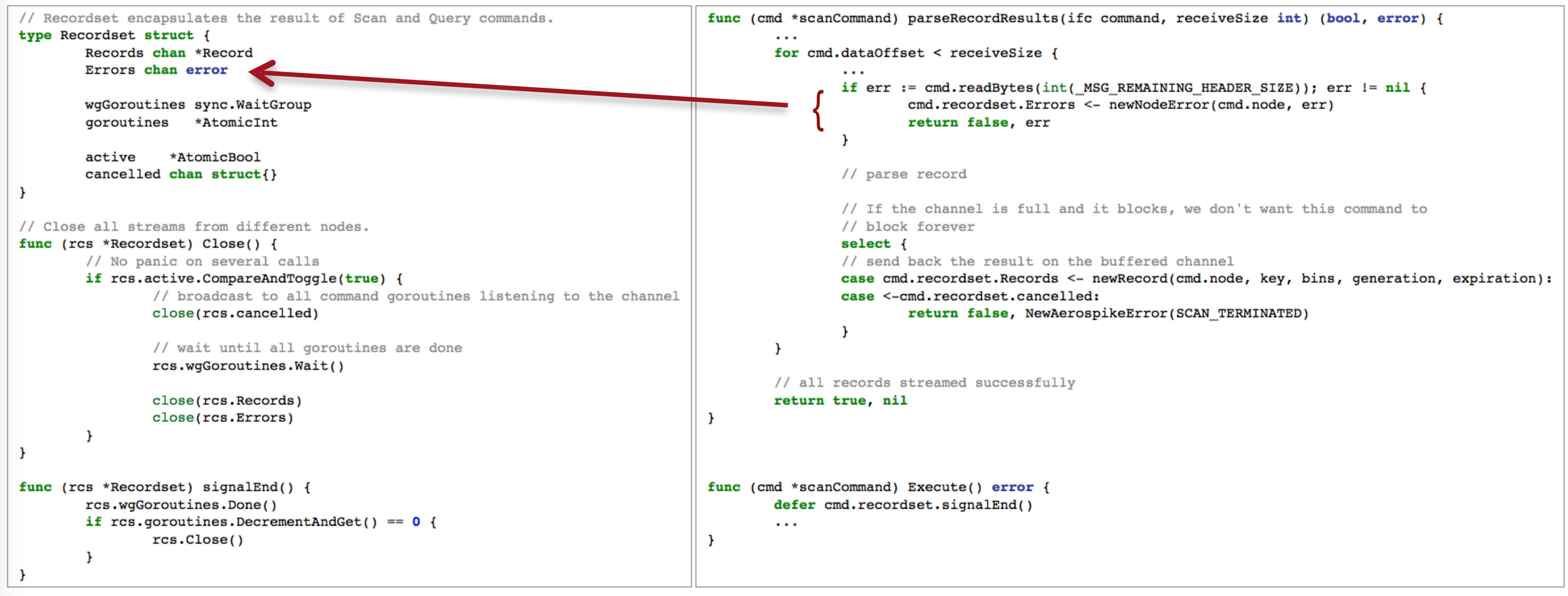


```go
// Recordset encapsulates the result of Scan and Query commands.
type Recordset struct {
        Records chan *Record
        Errors chan error

        wgGoroutines sync.WaitGroup
        goroutines   *AtomicInt

        active    *AtomicBool
        cancelled chan struct{}
}

// Close all streams from different nodes.
func (rcs *Recordset) Close() {
        // No panic on several calls
        if rcs.active.CompareAndToggle(true) {
                // broadcast to all command goroutines listening to the channel
                close(rcs.cancelled)

                // wait until all goroutines are done
                rcs.wgGoroutines.Wait()

                close(rcs.Records)
                close(rcs.Errors)
        }
}

func (rcs *Recordset) signalEnd() {
        rcs.wgGoroutines.Done()
        if rcs.goroutines.DecrementAndGet() == 0 {
                rcs.Close()
        }
}
```

```go
func (cmd *scanCommand) parseRecordResults(ifc command, receiveSize int) (bool, error) {
        ...
        for cmd.dataOffset < receiveSize {
                ...
                if err := cmd.readBytes(int(_MSG_REMAINING_HEADER_SIZE)); err != nil {
                        cmd.recordset.Errors <- newNodeError(cmd.node, err)
                        return false, err
                }

                // parse record

                // If the channel is full and it blocks, we don't want this command to
                // block forever
                select {
                // send back the result on the buffered channel
                case cmd.recordset.Records <- newRecord(cmd.node, key, bins, generation, expiration):
                case <-cmd.recordset.cancelled:
                        return false, NewAerospikeError(SCAN_TERMINATED)
                }
        }

        // all records streamed successfully
        return true, nil
}


func (cmd *scanCommand) Execute() error {
        defer cmd.recordset.signalEnd()
        ...
}
```

Same buffered channel is used for all goroutines

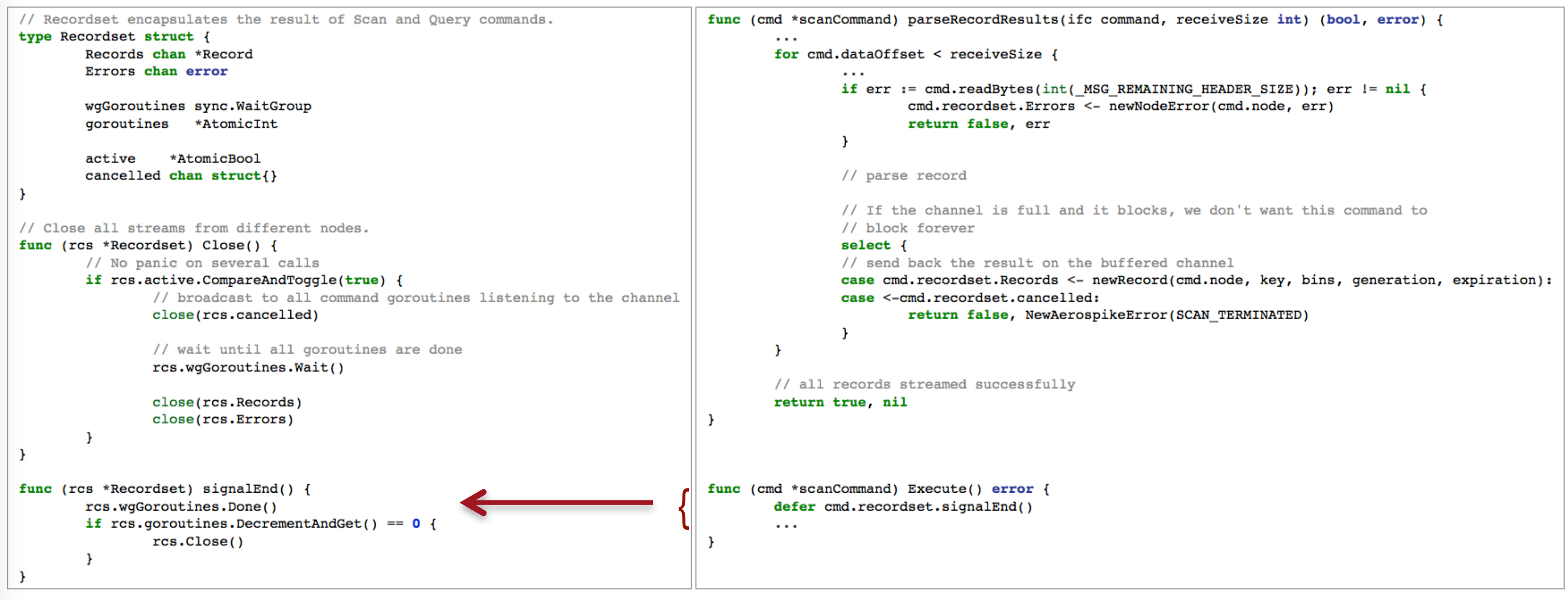
# Queries



```go
// Recordset encapsulates the result of Scan and Query commands.
type Recordset struct {
        Records chan *Record
        Errors chan error

        wgGoroutines sync.WaitGroup
        goroutines   *AtomicInt

        active    *AtomicBool
        cancelled chan struct{}
}

// Close all streams from different nodes.
func (rcs *Recordset) Close() {
        // No panic on several calls
        if rcs.active.CompareAndToggle(true) {
                // broadcast to all command goroutines listening to the channel
                close(rcs.cancelled)

                // wait until all goroutines are done
                rcs.wgGoroutines.Wait()

                close(rcs.Records)
                close(rcs.Errors)
        }
}

func (rcs *Recordset) signalEnd() {
        rcs.wgGoroutines.Done()
        if rcs.goroutines.DecrementAndGet() == 0 {
                rcs.Close()
        }
}
```

```go
func (cmd *scanCommand) parseRecordResults(ifc command, receiveSize int) (bool, error) {
        ...
        for cmd.dataOffset < receiveSize {
                ...
                if err := cmd.readBytes(int(_MSG_REMAINING_HEADER_SIZE)); err != nil {
                        cmd.recordset.Errors <- newNodeError(cmd.node, err)
                        return false, err
                }

                // parse record

                // If the channel is full and it blocks, we don't want this command to
                // block forever
                select {
                // send back the result on the buffered channel
                case cmd.recordset.Records <- newRecord(cmd.node, key, bins, generation, expiration):
                case <-cmd.recordset.cancelled:
                        return false, NewAerospikeError(SCAN_TERMINATED)
                }
        }

        // all records streamed successfully
        return true, nil
}


func (cmd *scanCommand) Execute() error {
        defer cmd.recordset.signalEnd()
        ...
}
```

Errors are sent back on a separate channel



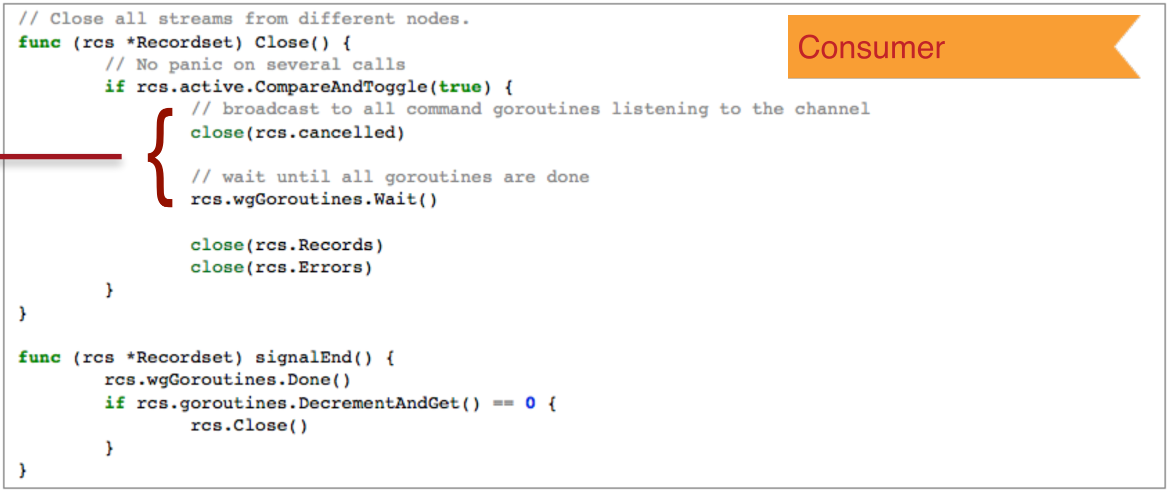
```
// Recordset encapsulates the result of Scan and Query commands.
type Recordset struct {
        Records chan *Record
        Errors chan error

        wgGoroutines sync.WaitGroup
        goroutines   *AtomicInt

        active    *AtomicBool
        cancelled chan struct{}
}

// Close all streams from different nodes.
func (rcs *Recordset) Close() {
        // No panic on several calls
        if rcs.active.CompareAndToggle(true) {
                // broadcast to all command goroutines listening to the channel
                close(rcs.cancelled)

                // wait until all goroutines are done
                rcs.wgGoroutines.Wait()

                close(rcs.Records)
                close(rcs.Errors)
        }
}

func (rcs *Recordset) signalEnd() {
        rcs.wgGoroutines.Done()
        if rcs.goroutines.DecrementAndGet() == 0 {
                rcs.Close()
        }
}
```

```
func (cmd *scanCommand) parseRecordResults(ifc command, receiveSize int) (bool, error) {
        ...
        for cmd.dataOffset < receiveSize {
                ...
                if err := cmd.readBytes(int(_MSG_REMAINING_HEADER_SIZE)); err != nil {
                        cmd.recordset.Errors <- newNodeError(cmd.node, err)
                        return false, err
                }

                // parse record

                // If the channel is full and it blocks, we don't want this command to
                // block forever
                select {
                // send back the result on the buffered channel
                case cmd.recordset.Records <- newRecord(cmd.node, key, bins, generation, expiration):
                case <-cmd.recordset.cancelled:
                        return false, NewAerospikeError(SCAN_TERMINATED)
                }
        }

        // all records streamed successfully
        return true, nil
}


func (cmd *scanCommand) Execute() error {
        defer cmd.recordset.signalEnd()
        ...
}
```

Defer on all producers will ensure `Close()` is called at least once

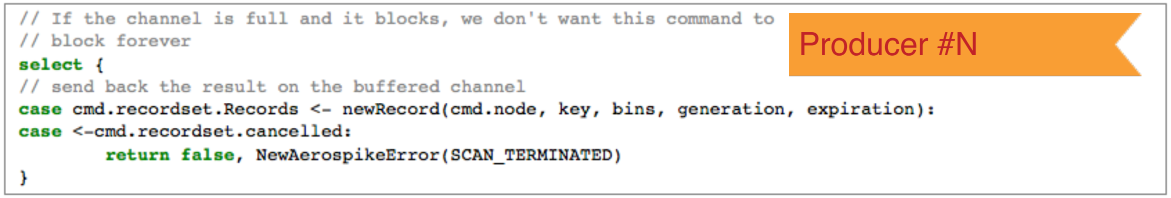
# Queries

Will not deadlock, all producers will return



Consumer

```
// Close all streams from different nodes.
func (rcs *Recordset) Close() {
        // No panic on several calls
        if rcs.active.CompareAndToggle(true) {
                // broadcast to all command goroutines listening to the channel
                close(rcs.cancelled)

                // wait until all goroutines are done
                rcs.wgGoroutines.Wait()

                close(rcs.Records)
                close(rcs.Errors)
        }
}

func (rcs *Recordset) signalEnd() {
        rcs.wgGoroutines.Done()
        if rcs.goroutines.DecrementAndGet() == 0 {
                rcs.Close()
        }
}
```



Producer #N

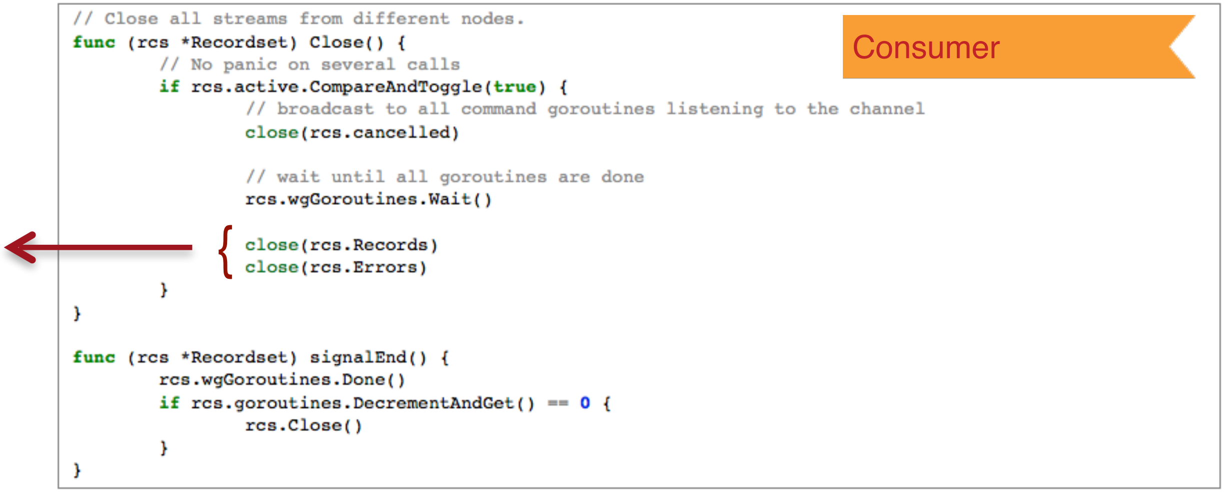
```
// If the channel is full and it blocks, we don't want this command to
// block forever
select {
// send back the result on the buffered channel
case cmd.recordset.Records <- newRecord(cmd.node, key, bins, generation, expiration):
case <-cmd.recordset.cancelled:
        return false, NewAerospikeError(SCAN_TERMINATED)
}
```

The Above code will never panic, or block indefinitely:

- **`defer`** on **`Execute()`** will ensure **`Close()`** is called at least once
- When **`cancelled`** channel is closed, even if the **`Records`** channel is full and blocked, **`select`** will go through and return from goroutine

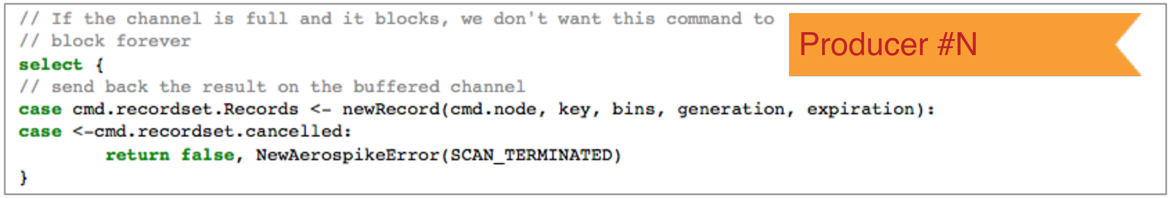
# Queries

Will not panic, all producers have already stopped



Consumer

```go
// Close all streams from different nodes.
func (rcs *Recordset) Close() {
        // No panic on several calls
        if rcs.active.CompareAndToggle(true) {
                // broadcast to all command goroutines listening to the channel
                close(rcs.cancelled)

                // wait until all goroutines are done
                rcs.wgGoroutines.Wait()

                close(rcs.Records)
                close(rcs.Errors)
        }
}

func (rcs *Recordset) signalEnd() {
        rcs.wgGoroutines.Done()
        if rcs.goroutines.DecrementAndGet() == 0 {
                rcs.Close()
        }
}
```



Producer #N

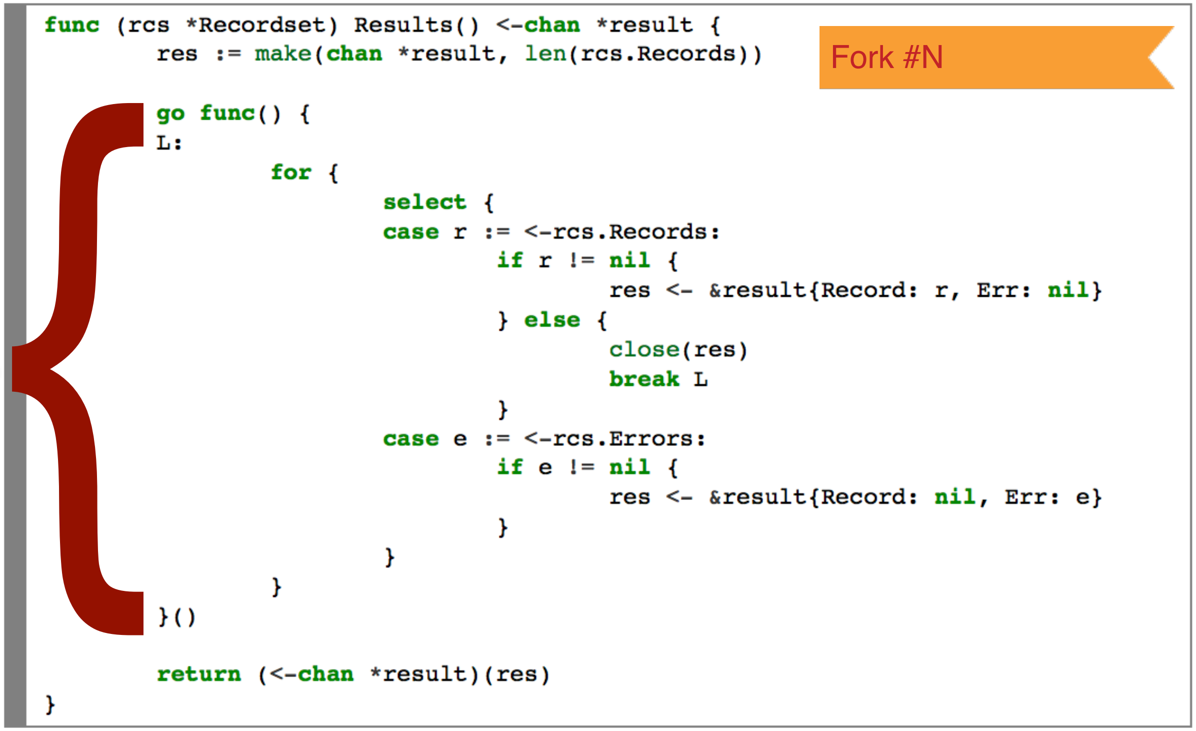
```go
// If the channel is full and it blocks, we don't want this command to
// block forever
select {
// send back the result on the buffered channel
case cmd.recordset.Records <- newRecord(cmd.node, key, bins, generation, expiration):
case <-cmd.recordset.cancelled:
        return false, NewAerospikeError(SCAN_TERMINATED)
}
```

The Above code will never panic, or block indefinitely:

- **`defer`** on **`Execute()`** will ensure **`Close()`** is called at least once
- When **`cancelled`** channel is closed, even if the **`Records`** channel is full and blocked, **`select`** will go through and return from goroutine

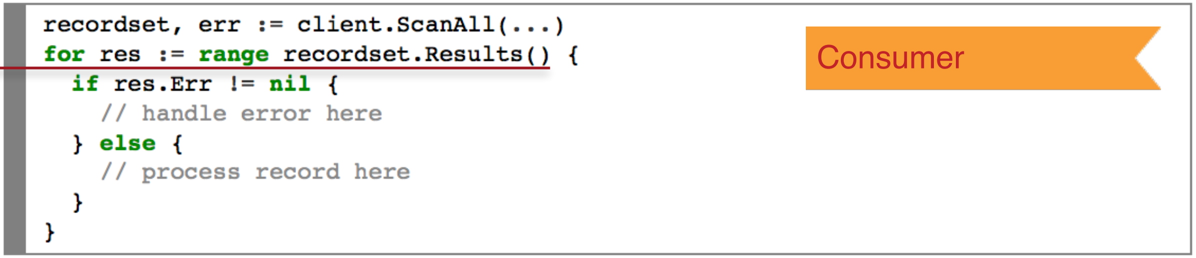
# Consumption Pattern

Supports numerous consumers



```
func (rcs *Recordset) Results() <-chan *result {
        res := make(chan *result, len(rcs.Records))

        go func() {
        L:
                for {
                        select {
                        case r := <-rcs.Records:
                                if r != nil {
                                        res <- &result{Record: r, Err: nil}
                                } else {
                                        close(res)
                                        break L
                                }
                        case e := <-rcs.Errors:
                                if e != nil {
                                        res <- &result{Record: nil, Err: e}
                                }
                        }
                }
        }()

        return (<-chan *result)(res)
}
```

**Range** on Channel, get either a **`Record`** or **`error`**



```
recordset, err := client.ScanAll(...)
for res := range recordset.Results() {
  if res.Err != nil {
    // handle error here
  } else {
    // process record here
  }
}
```

Errors do not mean the stream has ended. They carry information about ***what*** went wrong, and ***where***

# Take away

- **Channels are a good abstraction for data streaming**
    - Easier to avoid deadlocks

- **Use the least number of channels and goroutines**
    - Multiplex on demand
    - Goroutines are cheap, but not free
    - Pass channels throughout API, don’t ask for new ones

# How To Win Against The Garbage Collector

# Don’t Create Garbage

- **Avoid Memory Allocations**
    - Might be trickier than you think!

- **Pool All Objects**
    - Buffers
    - Hash Objects
    - Network Connections
    - Reusable Intermediate / Temporary Objects
    - Random Number Generators

- `sync.Pool` **is not ideal to pool long living objects**
    - Integrated into GC, sweeps and deallocates objects too often

- **LIFO pools tend to do better regarding CPU cache**
    - Channels for pool implementation still perform well though

# Don’t Reallocate Memory

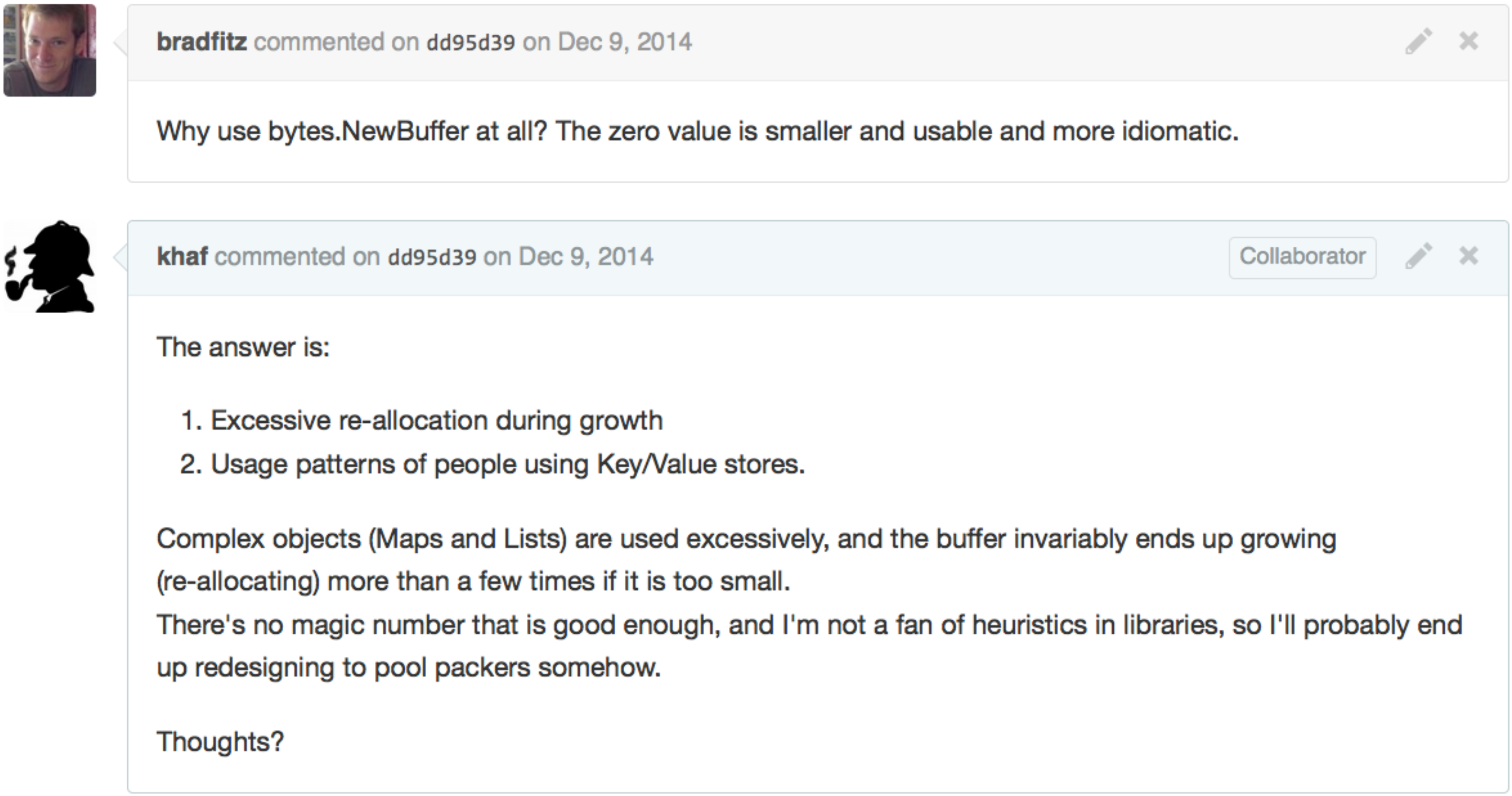
- **Avoid Memory Allocations**
  - Might be trickier than you think!
- **All following code snippets allocate memory:**

```
b := new(bytes.Buffer)
longBuf := bytes.Repeat([]byte{32}, 100)

// Allocation happens in Write to Grow the buffer
b.Write([]byte{32})

// Re-allocation happens in Write to Grow the buffer
b.Write(longBuf)
```

```
// allocation happens inside Trim
strings.Trim(" Trim this string ", " ")
```

```
h := ripemd160.New()
h.Write(data)

// allocation happens inside Sum()
h.Sum(nil)

// Why? To support the general case
func (d *digest) Sum(in []byte) []byte {
        // Make a copy of d0 so that caller can keep writing and summing.
        d := *d0
```

# Don’t Reallocate Memory



**bradfitz** commented on `dd95d39` on Dec 9, 2014

Why use bytes.NewBuffer at all? The zero value is smaller and usable and more idiomatic.

**khaf** commented on `dd95d39` on Dec 9, 2014 Collaborator

The answer is:

1. Excessive re-allocation during growth
2. Usage patterns of people using Key/Value stores.

Complex objects (Maps and Lists) are used excessively, and the buffer invariably ends up growing (re-allocating) more than a few times if it is too small.
There's no magic number that is good enough, and I'm not a fan of heuristics in libraries, so I'll probably end up redesigning to pool packers somehow.

Thoughts?

# Pools

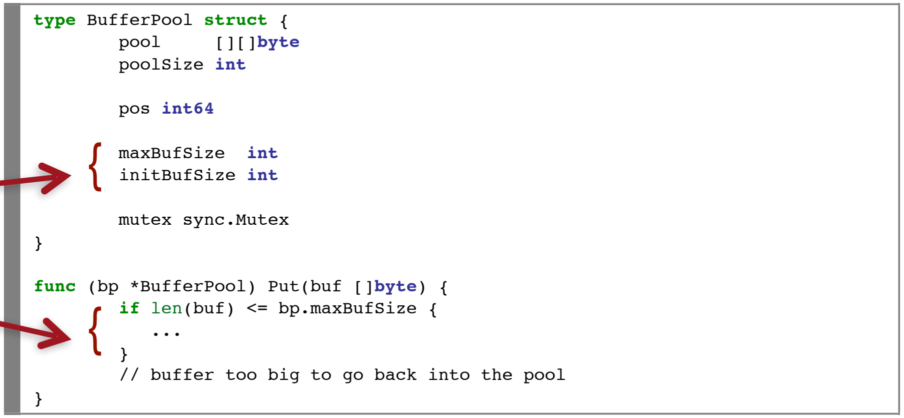
**Indeterministic Pools are Resource Leaks You Haven't Found Yet!** *

* Please apply wisdom to the quip above

# Pools

- **Make Pools Deterministic**
- **Let User Decide About Parameters Per Use-Case**
- **Document The Above!**

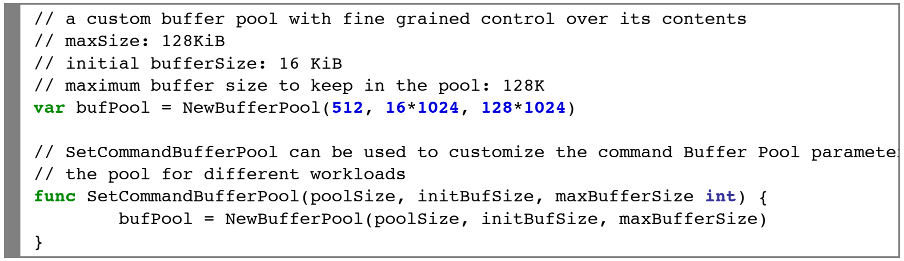
Set & Enforce Limits for Pooled Buffers



```
type BufferPool struct {
        pool     [][]byte
        poolSize int

        pos int64

        maxBufSize  int
        initBufSize int

        mutex sync.Mutex
}

func (bp *BufferPool) Put(buf []byte) {
        if len(buf) <= bp.maxBufSize {
           ...
        }
        // buffer too big to go back into the pool
}
```

Set Sensible Defaults

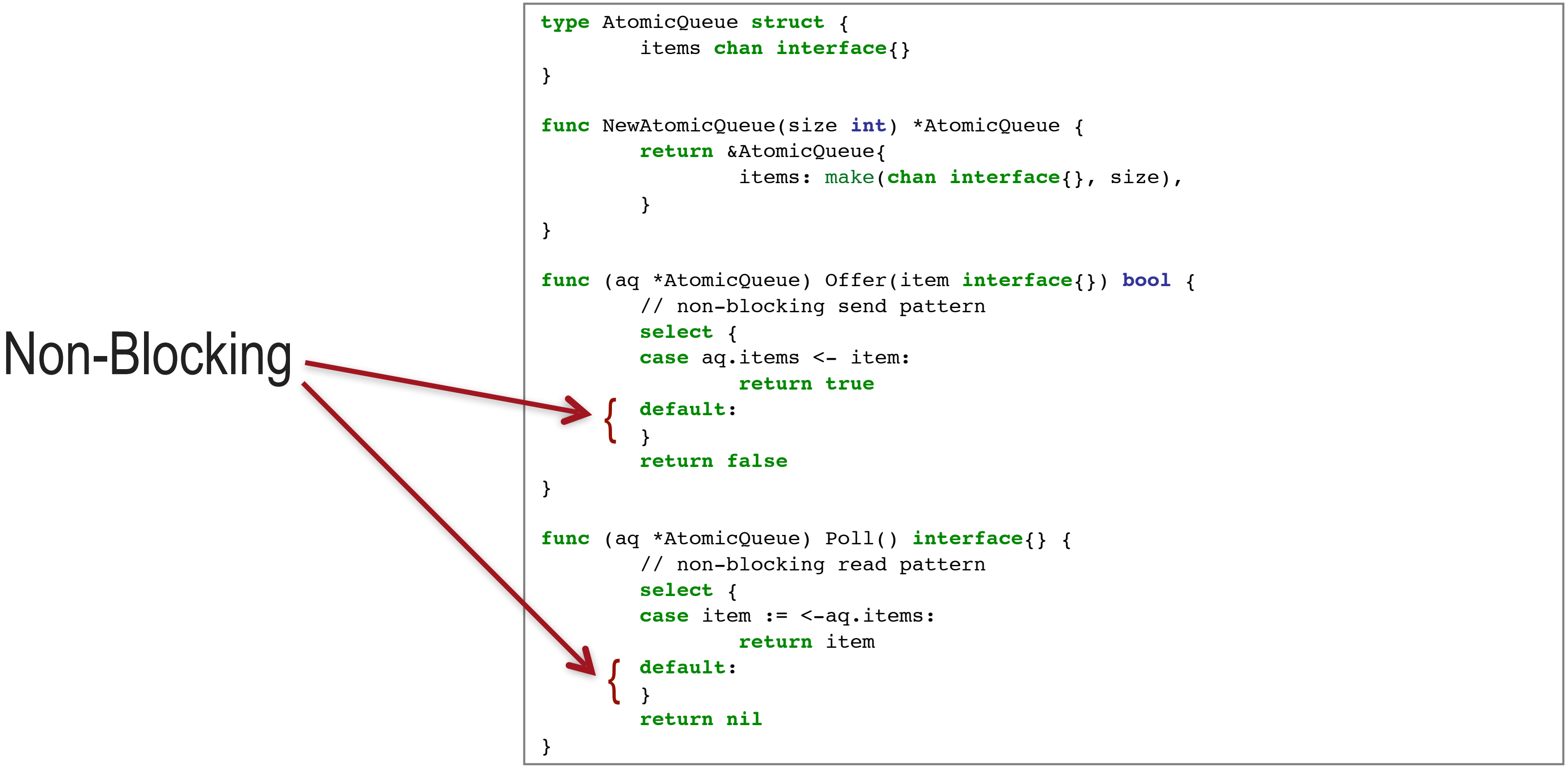
Let Users Tweak  Per Use-Case



```
// a custom buffer pool with fine grained control over its contents
// maxSize: 128KiB
// initial bufferSize: 16 KiB
// maximum buffer size to keep in the pool: 128K
var bufPool = NewBufferPool(512, 16*1024, 128*1024)

// SetCommandBufferPool can be used to customize the command Buffer Pool paramete
// the pool for different workloads
func SetCommandBufferPool(poolSize, initBufSize, maxBufferSize int) {
        bufPool = NewBufferPool(poolSize, initBufSize, maxBufferSize)
}
```

# Non-Blocking Sync’d Queues Using Channels

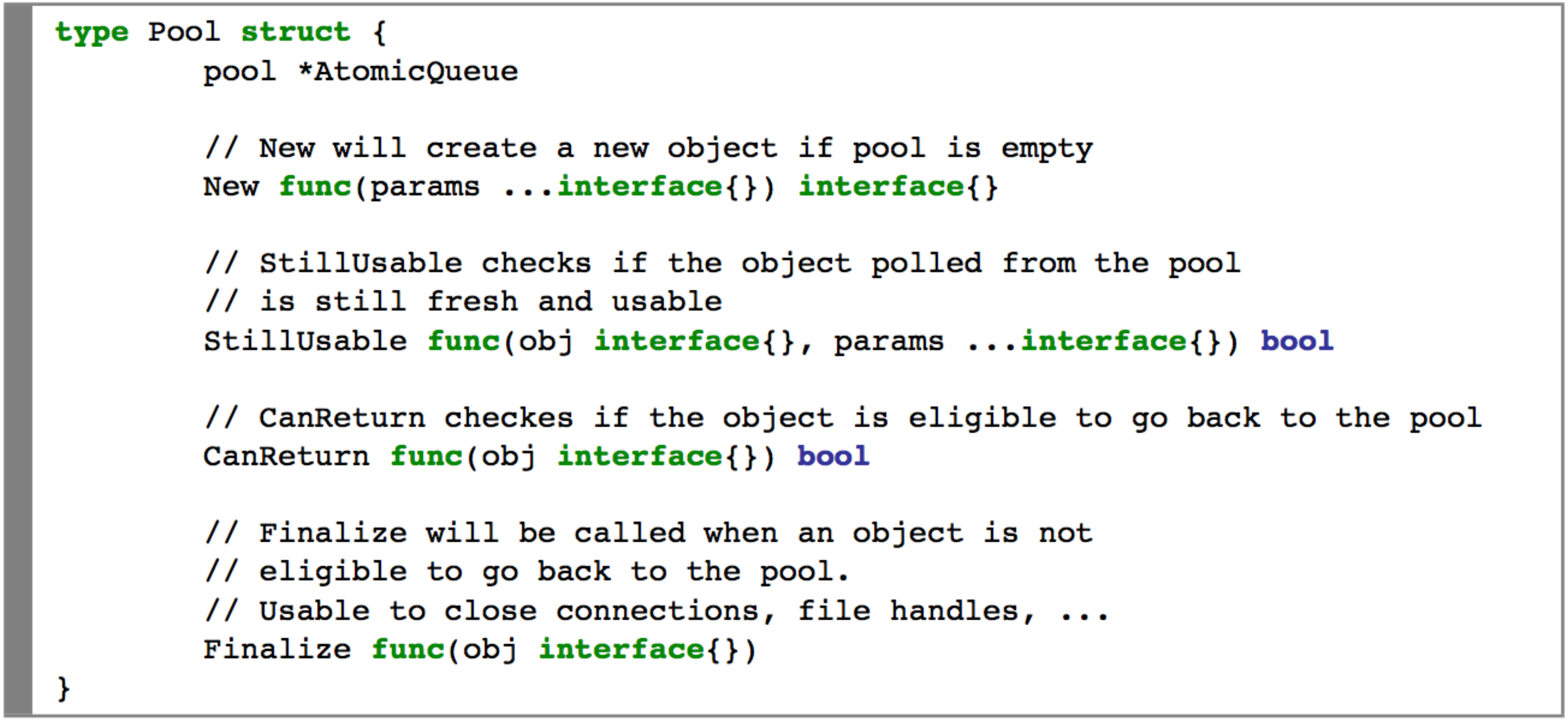
## Non-blocking Channel Read / Write Is Possible



Non-Blocking

```go
type AtomicQueue struct {
        items chan interface{}
}

func NewAtomicQueue(size int) *AtomicQueue {
        return &AtomicQueue{
                items: make(chan interface{}, size),
        }
}

func (aq *AtomicQueue) Offer(item interface{}) bool {
        // non-blocking send pattern
        select {
        case aq.items <- item:
                return true
        default:
        }
        return false
}

func (aq *AtomicQueue) Poll() interface{} {
        // non-blocking read pattern
        select {
        case item := <-aq.items:
                return item
        default:
        }
        return nil
}
```

# Generic Pool Using Sync’d Queue



```go
type Pool struct {
        pool *AtomicQueue

        // New will create a new object if pool is empty
        New func(params ...interface{}) interface{}

        // StillUsable checks if the object polled from the pool
        // is still fresh and usable
        StillUsable func(obj interface{}, params ...interface{}) bool

        // CanReturn checkes if the object is eligible to go back to the pool
        CanReturn func(obj interface{}) bool

        // Finalize will be called when an object is not
        // eligible to go back to the pool.
        // Usable to close connections, file handles, ...
        Finalize func(obj interface{})
}
```

# How Does It Perform?

# Comparison vs C and Java

## Data Type: String of size 2000

Transactions Per Second

| | 100% WRITE | 100% READ | 50% READ<br>50% WRITE |
|---|---|---|---|
| C | 261K | 308K | 273K |
| JAVA | 180K | 280K | 220K |
| GO | 306K | 238K | 276K |

🟩 Fastest 🟥 Slowest

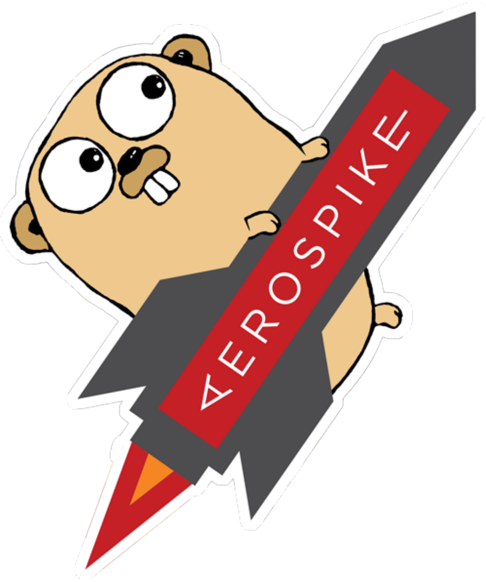
# Comparison vs C and Java

## Data Type: Integer (8 Bytes)

Transactions Per Second

| | *100% WRITE* | *100% READ* | *50% READ<br>50% WRITE* |
|---|---|---|---|
| *C* | 🟥 *299K* | *334K* | 🟥 *307K* |
| *JAVA* | *308K* | 🟩 *339K* | *325K* |
| *GO* | 🟩 *338K* | 🟥 *328K* | 🟩 *330K* |

🟩 Fastest 🟥 Slowest



# Thank you

**@sunilvirus**

sunil@aerospike.com

**@parshua**

khosrow@aerospike.com