mongoDB

Scaling for Humongous amounts of data with MongoDB

Alvin Richards

Technical Director, EMEA
alvin@10gen.com
@jonnyeight
alvinonmongodb.com

10gen | the MongoDB company

From here...

10gen | the MongoDB company

http://bit.ly/OT71M4

...to here...

10gen | the MongoDB company

http://bit.ly/Oxcsis

...without one of these.

http://bit.ly/cnP77L

Warning!

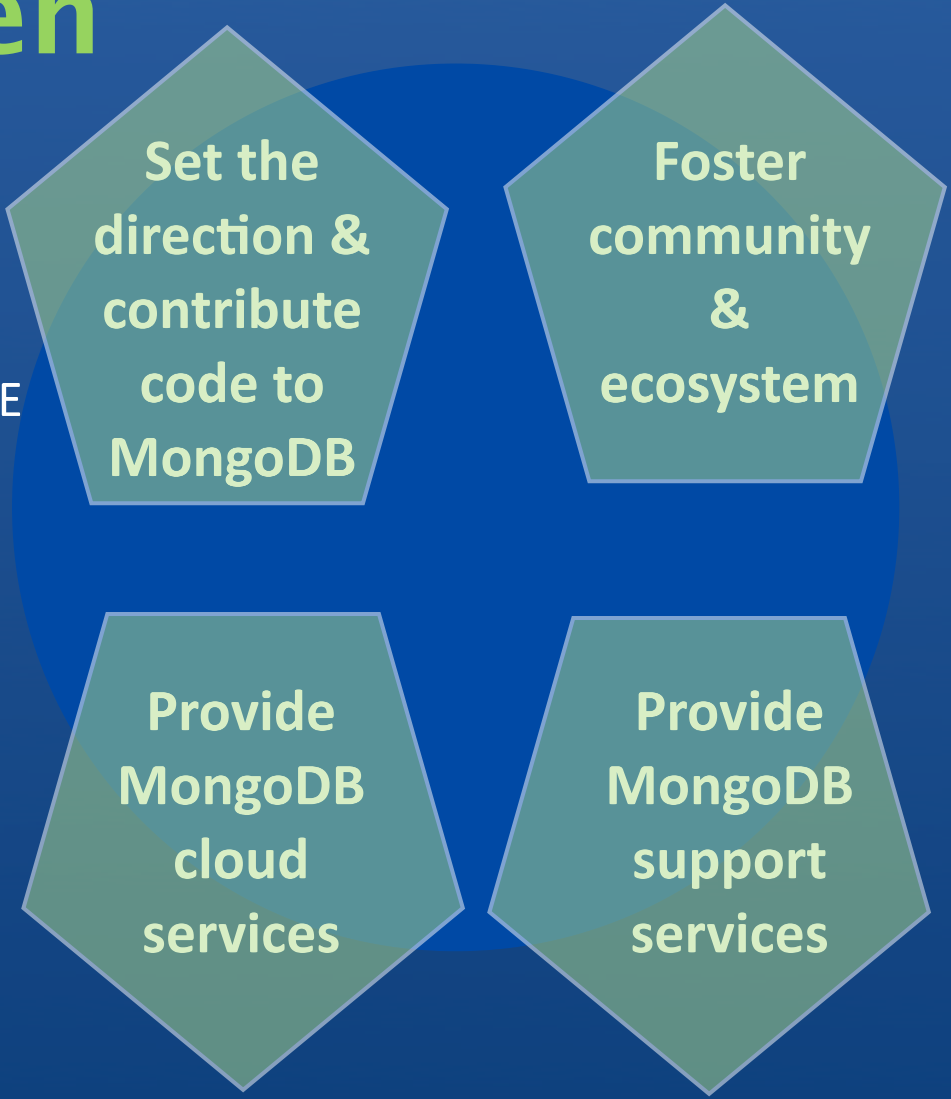
- This is a technical talk
- But MongoDB is very simple!

Solving real world data problems with MongoDB

- Effective schema design for scaling
  - Linking versus embedding
  - Bucketing
  - Time series
- Implications of sharding keys with alternatives
- Read scaling through replication
- Challenges of eventual consistency

A quick word from MongoDB sponsors, 10gen

- Founded in 2007
  - Dwight Merriman, Eliot Horowitz
- $73M+ in funding
  - Flybridge, Sequoia, Union Square, NEA
- Worldwide Expanding Team
  - 170+ employees
  - NY, CA, UK and Australia



Set the direction & contribute code to MongoDB

Foster community & ecosystem

Provide MongoDB cloud services

Provide MongoDB support services

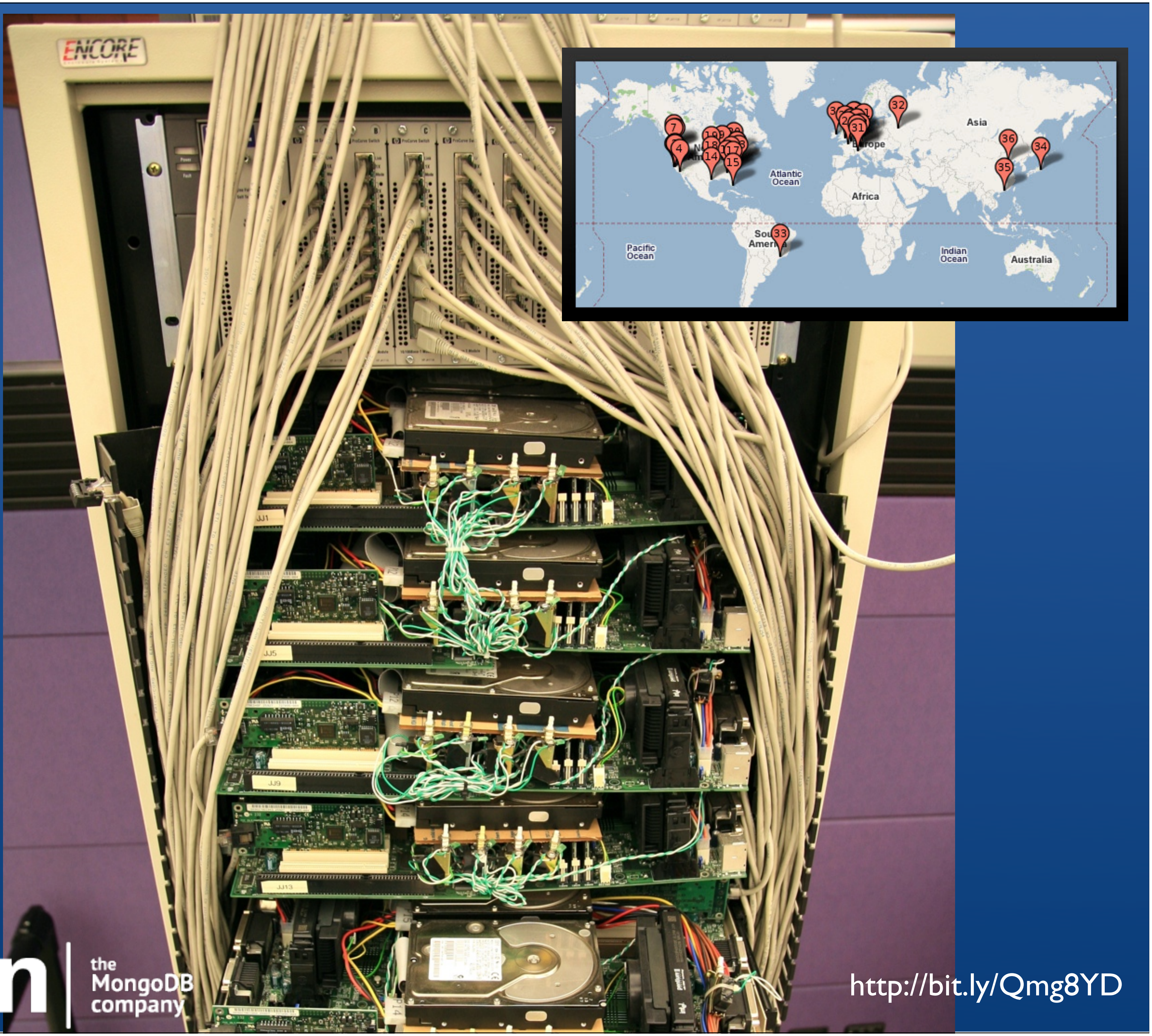
10gen | the MongoDB company

Since the dawn of the RDBMS

	1970	2012
Main memory	Intel 1103, 1k bits	4GB of RAM costs $25.99
Mass storage	IBM 3330 Model 1, 100 MB	3TB Superspeed USB for $129
Microprocessor	Nearly – 4004 being developed; 4 bits and 92,000 instructions per second	Westmere EX has 10 cores, 30MB L3 cache, runs at 2.4GHz

More recent changes

	A decade ago	Now
Faster	Buy a bigger server	Buy more servers
Faster storage	A SAN with more spindles	SSD
More reliable storage	More expensive SAN	More copies of local storage
Deployed in	Your data center	The cloud – private or public
Large data set	Millions of rows	Billions to trillions of rows
Development	Waterfall	Iterative



http://bit.ly/Qmg8YD

Is Scaleout Mission Impossible?

- What about the CAP Theorem?
  - Brewer's theorem
  - Consistency, Availability, Partition Tolerance

- It says if a distributed system is partitioned, you can’t be able to update everywhere and have consistency

- So, either allow inconsistency or limit where updates can be applied

What MongoDB solves

Agility

- Applications store complex data that is easier to model as **documents**
- **Schemaless** DB enables faster development cycles

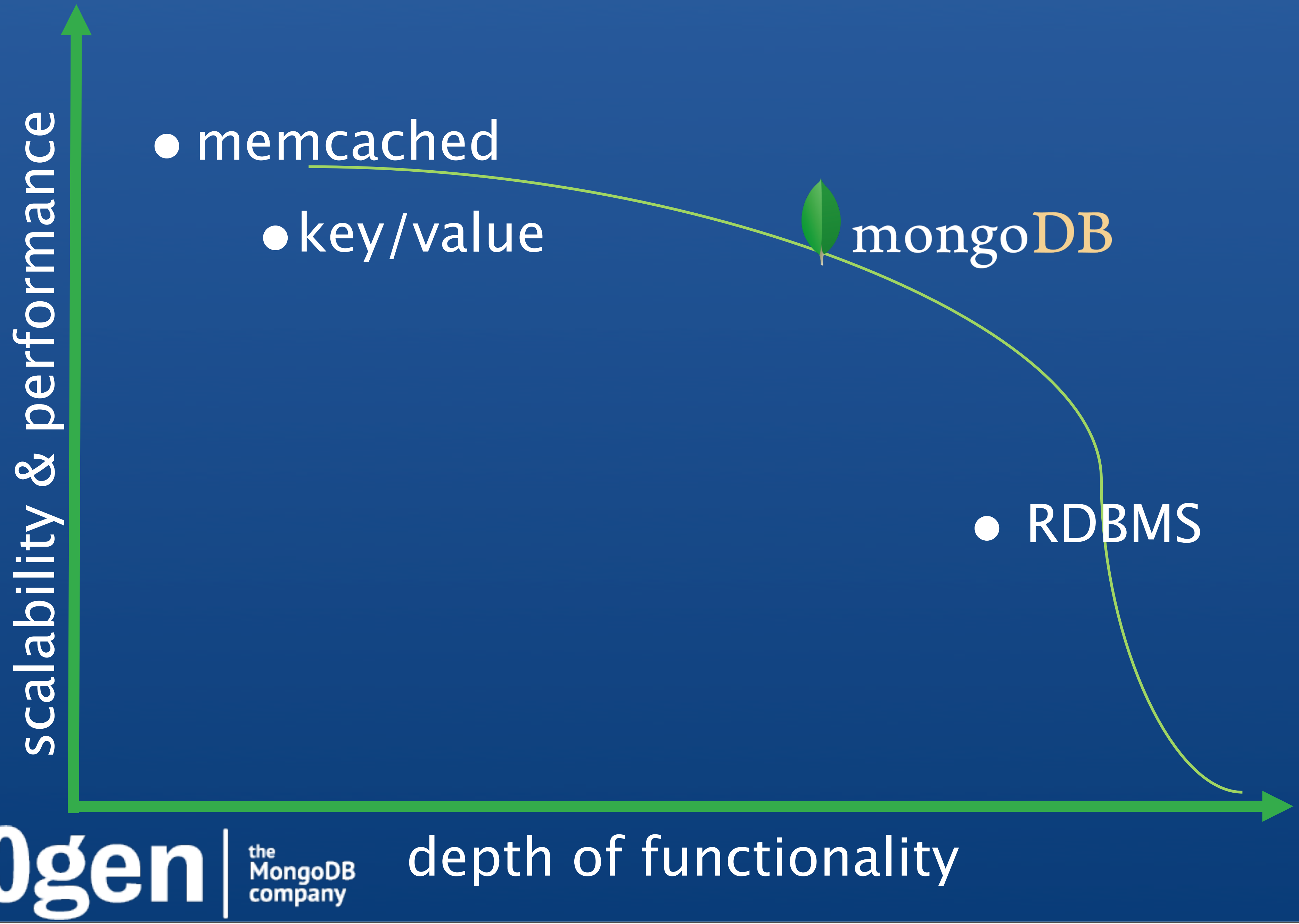
Flexibility

- Relaxed transactional semantics enable **easy scale out**
- **Auto Sharding** for scale down and scale up

Cost

- Cost effective operationalize abundant data (clickstreams, logs, tweets, ...)

Design Goal of MongoDB



Schema Design at Scale

10gen | the MongoDB company

Design Schema for Twitter

- Model each users activity stream
- Users
    - Name, email address, display name
- Tweets
    - Text
    - Who
    - Timestamp

Solution A
Two Collections - Normalized

```
// users - one doc per user
{ _id:     "alvin",
  email:   "alvin@10gen.com",
  display: "jonnyeight"
}

// tweets - one doc per user per tweet
{
  user:  "bob",
  tweet: "20111209-1231",
  text:  "Best Tweet Ever!",
  ts:    ISODate("2011-09-18T09:56:06.298Z")
}
```

Solution B
Embedded - Array of Objects

```
// users - one doc per user with all tweets
{  _id:     "alvin",
   email:   "alvin@10gen.com",
   display: "jonnyeight",
   tweets: [
     {
         user:  "bob",
         tweet: "20111209-1231",
         text:  "Best Tweet Ever!",
         ts:    ISODate("2011-09-18T09:56:06.298Z")
     }
  ]
}
```

Embedding

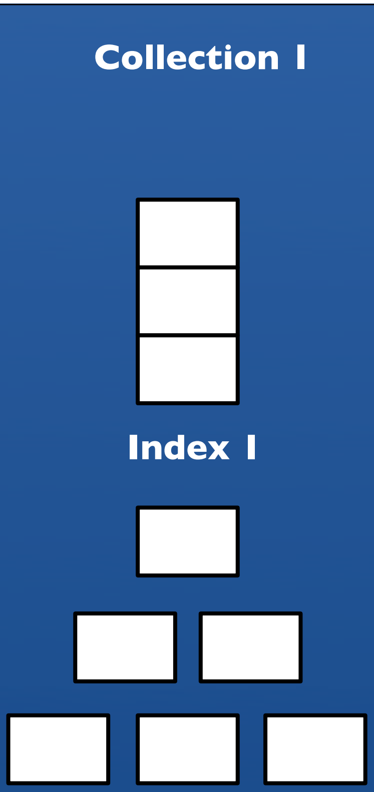
- Great for read performance
- One seek to load entire object
- One roundtrip to database
- Object grows over time when adding child objects

Linking or Embedding?

Linking can make some queries easy

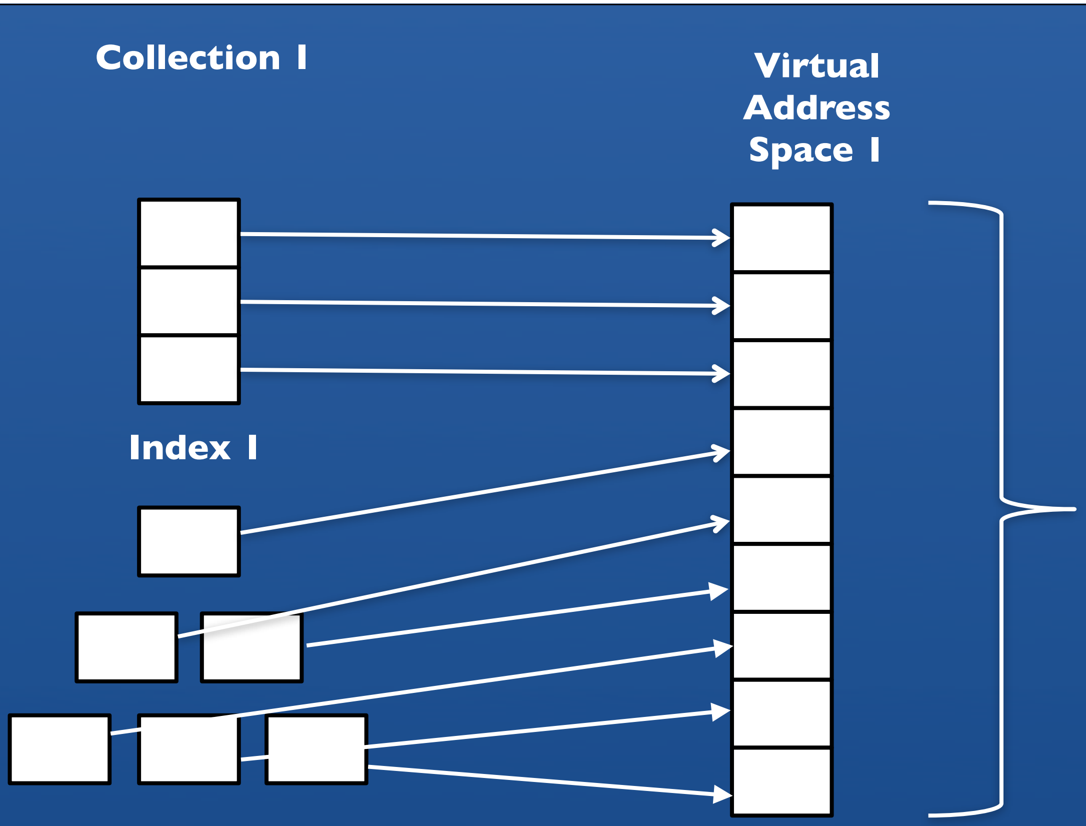
```
// Find latest 50 tweets for "alvin"
> db.tweets.find( { _id: "alvin" } )
           .sort( { ts: -1 } )
           .limit(10)
```

But what effect does this have on the systems?

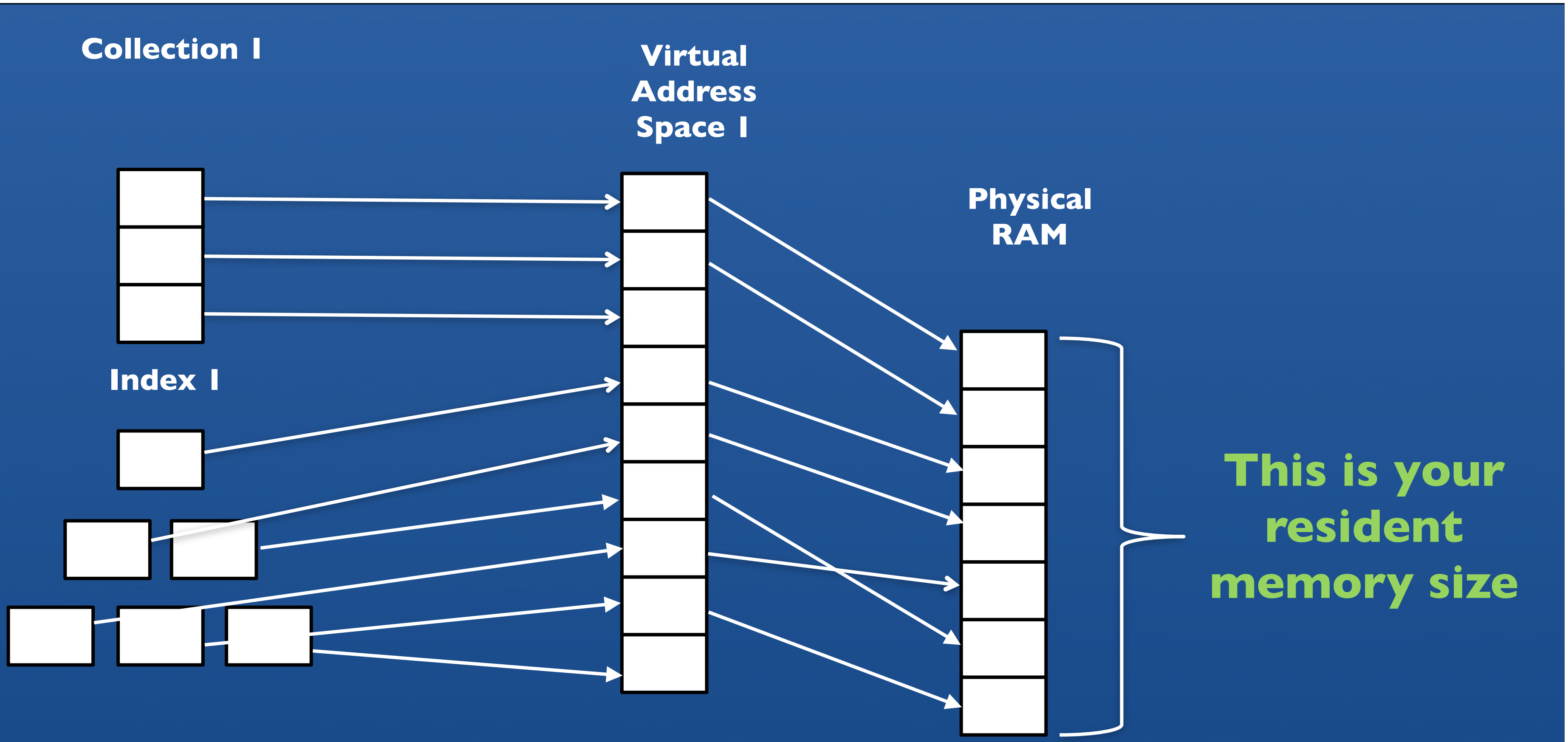


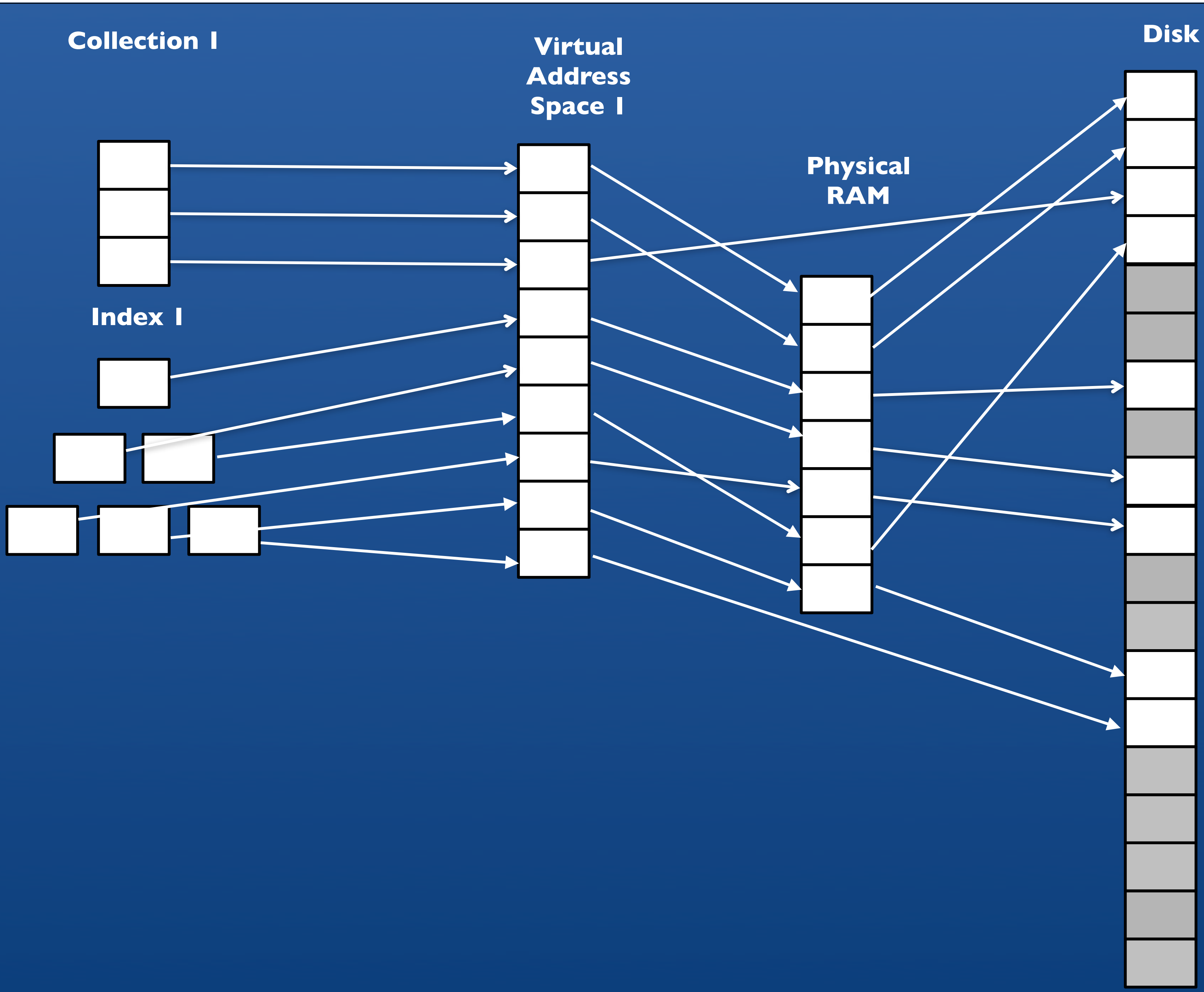
Collection 1

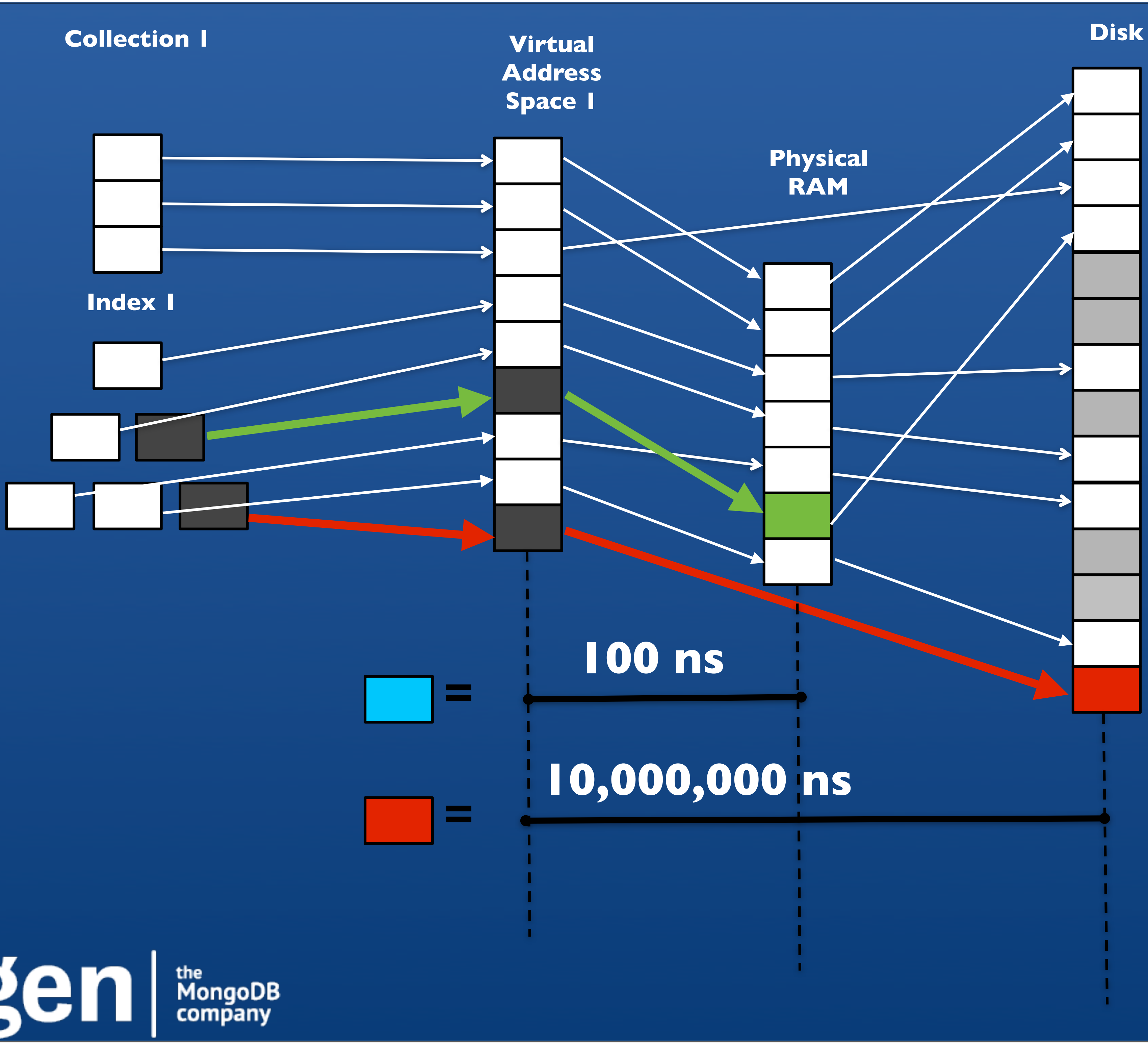
Index 1



This is your virtual memory size (mapped)


Collection 1
Virtual Address Space 1
Physical RAM
Index 1
This is your resident memory size


Collection 1
Virtual Address Space 1
Physical RAM
Disk
Index 1



Collection 1

Virtual Address Space 1

Physical RAM

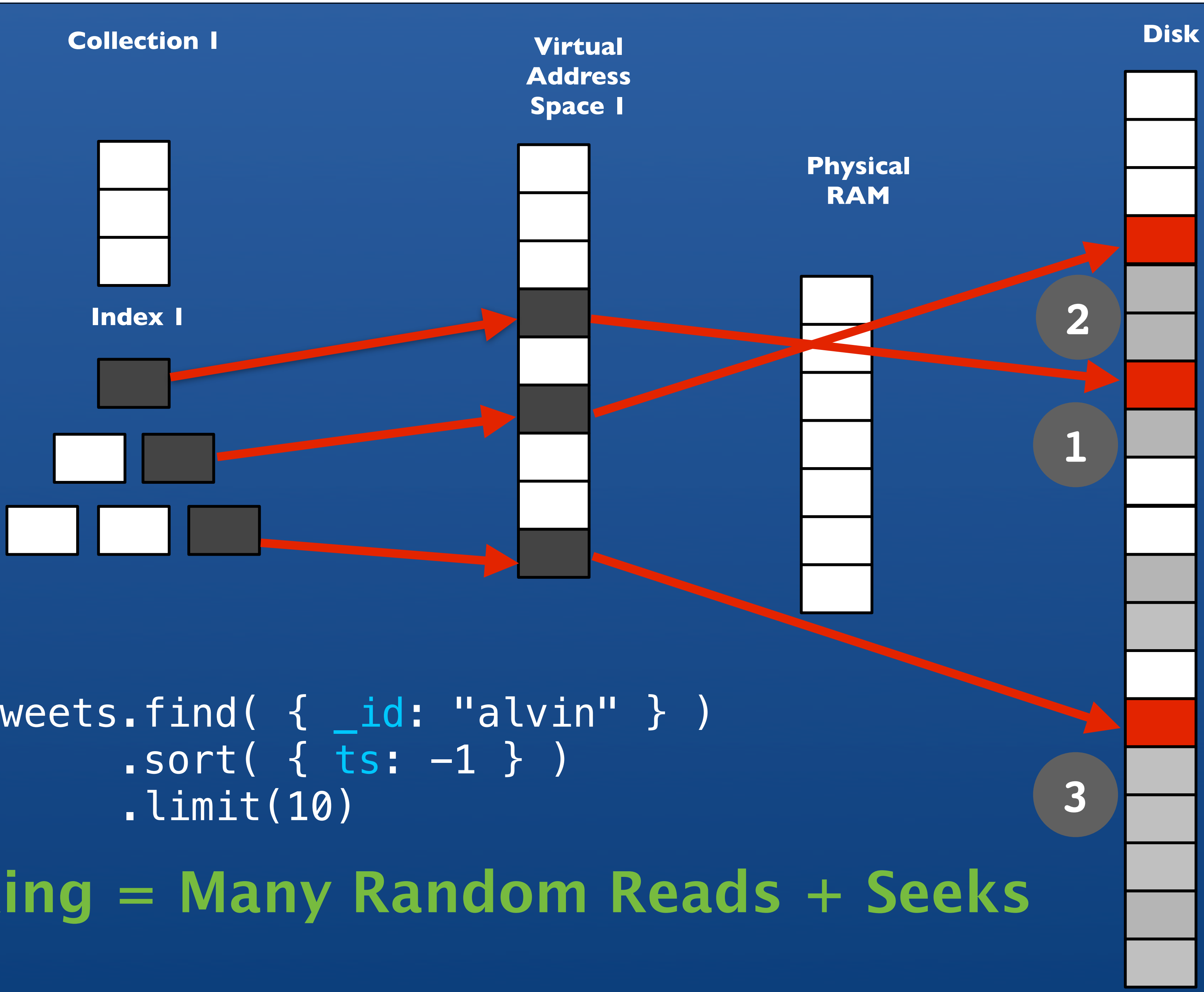
Disk

Index 1

🟦 = 100 ns

🟥 = 10,000,000 ns

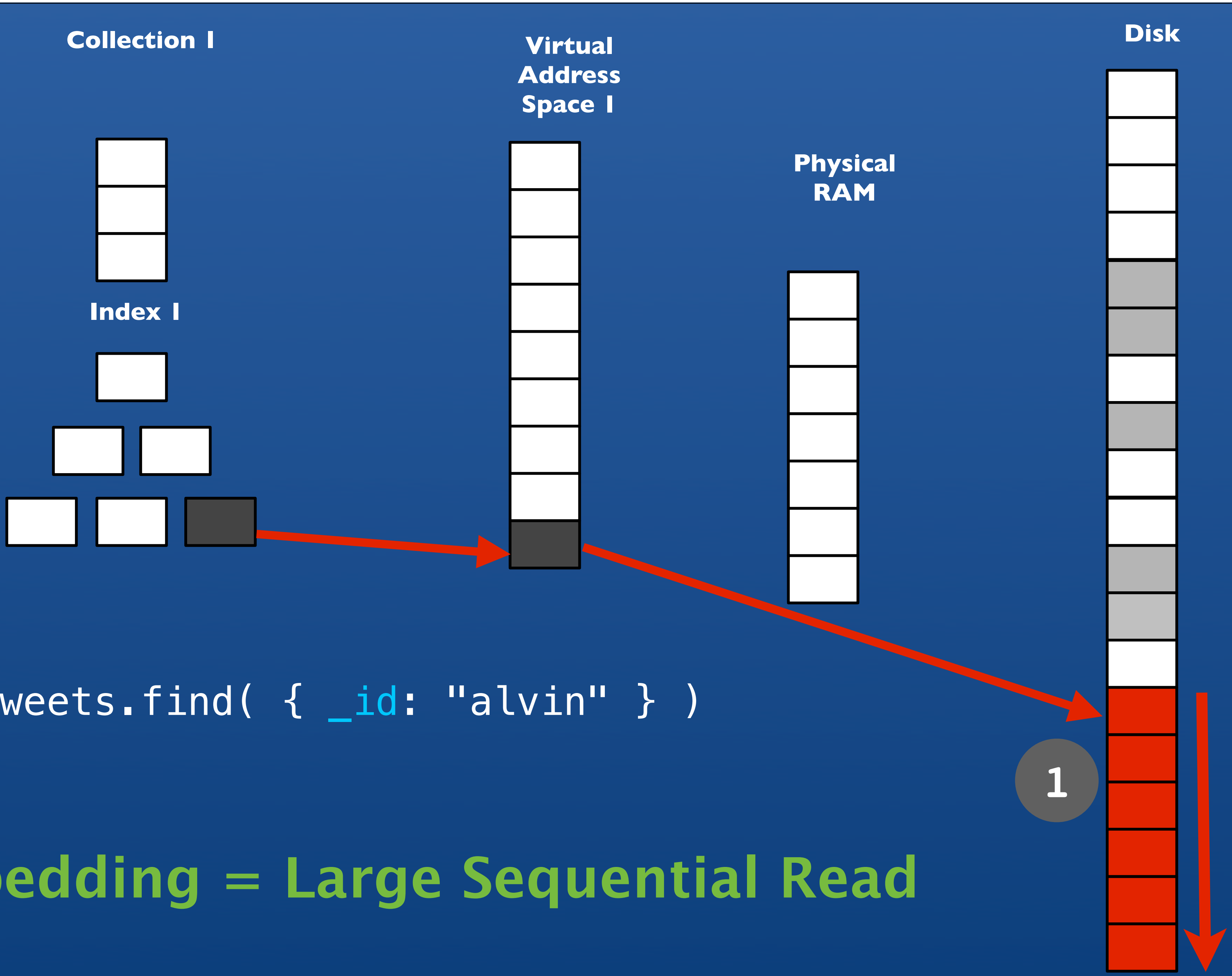
10gen | the MongoDB company



```
db.tweets.find( { _id: "alvin" } )
            .sort( { ts: -1 } )
            .limit(10)
```

Linking = Many Random Reads + Seeks

10gen | the MongoDB company



Collection I

Virtual Address Space I

Physical RAM

Disk

Index I

```
db.tweets.find( { _id: "alvin" } )
```

Embedding = Large Sequential Read

Problems

- Large sequential reads
  - Good: Disks are great at Sequential reads
  - Bad: May read too much data

- Many Random reads
  - Good: Easy of query
  - Bad: Disks are poor at Random reads (SSD?)

Solution C
Buckets

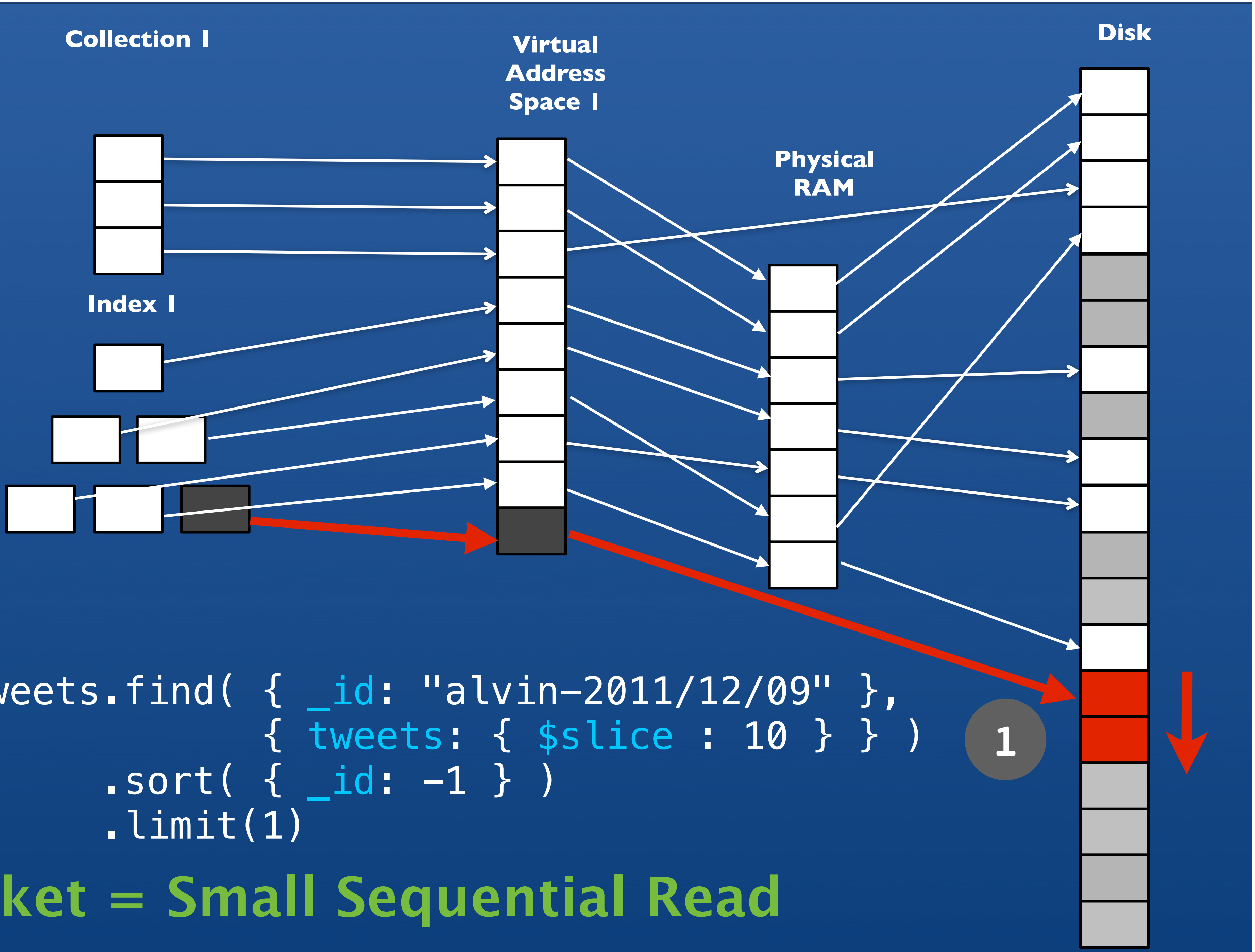
```
// tweets : one doc per user per day
> db.tweets.findOne()

   {
      _id:    "alvin-2011/12/09",
      email:  "alvin@10gen.com",
      tweets: [
        { user:  "Bob",
          tweet: "20111209-1231",
          text:  "Best Tweet Ever!" } ,
        { author: "Joe",
          date:   "May 27 2011",
          text:   "Stuck in traffic (again)" }
      ]
    }
```

Solution C
Last 10 Tweets

```
// Get the latest bucket, slice the last 10 tweets

db.tweets.find( { _id: "alvin-2011/12/09" },
                { tweets: { $slice : 10 } }  )
         .sort( { _id: -1 } )
         .limit(1)
```



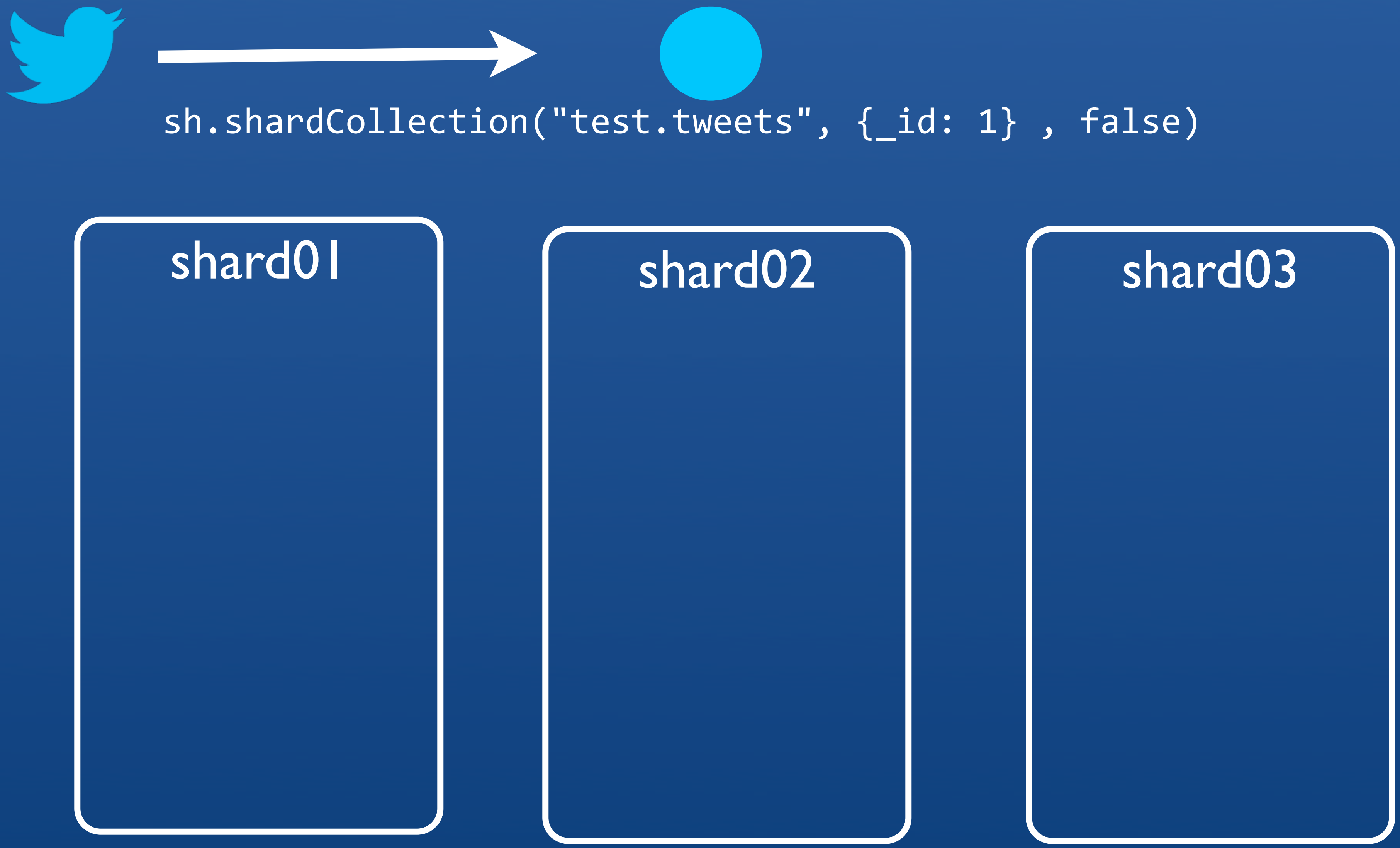
```
db.tweets.find( { _id: "alvin-2011/12/09" },
                { tweets: { $slice : 10 } } )
         .sort( { _id: -1 } )
         .limit(1)
```

Bucket = Small Sequential Read

Sharding - Goals

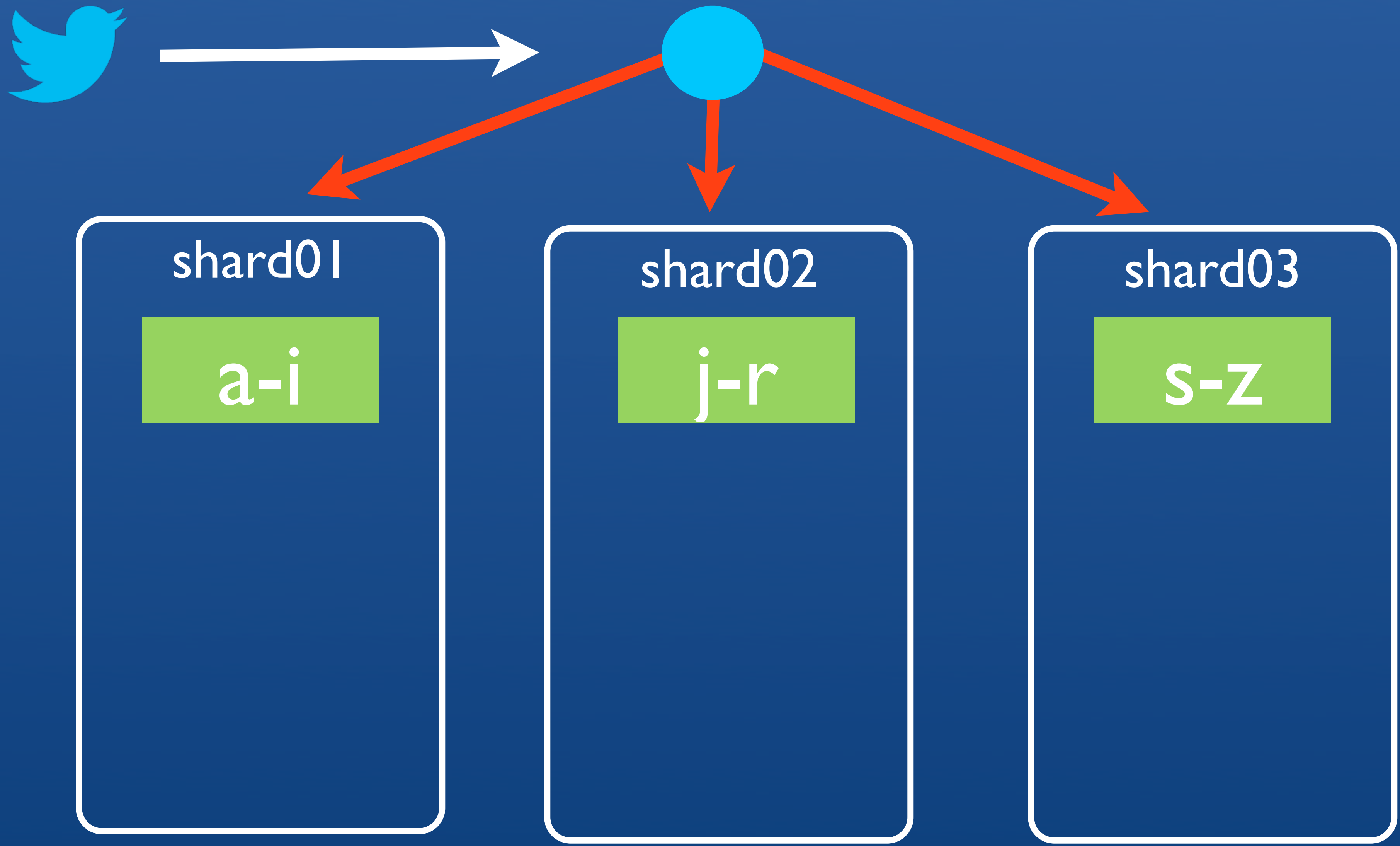
- Data location transparent to your code
- Data distribution is automatic
- Data re-distribution is automatic
- Aggregate system resources horizontally
- No code changes

Sharding - Range distribution



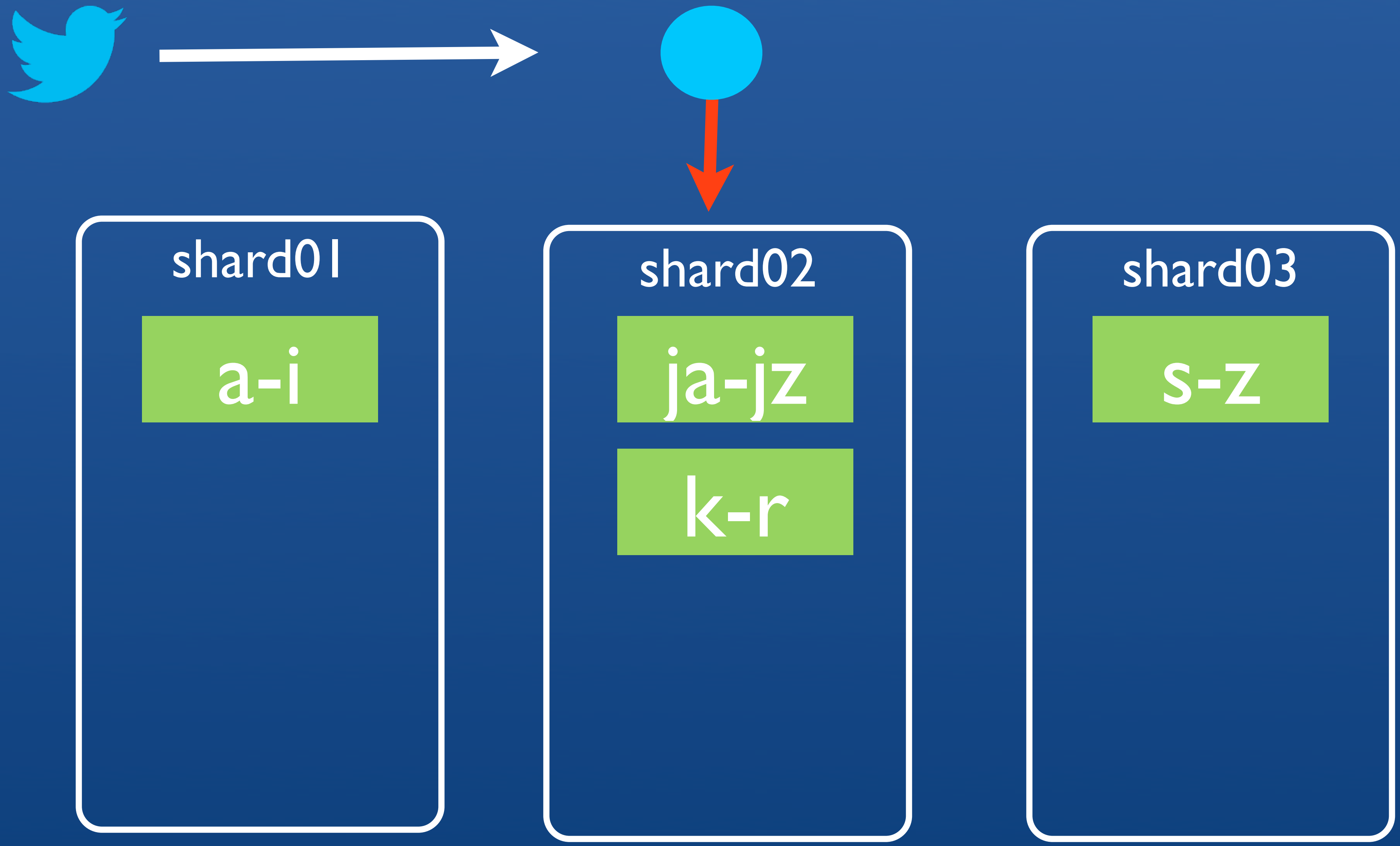
```
sh.shardCollection("test.tweets", {_id: 1} , false)
```

Sharding - Range distribution



10gen | the MongoDB company

Sharding - Splits



shard01

a-i

shard02

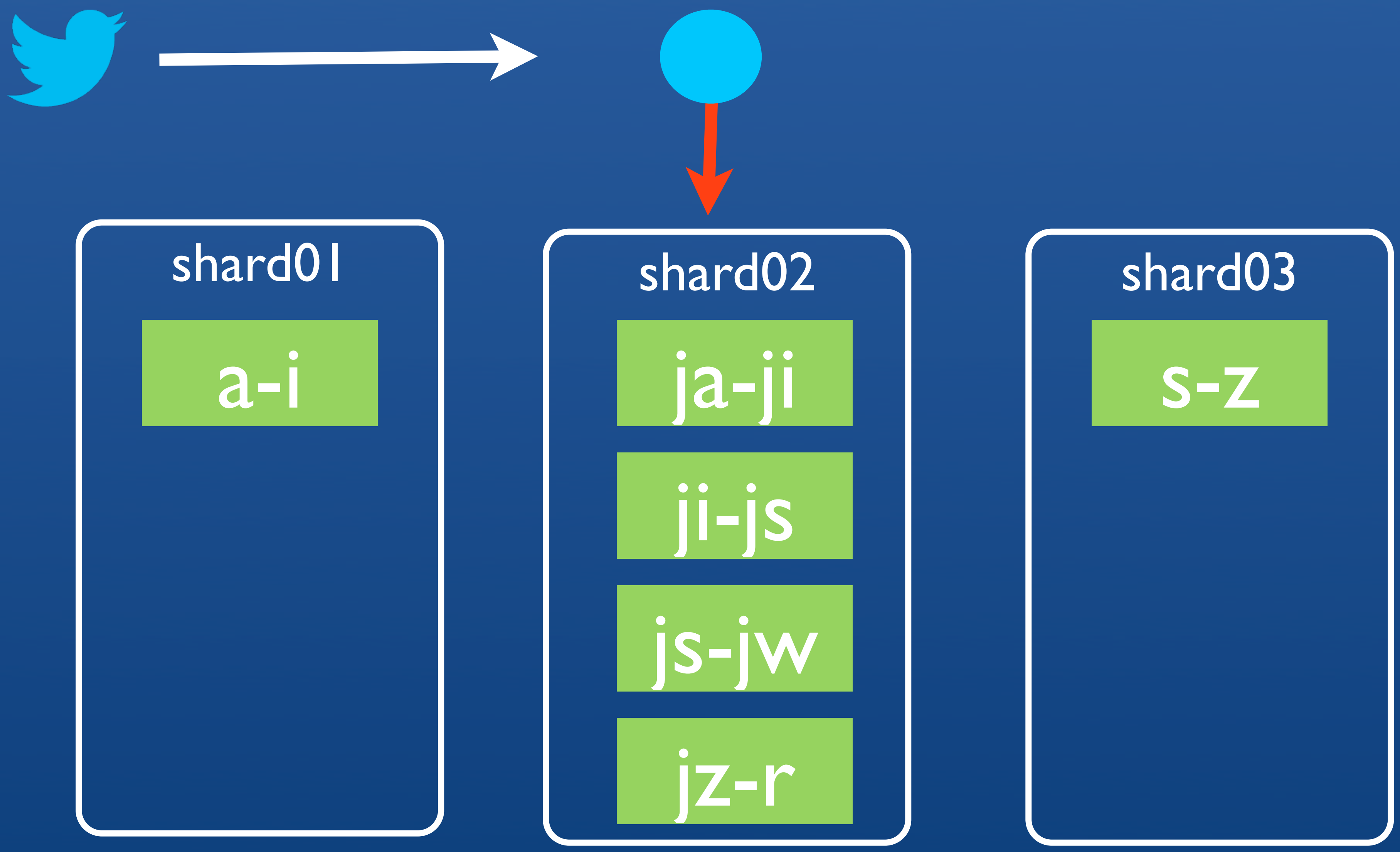
ja-jz

k-r

shard03

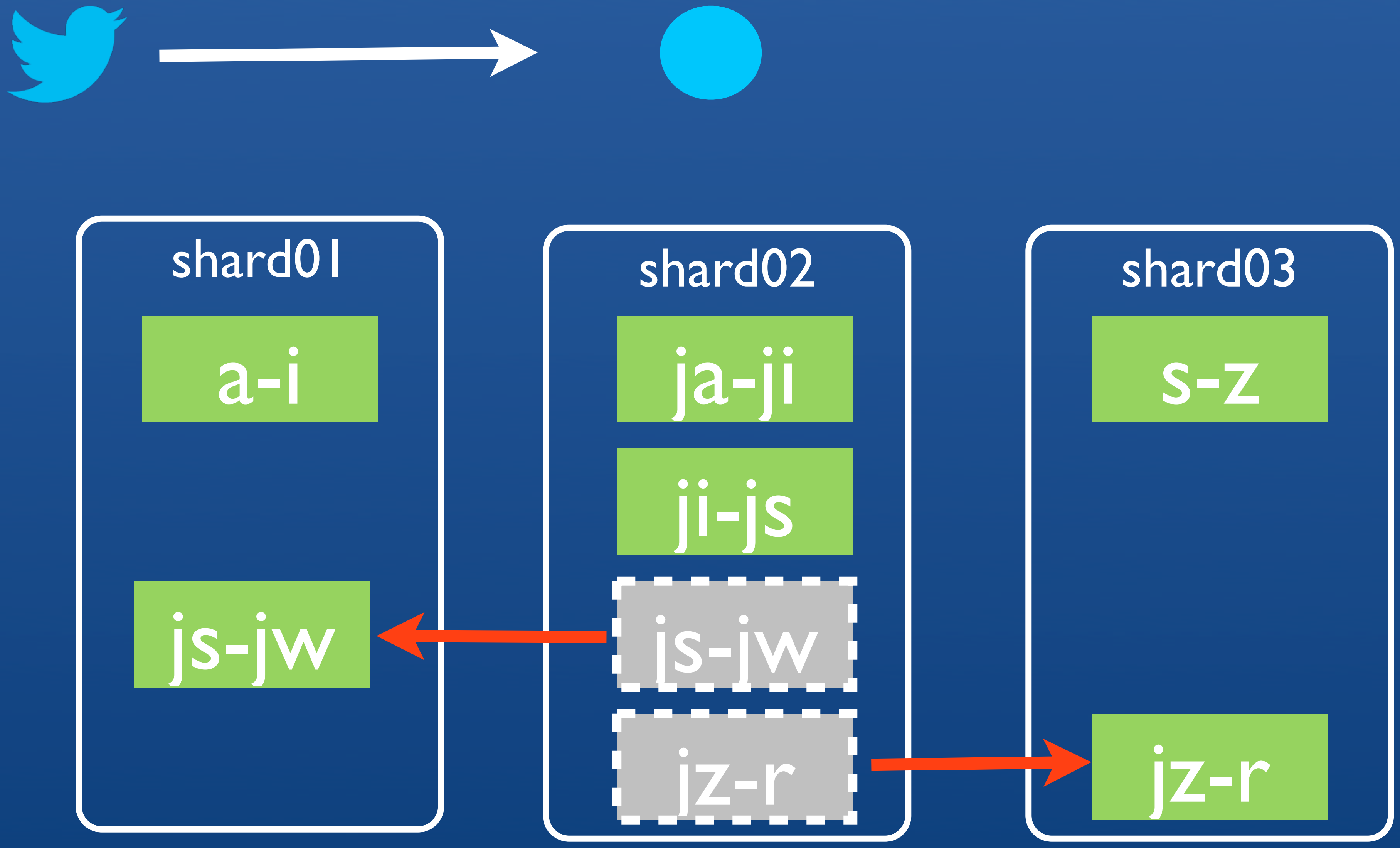
s-z

Sharding - Splits

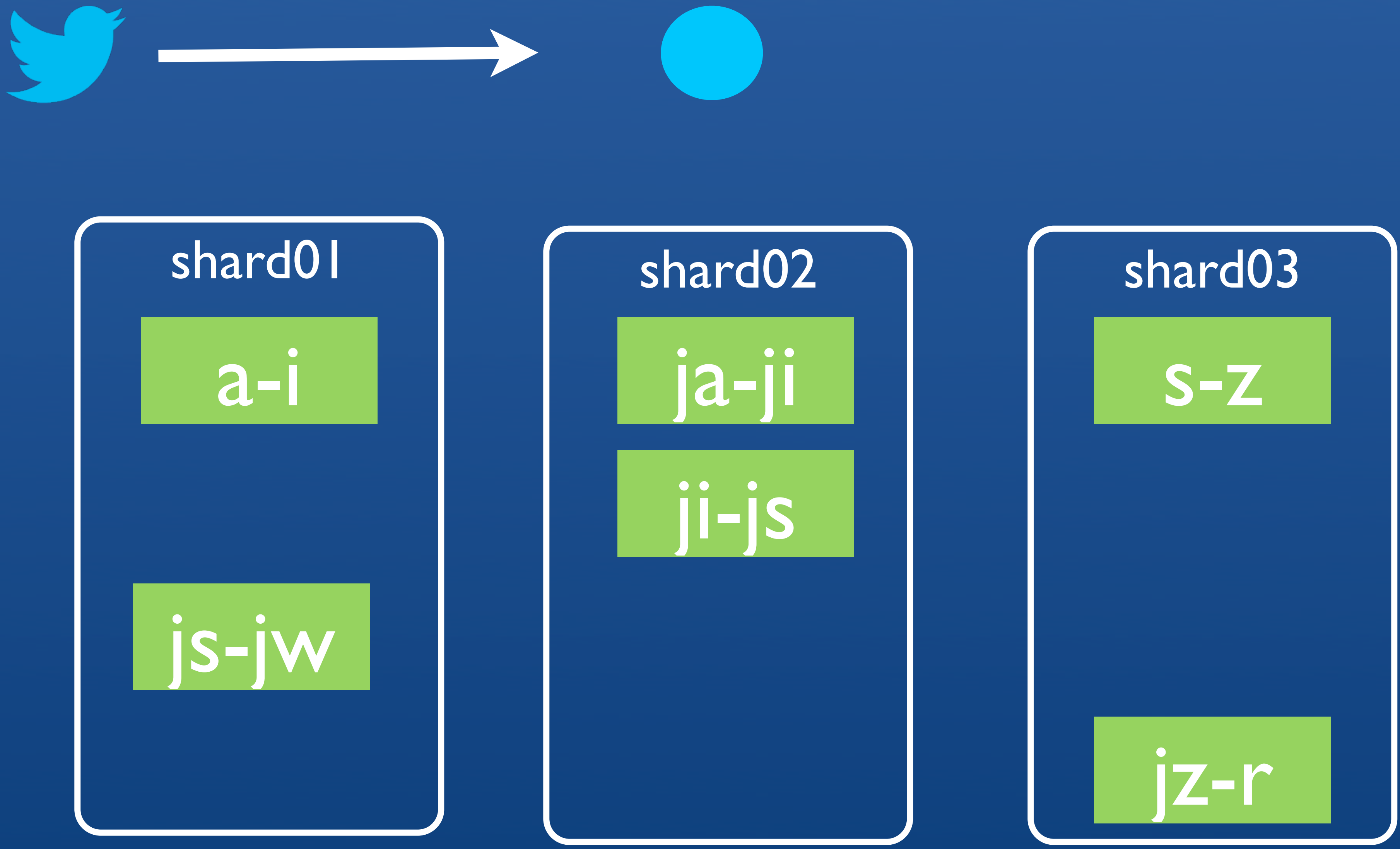


10gen | the MongoDB company

Sharding - Auto Balancing



Sharding - Auto Balancing



How does sharding effect Schema Design?

- Sharding key choice
- Access patterns (query versus write)

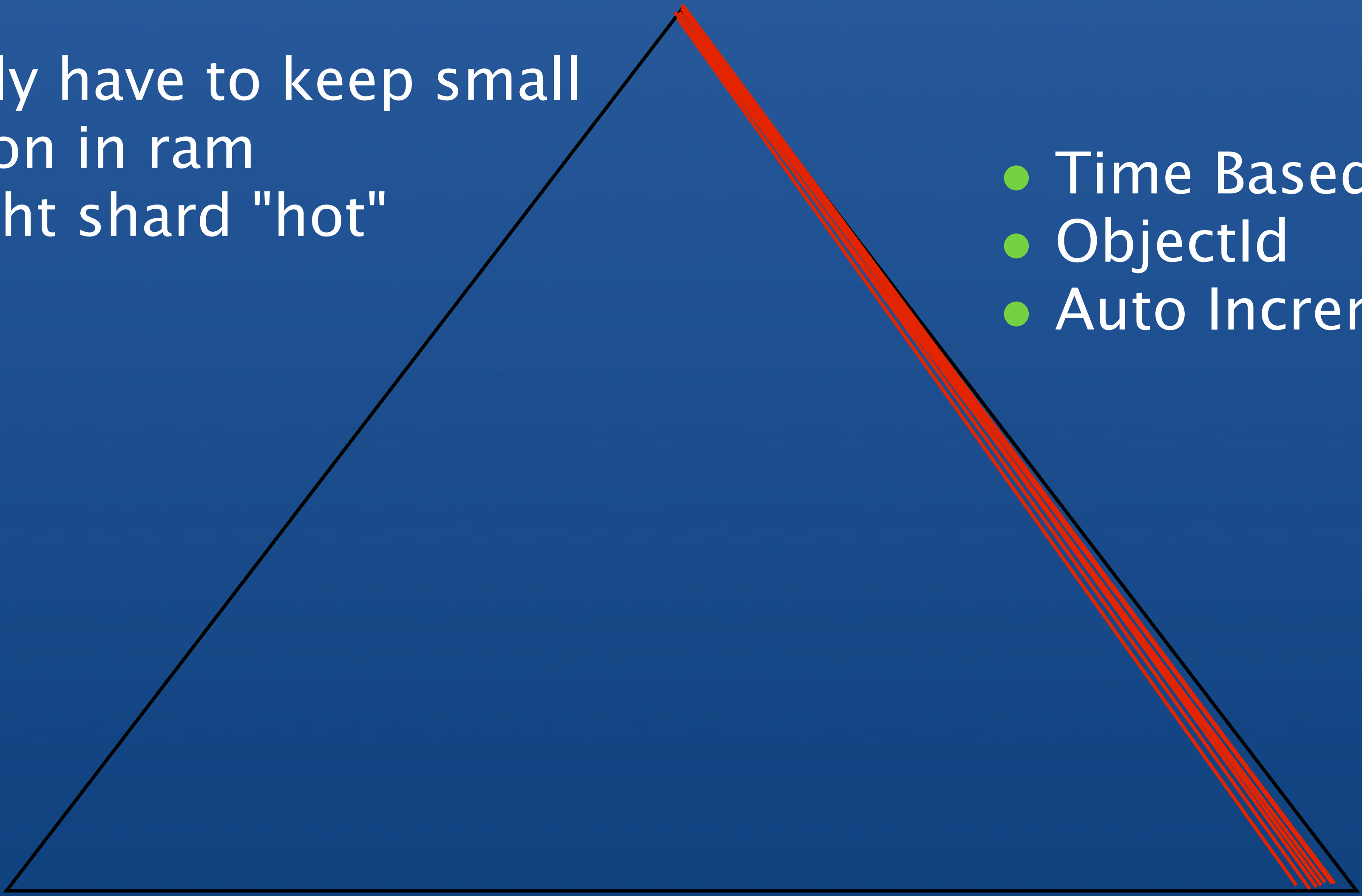
Sharding Key

```
{ photo_id :  ???? , data : <binary> }
```

- What’s the right key?
  - auto increment
  - MD5(data)
  - month() + MD5(data)

Right balanced access

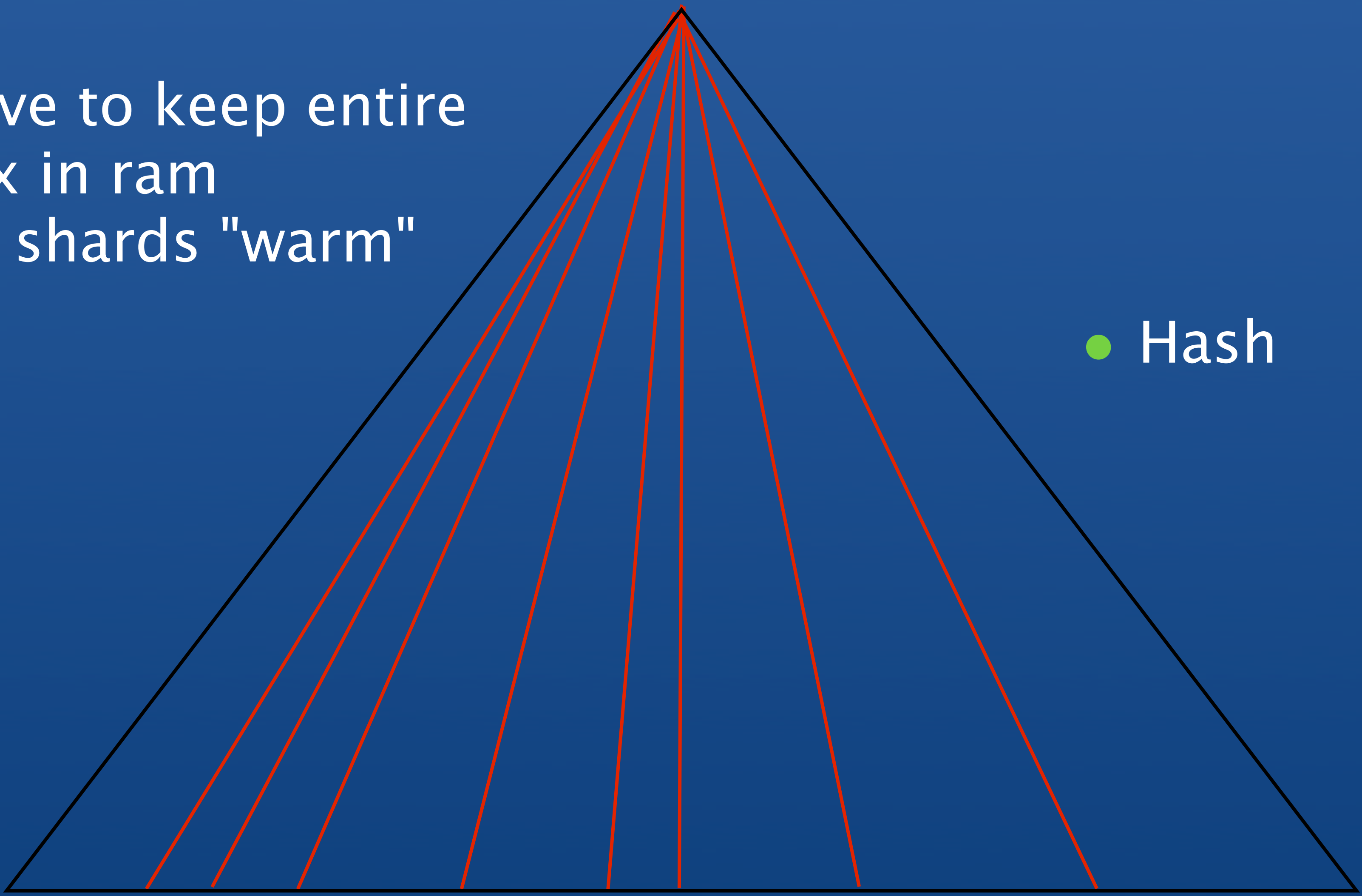
- Only have to keep small portion in ram
- Right shard "hot"



- Time Based
- ObjectId
- Auto Increment

Random access

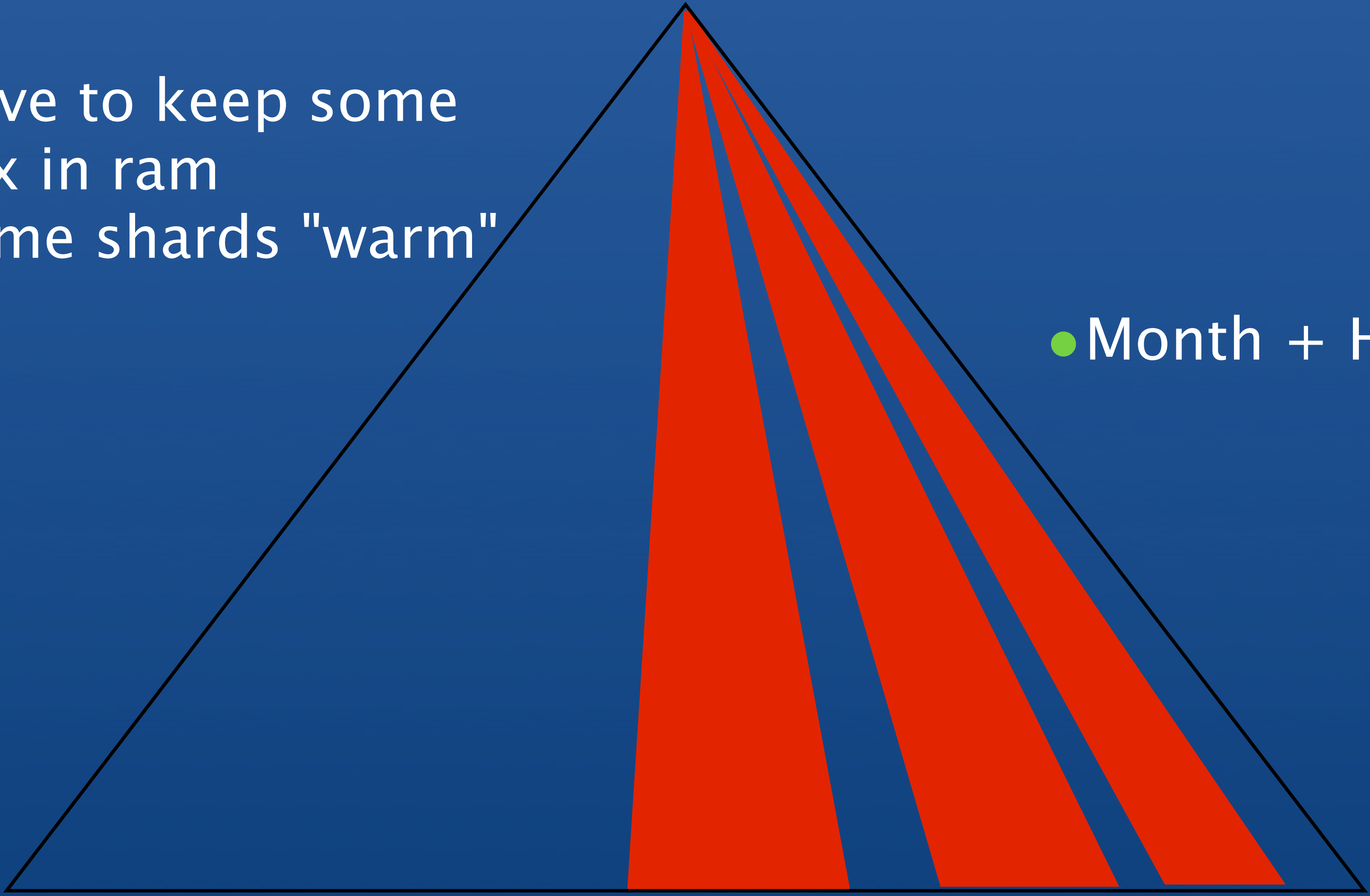
- Have to keep entire index in ram
- All shards "warm"



- Hash

Segmented access

- Have to keep some index in ram
- Some shards "warm"



- Month + Hash

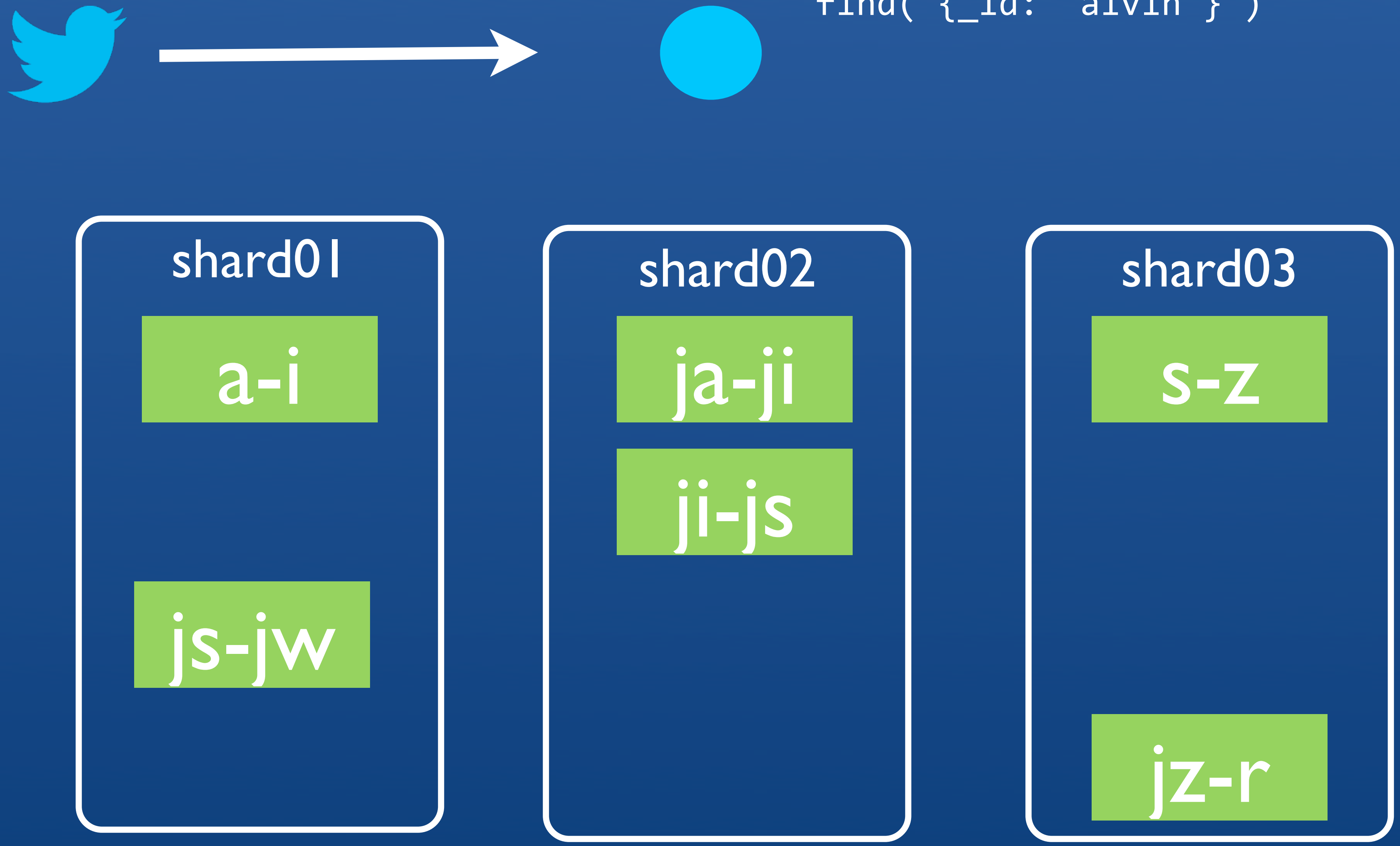
Solution A
Shard by a single identifier

```
{ _id :    "alvin",                // shard key
  email:   "alvin@10gen.com",
  display: "jonnyeight"
  li:      "alvin.j.richards",
  tweets:  [ ... ]
}
```

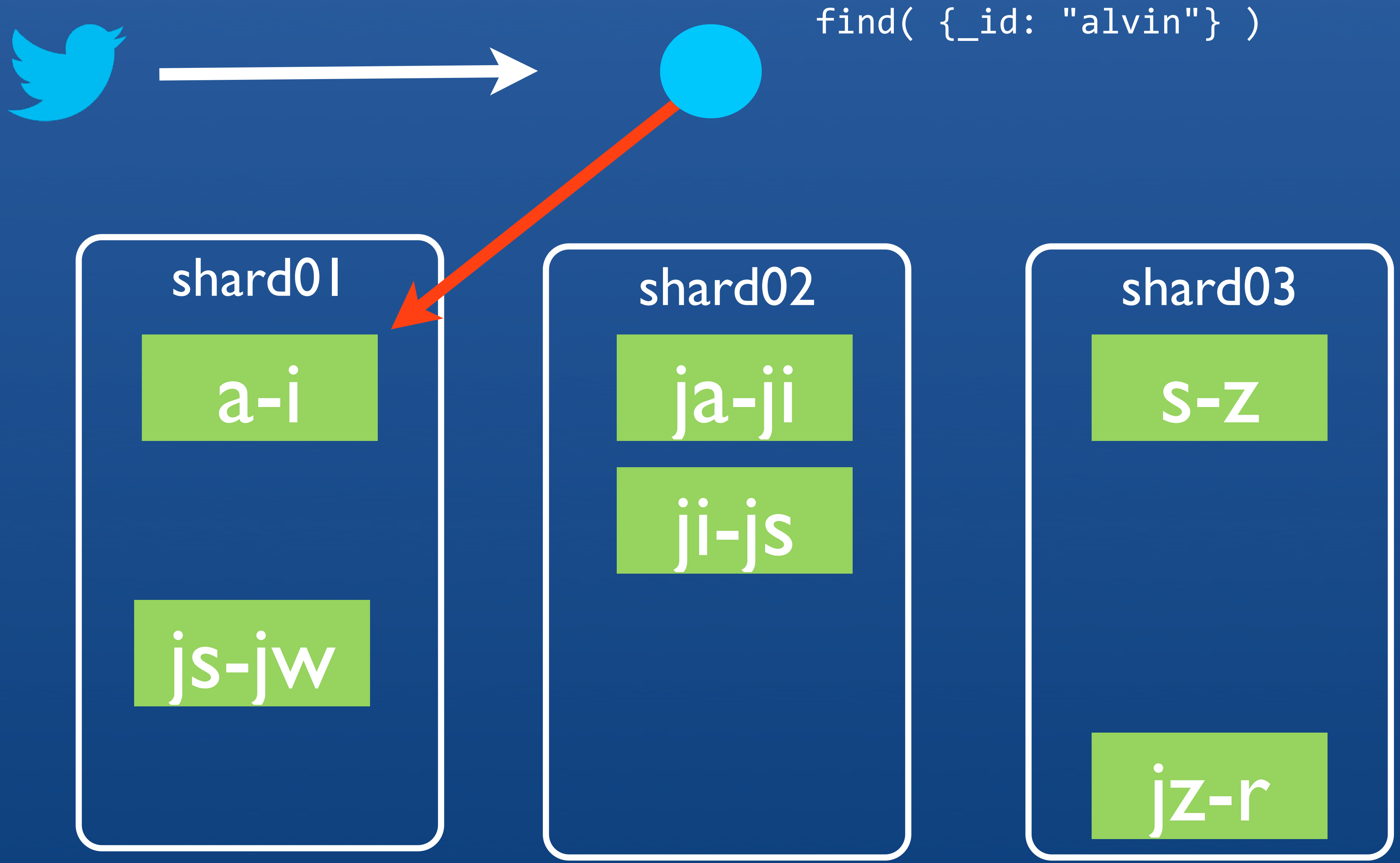
```
 Shard on { _id : 1 }
 Lookup by _id routed to 1 node
 Index on { “email” : 1 }
```

Sharding - Routed Query

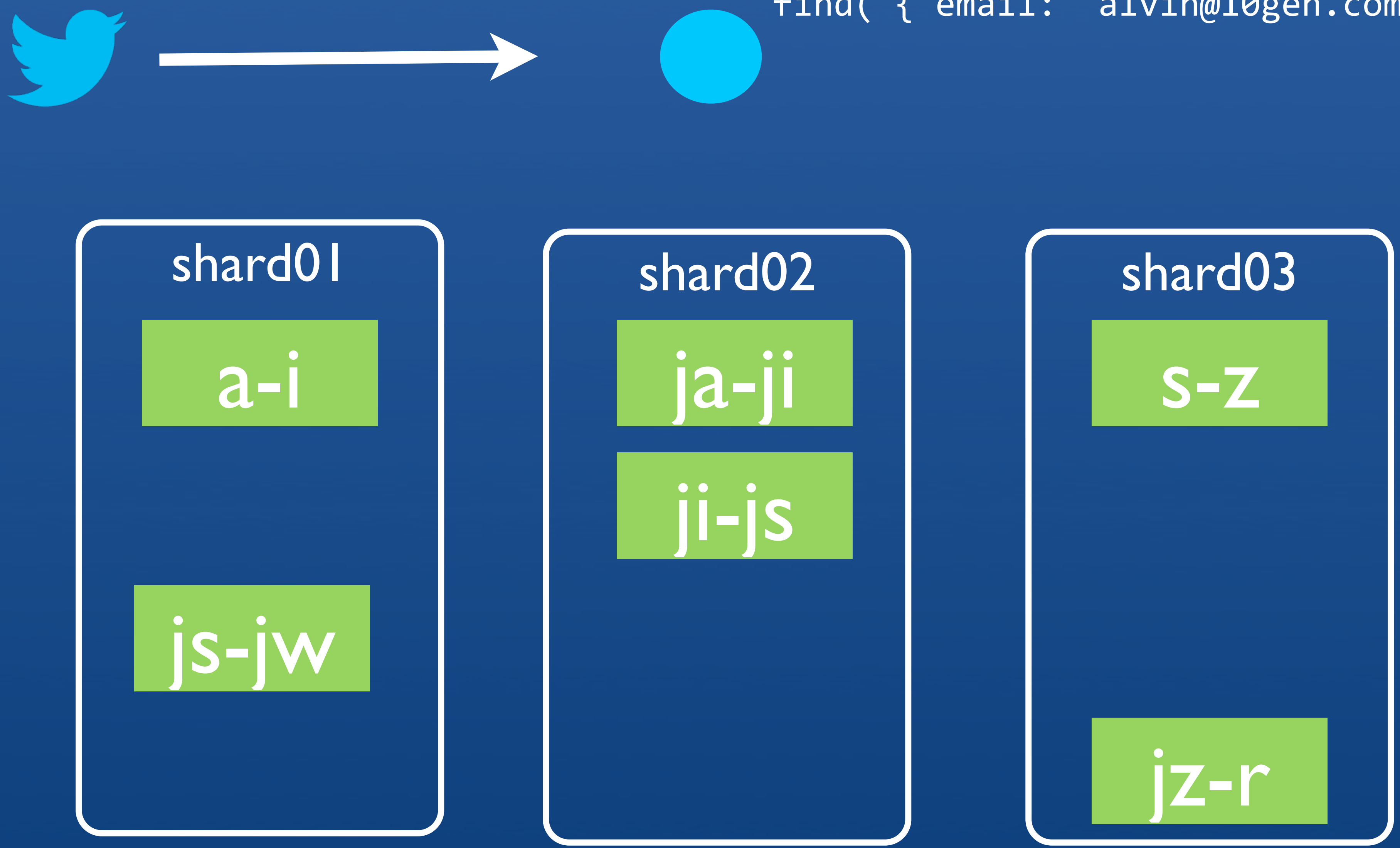
```
find( {_id: "alvin"} )
```



Sharding - Routed Query

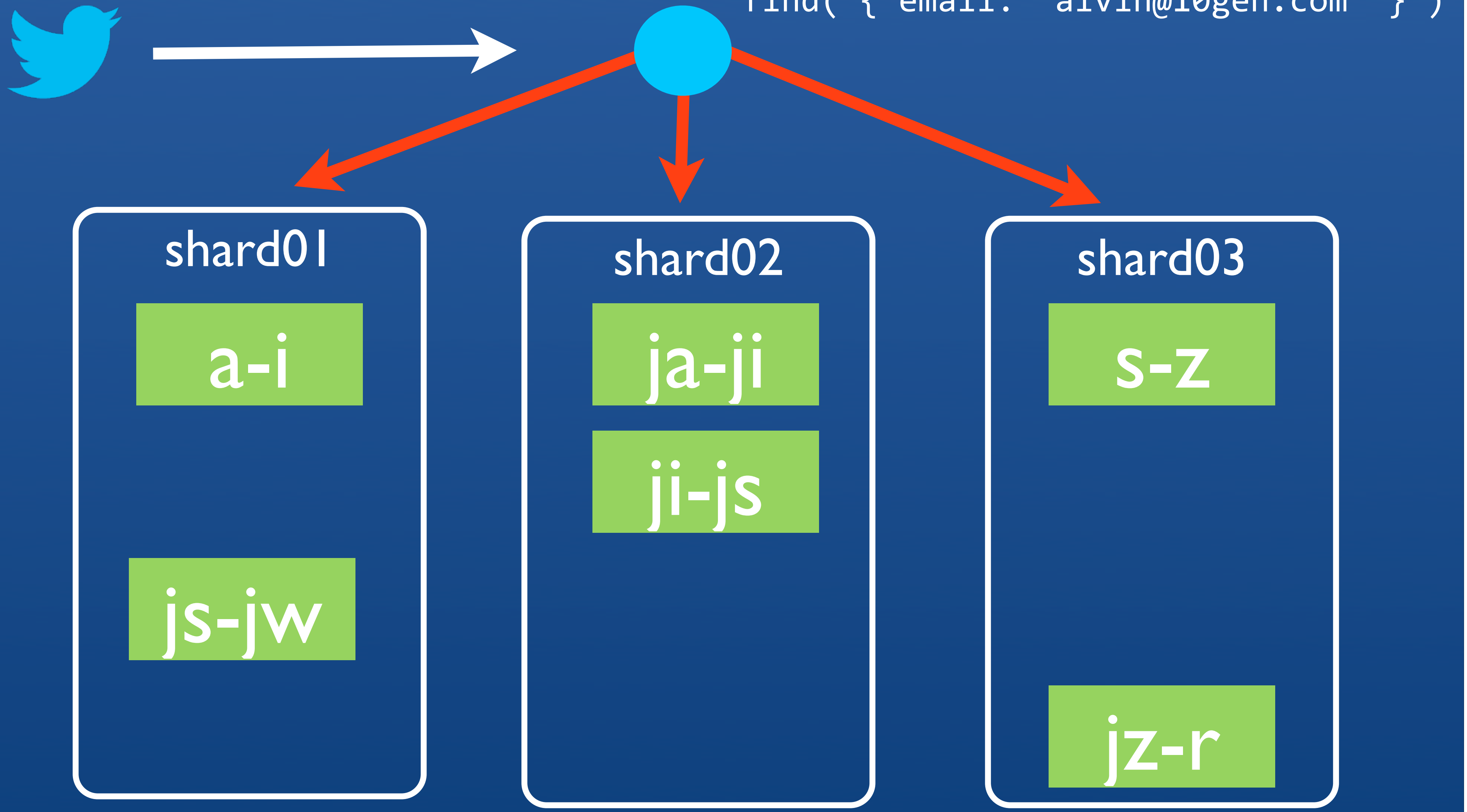


Sharding - Scatter Gather



```
find( { email: "alvin@10gen.com" } )
```

Sharding - Scatter Gather



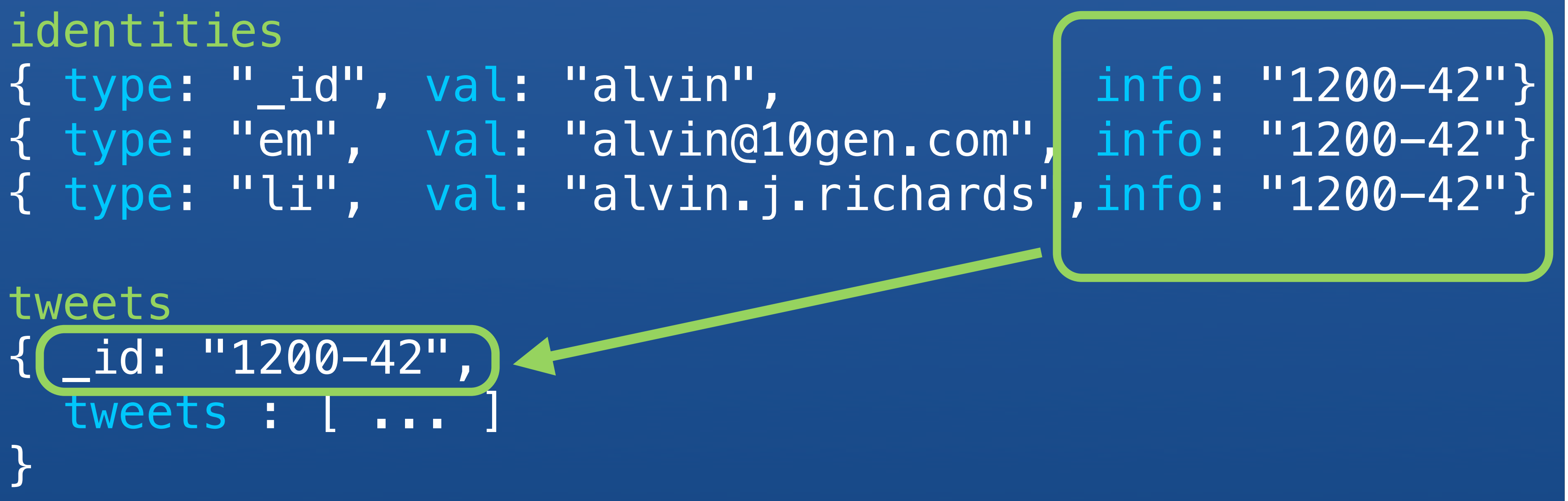
```
find( { email: "alvin@10gen.com" } )
```

Multiple Identities

- User can have multiple identities
  - twitter name
  - email address
  - etc.

- What is the best sharding key & schema design?

Solution B
Shard by multiple identifiers

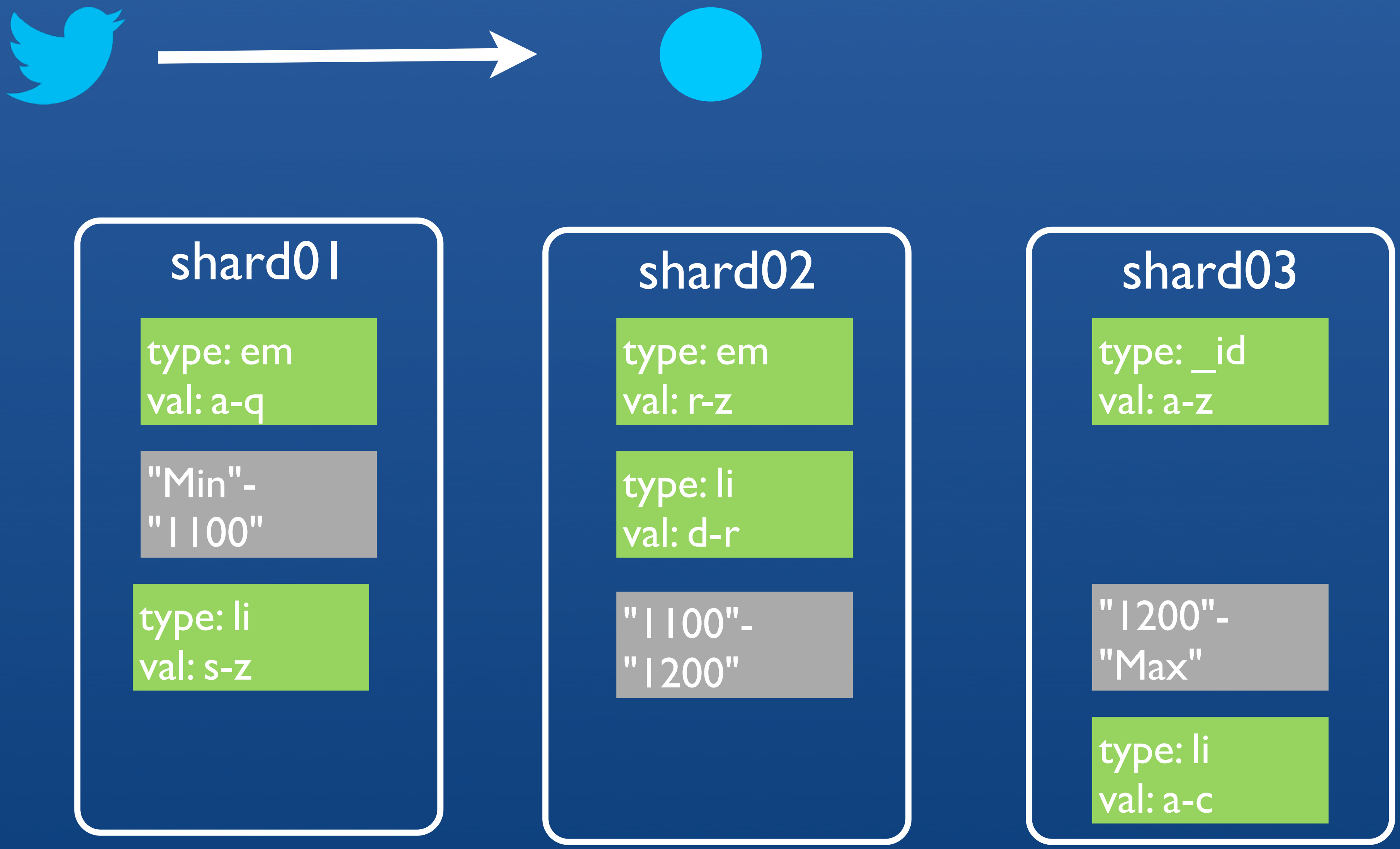


```
identities
{ type: "_id", val: "alvin",           info: "1200-42"}
{ type: "em",  val: "alvin@10gen.com", info: "1200-42"}
{ type: "li",  val: "alvin.j.richards",info: "1200-42"}

tweets
{ _id: "1200-42",
  tweets : [ ... ]
}
```

- Shard identities on { type : 1, val : 1 }
- Lookup by type & val routed to 1 node
- Can create unique index on type & val
- Shard info on { _id: 1 }
- Lookup info on _id routed to 1 node

Sharding - Routed Query



shard01

type: em
val: a-q

"Min"-
"1100"

type: li
val: s-z

shard02

type: em
val: r-z

type: li
val: d-r

"1100"-
"1200"

shard03

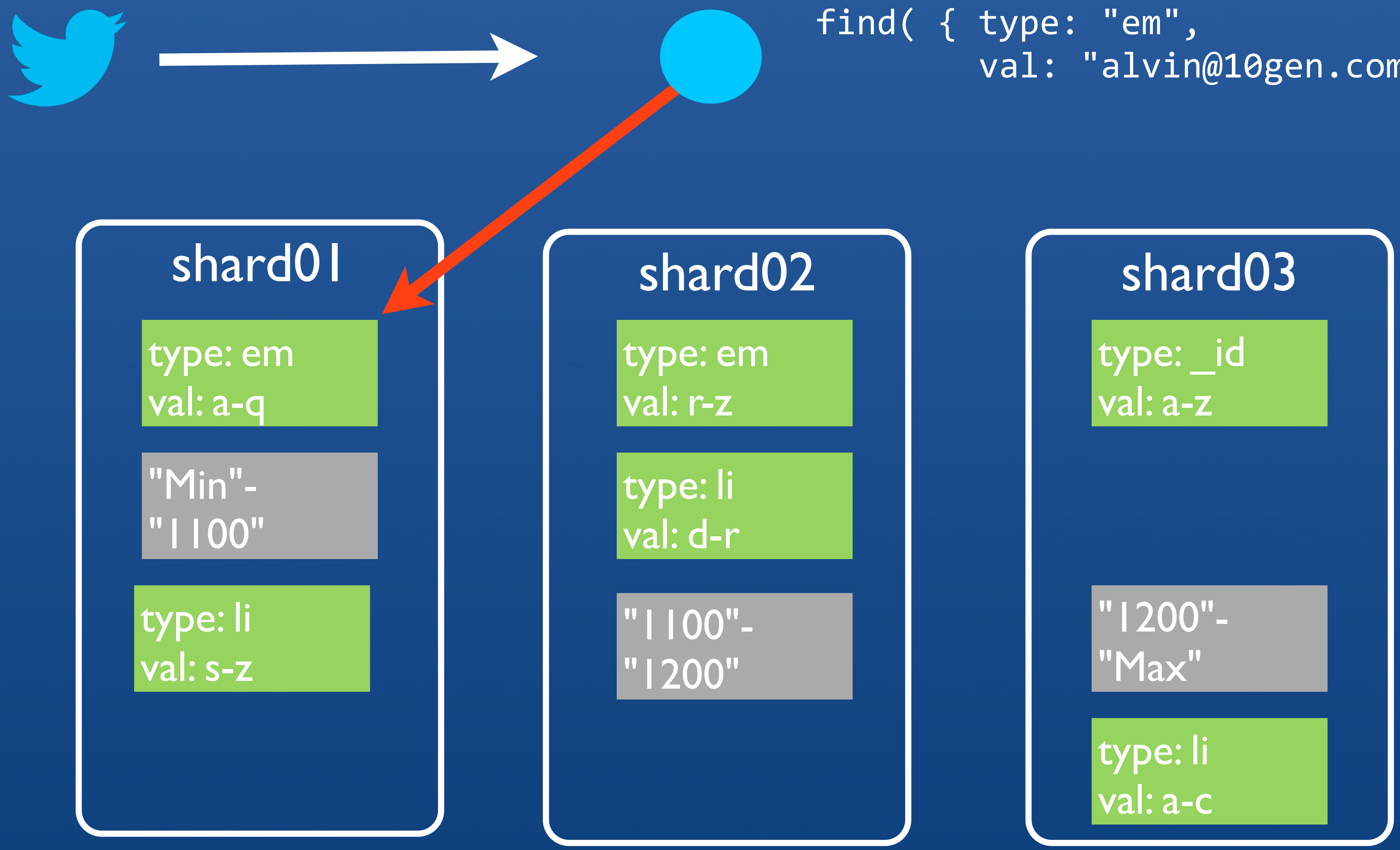
type: _id
val: a-z

"1200"-
"Max"

type: li
val: a-c

Sharding - Routed Query

```
find( { type: "em",
        val: "alvin@10gen.com } )
```



shard01

type: em
val: a-q

"Min"-
"1100"

type: li
val: s-z

shard02

type: em
val: r-z

type: li
val: d-r

"1100"-
"1200"

shard03

type: _id
val: a-z

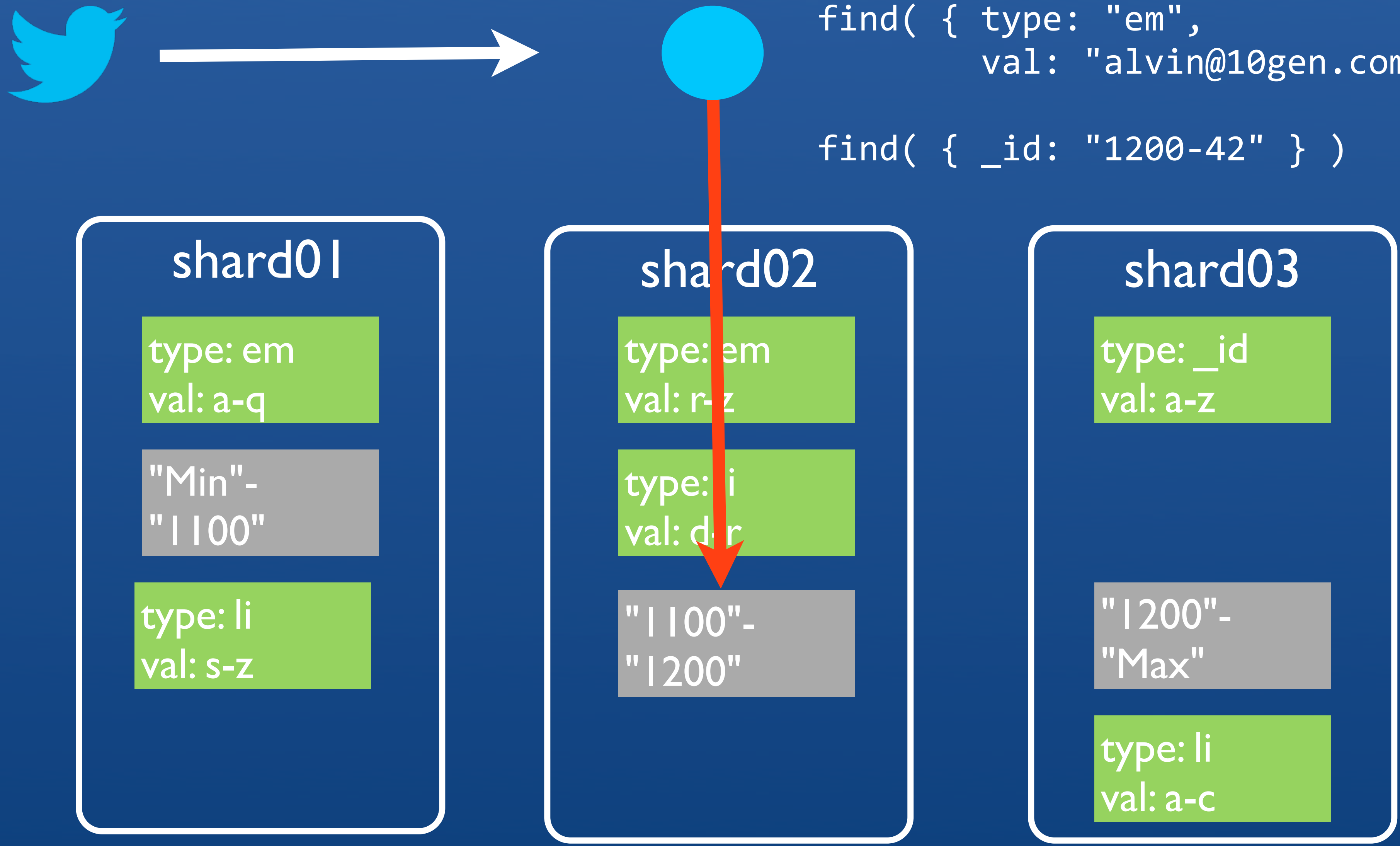
"1200"-
"Max"

type: li
val: a-c

Sharding - Routed Query

```
find( { type: "em",
        val: "alvin@10gen.com } )

find( { _id: "1200-42" } )
```



shard01

type: em
val: a-q

"Min"-
"1100"

type: li
val: s-z

shard02

type: em
val: r-z

type: li
val: d-r

"1100"-
"1200"

shard03

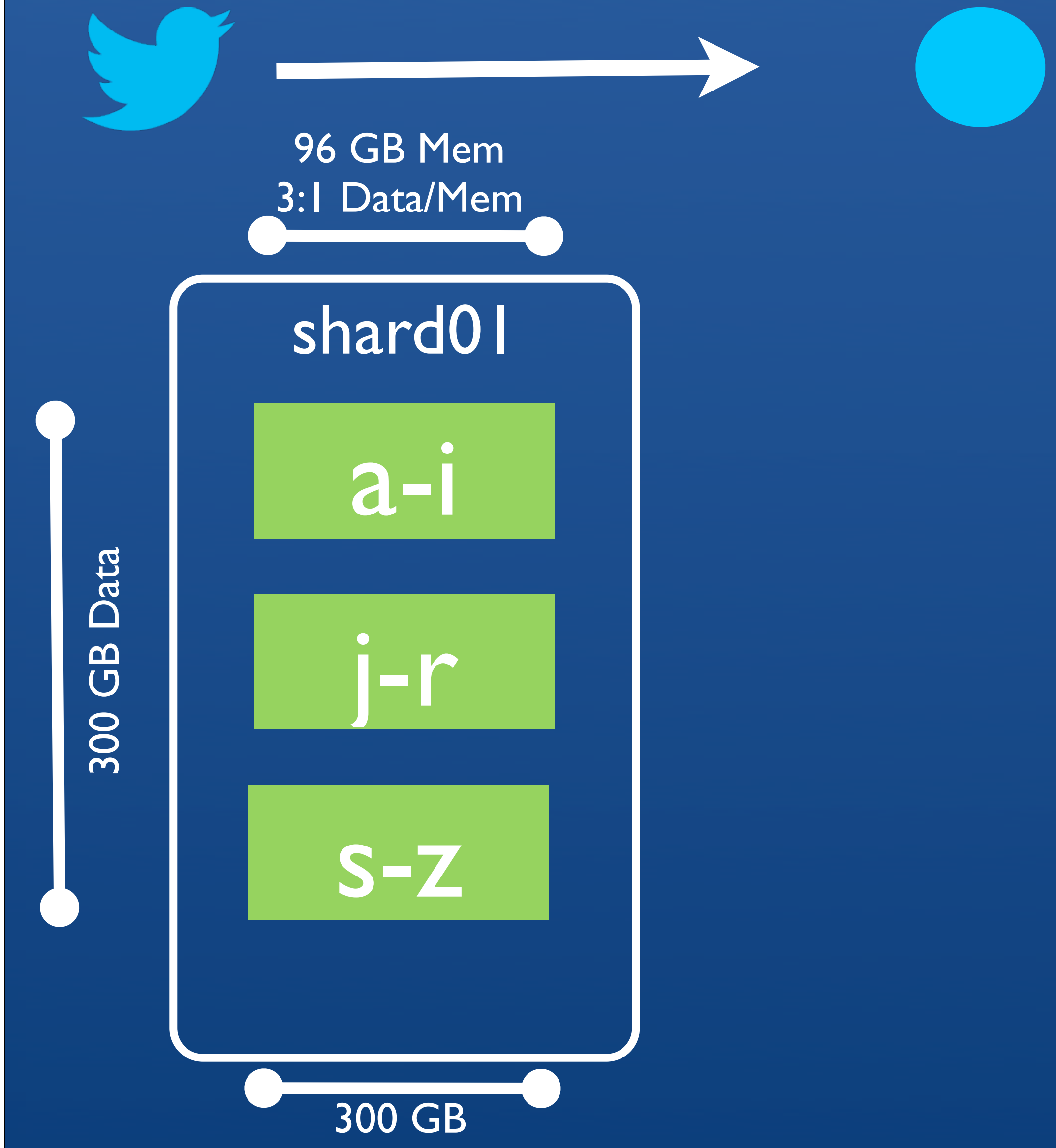
type: _id
val: a-z

"1200"-
"Max"

type: li
val: a-c

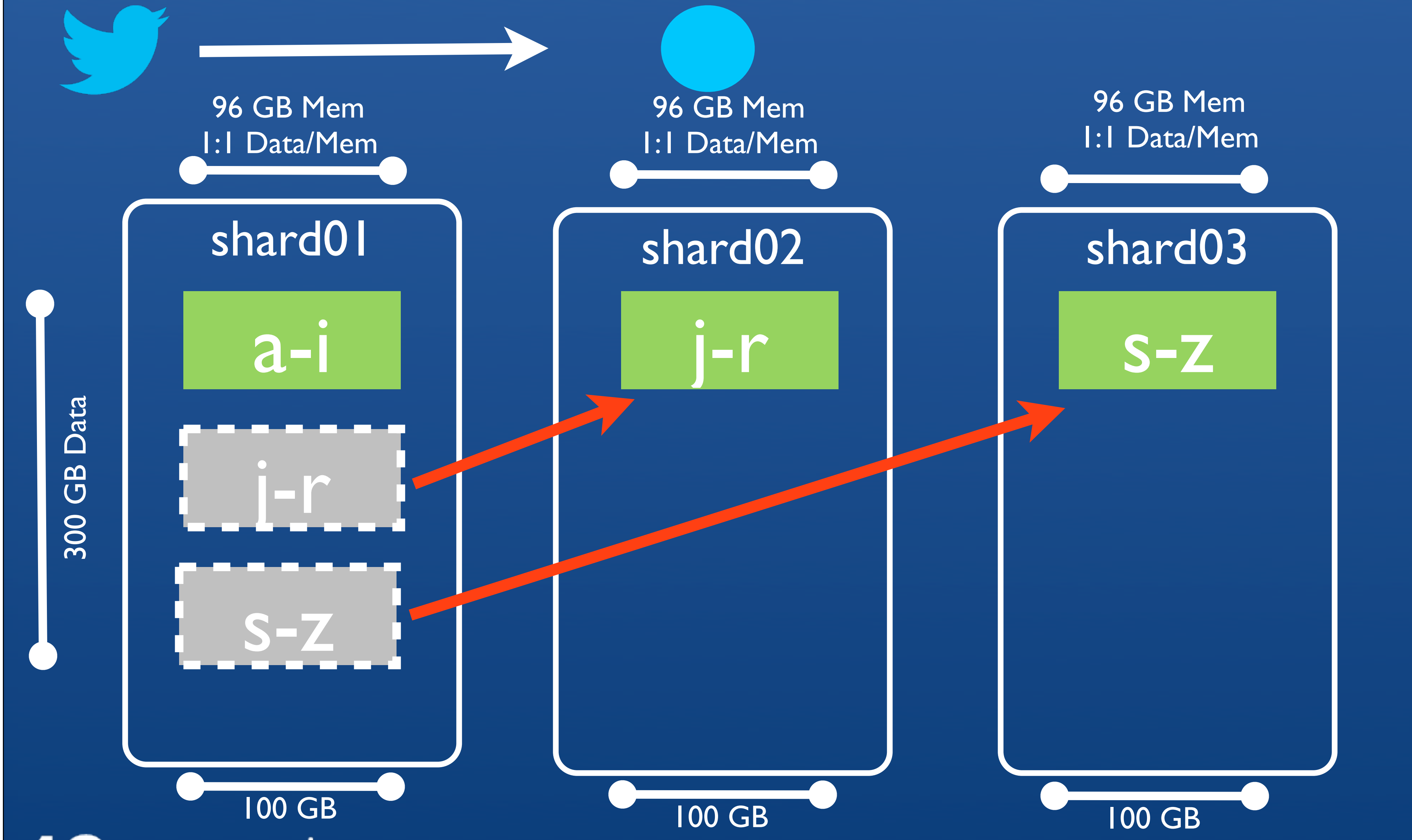
10gen | the MongoDB company

Sharding - Caching



10gen | the MongoDB company

Aggregate Horizontal Resources



10gen | the MongoDB company

Auto Sharding - Summary

- Fully consistent
- Application code unaware of data location
- Zero code changes
- Shard by Compound Key, Tag, Hash (2.4)
- Add capacity
  - On-line
  - When needed
  - Zero downtime

Time Series Data

- Records votes by
  - Day, Hour, Minute
- Show time series of votes cast

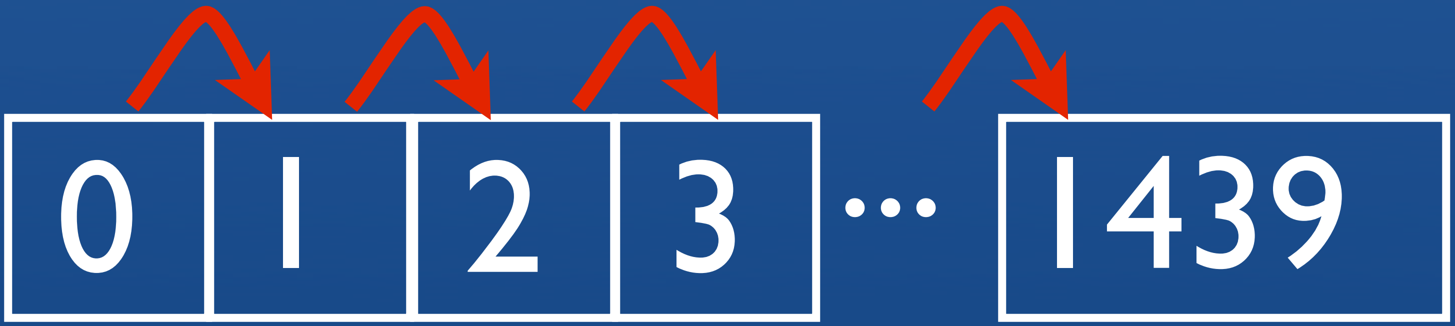
Solution A
Time Series

```
// Time series buckets, hour and minute sub-docs
{ _id: "20111209-1231",
  ts:  ISODate("2011-12-09T00:00:00.000Z")
  daily: 67,
  hourly: { 0: 3, 1: 14, 2: 19 ... 23: 72 },
  minute: { 0: 0, 1: 4,  2: 6 ...  1439: 0 }
}


// Add one to the last minute before midnight
> db.votes.update(
   { _id: "20111209-1231",
     ts:  ISODate("2011-12-09T00:00:00.037Z") },
   { $inc: { "hourly.23": 1 },
     $inc: { "minute.1439": 1 },
     $inc: { "daily": 1 } } )
```

BSON Storage

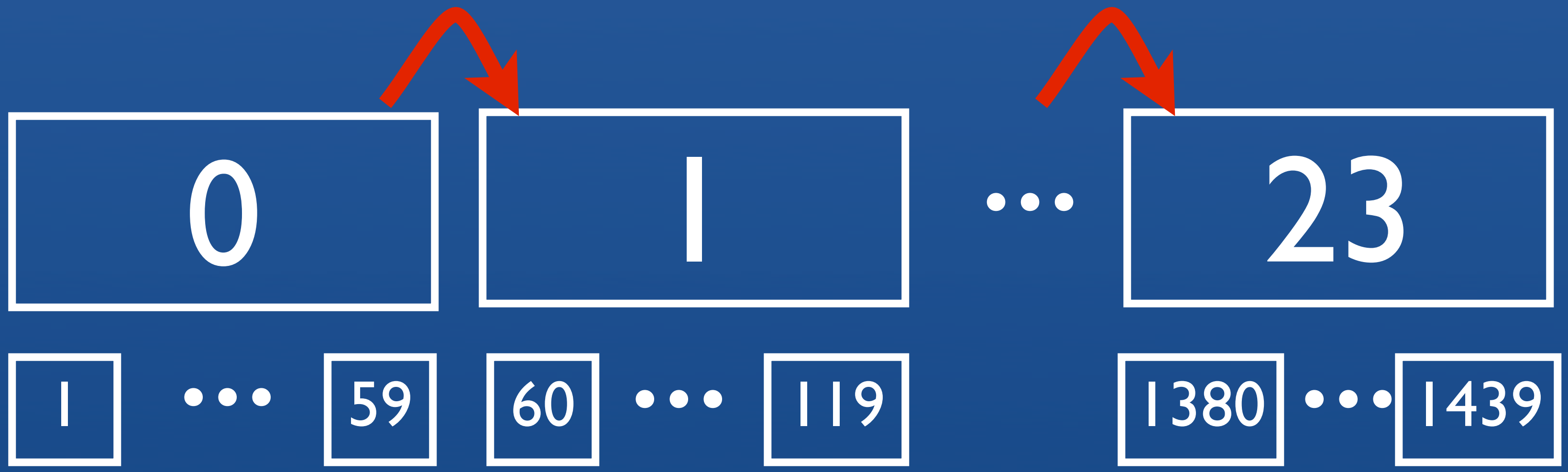
- Sequence of key/value pairs
- NOT a hash map
- Optimized to scan quickly



What is the cost of update the minute before midnight?

BSON Storage

- Can skip sub-documents



0 | 1 | ••• | 23

1 ••• 59 | 60 ••• 119 | 1380 ••• 1439

How could this change the schema?

Solution B
Time Series

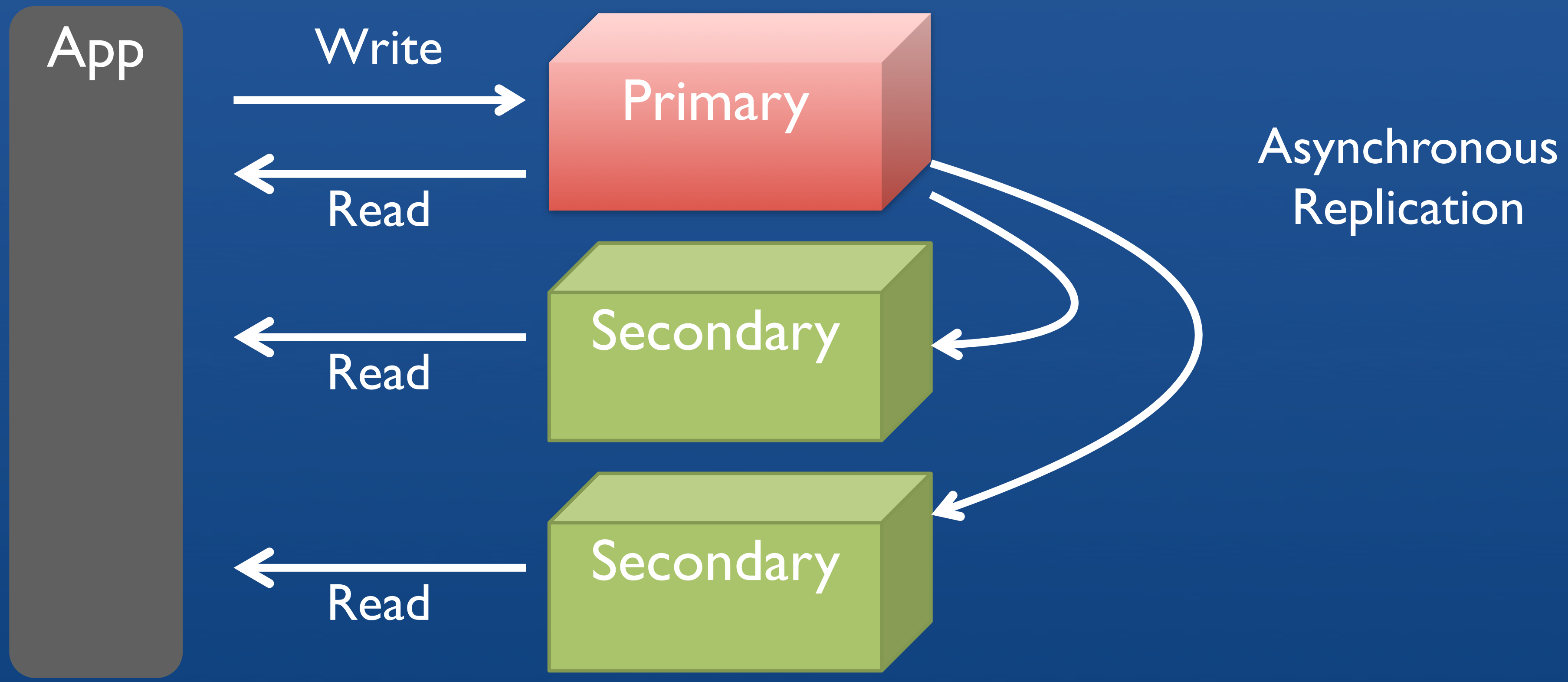
```
// Time series buckets, each hour a sub-document
{ _id: "20111209-1231",
  ts:  ISODate("2011-12-09T00:00:00.000Z")
  daily: 67,
  minute: { 0:  { 0: 0, 1: 7, ... 59: 2 },
            ...
            23: { 0: 15,   ... 59: 6 } }
}

// Add one to the last second before midnight
> db.votes.update(
   { _id: "20111209-1231" },
     ts:  ISODate("2011-12-09T00:00:00.000Z") },
   { $inc: { "minute.23.59": 1 },
     $inc: { daily: 1 } } )
```

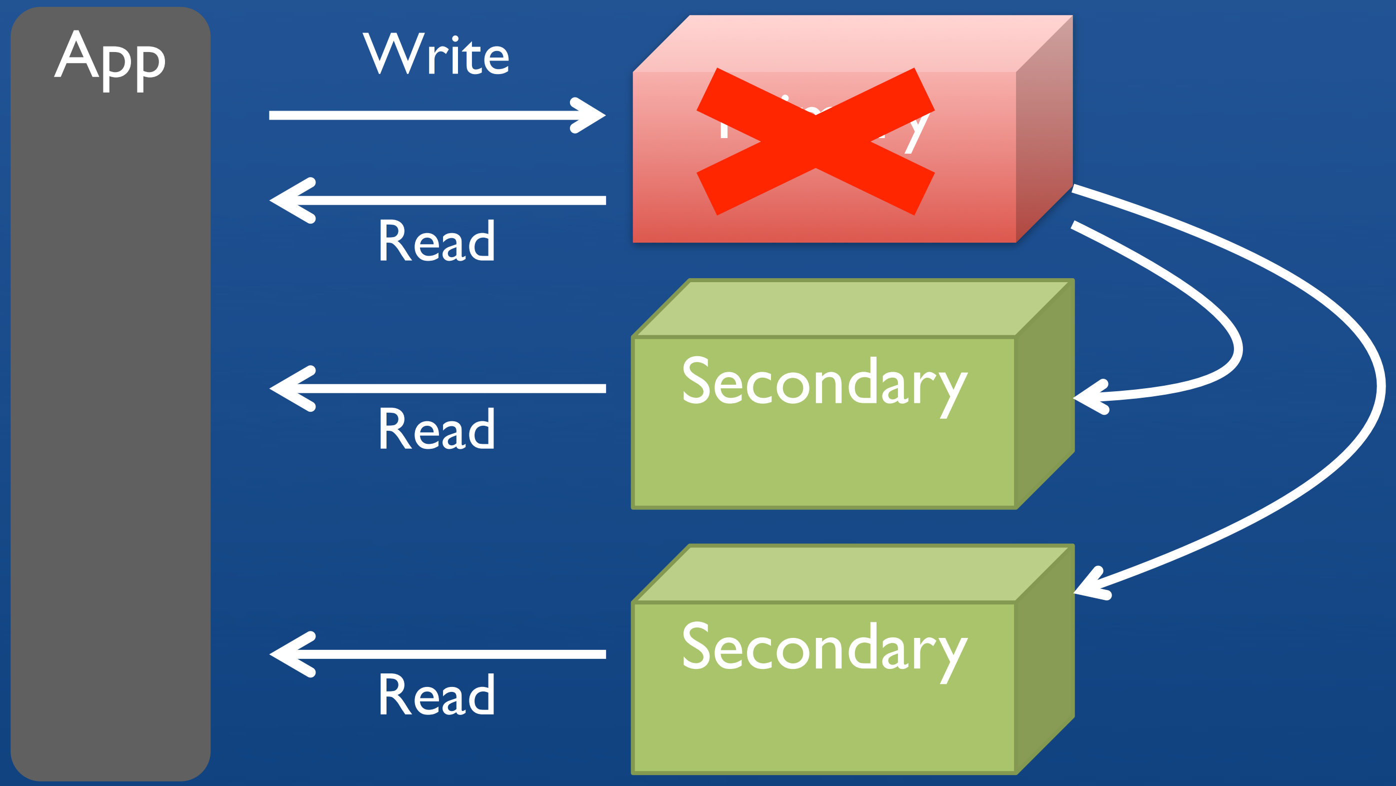
Replica Sets

- Data Protection
  - Multiple copies of the data
  - Spread across Data Centers, AZs
- High Availability
  - Automated Failover
  - Automated Recovery

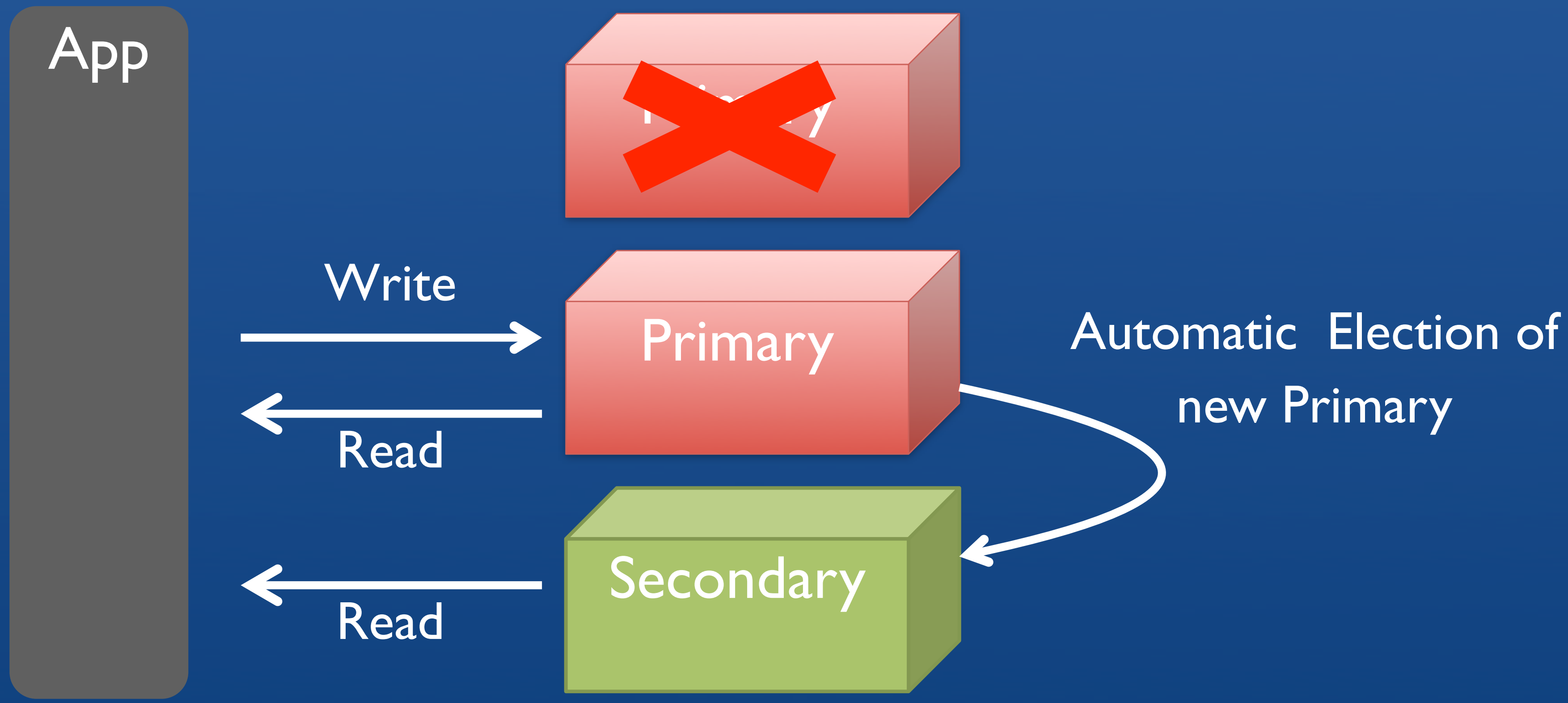
Replica Sets



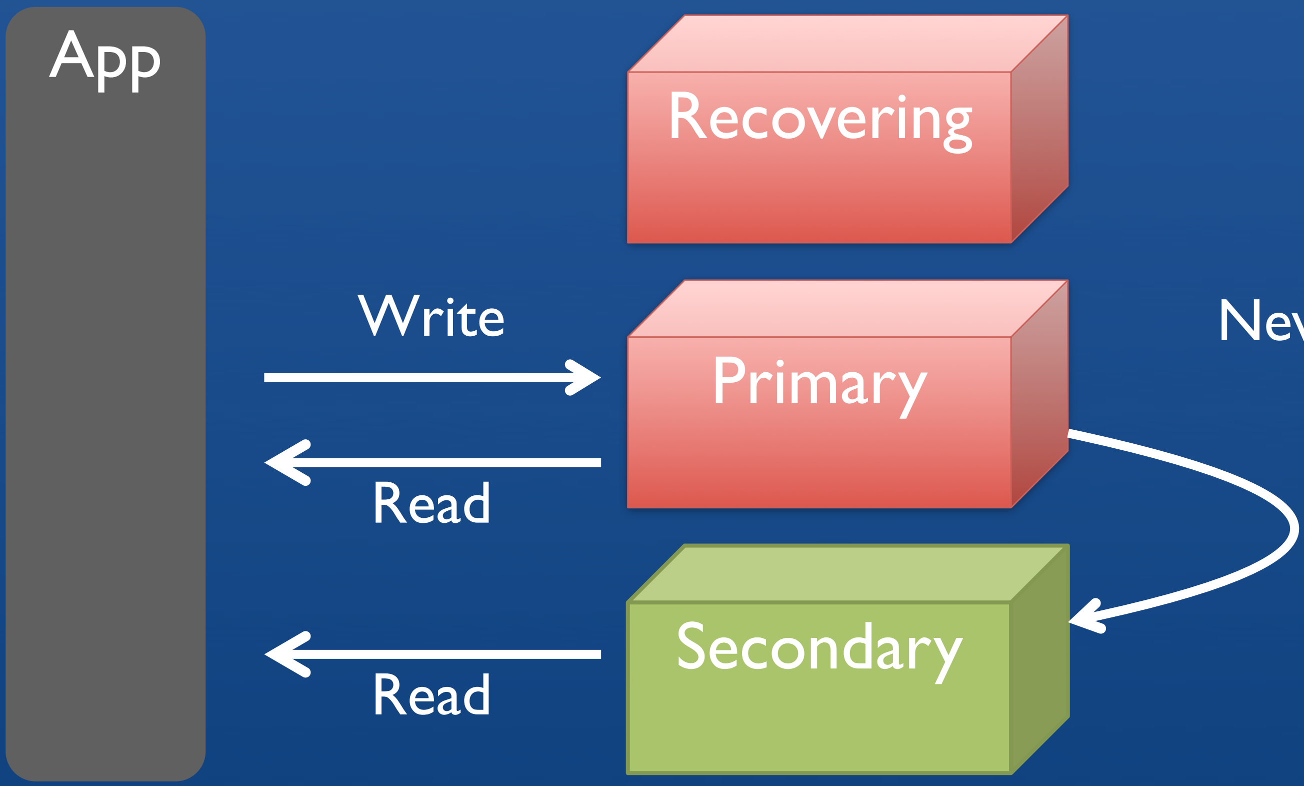
Replica Sets



Replica Sets

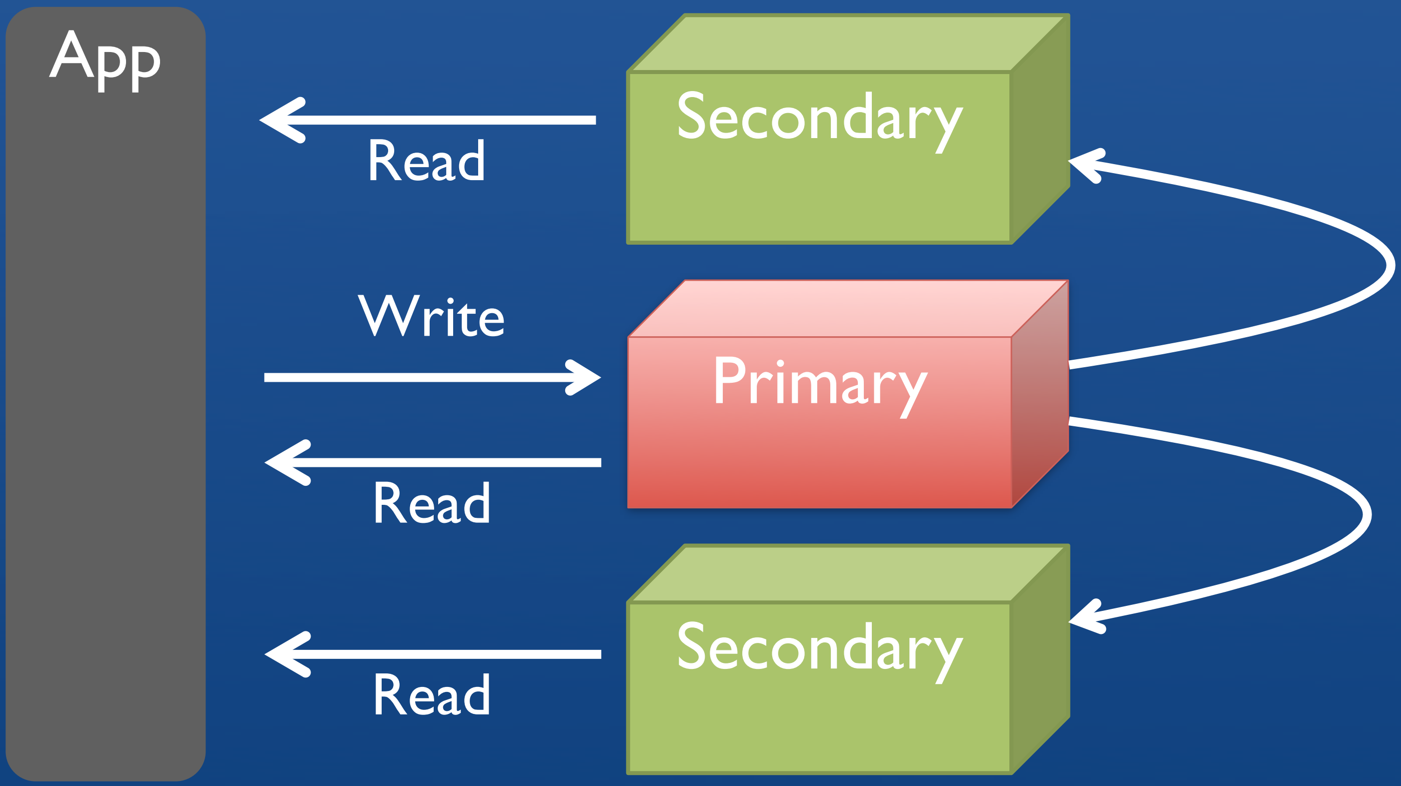


Replica Sets



New primary serves data

Replica Sets



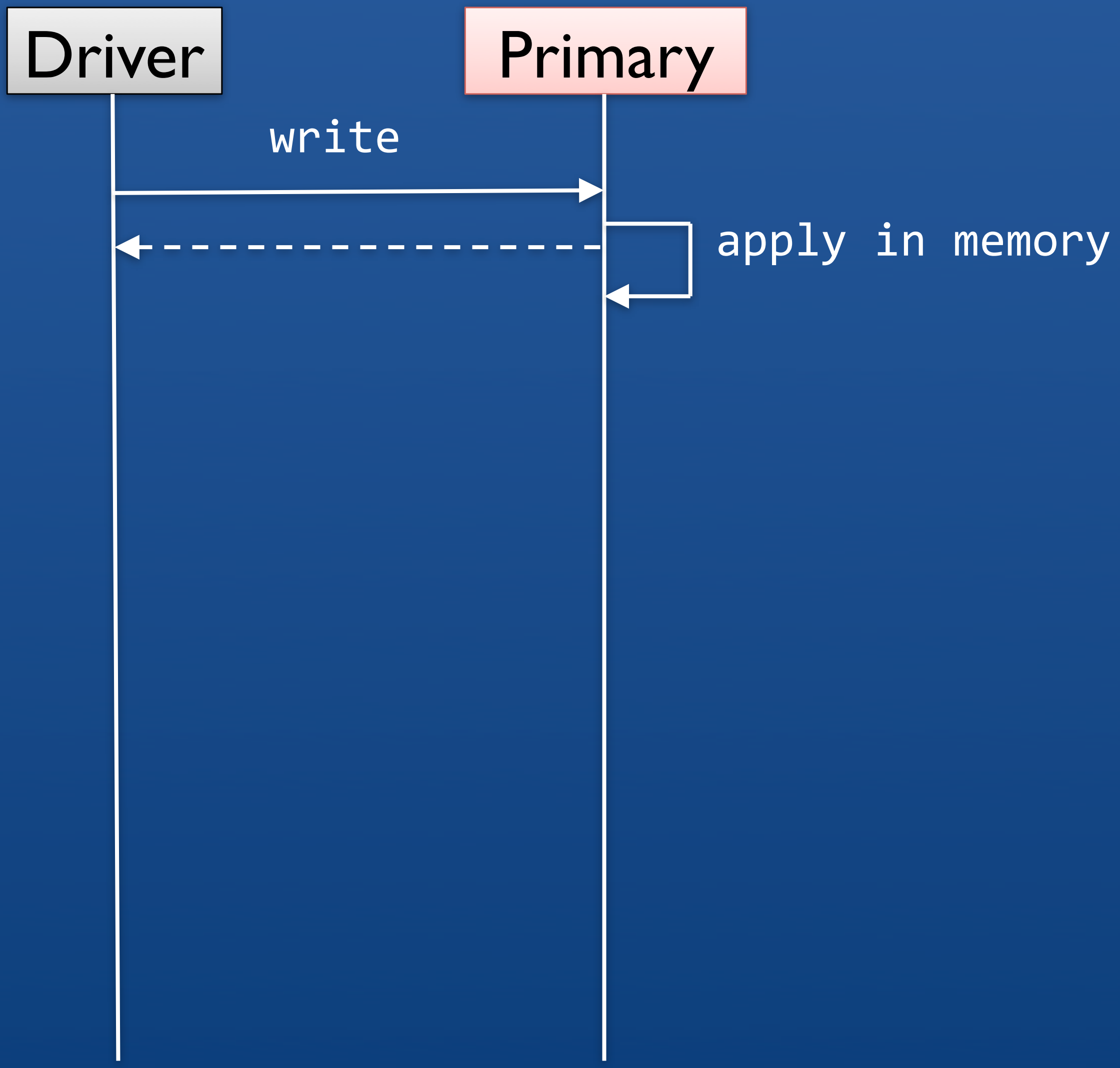
Replica Sets – Summary

- Data Protection
- High Availability
- Scaling eventual consistent reads
- Source to feed other systems
  - Backups
  - Indexes (Solr etc.)

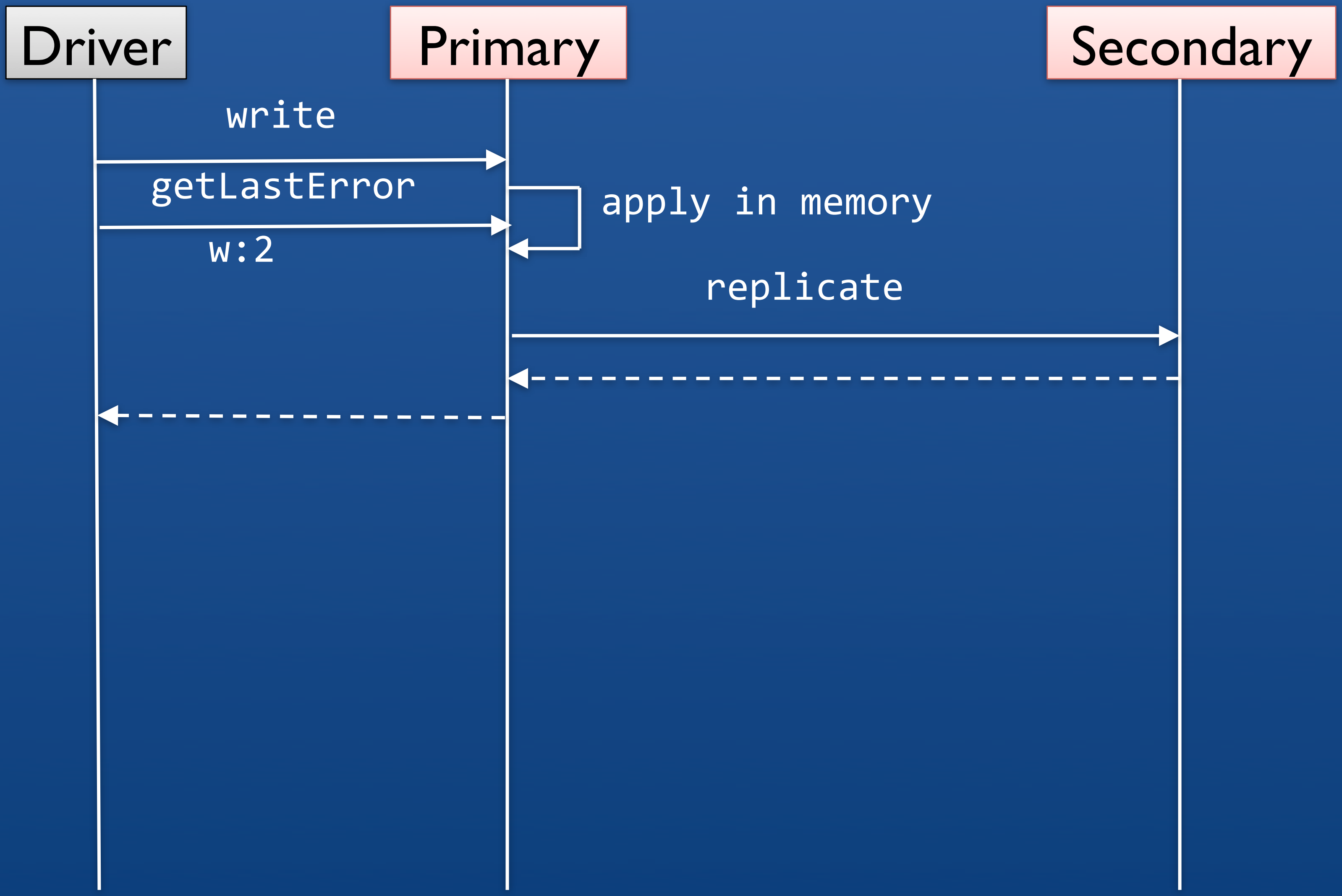
Types of Durability with MongoDB

- Fire and forget
- Wait for error
- Wait for fsync
- Wait for journal sync
- Wait for replication

Least durability - Don't use!

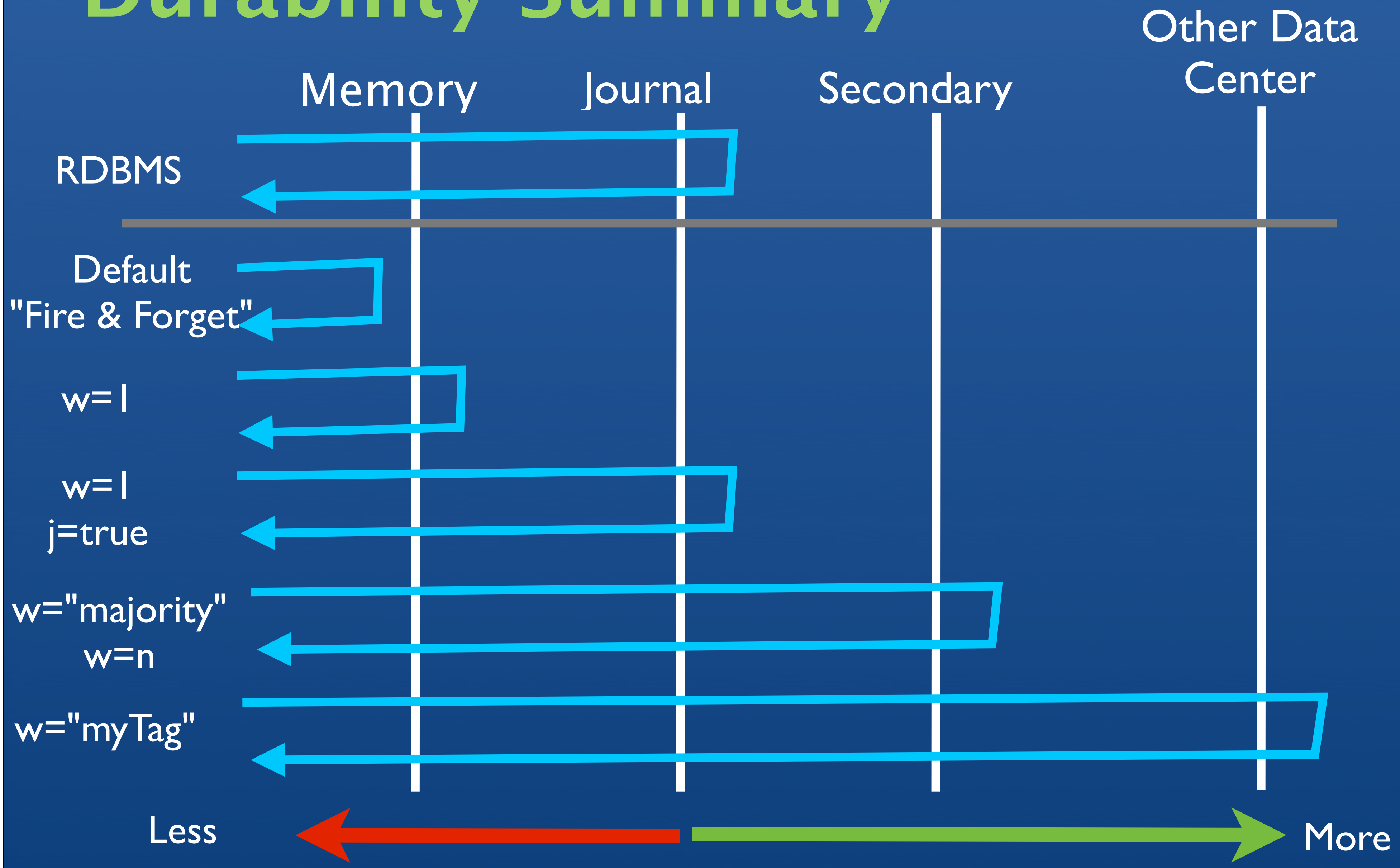


More durability



10gen | the MongoDB company

Durability Summary



10gen | the MongoDB company

Eventual Consistency
Using Replicas for Reads

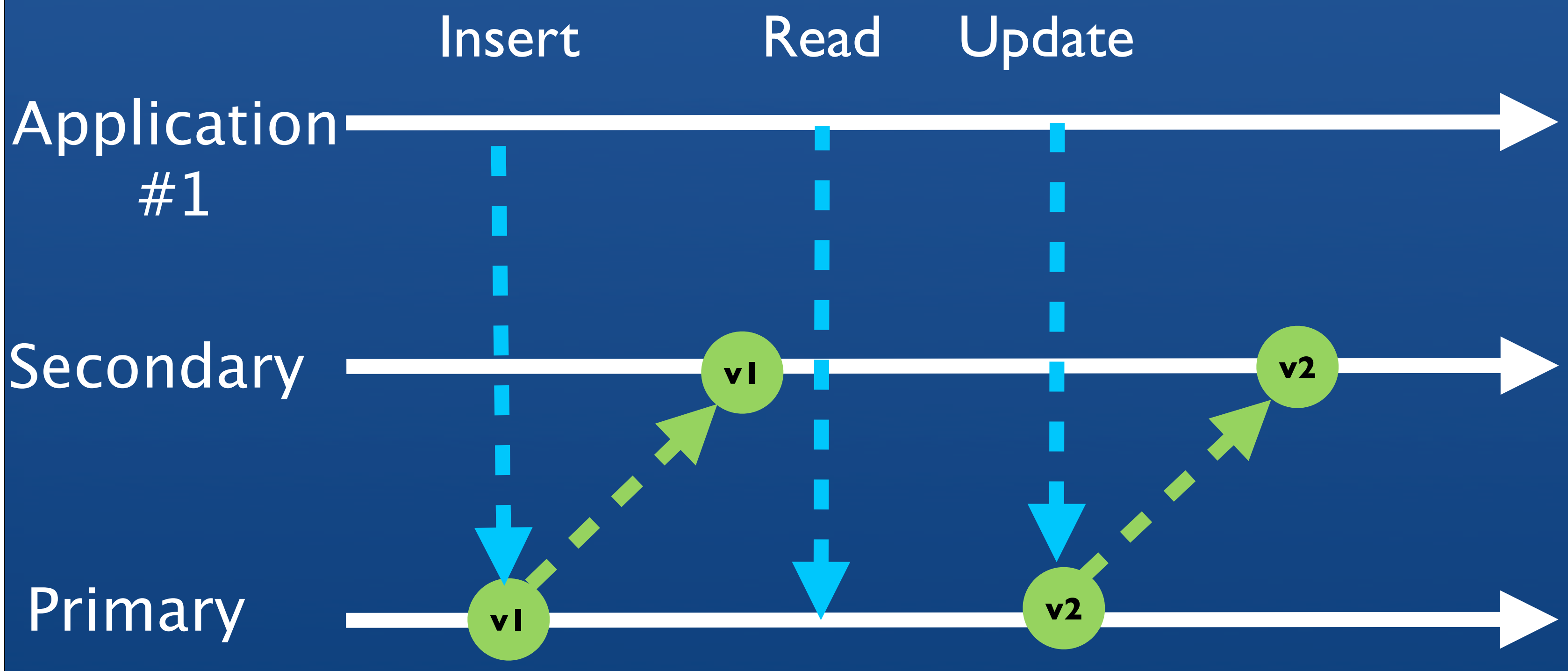
`slaveOk()`

- driver will send read requests to Secondaries
- driver will always send writes to Primary

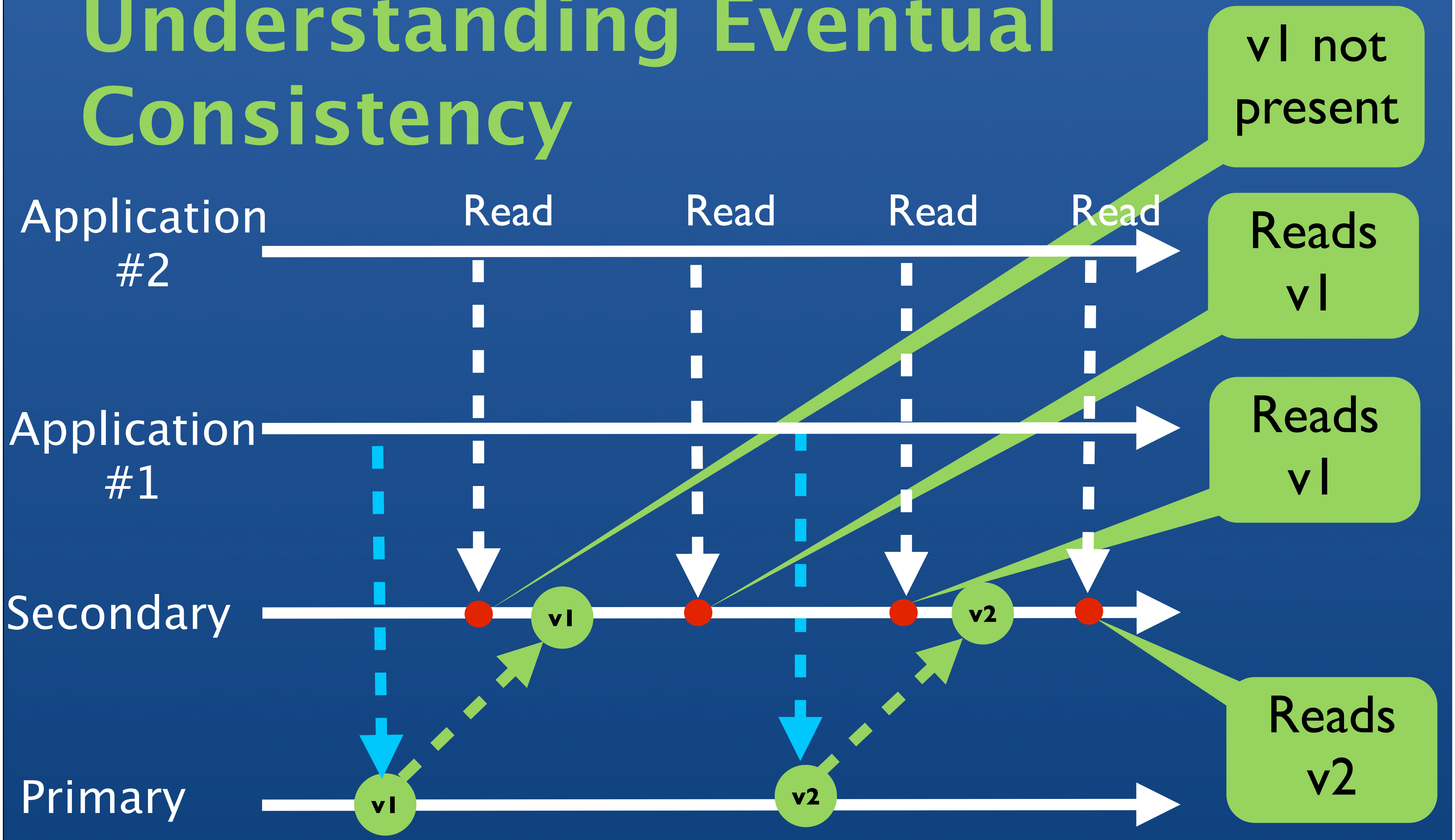
Java examples

- `DB.slaveOk()`
- `Collection.slaveOk()`
- `find(q).addOption(Bytes.QUERYOPTION_SLAVEOK);`

Understanding Eventual Consistency



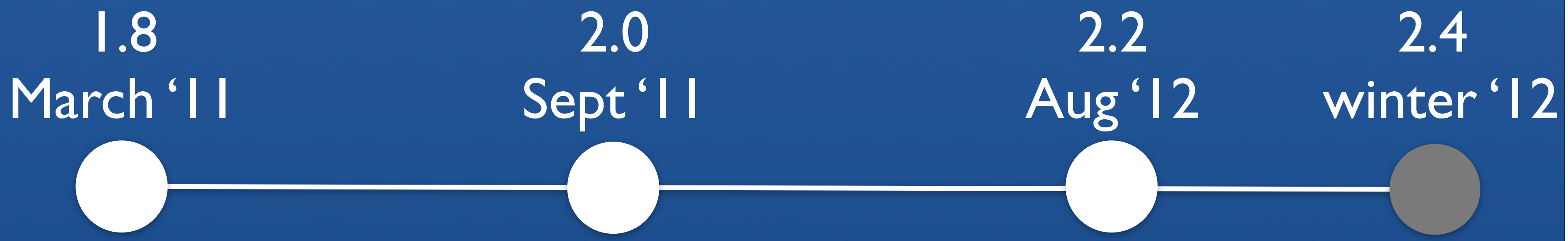
Understanding Eventual Consistency



Product & Roadmap

10gen | the MongoDB company

The Evolution of MongoDB



1.8
March ‘11

2.0
Sept ‘11

2.2
Aug ‘12

2.4
winter ‘12

1.8 – March ‘11

- Journaling
- Sharding and Replica set enhancements
- Spherical geo search

2.0 – Sept ‘11

- Index enhancements to improve size and performance
- Authentication with sharded clusters
- Replica Set Enhancements
- Concurrency improvements

2.2 – Aug ‘12

- Aggregation Framework
- Multi-Data Center Deployments
- Improved Performance and Concurrency

10gen | the MongoDB company

mongoDB

10gen | the MongoDB company