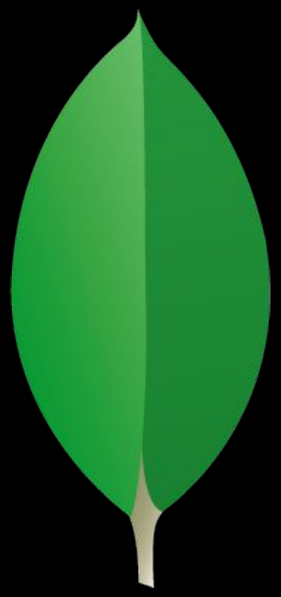
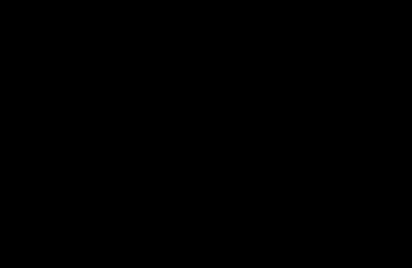


mongoDB

Technical Overview

Paul Pedersen, Deputy CTO
paul@10gen.com
Phoenix MUG
March 27, 2012

RDBMS systems



Costs go up

Productivity goes down

Project Start

→ **De-normalize data model**

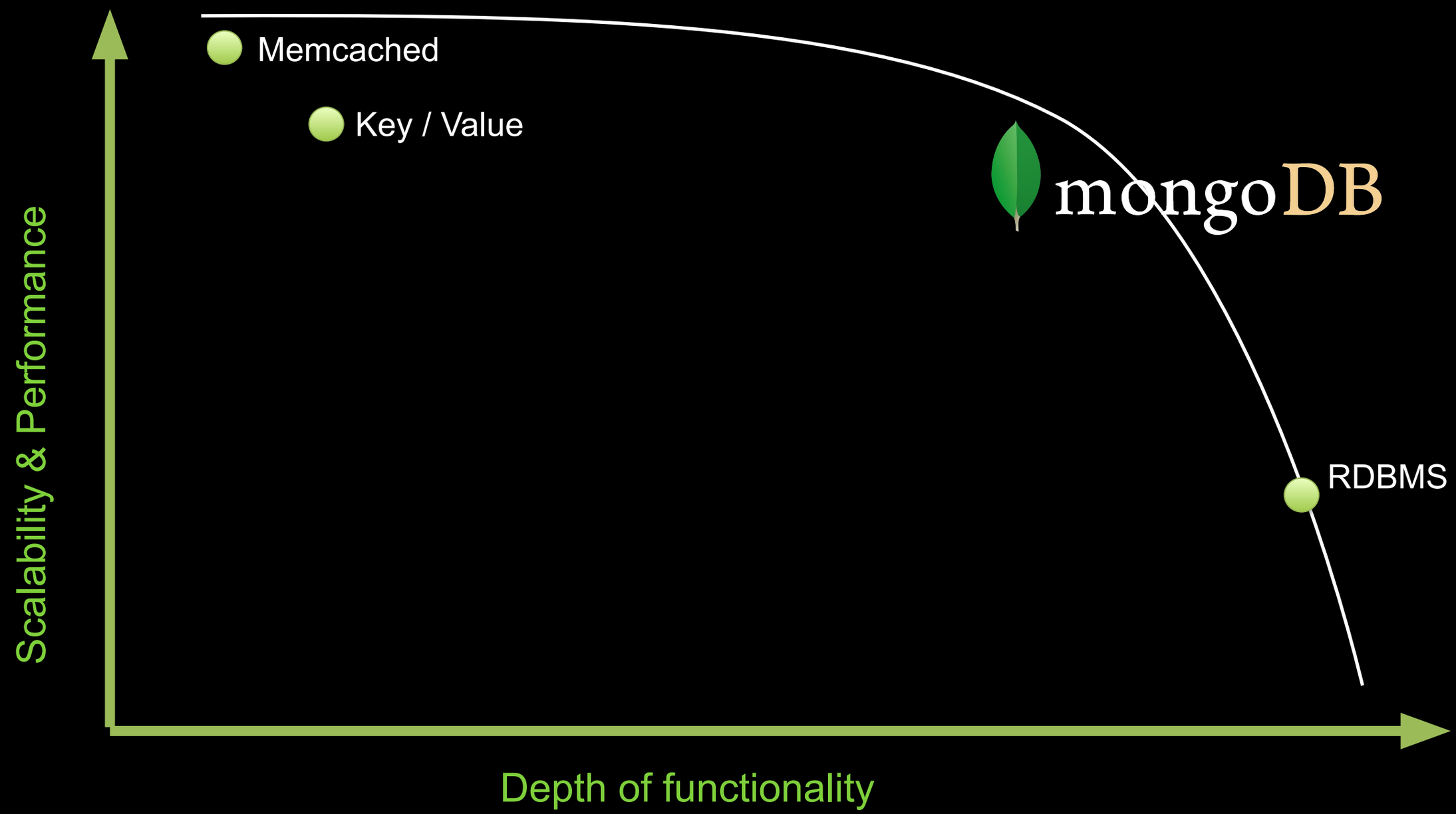
→ **Stop using joins**

→ **Custom caching layer**

→ **Custom sharding**

MongoDB / NoSQL objective:

Make it as simple as possible, but no simpler



mongodb stores BSON

```
{
  “_id” : 123456789,
  “name” : { “first”: “mongo”, “last”: “db” },
  “address” : “555 University Ave., Palo Alto CA 94301”,
  “checkins” : [ “starbucks”, “ace hardware”, “evvia” ],
  “loc” : [38.57, 121.62 ]
}
```

Types:

- unique id
- strings, dates, numbers, bool
- embedded documents
- arrays
- regex
- binary data

MongoDB architecture

- BSON object store, unique _id
- mmap memory management
- B-Tree indexing
- single-master replication
- logical key-range sharding
- global configuration manager

MongoDB design goals

- agile development
  - rich data model, well matched to OO data
  - general-purpose DBMS
  - simple, but powerful query syntax
  - multiple secondary indexes, geo indexes
  - smooth path to horizontal scaling
  - broad integration into existing ecosystems

MongoDB design goals

- web scale
  - eventual consistency: single master
  - availability: non-blocking writes, journal-blocking writes, replication-blocking writes
  - durability: journaling
  - partition: multi-data center replication
  - easy horizontal scaling by data sharding

MongoDB design goals

- High performance. For example:

- "Last week we learned about a government agency using mongodb to process about 2 million transactions per second. Around 50 machines in each of 8 data centers.."

- Wordnik stores its entire text corpus in MongoDB - 3.5T of data in 20 billion records. The speed to query the corpus was cut to 1/4 the time it took prior to migrating to MongoDB.

MongoDB scale

- We are running instances on the order of:
  - 25B objects
  - 50TB storage
  - 50K qps per server
  - 500 servers

SCHEMA DESIGN

Schema design checklist

- When do we embed data versus linking? Our decisions here imply the answer to question 2:
- How many collections do we have, and what are they?

1. When do we need atomic operations? These operations can be performed within the scope of a BSON document, but not across documents.

- What indexes will we create to make query and updates fast?
- How will we shard? What is the shard key?

Embedding vs. linking

Embedding is the nesting of objects and arrays inside a BSON document.

Links are references between documents (by _id)

There are no joins in MongoDB. (Distributed joins are inherently expensive.)

Embedding is a bit like "prejoined" data.

Embedding vs. linking

Operations within a document are easy for the server to handle. These operations can be fairly rich.

Links in contrast must be processed client-side by the application issuing a follow-up query.

For "contains" relationships between entities, embedding should be be chosen.

Use linking when not using linking would result in gross duplication of data.

Embedding vs. linking

Embedded objects are a bit harder to link to than "top level" objects in collections.
(http://www.mongodb.org/display/DOCS/Dot+Notation+(Reaching+into+Objects))

If the amount of data to embed is large, you may reach the limit on size of a single object. Then consider using GridFS.

If performance is an issue, embed.

Example

Our content management system will manage posts.

Posts have titles, dates, and authors.

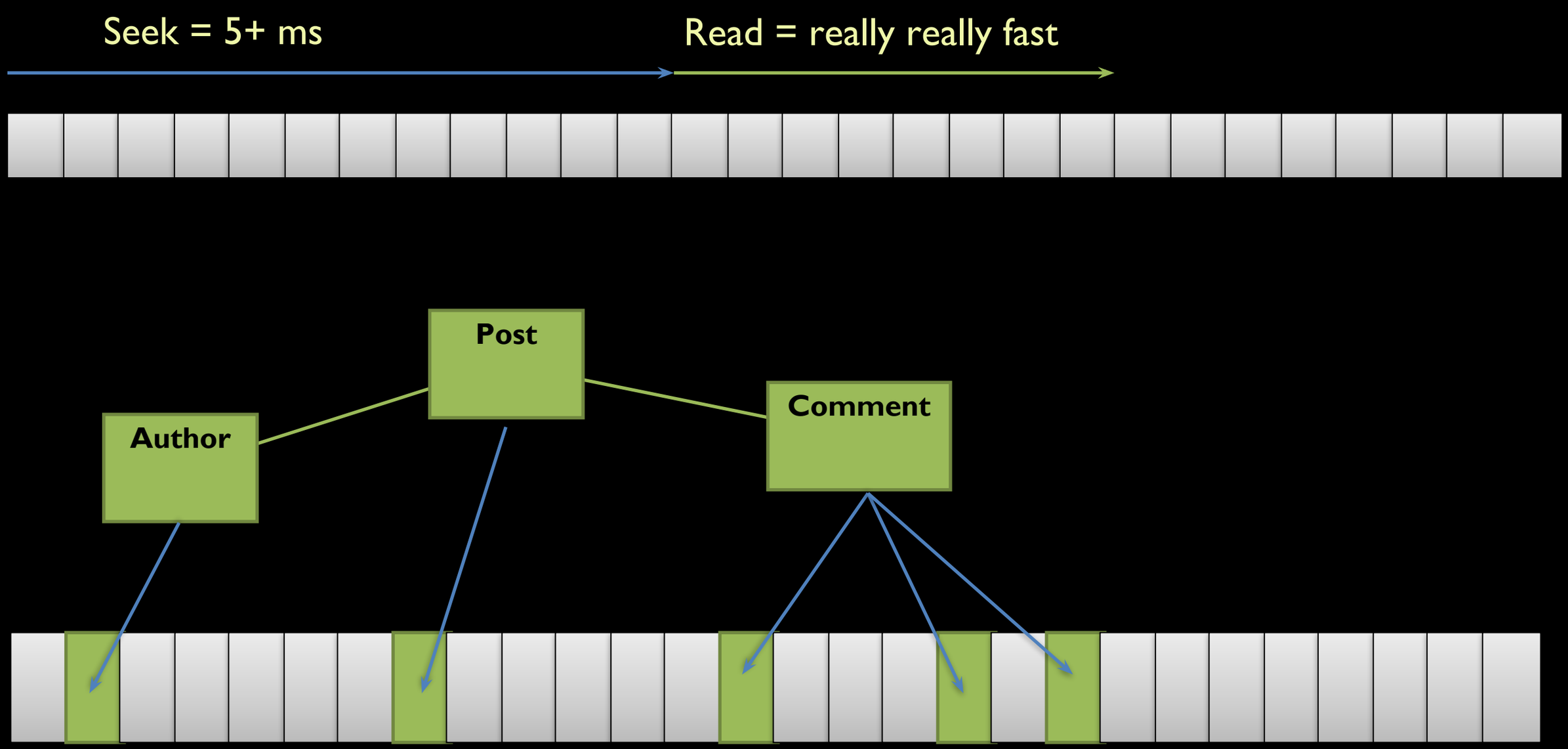
We'd like to support commenting and voting on posts.

We'd also like posts to be taggable for searching.

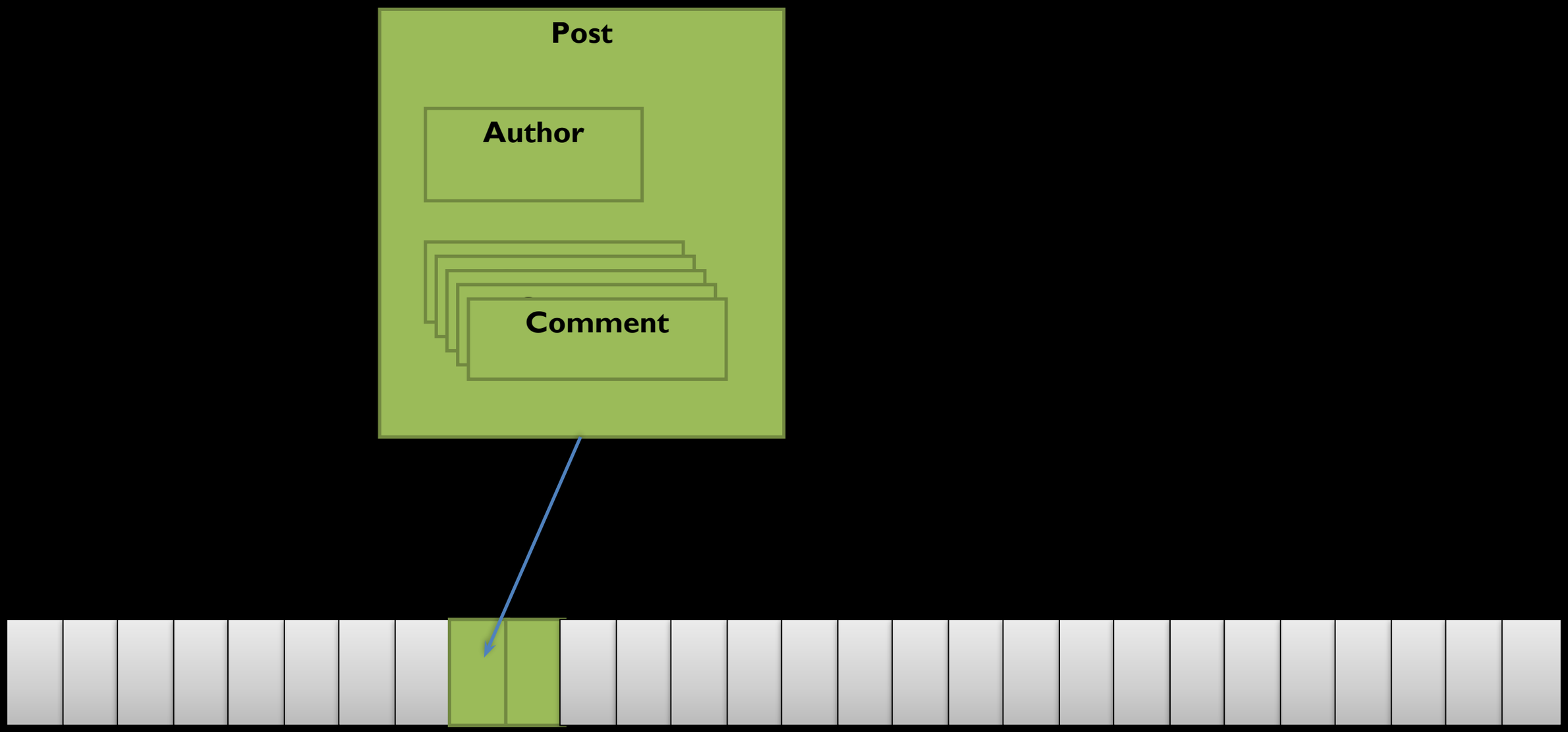
Example

```
> db.posts.findOne()
{
  _id : ObjectId("4e77bb3b8a3e000000004f7a"),
  when : Date("2011-09-19T02:10:11.3Z",
  author : "alex",
  title : "No Free Lunch",
  text : "This is the text of the post.  It could be very long.",
  tags : [ "business", "ramblings" ],
  votes : 5,
  voters : [ "jane", "joe", "spencer", "phyllis", "li" ],
  comments : [
    { who : "jane", when : Date("2011-09-19T04:00:10.112Z"),
      comment : "I agree." },
    { who : "meghan", when : Date("2011-09-20T14:36:06.958Z"),
      comment : "You must be joking.  etc etc ..." }
  ]
}
```

Disk seeks and data locality



Disk seeks and data locality



Collections

Collections of documents in MongoDB are analogous to tables of rows in a relational database.

There is no explicit schema declaration. No schema object.

There is a conceptual design of how the documents in the collection will be structured.

Collections

MongoDB does not require documents in a collection to have the same structure.

In practice, most collections are relatively homogenous.

We can move away from this when we need -- for example when adding a new field. No analog of "alter table" is required.

Atomicity

Some problems require the ability to perform atomic operations. For example:

```
atomically {
  if (doc.credits > 5) {
    doc.credits -= 5;
    doc.debits += 5;
  }
}
```

The scope of atomicity / ACID properties is the document.

We need to assure that all fields relevant to the atomic operation are in the same document.

Indexes

Indexes in MongoDB are highly analogous to indexes in a relational database. (B-Tree indexes)

Indexes are needed for efficient query processing: both for selection and sorting.

Indexes must be explicitly declared.

As in a relational database, indexes can be added to existing collections.

Choosing indexes

As a general rule -- where you want an index in a relational database, you want an index in MongoDB.

- The _id field is automatically indexed.
- Fields whose keys are looked up should be indexed.
- Sort fields generally should be indexed.

The MongoDB profiling facility provides useful information for where an index should be added that is missing.
Adding an index slows writes to a collection, but not reads.

Example

To grab the whole post we might execute:

```
> db.posts.findOne( { _id : ObjectId("4e77bb3b8a3e000000004f7a") } );
```

To get all posts written by alex:

```
> db.posts.find( { author : "alex" } )
```

If the above is a common query we would create an index on the author field:

```
> db.posts.ensureIndex( { author : 1 } )
```

To get just the titles of the posts by alex:

```
> db.posts.find( { author : "alex" }, { title : 1 } )
```

Example

We may want to search for posts by tag:

```
> // make and index of all tags so that the query is fast:
> db.posts.ensureIndex( { tags : 1 } )
> db.posts.find( { tags : "business" } )
```

What if we want to find all posts commented on by meghan?

```
> db.posts.find( { comments.who : "meghan" } )
```

Let's index that to make it fast:

```
> db.posts.ensureIndex( { "comments.who" : 1 } )
```

Example

We track voters above so that no one can vote more than once.

Suppose calvin wants to vote for the example post above.

The following update operation will record calvin's vote. Because of the $nin sub-expression, if calvin has already voted, the update will have no effect.

```
> db.posts.update( { _id : ObjectId("4e77bb3b8a3e000000004f7a"),
                     voters : { $nin : "calvin" } },
                   { votes : { $inc : 1 }, voters : { $push :
"calvin" } );
```

Note the above operation is atomic : if multiple users vote simultaneously, no votes would be lost.

Sharding

A collection may be sharded (i.e. partitioned).

When sharded, the collection has a shard key, which determines how the collection is partitioned among shards.

A BSON document (which may have significant amounts of embedding) resides on one and only one shard.

Typically (but not always) queries on a sharded collection involve the shard key as part of the query expression.

Changing shard keys is difficult. You will want to choose the right shard key from the start.

Sharding

mongos

balancer

config

config

config

Chunks = logical key range

1	2	3	4
5	6	7	8
9	10	11	12

Shard 1

13	14	15	16
17	18	19	20
21	22	23	24

Shard 2

25	26	27	28
29	30	31	32
33	34	35	36

Shard 3

37	38	39	40
41	42	43	44
45	46	47	48

Shard 4

Example

If the *posts* collection is small, we would not need to shard it at all. But if *posts* is huge, we probably want to shard it.

The shard key should be chosen based on the queries that will be common. We want those queries to involve the shard key.

- Sharding by _id is one option here.
- If finding the most recent posts is a very frequent query, we would then shard on the *when* field.

Shard key best practices

It is very important that the shard key is granular enough that data can be partitioned among many machines.

One of the primary reasons for using sharding is to distribute writes. To do this, its important that writes hit as many chunks as possible.

Another key consideration is how many shards any query has to hit. Ideally a query would go directly from mongos to the shard (mongod) that owns the document requested.

Shard key best practices

A query including a sort is sent to the same shards that would have received the equivalent query without the sort expression. Each such shard performs a sort on its subset of the data. Mongos merges the ordered .

it is usually much better performance-wise if only a portion of the index is read/updated, so that this "active" portion can stay in RAM most of the time.

Most shard keys described above result in an even spread over the shards but also within each mongod's index. It can be beneficial to factor in some kind of timestamp at the beginning of the shard key in order to limit the portion of the index that gets hit.

Example

Say you have a photo storage system, with photo entries:

```
{
    _id: ObjectId("4d084f78a4c8707815a601d7"),
    user_id : 42 ,
    title: "sunset at the beach",
    upload_time : "2011-01-02T21:21:56.249Z" ,
    data: ...,
}
```

You could instead build a custom *id* that includes the month of the upload time, and a unique identifier (ObjectId, MD5 of data, etc):

```
{
    _id: "2011-01_4d084f78a4c8707815a601d7",
    user_id : 42 ,
    title: "sunset at the beach",
    upload_time : "2011-01-02T21:21:56.249Z" ,
    data: ...,
}
```

This custom id would be the key used for sharding, and also the id used to retrieve a document. It gives both a good spread across all servers, and it reduces the portion of the index hit on most queries.

Sharding: How it works

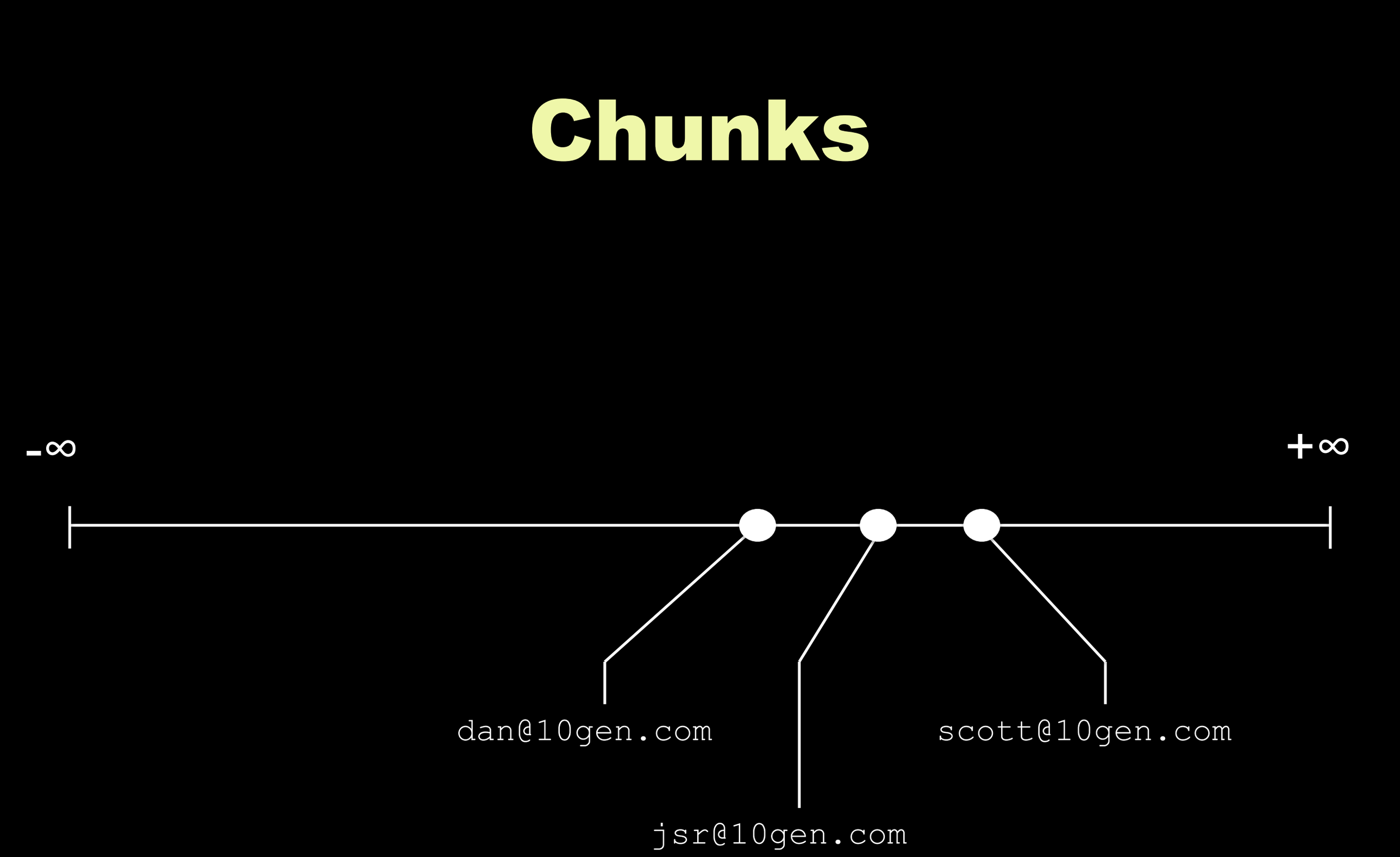
Keys

```
> db.runCommand( { shardcollection: "test.users",
                   key: { email: 1 }} )
```

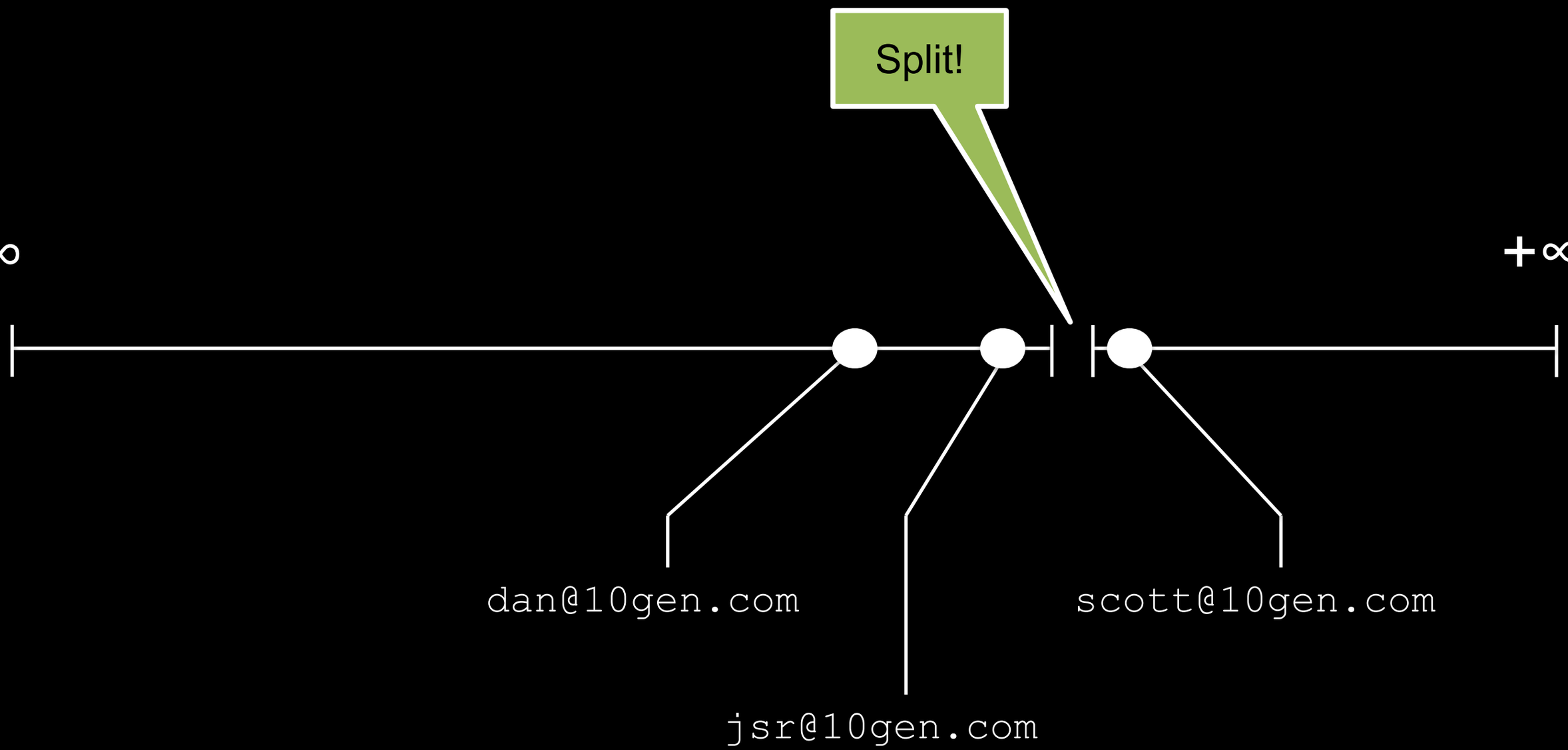
```
• {
•    name: "Jared",
•    email: "jsr@10gen.com",
• }
• {
•    name: "Scott",
•    email: "scott@10gen.com",
• }
• {
•    name: "Dan",
•    email: "dan@10gen.com",
• }
```

Chunks

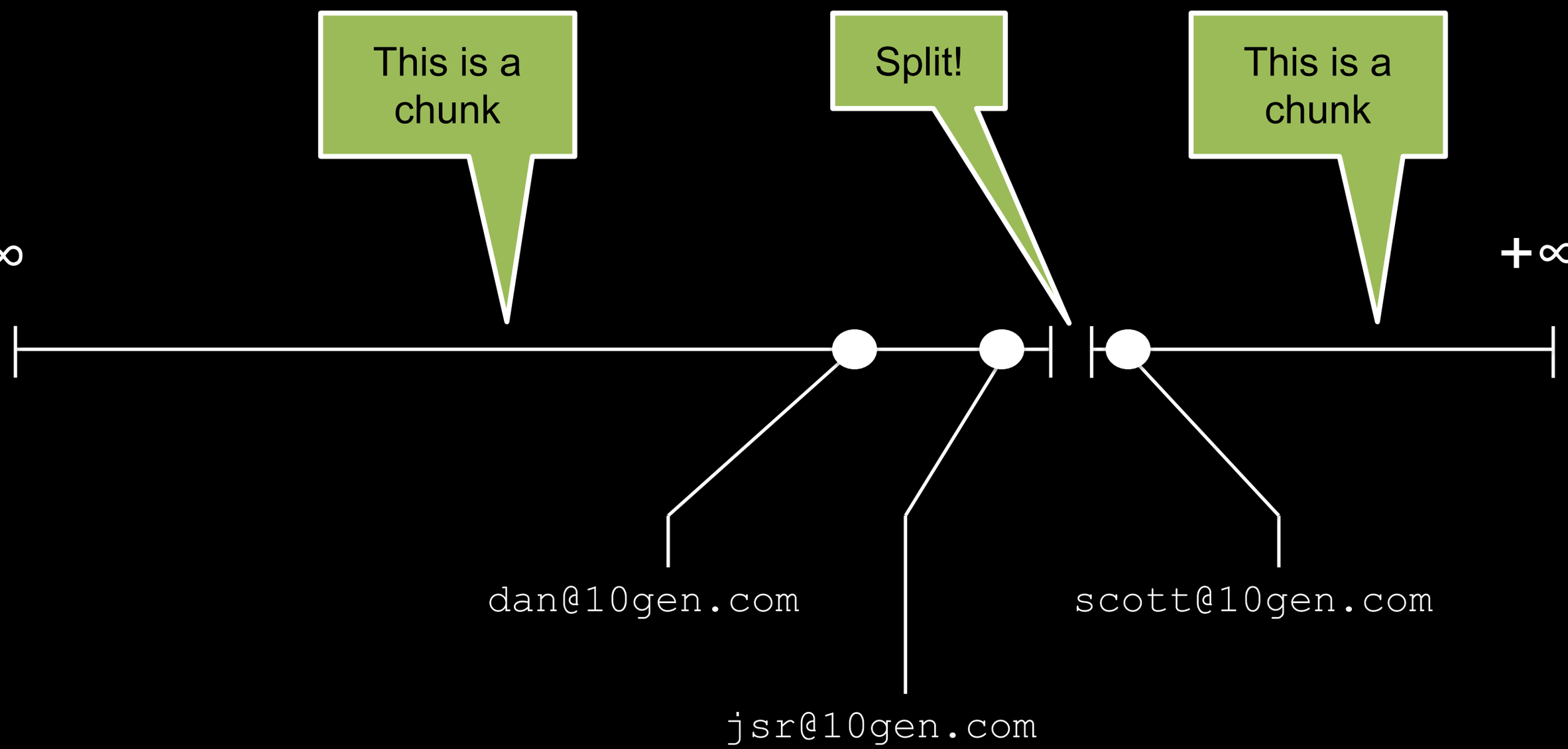
Chunks



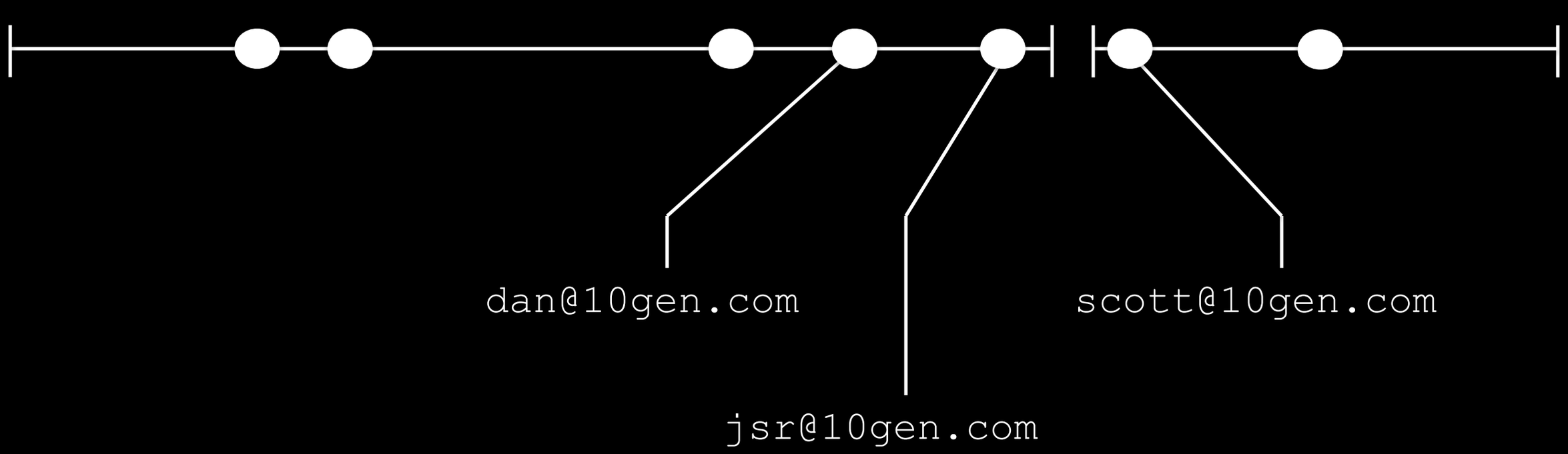
Chunks



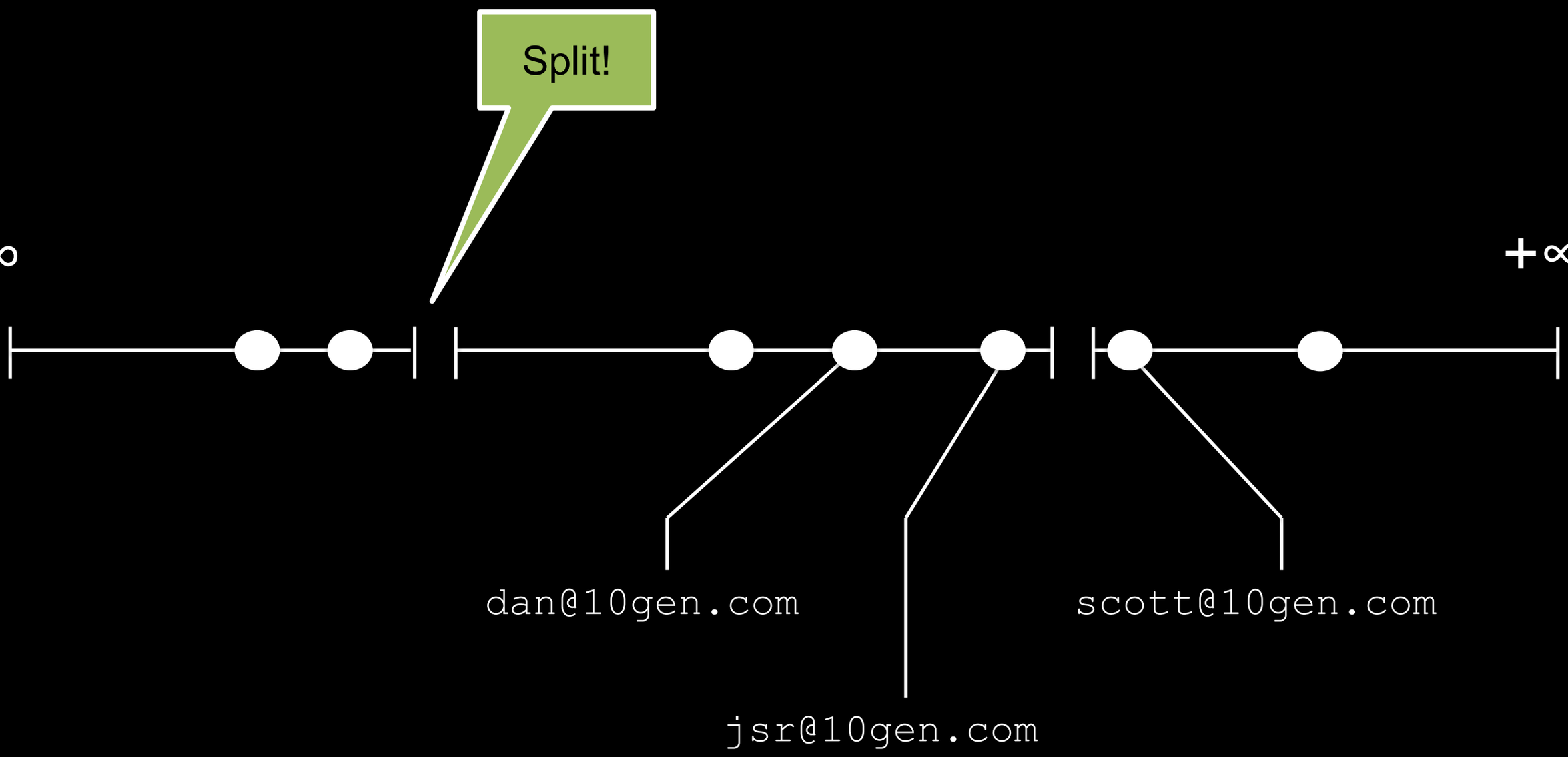
Chunks


This is a chunk
Split!
This is a chunk
-∞
+∞
dan@10gen.com
jsr@10gen.com
scott@10gen.com

Chunks



Chunks



Chunks

Min Key	Max Key	Shard
-∞	adam@10gen.com	1
adam@10gen.com	jared@10gen.com	1
jared@10gen.com	scott@10gen.com	1
scott@10gen.com	+∞	1

- Stored in the config servers
- Cached in mongos
- Used to route requests and keep cluster balanced

Balancing

mongos

balancer

config

config

config

Chunks!

1	2	3	4
5	6	7	8
9	10	11	12

Shard 1

13	14	15	16
17	18	19	20
21	22	23	24

Shard 2

25	26	27	28
29	30	31	32
33	34	35	36

Shard 3

37	38	39	40
41	42	43	44
45	46	47	48

Shard 4

Balancing

mongos

balancer

config

config

config

Imbalance

1	2	3	4
5	6	7	8
9	10	11	12

Shard 1

21	22	23	24

Shard 2

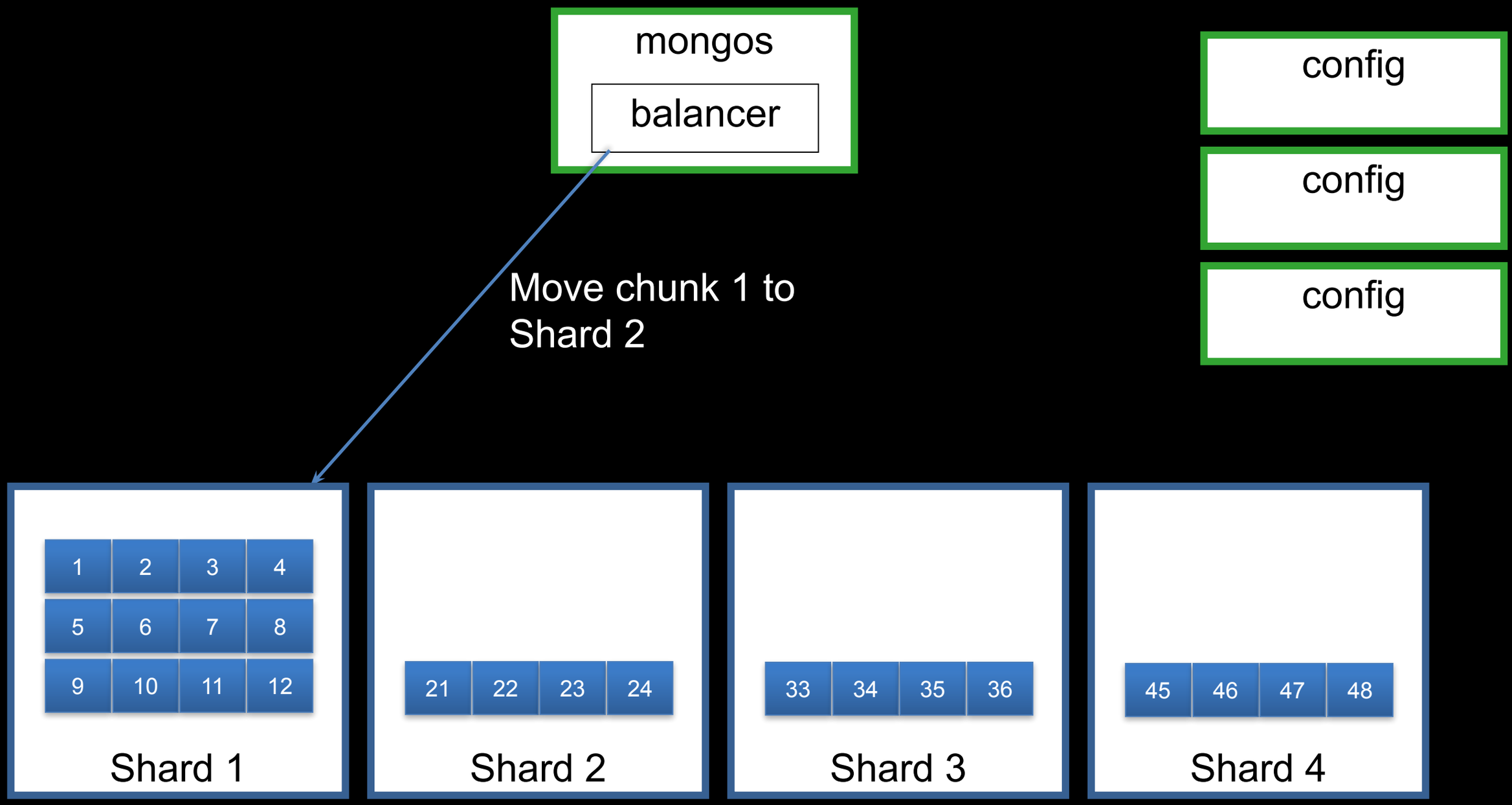
33	34	35	36

Shard 3

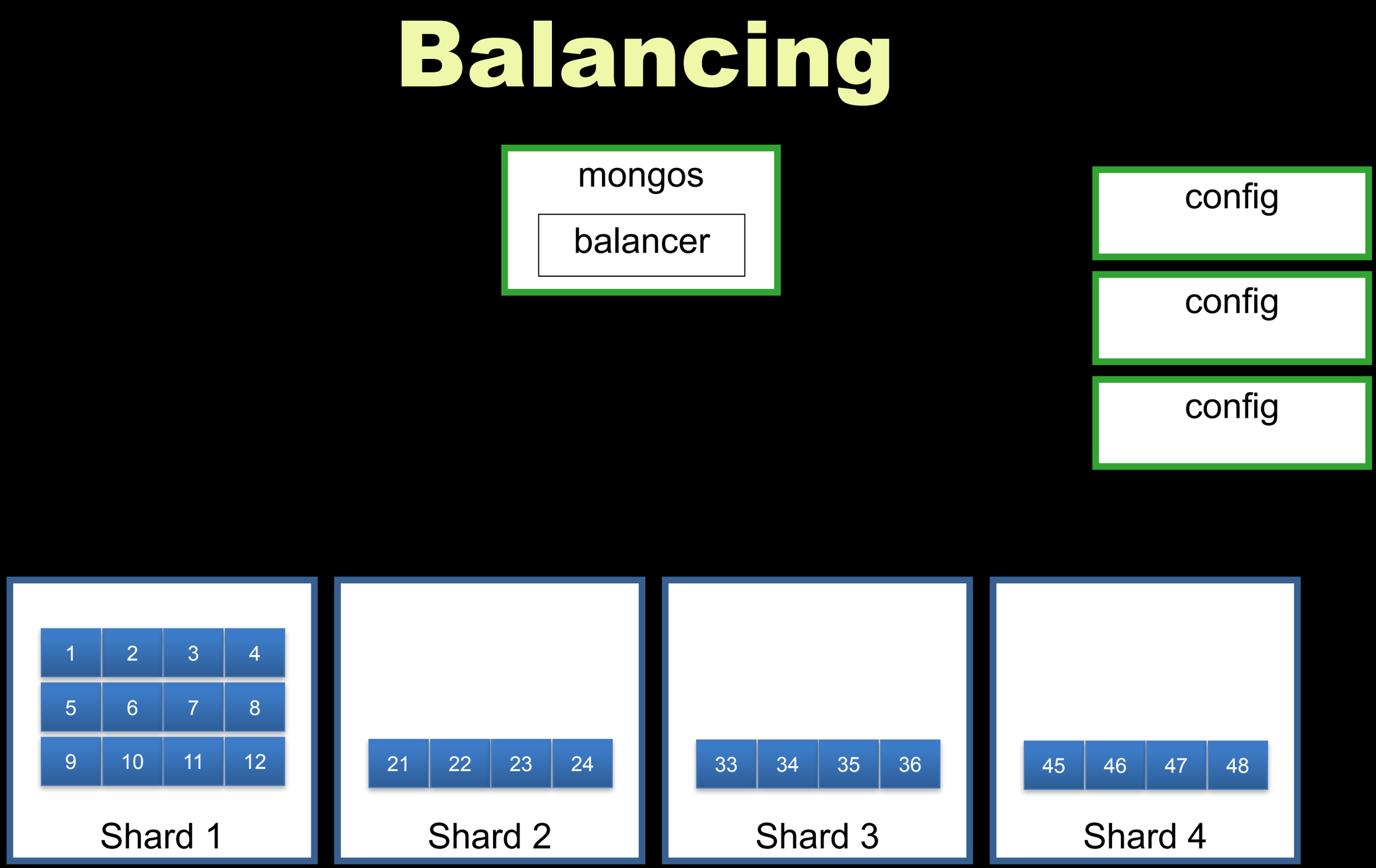
45	46	47	48

Shard 4

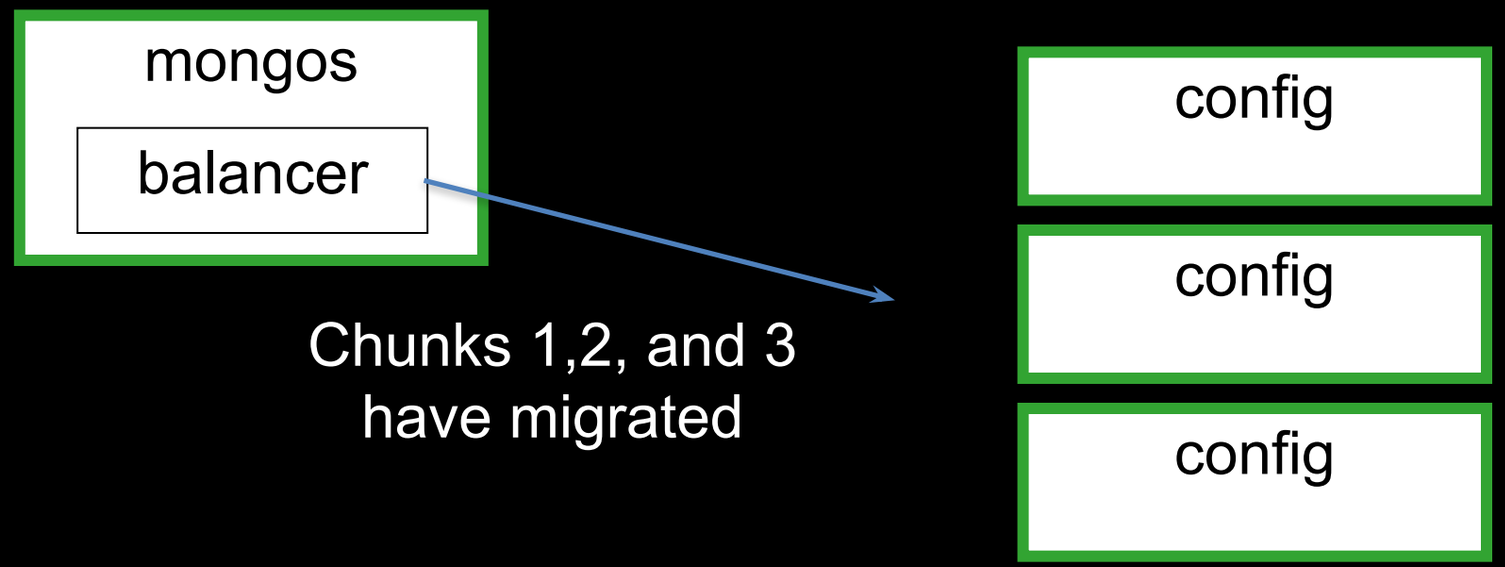
Balancing

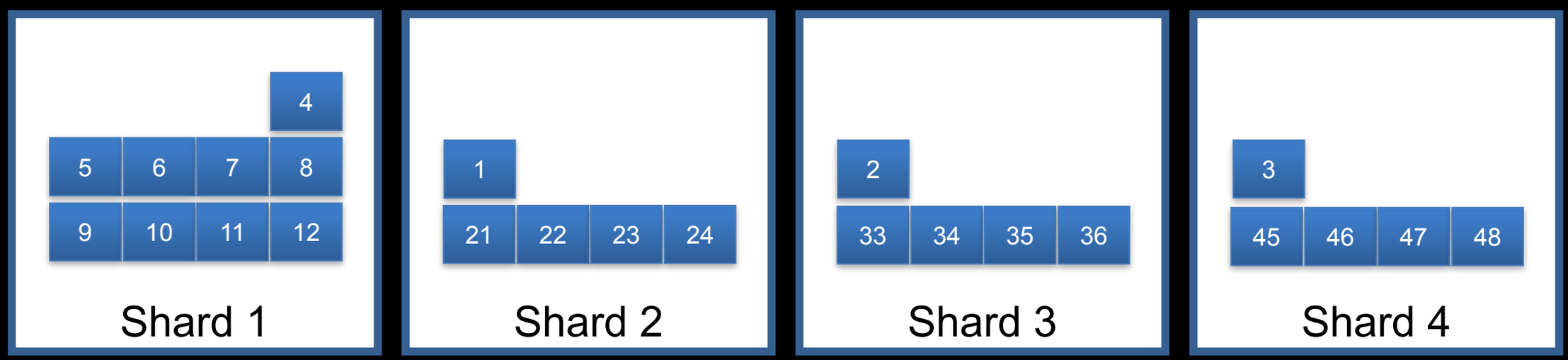

mongos
balancer
config
config
config
Move chunk 1 to
Shard 2
1
2
3
4
5
6
7
8
9
10
11
12
Shard 1
21
22
23
24
Shard 2
33
34
35
36
Shard 3
45
46
47
48
Shard 4

Balancing


mongos
balancer
config
config
config
1
2
3
4
5
6
7
8
9
10
11
12
Shard 1
21
22
23
24
Shard 2
33
34
35
36
Shard 3
45
46
47
48
Shard 4

Balancing

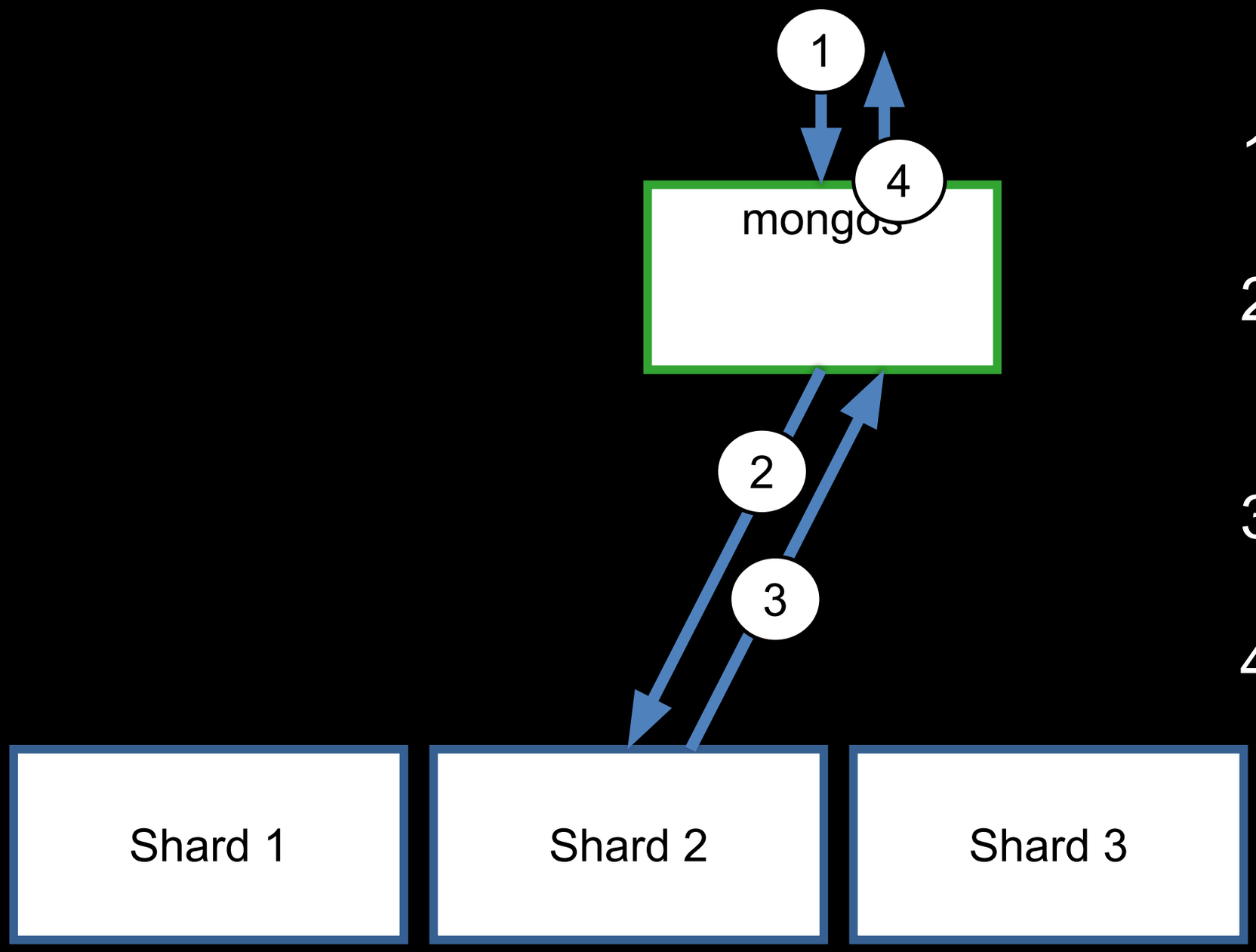

mongos
balancer
config
config
config
Chunks 1,2, and 3
have migrated


4
5
6
7
8
9
10
11
12
Shard 1
1
21
22
23
24
Shard 2
2
33
34
35
36
Shard 3
3
45
46
47
48
Shard 4

Choosing a Shard Key

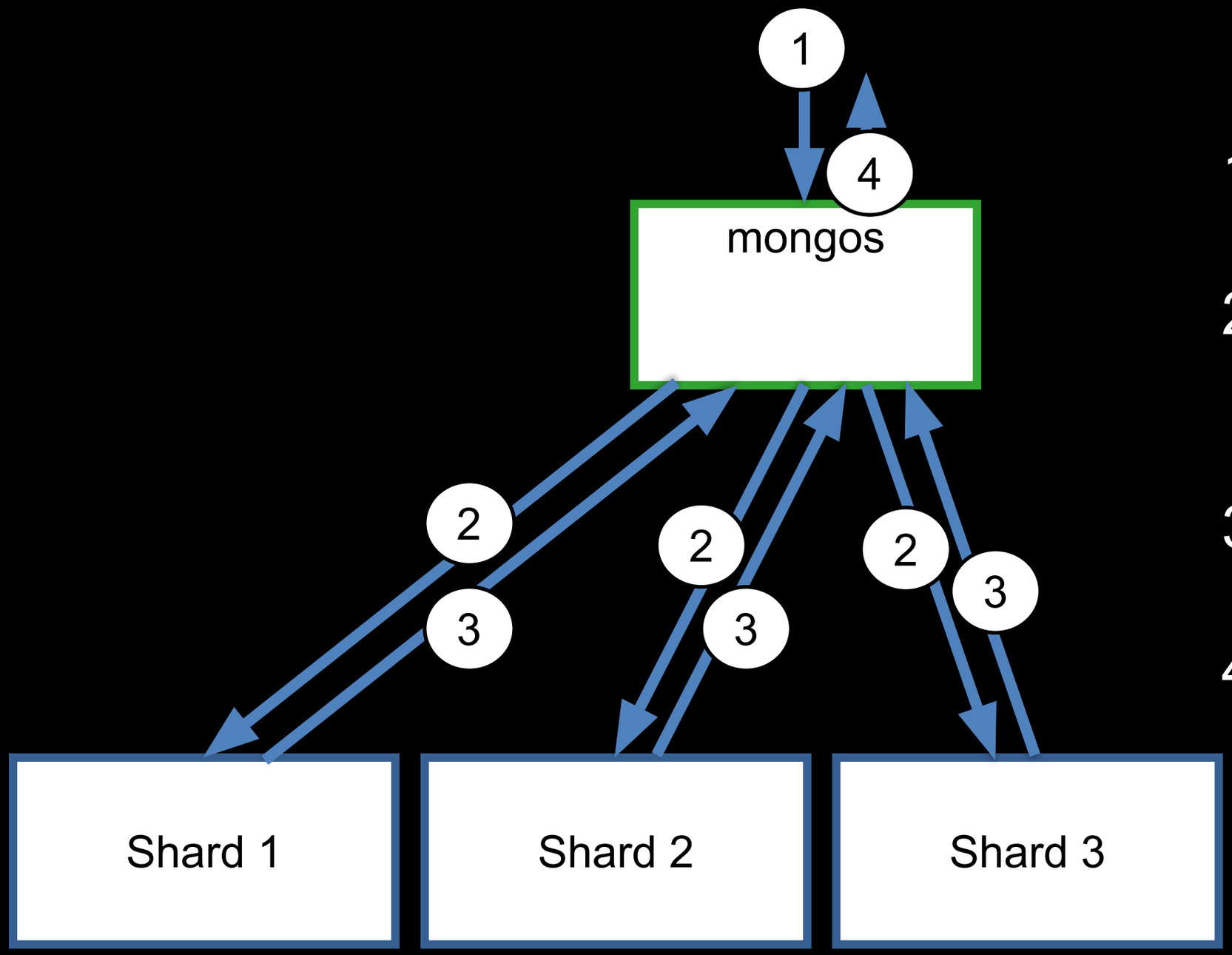
Routing

Routed Request



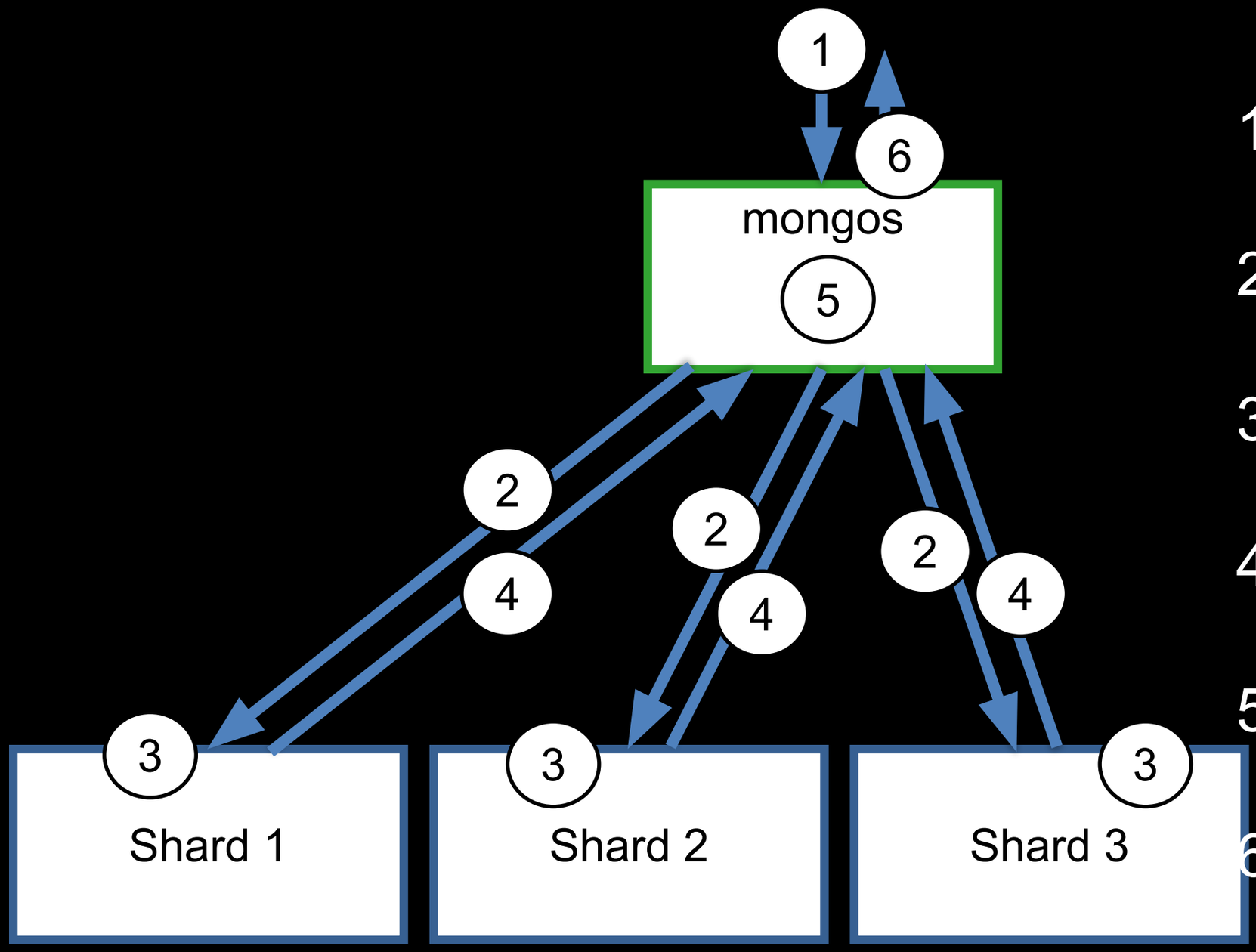
1. Query arrives at mongos
2. mongos routes query to a single shard
3. Shard returns results of query
4. Results returned to client

Scatter Gather



1. Query arrives at mongos
2. mongos broadcasts query to all shards
3. Each shard returns results for query
4. Results combined and returned to client

Distributed Merge Sort



1. Query arrives at mongos
2. mongos broadcasts query to all shards
3. Each shard locally sorts results
4. Results returned to mongos
5. mongos merge sorts individual results
6. Combined sorted result returned to client

Writes

Inserts	Requires shard key	`db.users.insert({`
		`  name: "Jared",`
		`  email: "jsr@10gen.com"})`

Removes	Routed	`db.users.delete({` `  email: "jsr@10gen.com"})`
	Scattered	`db.users.delete({name: "Jared"})`

Updates	Routed	`db.users.update(`
		`  {email: "jsr@10gen.com"},`
		`  {$set: { state: "CA"}})`

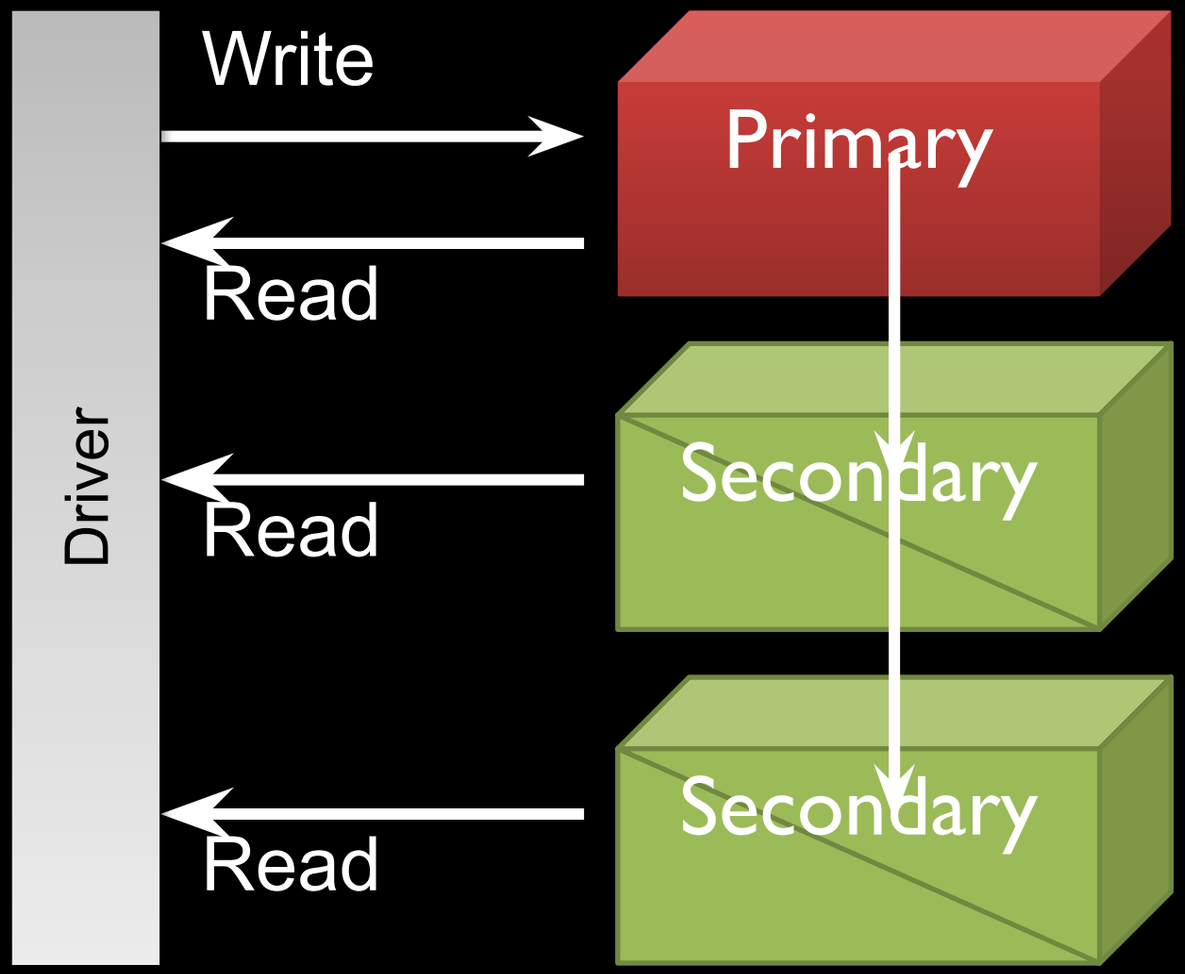
	Scattered	`db.users.update(`
		`  {state: "FZ"},`
		`  {$set:{ state: "CA"}} )`

Queries

By Shard Key	Routed	`db.users.find( {email: "jsr@10gen.com"})`
Sorted by shard key	Routed in order	`db.users.find().sort({email:-1})`
Find by non shard key	Scatter Gather	`db.users.find({state:"CA"})`
Sorted by non shard key	Distributed merge sort	`db.users.find().sort({state:1})`

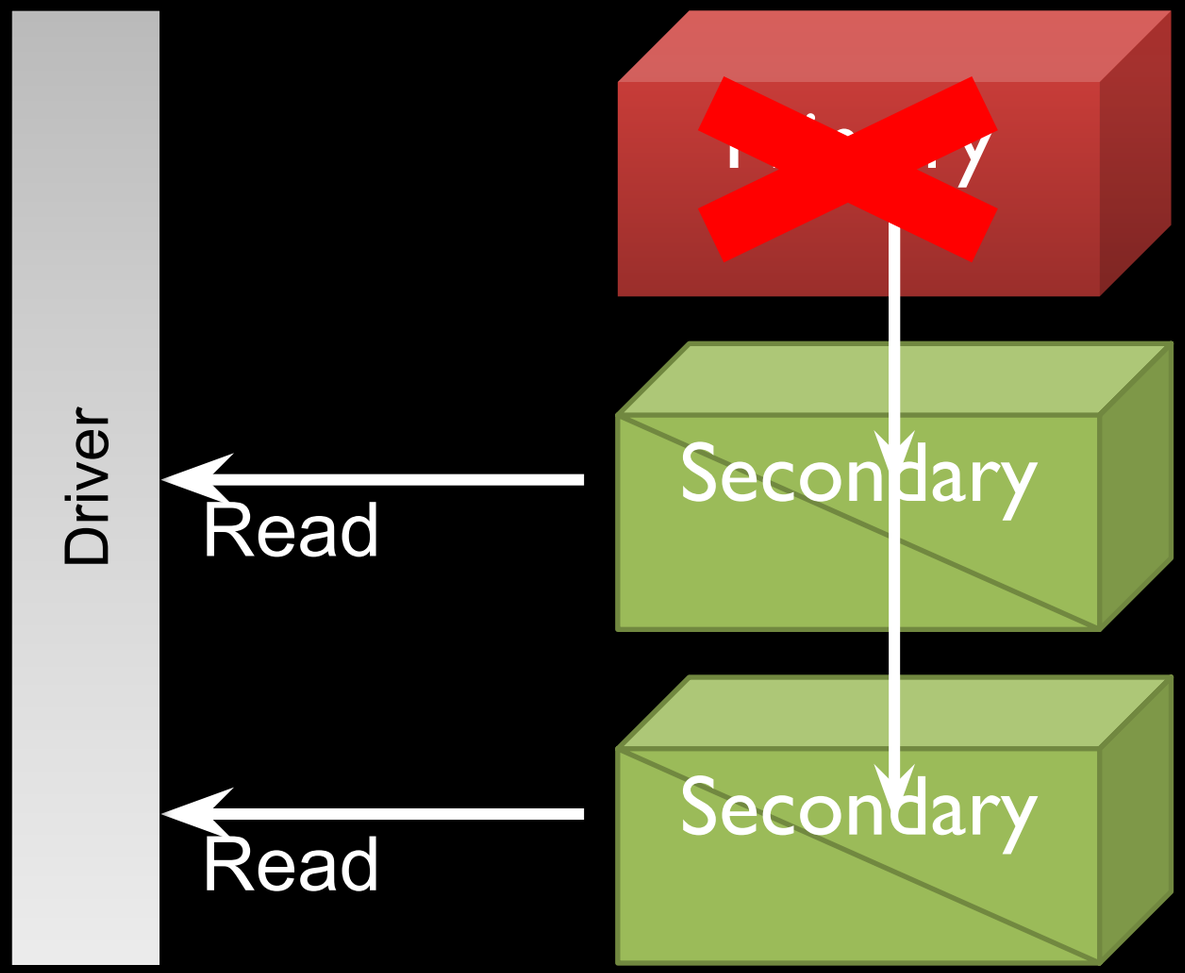
REPLICATION: HOW IT WORKS

Replica Sets

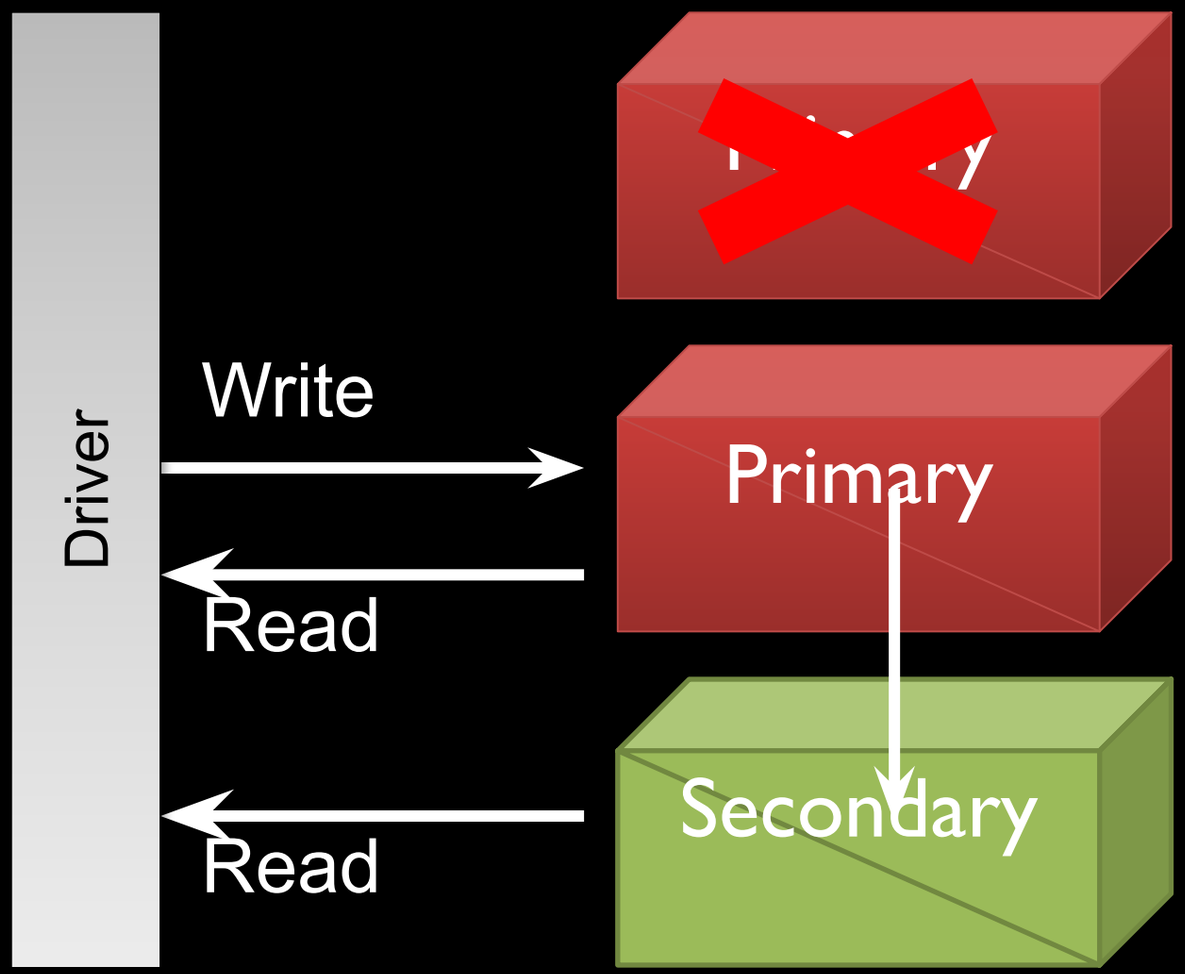


Asynchronous Replication

Replica Sets

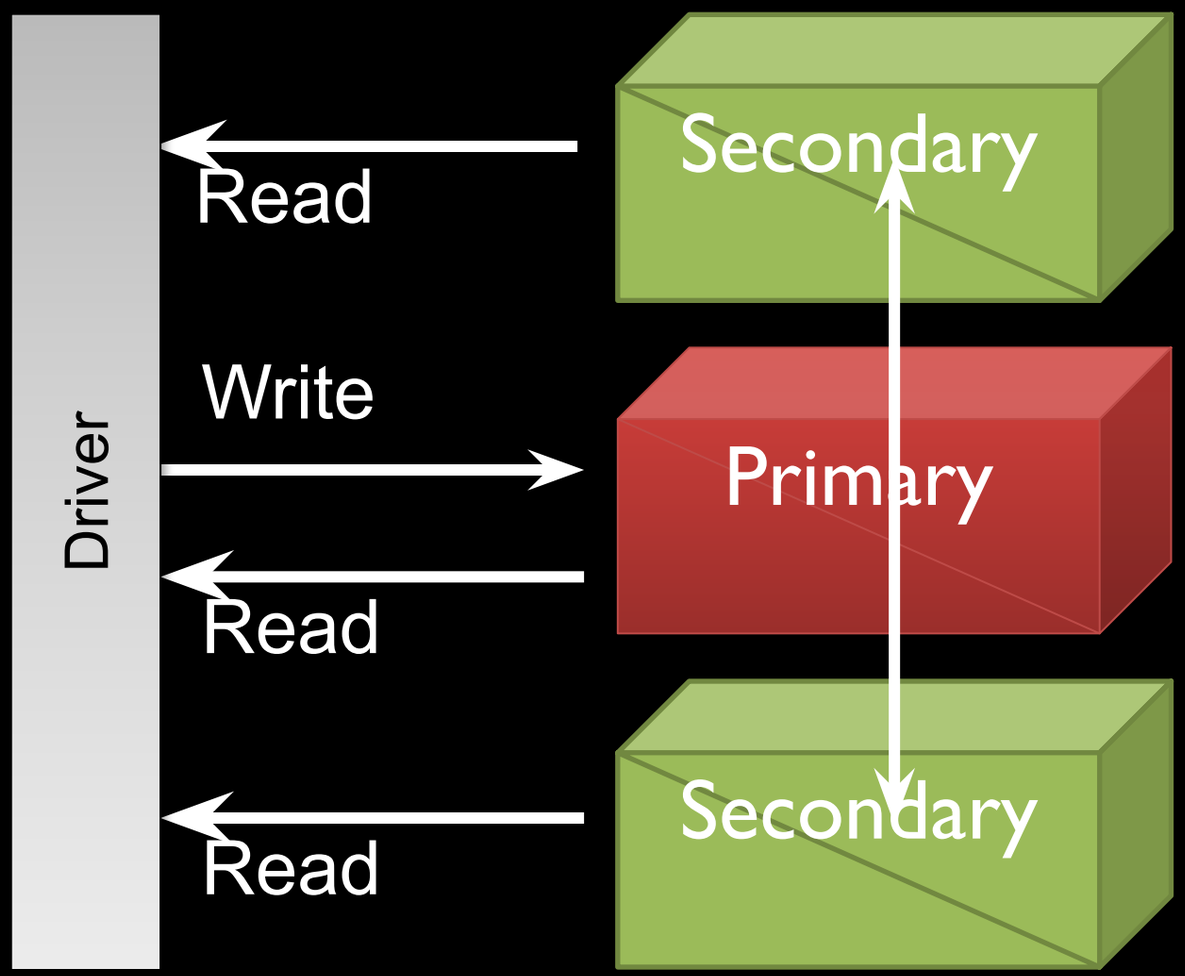


Replica Sets



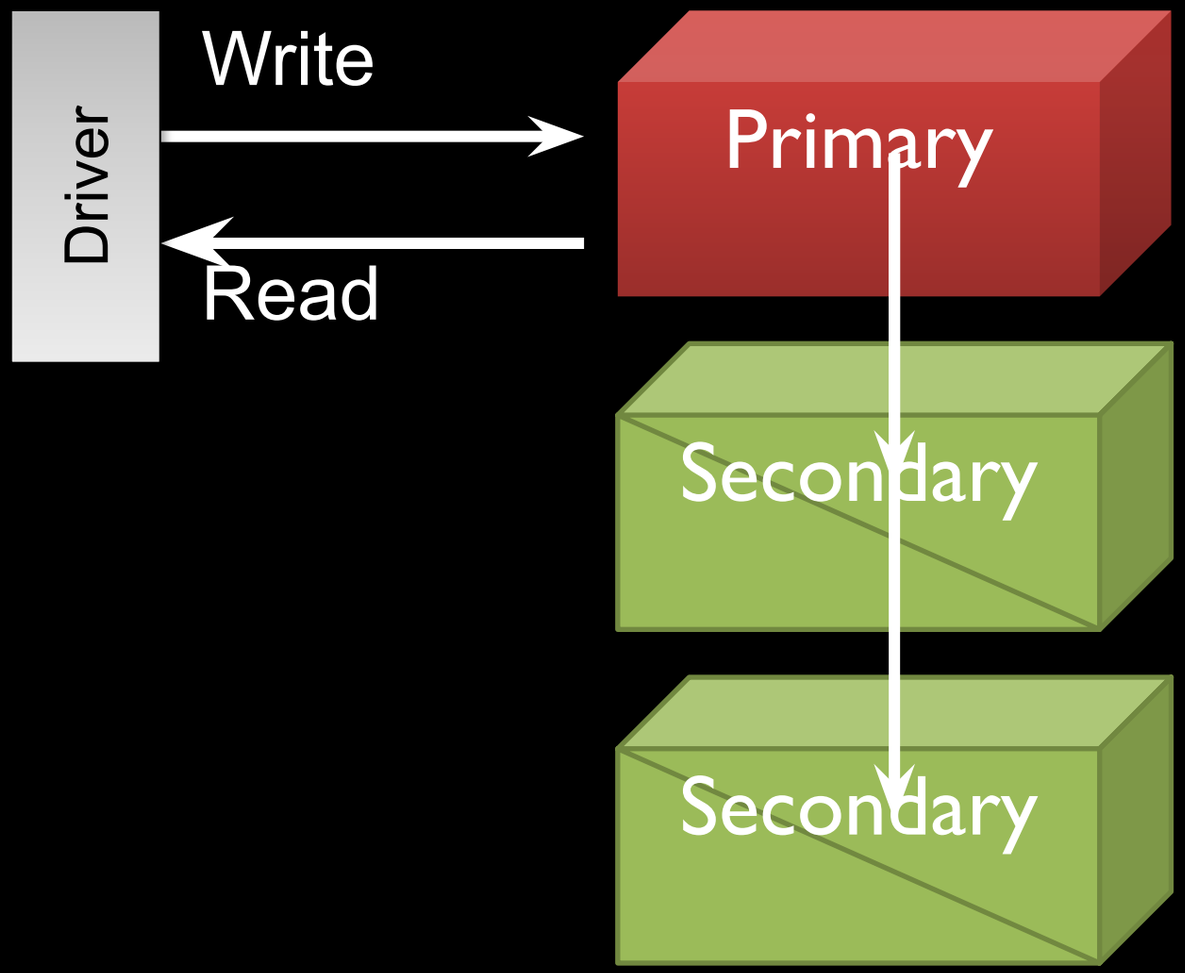
Automatic
Leader Election

Replica Sets

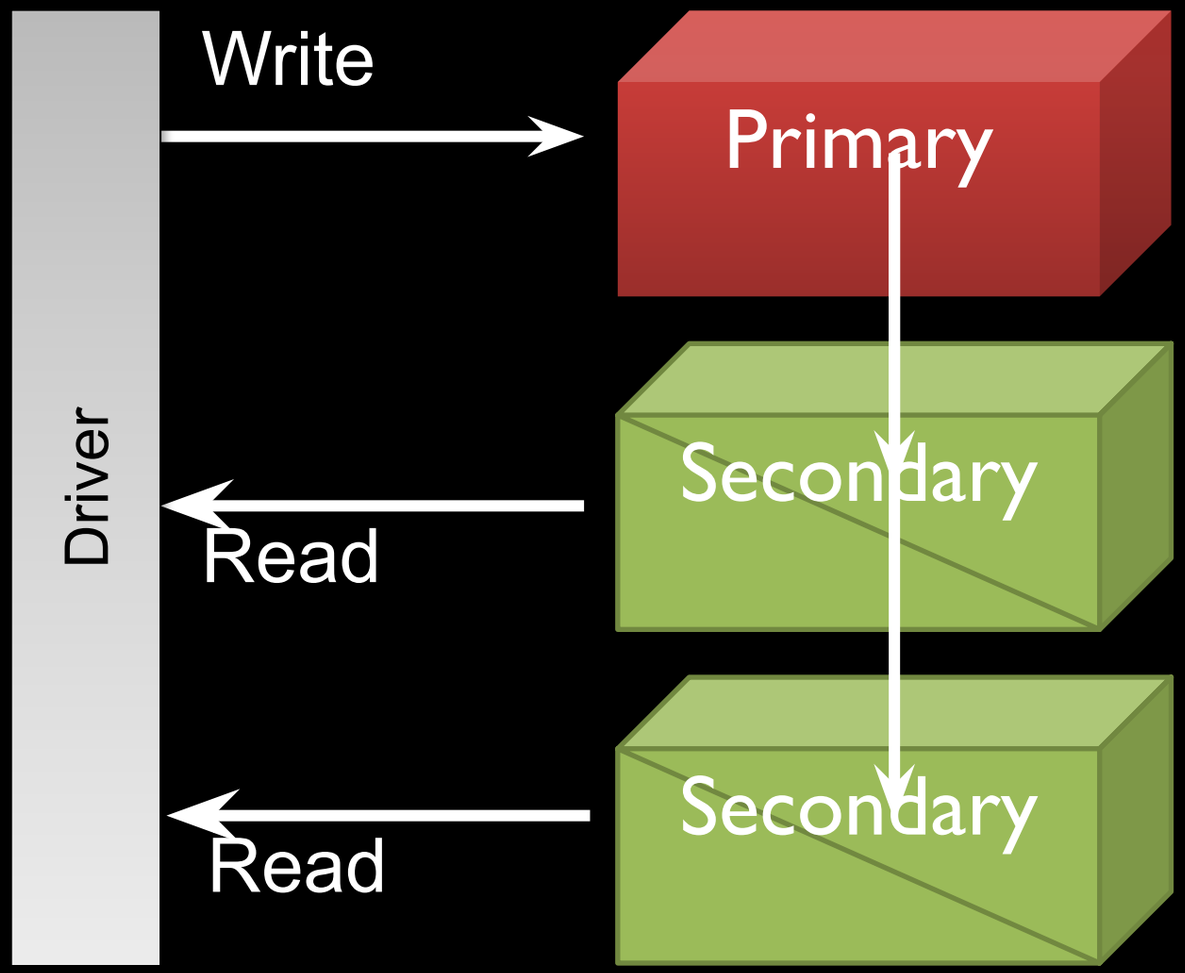


Consistency and durability

Strong Consistency



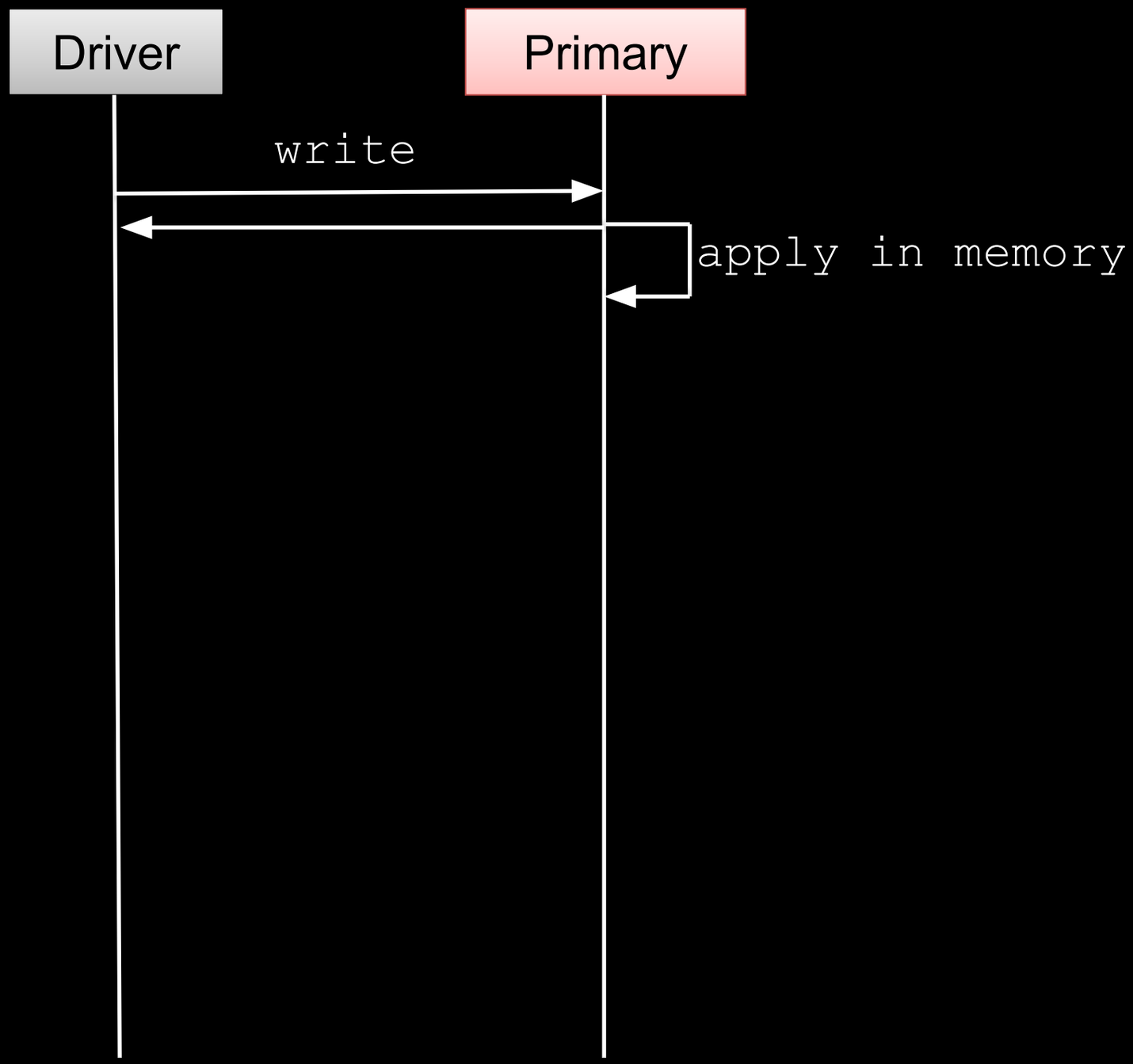
Eventual Consistency



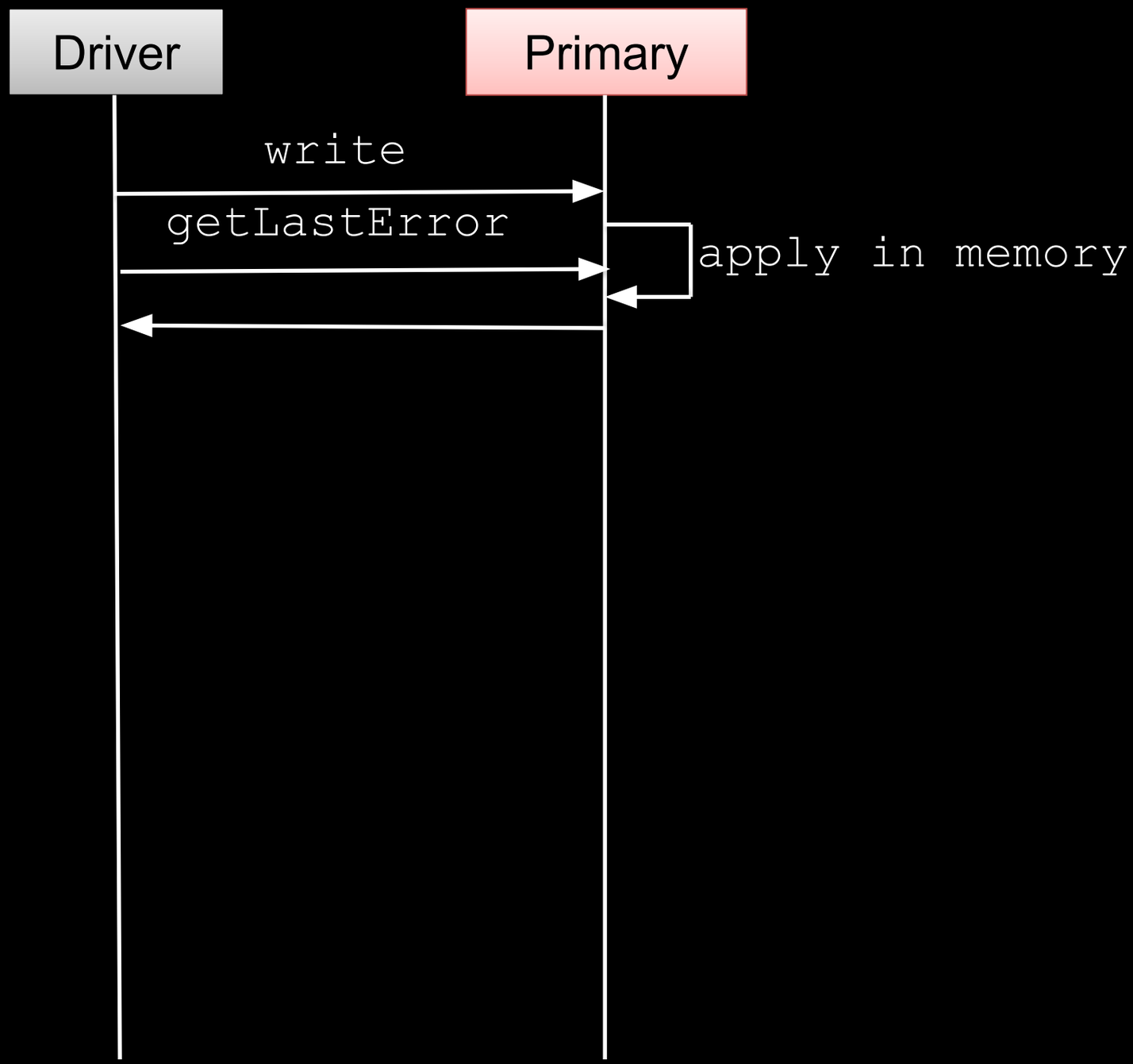
Durability

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

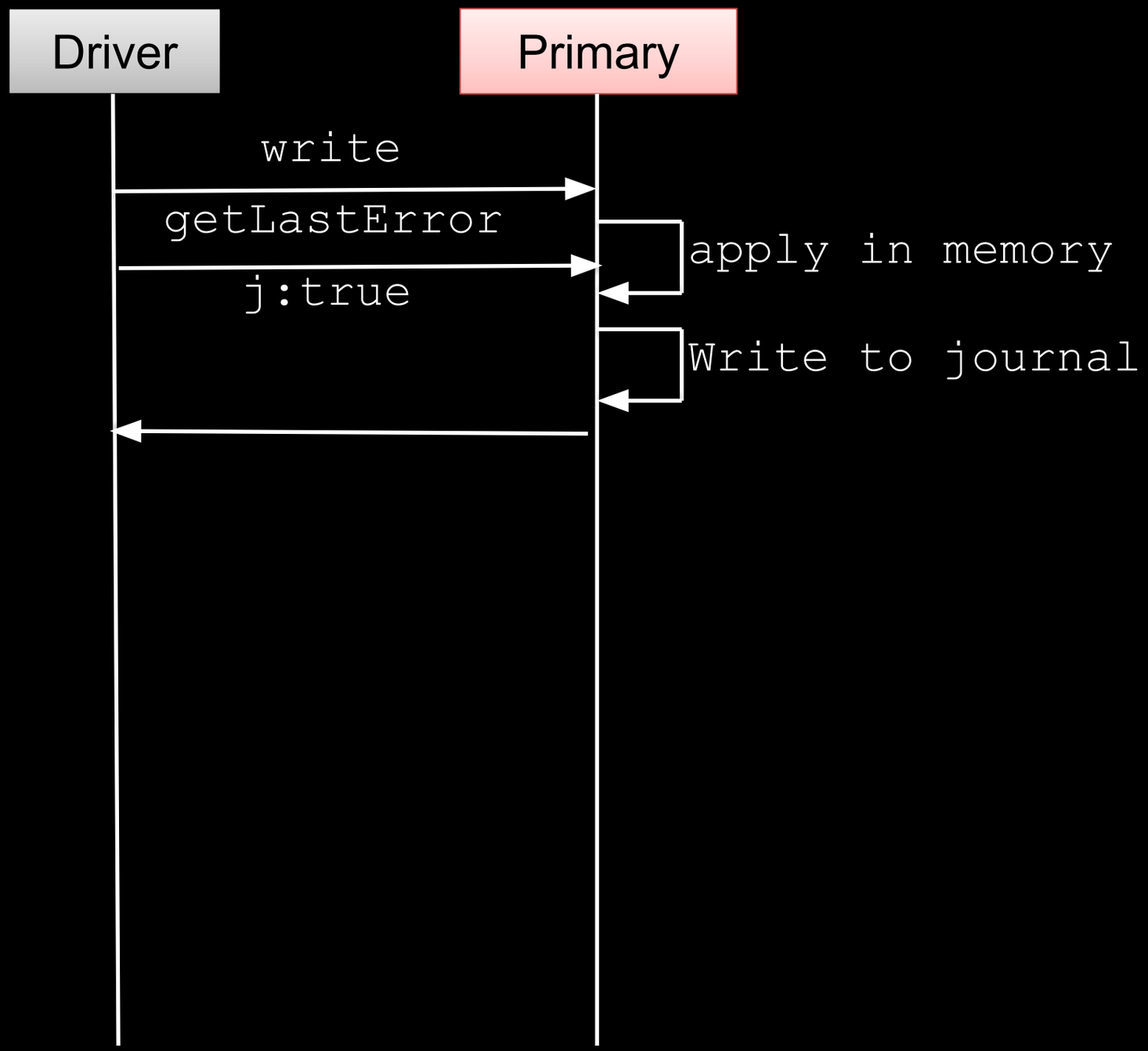
Fire and forget



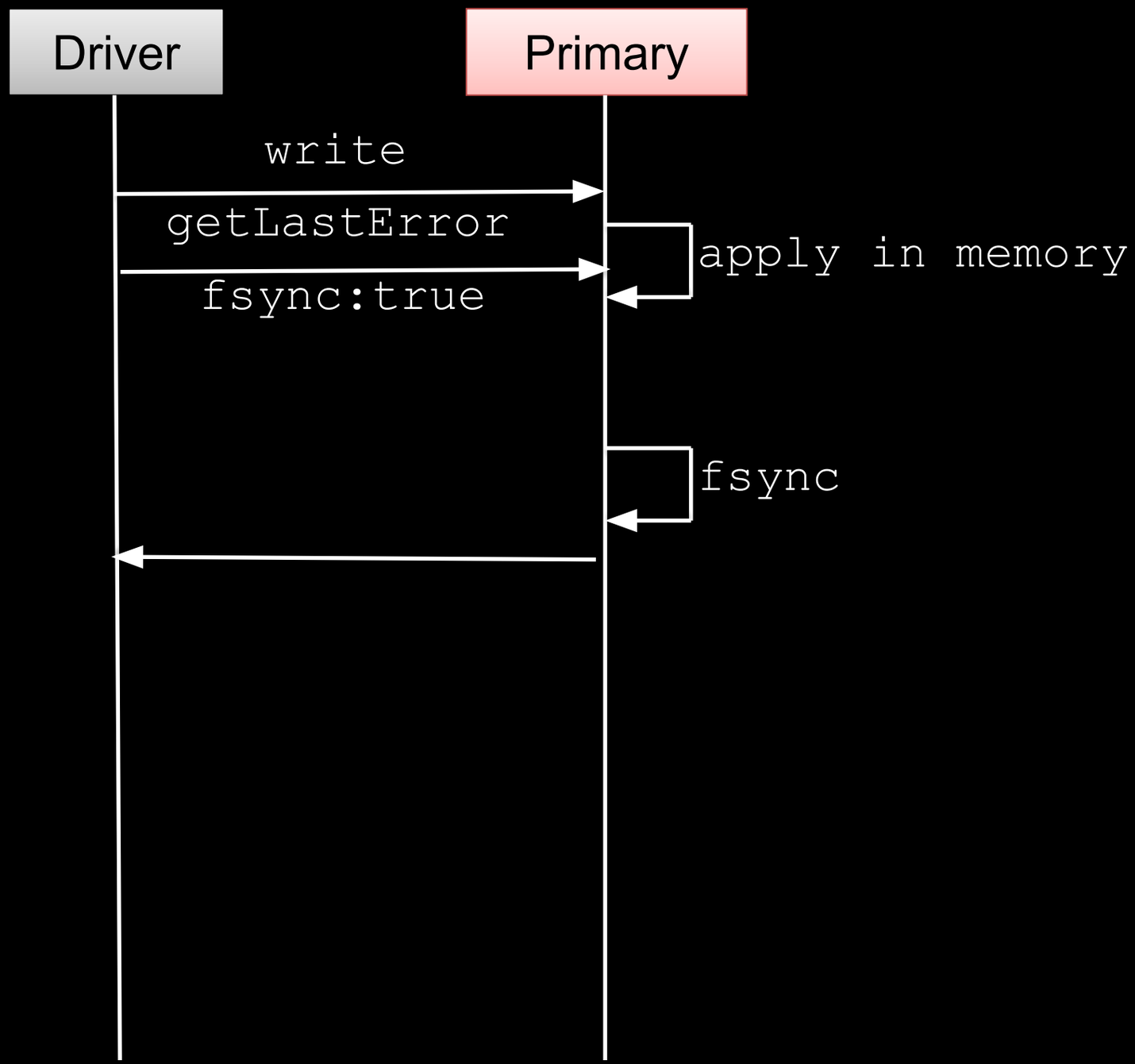
Get last error



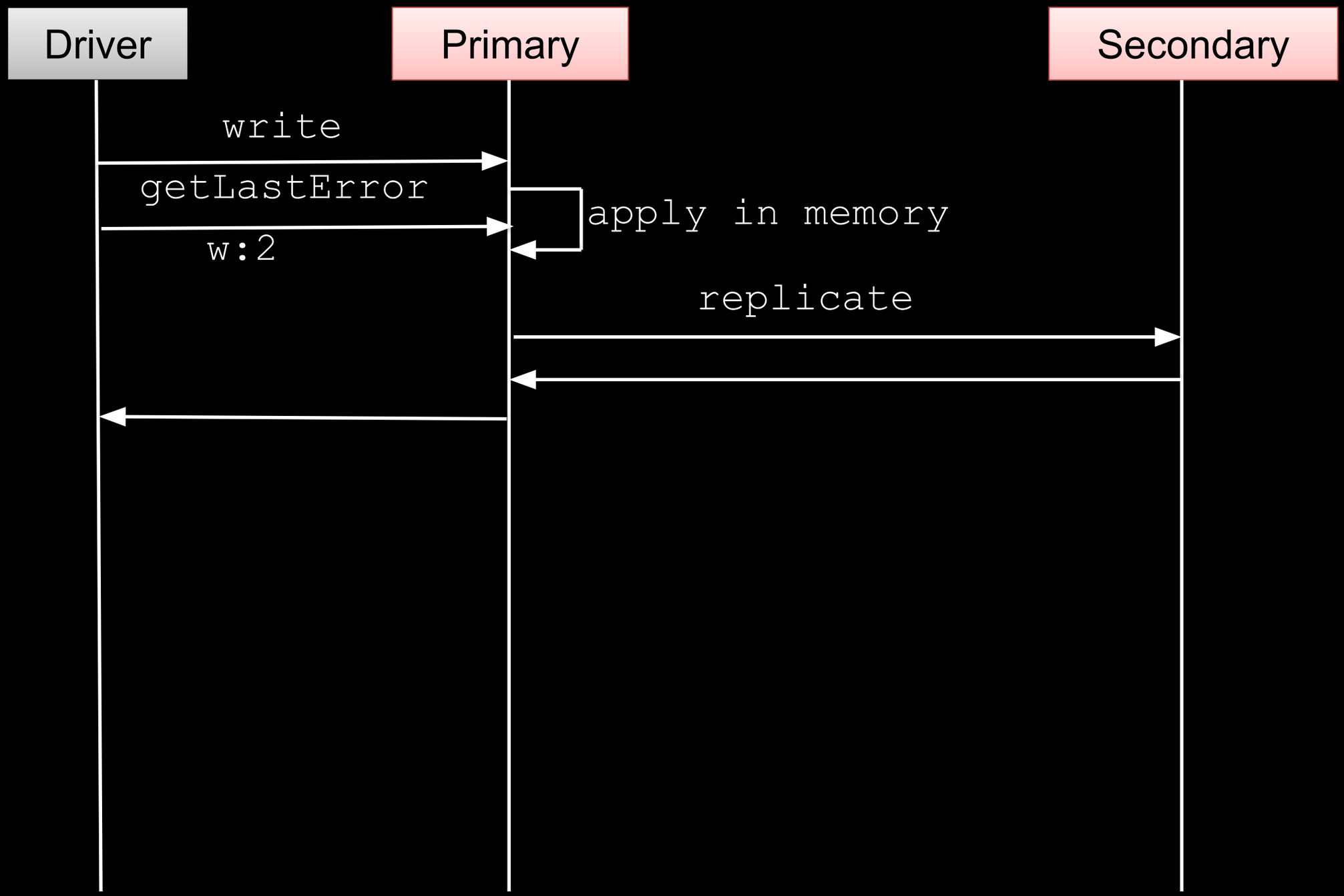
Wait for Journal Sync



Wait for fsync



Wait for replication



Write Concern Options

Value	Meaning
<n:integer>	Replicate to N members of replica set
“majority”	Replicate to a majority of replica set members
<m:modeName>	Use custom error mode name

Tagging

```
{ _id: "someSet",
  members: [
    { _id:0, host:"A", tags: { dc: "ny"}},
    { _id:1, host:"B", tags: { dc: "ny"}},
    { _id:2, host:"C", tags: { dc: "sf"}},
    { _id:3, host:"D", tags: { dc: "sf"}},
    { _id:4, host:"E", tags: { dc: "cloud"}},
  settings: {
      getLastErrorModes: {
          veryImportant: { dc: 3 },
          sortOfImportant: { dc: 2 }
      }
  }
}
```

These are the modes you can use in write concern

Replica set options

Priorities

- Between 0..1000
- Highest member that is up to date wins
  - Up to date == within 10 seconds of primary
- If a higher priority member catches up, it will force election and win

Slave Delay

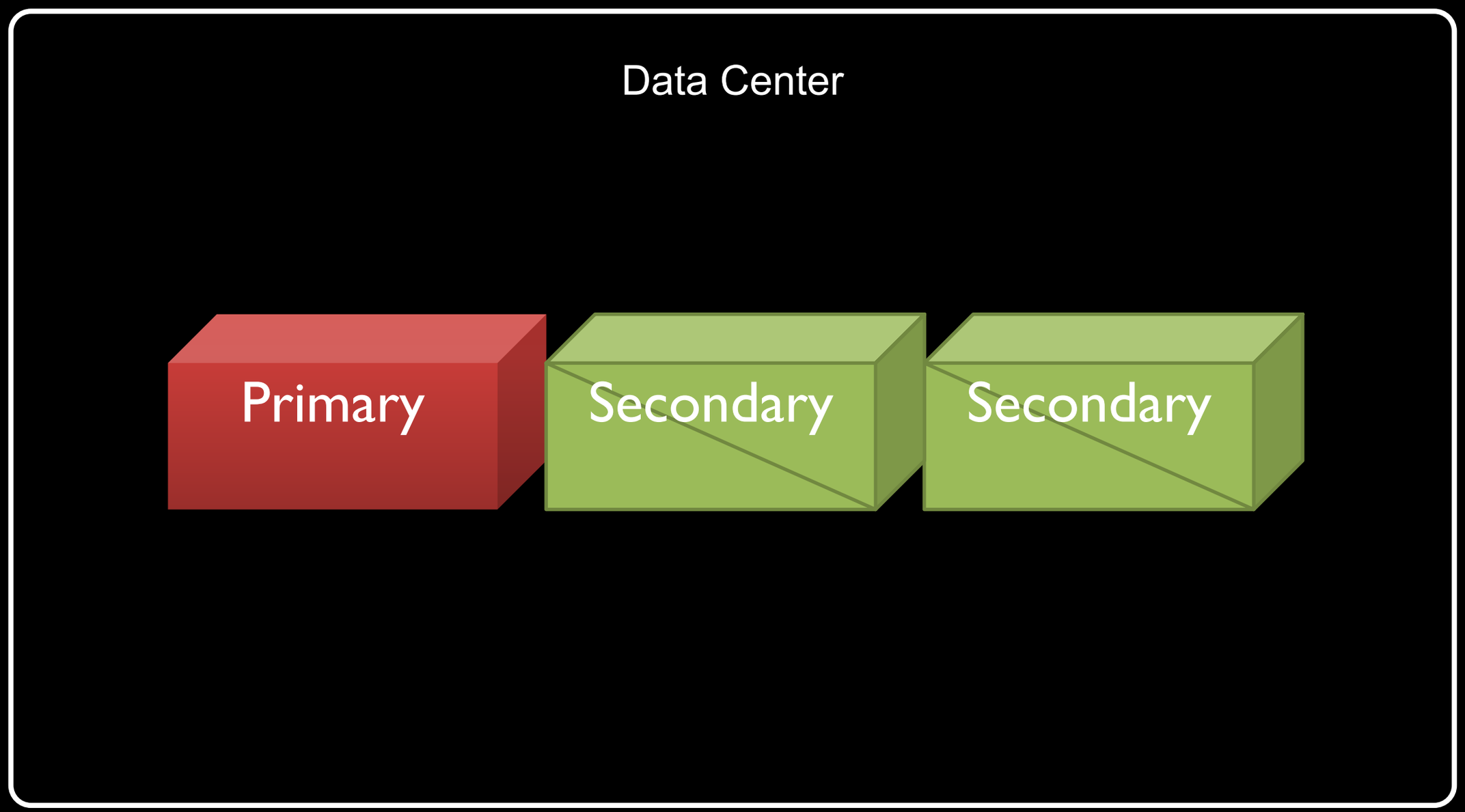
- Lags behind master by configurable time delay
- Automatically hidden from clients
- Protects against operator errors
  - Accidentally delete database
  - Application corrupts data

Arbiters

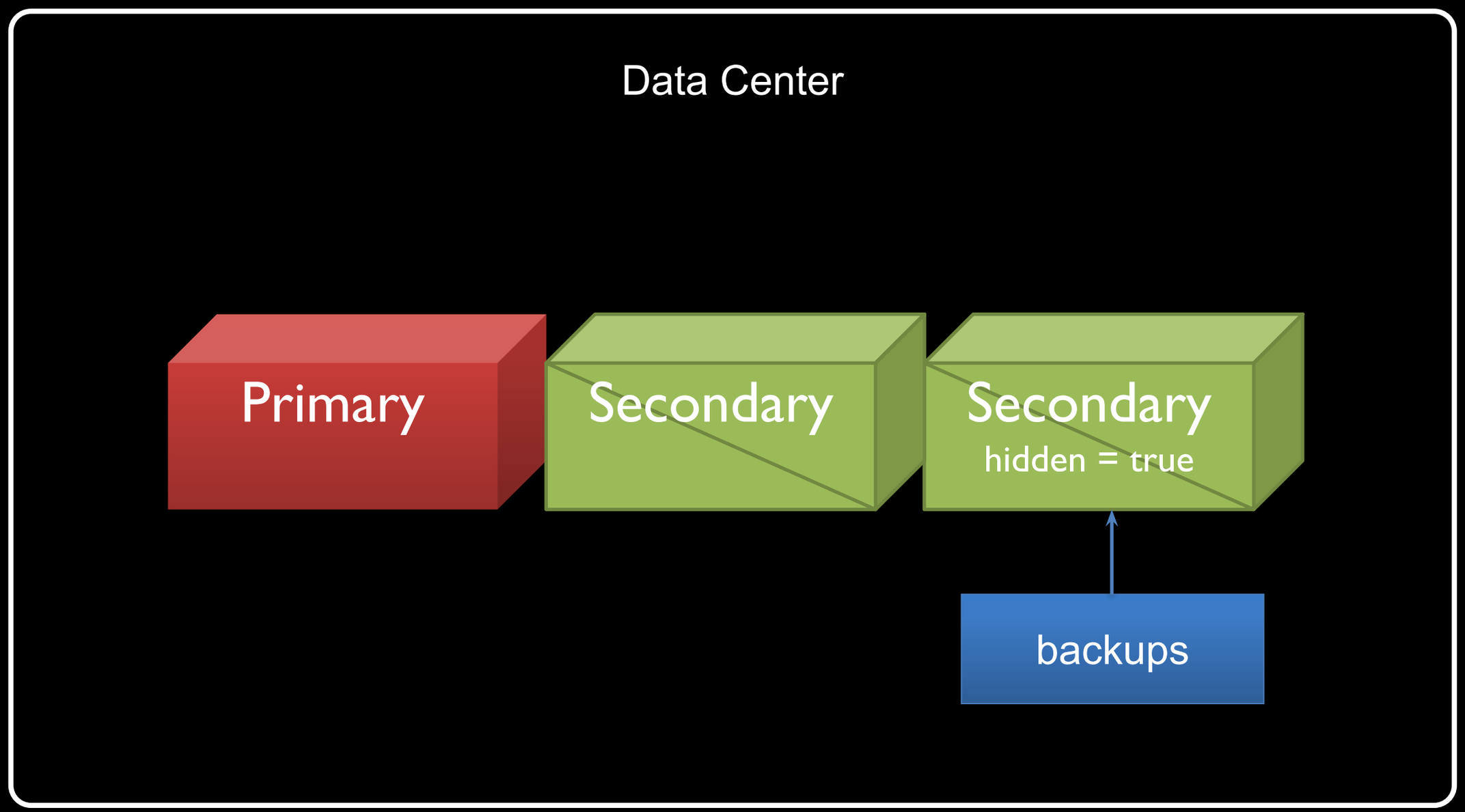
- Vote in elections
- Don’t store a copy of data
- Use as tie breaker

Common deployment scenarios

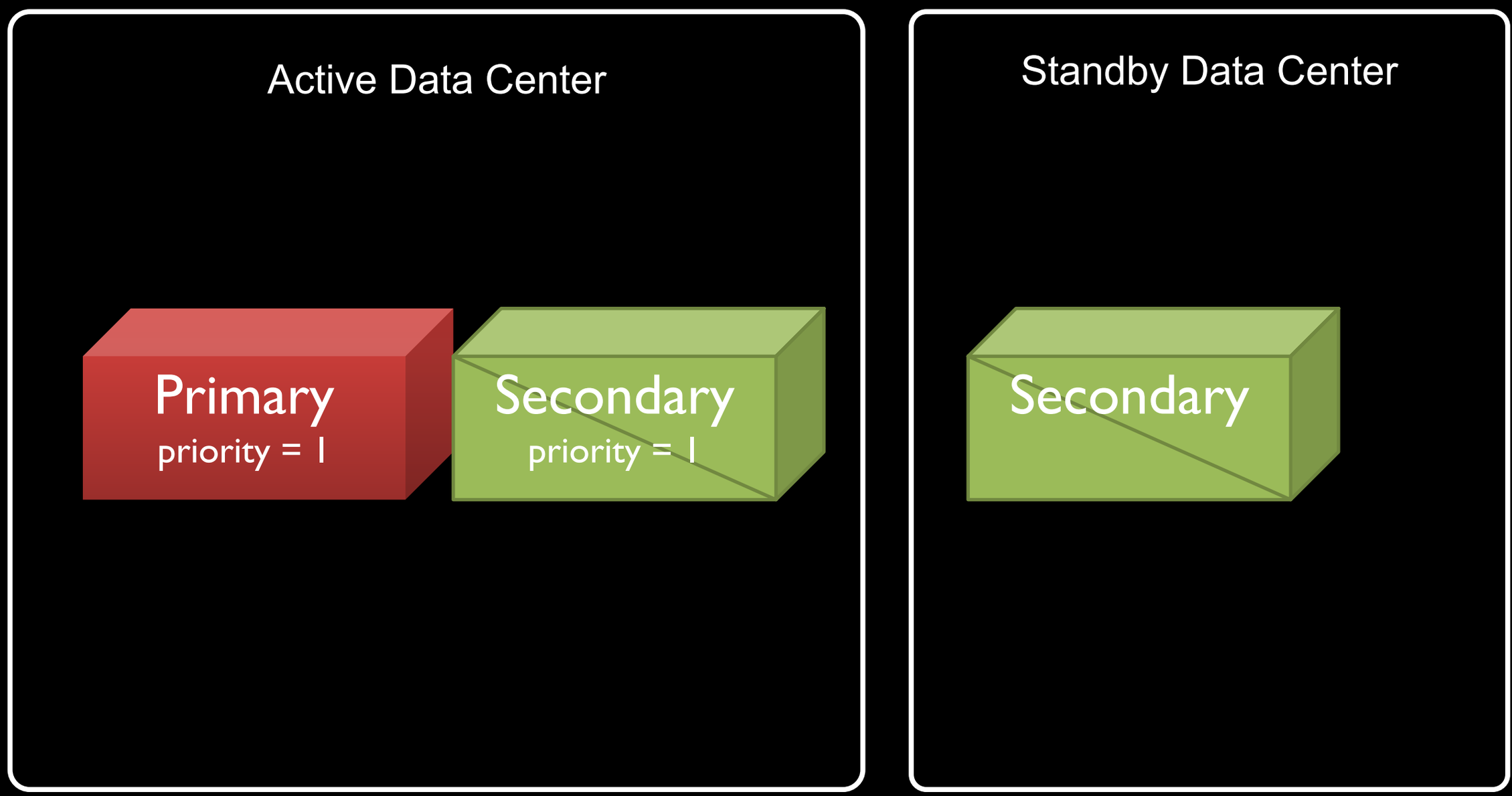
Typical



Backup Node


Data Center
Primary
Secondary
Secondary
hidden = true
backups

Disaster Recovery



Multi Data Center

West Coast DC

Secondary

Central DC

Primary
priority = 1

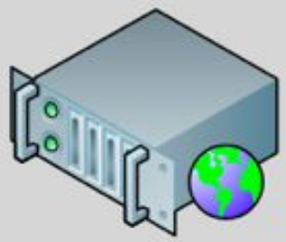
East Coast DC

Secondary

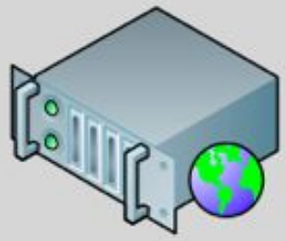
Application Tier

Data Tier

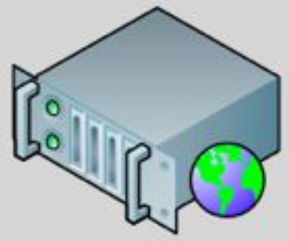
east data center



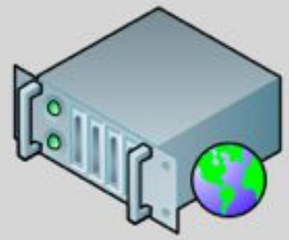
a1.acme.com:8080
s1.acme.com:27017



a2.acme.com:8080
s2.acme.com:27017



a3.acme.com:8080
s3.acme.com:27017



a4.acme.com:8080
s4.acme.com:27017

east data center

Replica Set A `rs_a`

e1.acme.com:27018
c1.acme.com:27019

e2.acme.com:27018

w1.acme.com:27018
c3.acme.com:27019

Replica Set B `rs_b`

e3.acme.com:27018

e4.acme.com:27018
c2.acme.com:27019

w2.acme.com:27018

Replica Set C `rs_c`

e5.acme.com:27018

e6.acme.com:27018

w3.acme.com:27018

west data center

Quick Release History

- 1.0 - August 2009: bson, BTree range query optimization
- 1.2 - December 2009: map-reduce
- 1.4 - March 2010: background indexing, geo indexes
- 1.6 - August 2010: sharding, replica sets
- 1.8 - March 2011: journaling, sparse and covered indexes
- 1.10 = 2.0 - Sep 2011: cumulative performance improvements

Roadmap

- v2.2 Projected 2012 Q1
  - Concurrency: yielding and db or collection-level locking
  - Improved free list implementation
  - New aggregation framework
  - TTL Collections
- Hash shard key

Roadmap: short List

- Full text Search
- Online compaction
- Internal compression
- Read tagging
- XML / JSON transcoder

Vote:
http://jira.mongodb.org/

Data Models

Relational (Comparison)
Key-value
Column-oriented/ Tabular
Document oriented

CA

RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica

AP

Each client can always read and write

Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak

CP

All clients always have the same view of the data

BigTable
Hypertable
HBase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis

The system works well despite physical network partitions

Thank you