MongoDB Essentials

Derick Rethans & Jeremy Mikola

derick@mongodb.com & jmikola@mongodb.com
—@derickr & @jmikola

https://joind.in/13776

About Me

Derick Rethans

- Dutchman living in London
- One of the PHP MongoDB driver maintainers
- Author of the PHP debugger tool Xdebug, PHP's Date/Time/Timezone support, and a bunch of other PHP extensions
- I ♥ maps

About Him

Jeremy Mikola

- Lives in Hoboken, NJ (which has a Dutch name)
- Also maintains the PHP MongoDB driver
- Contributor on React PHP, Doctrine MongoDB, and a few other things
- Sausage aficionado

Today's Topics

- Introduction into NoSQL
- Introduction into MongoDB with PHP
- Schema Design
- Indexes
- Introduction to Replication
- Introduction to Sharding
- Questions and Answers

Introduction

The Relational Model

Edgar Codd in 1969

- Row
- Column
- Table
- View (Result)

Normalisation

- **1NF**: A relation is in first normal form if the domain of each attribute contains only atomic values, and the value of each attribute contains only a single value from that domain.
- **2NF**: No non-prime attribute in the table is functionally dependent on a proper subset of any candidate key
- **3NF**: Every non-prime attribute is non-transitively dependent on every candidate key in the table. The attributes that do not contribute to the description of the primary key are removed from the table. In other words, no transitive dependency is allowed.

NoSQL

Key/value

Column

Graph

Document

NoSQL: Key/Value

Example implementations:

- Memcache
- Redis

Data model:

```
{ "derick-twitter": "derickr" }
{ "derick-email": "derick@derickrethans.nl" }
{ "jeremy-twitter": "jmikola" }

{ "derick-sites": [ "http://derickrethans.nl", "http://xdebug.org" ] }
```

NoSQL: Column

Example implementations:

- Cassandra
- HBase/Hadoop

Data model:

```
{ twitter: [ { "derick" : "derickr" }, { "jeremy" : "jmikola } ] }
{ email: [ { "derick" : "derick@derickrethans.nl" } ] }
{ sites: [ { "derick" : [ "http://derickrethans.nl", "http://xdebug.org" ] } ] }
```

NoSQL: Graph

Example implementations:

- neo4j

Data model:

```
[ "derick", "follows", "jeremy" ]
[ "jeremy", "follows", "derick" ]
[ "derick", "follows", "Queen_UK" ]
[ "derick", "twitters_with", "derickr" ]
[ "jeremy", "twitters_with", "jmikola" ]
```

NoSQL: Document

Example implementations:

- CouchDB
- MongoDB

Data model:

```
{
        _id: "derick",
        twitter: "derickr",
        email: "derick@derickrethans.nl",
        sites: [ "http://derickrethans.nl", "http://xdebug.org" ]
}
{
        _id: "jeremy",
        twitter: "jmikola",
}
```

CAP

Eric Brewer in 2002

- **C**onsistency:
  A read sees all previously completed writes
  (Not the same consistency as in ACID)
- **A**vailability:
  Guarantees that every request receives a response about whether it was successful or failed
- **P**artition Tolerance:
  Guaranteed properties are maintained even when network failures prevent some machines from communicating with others.

Terminology

- **Document**: the data (row)
- **Collection**: contains documents (table, view)
- **Index**
- **Embedded Document** (~join)
- **Sharding** (partitioning)

Documents

- Can have embedded documents
- Are **schemaless**

Document with embedded documents:

```
{
  "_id" : "derickr",
  "name" : "Derick Rethans",
  "talks" : [
    { "title" : "Profiling PHP Applications",
      "url" : "http://derickrethans.nl/talks/profiling-phptour.pdf",
    },
    { "title" : "Xdebug",
      "url" : "http://derickrethans.nl/talks/xdebug-phpbcn11.pdf",
    }
  ]
}
```

Connecting to MongoDB

No databases or collections have to do be explicitly created:

```
<?php
$m = new MongoClient();
$database = $m->selectDB( 'demo' );
$collection = $database->selectCollection( 'testCollection' );
// or
$collection = $m->selectCollection( 'demo', 'testCollection' );
?>
```

Different connection strings:

- `new MongoClient("mongodb://localhost");`
- `new MongoClient("mongodb://localhost:29000");`
- `new MongoClient("mongodb://mongo.example.com");`

Inserting a document (JavaScript)

```
doc = {
  "_id" : "derickr",
  "name" : "Derick Rethans",
  "talks" : [
    {
      "title" : "Profiling PHP Applications",
      "url" : "http://derickrethans.nl/talks/profiling-phptour.pdf",
    },
    {
      "title" : "Xdebug",
      "url" : "http://derickrethans.nl/talks/xdebug-phpbcn11.pdf",
    }
  ]
};

db.collectionName.insert( doc );
```

Inserting a document

```php
<?php
$document = [
  "_id" => "derickr",
  "name" => "Derick Rethans",
  "talks" => [
    [
      "title" => "Profiling PHP Applications",
      "url" => "http://derickrethans.nl/talks/profiling-phptour.pdf",
    ],
    [
      "title" => "Xdebug",
      "url" => "http://derickrethans.nl/talks/xdebug-phpbcn11.pdf",
    ]
  ]
];

$m = new MongoClient();
$c = $m->demo->talks; $c->drop();
var_dump( $c->insert( $document ) );
?>
```

Result:

```
array (size=4)
  'ok' => float 1
  'n' => int 0
  'err' => null
  'errmsg' => null
```

IDs

- Every document needs a unique, immutable ID
- ID consists of: 4-byte timestamp, 3-byte machine identifier, 2-byte PID and 3-byte counter:
  `507e6384 44670a 3e6e 000000`

```
<?php
$m = new MongoClient();
$c = $m->demo->articles;

$c->insert( [ 'name' => 'Derick', '_id' => 'derickr' ] );

$d = [ 'name' => 'Derick' ];
$c->insert( $d );
var_dump( $d );
?>
```

Exercise

Exercise: Import Beer

- `git clone https://github.com/derickr/mongo-tutorial.git`
- We will use beer ratings and types for examples
- `untappd.json` has one JSON document per line
- Write a script to import each beer into the **tutorial** database and the **beer** collection
- Verify the result on the mongo shell:

```
$ mongo tutorial
> db.beer.count();
```

Querying (one document)

```
<?php
$m = new MongoClient();
// First "found" item:
$record = $m->demo->talks->findOne();

// Search on the _id field being 'derickr'
$record = $m->demo->talks->findOne( [ '_id' => 'derickr' ] );

// Search on the 'talks' embedded document's 'title' field
$record = $m->demo->talks->findOne(
  [ 'talks.title' => 'Xdebug' ]
);
?>
```

Querying (with projection)

```
<?php
$m = new MongoClient();

// Find with 'projection'
$record = $m->demo->talks->findOne(
  [ 'talks.title' => 'Xdebug' ],
  [ 'name' => 1, 'talks.url' => 1 ]
);
?>
```

Advanced querying operators

Operators start with a '$', so use single quotes!

Array:

- $in (value needs to be one of the elements of the given array)
- $nin (not in)
- $all (value needs to match all elements of the given array)
- $size (exact size of array)
- $elemMatch (element in an array matches the specified match expression)

Querying and looping

Finding multiple documents is done with the `find()`, and returns an iterator in the form of a `MongoCursor` object.

```
<?php
$m = new MongoClient();
$c = $m->demo->talks;

$cursor = $c->find( [ 'talks.title' => [ '$exists' => true ] ] );

foreach ( $cursor as $r )
{
        echo $r['_id'], "\n";
}
?>
```

Result:

```
derickr
```

Cursor methods

Before starting iteration, several methods may be called to configure the query.

```
<?php
$m = new MongoClient();
$c = $m->demo->cities;

$cursor = $c->find( [ 'country_code' => 'RU' ] )
            ->sort( [ 'name' => 1 ] )
            ->limit( 5 )->skip( 25 );

foreach ( $cursor as $r ) {
    var_dump( $r['name'] );
}
?>
```

Result:

```
string 'Al’met’yevsk' (length=16)
string 'Amursk' (length=6)
string 'Anapa' (length=5)
string 'Angarsk' (length=7)
```

Exercise

Exercise: querying

- Outputs the **created at** date of the first "check-in" for **USA Hells**
- List all **beer names** (with rating and date), where the beer has a rating of **4.5 or higher**, ordered first by rating (highest first), and second by date (earliest first).

MongoDB supports many types

- null
- boolean
- integer (32 and 64-bit, MongoInt32, MongoInt64)
- double
- string (UTF-8)
- array
- document (associative array in PHP)
- ObjectIDs (MongoId)
- UTC date (MongoDate)

MongoDB types and PHP

- Unlike PHP, MongoDB does not type juggle
- There is a date type, but it is only a Unix timestamp
- When retrieving data, documents will be returned as an (associative) array, but never as an object

Exercise

Exercise: import with types

- Import `beer_abv`, `beer_ibu`, `venue_lat`, `venue_lng` and `rating_score` as float.
- Import `created_at` as `MongoDate` object
- Fix the finding script

Updating documents

Update only modifies the **first** document it finds by default.

You can set an option to modify **all** matched documents

```
<?php
$m = new MongoClient;
$c = $m->demo->products;
$c->drop();

$c->insert( [ '_id' => 'blue-elephpant',  'generation' => 1, 'price' => 7.5 ] );
$c->insert( [ '_id' => 'pink-elephpant',  'generation' => 2, 'price' => 8.0 ] );
$c->insert( [ '_id' => 'green-elephpant', 'generation' => 3, 'price' => 7.5 ] );

$c->update(
    [ 'generation' => [ '$lte' => 2 ] ],  // criteria
    [ '$inc' => [ 'price' => -2.0 ] ],    // update spec
    [ 'multiple' => true ]                // options: multiple
);
?>
```

Deleting documents

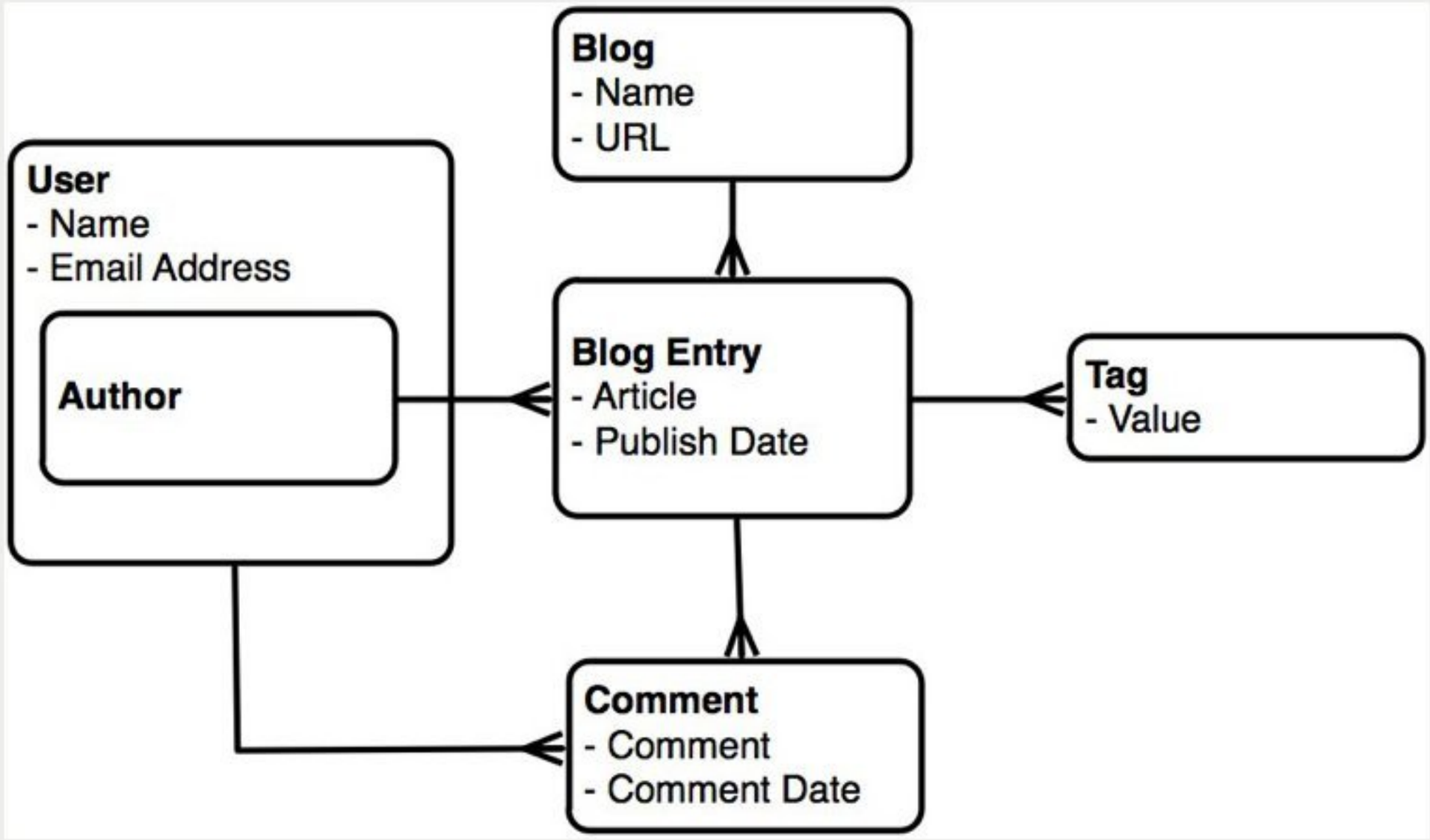
Unlike `update`, `remove` deletes **all** matched documents by default.

You can set an option to remove only the **first** matched document

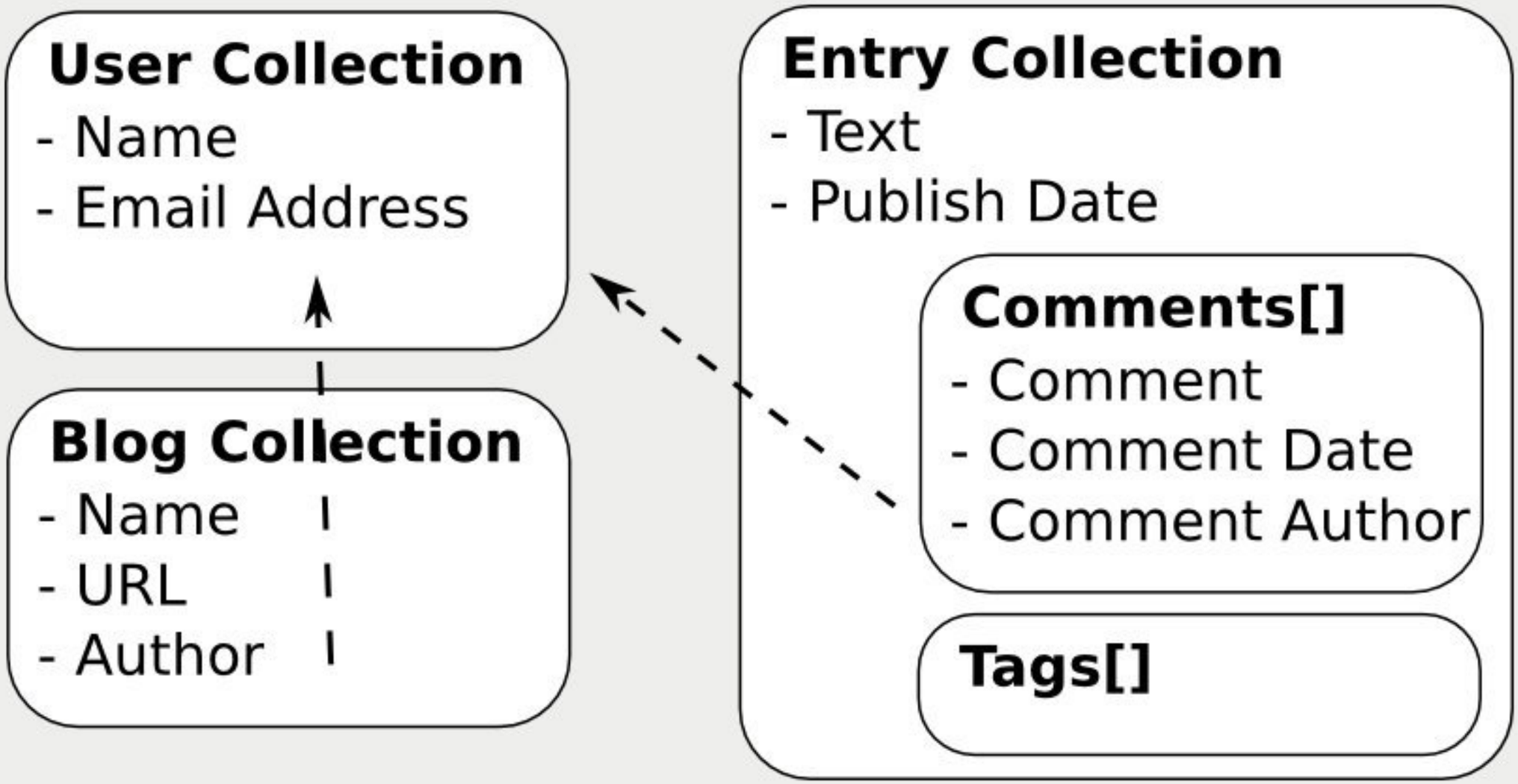
```
<?php
$m = new MongoClient;
$c = $m->demo->products;
$c->drop();

$c->insert( [ '_id' => 'blue-elephpant',  'generation' => 1, 'price' => 7.5 ] );
$c->insert( [ '_id' => 'pink-elephpant',  'generation' => 2, 'price' => 8.0 ] );
$c->insert( [ '_id' => 'green-elephpant', 'generation' => 3, 'price' => 7.5 ] );

$c->remove(
    [ 'price' => [ '$lt' => 8.0 ] ],  // criteria
    [ 'justOne' => true ]             // options: justOne
);
?>
```

Schema Design

Blog in a relational model



Blog in a document model



Schema considerations

Access Patterns

- Read / Write Ratio
- Types of queries and updates
- Data life-cycle

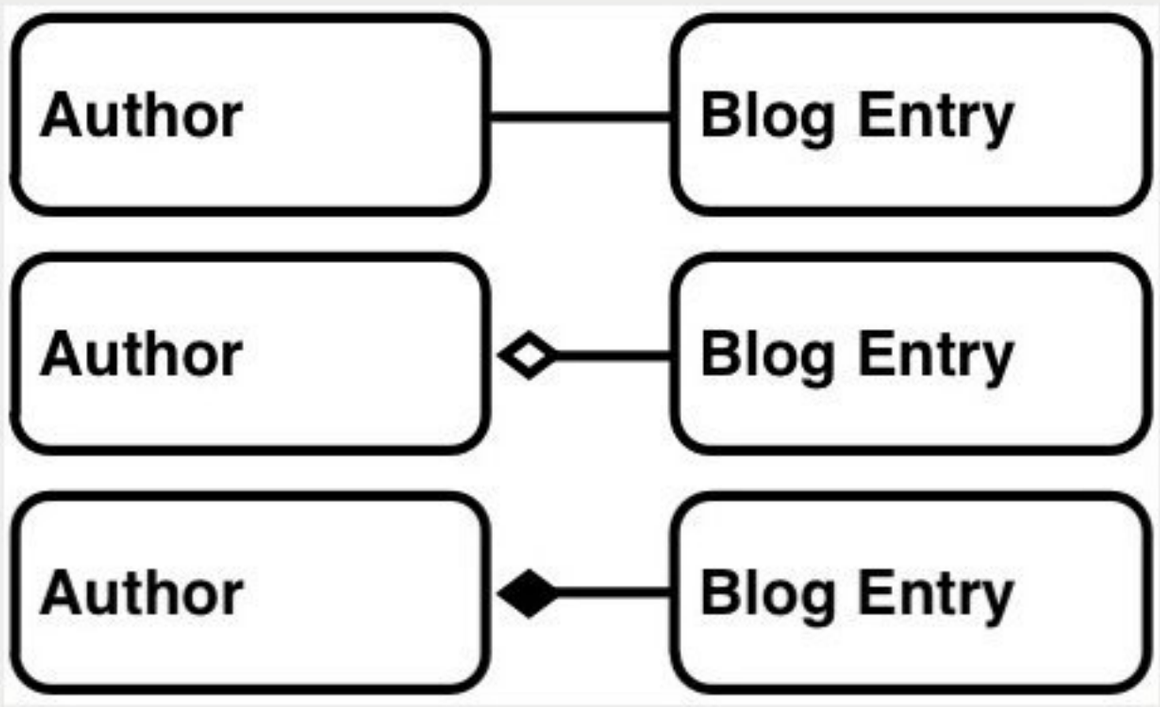
Considerations

- Only single-document writes are atomic
- No joins (reference vs. embed)
- Growing documents may require relocation

One to Many (1:n)

One to Many relationships can specify:

- Degree of association between objects
- Containment



Embedding Comments and Views

```
{
  name: "Derick's MongoDB Tour Wrap-up",
  date: "2012-10-01",
  comments: [
    { name: "Adam", text: "It was great having you in Sou..." },
    { name: "Jake", text: "I don't think you can ever re..." },
  ],
  views: [
    { time: 1350297458, ip: "192.168.42.101" },
    { time: 1350297729, ip: "192.168.42.103" },
    { time: 1350298912, ip: "192.168.42.102" },
  ]
}
{
  name: "What is PHP doing?",
  date: "2012-07-13",
  comments: [
    { name: "Chris", text: "The .gdbinit trick was somethi..." },
  ],
  views: [
    { time: 1350297458, ip: "192.168.42.101" },
  ]
}
```

Linking Articles and Views

Article collection:

```
{ _id: 4,
  name: "Derick's MongoDB Tour Wrap-up",
  date: "2012-10-01",
  comments: [
    { name: "Adam", text: "It was great having you in Sou..." },
    { name: "Jake", text: "I don't think you can ever re..." },
  ],
}
{ _id: 7,
  name: "What is PHP doing?",
  date: "2012-07-13",
  comments: [
    { name: "Chris", text: "The .gdbinit trick was somethi..." },
  ]
}
```

Views collection:

```
{ article_id: 4, time: 1350297458, ip: "192.168.42.101" }
{ article_id: 4, time: 1350298912, ip: "192.168.42.102" }
{ article_id: 7, time: 1350297458, ip: "192.168.42.101" }
```

Tips and Hints: document keys

Don't do:

```
temperature: {
  _id: 42,
  points: [
    { 1332942067: 17.3 },
    { 1332942118: 17.5 },
  ]
}
```

Instead, do:

```
temperature: {
  _id: 42,
  points: [
    { ts: 1332942067, temp: 17.3 },
    { ts: 1332942118, temp: 17.5 },
  ]
}
```

Define and document your keys!

Exercise

Distilleries

A distillery has many whiskies. Design a schema that stores information about whiskies, and about distilleries and the whiskies they make.

Information to store is:

- Distillery: country, region (optional), name, opening year
- Whisky: name, age, strength, vintage (optional)
- The relationship between them

Distilleries: result

Collection: distillery

```
{
        name: "Glenfiddich",
        country: "Scotland",
        region: "Speyside",
        opening_year: 1887
}
```

Collection: whisky

```
{
        name: "Glenfiddich 15",
        age: 15,
        strength: 50.3,
        distillery: "Glenfiddich"
}
```

Embedding versus Linking

Embedding

- Simple data structure
- How often do you update?
- Will the document grow and grow?

Linking

- More complex data structure
- Unlimited data size
- More, smaller documents

Exercise

Tasting notes

Tasting notes are like a review of whiskies. Each user can have a note for each whisky. Design a schema that allows for:

- User profiles: name, country
- Whiskies: name, age, etc (ignore the distillery bit)
- Tasting notes: date, text, likes **(just a counter)**, and rating

Tasting notes

Collection: whisky

```
{
    name: "Glenfiddich 15",
    age: 15,
    strength: 50.3,
    tasting_notes: [
        {
            user_name: "Derick",
            date: ISODate("2013-10-02"),
            note: "Caramel and honey, fruity with some hints of earth",
            likes: 17,
            rating: 3,
        }
    ]
}
```

Collection: user

```
{
    name: "Derick",
    country: "England",
}
```

Adding a review

```
{
    name: "Glenfiddich 15",
    notes: [
        {
            user_name: "Derick",
            date: ISODate("2013-10-02"),
            note: "Caramel and honey, fruity with some hints of earth",
            likes: 17,
            rating: 3,
        }
    ]
}
```

```
<?php
$m = new MongoClient;
$c = $m->demo->whisky;

$newReview = [
    user_name => "Michael",
    date => new MongoDate(strtotime("2013-10-01")),
    note => "No peat, but fruity",
    rating => 1,
];

$c->update(
    [ 'name' => 'Glenfiddich 15' ],
    [ '$push => [ 'notes' => $newReview ] ]
);
?>
```

Recording a like

```
{
    name: "Glenfiddich 15",
    notes: [
        {
            user_name: "Derick",
            date: ISODate("2013-10-02"),
            note: "Caramel and honey, fruity with some hints of earth",
            likes: 17,
            rating: 3,
        }
    ]
}
```

```
<?php
$m = new MongoClient;
$c = $m->demo->whisky;

$c->update(
    [ 'name' => 'Glenfiddich 15', 'notes.user_name' => 'Derick' ],
    [ '$inc => [ 'notes.$.likes' => 1 ] ]
);
?>
```

Commands

- Are run against a database
- Some (aggregation) return a cursor
- Others return one document
- Some common commands have helpers in the drivers and shell, most commands don't have a helper

Aggregation

Aggregation Framework

- Declared in BSON, executes in C++
- Flexible, functional, and **simple**
  - Operation pipeline
  - Computational expressions
- Plays nice with sharding

Pipeline

- Process a stream of documents
  - Original input is a collection
  - Final output is a cursor (or inline result)
- Series of operators
  - Filter or transform data
  - Input/output chain

```
ps ax | grep mongod | head -n 1
```

Pipeline Operators

- `$match`
- `$geoNear`
- `$project`
- `$group`
- `$unwind`

- `$sort`
- `$limit`
- `$skip`

Our Example data


```
{
        "_id" : ObjectId("5400a24f44670a01098b491d"),
        "beer_name" : "Aurora Australis",
        "brewery_name" : "Bridge Road Brewers",
        "beer_type" : "Belgian Quad",
        "beer_abv" : 11,
        "beer_ibu" : 30,
        "comment" : "Red wine flavours. Brewed in Australia but packaged in Norway.",
        "venue_name" : "Olympen",
        "venue_city" : "Oslo",
        "venue_state" : "Oslo",
        "venue_lat" : 59.9121,
        "venue_lng" : 10.7643,
        "rating_score" : 4.5,
        "created_at" : ISODate("2014-07-06T20:38:17Z"),
        "checkin_url" : "https://untappd.com/c/97860912",
        "beer_url" : "https://untappd.com/beer/300283",
        "brewery_url" : "https://untappd.com/brewery/3174",
        "brewery_country" : "Australia"
}
```

$match

- Filter documents
- Uses existing query syntax
- No geospatial operations or `$where`

Matching Field Values

```
{
    "beer_name" : "Gentleman's Wit",
    "brewery_name" : "Camden Town Brewery",
    "beer_type" : "Witbier"
}
{
    "beer_name" : "Old Rascal",
    "brewery_name" : "Thatchers",
    "beer_type" : "Cider"
}
{
    "beer_name" : "Byron Lager",
    "brewery_name" : "Camden Town Brewery",
    "beer_type" : "Euro Lager"
}
```

→

```
{
  $match: {
    "beer_type": "Cider"
  }
}
```

↓

```
{
    "beer_name" : "Old Rascal",
    "brewery_name" : "Thatchers",
    "beer_type" : "Cider"
}
```

Matching with Query Operators

```
{
    "beer_name" : "Organic Cider",
    "brewery_name" : "Dunkertons",
    "beer_abv" : 6.8
}
{
    "beer_name" : "Belgian White",
    "brewery_name" : "Blue Moon",
    "beer_abv" : 5.4
}
{
    "beer_name" : "Cloudy Cider",
    "brewery_name" : "Addlestones",
    "beer_abv" : 5.2
}
```

→

```
{
  $match: {
    "beer_abv": { $gte: 6 }
  }
}
```

↓

```
{
    "beer_name" : "Organic Cider",
    "brewery_name" : "Dunkertons",
    "beer_abv" : 6.8
}
```

$project

- Reshape documents
- Include, exclude or rename fields
- Inject computed fields
- Create sub-document fields

Including and Excluding Fields

```
{
  "_id" : "honey-daughters",
  "beer_name" : "Midford Cider",
  "brewery_name" : "Honey",
  "beer_type" : "Cider",
  "beer_abv" : 6.5,
  "beer_ibu" : 0,
  "rating_score" : 4,
  "brewery_country" : "England"
}
```

→

```
{
  $project: {
    _id: 0,
    beer_name: 1,
    brewery_name: 1
  }
}
```

↓

```
{
  "beer_name" : "Midford Cider",
  "brewery_name" : "Honey",
}
```

Renaming and Computing Fields

```
{
  "beer_name" : "Crazy Goat",
  "brewery_name" : "Lilleys",
  "beer_type" : "Cider",
  "beer_abv" : 6.8,
  "beer_ibu" : 0,
  "brewery_country" : "England"
}
```

→

```
{ $project: {
  beer: { $concat: [
    '$beer_name',
    ' (',
    '$beer_type'
    ')',
  ] },
  'country': '$brewery_country'
} }
```

↓

```
{
  "_id" : ObjectId("5400a24f44670a01098b4657"),
  "beer" : "Grazy Goat (Cider)",
  "country" : "England"
},
```

Reformat Sub-Documents

```
{
    "beer_name" : "Crazy Goat",
    "brewery_name" : "Lilleys",
    "beer_type" : "Cider",
    "beer_abv" : 6.8,
    "beer_ibu" : 0,
    "brewery_country" : "England"
}
```

→

```
{ $project: {
  name:    '$beer_name',
  brewery: {
    name:    '$brewery_name',
    country: '$brewery_country',
  }
} }
```

↓

```
{
    "name": "Crazy Goat",
    "brewery": {
        "name": "Lilleys",
        "country": "England",
    }
}
```

$group

- Group documents by an ID
  - Field reference, object, constant
- Other output fields are computed
  - `$max`, `$min`, `$avg`, `$sum`
  - `$addToSet`, `$push`
  - `$first`, `$last`
- Processes all data in memory

Summating Fields and Counting

```
{
    "beer_name" : "Floris Framboise",
    "brewery_name" : "Huyghe"
}
{
    "beer_name" : "Arabier",
    "brewery_name" : "De Dolle Brouwers"
}
{
    "beer_name" : "La Trappe Dubbel",
    "brewery_name" : "De Koningshoeven"
}
{
    "beer_name" : "Belgian Winter Beer",
    "brewery_name" : "Huyghe"
}
```

→

```
{ $group: {
  _id: '$brewery_name',
  count: { '$sum': 1 }
} }
```

↓

```
{
  "_id" : "Huyghe",
  "count" : 2
},
{
  "_id" : "De Dolle Brouwers",
  "count" : 1
},
{
  "_id" : "De Koningshoeven",
  "count" : 1
},
```

Collecting Distinct Values

```
{
  "beer_name" : "Reveller",
  "brewery_name" : "Orchard Pig",
  "beer_type" : "Cider"
}
{
  "beer_name" : "Hefe Weisse",
  "brewery_name" : "Paulaner Gruppe",
  "beer_type" : "Hefeweizen"
}
{
  "beer_name" : "Münchener Gold",
  "brewery_name" : "Paulaner Gruppe",
  "beer_type" : "Dortmunder Lager"
}
```

→

```
{ $group: {
  _id: '$brewery_name',
  beers: { '$addToSet': '$beer_name' }
} }
```

↓

```
{
  "_id" : "Paulaner Gruppe",
  "beers" : [
    "Münchener Gold",
    "Hefe Weisse"
  ]
}
{
  "_id" : "Orchard Pig",
  "beers" : [
    "Reveller"
  ]
}
```

Grouping with a Compound Key

```
{
  "beer_name" : "Sleigh Ride",
  "beer_type" : "Cider",
  "brewery_country" : "Wales"
}
{
  "beer_name" : "Mulled Cider",
  "beer_type" : "Cider",
  "brewery_country" : "Wales"
}
{
  "beer_name" : "Mulled Cider",
  "beer_type" : "Cider",
  "brewery_country" : "Wales"
}
{
  "beer_name" : "Two Trees",
  "beer_type" : "Perry",
  "brewery_country" : "Wales"
}
{
  "beer_name" : "Longdon Perry",
  "beer_type" : "Perry",
  "brewery_country" : "England"
}
```

→

```
{ $group: {
  _id: {
    'country' : '$brewery_country',
    'type' : '$beer_type'
  }
  beers: { '$addToSet': '$beer_name' }
} }
```

↓

```
{
  "_id" : { "country" : "England", "type" : "Perry" },
  "beers" : [ "Longdon Perry" ]
}
{
  "_id" : { "country" : "Wales", "type" : "Cider" },
  "beers" : [ "Mulled Cider", "Sleigh Ride" ]
}
{
  "_id" : { "country" : "Wales", "type" : "Perry" },
  "beers" : [ "Two Trees" ]
}
```

$unwind

- Operate on an array field
- Yield new documents for each array element
  - Array replaced by element value
  - Missing/empty fields → no output
  - Non-array fields → error
- Pipe to `$group` to aggregate array values

Yielding Multiple Documents from One

```
{
  "_id" : "Denmark",
  "beers" : [
    "Mochaccino Messiah",
    "Texas Ranger",
    "Tipsy Gypsy",
    "Carlsberg",
    "Black Ball",
    "Final Frontier DIPA",
    "Imperial Stout",
    "Svaneke Choko Stout",
  ]
}
```

→

```
{
  $unwind: '$beers'
}
```

↓

```
{ "_id": "Denmark", "beers": "Mochaccino Messiah" }
{ "_id": "Denmark", "beers": "Texas Ranger" }
{ "_id": "Denmark", "beers": "Tipsy Gypsy" }
{ "_id": "Denmark", "beers": "Carlsberg" }
{ "_id": "Denmark", "beers": "Black Ball" }
{ "_id": "Denmark", "beers": "Final Frontier DIPA" }
{ "_id": "Denmark", "beers": "Imperial Stout" }
{ "_id": "Denmark", "beers": "Svaneke Choko Stout" }
```

$sort, $limit, $skip

- Sort documents by one or more fields
  - Same order syntax as cursors
  - Waits for earlier pipeline operator to return
  - In-memory unless early and indexed
- Limit and skip follow cursor behaviour

Sort All the Documents in the Pipeline

```
{ "beer_name" : "Old Engine Oil" }
{ "beer_name" : "Pilsner" }
{ "beer_name" : "Du Monde Blanc" }
{ "beer_name" : "Rallar Amber Ale" }
{ "beer_name" : "Beer Geek Breakfast" }
{ "beer_name" : "Dirty Bastard" }
{ "beer_name" : "Brixton Porter" }
```



```
{ $sort: {
  beer_name: 1
} }
```

```
{ "beer_name" : "Beer Geek Breakfast" }
{ "beer_name" : "Brixton Porter" }
{ "beer_name" : "Dirty Bastard" }
{ "beer_name" : "Du Monde Blanc" }
{ "beer_name" : "Old Engine Oil" }
{ "beer_name" : "Pilsner" }
{ "beer_name" : "Rallar Amber Ale" }
```

Limit Documents through the Pipeline

```
{ "beer_name" : "Old Engine Oil" }
{ "beer_name" : "Pilsner" }
{ "beer_name" : "Du Monde Blanc" }
{ "beer_name" : "Rallar Amber Ale" }
{ "beer_name" : "Beer Geek Breakfast" }
{ "beer_name" : "Dirty Bastard" }
{ "beer_name" : "Brixton Porter" }
```

```
{ $limit : 5 }
```

```
{ "beer_name" : "Old Engine Oil" }
{ "beer_name" : "Pilsner" }
{ "beer_name" : "Du Monde Blanc" }
{ "beer_name" : "Rallar Amber Ale" }
{ "beer_name" : "Beer Geek Breakfast" }
```

Skip Over Documents in the Pipeline

```
{ "beer_name" : "Old Engine Oil" }
{ "beer_name" : "Pilsner" }
{ "beer_name" : "Du Monde Blanc" }
{ "beer_name" : "Rallar Amber Ale" }
{ "beer_name" : "Beer Geek Breakfast" }
{ "beer_name" : "Dirty Bastard" }
{ "beer_name" : "Brixton Porter" }
```



```
{ $skip : 2 }
```

```
{ "beer_name" : "Du Monde Blanc" }
{ "beer_name" : "Rallar Amber Ale" }
{ "beer_name" : "Beer Geek Breakfast" }
{ "beer_name" : "Dirty Bastard" }
{ "beer_name" : "Brixton Porter" }
```

Order of $sort, $limit, $skip is Important

```
{ "beer_name" : "Old Engine Oil" }
{ "beer_name" : "Pilsner" }
{ "beer_name" : "Du Monde Blanc" }
{ "beer_name" : "Rallar Amber Ale" }
{ "beer_name" : "Beer Geek Breakfast" }
{ "beer_name" : "Dirty Bastard" }
{ "beer_name" : "Brixton Porter" }
```

→

```
[
  { $limit : 4 },
  { $sort : { beer_name: 1 } }
]
```

```
{ "beer_name" : "Du Monde Blanc" }
{ "beer_name" : "Old Engine Oil" }
{ "beer_name" : "Pilsner" }
{ "beer_name" : "Rallar Amber Ale" }
```

```
{ "beer_name" : "Old Engine Oil" }
{ "beer_name" : "Pilsner" }
{ "beer_name" : "Du Monde Blanc" }
{ "beer_name" : "Rallar Amber Ale" }
{ "beer_name" : "Beer Geek Breakfast" }
{ "beer_name" : "Dirty Bastard" }
{ "beer_name" : "Brixton Porter" }
```

→

```
[
  { $sort : { beer_name: 1 } },
  { $limit : 4 }
]
```

```
{ "beer_name" : "Beer Geek Breakfast" }
{ "beer_name" : "Brixton Porter" }
{ "beer_name" : "Dirty Bastard" }
{ "beer_name" : "Du Monde Blanc" }
```

Aggregation example

Find 10 venues with the most distinct beers

```
<?php
$m = new MongoClient();

$res = $m->tutorial->beer->aggregate( [
    [ '$match' => [ 'venue_name' => [ '$ne' => null ] ] ],
    [ '$group' => [ '_id' => [ 'venue' => '$venue_name', 'beer' => '$beer_name' ] ] ],
    [ '$group' => [ '_id' => '$_id.venue', count => [ '$sum' => 1 ] ] ],
    [ '$sort' => [ 'count' => -1 ] ],
    [ '$limit' => 10 ],
] );

foreach ( $res['result'] as $result )
{
    echo $result['_id'], ': ', $result['count'], "\n";
}
?>
```

Result:

```
Kilburn Park: 195
Euston Cider Tap: 97
Lowlander Grand Cafe: 80
London Craft Beer Festival: 45
The Earl Derby: 35
The Euston Tap: 34
```

Exercise

Exercise: Aggregation

Write a script that uses the Aggregation Framework to:

- List the average `rating_score` per `beer_type`
- Only include types if there are more than 5 ratings
- Sort by highest rated type
- Show only the top 8

Any questions?

Indexes

Creating Indexes

The `_id` field is always indexed. Additional indexes can be created with `createIndex`:

```
<?php
$m = new MongoClient;
$c = $m->tutorial->beer;

// Create an index on the beer name attribute
$c->createIndex( [ 'beer_name' => 1 ] );

// Create a compound index on beer name and brewery name
$c->createIndex( [ 'beer_name' => 1, 'brewery_name' => 1 ] );

// Create a unique index on the name field of the venue attribute
$c->createIndex( [ 'venue.name' => 1 ] );
?>
```

Let's imagine a compound index

Country Code	Population	Name
AD	15853	les Escaldes
AD	20430	Andorra la Vella
...	...	...
GB	435791	Edinburgh
GB	447047	Sheffield
GB	455123	Leeds
GB	468945	Liverpool
GB	610268	Glasgow
GB	984333	Birmingham
GB	7556900	London
...	...	...

```
db.cities.createIndex( { country_code: 1, population: 1 } );
```

Let's look at the query plan

```
db.cities.find( { country_code: 'GB', population: { $gte: 500000 } } ).explain();
```

```
{
        "cursor" : "BtreeCursor country_code_1_population_1",
        "nscanned" : 4,
        "nscannedObjects" : 4,
        "n" : 4,
        "nscannedObjectsAllPlans" : 9,
        "nscannedAllPlans" : 10,
        "millis" : 0,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {
                "country_code" : [ [ "GB", "GB" ] ],
                "population" : [ [ 500000, Infinity ] ]
        }
}
```

```
{ "name" : "Glasgow", "country_code" : "GB", "population" : 610268 }
{ "name" : "Birmingham", "country_code" : "GB", "population" : 984333 }
{ "name" : "City of London", "country_code" : "GB", "population" : 7556900 }
{ "name" : "London", "country_code" : "GB", "population" : 7556900 }
```

Let's look at the query plan when we try to sort

```
db.cities.find( { country_code: 'GB', population: { $gte: 500000 } } )
         .sort( { population: -1 }).explain();
```

```
{
        "cursor" : "BtreeCursor country_code_1_population_1 reverse",
        "nscanned" : 4,
        "nscannedObjects" : 4,
        "n" : 4,
        "nscannedObjectsAllPlans" : 9,
        "nscannedAllPlans" : 10,
        "millis" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {
                "country_code" : [ [ "GB", "GB" ] ],
                "population" : [ [ Infinity, 500000 ] ]
        }
}
```

```
{ "name" : "London", "country_code" : "GB", "population" : 7556900 }
{ "name" : "City of London", "country_code" : "GB", "population" : 7556900 }
{ "name" : "Birmingham", "country_code" : "GB", "population" : 984333 }
{ "name" : "Glasgow", "country_code" : "GB", "population" : 610268 }
```

Exercise

Exercise: Indexes

A common query is to find the ten highest rated beers ordered by `rating_score` first, and `created_at` second. Write a script to:

- Run the query
- Run the query with explain
- Add the index
- Run the query again with explain
- Delete the index that you've created

Let's imagine another compound index

Country	Name	Population	Elevation
AD	les Escaldes	15853	1033
AD	Andorra la Vella	20430	1037
...	...	...	...
ET	Addis Ababa	2757729	2405
ET	Ādīgrat	65000	2451
...	...	...	...
MX	Mexico City	12294193	2240
...	...	...	...

```
db.cities.createIndex( { population: 1, elevation: 1 } );
```

Let's look at the query plan

```
db.cities.find( { population: { $gte: 2000000 }, elevation: { $gte: 1654 } } ).explain();
```


```
{
        "cursor" : "BtreeCursor population_1_elevation_1",
        "nscanned" : 131,
        "nscannedObjects" : 6,
        "n" : 6,
        "nscannedObjectsAllPlans" : 6,
        "nscannedAllPlans" : 131,
        "millis" : 3,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {
                "population" : [ [ 2000000, Infinity ] ],
                "elevation" : [ [ 1654, Infinity ] ]
        }
}
```



```
{ "name" : "Johannesburg", "population" : 2026469, "elevation" : 1767 }
{ "name" : "Nairobi", "population" : 2750547, "elevation" : 1684 }
{ "name" : "Addis Ababa", "population" : 2757729, "elevation" : 2405 }
{ "name" : "Kabul", "population" : 3043532, "elevation" : 1798 }
{ "name" : "Bogotá", "population" : 7102602, "elevation" : 2582 }
{ "name" : "Mexico City", "population" : 12294193, "elevation" : 2240 }
```

A quick look on how index look-ups work

```
db.cities.find( { population: { $gte: 2000000 }, elevation: { $gte: 1654 } } );
```

	...	...	...
	Shijiazhuang	1992474	77
	Medellín	1999979	1500
	Almaty	2000900	793
	Zhengzhou	2014125	104
	Al Başrat al Qadīmah	2015483	6
	Kowloon	2019533	92
	Johannesburg	2026469	1767
	Dalian	2035307	33
⇒	...	...	...

```
n = 1 - nscanned = 6
```

Sorting by descending elevation

```
db.cities.find( { population: { $gte: 2000000 }, elevation: { $gte : 1654 } } )
         .sort( { elevation: -1 } ).explain();
```

```
{
        "cursor" : "BtreeCursor population_1_elevation_1",
        "nscanned" : 127,
        "nscannedObjects" : 6,
        "n" : 6,
        "scanAndOrder" : true,
        "millis" : 1,
...
```

- Batch and sort results in memory
- No streaming (with getMore) of results
- Data limit for sorting

Querying by just the elevation

```
db.cities.find( { elevation: { $gte : 1654 } } ).explain();
```

```
{
        "cursor" : "BasicCursor",
        "nscanned" : 22840,
        "nscannedObjects" : 22840,
        "n" : 614,
        "millis" : 26,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {
        }
}
```

MongoDB uses indexes in order from left to right

Sorting by descending elevation - take 2

```
db.cities.createIndex( { elevation: -1 } );
db.cities.find( { population: { $gte: 2000000 }, elevation: { $gte : 1654 } } )
         .sort( { elevation: -1 } ).explain();
```

```
{
        "cursor" : "BtreeCursor population_1_elevation_1",
        "nscanned" : 127,
        "nscannedObjects" : 6,
        "n" : 6,
        "scanAndOrder" : true,
        "millis" : 1,
...
```

In most cases, MongoDB can only use one index at the same time.

Indexes

- MongoDB uses B-trees for indexes
- Many of the principles that apply to an RDBMS also apply to MongoDB
- e.g., indexes require additional work on inserts, updates and deletes
- A collection can have 64 indexes at most
- MongoDB can only use one index per query (with a few exceptions)

Indexes (2)

Indexes are used from left to right.

```
$c->createIndex( [ 'country_code' => 1, 'timezone' => 1, 'asciiname' => 1 ] )
```

✔ `$c->find(['country_code' => 'GB']);`

✔ `$c->find(['country_code' => 'GB', 'timezone' => 'Europe/London']);`

✔ `$c->find(['country_code' => 'GB']).sort(['timezone' => 1]);`

✘ `$c->find(['country_code' => 'GB', 'asciiname' => 'London']);`

✘ `$c->find(['country_code' => 'GB']).sort(['asciiname' => 1]);`

✘ `$c->find(['timezone' => 'Europe/London']);`

✘ `$c->find(['asciiname' => 'Europe/London']);`

✔ `$c->find(['country_code' => 'GB']).sort(['timezone' => 1, 'asciiname' => 1]);`

✘ `$c->find(['country_code' => 'GB']).sort(['timezone' => 1, 'asciiname' => -1]);`

✘ `$c->find(['country_code' => 'GB']).sort(['timezone' => -1, 'asciiname' => 1]);`

✔ `$c->find(['country_code' => 'GB']).sort(['timezone' => -1, 'asciiname' => -1]);`

When can indexes not be used?

- Many types of negations (`$ne`, `$not`, `$nin`).
- Tricky arithmetic (`$mod`).
- Most regular expressions (`/ondo/`), unless anchored to the start of the string (`/^Lon/`).
- JavaScript expressions in `$where` clauses, such as `where elevation > population`. In that case, try to pre-compute.

How does MongoDB pick an index?

- MongoDB does not keep cardinality statistics
- MongoDB tries all the possible indexes
- When it has found the fastest one, it remembers it
- MongoDB will re-try plans once in a while
- Indexes can be "hinted":

```
db.cities.createIndex( { population: 1, elevation: 1 } );
db.cities.createIndex( { elevation: -1 } );
db.cities.find( { population: { $gte: 2000000 }, elevation: { $gte : 1654 } })
         .sort( { elevation: -1 } )
         .hint( { elevation: -1 } );
```

```
{
        "cursor" : "BtreeCursor elevation_-1",
        "nscanned" : 614,
        "nscannedObjects" : 614,
        "n" : 6,
        "millis" : 2,
...
```

Any questions?

Replication

Replication

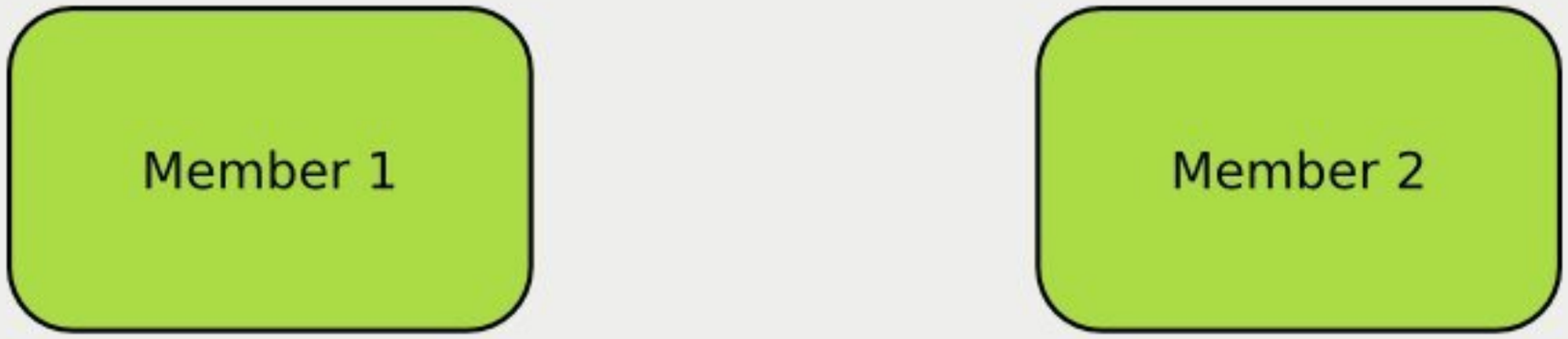
Use cases

- High Availability (auto-failover)
- Backups
  - Online, Delayed Copy (fat finger)
  - Point in Time (PiT) backups
- Use (hidden) replica for secondary workload
  - Analytics
  - Data-processing
  - Integration with external systems (f.e. Solr)

MongoDB Replica Set Features

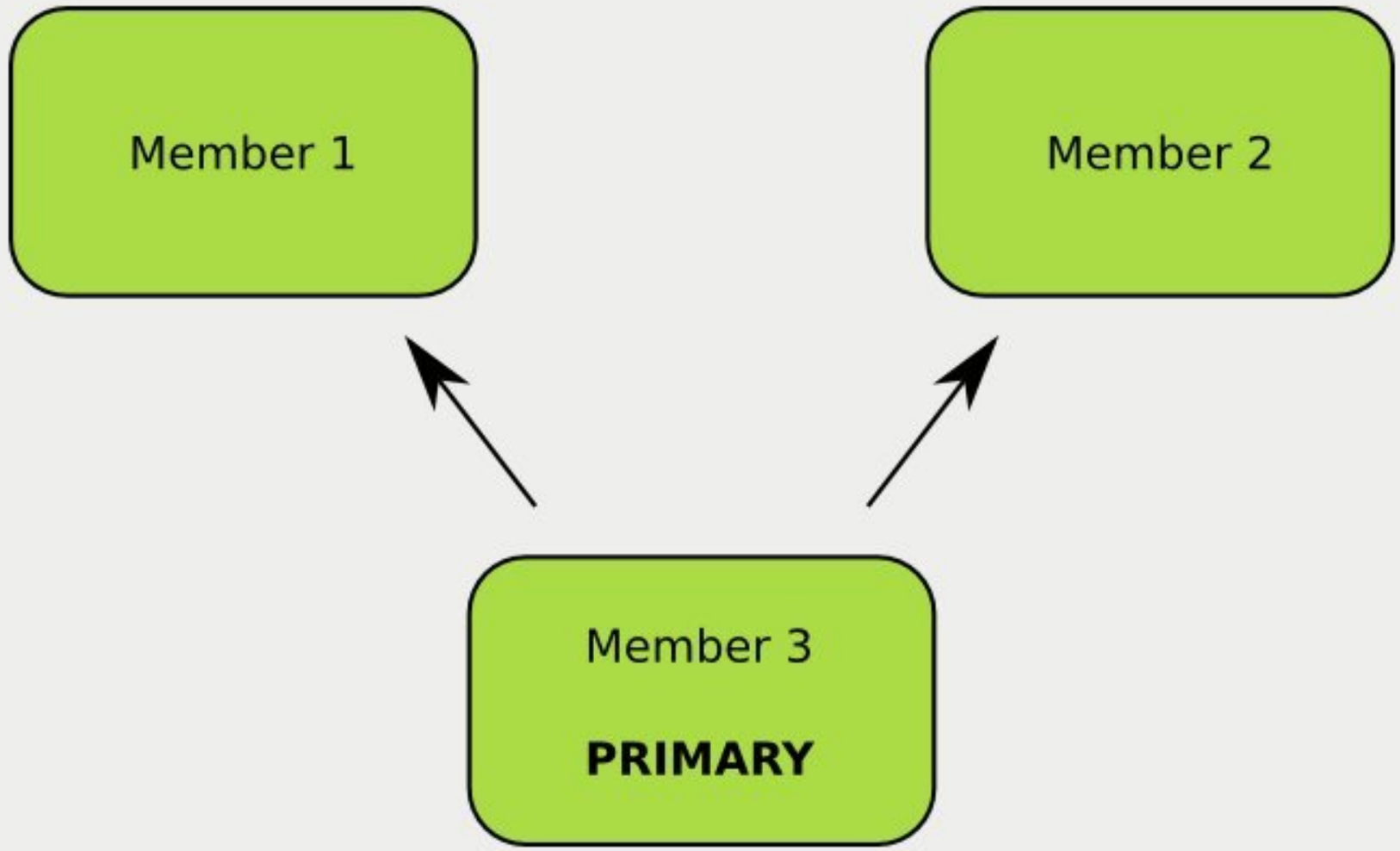
- A cluster of N servers
- All writes to primary
- Reads can be from primary (default) or a secondary
- Any (one) node can be primary
- Consensus election of primary
- Automatic failover and recovery

How MongoDB replication works



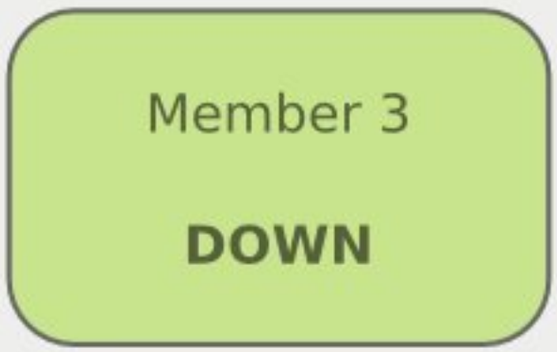
- A set should have 3 or more nodes, and always an **odd** number of voters (for calculating majority)

How MongoDB replication works



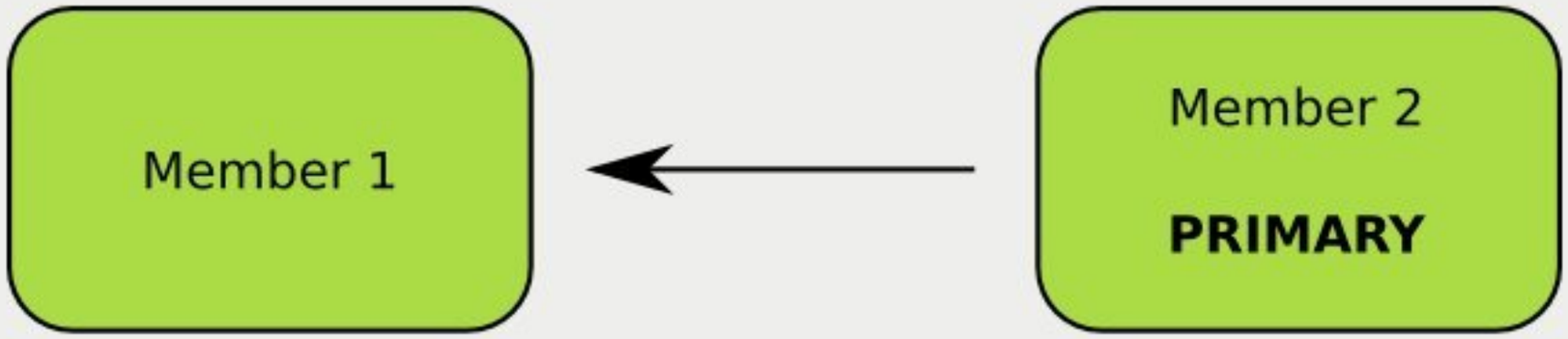
- Election establishes the PRIMARY
- Data replication from PRIMARY to SECONDARY

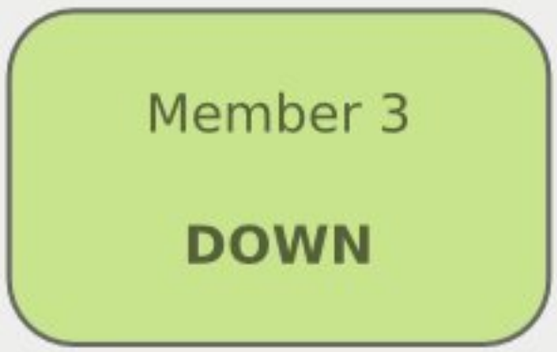
How MongoDB replication works



- PRIMARY may fail
- Automatic election of new PRIMARY if majority exists

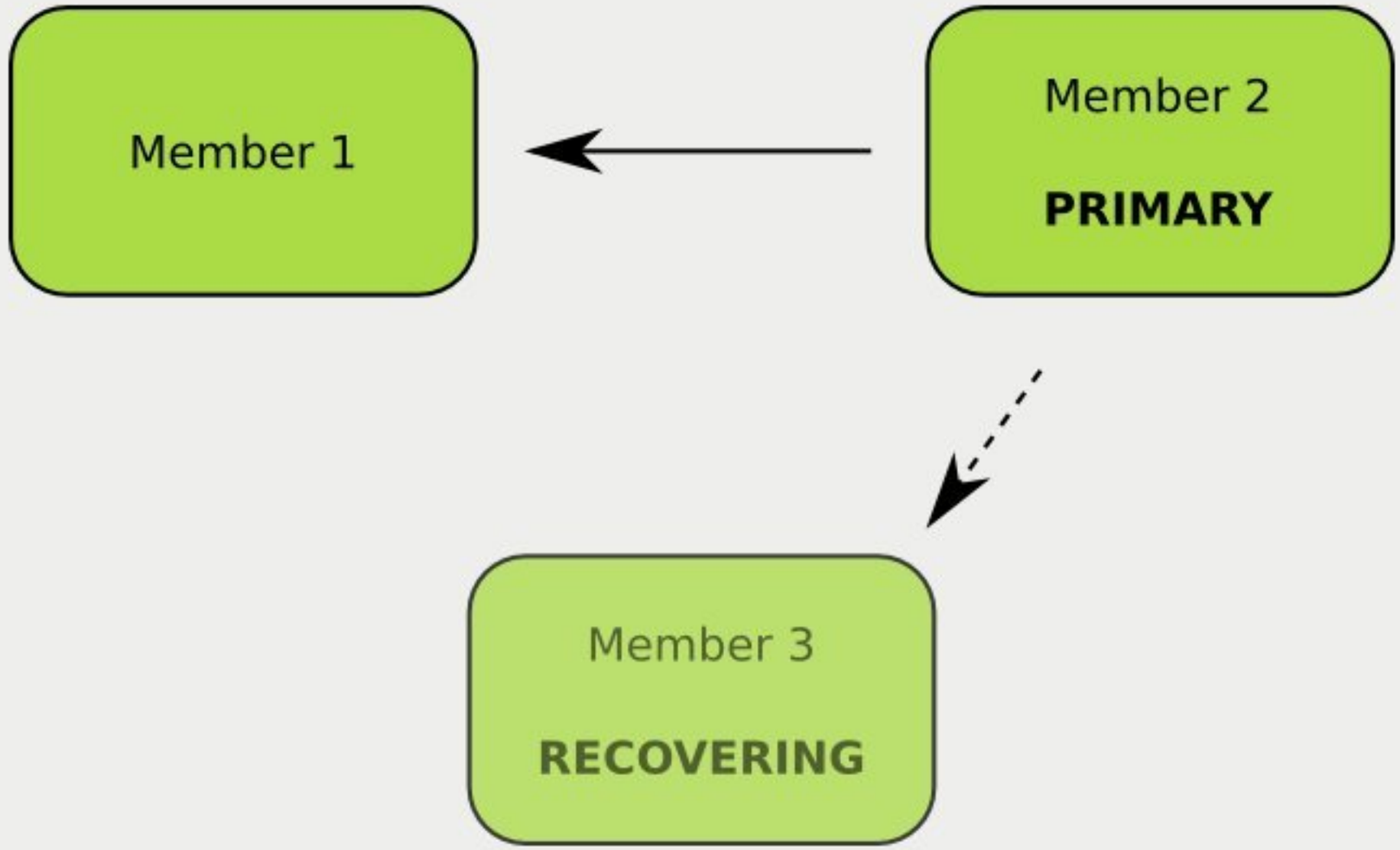
How MongoDB replication works





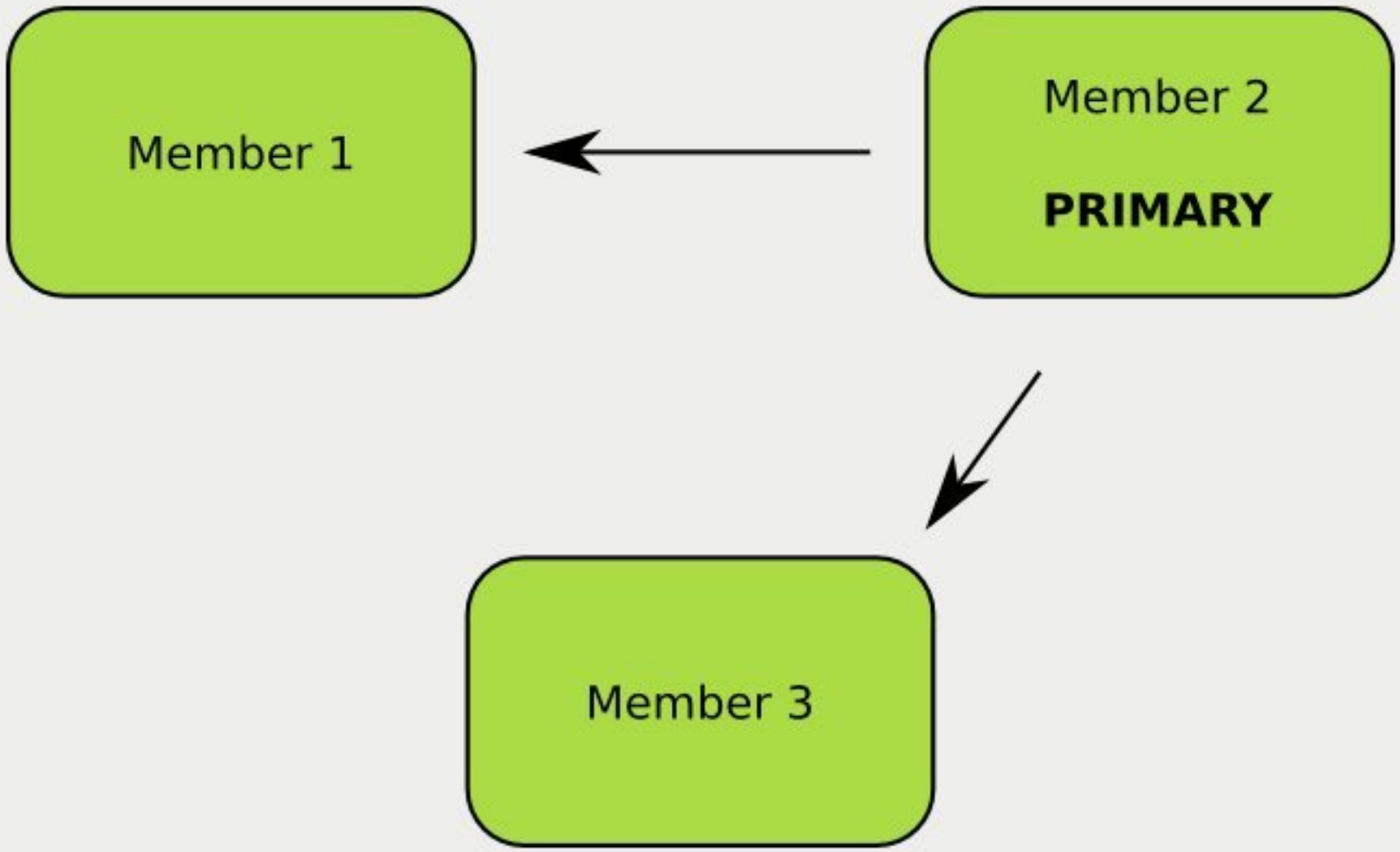
- New PRIMARY elected
- Replica set re-established

How MongoDB replication works



- Automatic recovery

How MongoDB replication works



- Replica set re-established

How does replication work?

- Change operations are written to the oplog
  - A single update/delete affecting multiple documents can result in many oplog entries
  - The oplog is a capped collection (fixed size)
    - Must have enough space to allow new secondaries to catch up after syncing from a primary, and cope with any applicable slaveDelay
- Secondaries query the primary's oplog and apply what they find
- All replica set members contain an oplog

Setting up replication

Starting the daemons (2 data, 1 arbiter):

```
mongod -f /etc/mongodb.conf --dbpath=~/repl-slave0 --port 13000 --replSet seta
mongod -f /etc/mongodb.conf --dbpath=~/repl-slave1 --port 13001 --replSet seta
mongod -f /etc/mongodb.conf --dbpath=~/repl-arb --port 13002 --replSet seta --rest
```

Configure the set:

```
mongo whisky:13000

> db.adminCommand( {
  replSetInitiate: {
    _id: 'seta',
    members: [
     { _id: 0, host: 'whisky:13000', tags: { 'dc': 'west', 'use': 'accounting' } },
     { _id: 1, host: 'whisky:13001', tags: { 'dc': 'east', 'use': 'reporting' } },
    ]
  }
} );

PRIMARY> rs.addArb('whisky:13002');
```

Connection String

```
<?php
$options = [ 'replicaSet' => 'seta' ];

$m = new MongoClient( 'mongodb://localhost:13000/?replicaSet=seta' );

$m = new MongoClient( 'mongodb://localhost:13000,localhost:13001', $options );

$m = new MongoClient( 'mongodb://user:password@localhost:13000/demo', $options );
?>
```

- Add more than one host for seeding
- Don't add the arbiters to the connection string
- Don't use `->authenticate()`

Eventual consistency

- Read Preference / Slave Okay
  - Driver will always send writes to primary
  - Driver will send read requests to secondaries
  - Options to prefer primary, secondary, or nearest
- Warning!
  - Secondaries may be out of date
  - Not appropriate for all applications

Sharding

Agenda

- Short history of scaling data
- Why shard
- MongoDB's approach
- Architecture
- Configuration
- Mechanics
- Solutions

The story of scaling data

www.logoopenstock.com
www.etiennemansard.com

1970 - 2000: Vertical Scalability (scale up)

Google, ~2000: Horizontal Scalability (scale out)

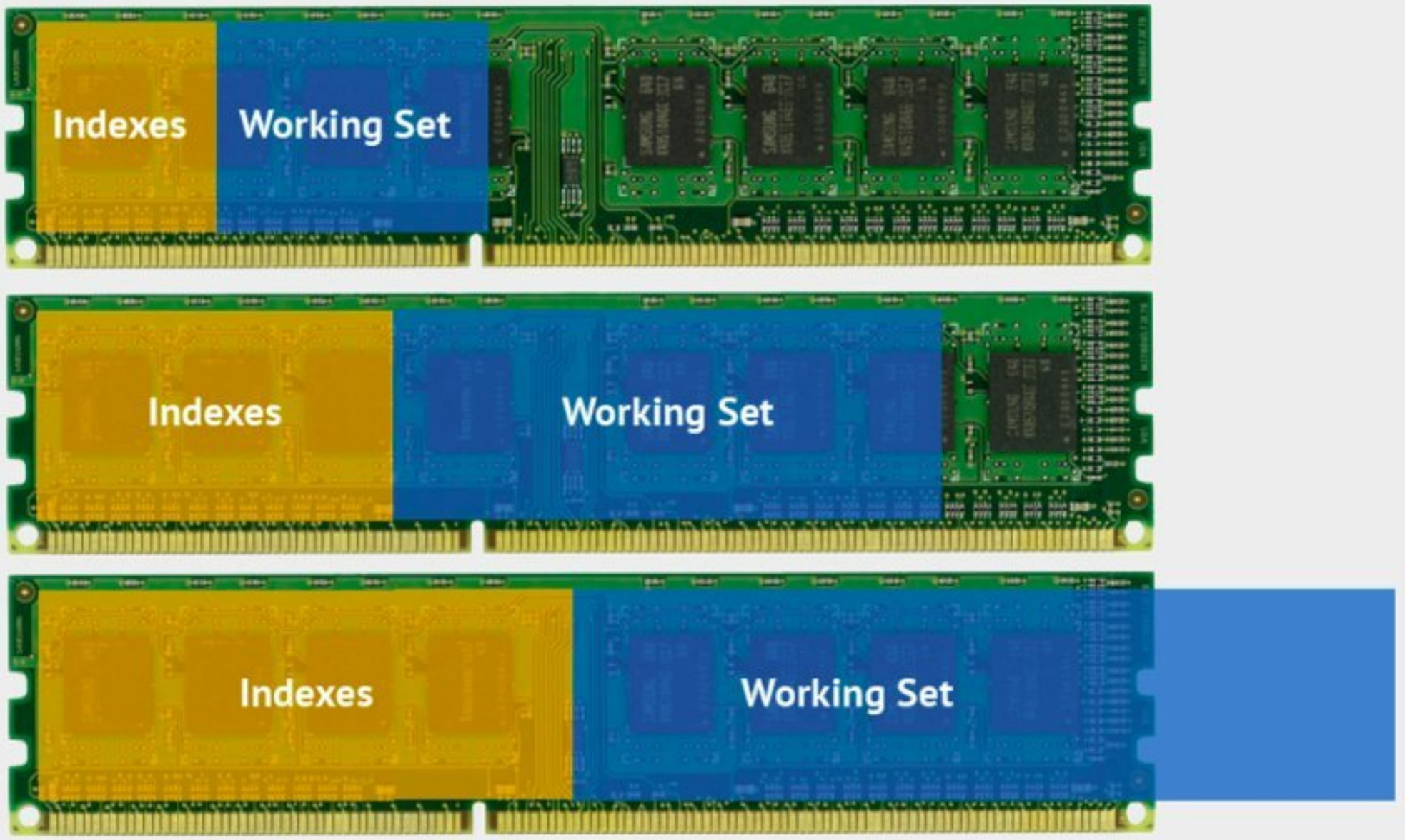
Data Store Scalability in 2005

- Custom Hardware
  - Oracle
- Custom Software
  - Facebook + MySQL

Data Store Scalability Today

- MongoDB auto-sharding available in 2009
- A data store that is
  - Free and publicly available
  - Open source (https://github.com/mongodb/mongo)
  - Horizontally scalable
  - Application independent

Why shard?



Working Set Exceeds Physical Memory

Read/Write Throughput Exceeds I/O

MongoDB's approach to sharding

Partition data based on ranges

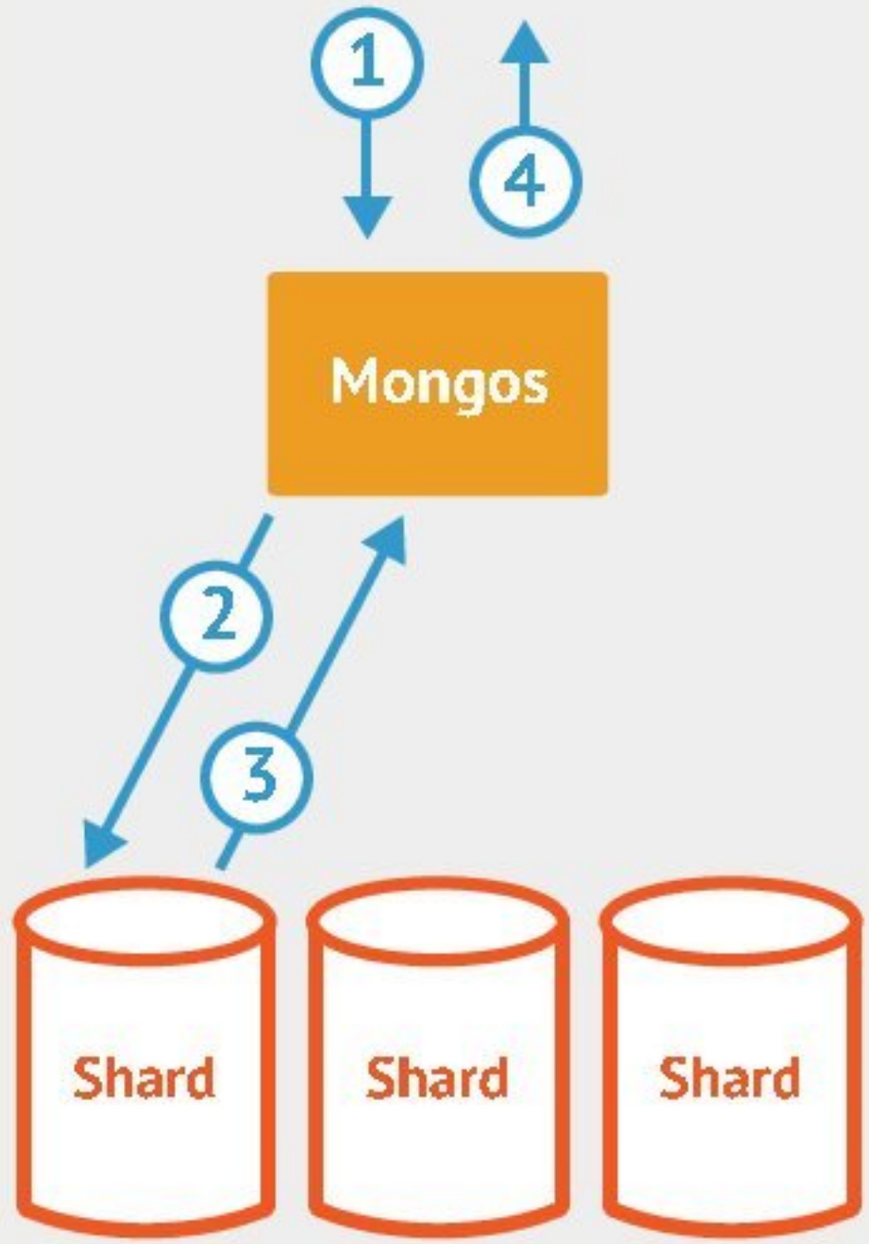
- User defines shard key
- Shard key defines range of data
- Key space is like points on a line
- Range is a segment of that line

Distribute data in chunks across nodes

- Initially one chunk
- Default max chunk size: 64mb
- MongoDB automatically splits & migrates chunks when max reached

MongoDB manages data

- Queries routed to specific shards
- MongoDB balances cluster
- MongoDB migrates data to new nodes



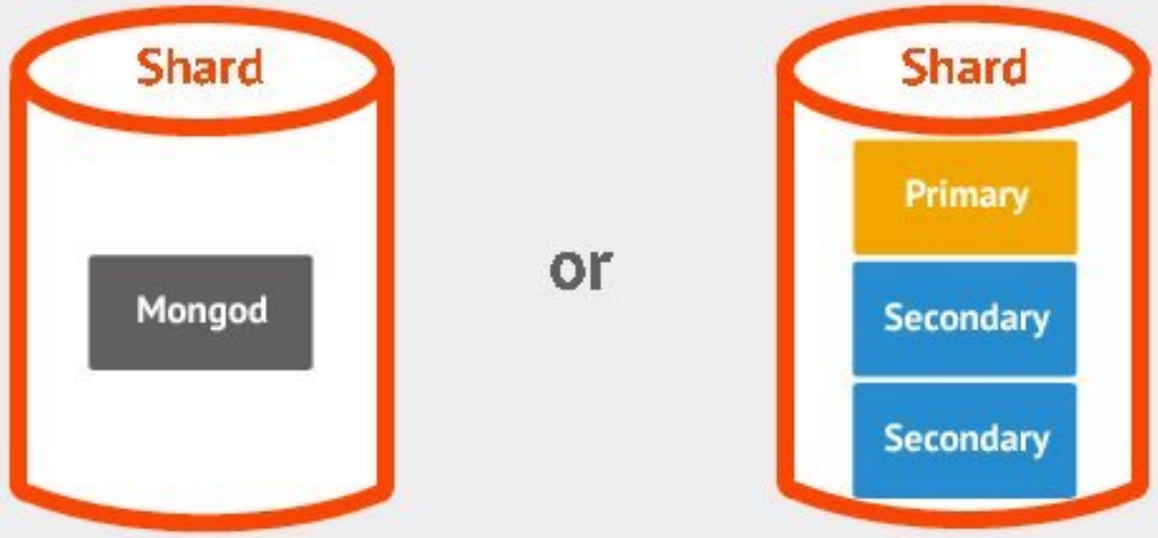
MongoDB Auto-Sharding

- Minimal effort required
  - Same interface as single mongod
- Two steps
  - Enable sharding for a database
  - Shard collection(s) within database

Architecture

Data stored in a shard

- Shard is a node of the cluster
- Shard can be a single mongod or a replica set



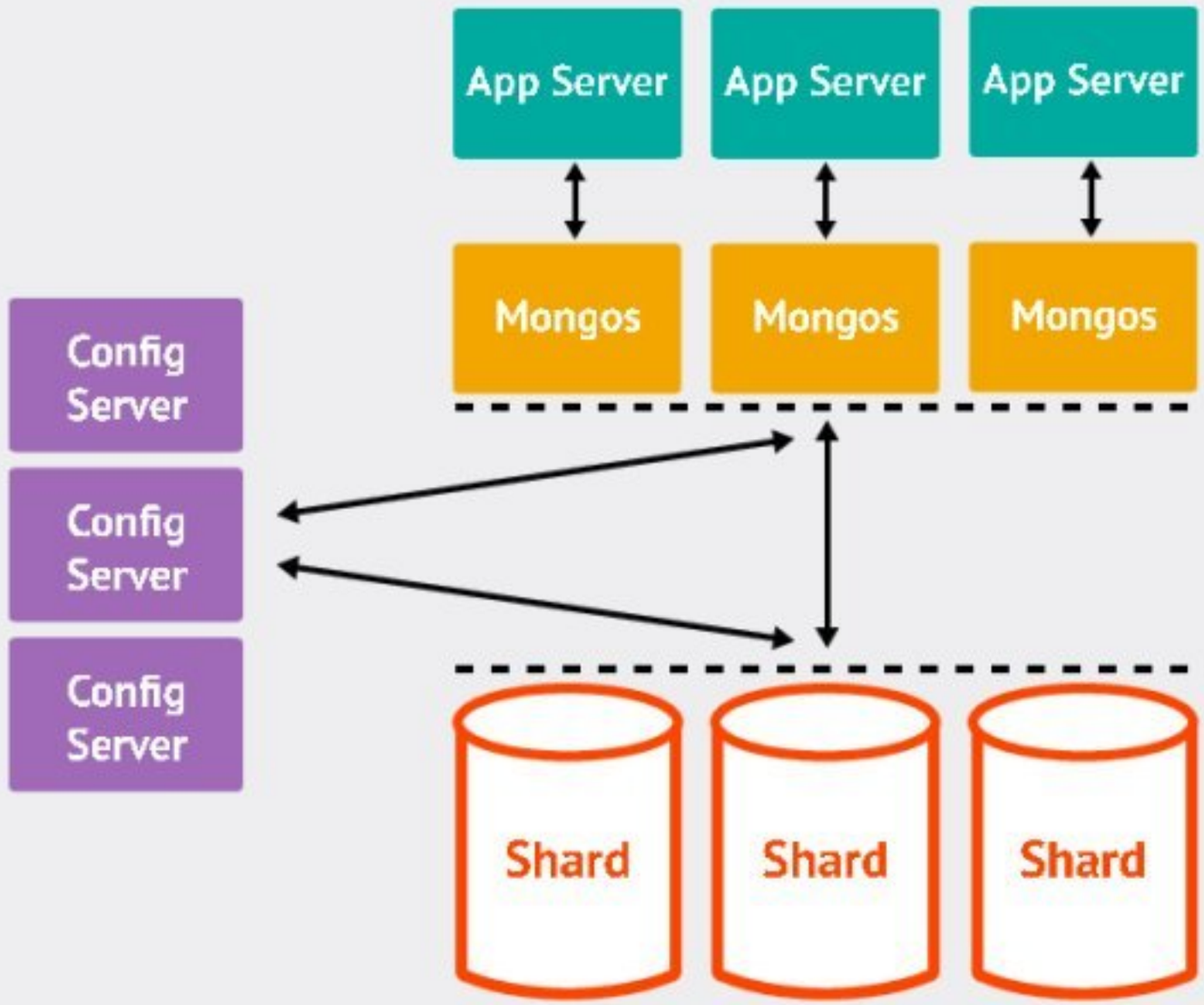
Config server stores meta data

- Config server
  - Stores cluster chunk ranges and locations
  - Can have only 1 or 3 (production must have 3)
  - Two-phase commit (not a replica set)

MongoS manages the data

Mongos

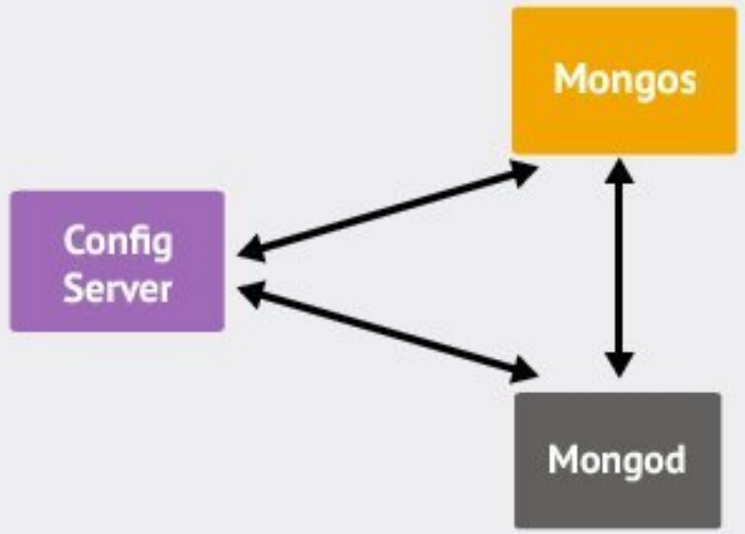
- Acts as a router / balancer
- No local data (persists to config database)
- Can have one or many (e.g. each application server)



Sharding Infrastructure

Configuration

Example cluster setup

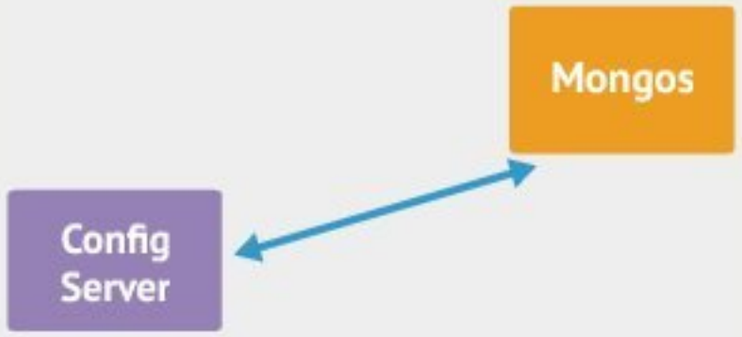


- **Don’t use this setup in production!**
  - Only one config server (no fault tolerance)
  - Shard is not a replica set (low availability)
  - Only one mongos and shard (no performance improvement)
  - Useful for development or demonstrating configuration mechanics

Start the config server

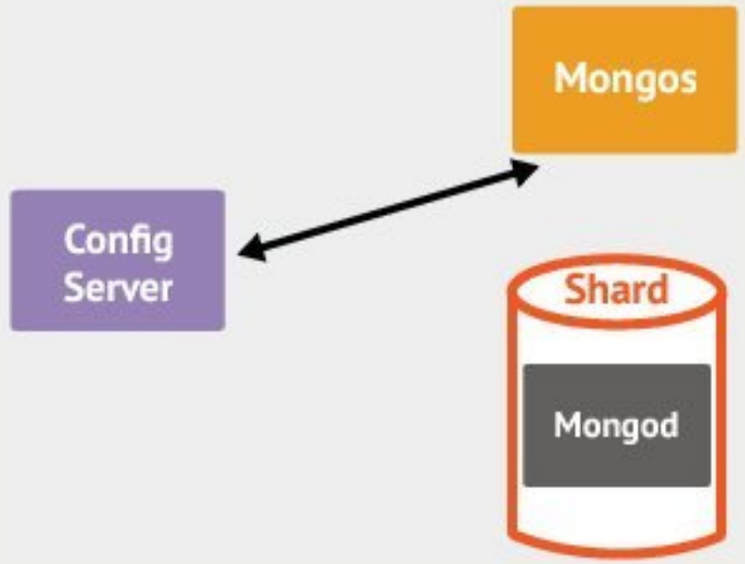
- `mongod --configsvr`
- Starts a config server on the default port (27019)

Start the mongos router



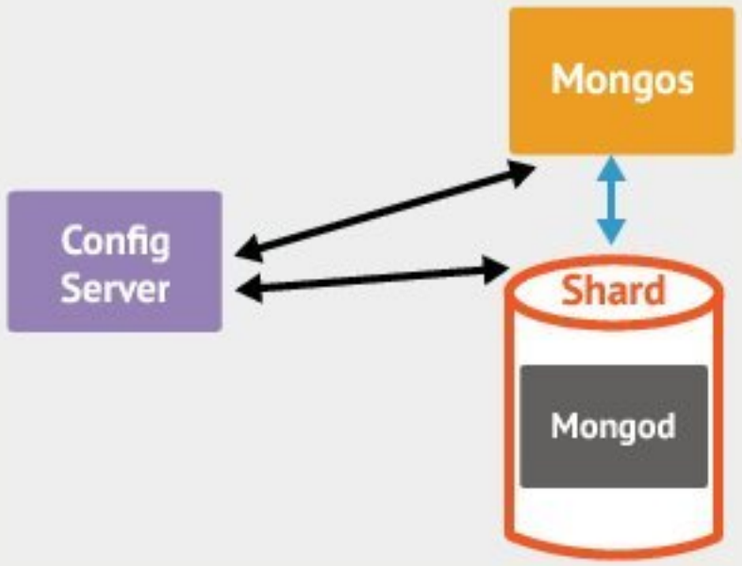
- `mongos --configdb <hostname>:27019`
- For 3 config servers:
  `mongos --configdb <host1>:<port1>,<host2>:<port2>, <host3>:<port3>`
- This is how one always starts a new mongos, even if the cluster is already running

Start the shard database



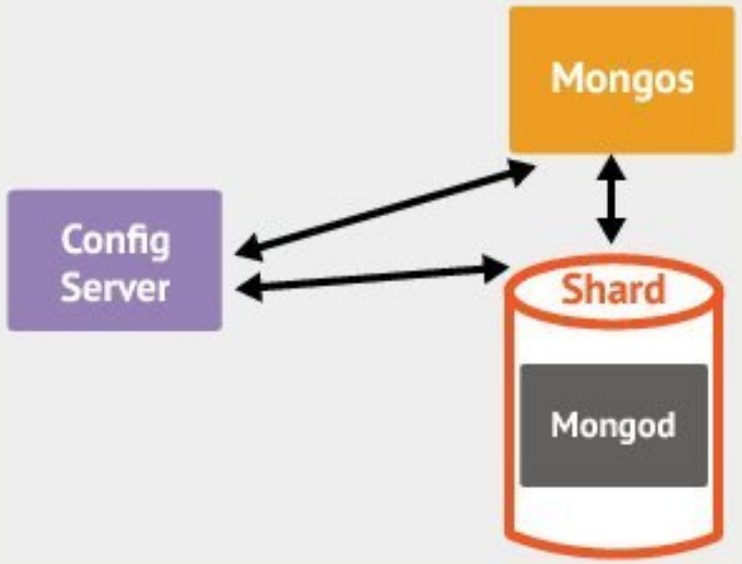
- `mongod --shardsvr`
- Starts a mongod with the default shard port (27018)
- Shard is not yet connected to the rest of the cluster
- Shard may have already been running in production

Add the shard



- On mongos:
  `sh.addShard('<host>:<port>')`
- Adding a replica set:
  `sh.addShard('<rsname>/<seedlist>')`

Verify that the shard was added



- db.runCommand({ listshards:1 })

```
{
    "shards" : [
        { "_id": "shard0000", "host": "<hostname>:27018" }
    ],
    "ok" : 1
}
```

Enabling Sharding

- Enable sharding on a database:
  `sh.enableSharding("<dbname>")`
- Shard a collection with the given key:
  `sh.shardCollection("<dbname>.people", {"country":1})`
- Granular shard keys can avoid indivisible chunks:
  `sh.shardCollection("<dbname>.cars", {"year":1, "uniqueid":1})`

Shard Key

Shard Key

- Shard key is **immutable**
- Shard key values are **immutable**
- Shard key requires index on fields contained in key
- Uniqueness of `_id` field is only guaranteed within individual shard
- Shard key limited to 512 bytes in size

Shard Key Considerations

- Cardinality
- Write distribution
- Query isolation
- Data Distribution

Example: email storage

```
{
        _id: ObjectId("51156a11056d6f966f268f7f"),
        user: 37113,
        time: ISODate( '2013-04-24 10:38:05 '),
        subject: "This is a test",
        recipients: [ "derick@10gen.com", "derick@example.com" ],
        body: "..."
}
```

- Lots of emails per user
- Most common query: get user emails sorted by time
- Index on `{ _id: 1 }`, `{ user: 1, time: -1 }`, and `{ recipients: 1 }`

Example: email storage

	Cardinality	Write scaling	Query isolation	Index Locality
_id	Doc level	1 shard	All shards, merge sort	Great
hash(_id)	Hash level	All shards	All shards, merge sort	Poor
user	Many docs	All shards	One shard, index sort	So-so
user,time	Doc level	All shards	One shard, index sort	Good

```
{
        _id: ObjectId("51156a11056d6f966f268f7f"),
        user: 37113,
        time: ISODate( '2013-04-24 10:38:05 '),
        subject: "This is a test",
        recipients: [ "derick@10gen.com", "derick@example.com" ],
        body: "..."
}
```

Connection String

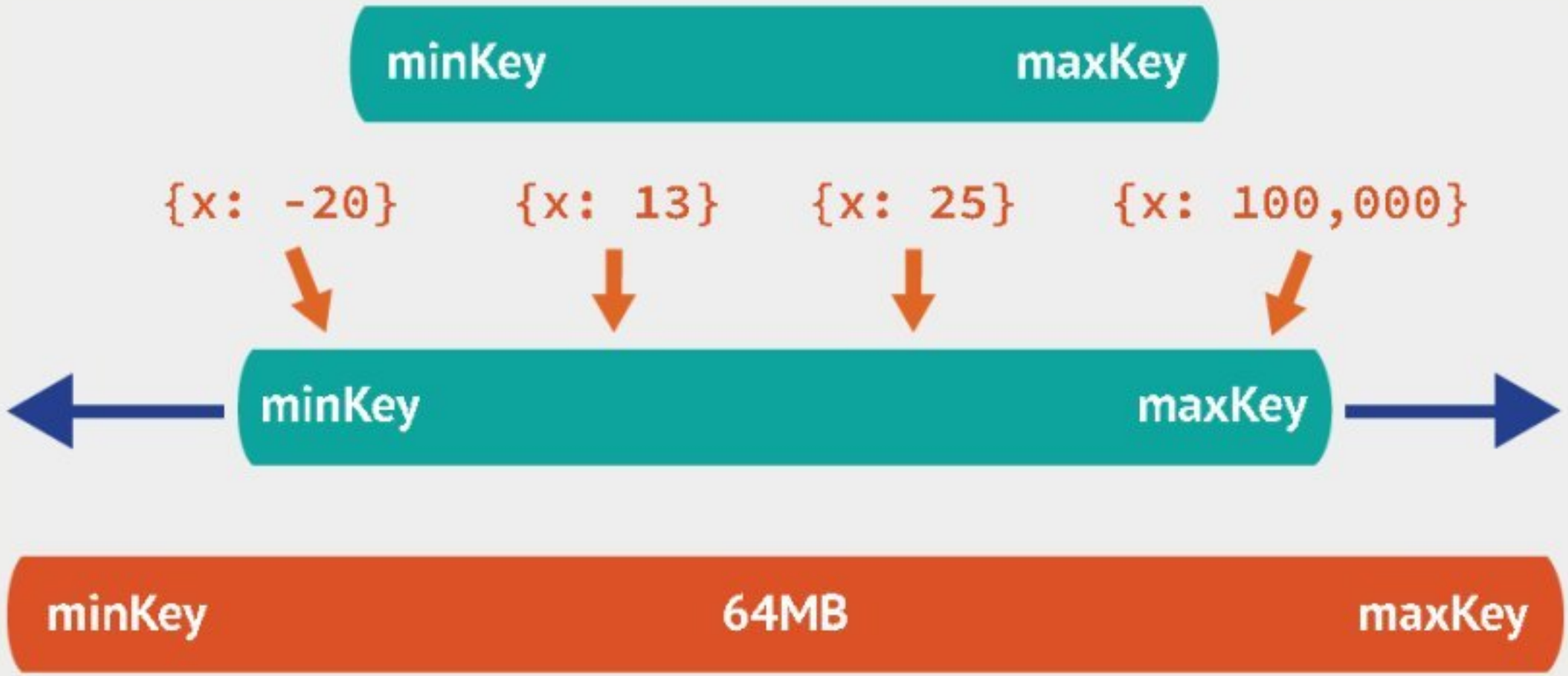
```
<?php
$m = new MongoClient( 'mongodb://localhost:27019' );

$m = new MongoClient( 'mongodb://localhost:27019,example.com:27019' );

$m = new MongoClient( 'mongodb://user:password@localhost:27019/demo', $options );
?>
```

- Remember: mongos may be run on the app server
- Add multiple hosts for redundancy
- Don't use `->authenticate()`

Mechanics

Partitioning

- Remember: it's based on ranges

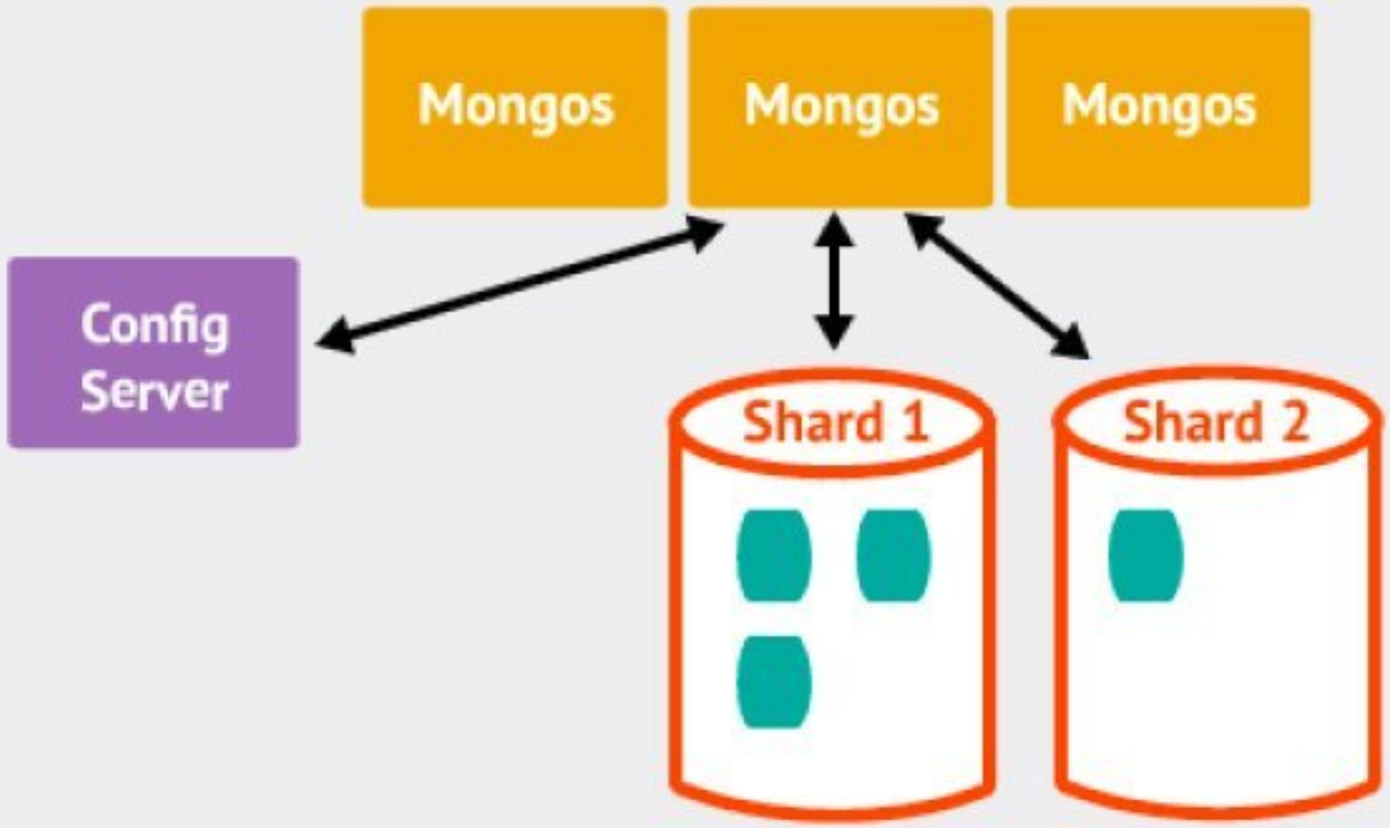


Chunk is a section of an entire range

Chunk splitting

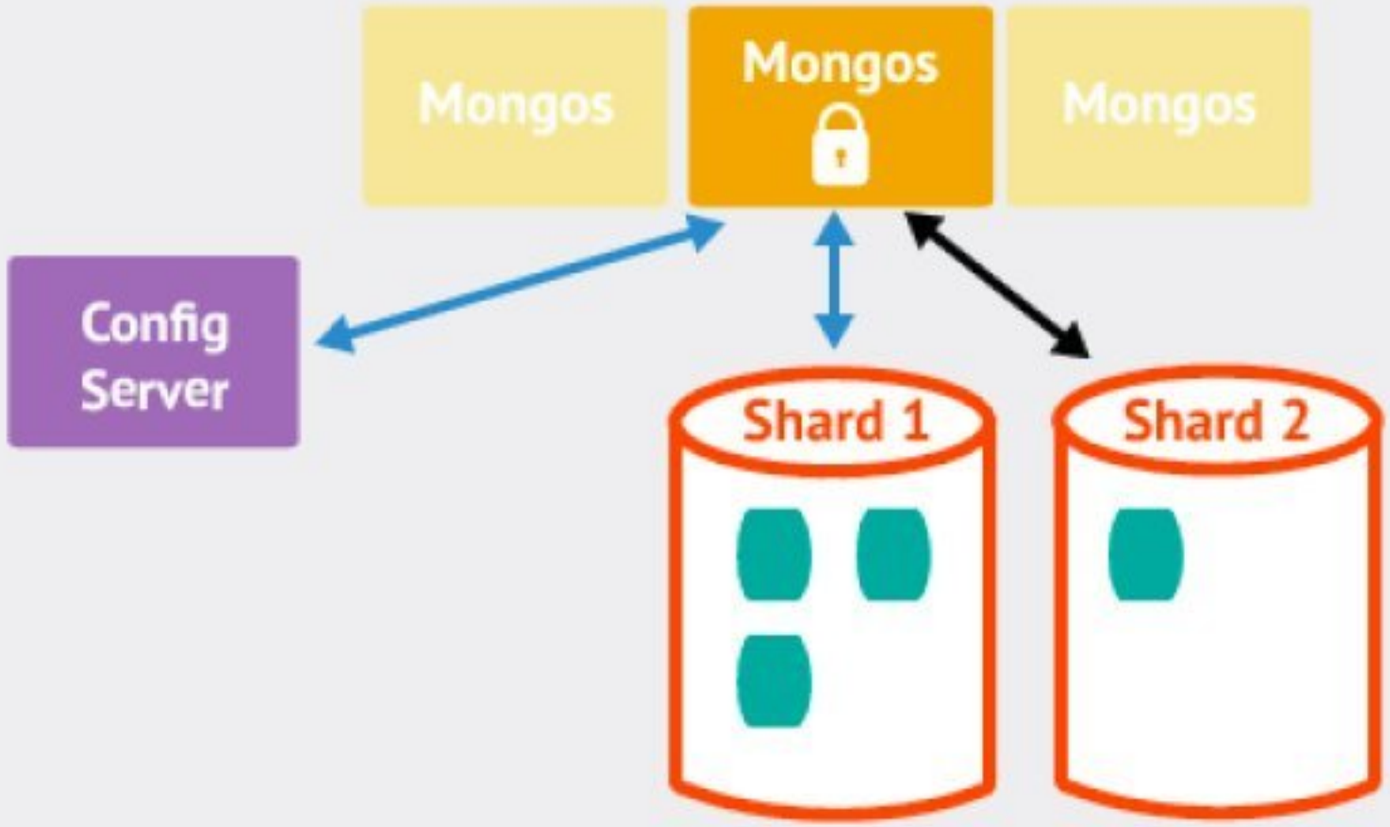
- A chunk is split once it exceeds the maximum size
- There is no split point if all documents have the same shard key
- Chunk split is a logical operation (no data is moved)
- If splitting creates too large a difference in chunks across the cluster, a balancing round starts

Balancing



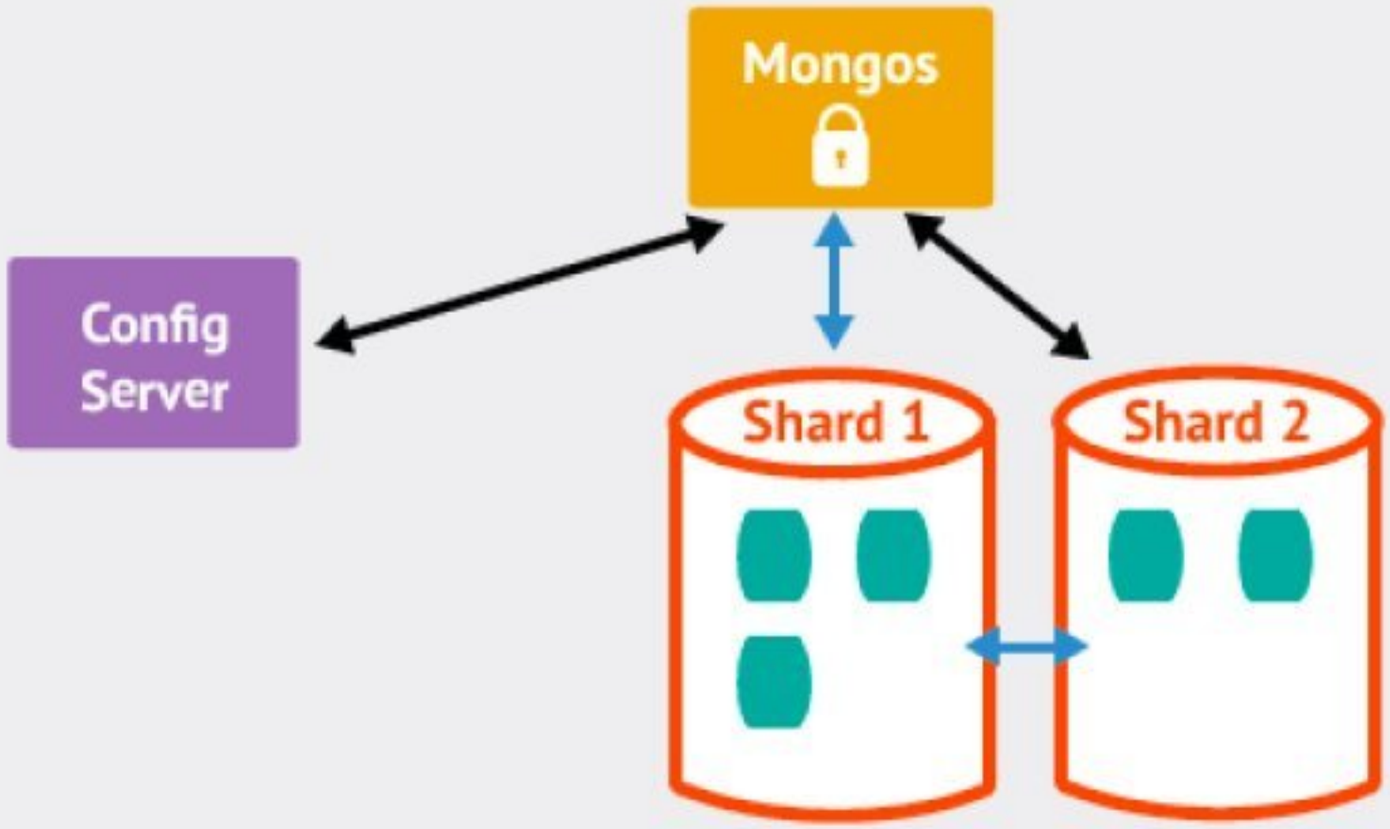
- Balancer is running on mongos
- Once the difference in chunks between the most dense shard and the least dense shard is above the migration threshold, a balancing round starts

Acquiring the balancer lock



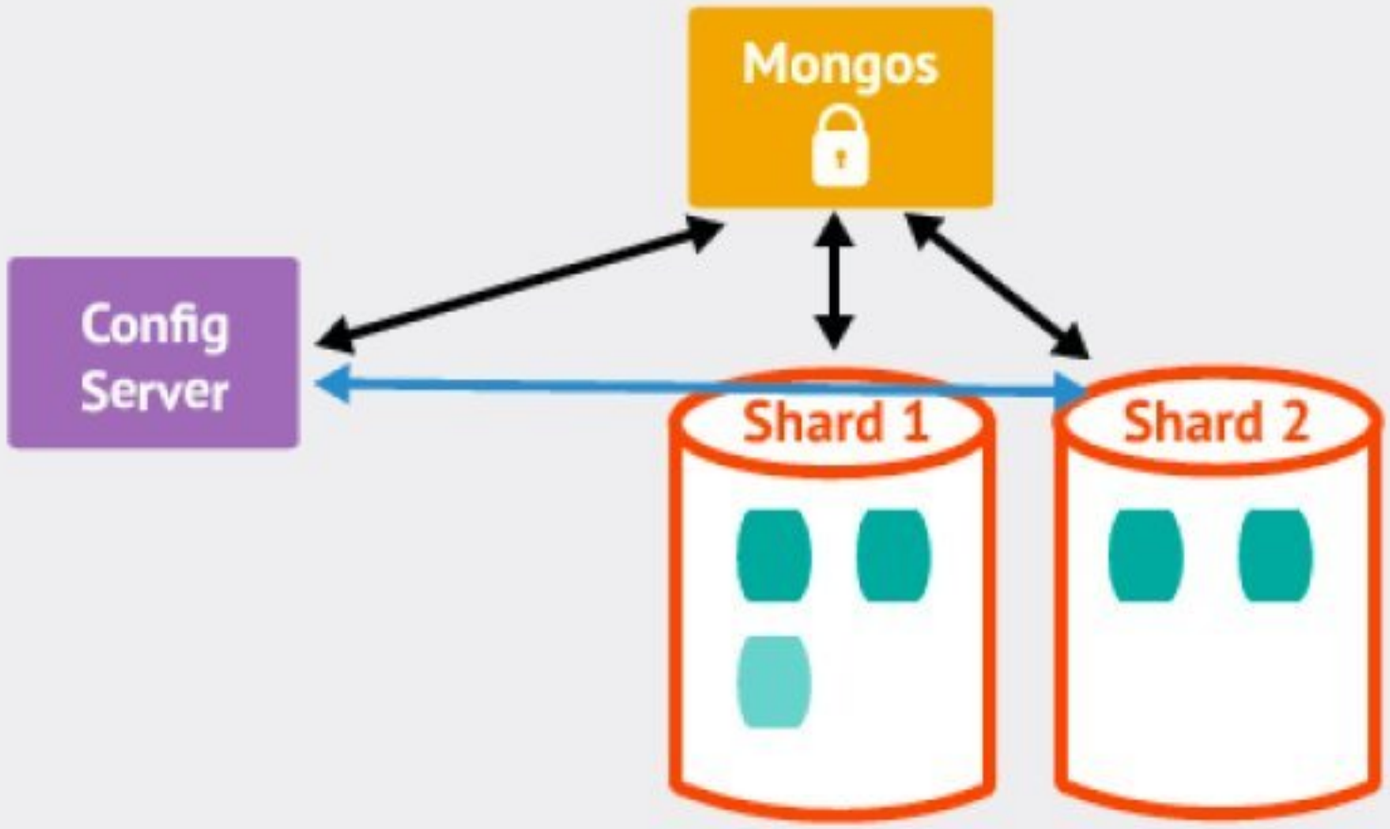
- The balancer on mongos takes out a "balancer" lock
- To see the status of this lock:
  - `use config`
  - `db.locks.find({ _id: "balancer" })`

Moving the chunk



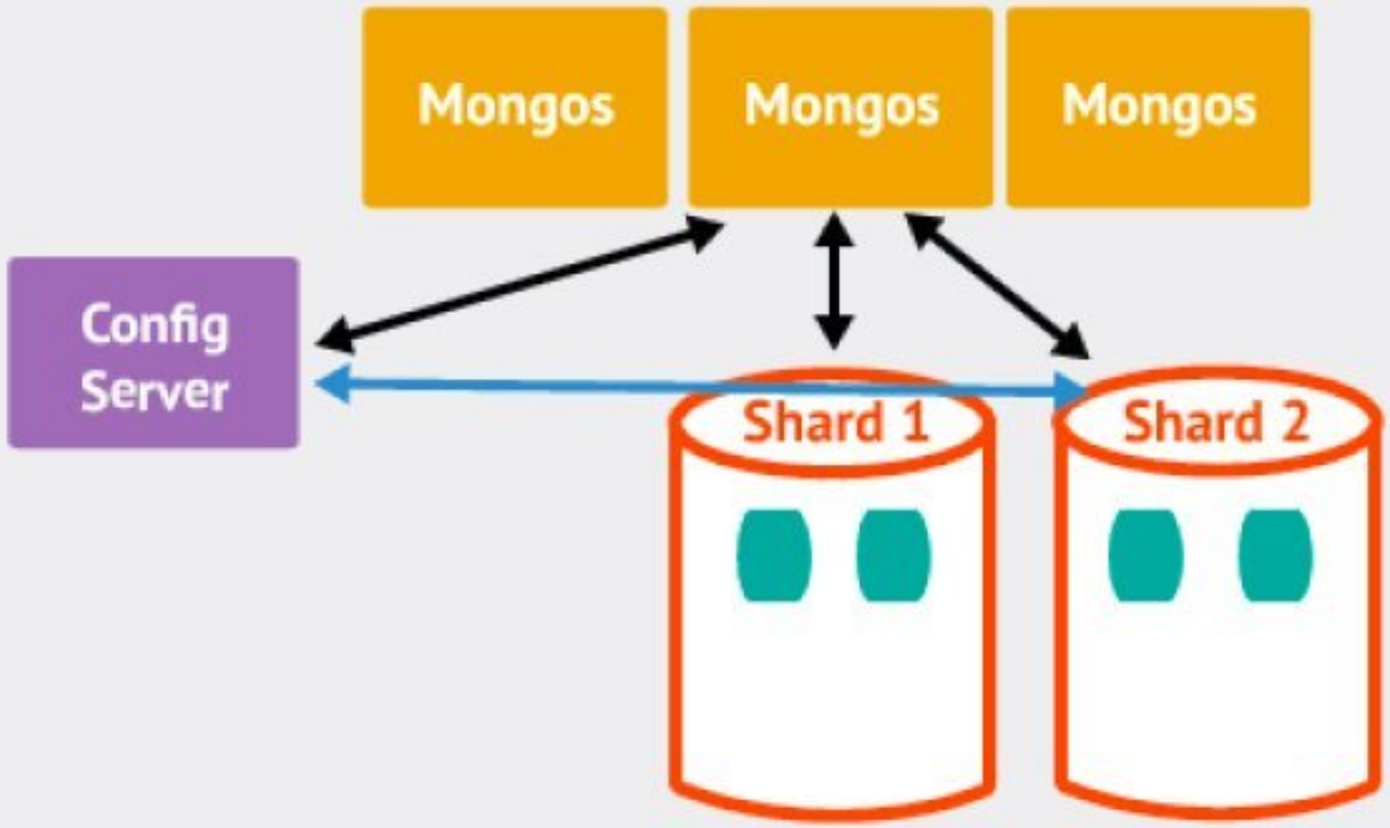
- The mongos sends a "moveChunk" command to source shard
- The source shard then notifies destination shard
- The destination claims the chunk's shard key range

Committing Migration



- When complete, destination shard updates config server
- Provides new locations of the chunks

Cleanup



- Source shard deletes moved data
  - Must wait for open cursors to either close or time out
- Mongos releases balancer lock after old chunks are deleted

Balancing tips

- Run the balancer during low traffic periods:

```
use config;

db.settings.update(
        { _id: 'balancer' },
        { $set: { activeWindow: { start: "23:00", stop: "4:00" } } }
);
```

- Can be triggered manually using `moveChunk`
- Pre-creating ranges for heavy insertion loads can avoid over-populating a single shard
- Feel free to write your own smart balancer!

Routing

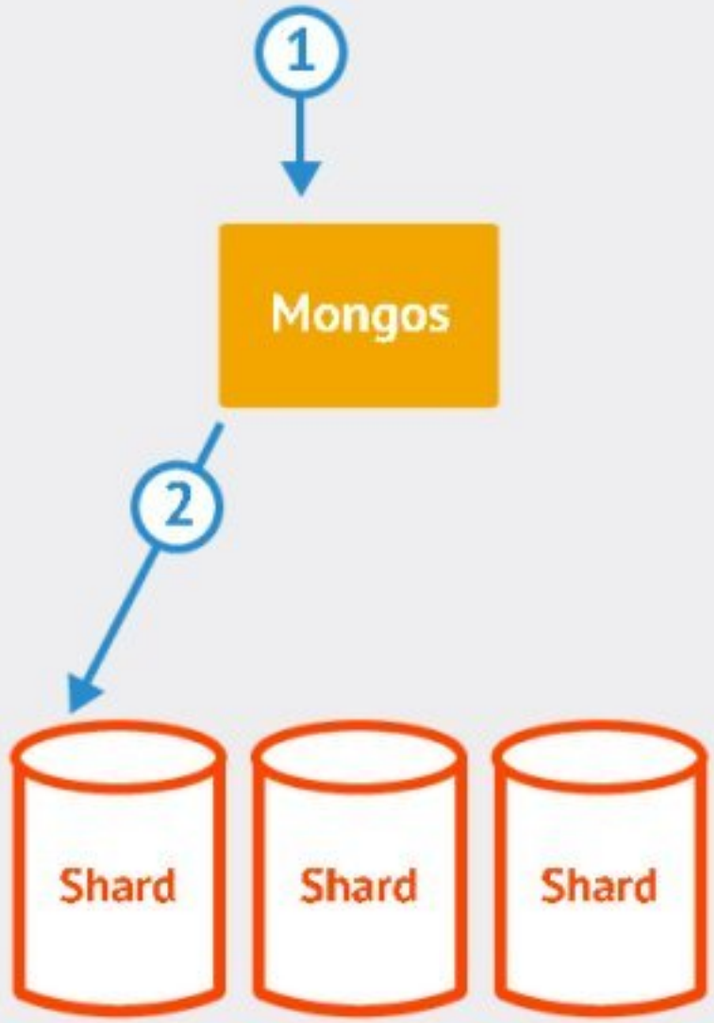
Cluster Request Routing

- Targeted Queries
- Scatter / Gather Queries
- Scatter / Gather Queries with Sort

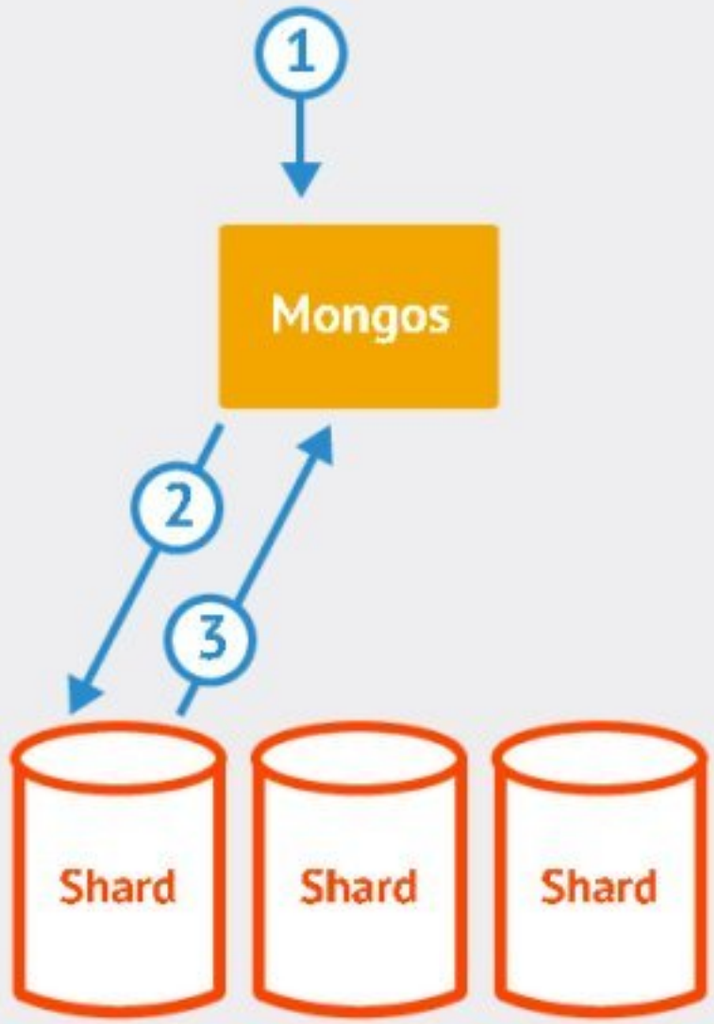
Mongos

Cluster Request Routing: Targeted Query

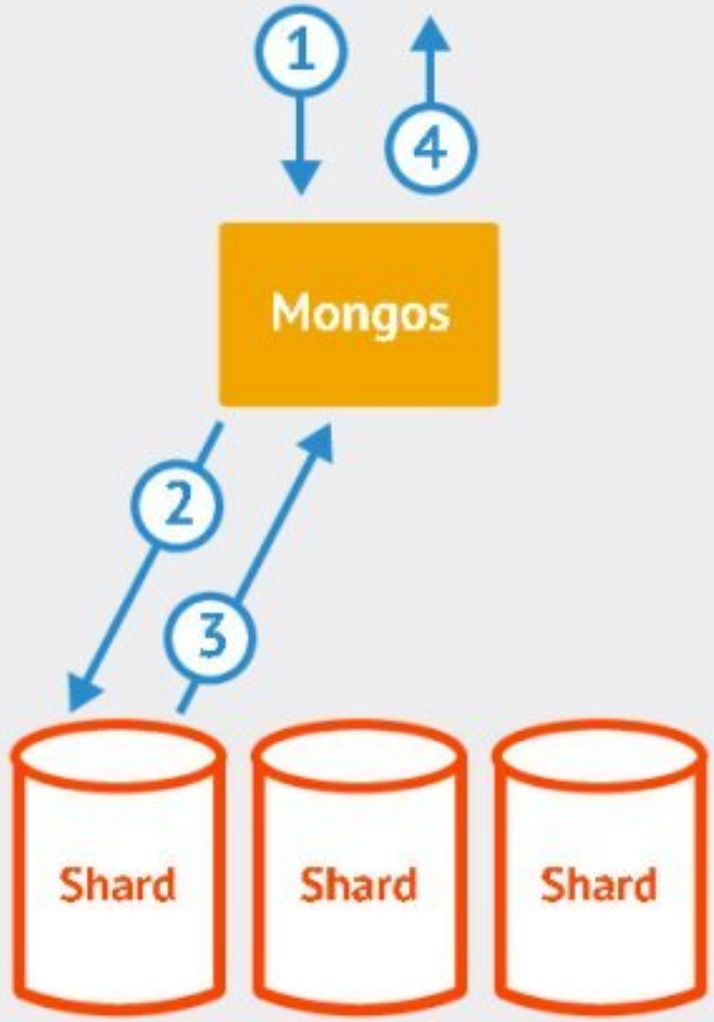
Routable request received



Request routed to appropriate shard



Shard returns results

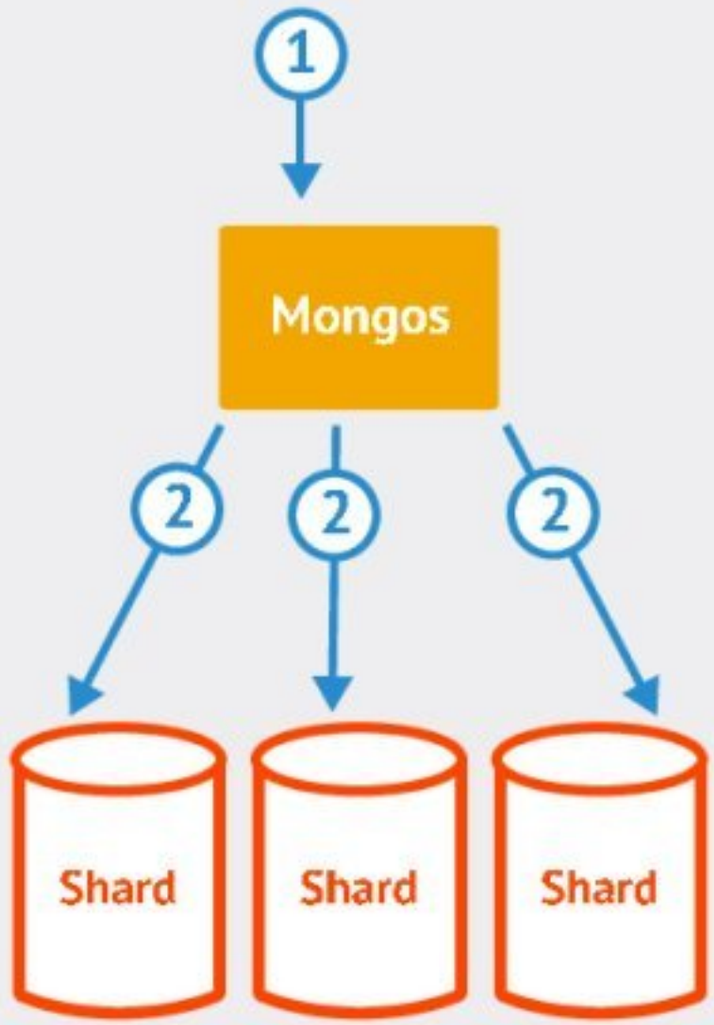


Mongos returns results to client

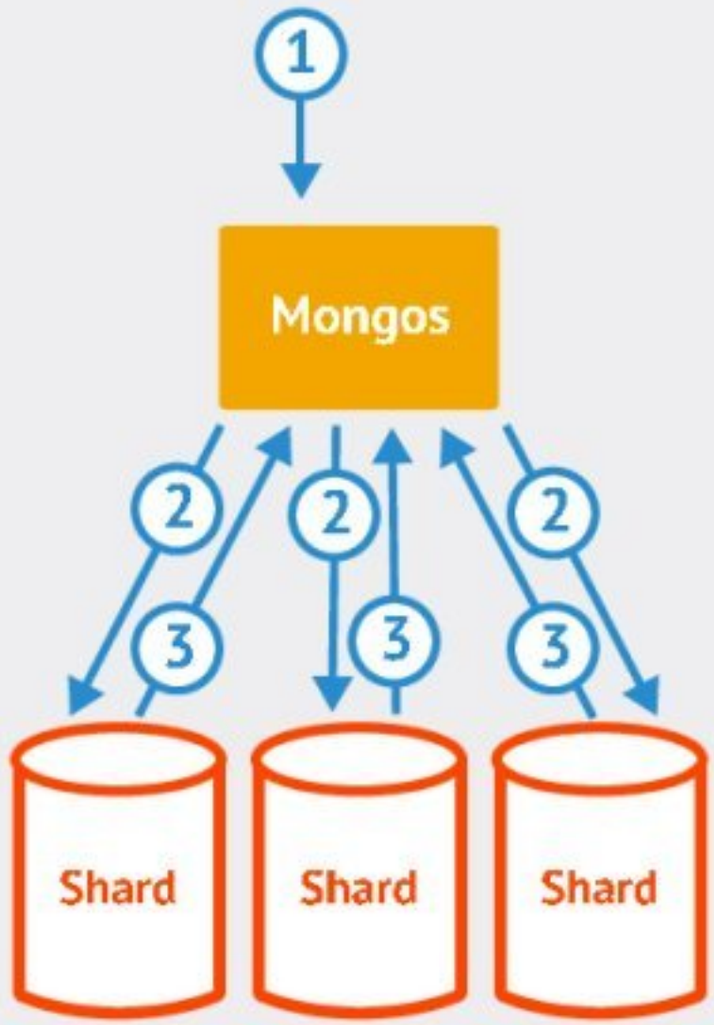
Mongos

Cluster Request Routing: Scatter / Gather Query

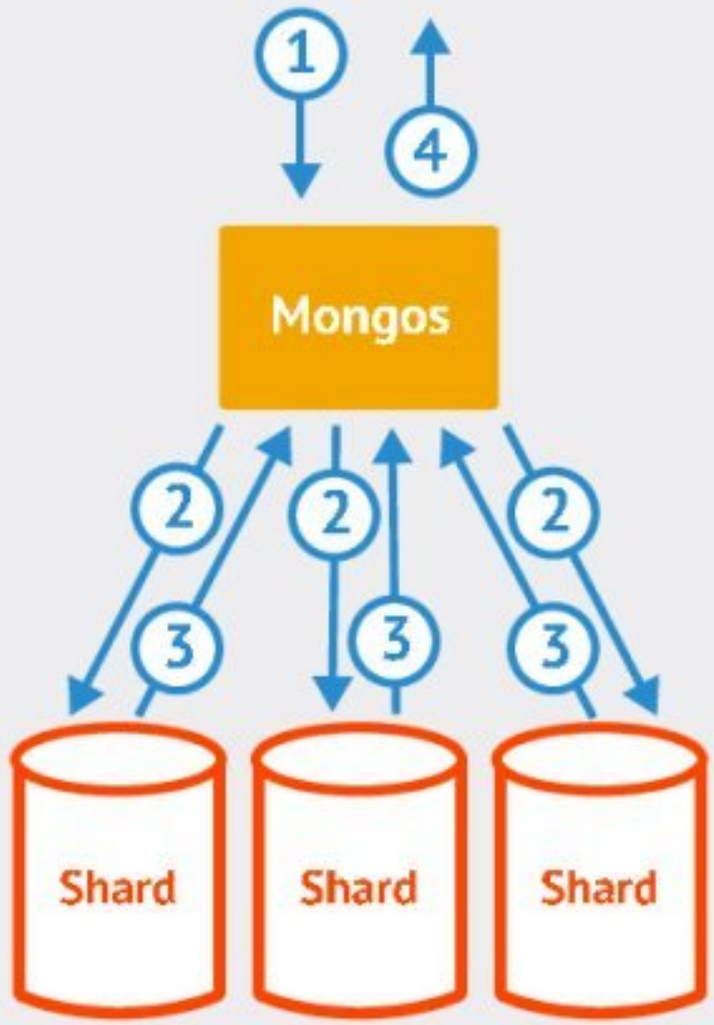
Scatter / Gather Request Received



Request sent to all shards



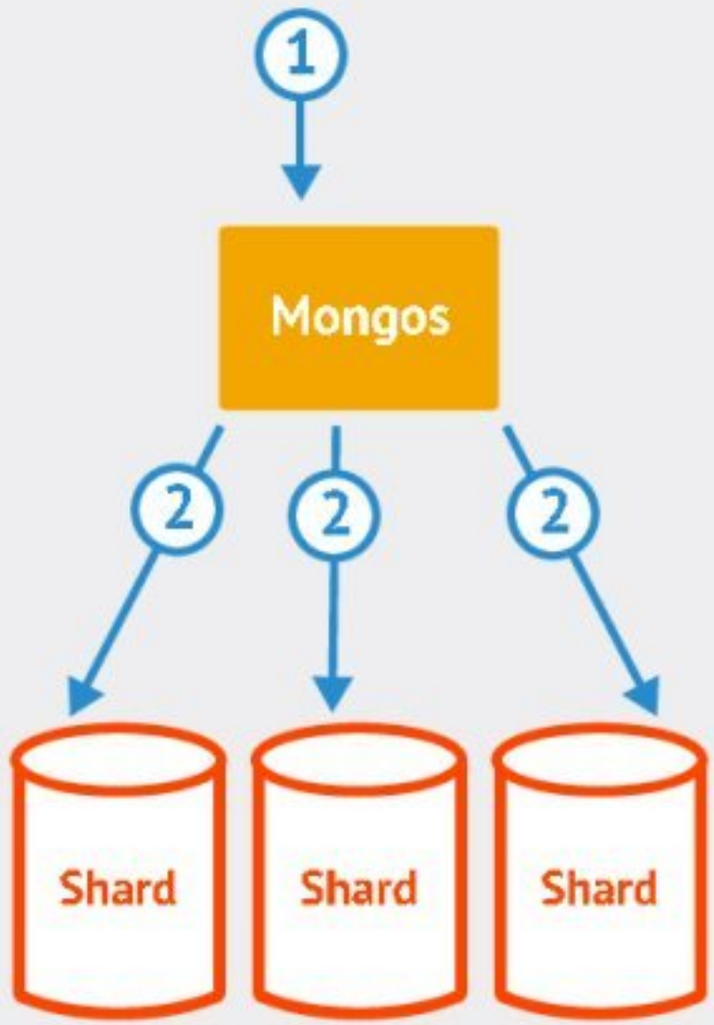
Shards return results to mongos



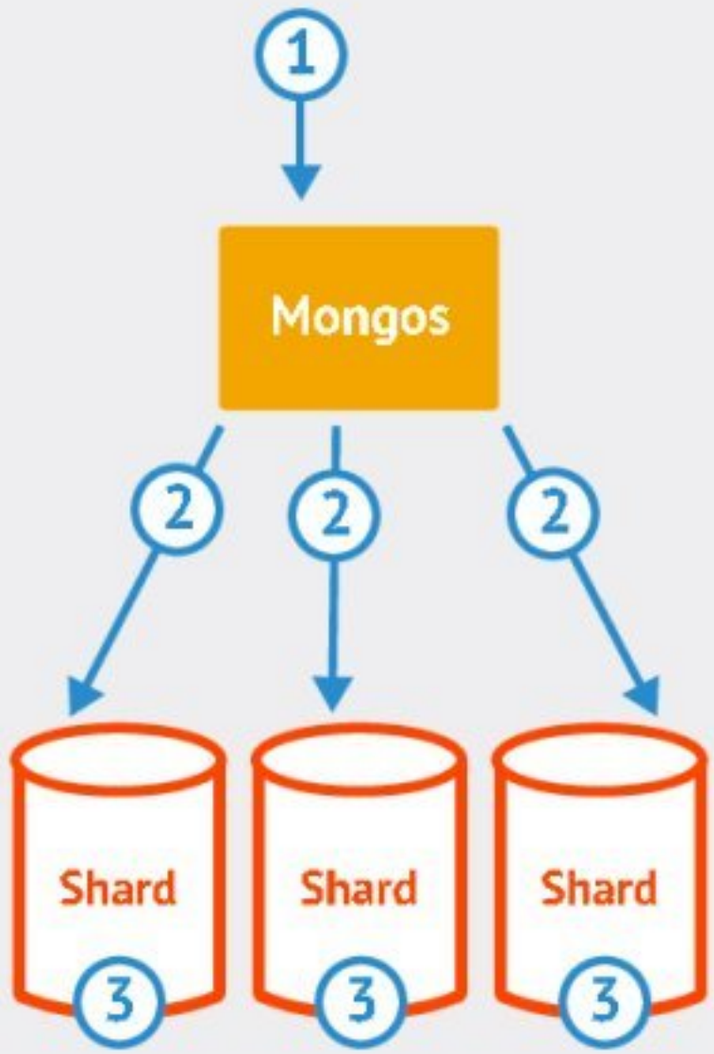
Mongos returns results to client

Cluster Request Routing: Scatter / Gather Query with Sort

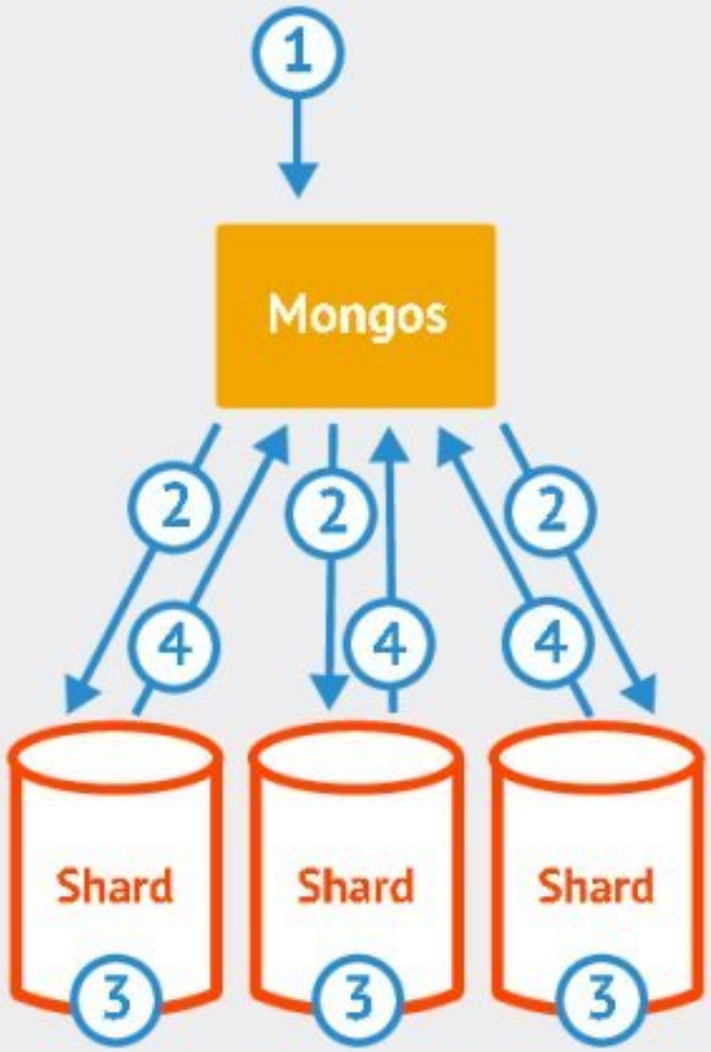
Scatter / Gather Request with Sort Received



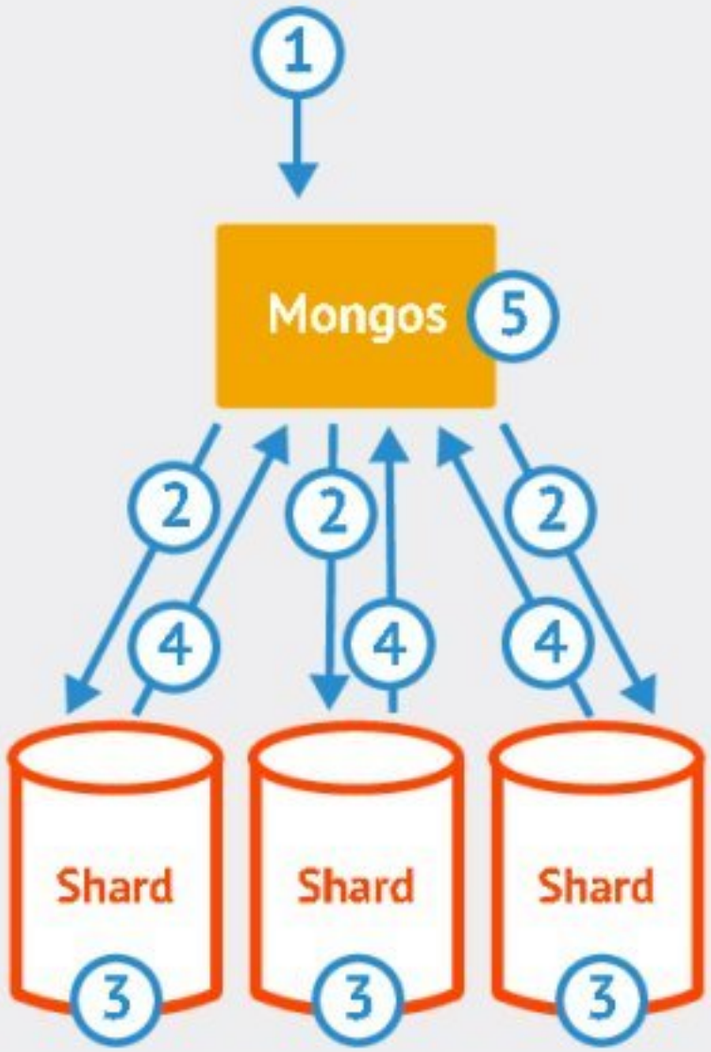
Request sent to all shards



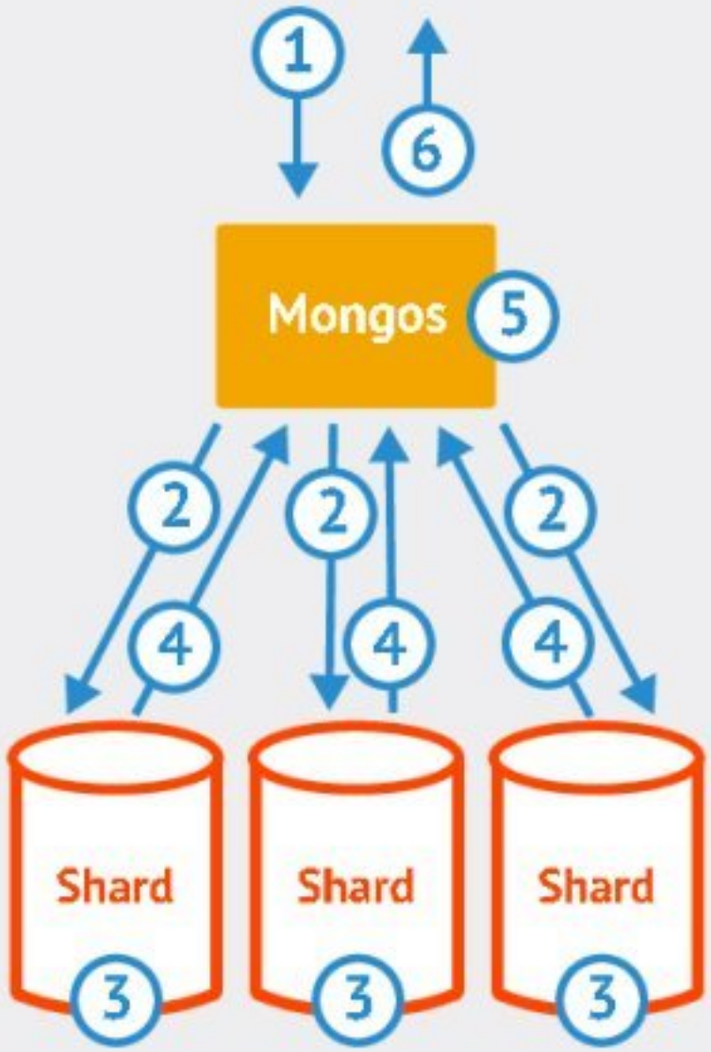
Query and sort performed locally



Shards return results to mongos



Mongos merges sorted results



Mongos returns results to client

Tag Aware Sharding

Tagging a shard

```
sh.addShardTag("shard0000", "ASIA");
sh.addShardTag("shard0001", "ASIA");
sh.addShardTag("shard0002", "ASIA");
sh.addShardTag("shard0003", "US");
sh.addShardTag("shard0004", "AUS");
```

- The same tag can be used with multiple shards
- A shard can have more than one tag

Tag a Shard Key Range

- Define ranges based on the **shard key**
- Any given range may only have **one** assigned tag

Example:

```
sh.addTagRange(
        "demo.emails",                                          // database.collection
        { continent: "Asia" }, { continent: "Australia" }, // range (min, max)
        "ASIA"                                                  // tag
);
sh.addTagRange(
        "demo.emails",
        { continent: "Europe" }, { continent: "Europe" },
        "EUR"
);
```

Any questions?

https://joind.in/13776

derick@mongodb.com & jmikola@mongodb.com—@derickr & @jmikola