A. Im, G. Cai, H. Tunc, J. Stevens, Y. Barve, S. Hei
Vanderbilt University

Content

- Part 1: Introduction & Basics
- 2: CRUD
- 3: Schema Design
- 4: Indexes
- 5: Aggregation
- 6: Replication & Sharding

History

- mongoDB = "Hu**mongo**us DB"
  - Open-source
  - Document-based
  - "High performance, high availability"
  - Automatic scaling
  - C-P on CAP

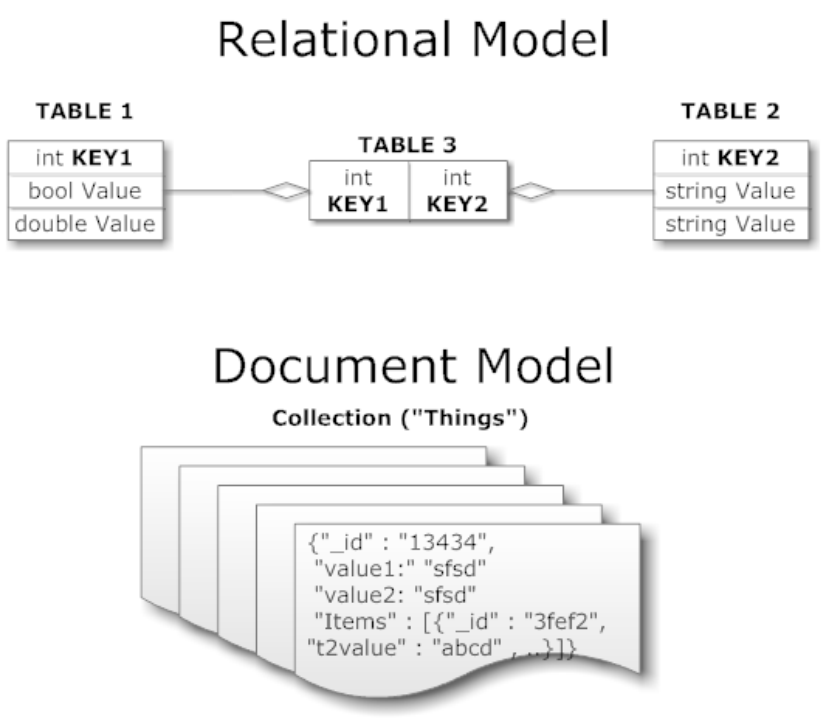
-blog.mongodb.org/post/475279604/on-distributed-consistency-part-1
-mongodb.org/manual

Other NoSQL Types

Key/value (Dynamo)

Columnar/tabular (HBase)

Document (mongoDB)



http://www.aaronstannard.com/post/2011/06/30/MongoDB-vs-SQL-Server.aspx

Motivations

- Problems with SQL
  - Rigid schema
  - Not easily scalable (designed for 90's technology or worse)
  - Requires unintuitive joins
- Perks of mongoDB
  - Easy interface with common languages (Java, Javascript, PHP, etc.)
  - DB tech should run anywhere (VM's, cloud, etc.)
  - Keeps essential features of RDBMS's while learning from key-value noSQL systems

http://www.slideshare.net/spf13/mongodb-9794741?v=qf1&b=&from_search=13

In Good Company

mongoDB

-Steve Francia, http://www.slideshare.net/spf13/mongodb-9794741?v=qf1&b=&from_search=13

Data Model

- Document-Based (max 16 MB)
- Documents are in BSON format, consisting of field-value pairs
- Each document stored in a collection
- Collections
  - Have index set in common
  - Like tables of relational db's.
  - Documents do not have to have uniform structure

-docs.mongodb.org/manual/

JSON

- "JavaScript Object Notation"
- Easy for humans to write/read, easy for computers to parse/generate
- Objects can be nested
- Built on
  - name/value pairs
  - Ordered list of values

http://json.org/

BSON

- "Binary JSON"
- Binary-encoded serialization of JSON-like docs
- Also allows "referencing"
- Embedded structure reduces need for joins
- Goals
  - Lightweight
  - Traversable
  - Efficient (decoding and encoding)

http://bsonspec.org/

BSON Example

```
{
"_id" :     "37010"
"city" :    "ADAMS",
"pop" :     2660,
"state" :   "TN",
“councilman” : {
                name: “John Smith”
                address: “13 Scenic Way”
              }
}
```

BSON Types

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

The number can be used with the $type operator to query by type!

http://docs.mongodb.org/manual/reference/bson-types/

The _id Field

- By default, each document contains an _id field. This field has a number of special characteristics:
  - Value serves as primary key for collection.
  - Value is unique, immutable, and may be any non-array type.
  - Default data type is ObjectId, which is "small, likely unique, fast to generate, and ordered." Sorting on an ObjectId value is roughly equivalent to sorting on creation time.

http://docs.mongodb.org/manual/reference/bson-types/

mongoDB vs. SQL

mongoDB	SQL
Document	Tuple
Collection	Table/View
PK: _id Field	PK: Any Attribute(s)
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Structure	Joins
Shard	Partition

CRUD

Create, Read, Update, Delete

Getting Started with mongoDB

Open your mongodb/bin directory and run mongod.exe to start the database server.

To establish a connection to the server, open another command prompt window and go to the same directory, entering in mongo.exe. This engages the mongodb shell—it's that easy!

http://docs.mongodb.org/manual/tutorial/getting-started/

CRUD: Using the Shell

To insert documents into a collection/make a new collection:

```
db.<collection>.insert(<document>)
```

<=>

```
INSERT INTO <table>
VALUES(<attributevalues>);
```

CRUD: Inserting Data

Insert one document

```
db.<collection>.insert({<field>:<value>})
```

Inserting a document with a field name new to the collection is inherently supported by the BSON model.

To insert multiple documents, use an array.

CRUD: Querying

- Done on collections.
- Get all docs: db.<collection>.find()
  - Returns a cursor, which is iterated over shell to display first 20 results.
  - Add .limit(<number>) to limit results
  - SELECT * FROM <table>;
- Get one doc: db.<collection>.findOne()

CRUD: Querying

To match a specific value:

```
db.<collection>.find({<field>:<value>})
"AND"
db.<collection>.find({<field1>:<value1>,
                      <field2>:<value2>
                      })
```

```
SELECT *
FROM <table>
WHERE <field1> = <value1> AND <field2> =
<value2>;
```

CRUD: Querying

OR

```
db.<collection>.find({ $or: [
<field>:<value1>
<field>:<value2>        ]
})
```

```
SELECT *
FROM <table>
WHERE <field> = <value1> OR <field> = <value2>;
```

Checking for multiple values of same field

```
db.<collection>.find({<field>: {$in [<value>, <value>]}})
```

CRUD: Querying

Including/excluding document fields

```
db.<collection>.find({<field1>:<value>}, {<field2>: 0})
```

```
SELECT field1
FROM <table>;
```

```
db.<collection>.find({<field>:<value>}, {<field2>: 1})
```

Find documents with or w/o field

```
db.<collection>.find({<field>: { $exists: true}})
```

CRUD: Updating

```
db.<collection>.update(
{<field1>:<value1>},      //all docs in which field = value
{$set: {<field2>:<value2>}},      //set field to value
{multi:true} )            //update multiple docs
```

upsert: if true, creates a new doc when none matches search criteria.

```
UPDATE <table>
SET <field2> = <value2>
WHERE <field1> = <value1>;
```

CRUD: Updating

To remove a field

```
db.<collection>.update({<field>:<value>},
{ $unset: { <field>: 1}})
```

Replace all field-value pairs

```
db.<collection>.update({<field>:<value>},
{ <field>:<value>,
  <field>:<value>})
```

*NOTE: This overwrites ALL the contents of a document, even removing fields.

CRUD: Removal

Remove all records where field = value

```
db.<collection>.remove({<field>:<value>})
```

```
DELETE FROM <table>
WHERE <field> = <value>;
```

As above, but only remove first document

```
db.<collection>.remove({<field>:<value>}, true)
```

CRUD: Isolation

- By default, all writes are atomic **only** on the level of a single document.
- This means that, by default, all writes can be interleaved with other operations.
- You can isolate writes on an **unsharded** collection by adding $isolated:1 in the query area:

```
db.<collection>.remove({<field>:<value>,
                        $isolated: 1})
```

Schema Design

RDBMS		MongoDB
Database	→	Database
Table	→	Collection
Row	→	Document
Index	→	Index
Join	→	Embedded Document
Foreign Key	→	Reference

Intuition – why database exist in the first place?

- Why can't we just write programs that operate on objects?
  - Memory limit
  - We cannot swap back from disk merely by OS for the page based memory management mechanism
- Why can't we have the database operating on the same data structure as in program?
  - That is where mongoDB comes in

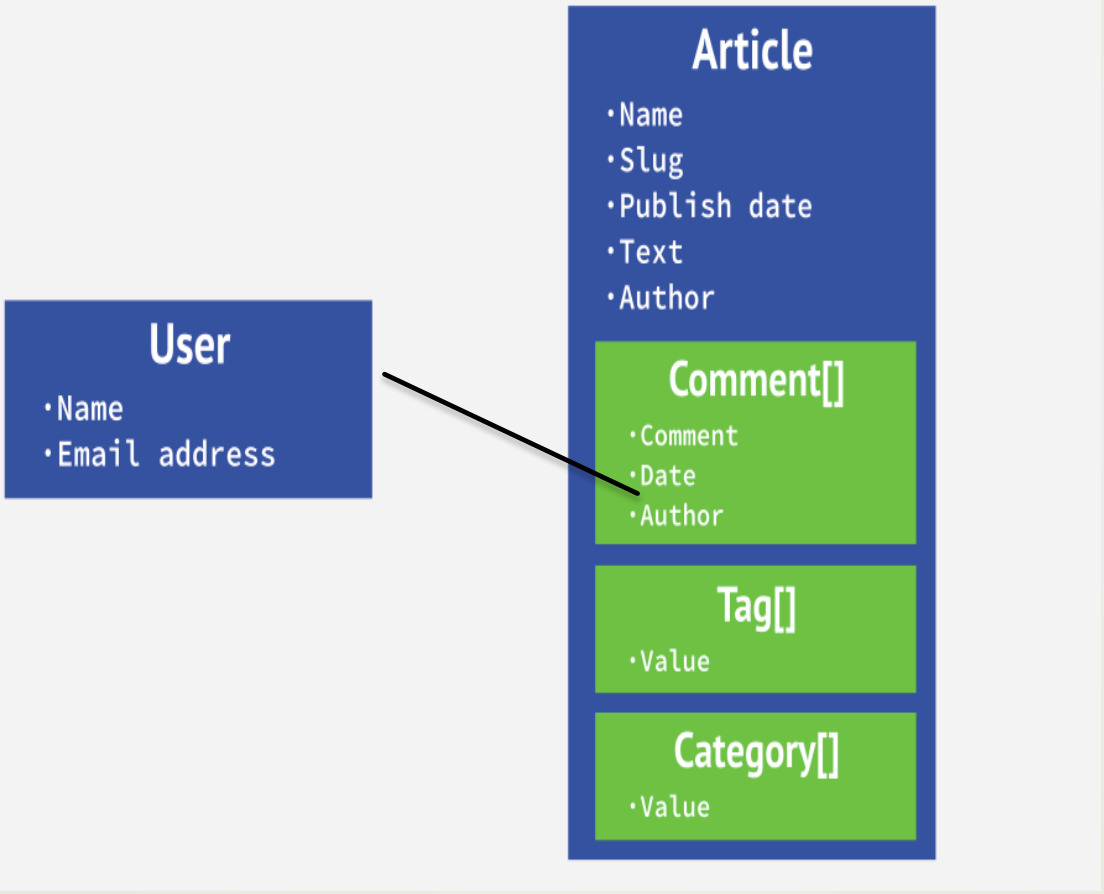
Mongo is basically schema-free

- The purpose of schema in SQL is for meeting the requirements of tables and quirky SQL implementation
- Every "*row*" in a database "*table*" is a data structure, much like a "struct" in C, or a "class" in Java. A table is then an array (or list) of such data structures
- So we what we design in mongoDB is basically same way how we design a compound data type binding in JSON

There are some patterns

- Embedding
- Linking

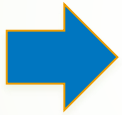
Embedding & Linking



One to One relationship

```
zip = {
 _id: 35004,
city: “ACMAR”,
 loc: [-86, 33],
pop: 6065,
State: “AL”
}

Council_person = {
zip_id = 35004,
name: “John Doe",
address: “123 Fake St.”,
Phone: 123456
}
```



```
zip = {
 _id: 35004 ,
city: “ACMAR”
 loc: [-86, 33],
 pop: 6065,
 State: “AL”,

 council_person: {
 name: “John Doe",
 address: “123 Fake St.”,
 Phone: 123456
}
}
```

Example 2

MongoDB: The Definitive Guide,

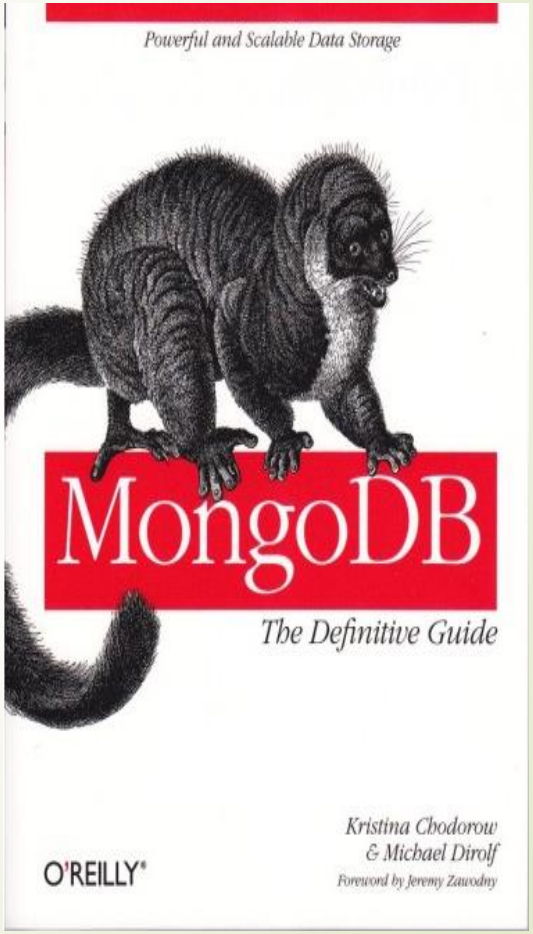
By Kristina Chodorow and Mike Dirolf

Published: 9/24/2010

Pages: 216

Language: English

Publisher: O'Reilly Media, CA



One to many relationship - Embedding

```
book = {
title: "MongoDB: The Definitive Guide",
authors: [ "Kristina Chodorow", "Mike Dirolf" ]
published_date: ISODate("2010-09-24"),
pages: 216,
language: "English",
publisher: {
name: "O'Reilly Media",
founded: "1980",
location: "CA"
}
}
```

One to many relationship – Linking

```
publisher = {
    _id: "oreilly",
    name: "O'Reilly Media",
    founded: "1980",
    location: "CA"
}
book = {
    title: "MongoDB: The Definitive Guide",
    authors: [ "Kristina Chodorow", "Mike Dirolf" ]
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher_id: "oreilly"
}
```

Linking vs. Embedding

- Embedding is a bit like pre-joining data
- Document level operations are easy for the server to handle
- Embed when the “many” objects always appear with (viewed in the context of) their parents.
- Linking when you need more flexibility

Many to many relationship

- Can put relation in either one of the documents (embedding in one of the documents)
- Focus how data is accessed queried

Example

```
book = {
    title: "MongoDB: The Definitive Guide",
    authors : [
        { _id: "kchodorow", name: "Kristina Chodorow" },
        { _id: "mdirolf", name: "Mike Dirolf" }
    ]
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

author = {
    _id: "kchodorow",
    name: "Kristina Chodorow",
    hometown: "New York"
}

db.books.find( { authors.name : "Kristina Chodorow" } )
```

What is bad about SQL (semantically)

- "Primary keys" of a database table are in essence persistent memory addresses for the object. The address may not be the same when the object is reloaded into memory. This is why we need primary keys.
- Foreign key functions just like a pointer in C, persistently point to the primary key.
- Whenever we need to deference a pointer, we do JOIN
- It is not intuitive for programming and also JOIN is time consuming

Example 3

- Book can be checked out by one student at a time
- Student can check out many books

Modeling Checkouts

```
student = {
    _id: "joe"
    name: "Joe Bookreader",
    join_date: ISODate("2011-10-15"),
    address: { ... }
}

book = {
    _id: "123456789"
    title: "MongoDB: The Definitive Guide",
    authors: [ "Kristina Chodorow", "Mike Dirolf" ],
    ...
}
```

Modeling Checkouts

```
student = {
    _id: "joe"
    name: "Joe Bookreader",
    join_date: ISODate("2011-10-15"),
    address: { ... },
    checked_out: [
        { _id: "123456789", checked_out: "2012-10-15" },
        { _id: "987654321", checked_out: "2012-09-12" },
        ...
    ]
}
```

What is good about mongoDB?

- find() is more semantically clear for programming

```
(map (lambda (b) b.title)
     (filter (lambda (p) (> p 100)) Book)
```

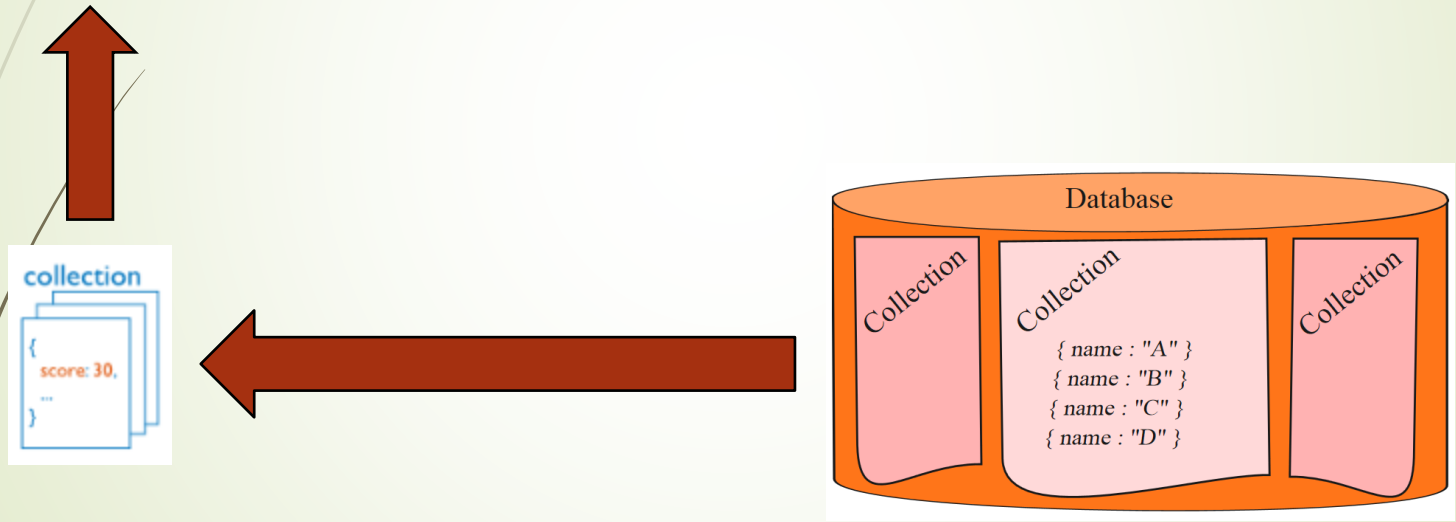
- De-normalization provides **Data locality, and Data locality provides speed**

Part 4: Index in MongoDB

Before Index

- What does database normally do when we query?
  - MongoDB must scan **every** document.
  - Inefficient because process **large volume** of data

```
db.users.find( { score: { “$lt” : 30} } )
```



Definition of Index

- **Definition**
  - **Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.**

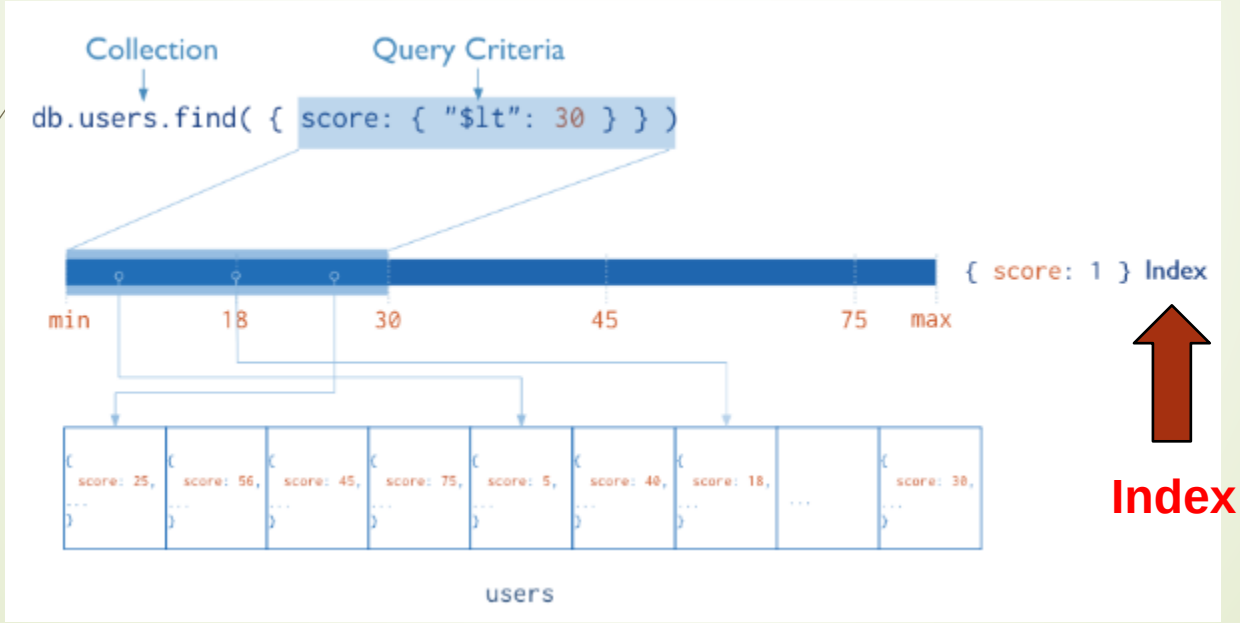


Diagram of a query that uses an index to select

Index in MongoDB

Operations

- **Creation index**
  - db.users.ensureIndex({ score: 1 })
- **Show existing indexes**
  - db.users.getIndexes()
- **Drop index**
  - db.users.dropIndex({score: 1})
- **Explain—Explain**
  - db.users.find().explain()
  - Returns a document that describes the process and indexes
- **Hint**
  - db.users.find().hint({score: 1})
  - Overide MongoDB's default index selection

Index in MongoDB

Types

- **Single Field Indexes**
- **Compound Field Indexes**
- **Multikey Indexes**

- **Single Field Indexes**
  - db.users.ensureIndex({ score: 1 })

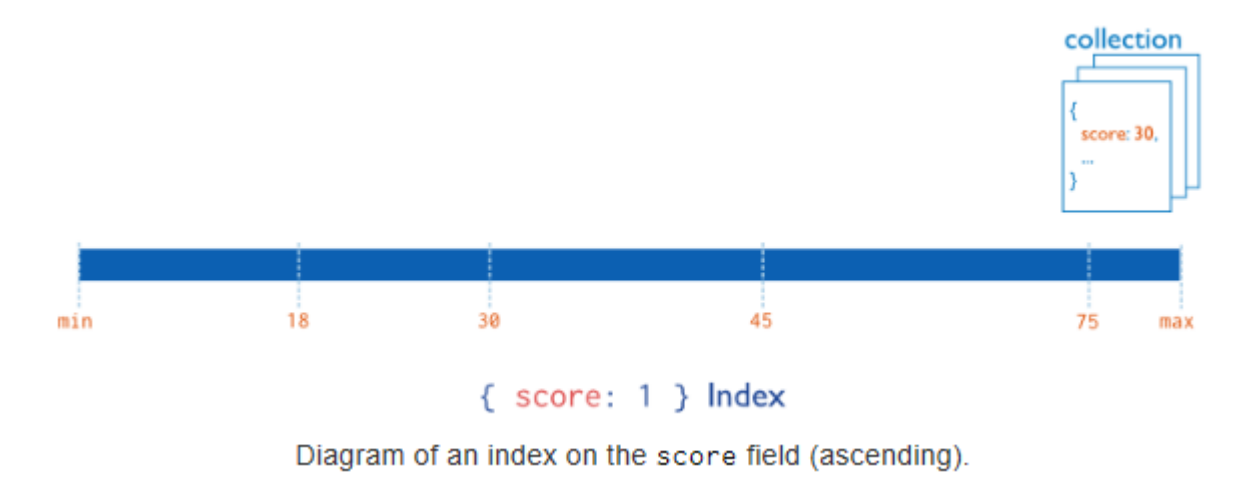


Diagram of an index on the score field (ascending).

Index in MongoDB

Types

- **Single Field Indexes**
- **Compound Field Indexes**
- **Multikey Indexes**

- **Compound Field Indexes**
  - db.users.ensureIndex({ userid:1, score: -1 })

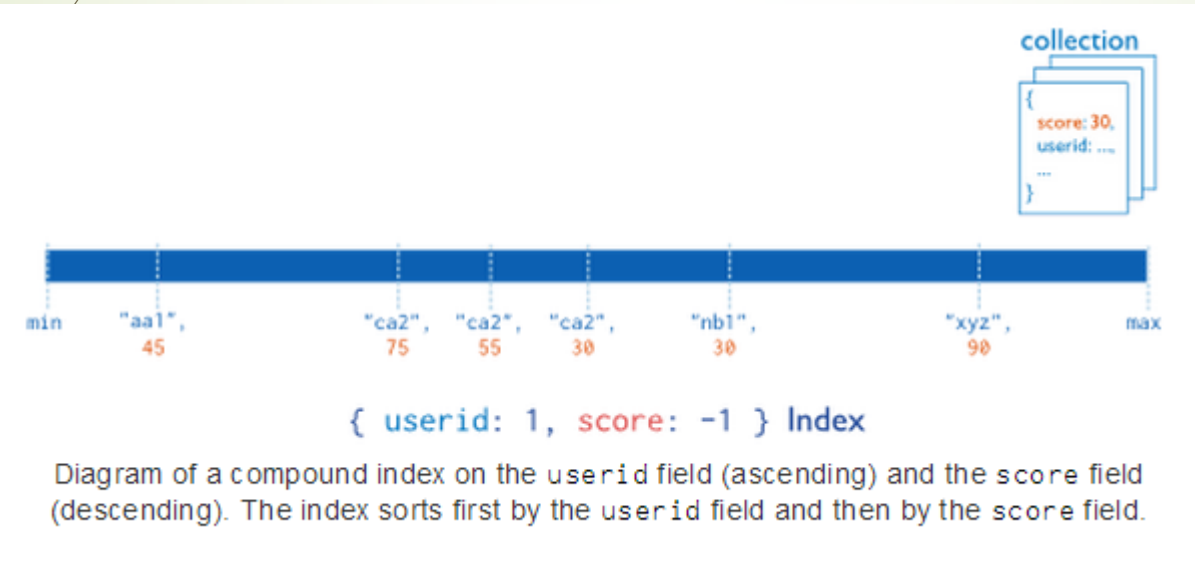


Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

Index in MongoDB

Types

- **Single Field Indexes**
- **Compound Field Indexes**
- **Multikey Indexes**

- **Multikey Indexes**
  - db.users.ensureIndex({ addr.zip:1})

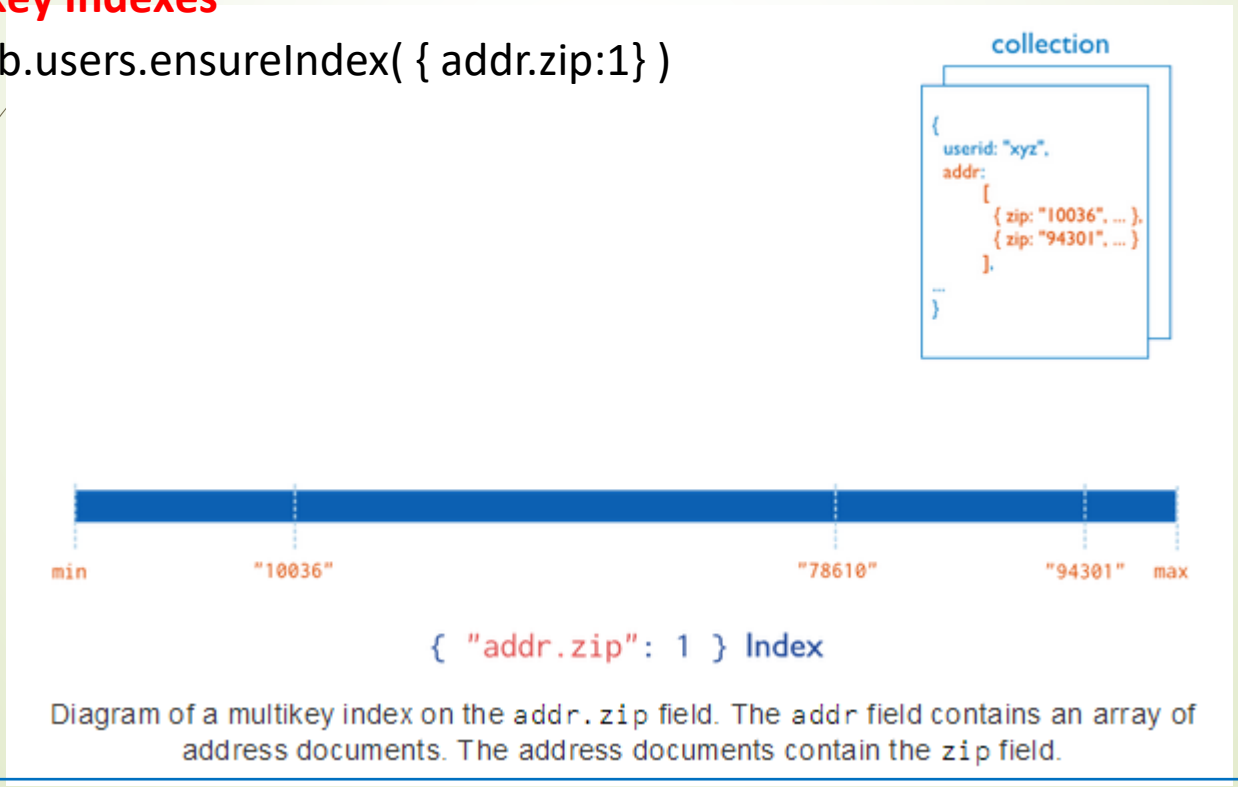


Diagram of a multikey index on the addr.zip field. The addr field contains an array of address documents. The address documents contain the zip field.

Demo of indexes in MongoDB

- Import Data
- Create Index
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Show Existing Index
- Hint
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Explain
- Compare with data without indexes

Demo of indexes in MongoDB

- Import Data
- Create Index
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Show Existing Index
- Hint
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Explain
- Compare with data without indexes

```
db.zips.ensureIndex({pop: -1})
db.zips.ensureIndex({state: 1, city: 1})
db.zips.ensureIndex({loc: -1})
```

Demo of indexes in MongoDB

- Import Data
- Create Index
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- **Show Existing Index**
- Hint
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Explain
- Compare with data without indexes

```
> db.zips.getIndexes()
[
        {
                "v" : 1,
                "key" : {
                        "_id" : 1
                },
                "ns" : "blog.zips",
                "name" : "_id_"
        },
        {
                "v" : 1,
                "key" : {
                        "pop" : 1
                },
                "ns" : "blog.zips",
                "name" : "pop_1"
        },
        {
                "v" : 1,
                "key" : {
                        "state" : 1,
                        "city" : 1
                },
                "ns" : "blog.zips",
                "name" : "state_1_city_1"
        },
        {
                "v" : 1,
                "key" : {
                        "loc" : 1
                },
                "ns" : "blog.zips",
                "name" : "loc_1"
        }
]
```

Demo of indexes in MongoDB

- Import Data
- Create Index
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Show Existing Index
- Hint
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Explain
- Compare with data without indexes

```
> db.zips.find().limit(20).hint({pop: -1})
{ "city" : "CHICAGO", "loc" : [  -87.7157,  41.849015 ], "pop" : 112047, "state" : "IL", "_id" : "60623" }
{ "city" : "BROOKLYN", "loc" : [  -73.956985,  40.646694 ], "pop" : 111396, "state" : "NY", "_id" : "11226" }
{ "city" : "NEW YORK", "loc" : [  -73.958805,  40.768476 ], "pop" : 106564, "state" : "NY", "_id" : "10021" }
{ "city" : "NEW YORK", "loc" : [  -73.968312,  40.797466 ], "pop" : 100027, "state" : "NY", "_id" : "10025" }
{ "city" : "BELL GARDENS", "loc" : [  -118.17205,  33.969177 ], "pop" : 99568, "state" : "CA", "_id" : "90201" }
{ "city" : "CHICAGO", "loc" : [  -87.556012,  41.725743 ], "pop" : 98612, "state" : "IL", "_id" : "60617" }
{ "city" : "LOS ANGELES", "loc" : [  -118.258189,  34.007856 ], "pop" : 96074, "state" : "CA", "_id" : "90011" }
{ "city" : "CHICAGO", "loc" : [  -87.704322,  41.920903 ], "pop" : 95971, "state" : "IL", "_id" : "60647" }
{ "city" : "CHICAGO", "loc" : [  -87.624277,  41.693443 ], "pop" : 94317, "state" : "IL", "_id" : "60628" }
{ "city" : "NORWALK", "loc" : [  -118.081767,  33.90564 ], "pop" : 94188, "state" : "CA", "_id" : "90650" }
{ "city" : "CHICAGO", "loc" : [  -87.654251,  41.741119 ], "pop" : 92005, "state" : "IL", "_id" : "60620" }
{ "city" : "CHICAGO", "loc" : [  -87.706936,  41.778149 ], "pop" : 91814, "state" : "IL", "_id" : "60629" }
{ "city" : "CHICAGO", "loc" : [  -87.653279,  41.809721 ], "pop" : 89762, "state" : "IL", "_id" : "60609" }
{ "city" : "CHICAGO", "loc" : [  -87.704214,  41.946401 ], "pop" : 88377, "state" : "IL", "_id" : "60618" }
{ "city" : "JACKSON HEIGHTS", "loc" : [  -73.878551,  40.740388 ], "pop" : 88241, "state" : "NY", "_id" : "11373" }
{ "city" : "ARLETA", "loc" : [  -118.420692,  34.258081 ], "pop" : 88114, "state" : "CA", "_id" : "91331" }
{ "city" : "BROOKLYN", "loc" : [  -73.914483,  40.662474 ], "pop" : 87079, "state" : "NY", "_id" : "11212" }
{ "city" : "SOUTH GATE", "loc" : [  -118.201349,  33.94617 ], "pop" : 87026, "state" : "CA", "_id" : "90280" }
{ "city" : "RIDGEWOOD", "loc" : [  -73.896122,  40.703613 ], "pop" : 85732, "state" : "NY", "_id" : "11385" }
{ "city" : "BRONX", "loc" : [  -73.871242,  40.873671 ], "pop" : 85710, "state" : "NY", "_id" : "10467" }
```

Demo of indexes in MongoDB

- Import Data
- Create Index
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Show Existing Index
- Hint
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Explain
- Compare with data without indexes

```
> db.zips.find().limit(20).hint({state: 1, city: 1})
{ "city" : "98791", "loc" : [  -176.310048,  51.938901 ], "pop" : 5345, "state" : "AK", "_id" : "98791" }
{ "city" : "AKHIOK", "loc" : [  -152.500169,  57.781967 ], "pop" : 13309, "state" : "AK", "_id" : "99615" }
{ "city" : "AKIACHAK", "loc" : [  -161.39233,  60.891854 ], "pop" : 481, "state" : "AK", "_id" : "99551" }
{ "city" : "AKIAK", "loc" : [  -161.199325,  60.890632 ], "pop" : 285, "state" : "AK", "_id" : "99552" }
{ "city" : "AKUTAN", "loc" : [  -165.785368,  54.143012 ], "pop" : 589, "state" : "AK", "_id" : "99553" }
{ "city" : "ALAKANUK", "loc" : [  -164.60228,  62.746967 ], "pop" : 1186, "state" : "AK", "_id" : "99554" }
{ "city" : "ALEKNAGIK", "loc" : [  -158.619882,  59.269688 ], "pop" : 185, "state" : "AK", "_id" : "99555" }
{ "city" : "ALLAKAKET", "loc" : [  -152.712155,  66.543197 ], "pop" : 170, "state" : "AK", "_id" : "99720" }
{ "city" : "AMBLER", "loc" : [  -156.455652,  67.46951 ], "pop" : 8, "state" : "AK", "_id" : "99786" }
{ "city" : "ANAKTUVUK PASS", "loc" : [  -151.679005,  68.11878 ], "pop" : 260, "state" : "AK", "_id" : "99721" }
{ "city" : "ANCHORAGE", "loc" : [  -149.876077,  61.211571 ], "pop" : 14436, "state" : "AK", "_id" : "99501" }
{ "city" : "ANCHORAGE", "loc" : [  -150.093943,  61.096163 ], "pop" : 15891, "state" : "AK", "_id" : "99502" }
{ "city" : "ANCHORAGE", "loc" : [  -149.893844,  61.189953 ], "pop" : 12534, "state" : "AK", "_id" : "99503" }
{ "city" : "ANCHORAGE", "loc" : [  -149.74467,  61.203696 ], "pop" : 32383, "state" : "AK", "_id" : "99504" }
{ "city" : "ANCHORAGE", "loc" : [  -149.828912,  61.153543 ], "pop" : 20128, "state" : "AK", "_id" : "99507" }
{ "city" : "ANCHORAGE", "loc" : [  -149.810085,  61.205959 ], "pop" : 29857, "state" : "AK", "_id" : "99508" }
{ "city" : "ANCHORAGE", "loc" : [  -149.897401,  61.119381 ], "pop" : 17094, "state" : "AK", "_id" : "99515" }
{ "city" : "ANCHORAGE", "loc" : [  -149.779998,  61.10541 ], "pop" : 18356, "state" : "AK", "_id" : "99516" }
{ "city" : "ANCHORAGE", "loc" : [  -149.936111,  61.190136 ], "pop" : 15192, "state" : "AK", "_id" : "99517" }
{ "city" : "ANCHORAGE", "loc" : [  -149.886571,  61.154862 ], "pop" : 8116, "state" : "AK", "_id" : "99518" }
```

Demo of indexes in MongoDB

- Import Data
- Create Index
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Show Existing Index
- Hint
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Explain
- Compare with data without indexes

```
> db.zips.find().limit(20).hint({loc: -1})
{ "city" : "BARROW", "loc" : [  -156.817409,  71.234637 ], "pop" : 3696, "state" : "AK", "_id" : "99723" }
{ "city" : "WAINWRIGHT", "loc" : [  -160.012532,  70.620064 ], "pop" : 492, "state" : "AK", "_id" : "99782" }
{ "city" : "NUIQSUT", "loc" : [  -150.997119,  70.192737 ], "pop" : 354, "state" : "AK", "_id" : "99789" }
{ "city" : "PRUDHOE BAY", "loc" : [  -148.559636,  70.070057 ], "pop" : 153, "state" : "AK", "_id" : "99734" }
{ "city" : "KAKTOVIK", "loc" : [  -143.631329,  70.042889 ], "pop" : 245, "state" : "AK", "_id" : "99747" }
{ "city" : "POINT LAY", "loc" : [  -162.906148,  69.705626 ], "pop" : 139, "state" : "AK", "_id" : "99759" }
{ "city" : "POINT HOPE", "loc" : [  -166.72618,  68.312058 ], "pop" : 640, "state" : "AK", "_id" : "99766" }
{ "city" : "ANAKTUVUK PASS", "loc" : [  -151.679005,  68.11878 ], "pop" : 260, "state" : "AK", "_id" : "99721" }
{ "city" : "ARCTIC VILLAGE", "loc" : [  -145.423115,  68.077395 ], "pop" : 107, "state" : "AK", "_id" : "99722" }
{ "city" : "KIVALINA", "loc" : [  -163.733617,  67.665859 ], "pop" : 689, "state" : "AK", "_id" : "99750" }
{ "city" : "AMBLER", "loc" : [  -156.455652,  67.46951 ], "pop" : 8, "state" : "AK", "_id" : "99786" }
{ "city" : "KIANA", "loc" : [  -158.152204,  67.18026 ], "pop" : 349, "state" : "AK", "_id" : "99749" }
{ "city" : "BETTLES FIELD", "loc" : [  -151.062414,  67.100495 ], "pop" : 156, "state" : "AK", "_id" : "99726" }
{ "city" : "VENETIE", "loc" : [  -146.413723,  67.010446 ], "pop" : 184, "state" : "AK", "_id" : "99781" }
{ "city" : "NOATAK", "loc" : [  -160.509453,  66.97553 ], "pop" : 395, "state" : "AK", "_id" : "99761" }
{ "city" : "SHUNGNAK", "loc" : [  -157.613496,  66.958141 ], "pop" : 0, "state" : "AK", "_id" : "99773" }
{ "city" : "KOBUK", "loc" : [  -157.066864,  66.912253 ], "pop" : 306, "state" : "AK", "_id" : "99751" }
{ "city" : "KOTZEBUE", "loc" : [  -162.126493,  66.846459 ], "pop" : 3347, "state" : "AK", "_id" : "99752" }
{ "city" : "NOORVIK", "loc" : [  -161.044132,  66.836353 ], "pop" : 534, "state" : "AK", "_id" : "99763" }
{ "city" : "CHALKYITSIK", "loc" : [  -143.638121,  66.719 ], "pop" : 99, "state" : "AK", "_id" : "99788" }
```

Demo of indexes in MongoDB

- Import Data
- Create Index
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Show Existing Index
- Hint
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Explain
- Compare with data without indexes

```
> db.zips.find({city: 'NASHVILLE', state: 'TN'}).explain()
{
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 19,
        "nscannedObjects" : 29467,
        "nscanned" : 29467,
        "nscannedObjectsAllPlans" : 29467,
        "nscannedAllPlans" : 29467,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 33,
        "indexBounds" : {

        },
        "server" : "g:27017"
}
```

Demo of indexes in MongoDB

- Import Data
- Create Index
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Show Existing Index
- Hint
  - Single Field Index
  - Compound Field Indexes
  - Multikey Indexes
- Explain
- Compare with data without indexes

```
> db.zips.dropIndexes()
{
        "nIndexesWas" : 4,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
> db.zips.find({city: 'NASHVILLE', state: 'TN'}).explain()
{
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 19,
        "nscannedObjects" : 29467,
        "nscanned" : 29467,
        "nscannedObjectsAllPlans" : 29467,
        "nscannedAllPlans" : 29467,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 33,
        "indexBounds" : {

        },
        "server" : "g:27017"
}
```

Without Index

```
> db.zips.find({city: 'NASHVILLE', state: 'TN'}).explain()
{
        "cursor" : "BtreeCursor state_1_city_1",
        "isMultiKey" : false,
        "n" : 19,
        "nscannedObjects" : 19,
        "nscanned" : 19,
        "nscannedObjectsAllPlans" : 19,
        "nscannedAllPlans" : 19,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 0,
        "indexBounds" : {
                "state" : [
                        [
                                "TN",
                                "TN"
                        ]
                ],
                "city" : [
                        [
                                "NASHVILLE",
                                "NASHVILLE"
                        ]
                ]
        },
        "server" : "g:27017"
}
```

With Index

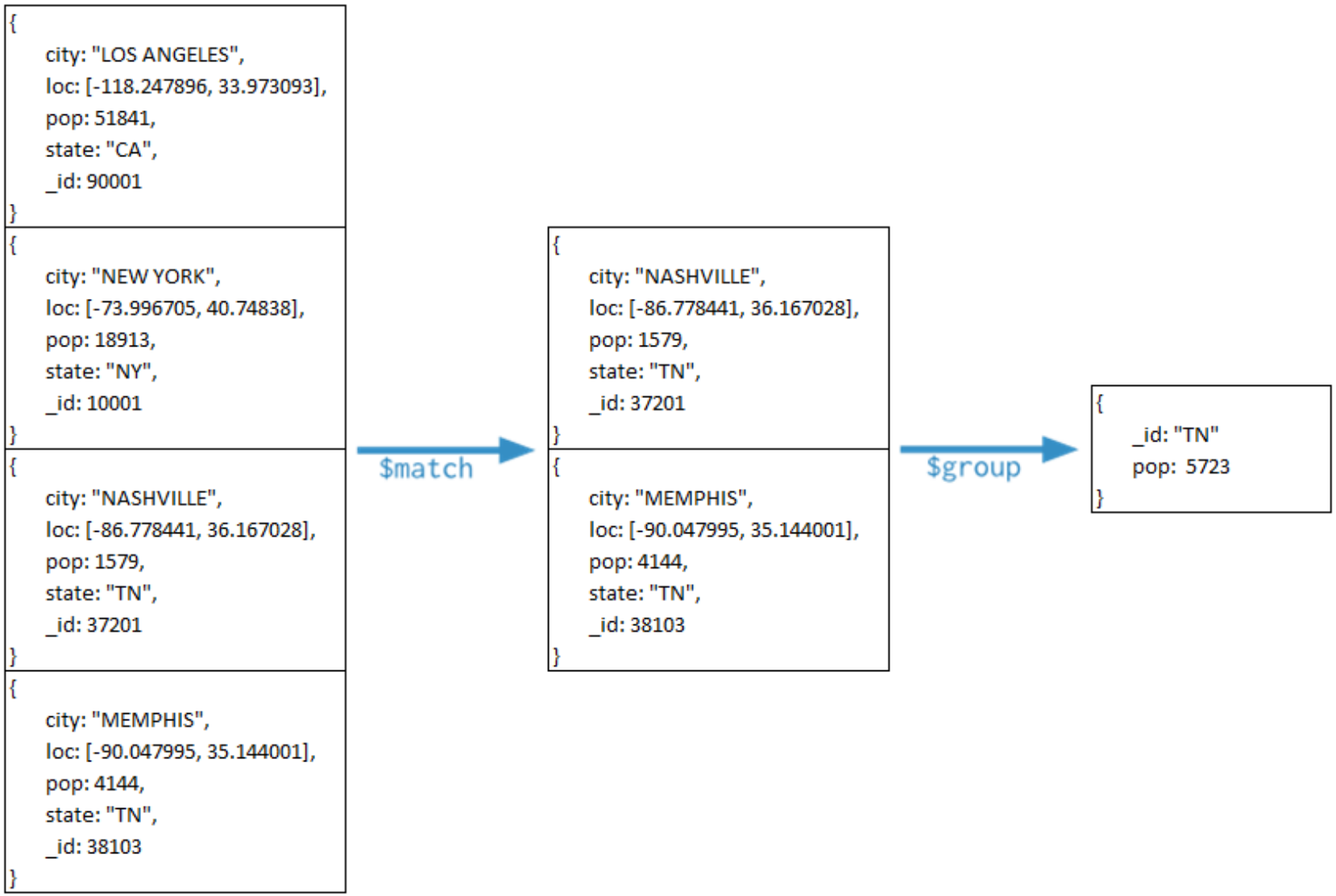
Aggregation

- Operations that process data records and return computed results.
- MongoDB provides aggregation operations
- Running data aggregation on the mongod instance simplifies application code and limits resource requirements.

Pipelines

- Modeled on the concept of data processing pipelines.
- Provides:
  - *filters* that operate like queries
  - *document transformations* that modify the form of the output document.
- Provides tools for:
  - grouping and sorting by field
  - aggregating the contents of arrays, including arrays of documents
- Can use operators for tasks such as calculating the average or concatenating a string.

```
db.zips.aggregate(
                    { $match: { state: "TN" } },
                    { $group: {_id: "TN", pop: { $sum: "$pop" } } }
                  );
```



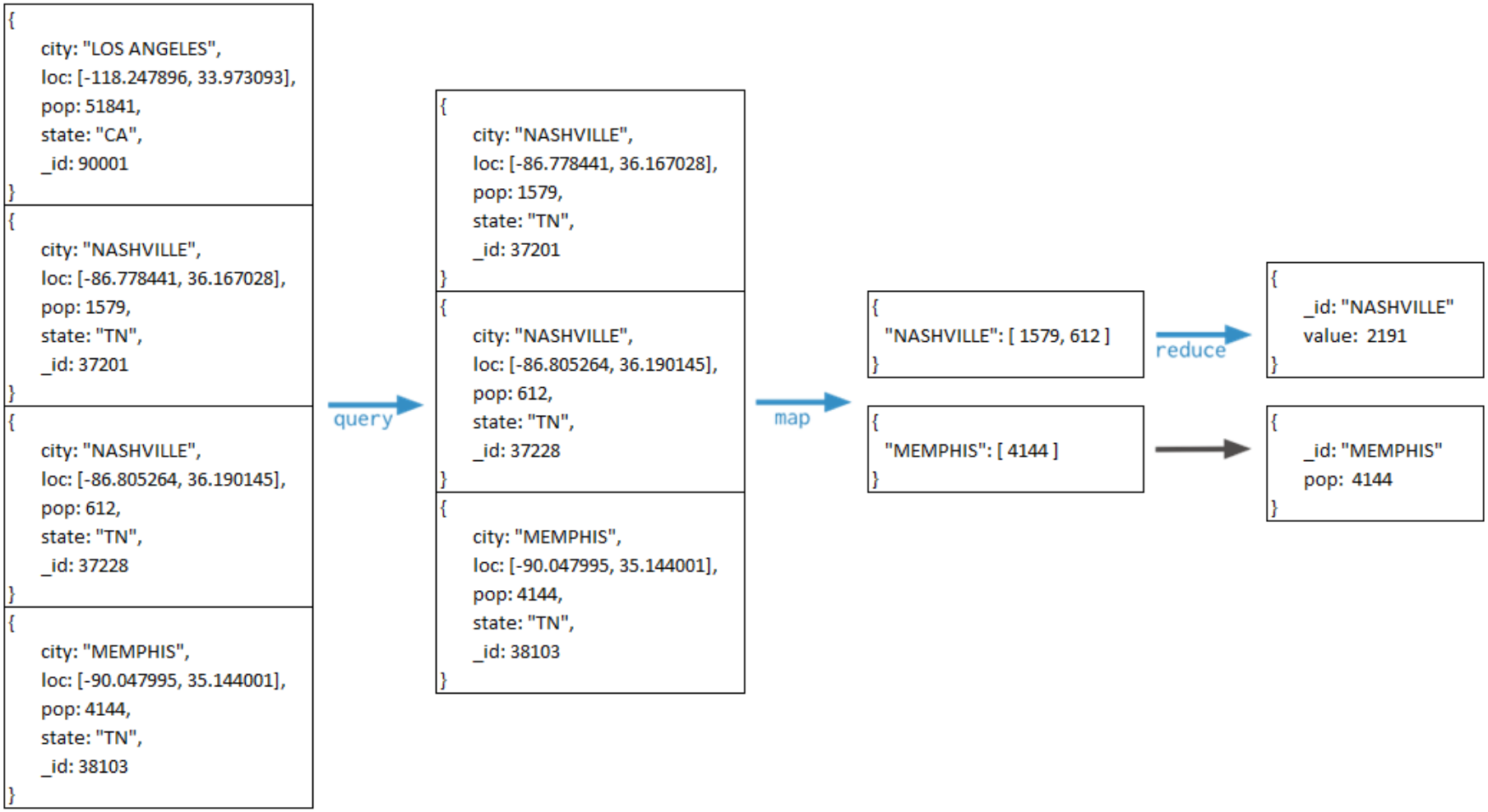
Pipelines

- $limit
- $skip
- $sort

Map-Reduce

- Has two phases:
  - A *map* stage that processes each document and *emits* one or more objects for each input document
  - A *reduce* phase that combines the output of the map operation.
  - An optional *finalize* stage for final modifications to the result
- Uses Custom JavaScript functions
  - Provides greater flexibility but is less efficient and more complex than the aggregation pipeline
- Can have output sets that exceed the 16 megabyte output limitation of the aggregation pipeline.

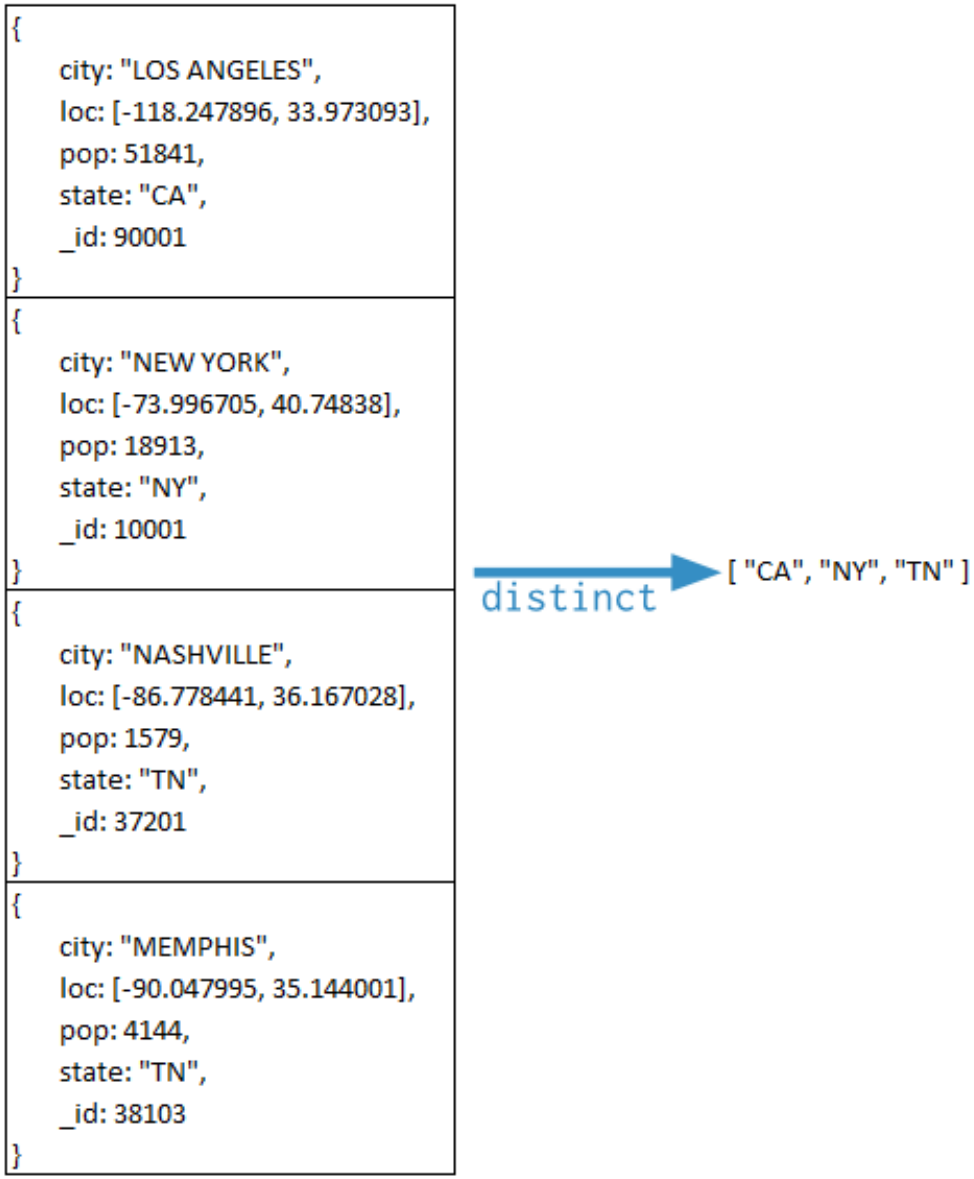
```
db.zips.mapReduce(
                        function() { emit( this.city, this.pop ); },
                        function(key, values) { return Array.sum( values ) },
                        {
                          query: { state: "TN" },
                          out: "city_pop_totals"
                        }
                      );
```



Single Purpose Aggregation Operations

- Special purpose database commands:
  - returning a count of matching documents
  - returning the distinct values for a field
  - grouping data based on the values of a field.
- Aggregate documents from a single collection.
- Lack the flexibility and capabilities of the aggregation pipeline and map-reduce.

```
db.zips.distinct( "state" );
```



	aggregate	mapReduce	group
Description	*New in version 2.2.* Designed with specific goals of improving performance and usability for aggregation tasks. Uses a "pipeline" approach where objects are transformed as they pass through a series of pipeline operators such as `$group`, `$match`, and `$sort`. See Aggregation Reference for more information on the pipeline operators.	Implements the Map-Reduce aggregation for processing large data sets.	Provides grouping functionality. Is slower than the `aggregate` command and has less functionality than the `mapReduce` command.
Key Features	Pipeline operators can be repeated as needed. Pipeline operators need not produce one output document for every input document. Can also generate new documents or filter out documents.	In addition to grouping operations, can perform complex aggregation tasks as well as perform incremental aggregation on continuously growing datasets. See Map-Reduce Examples and Perform Incremental Map-Reduce.	Can either group by existing fields or with a custom `keyf` JavaScript function, can group by calculated fields. See `group` for information and example using the `keyf` function.

Flexibility	Limited to the operators and expressions supported by the aggregation pipeline. However, can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results by using the `$project` pipeline operator. See `$project` for more information as well as Aggregation Reference for more information on all the available pipeline operators.	Custom `map`, `reduce` and `finalize` JavaScript functions offer flexibility to aggregation logic. See `mapReduce` for details and restrictions on the functions.	Custom `reduce` and `finalize` JavaScript functions offer flexibility to grouping logic. See `group` for details and restrictions on these functions.
Output Results	Returns results inline. The result is subject to the BSON Document size limit.	Returns results in various options (inline, new collection, merge, replace, reduce). See `mapReduce` for details on the output options. *Changed in version 2.2:* Provides much better support for sharded map-reduce output than previous versions.	Returns results inline as an array of grouped items. The result set must fit within the maximum BSON document size limit. *Changed in version 2.2:* The returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. Previous versions had a limit of 10,000 elements.
Sharding	Supports non-sharded and sharded input collections.	Supports non-sharded and sharded input collections.	Does **not** support sharded collection.

```
C:\mongodb\bin>mongo.exe
MongoDB shell version: 2.4.9
connecting to: test
> show dbs
blog    0.203125GB
local   0.078125GB
test    0.203125GB
> use blog
switched to db blog
> db.zips.aggregate( {$match: {state: "TN"}}, {$group: {_id: "TN", pop: {$sum: "
$pop"}}})
{ "result" : [ { "_id" : "TN", "pop" : 4876457 } ], "ok" : 1 }
> db.zips.mapReduce(
... function() { emit( this.city, this.pop ); },
... function(key, values) { return Array.sum( values ) },
... {
...   query: { state: "TN" },
...   out: "city_pop_totals"
... }
... );
{
        "result" : "city_pop_totals",
        "timeMillis" : 198,
        "counts" : {
                "input" : 582,
                "emit" : 582,
                "reduce" : 13,
                "output" : 505
        },
        "ok" : 1,
}
> db.city_pop_totals.find({_id: "NASHVILLE"})
{ "_id" : "NASHVILLE", "value" : 349822 }
>
```

```
> db.zips.distinct( "state" )
[
        "AL",
        "AK",
        "AZ",
        "AR",
        "CA",
        "CO",
        "CT",
        "DE",
        "DC",
        "FL",
        "GA",
        "HI",
        "ID",
        "IL",
        "IN",
        "IA",
        "KS",
        "KY",
        "LA",
        "ME",
        "MD",
        "MA",
        "MI",
        "MN",
        "MS",
        "MO",
        "MT",
        "NE",
        "NV",
        "NH",
        "NJ",
        "NM",
        "NY",
        "NC",
        "ND",
        "OH",
        "OK",
        "OR",
        "PA",
        "RI",
        "SC",
        "SD",
        "TN",
        "TX",
        "UT",
        "VT",
        "VA",
        "WA",
        "WV",
        "WI",
        "WY"
]
>
```

Replication & Sharding

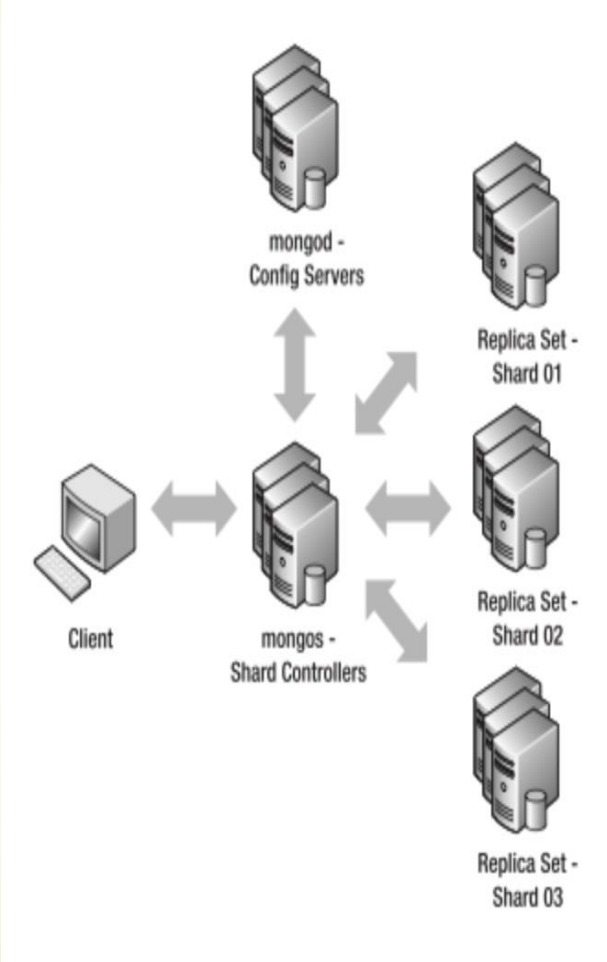


Image source: http://mongodb.in.th

Replication

- What is replication?
- Purpose of replication/redundancy
  - Fault tolerance
  - Availability
    - Increase read capacity

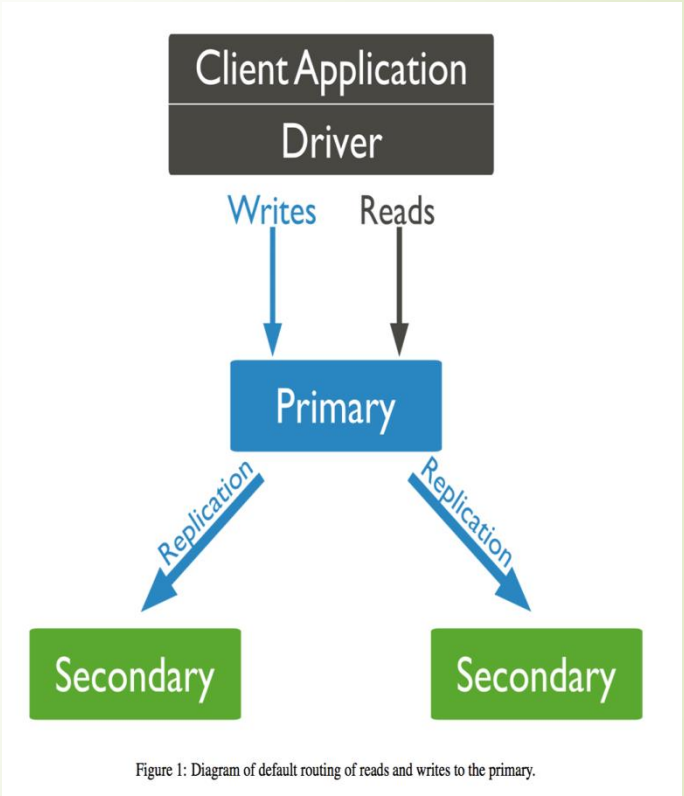


Figure 1: Diagram of default routing of reads and writes to the primary.

Replication in MongoDB

- Replica Set Members
  - Primary
    - Read, Write operations
  - Secondary
    - Asynchronous Replication
    - Can be primary
  - Arbiter
    - Voting
    - Can't be primary
  - Delayed Secondary
    - Can't be primary

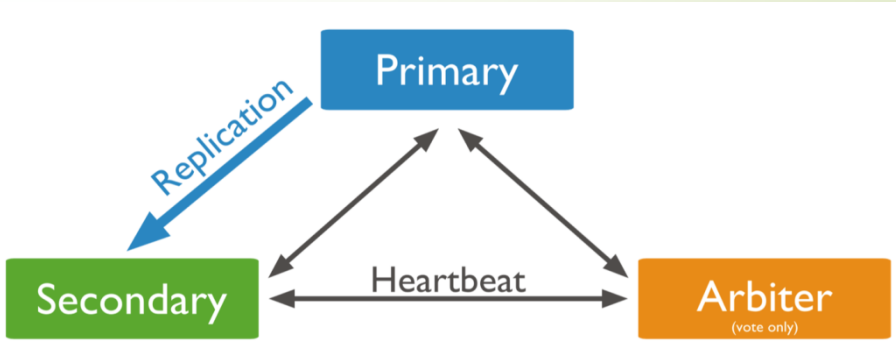


Figure 3: Diagram of a replica set that consists of a primary, a secondary, and an arbiter.

Replication in MongoDB

- Automatic Failover
  - Heartbeats
  - Elections
- The Standard Replica Set Deployment
- Deploy an Odd Number of Members
- Rollback
- Security
  - SSL/TLS

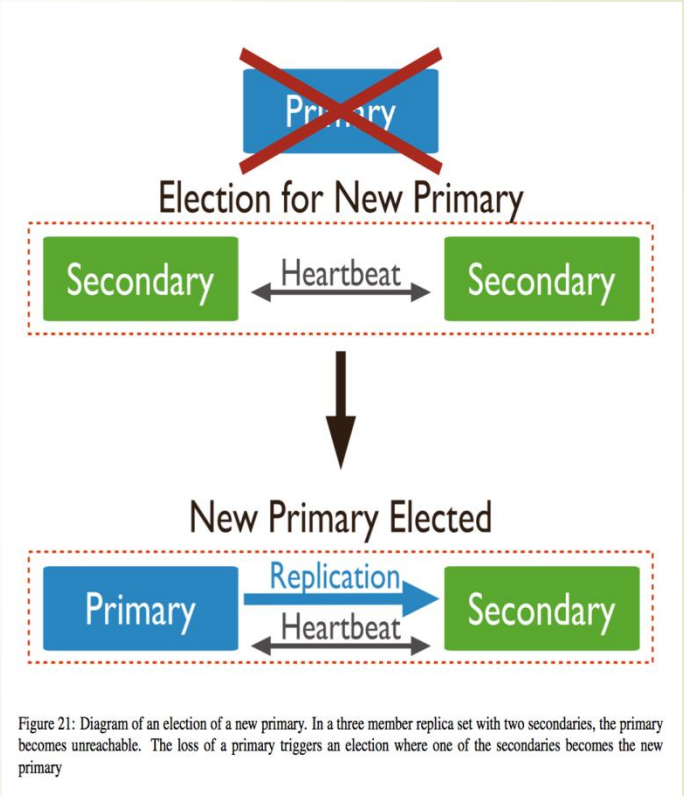
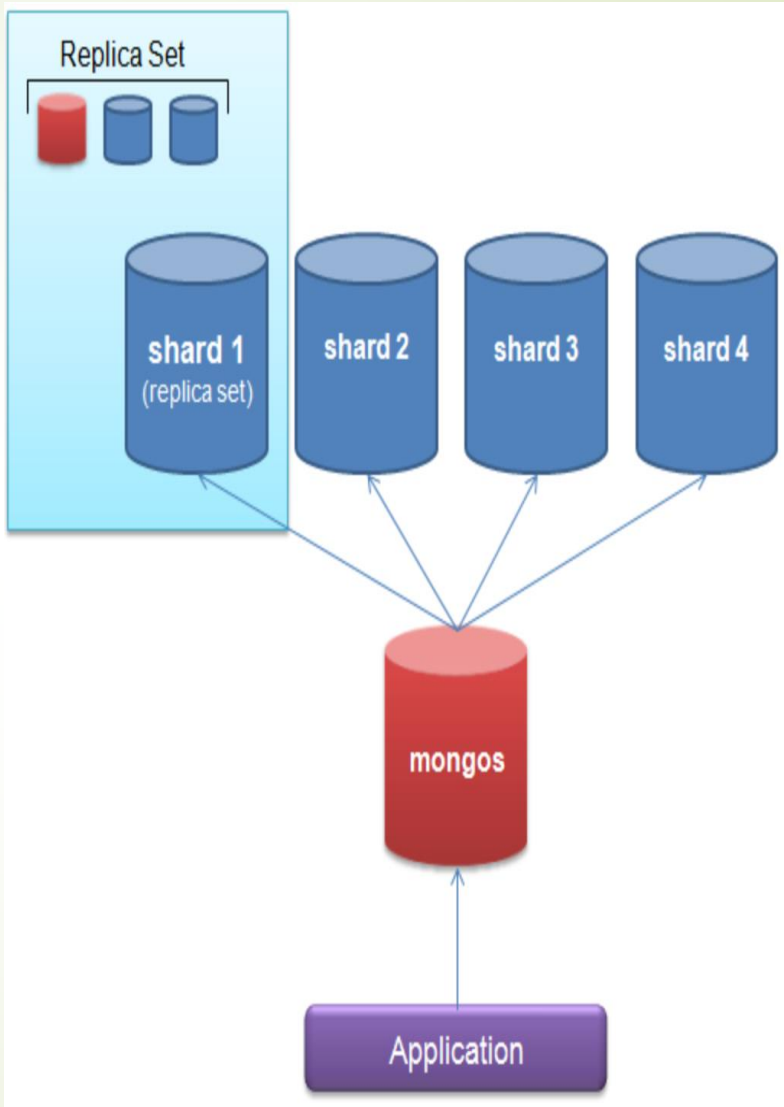


Figure 21: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

Demo for Replication

Sharding

- What is sharding?
- Purpose of sharding
  - Horizontal scaling out
- Query Routers
  - mongos
- Shard keys
  - Range based sharding
  - Cardinality
  - Avoid hotspotting



Demo for Sharding

Thanks