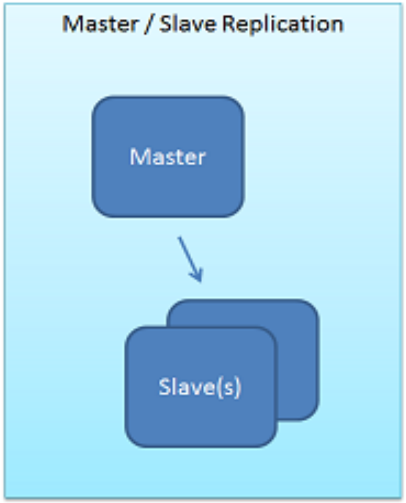
MongoDB-4

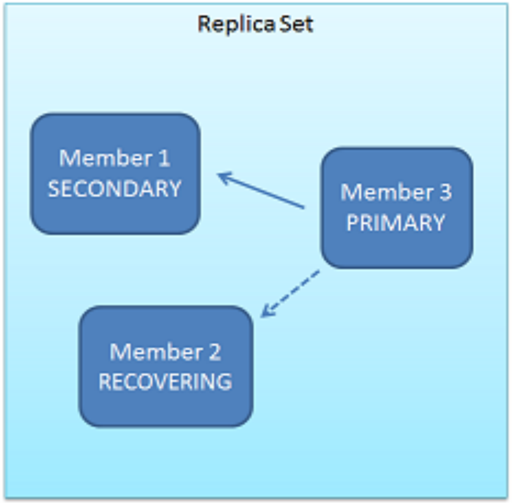
WPI, Mohamed Eltabakh

Architecture
Replication & Sharding
(Chapters 9, 10)

Replication (Chapter 9)

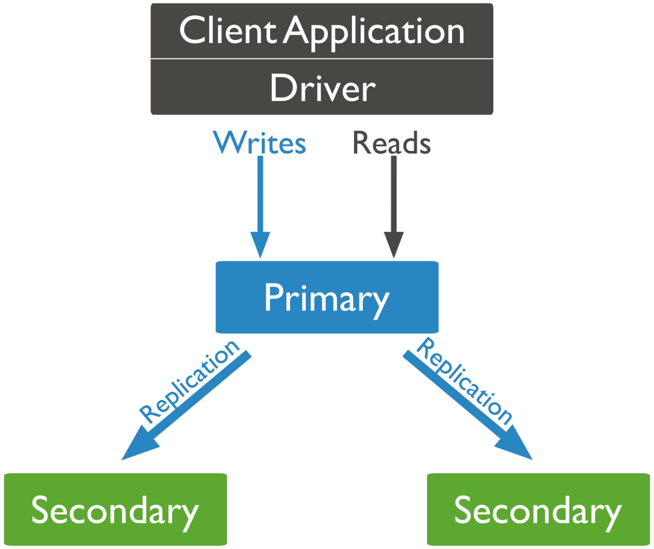
- **Replica Set**
  - Similar in concept to Master-Slave architecture
  - Goal: Availability, Fault Tolerance, Load Balancing
  - Replica sets are more recent mechanisms
  - Give more flexibility (fine tuning)





Replica Set

- Consists of one "***Primary***" and multiple "***Secondary***"



- All write ops must go to the primary

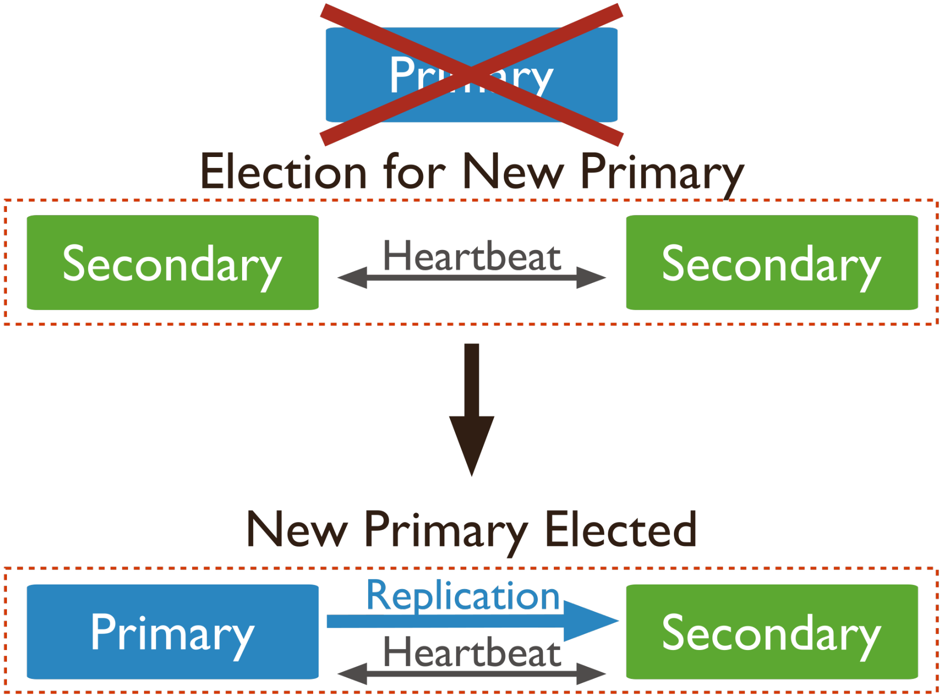
- Primary maintains a log "oplog"

- Secondary sites periodically read & apply the log from the primary site

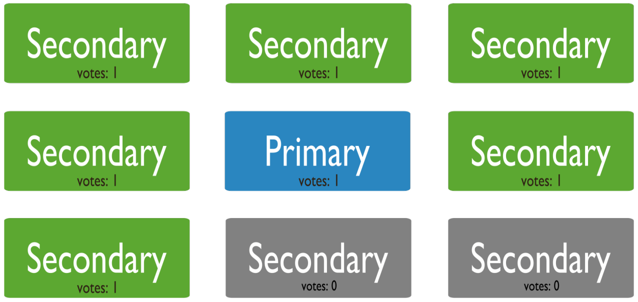
Eventual Consistency

Election when Primary Fails

- Based on majority voting
- Number of members should be odd
- During election, no writes are accepted



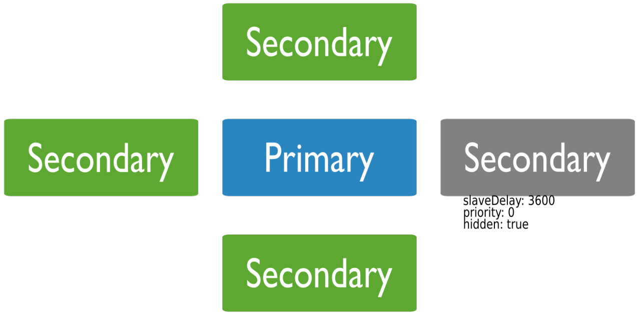
Configuring Secondary Sites



Secondary votes: 1 | Secondary votes: 1 | Secondary votes: 1

Secondary votes: 1 | Primary votes: 1 | Secondary votes: 1

Secondary votes: 1 | Secondary votes: 0 | Secondary votes: 0



Secondary

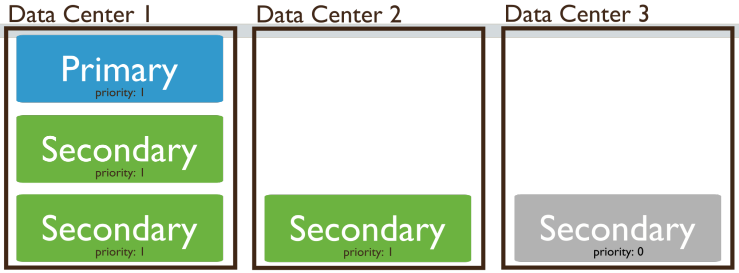
Secondary | Primary | Secondary

slaveDelay: 3600
priority: 0
hidden: true

Secondary

- Number of secondaries
- Priority = 0 🡐 cannot be elected as primary
- Hidden = True 🡐 Cannot serve client operations
- SlaveDelay = m 🡐 waits m msec before getting the updates from the primary site

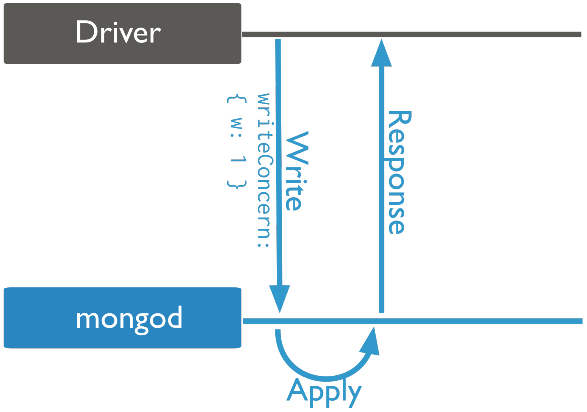
Configuring Secondary Sites



- **Priority = 0**
  - Cannot be primary
  - Cannot accept write
  - Still has data & accept reads
  - May want some data centers not to accept write ops
- **Hidden = True**
  - Imply Priority = 0
  - But also cannot accept reads from clients
  - Good for dedicated offline tasks, e.g., reporting
- **SlaveDelay = m**
  - Should be Hidden = True
  - Good to recover from bad transactions

Writing/Reading: Default Behavior

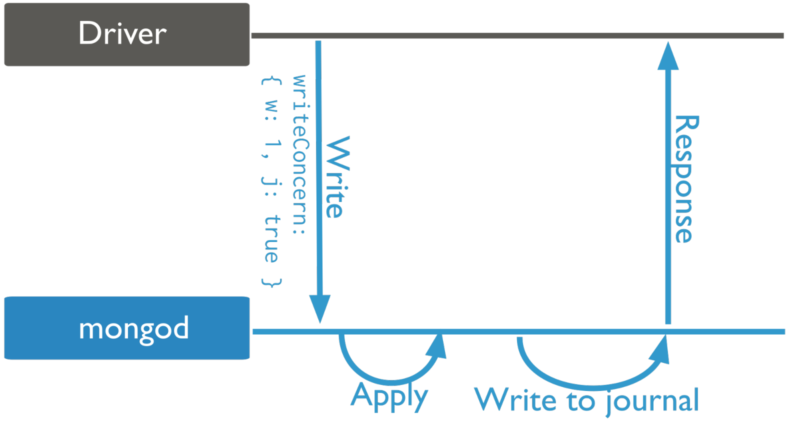
- **Write**
  - All writes go to the primary
  - A write is accepted once the primary accept op. (in memory)
  - Secondaries are not updated yet
- **Read**
  - All reads go to the primary
  - Ensures ***Strict Consistency***



Accepted data can be lost

In this case Secondaries are mostly for Availability & Fault Tolerance

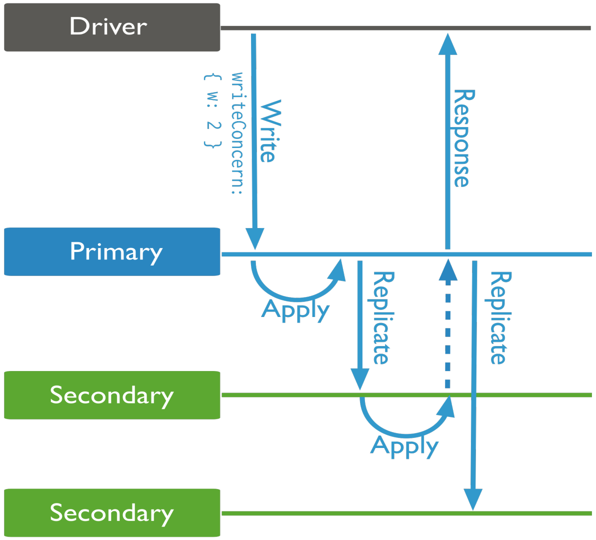
Journaling: Persistent Data



- As before, but a write is accepted only after written to a log on disk
- Still on the primary site
- Accepted data become persistent

Higher Consistency For Reads

- **Option 1- Read From Primary**
  - Keep writing as is
  - Enforce the read from Primary
  - ➔ Strict Consistency



- **Option 2: Expensive Write**
  - Write is not accepted until m secondaries are also updated

```
db.products.insert(
    { item: "envelopes", qty : 100, type: "Clasp" },
    { writeConcern: { w: 2, wtimeout: 5000 } }
)
```

Read Modes

Primary

PrimaryPreferred

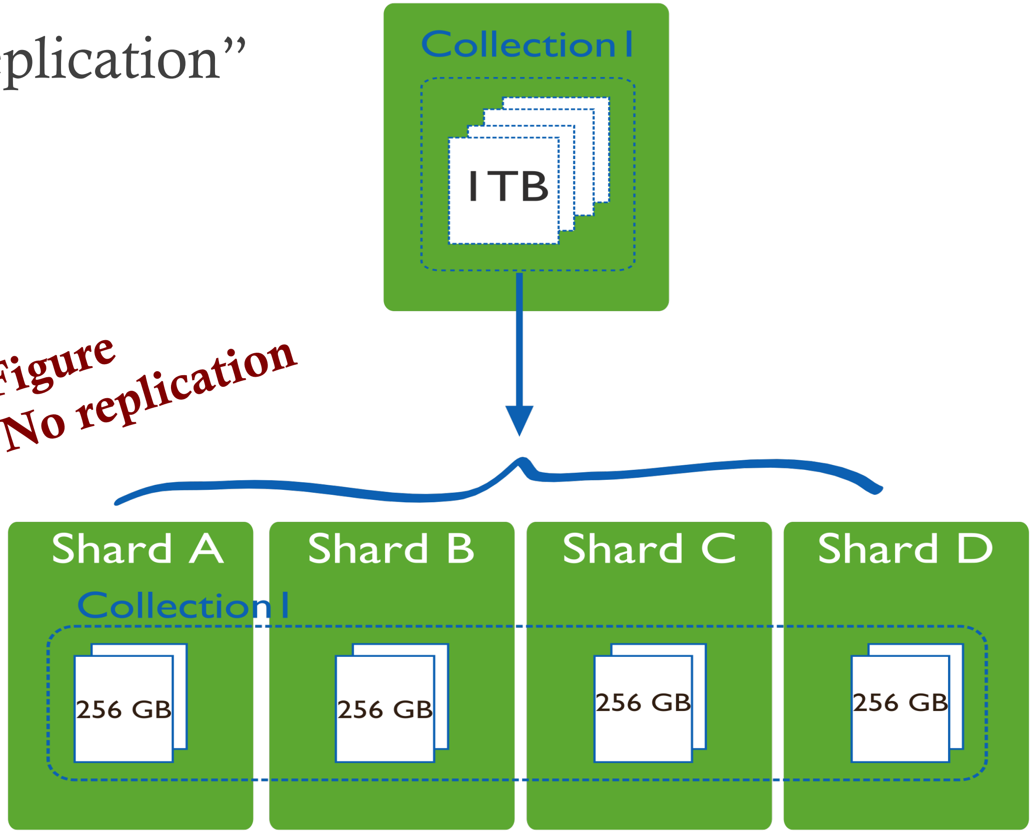
Secondary

SecondayPreferred

Nearest

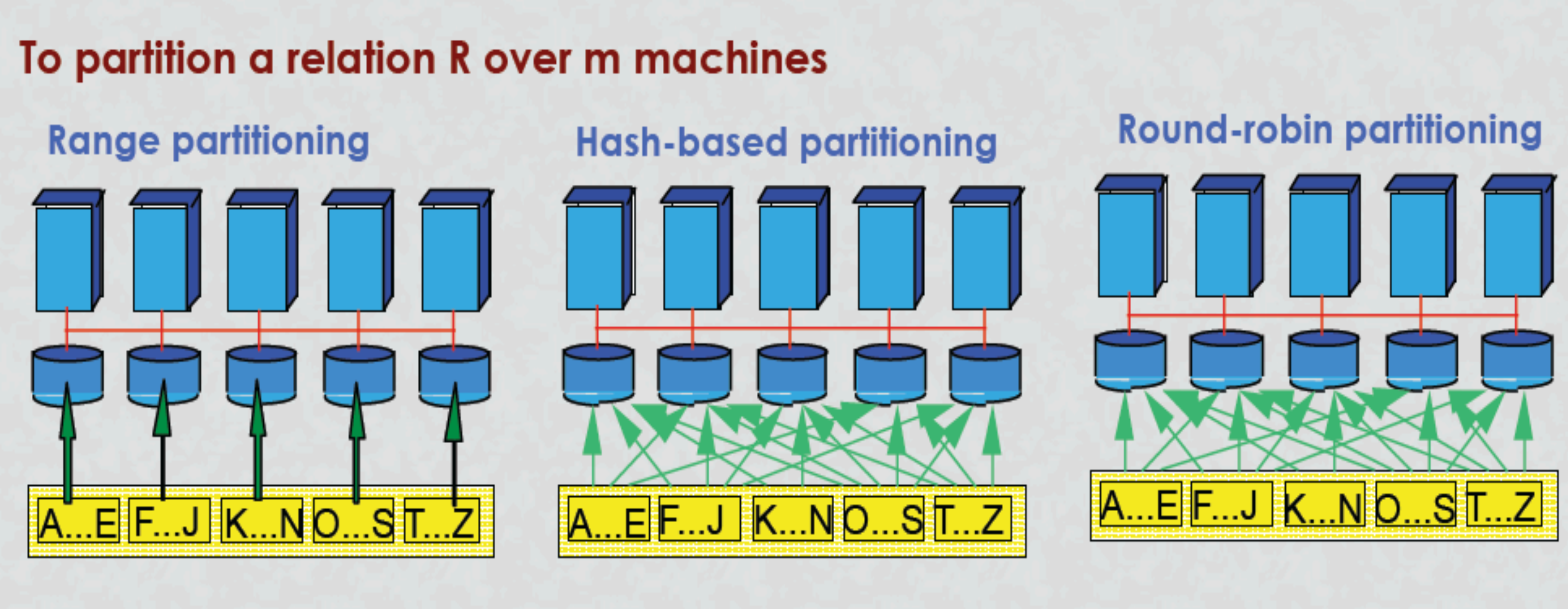
Sharding (Chapter 10)

- Partitioning the data across many machine
- Orthogonal to “Replication”



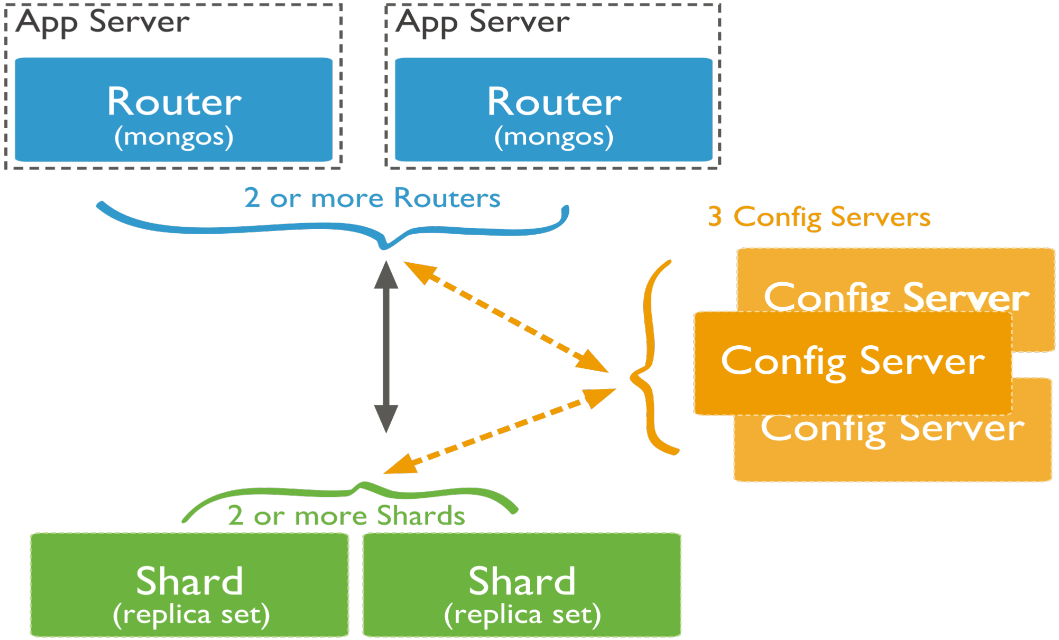
In this Figure
Only sharding, No replication

Similar Concept in DDBMS



To partition a relation R over m machines

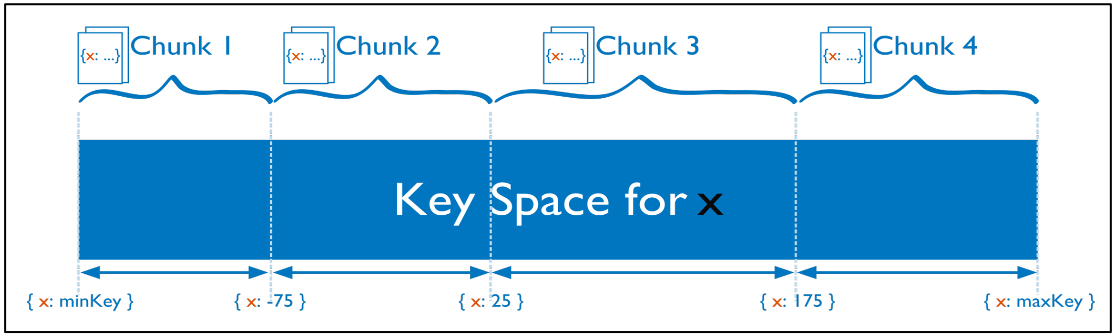
MongoDB Sharded Cluster



- Shard: storing data, can be replicated (replica set)
- Config Server: Storing metadata info
- Router: Accepts and routes client’s queries & update operations

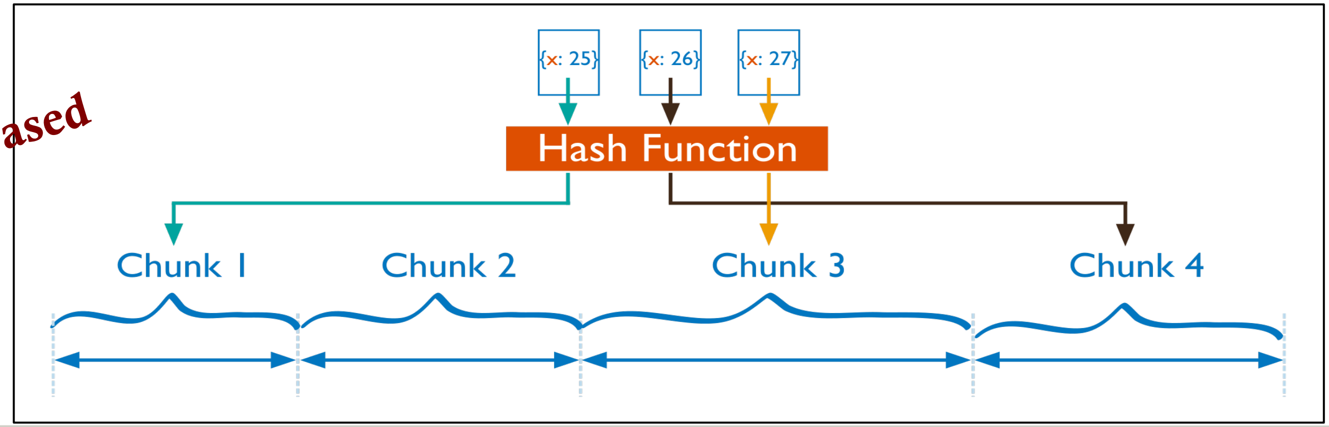
Shard Key

- A collection is sharded based on a ***key*** into chunks
- ***Key:*** must be present in each document (and indexed)



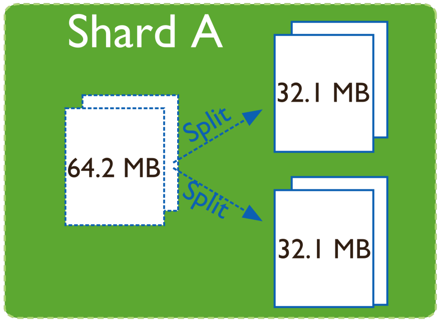
Range-Based

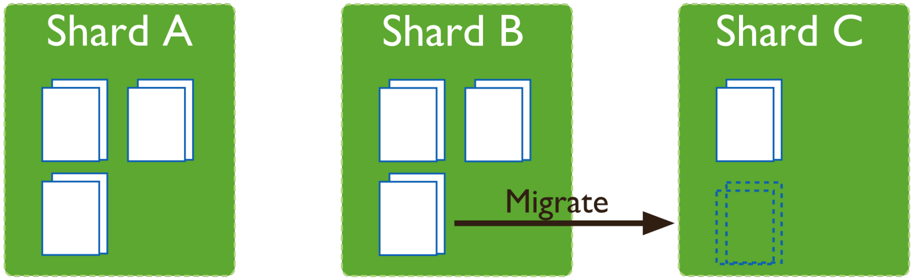
Hash-Based



Keeping Balanced Shards

- Splitter
  - Splits a big chunk into two
  - No change in metadata info
  - Triggered by inserts/updates

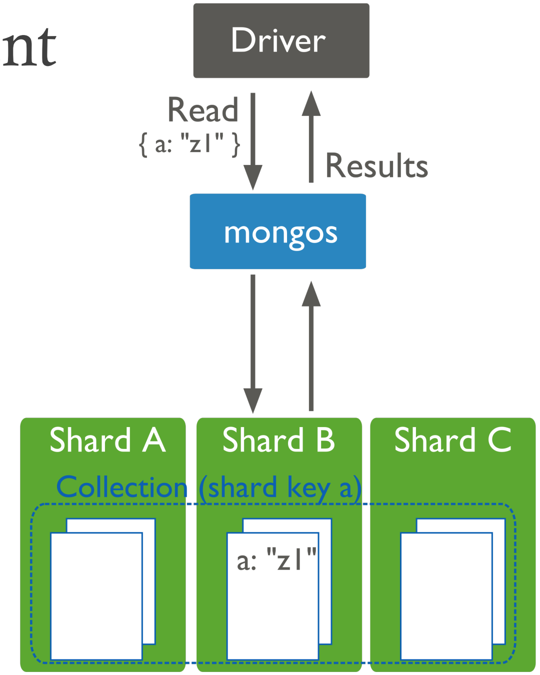


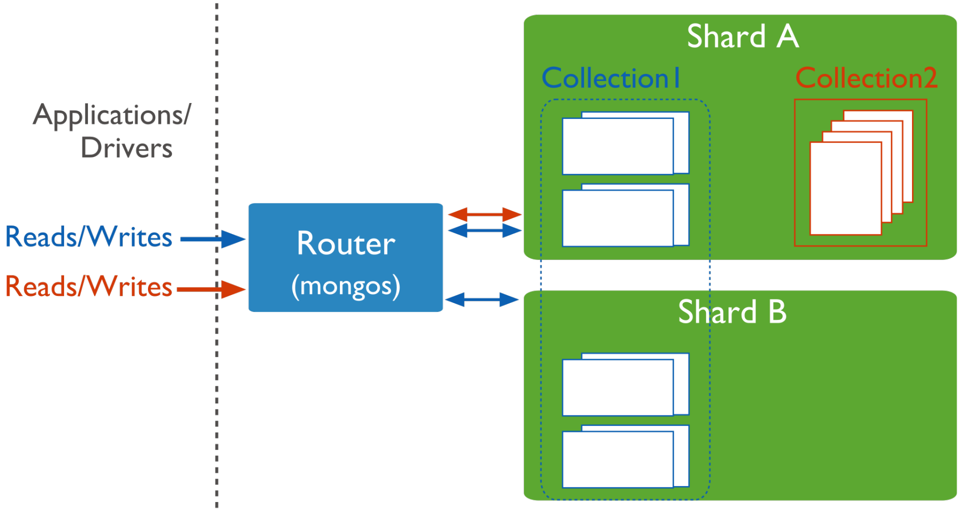


- Balancer
  - Migrates chunks from one shard (largest in number) to another (least in number)
  - Changes the metadata into

Routing Operations to Shards

- Read/write operations are sent from client to mongos
- Mongos routes them to the appropriate shards(s)





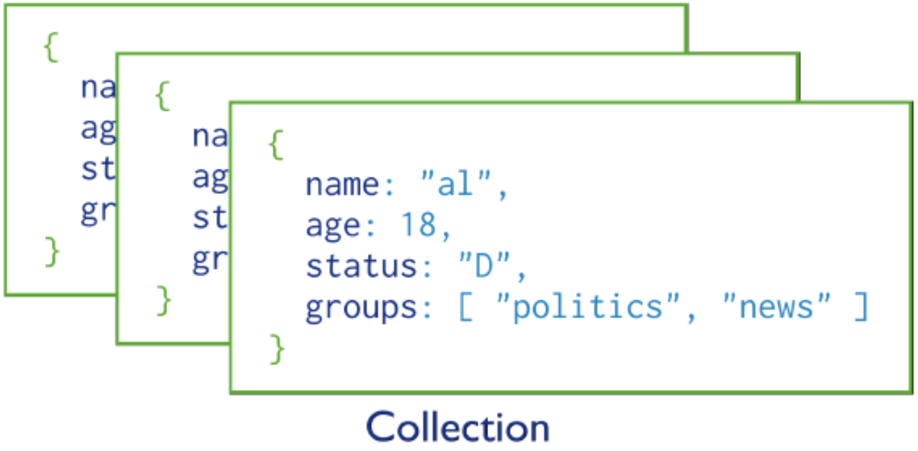
Indexing
(Chapter 8)

Indexes

- Speedup queries
- MongoDB uses B-Tree indexes
- Can build the index on any field of the document
- Skips documents that do not have the indexed field (Sparse index)

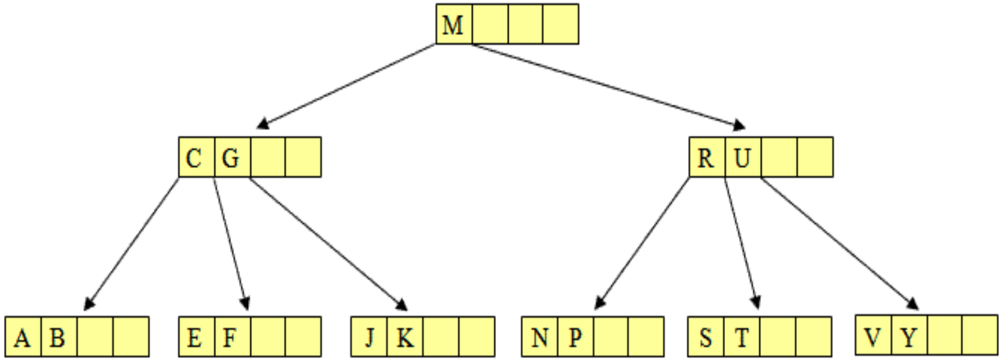
Indexes

- Index is an auxiliary data structure
- Stores the values of specific field(s) in a sorted order
- Organized in a certain structure to speedup the search

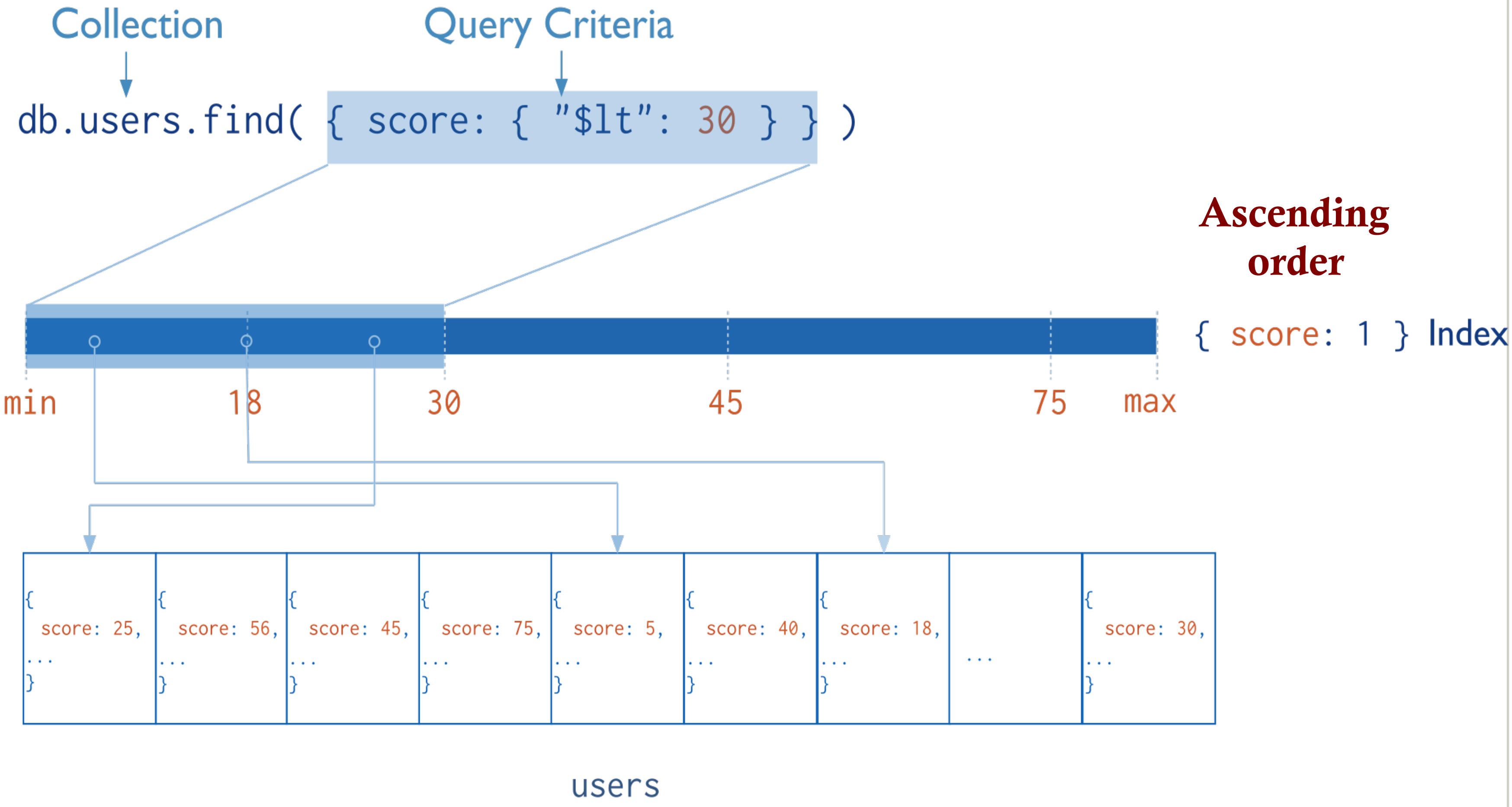


```
{
  name: "al",
  age: 18,
  status: "D",
  groups: [ "politics", "news" ]
}
```

Collection



Index Usage

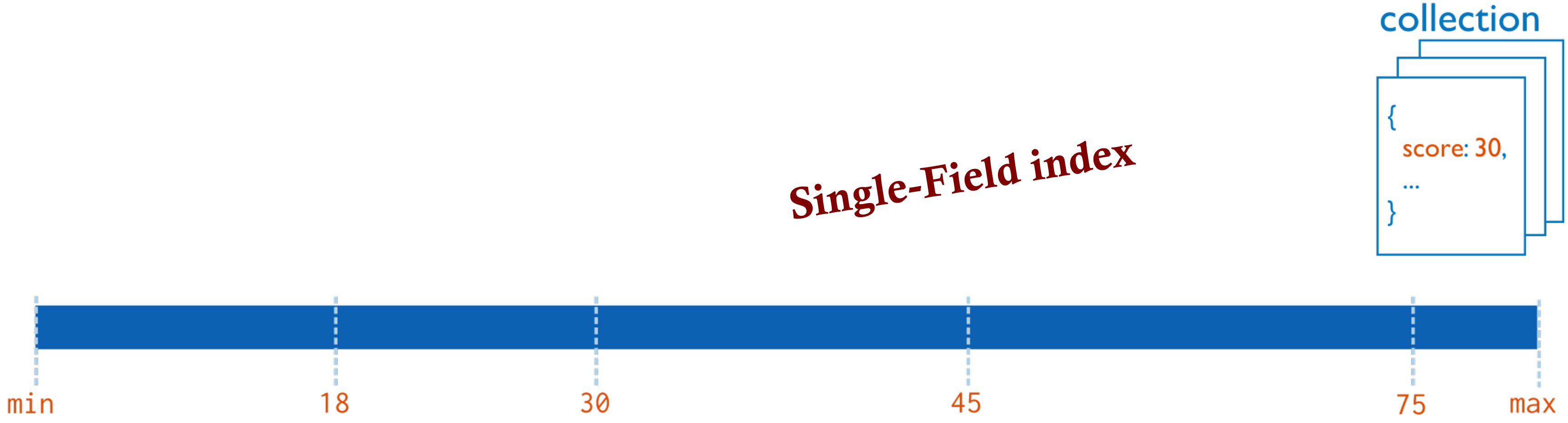


```
db.users.find( { score: { "$lt": 30 } } )
```

Ascending order

Indexed Fields

- _id: Unique, automatically has a B-Tree index
- Others are user-defined indexes



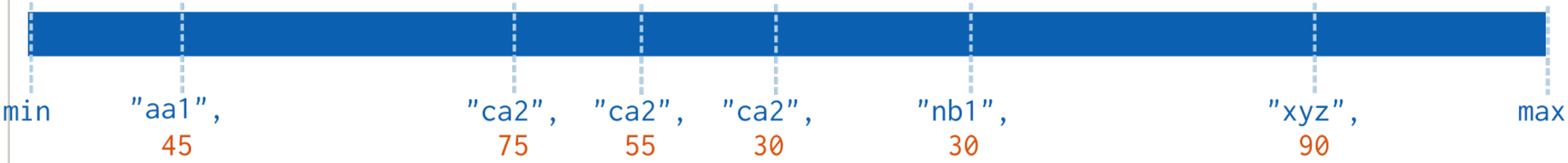
Single-Field index

{ score: 1 } Index

Indexed Fields: Compound-Fields

collection
{
score: 30,
userid: ....,
...
}

Searching has to involve the 1st level field (userid in the example)

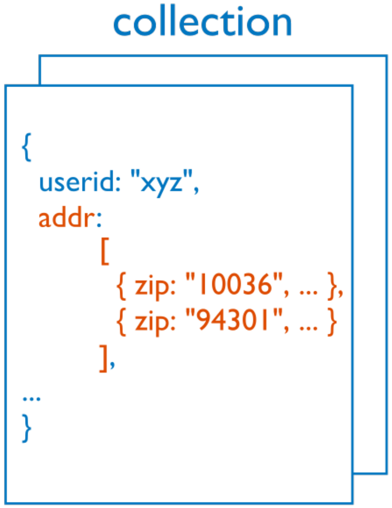


{ userid: 1, score: -1 } Index

descending order

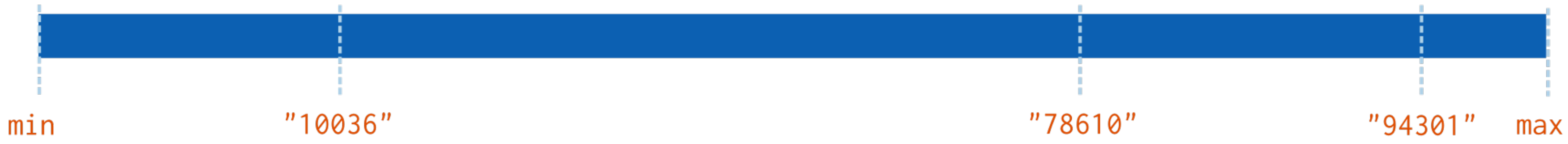
Indexed Fields: Arrays

- MongoDB automatically detects that “addr” is an array
- Indexes all the fields inside the array
- Many index values will point to the same document



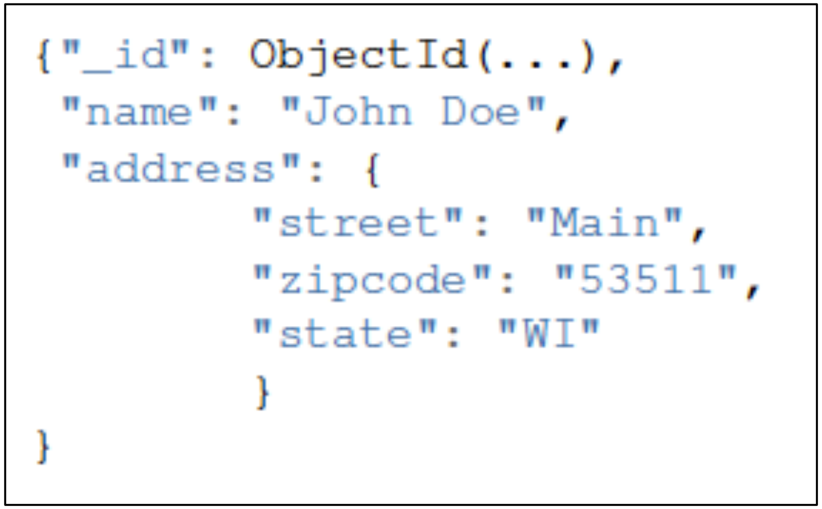
collection

```
{
  userid: "xyz",
  addr:
       [
         { zip: "10036", ... },
         { zip: "94301", ... }
       ],
...
}
```



`{ "addr.zip": 1 }` Index

Examples



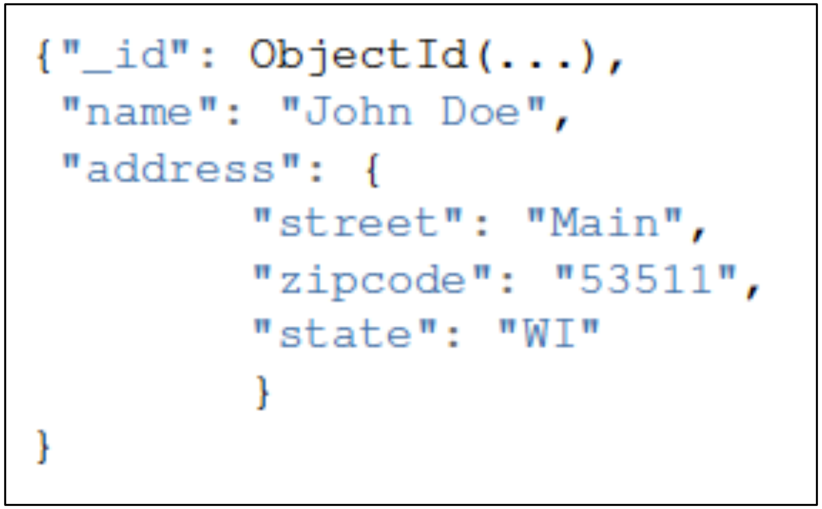
```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
        "street": "Main",
        "zipcode": "53511",
        "state": "WI"
        }
}
```

db.people.createIndex(“name”: 1) **Field Level**

db.people.createIndex(“address.zipcode”: 1) **Sub-Field Level**

db.people.createIndex(“address”: 1) **Embedded document Level**
(equality search only)

Examples



```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
        "street": "Main",
        "zipcode": "53511",
        "state": "WI"
        }
}
```

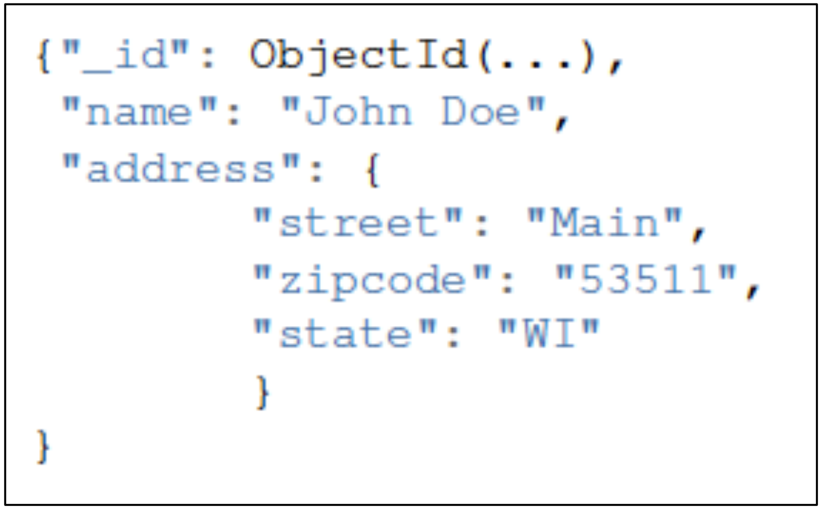
db.people.createIndex({“name”: 1, “_id”: -1})

Compound-Field Index

db.people.find(“_id”: 1000})

Index cannot answer this query
(must have a predicate on “name”)

Index Creation Options



```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
        "street": "Main",
        "zipcode": "53511",
        "state": "WI"
        }
}
```

```
db.people.createIndex({“name”: 1,  “_id”: -1},
                      {“background: True”, “Sparse”: True,
                       “unique”: True})
```

Text Indexes

- Over fields that are strings or array of strings
- Index is used when using ***$text*** search operator
- Only one index on the collection
  - But it can include multiple fields

One field

```
db.collection.createIndex({content: "text”});
```

Two fields

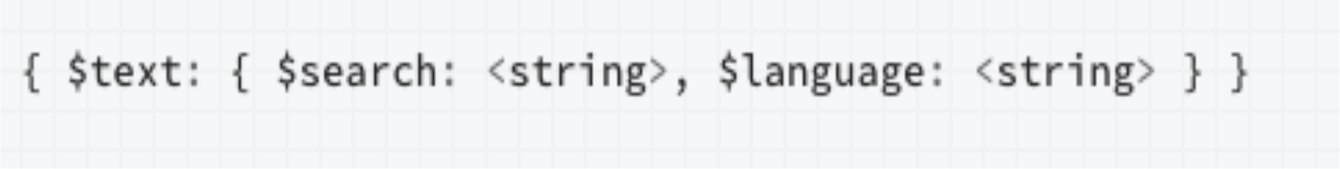
```
db.collection.createIndex({subject: "text”,content: "text”});
```

All text fields

```
db.collection.createIndex({”$**": "text”});
```

$Text

- Text search in mongoDB (Exact match)
- Uses a text index and searches the indexed fields



```
{ $text: { $search: <string>, $language: <string> } }
```

```
db.articles.find( { $text: { $search: "coffee" } } )
```

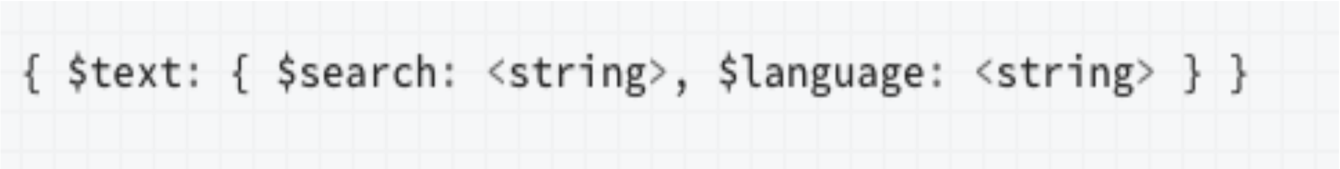
Search for “coffee” in the indexed field(s)

```
db.articles.find( { $text: { $search: "bake coffee cake" } } )
```

Apply “OR” semantics

$Text

- Text search in mongoDB
- Uses a text index and searches the indexed fields



```
{ $text: { $search: <string>, $language: <string> } }
```

```
db.articles.find( { $text: { $search: "\"coffee cake\"" } } )
```

Treated as one sentence

```
db.articles.find( { $text: { $search: "bake coffee -cake" } } )
```

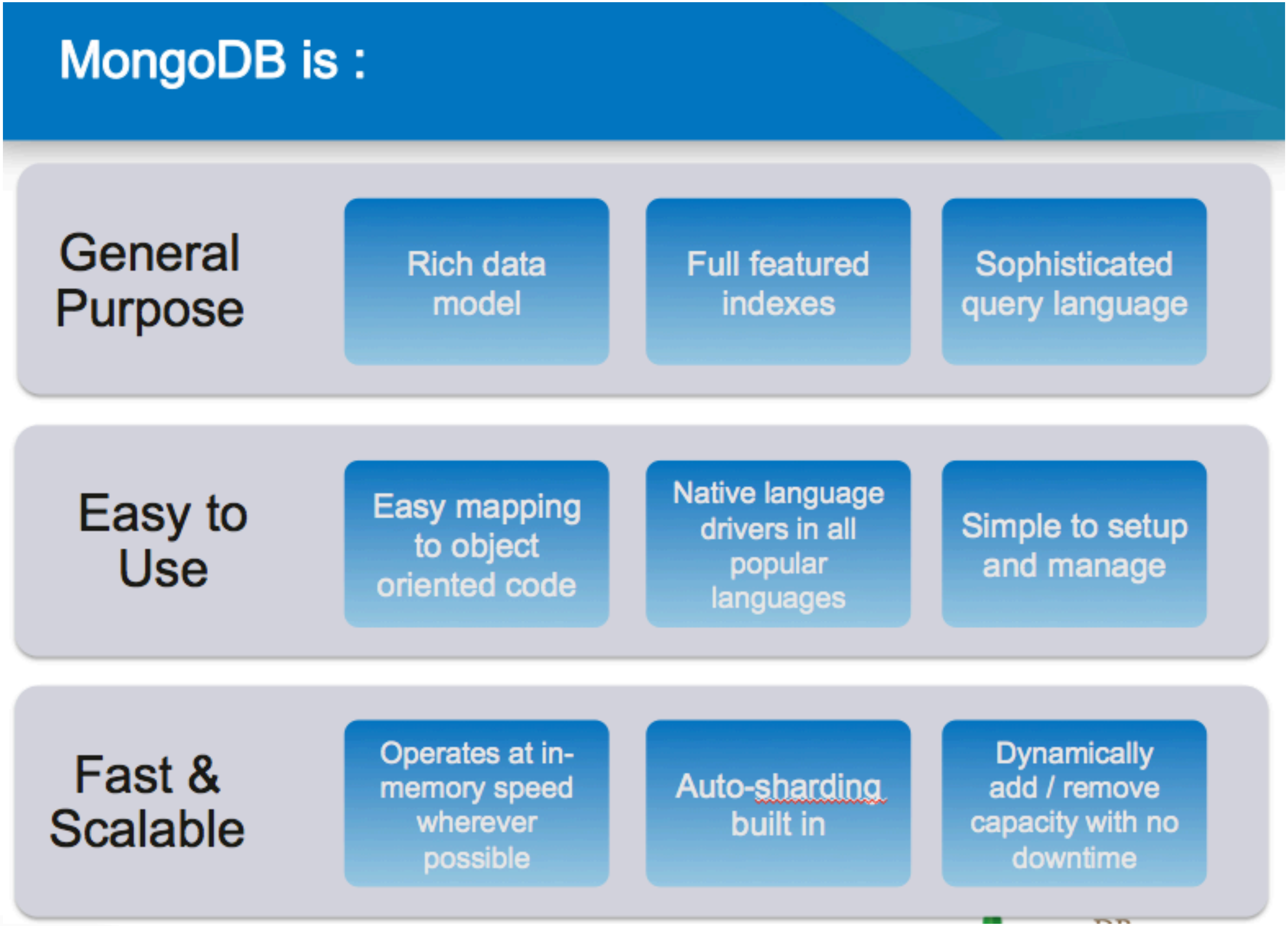
“bake” or “coffee” but not “cake”

$Text Score

- $Text returns a score for each matching document
- Score can be used in your query

```
db.articles.find(
   { $text: { $search: "cake" } },
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } ).limit(3)
```

For regular expression match use $regex operator



MongoDB is :

General Purpose	Rich data model	Full featured indexes	Sophisticated query language
Easy to Use	Easy mapping to object oriented code	Native language drivers in all popular languages	Simple to setup and manage
Fast & Scalable	Operates at in-memory speed wherever possible	Auto-sharding built in	Dynamically add / remove capacity with no downtime