

Document-Oriented Databases in Depth

Outline

- Introduction
- What is a Document
- DocumentDBs
- MongoDB
 - Data Model
 - Indexes
 - CRUD
 - Scaling
 - Pros and Cons


Document DB introduction

- Documents are the main concept.
- A Document-oriented database stores and retrieves documents (XML, JSON, BSON and so on).
- Documents are:
 - Self-describing
 - Hierarchical tree data structures (maps, collection and scalar values)

What is a Document DB?

- Document databases store documents in the value part of the key-value store where:
 - Documents are indexed using a BTree
 - and queried using a JavaScript query engine

```
{  
  name: "sue",  
  age: 26,  
  status: "A",  
  groups: [ "news", "sports" ]  
}
```



← field: value
← field: value
← field: value
← field: value

What is a Document DB?

```
{  
  "name": "Phil",  
  "age": 26,  
  "status": "A"  
}
```

```
{  
  "name": "Phil",  
  "age": 26,  
  "status": "A",  
  "citiesVisited" : ["Chicago", "LA", "San  
Francisco"]  
}
```

- Documents have differences in their attributes
- But belongs to the same collection
- This is different from relational databases where columns:
 - Stores the same type of values
 - Or null

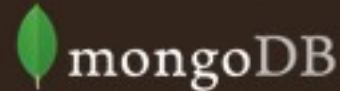
RDBMS vs Document DB: Terminology

Oracle	Document-Oriented
Database instance	DocumentDB instance
schema	database
table	collection
row	document
Rowid	_id
Join	DBRef

Document DBs

- MongoDB
- CouchDB
- RethinkDB
- RavenDB

RethinkDB

The MongoDB logo, featuring a green leaf icon to the left of the text "mongoDB" in a lowercase, sans-serif font.

A rock-solid database.

FoundationDB is a rock-solid, high performance database that provides NoSQL and SQL access.



Mongo DB

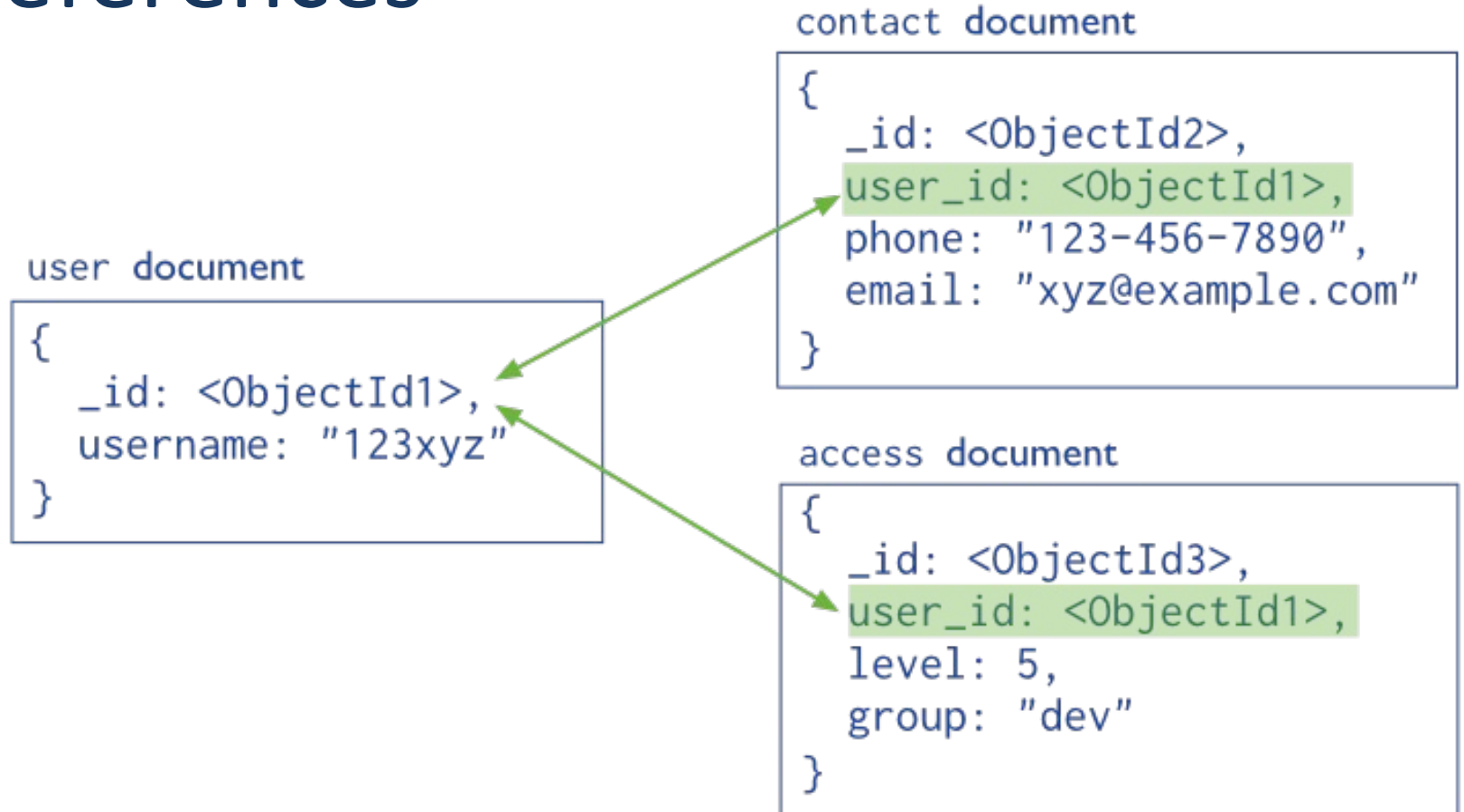
Documents: Data Model

Documents: Data Model

- Data has a flexible schema
- This helps in matching document to objects
 - Each document can match the fields of a document also if with different structure
- Data is represented as a map
- Relations can be represented as: *references* and *embedded documents*

Documents: Structure

References



Documents: Structure Embedded

```
{  
  _id: <ObjectId>,  
  username: "123xyz",  
  contact: {  
    phone: "123-456-7890",  
    email: "xyz@example.com"  
  },  
  access: {  
    level: 5,  
    group: "dev"  
  }  
}
```

Embedded sub-
document

Embedded sub-
document

Documents: Write Operations

- Writes are atomic at the document level
 - A Denormalized data model facilitates atomic write operations.
 - Normalizing the data over multiple collection would require multiple write operation that are not atomic.

Documents: Growth

- Each time a document is updated the modification are done changing affected attributes
- Each document has a maximum size of 16MB
- If the document size exceeds MongoDB relocates the document on disk.
- In MongoDB 3.0 this problem is minimized using the **Power of 2 Sized Allocation**

Documents: ObjectId

- ObjectId is a 12-byte BSON type, constructed using:
 - a 4-byte value representing the seconds since the Unix epoch,
 - a 3-byte machine identifier,
 - a 2-byte process id, and
 - a 3-byte counter, starting with a random value.
- It is an interesting approach to generate keys considering that documents are retrieved using document queries

Documents: Indexing

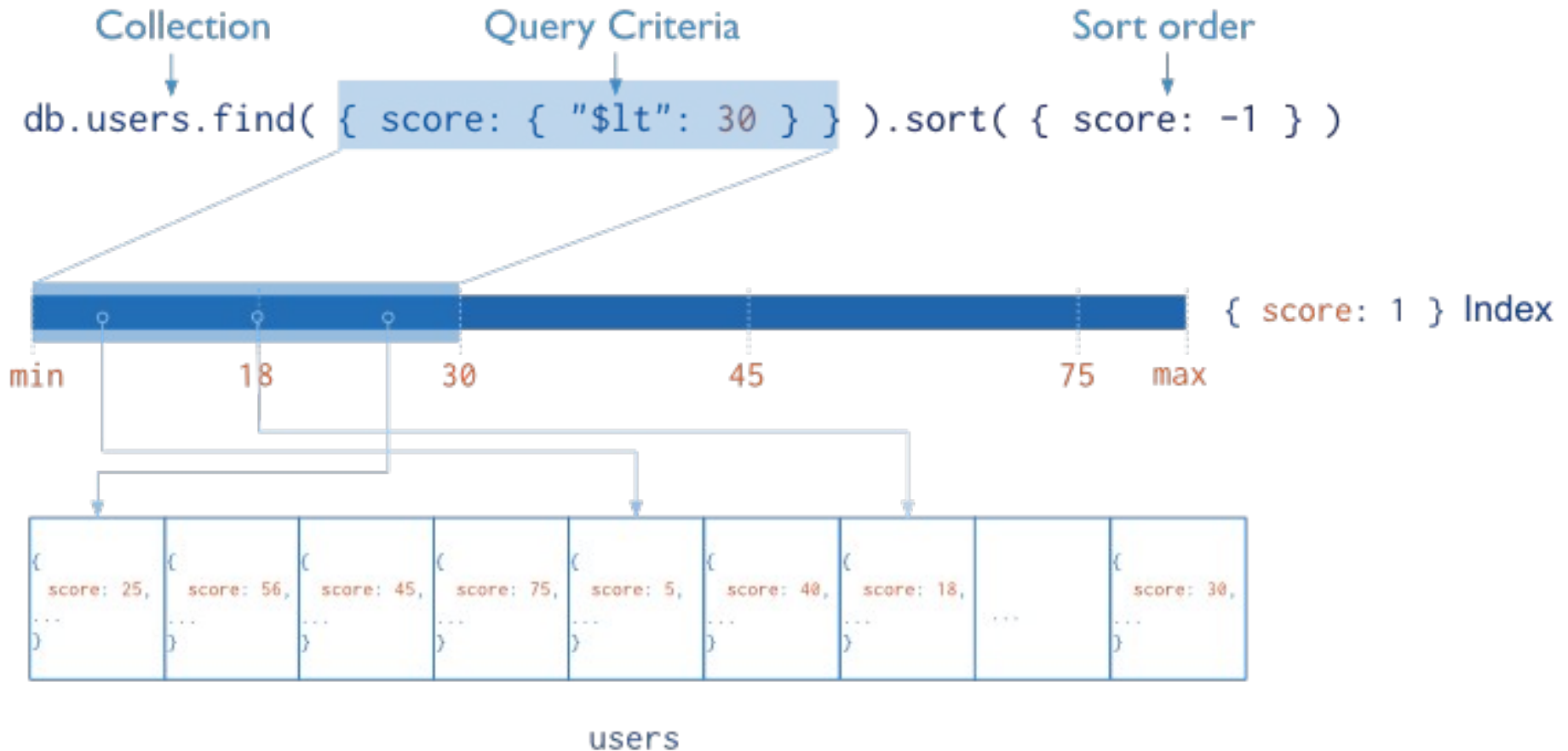
Documents: Indexing

- Indexes allows efficient queries on MongoDB.
- They are used to limit the number of documents to inspect
- Otherwise, it has to scan every document in a collection.
- By default MongoDB create indexes only on the `_id` field

Documents: Indexing

- Indexes are created using B-tree and stores data of fields ordered by values.
- In addition MongoDB returns sorted results by using the index.

Documents: Indexing



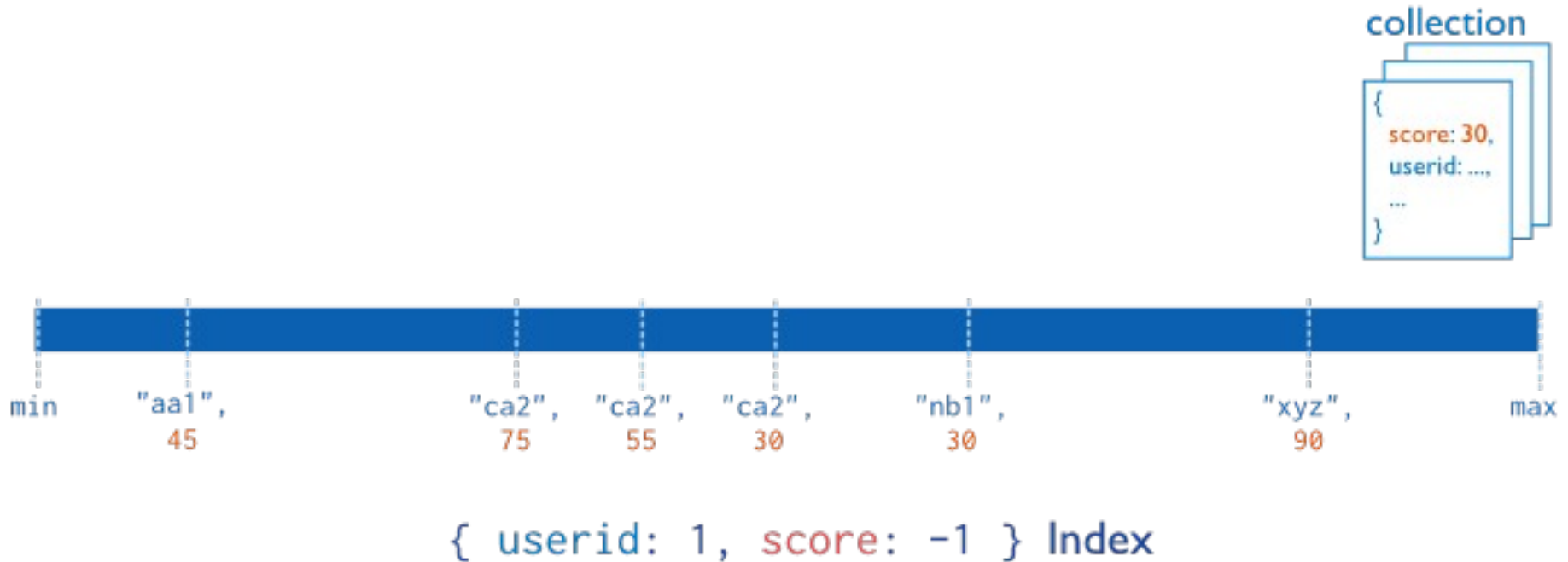
Documents: Index Types

- Single Field



Documents: Index Types

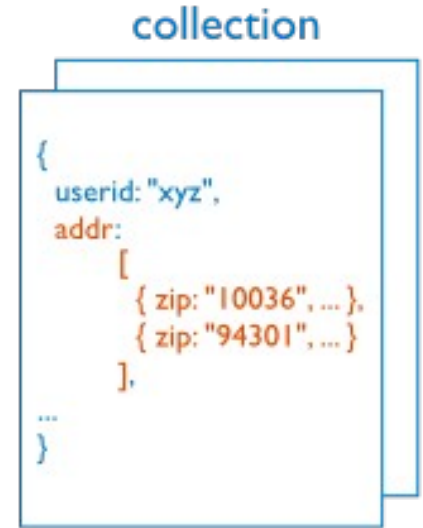
- Compound Index: indexed by attributes (left to right)



Documents: Index Types

Multikey Index:

- to index content in arrays



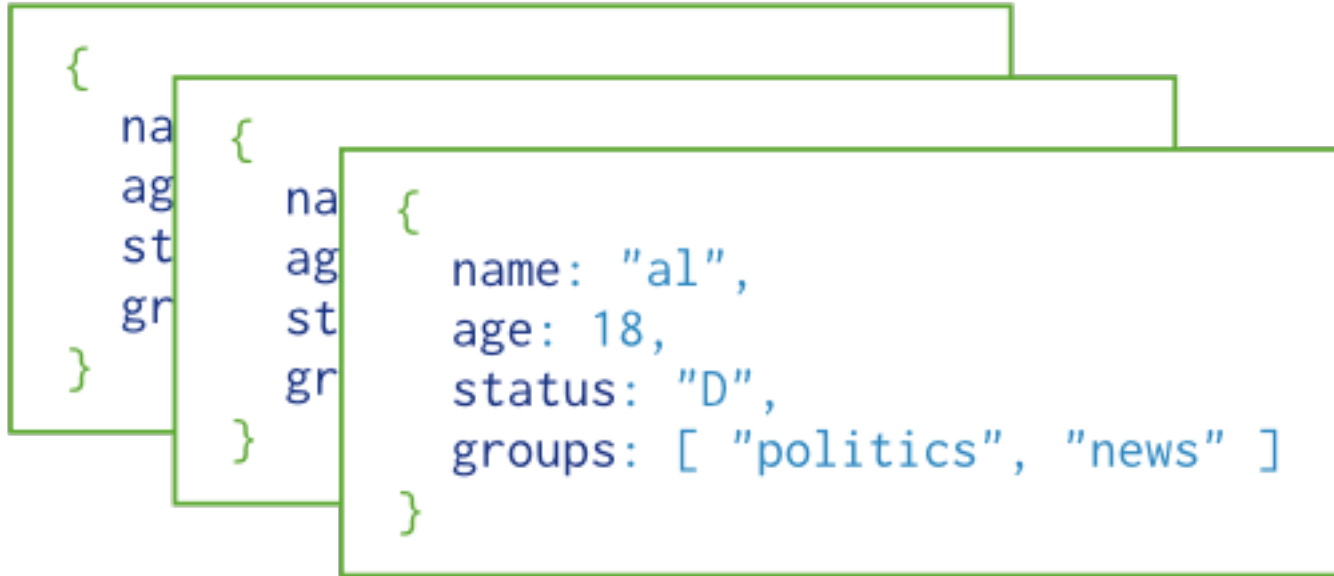
{ "addr.zip": 1 } Index

Documents: Index Types

- Geospatial Index: 2d and 2sphere indexes
- Text Indexes: performs tokenization, stopwords removal and stemming.
- Hashed Indexes: used to provide an hash based sharding

Documents: CRUD

Query a Collection



Collection

Queries

- Queries specify criteria, or condition that identify documents
- A query may include projections to specify the fields to return.
- It is possible to impose limits, skips and sort orders.

Query Interface

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
) .limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

The same query in SQL

```
SELECT _id, name, address  
FROM users  
WHERE age > 18  
LIMIT 5
```

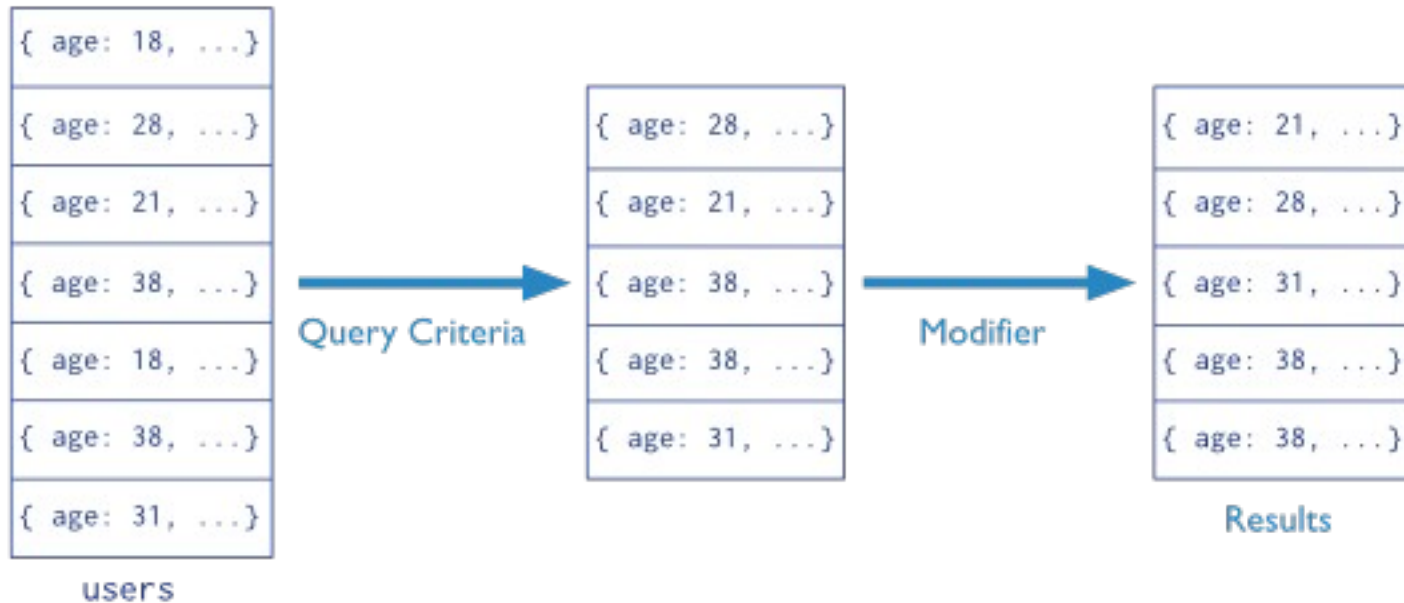
← projection
← table
← select criteria
← cursor modifier

Query Behavior

- All queries in MongoDB address a single collection.
- We impose limits, skips, and sort orders.
- The order of documents returned by a query is not defined unless you specify a *sort()*.
- Operations that modify existing documents (i.e. updates) use the same query syntax as queries to select documents to update.
- In aggregation pipeline, the *\$match* pipeline stage provides access to MongoDB queries.

Query Statements

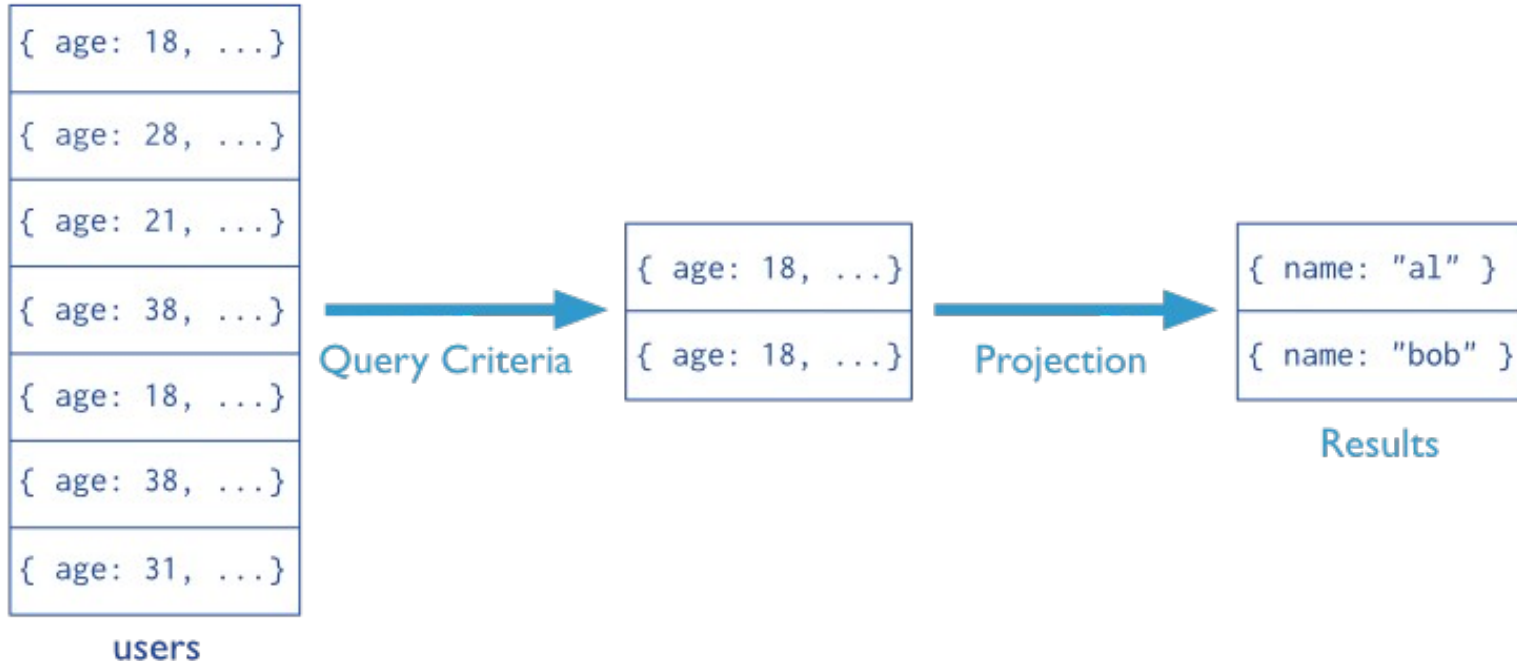
Collection Query Criteria Modifier
`db.users.find({ age: { $gt: 18 } }).sort({age: 1 })`



Projections

Collection Query Criteria Projection

```
db.users.find( { age: 18 }, { name: 1, _id: 0 } )
```



Cursors

- Each Query like *db.collection.find()* returns a cursor
- To access the documents, you need to iterate the cursor.
- By default MongoDB closes a cursor
 - after 10 minutes of inactivity
 - or if the client has exhausted the cursor

```
var myCursor = db.inventory.find().addOption(DBQuery.Option.noTimeout);
```

Data Modification

- Data modification refers to operations that create, update, or delete data. In MongoDB.
- These operations modify the data of a single collection.
- For the update and delete operations, it is possible to specify the criteria to select the documents to update or remove.

Insert

```
db.users.insert (  ← collection
{
  name: "sue",      ← field: value
  age: 26,           ← field: value
  status: "A"        ← field: value
}
)                  } document
```

Which corresponds to the SQL query

```
INSERT INTO users      ← table
      ( name, age, status ) ← columns
VALUES ( "sue", 26, "A" ) ← values/row
```

Update

```
db.users.update(  
  { age: { $gt: 18 } },  
  { $set: { status: "A" } },  
  { multi: true }  
)
```

← collection
← update criteria
← update action
← update option

```
UPDATE users  
SET      status = 'A'  
WHERE    age > 18
```

← table
← update action
← update criteria

Update: Example

Collection
↓
db.users.insert(
Document
↓
{
 name: "sue",
 age: 26,
 status: "A",
 groups: ["news", "sports"]
}
)

Document

```
{  
  name: "sue",  
  age: 26,  
  status: "A",  
  groups: [ "news", "sports" ]  
}
```

insert

Collection

{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }

users

Update Behavior

- By default, the `db.collection.update()` method updates a single document.
- However, with the `multi` option, `update()` can update all documents in a collection.
- If the `update()` method includes `upsert: true` and no documents match the query portion of the update operation, then the update operation creates a new document.

Remove

```
db.users.remove(  
    { status: "D" }  
)
```

← collection
← remove criteria

```
DELETE FROM users  
WHERE status = 'D'
```

← table
← delete criteria

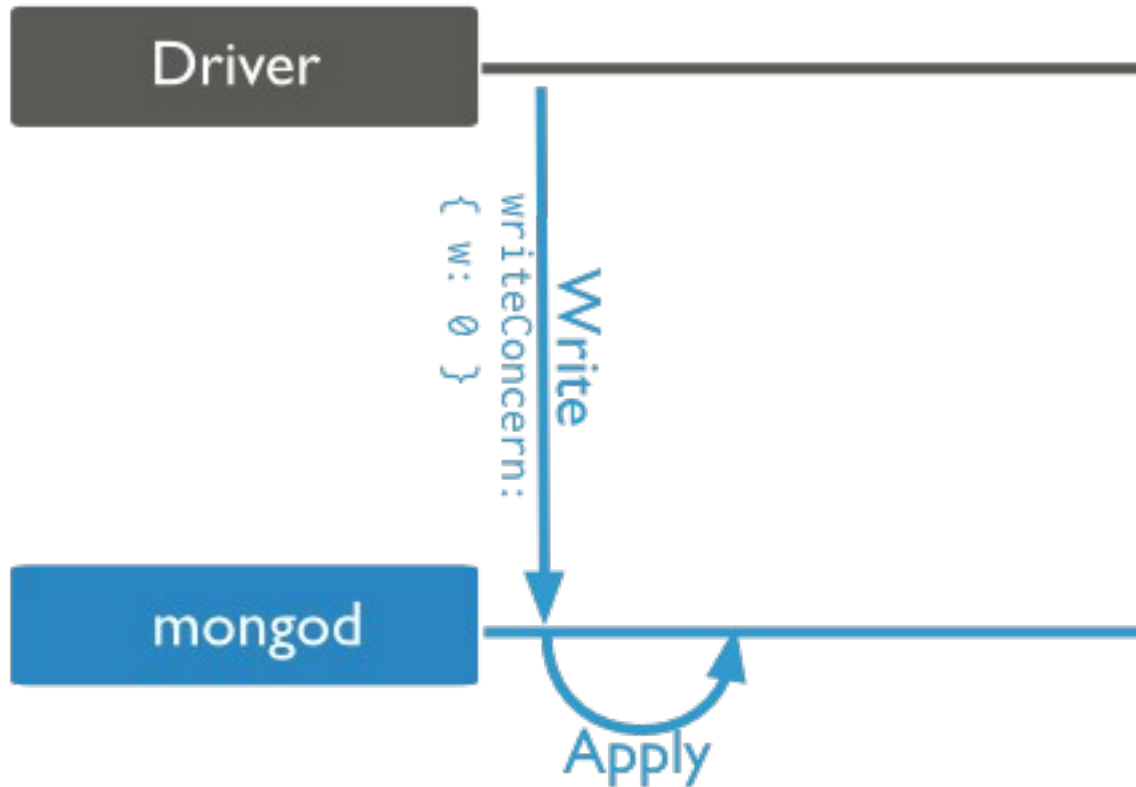
Remove Behavior

- By default, *db.collection.remove()* method removes all documents that match its query.
- However, the method can accept a flag to limit the delete operation to a single document.

Write Concern

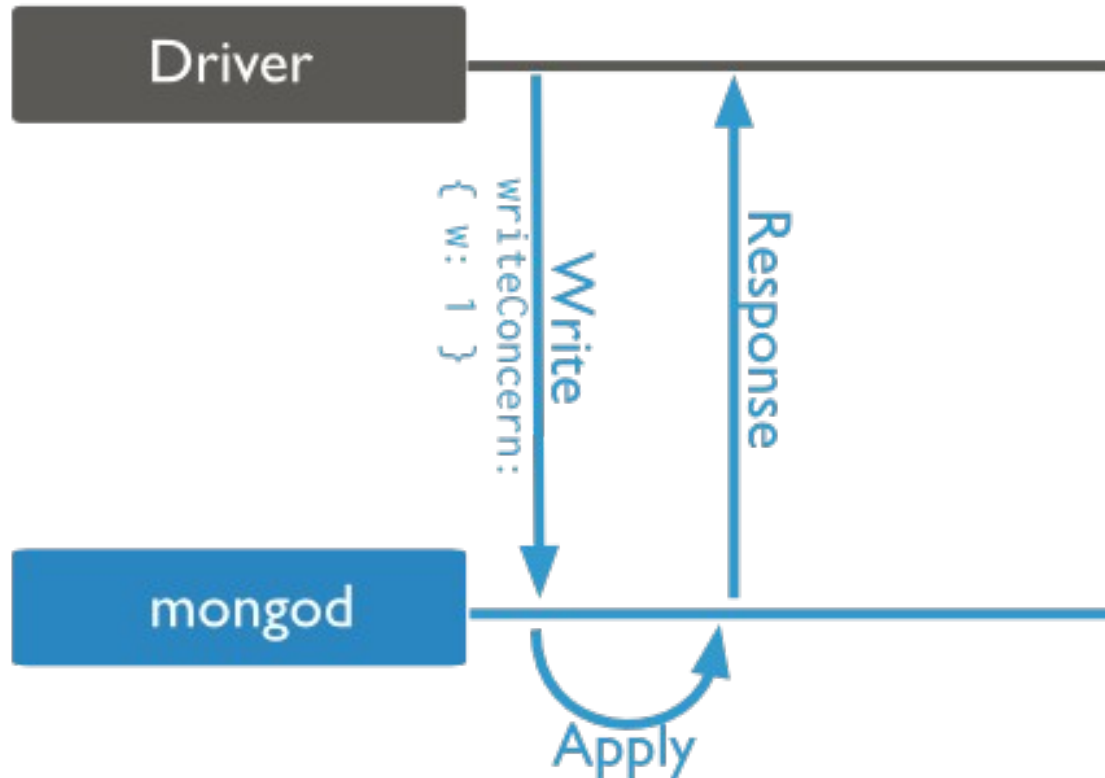
- There are different levels of guarantee for writes
- When inserts, updates and deletes have a weak write concern, write operations return quickly.
- In some failure cases, write operations issued with weak write concerns may not persist.
- With stronger write concerns, clients wait after sending a write operation for MongoDB to confirm the write operations.

Unacknowledged: Default Write



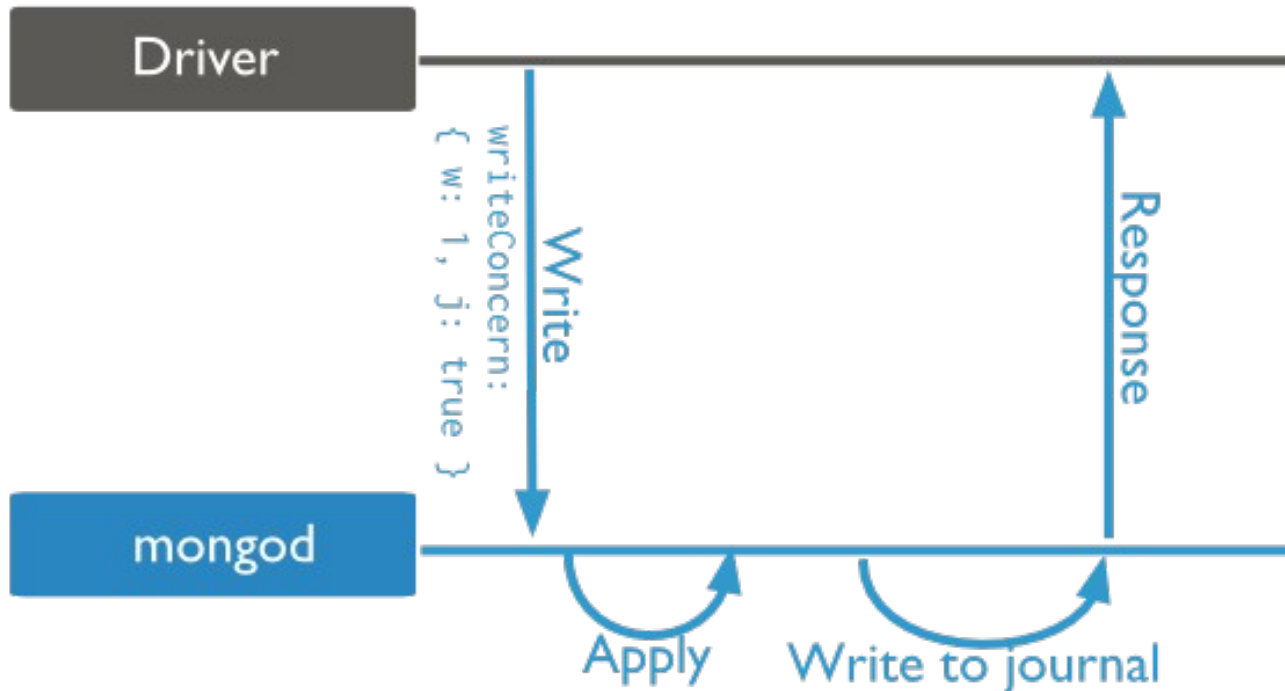
Acknowledged

- This is defined in the drivers

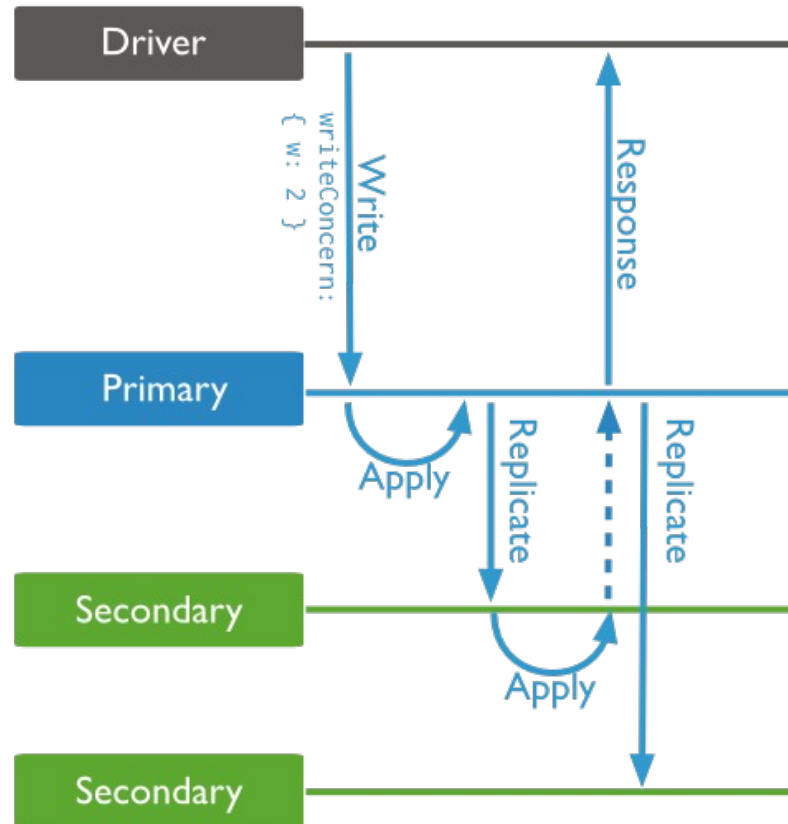


Journalled

- Operation are acknowledges only after operations are saved to the journal.



Replica Acknowledged



Scaling

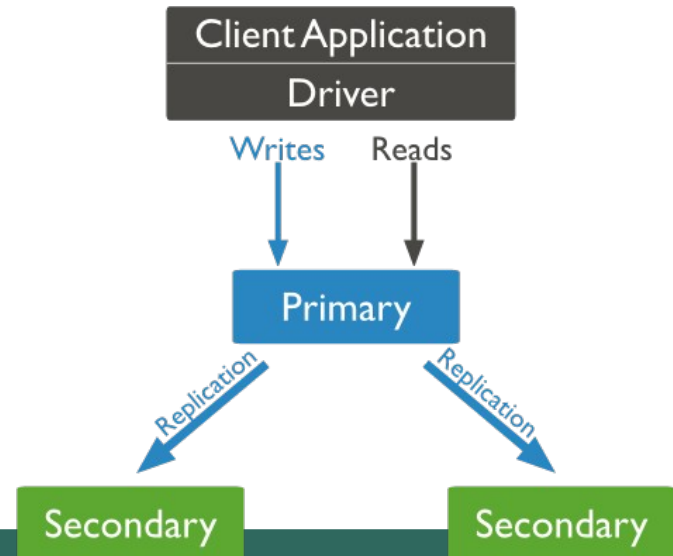
Scaling

- The idea of scaling is to add more node to the cluster of nodes.
- There are two different context to consider:
 - Heavy reads
 - Heavy writes

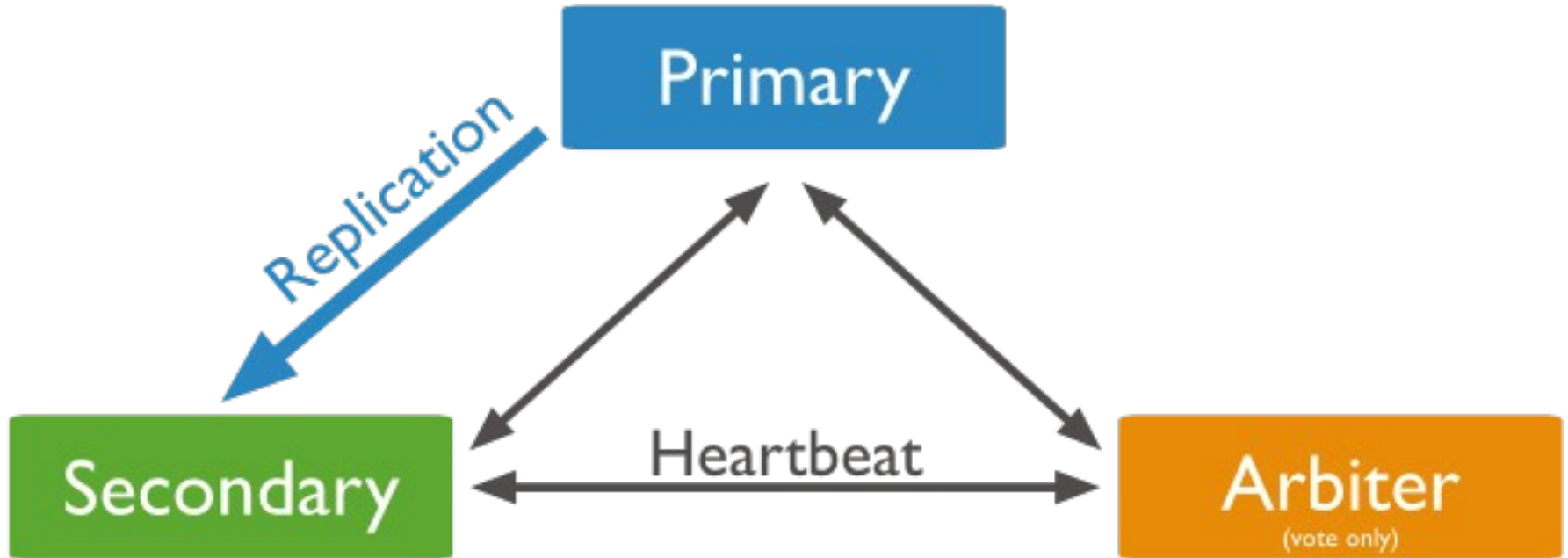
Scaling: Heavy Reads

- Scaling here can be achieved by adding more read slaves
- All the reads can be directed to the slaves.
- When a node is added it will sync with the other nodes.
- The advantage of this setting is that we do not need to stop the cluster.

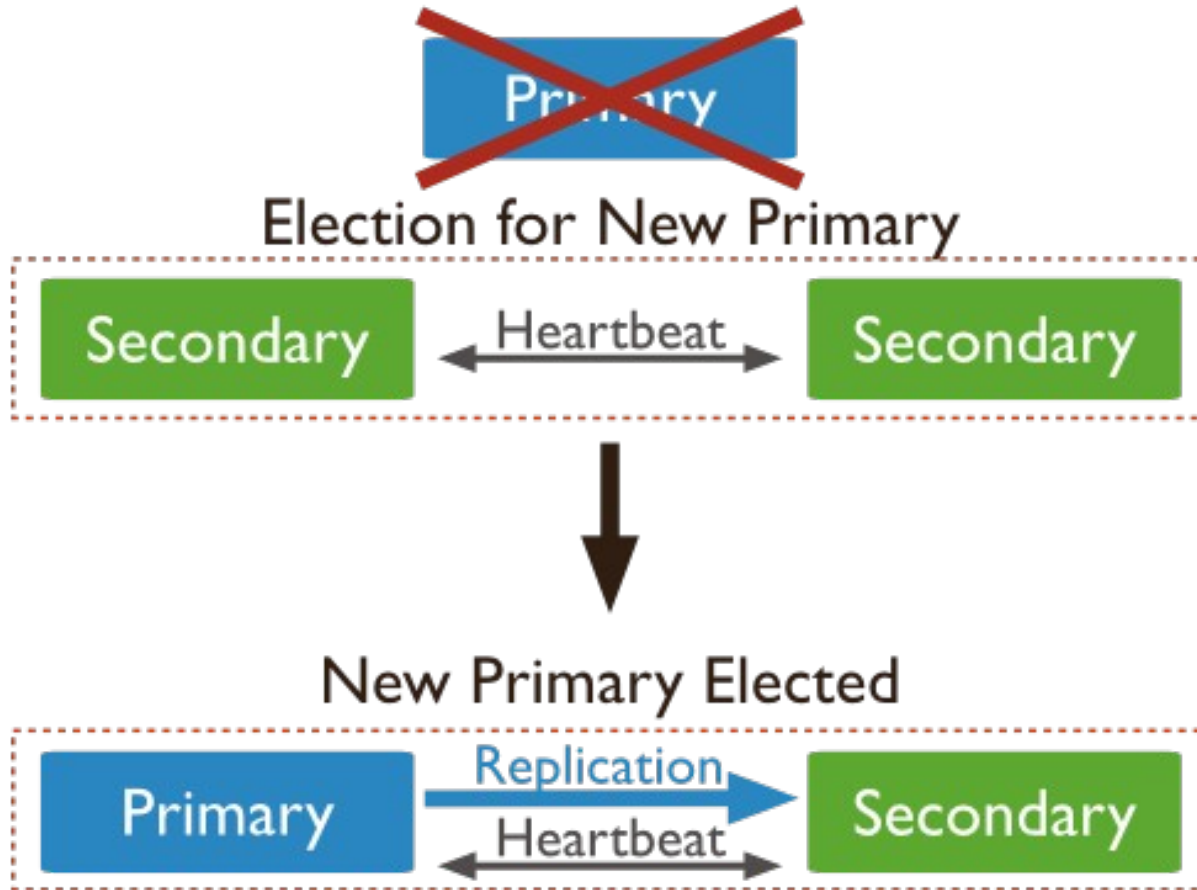
```
rs.add("mongo_address:27017")
```



Data Replication



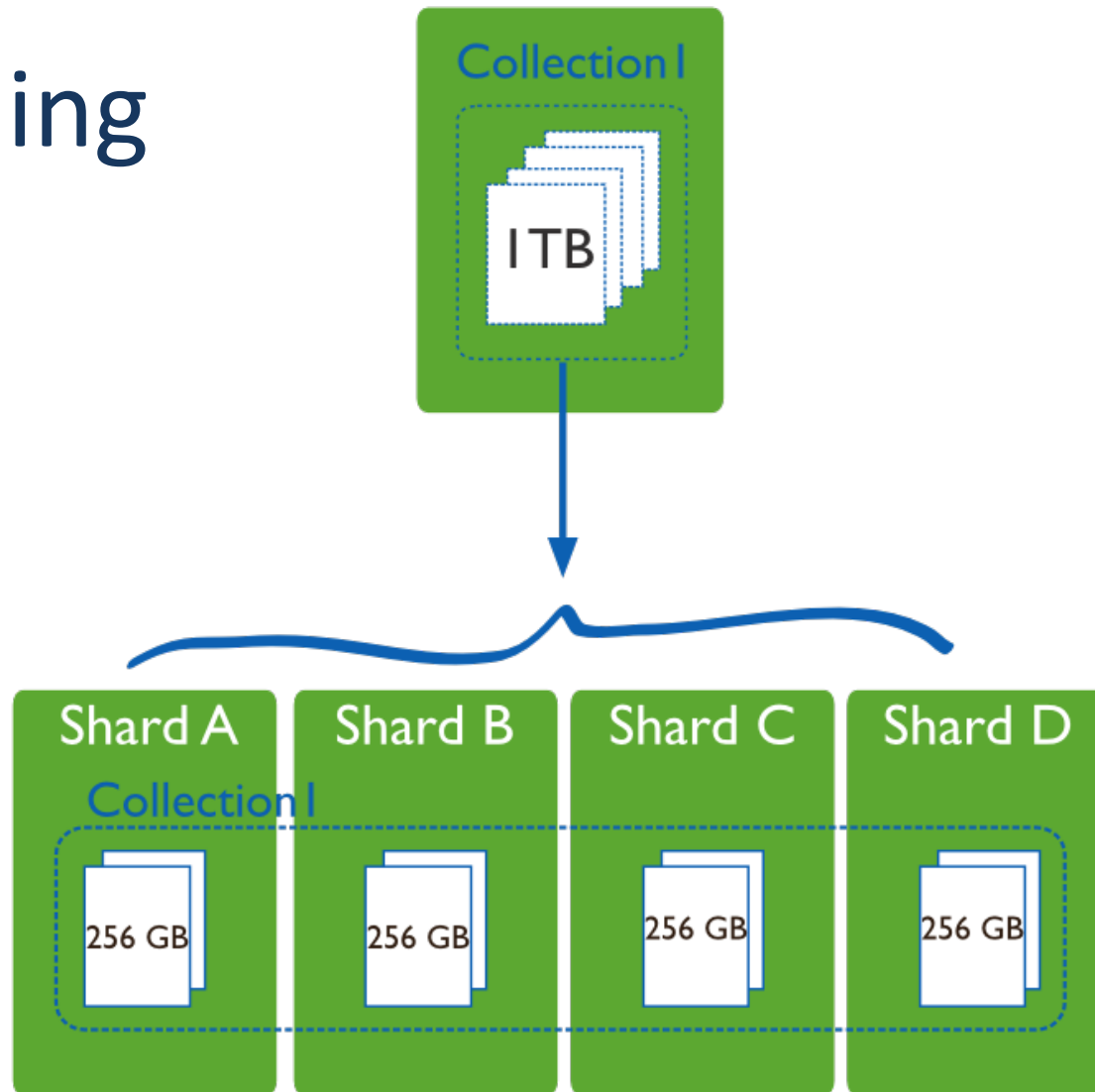
Automatic Failover



Scaling: Heavy Writes

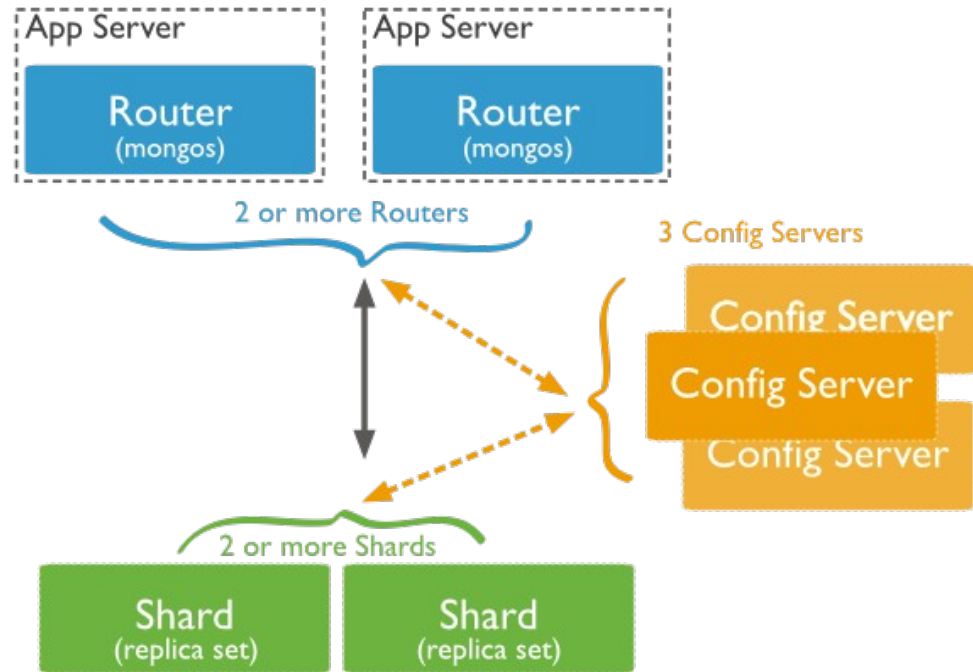
- We can start using the **Sharding** feature.
- Sharding, or horizontal scaling divides the data set and distributes the data over multiple servers.
- Each shard is an independent database, and collectively, the shards make up a single logical database.

Sharding



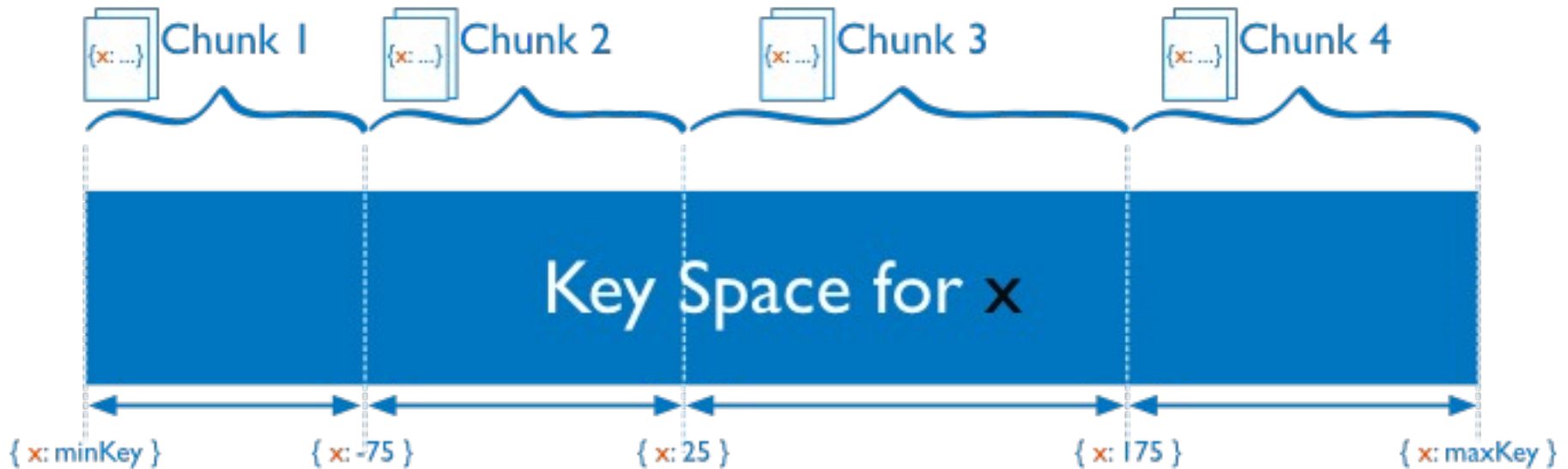
Sharding in MongoDB

- Shard: store the data
- Query Routers: interface to client and direct queries
- Config Server: store cluster's metadata.



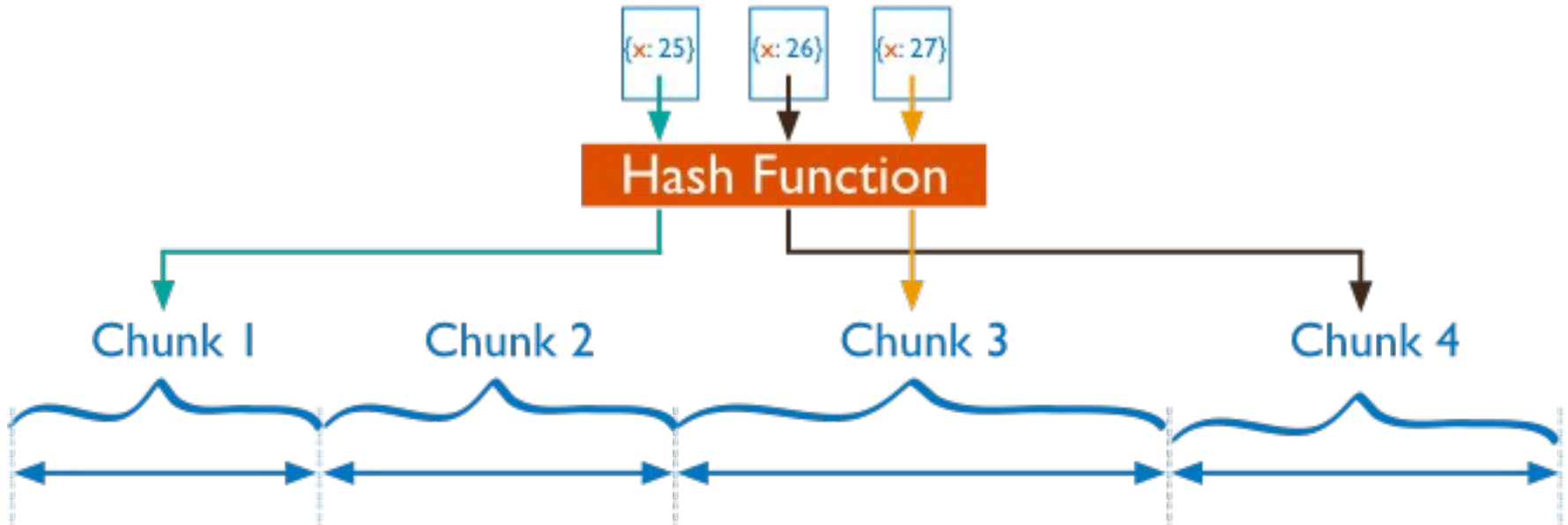
Range Based Sharding

- MongoDB divides the data set into ranges determined by the shard key values to provide range based partitioning.



Hash Based Sharding

- MongoDB computes a hash of a field's value, and then uses these hashes to create chunks



Performance Comparison

- Range based partitioning supports more efficient range queries.
- However, range based partitioning can result in an uneven distribution of data.
- Hash based partitioning, by contrast, ensures an even distribution of data at the expense of efficient range queries.

Other DocumentDBs

CouchDB

- Written in Erlang.
- Documents are stored using JSON.
- The query language is in Javascript and supports MapReduce integration.
- One of its distinguishing features is multi-master replication.
- ACID: It implements a form of Multi-Version Concurrency Control (MVCC) in order to avoid the need to lock the database file during writes.
- CouchDB does not guarantees (eventual) consistency to be able to provide both availability and partition tolerance.

CouchDB: Features

- Master-Master Replication - Because of the append-only style of commits.
- Reliability of the actual data store backing the DB (Log Files)
- Mobile platform support. CouchDB actually has installs for iOS and Android.
- HTTP REST JSON interaction only. No binary protocol

RethinkDB

- The RethinkDB server is written in C++ and runs on 32-bit and 64-bit Linux systems.
- It is a JSON based database.
- It is characterized by push update queries.
- It supports a MapReduce style API, and geospatial queries.

```
r.table('tv_shows').insert([ { name: 'Star Trek TNG', episodes: 178 },  
                             { name: 'Battlestar Galactica', episodes: 75 } ])
```

RavenDB

- It is a DocumentDB written in C#



Schema-free

Just store your stuff.
With Raven, you're not
constrained by rigid database
schemas.



Scalable

Going big? No problem.
Raven supports replication,
sharding, and multi-tenancy to
work with your big data.



Transactional

ACID is good.
In RavenDB, all operations
performed on documents are
fully transactional.



High Performance

Speed-obsessed.
Self-tuning, intelligent
indexes, optimized for blazing
fast reads, never blocked by
locks.



Easy to use

Built in .NET, for .NET.
Raven has first-class LINQ
support with an idiomatic,
clean .NET API served over
HTTP.



Extensible

Yes, Raven can do that.
Use bundles like versioning or
encryption, or extend RavenDB
yourself using well-defined
extensibility points.



Designed with Care

SELECT N+1 begone!
Raven avoids the gotchas of
ORMs and first-gen NoSQL
databases, helping you fall
into the pit of success.



Lean, powerful, productive

Read more about Raven's
awesomeness in the full
feature list.

Document Store: Advantages

- Documents are independent units
- Application logic is easier to write. (JSON).
- Schema Free:
 - Unstructured data can be stored easily, since a document contains whatever keys and values the application logic requires.
 - In addition, costly migrations are avoided since the database does not need to know its information schema in advance.

Suitable Use Cases

- **Event Logging:** where we need to store different types of event (order_processed, customer_logged).
- **Content Management System:** because the schema-free approach is well suited
- **Web analytics or Real-Time Analytics:** useful to update counters, page views and metrics in general.

When Not to Use

- **Complex Transactions:** when you need atomic cross-document operations, but we can use RavenDB or RethinkDB
- **Queries against Varying Aggregate Structure:** that is when the structure of your aggregates vary because of data continuous data evolutions