# COMP5338 – Advanced Data Models

**Week 5:** MongoDB – Replication and Sharding

Dr. Ying Zhou
School of Computer Science

# Outline

- **Replication**
  - **Replica Set**
  - **Write Concern**
  - **Read Preference**
  - **Read Concern**
- **Sharding**

# Replication in MongoDB

- MongoDB uses replication to achieve durability, availability and/or read scalability.
- A basic replication component in MongoDB is called a **`replica set`**
- A replica set contains several data bearing nodes and optionally one **`arbiter`** node.
- The data bearing nodes are organized following traditional master/slave replication mechanism
  - One primary and one to multiple secondary members
- The arbiter node does not contain data, it is only used for primary election to establish majority

# Primary and Secondary Members

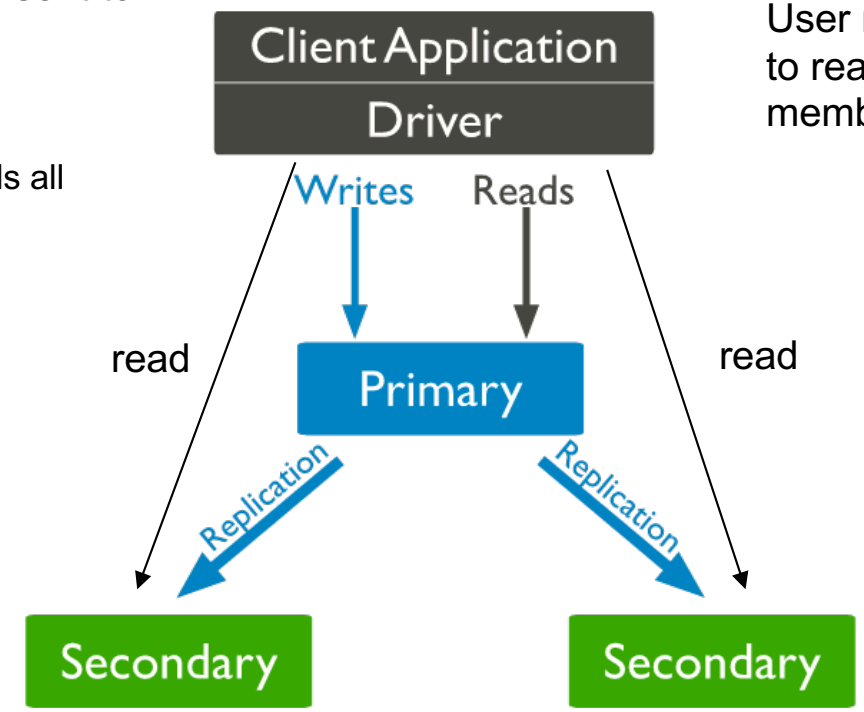
Primary and secondary contain the same data set

By default all reads/writes are sent to primary member only

In write operation, primary records all changes to its data set in **`oplog`** (operation log)

The operation log is then sent to secondary member

Secondary member uses the operation log to update its own data set

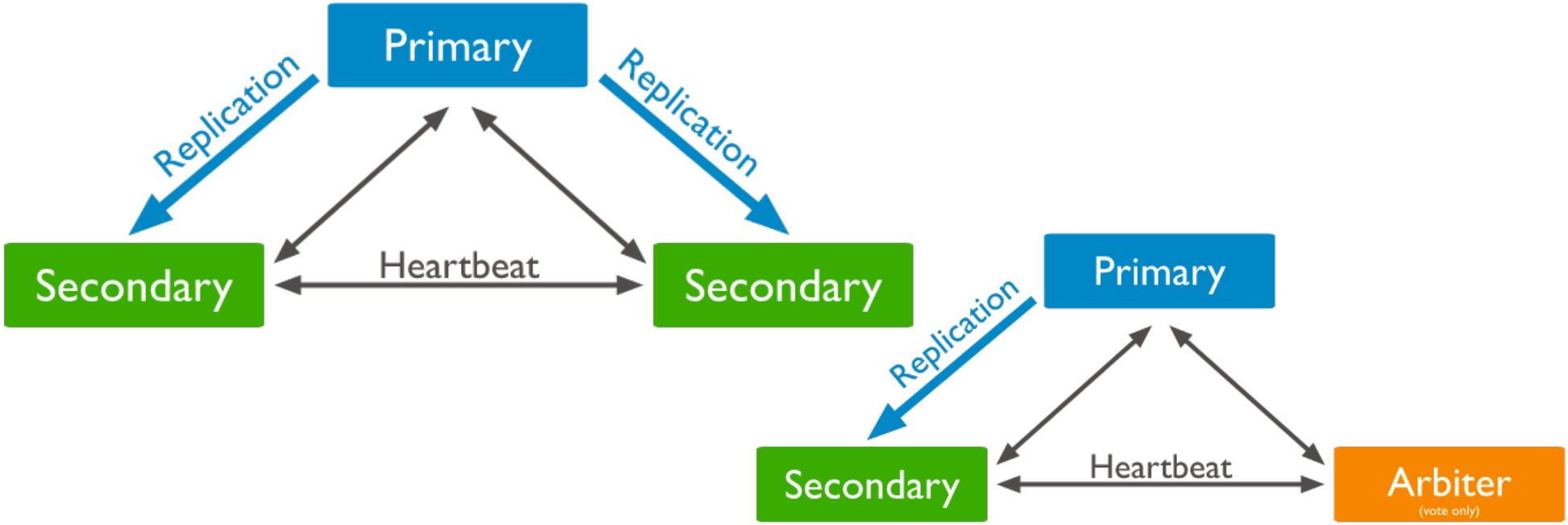



User may indicate that it is safe to read from secondary member;

There is replication lag between primary and secondary. The data set on secondary may not reflect the latest change

# Member Communication

- Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds, the other members mark the delinquent member as inaccessible.




https://docs.mongodb.com/manual/core/replica-set-elections/#replica-set-elections

# Fault Tolerance on Primary

- When a primary is deemed inaccessible, other members will hold a primary election
- The replica set cannot process *write* operations until the election completes successfully.
  - One secondary is elected as the primary
- Network Partition
  - A network partition may segregate a **primary** into a partition with a *minority* of nodes.
  - When the primary detects that it can only see a minority of nodes in the replica set, the primary steps down as primary and becomes a secondary.
  - Independently, a member in the partition that can communicate with a majority of the nodes (including itself) holds an election to become the new primary.

# Replica Set Read/Write Semantics

- By *default*, read operations are answered by the primary member
- Client can specify "**Read Preference**" to read from different members
  - Primary(*default*), secondary, nearest, etc
- Client can also specify "**Read Concern**" to indicate the consistency level they expect.
- By *default*, write operation is considered successful when it is written on the primary member
- To maintain consistency requirements, client can specify different levels of "**Write Concern**"

# Write Concern

- "**Write concern** describes *the level of acknowledgment* requested from MongoDB for write operations to a ***standalone mongod*** or to ***replica sets*** or to ***sharded clusters.***"
- The specification is attached to a write query with the format
  - { **w**: <value>, **j**: <boolean>, **wtimeout**: <number> }
    - **w**: how many or whose acknowledgements receive before replying clients
      - The value can be a number, 'majority' or custom name
    - **j**: send the acknowledgement before or after logging to disk
      - true, false or unspecified
    - **wtimeout** : time limit in ms to prevent write operations from blocking indefinitely
- The complete signature of `updateMany` is:

```
db.collection.updateMany(
   <filter>,
   <update>,
   {
     upsert: <boolean>,
     writeConcern: <document>,
     collation: <document>,
     arrayFilters: [ <filterdocument1>, ... ],
     hint:  <document|string>        // Available starting in MongoDB 4.2.1
   }
)
```

# Write Concern Specification

- Write concern values control how soon a write request will be returned to the client
- Remember any database write operation involves
  - In memory update
  - log appending
  - data file update
    - happens after the write acknowledgement in MongoDB
- The write concern specifies
  - when should each db instance send acknowledgement
    - controlled by **`j`** value
  - How many db instances should acknowledge before acknowledging the client
    - controlled by **`w`** value
  - The maximum time client should wait for an acknowledgement.
    - Controlled by **`wtimeout`** value

# Write Acknowledge Behavior

- The meaning of `j` value
  - **true**: the db instance waits until log is successfully appended to send acknowledgement
  - **false**: the db instance sends acknowledge once the memory is updated
  - *Unspecified*: the meaning depends on db environment and other setting
- The meaning of `w` value
  - 0: no acknowledgement is required
  - 1: the standalone db instance need to acknowledge or the primary member needs to acknowledge
  - > 1: only applicable in replica set, require the specified member(s) to acknowledge
  - 'majority': require majority of the members in replica set to acknowledge; or it can override `j` setting in standalone setting.

# Write Concern Behaviour

- Default write concern is `{w=1}`
  - Standalone: acknowledge after in memory update
  - Replica set: acknowledge after in memory update in primary
- **`wtimeout`** is only applicable for w values greater than 1
- **`wtimeout`** causes write operations to return with an error after the specified limit, **but** it does not necessarily means the write operation is failed. It only indicates that the required number of acknowledgements are not received with in the limit.
  - MongoDB will not perform any roll back
  - Applications should not interpret time out error as fail and act accordingly
- Unspecified wtimeout may block the write operation indefinitely

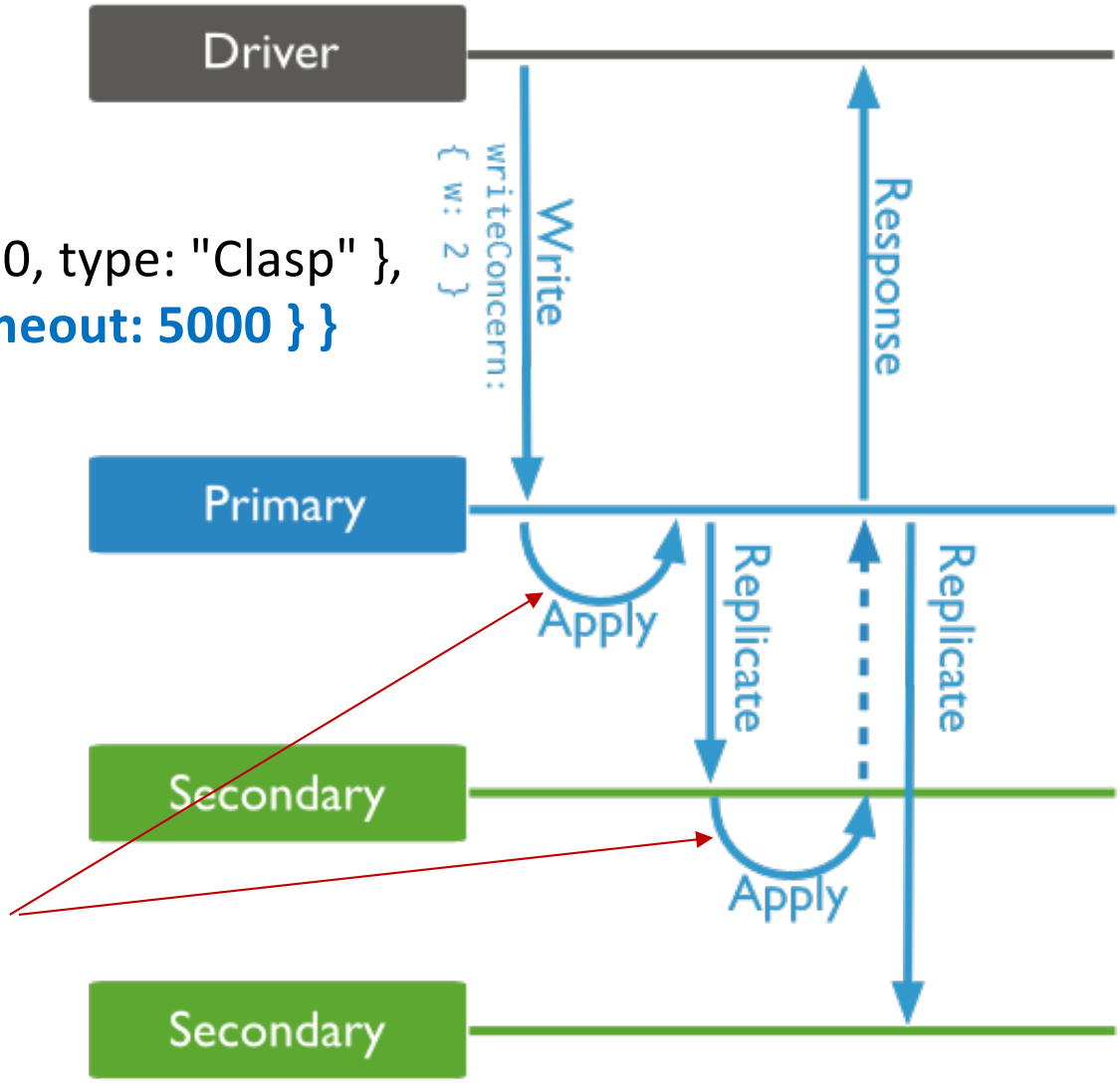
# Replica Set Write Concern Example

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

The j value is unspecified

Acknowledgment behaviour is equivalent to j=false

Member send acknowledgement after applying the in memory update




http://docs.mongodb.org/manual/core/replica-set-write-concern/

# Read Preference

- Read preference describes how MongoDB clients ***route*** read operations to the members of a replica set.

| Mode | Description |
|---|---|
| Primary | Default one. All operations read from the primary node |
| PrimaryPreferred | in most situations, operations read from the primary but if it is unavailable, operations read from secondary members. |
| Secondary | All operations read from the secondary members of the replica set. |
| SecondaryPreferred | In most situations, operations read from secondary members but if no secondary members are available, operations read from the primary. |
| nearest | Operations read from member of the replica set with the least network latency, irrespective of the member's type. |

# Read Isolation (Read Concern)

- How read operation is carried out **inside** db engine to control the consistency and availability
- There are many levels
- New release may introduce new level(s) to satisfy growing consistency requirement
- To understand what sort of consistency you will get, all three properties need to be looked at
  - Write Concern
  - Read Preference
  - Read Concern

# Read Concern Levels

- *local*: the query returns data from the instance with no guarantee that the data has been written to a majority of the replica set members (i.e. may be rolled back)
  - Default for read against primary, or reads against secondaries if the reads are associated with causally consistent sessions

Read preference level

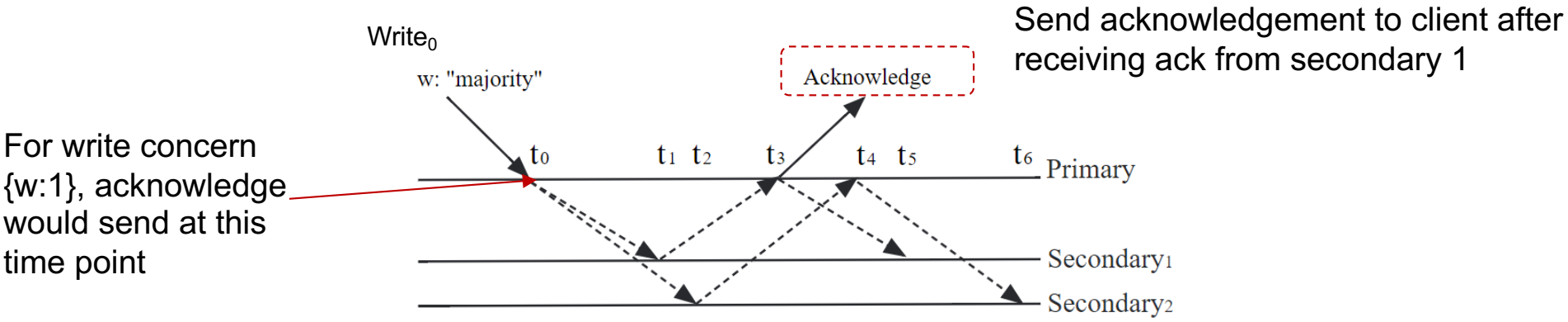
- *available*: the query returns data from the instance with no guarantee that the data has been written to a majority of the replica set members
  - Default for read against secondaries if the reads are **not** associated with causally consistent sessions
- *majority*: The query returns the data that has been acknowledged by a majority of the replica set members. The documents returned by the read operation are durable, even in the event of failure.
- linearizable
- Snapshot
- Setting read concern level in find query:
  `db.collection.find().readConcern(<level>)`

# Replica Set Default Behaviour

- Write concern:
  - Write is considered successful when it is written on the **memory** of primary member
    - w:1, j:false
  - replication to the secondary members happen asynchronously
- Read Preference:
  - primary: All read operations are sent to the primary
- Read Concern
  - Local: returns data from the instance (in this case, the primary) with no guarantee that the data has been written to a majority of the replica set members
- Read Uncomitted
  - Client can see the results of writes before the writes are durable
    - Concurrent read may see the result of a write before the write is acknowledged to the client
    - Client may see data that are subsequently rolled back during replica set failovers

# Customized Behaviour: Write: majority

- Write concern: “majority”
  - Requests acknowledgement that write operations have propagated to the majority of data bearing nodes, including the primary

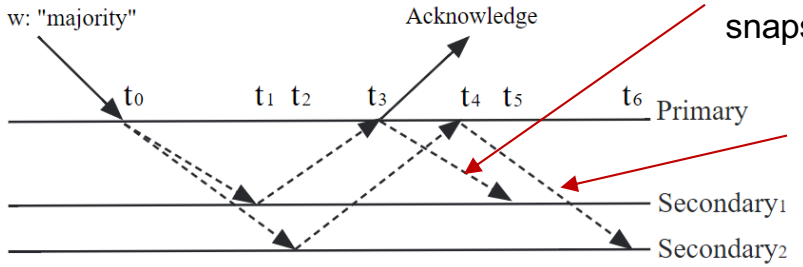



Assume the following:

- All writes prior to $Write_0$ have been successfully replicated to all members.
- $Write_{prev}$ is the previous write before $Write_0$.
- No other writes have occurred after $Write_0$.

https://docs.mongodb.com/manual/reference/read-concern-local/#readconcern.%22local%22

# Write: majority example case




It also notifies s1 to update its relevant snapshots

It also notifies s2 to update its relevant snapshots

| Time | Event | Most Recent Write | Most Recent w: "majority" write |
|---|---|---|---|
| $t_0$ | Primary applies $Write_0$ | Primary: **$Write_0$**<br>$Secondary_1$: $Write_{prev}$<br>$Secondary_2$: $Write_{prev}$ | Primary: **$Write_{prev}$**<br>$Secondary_1$: $Write_{prev}$<br>$Secondary_2$: $Write_{prev}$ |
| $t_1$ | $Secondary_1$ applies $write_0$ Send acknowledgement to primary | Primary: **$Write_0$**<br>$Secondary_1$: $Write_0$<br>$Secondary_2$: $Write_{prev}$ | Primary: **$Write_{prev}$**<br>$Secondary_1$: $Write_{prev}$<br>$Secondary_2$: $Write_{prev}$ |
| $t_2$ | $Secondary_2$ applies $write_0$ Send acknowledgement to primary | Primary: **$Write_0$**<br>$Secondary_1$: $Write_0$<br>$Secondary_2$: $Write_0$ | Primary: **$Write_{prev}$**<br>$Secondary_1$: $Write_{prev}$<br>$Secondary_2$: $Write_{prev}$ |
| $t_3$ | Primary is aware of successful replication to $Secondary_1$ and **sends acknowledgement to client** | Primary: **$Write_0$**<br>$Secondary_1$: $Write_0$<br>$Secondary_2$: $Write_0$ | Primary: **$Write_0$**<br>$Secondary_1$: $Write_{prev}$<br>$Secondary_2$: $Write_{prev}$ |
| $t_4$ | Primary is aware of successful replication to $Secondary_2$ | Primary: **$Write_0$**<br>$Secondary_1$: $Write_0$<br>$Secondary_2$: $Write_0$ | Primary: **$Write_0$**<br>$Secondary_1$: $Write_{prev}$<br>$Secondary_2$: $Write_{prev}$ |
| $t_5$ | $Secondary_1$ receives notice (through regular replication mechanism) to update its snapshot of its most recent w: "majority" write | Primary: **$Write_0$**<br>$Secondary_1$: **$Write_0$**<br>$Secondary_2$: $Write_0$ | Primary: **$Write_0$**<br>$Secondary_1$: **$Write_0$**<br>$Secondary_2$: $Write_{prev}$ |
| $t_6$ | $Secondary_2$ receives notice (through regular replication mechanism) to update its snapshot of its most recent w: "majority" write | Primary: **$Write_0$**<br>$Secondary_1$: **$Write_0$**<br>$Secondary_2$: **$Write_0$** | Primary: **$Write_0$**<br>$Secondary_1$: **$Write_0$**<br>$Secondary_2$: **$Write_0$** |

# Multiple Versions of Data Item

- MongoDB storage engine uses **M**ulti **V**ersion **C**oncurrency **C**ontrol (MVCC) to provide concurrent access to the database.
- "When an MVCC database needs to update a piece of data, it will not overwrite the original data item with new data, but instead creates a newer version of the data item. Thus there are multiple versions stored"

  [https://en.wikipedia.org/wiki/Multiversion_concurrency_control]
- Read operation can specify which version to use
- MongoDB achieves document level atomicity
  - At the start of write operation, WiredTiger provides a snapshot of the document to and update that snapshot accordingly

# Concurrent Access

```
{
  _id: xxxx,
  cid: 1,
  uid: 1,
  items: [
            { name: “A”,
              price: 12.0,
              q: 1},
            { name: “B”,
              price: 10.0,
              q: 2}
            ]
}
```

```
orders.updateOne(
  {cid:1},
  {$set:
    {“items.0.price”: 11.5,
     “items.1.q”: 1}
   }
 )
```

create a snapshot

make update here

```
orders.find({cid:1})
```

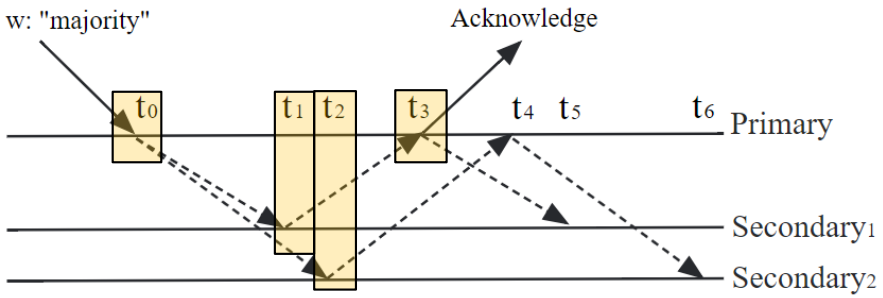
The read query can see original version

Or updated version

```
{
  _id: xxxx,
  cid: 1,
  uid: 1,
  items: [
            { name: “A”,
              price: 11.5,
              q: 1},
            { name: “B”,
              price: 10.0,
              q: 1}
            ]
}
```

# Read Concern: *local* example

Read Preference:
*Primary*,
*PrimaryPreferred*,
*SecondaryPreferred*,
*Nearest*

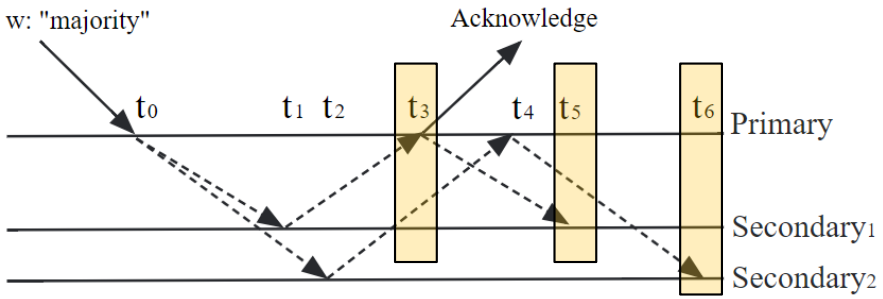



Read uncommitted before t3

| Read Target | Time T | State of Data |
|---|---|---|
| Primary | After $t_0$ | Data reflects $Write_0$. |
| $Secondary_1$ | Before $t_1$ | Data reflects $Write_{prev}$ |
| $Secondary_1$ | After $t_1$ | Data reflects $Write_0$ |
| $Secondary_2$ | Before $t_2$ | Data reflects $Write_{prev}$ |
| $Secondary_2$ | After $t_2$ | Data reflects $Write_0$ |

Read Concern: available has similar behaviour

# Read Concern: *majority* example




Primary has the most recent update $Write_0$ since $t_0$, but before $t_3$ it knows that majority of the replica has the previous value $Write_{prev}$

| Read Target | Time T | State of Data |
|---|---|---|
| Primary | Before $t_3$ | Data reflects $Write_{prev}$ |
| Primary | After $t_3$ | Data reflects $Write_0$ |
| $Secondary_1$ | Before $t_5$ | Data reflects $Write_{prev}$ |
| $Secondary_1$ | After $t_5$ | Data reflects $Write_0$ |
| $Secondary_2$ | Before or at $t_6$ | Data reflects $Write_{prev}$ |
| $Secondary_2$ | After $t_6$ | Data reflects $Write_0$ |

| | | | |
|---|---|---|---|
| $t_2$ | $Secondary_2$ applies $write_0$ | Primary: $Write_0$<br>$Secondary_1$: $Write_0$<br>$Secondary_2$: $Write_0$ | **Primary: $Write_{prev}$**<br>$Secondary_1$: $Write_{prev}$<br>$Secondary_2$: $Write_{prev}$ |
| $t_3$ | Primary is aware of successful replication to $Secondary_1$ and sends acknowledgement to client | Primary: $Write_0$<br>$Secondary_1$: $Write_0$<br>$Secondary_2$: $Write_0$ | **Primary: $Write_0$**<br>$Secondary_1$: $Write_{prev}$<br>$Secondary_2$: $Write_{prev}$ |

# Consequence

- Read concern "*local*" returns the latest value as soon as it is applied locally, it has the danger of **read uncommitted**, e.g. return a value before the write is durable, the value may not exist if rolled back
- Read concern "*majority*" will return old value some time after the write happens even if the target is set to primary node; it does not return uncommitted value regardless of the targe node.
- Customized setting will have better scalability by allowing read to happen at the secondary node
  - There are various trade offs depending on the actual setting

# Outline

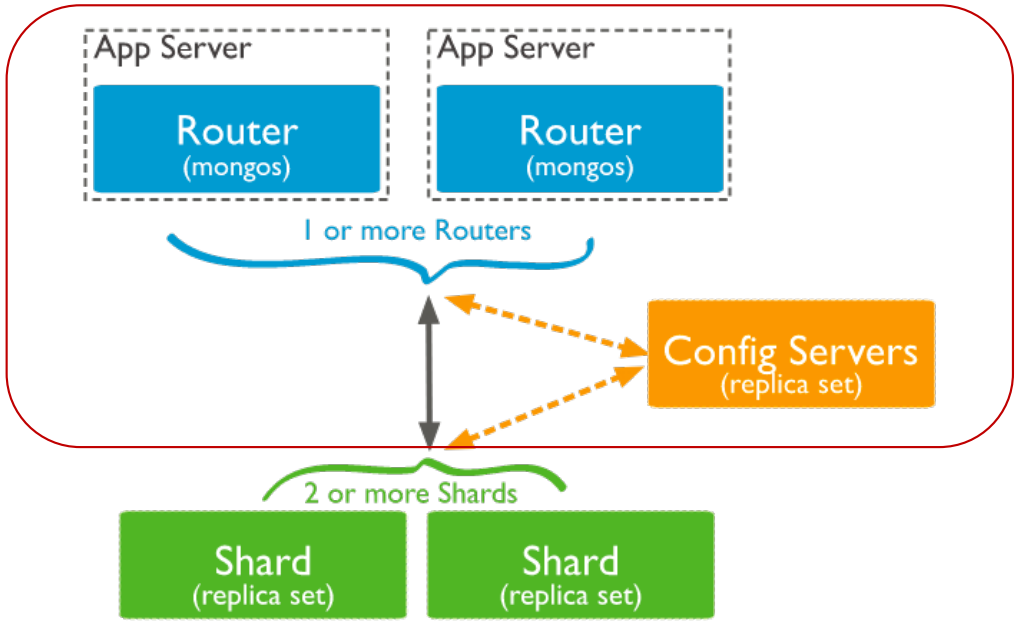
- Replication
- **Sharding**

# When Both Data Size and Traffic Grow

- Replication solves certain scalability issue by distributing read traffic to secondary members
- When data size grows beyond the capacity of a single node
  - Sharding/partition is needed
- Sharding is a method for distributing data across multiple machines.
- MongoDB uses horizontal sharding
  - E.g. instead of putting all documents of a collection in a single node, we can put subsets of documents in different nodes.
  - Users indicate how to divide the full collection into subsets of documents
  - System manages the actual data partition and query

# MongoDB Sharding

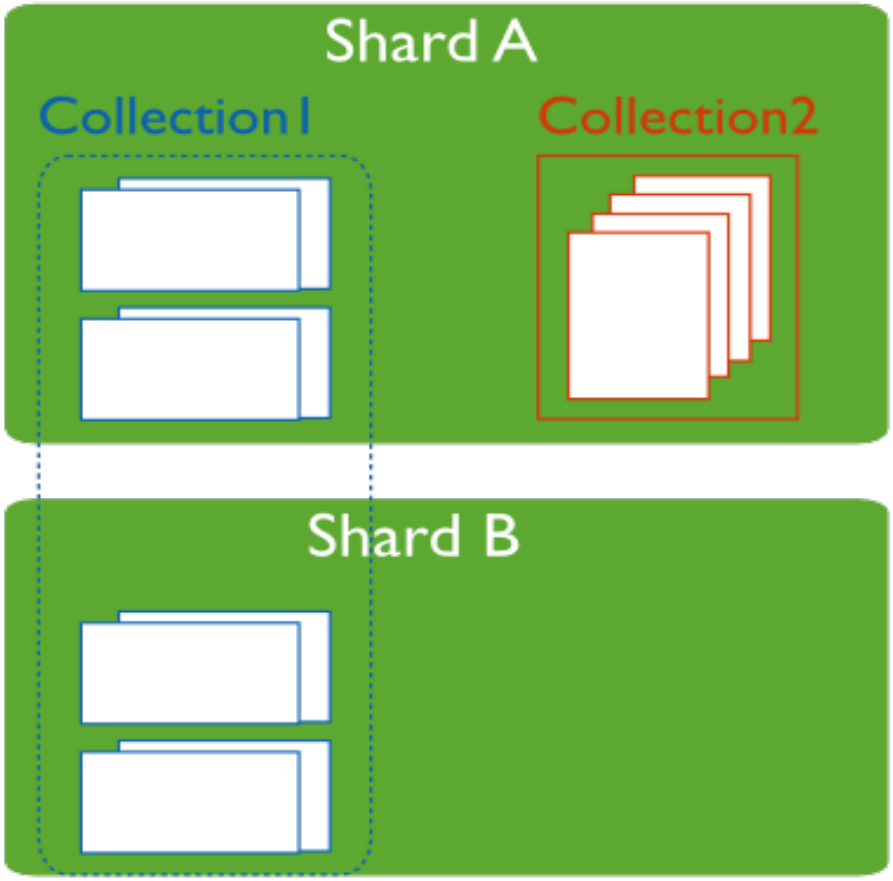
- The main database engine **`mongod`** is not distributed
  - Running on a single node
- Sharding is achieved by running an extra coordinator service **`mongos`** together with a set of **config servers** on top of **`mongod`**




https://docs.mongodb.com/manual/sharding/#sharded-cluster

# Shard

- Each shard is a standalone mongod server or a replica set ( with one primary and a few secondary members)
  - Shard must be a replica set since version 3.6
- Each shard stores some portion of a sharded cluster's total data set.
- Primary Shard
  - Every database has a primary shard that holds all unsharded collections for a database



# Data Partitioning with Chunks

- Data stored in each shard are organized as fixed sized **chunks** (default 64MB, but configurable)
- A chunk contains partition of documents belonging to the same collection
- Document-chunk distribution is determined by sharding key and sharding strategy
  - Sharding key is specified by use as single or compound field
- A shard may contain many chunks of a collection
- A cluster balancer is responsible to ensure chunks of a sharded collection are evenly distributed among the shards

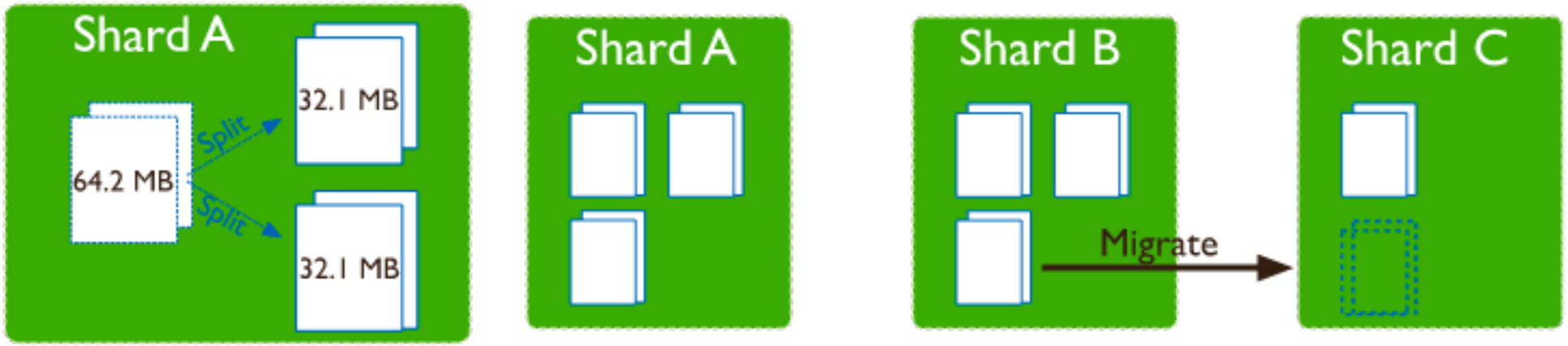
https://docs.mongodb.com/manual/core/sharding-data-partitioning/

# Initial Chunk creation

- Initial chunks may be created for populated collections or for empty ones
- Populated collection
  - The sharding operation creates the initial chunk(s) to cover the entire range of the shard key values. The number of chunks created depends on the chunk size and collection size
- Empty collection
  - One or many empty chunks may be created and placed on shards depending on the various settings
- Write operations would cause chunk size to vary along the time

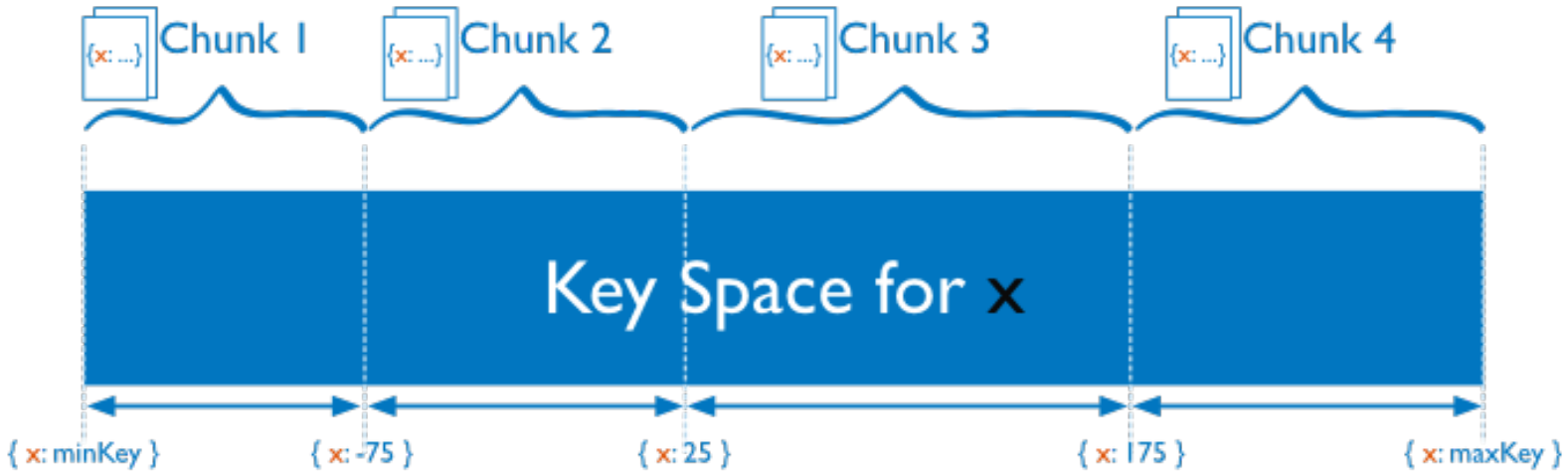
# Chunk Split and Migration

- When a chunk grows beyond the specified size, it will be split
- When chunks of the same collection are not distributed evenly among shards, some chunk will migrate between shards
- In some cases, chunks can grow beyond the specified chunk size but cannot undergo a split. The most common scenario is when a chunk represents a single shard key value. Since the chunk cannot split, it continues to grow beyond the chunk size, becoming a **jumbo** chunk.



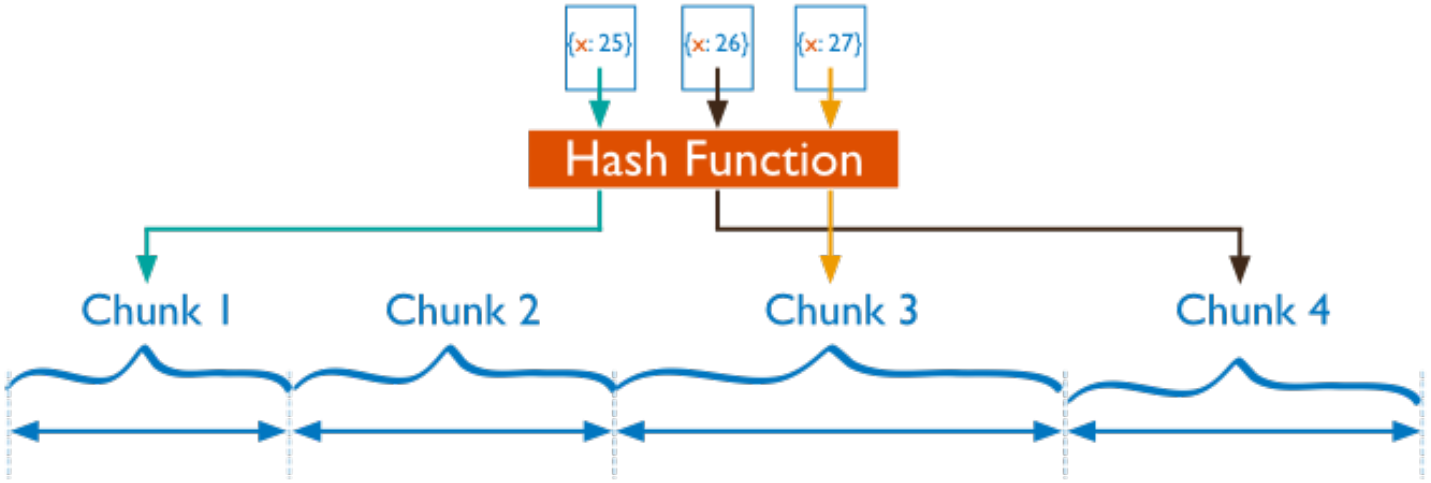
# Sharding Strategy: Range Sharding

- Chunk is created based on actual sharding key's value range
- Each chunk represents a range of the sharding key value
- Contiguous sharding key values are likely to be stored on the same shard



# Sharding Strategy: Hash Sharding

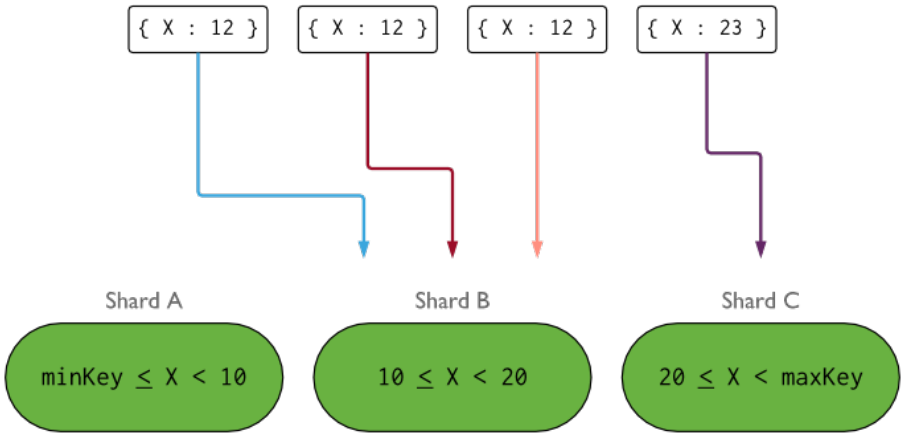
- Chunk is created based on the hash value of sharding key
- Each chunk represent a range of the hash value
- Contiguous sharding key values are likely to be distributed in different chunks



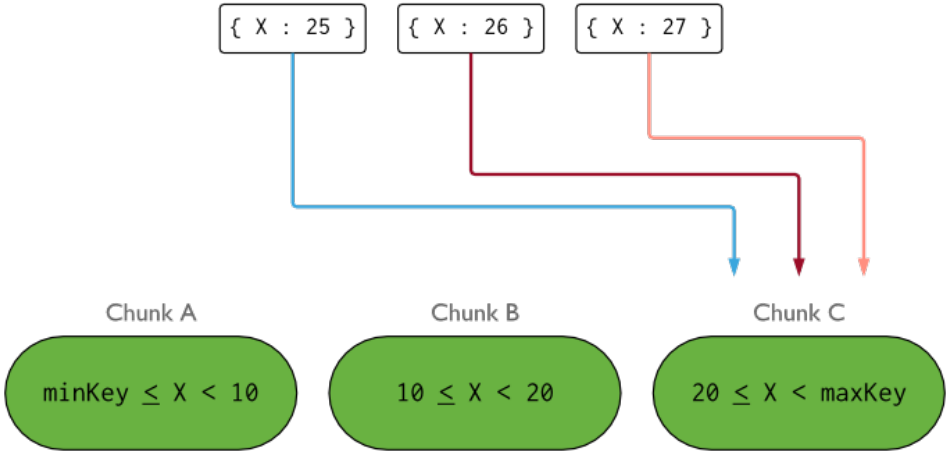
# Shard Key Selection

- The ideal shard key should distribute data and query evenly in shards
  - High cardinality
    - Gender is not a good sharding key candidate
  - Distribution not skewed
    - Key with zipf value distribution is not a good sharding key candidate
  - Change pattern
    - Timestamp is perhaps not a very good shard key candidate

Shard key with skewed distribution would create query hot spot




Monotonically increasing shard key would create insert hot spot



# Example of Good sharding Key

| Machine 1 | Machine 2 | Machine 3 |
|---|---|---|
| Alabama → Arizona | Colorado → Florida | Arkansas → California |
| Indiana → Kansas | Idaho → Illinois | Georgia → Hawaii |
| Maryland → Michigan | Kentucky → Maine | Minnesota → Missouri |
| Montana → Montana | Nebraska → New Jersey | Ohio → Pennsylvania |
| New Mexico → North Dakota | Rhode Island → South Dakota | Tennessee → Utah |
| | Vermont → West Virgina | Wisconsin → Wyoming |

a chunk

**`user`** collection partitioned by field "**`state`**" as shard key

# Indexing on Sharded Collection

- There is no global index structure on sharded collections
- When an index is created on a sharded collection
  - Local index structure is created by each shard for the data portion it is responsible
  - Shards do not communicate with each other
- Certain index properties cannot be maintained
  - Uniqueness can only be guaranteed for sharding key
  - No other index should be created with the uniqueness property
    - A **`users`** collection sharded by `userid` can guarantee each document has a unique `userid` but cannot guarantee no duplication of TFN
- MongoDB also require index support for sharding key

# Config Server

- Config servers are deployed as a replica set
  - Admin and shard metadata are stored as MongoDB collections
- Config servers maintain an `admin` database and a `config` database
  - The admin database stores data related to the authentication and authorization  and other system internal information
  - Config database stores data about chunks and their locations in shard

**`user`** collection partitioned by field "**`name`**" as shard key and are stored as chunks in different shards

| collection | minkey | maxkey | location |
|---|---|---|---|
| users | { name : 'Miller' } | { name : 'Nessman' } | $shard_2$ |
| users | { name : 'Nessman' } | { name : 'Ogden' } | $shard_4$ |
| ... | | | |

# Config Servers Read/Write

- Users are not supposed to read/write collections maintained by config servers
  - Various MongoDB components will read/write data from them
- MongoDB writes data to the config database when the metadata changes, such as after a chunk migration or a chunk split.
  - When writing to the config servers, MongoDB uses a write concern of "majority".
- Admin and config database are read by various MongoDB components for authentication data and collection distribution data
  - When reading from the config servers, MongoDB uses a read concern of “majority”

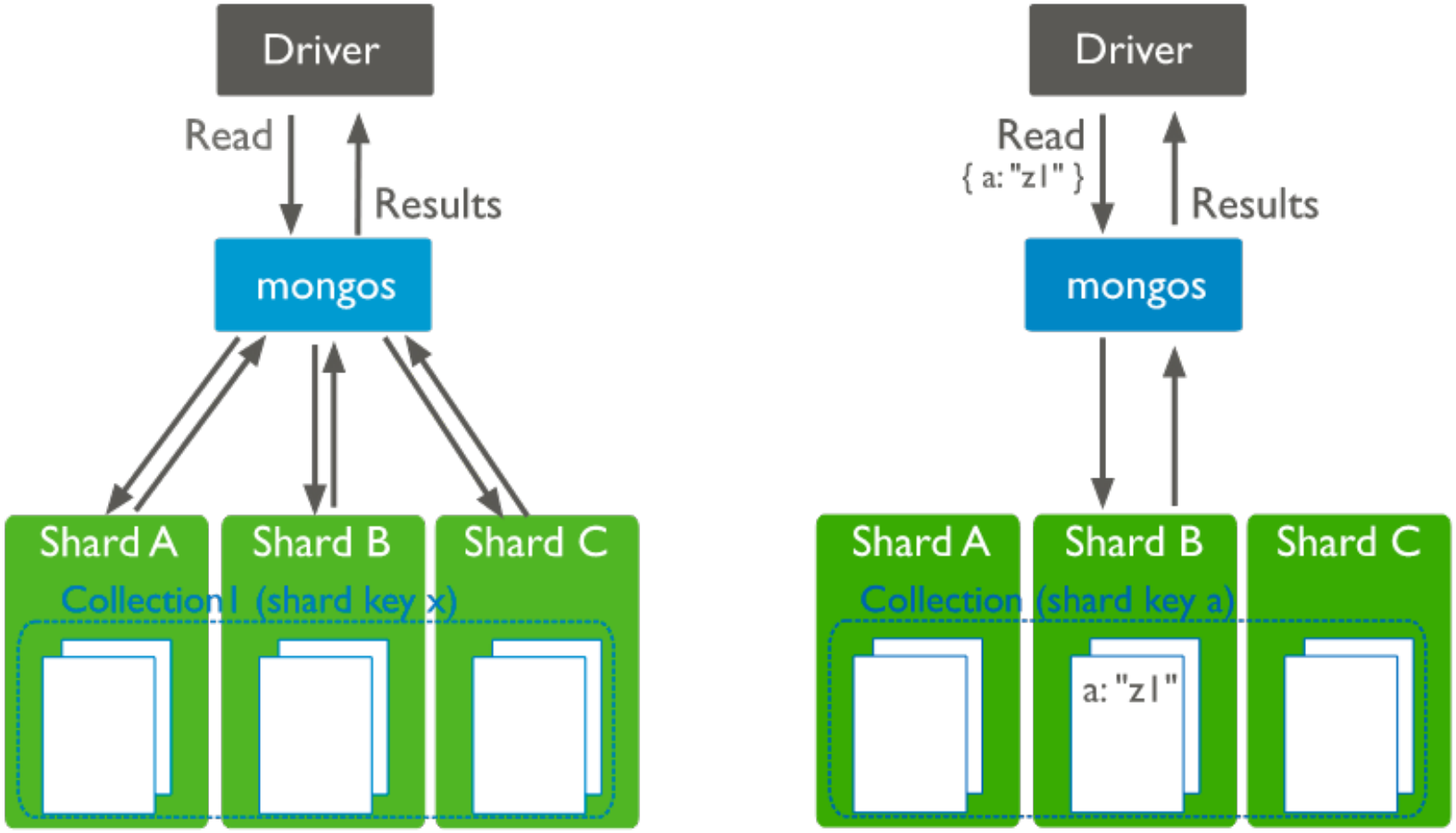
# Config Server Availability

- If the config server loses its primary, the cluster's metadata becomes *read only*. You can still read and write data from the shards, but no chunk migration or chunk splits will occur until a new primary is elected.
- If all config servers become unavailable, the cluster can become inoperable.

# Routing Processes -- `mongos`

- In a sharded cluster, **mongos** is the front end for client request
  - When receiving client requests, the **mongos** process ***routes*** the request to the appropriate server(s) and ***merges*** any results to be sent back to the client
  - It has no persistent state, the meta data are pulled from **config servers**
  - There is no limits on the number of **`mongos`** processes. They are independent to each other
- Query types
  - Targeted at a single shard or a limited group of shards based on the **shard key**.
  - Broadcast to all shards in the cluster that hold documents in a collection.

# Targeted and Broadcast Operations



# Targeted and Broadcast Operation Examples

- Assuming shard key is field x

| Operation | Type | Execution |
|---|---|---|
| db.food.find({x:300}) | Targeted | Query a single shard |
| db.foo.find( { x : 300, age : 40 } ) | Targeted | Query a single shard |
| db.foo.find( { age : 40 } ) | Global | Query all shards |
| db.foo.find() | Global | Query all shards, sequential |
| db.foo.find(...).count() | Variable | Same as the corresponding find() operation |
| db.foo.count() | Global | Parallel counting on each shard, merge results on mongos |
| db.foo.insert( <object> ) | Targeted | Insert on a single shard |
| db.foo.createIndex(...) | Global | Parallel indexing on each shard |

# Summary

- **MongoDB is a general purpose NoSQL storage system**
  - Lots of resemblance with RDBMS
    - Indexing, queries on any field
    - It supports spatial and graph queries
  - Single document update is always atomic
  - Later version has support for multi-document transaction
- **Key Features**
  - Flexible schema
    - Collection and Document
    - Documents are stored in binary JSON format
    - Natural support  for object style query (array and dot notation)
  - Scalability
    - Sharding and Replication
  - Various consistency levels achieved through write concern, read preference and read concern property combination

# Refrences

- MongDB Replication
  - https://docs.mongodb.com/manual/replication/
- MongoDB Replica Set Read and Write Semantics
  - https://docs.mongodb.com/manual/applications/replication/
- MongoDB Write Concern
  - https://docs.mongodb.com/manual/reference/write-concern
- MongoDB Read Isolation (Read Concern)
  - https://docs.mongodb.com/manual/reference/read-concern/