CS572 Modern Web Applications Programming
Maharishi University of Management
Department of Computer Science
Assistant Professor Asaad Saad

PREV NEXT

OUTLINE

Search...

Aggregation Framework

Aggregations operations process data records and return computed results. Aggregation operations **group values** (Similar to Group By) from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

MongoDB provides three ways to perform aggregation:

- The aggregation pipeline
- The map-reduce function
- Single purpose aggregation methods
  (`db.collection.count(), db.collection.group(),db.collection.distinct()`)

Aggregation Pipeline

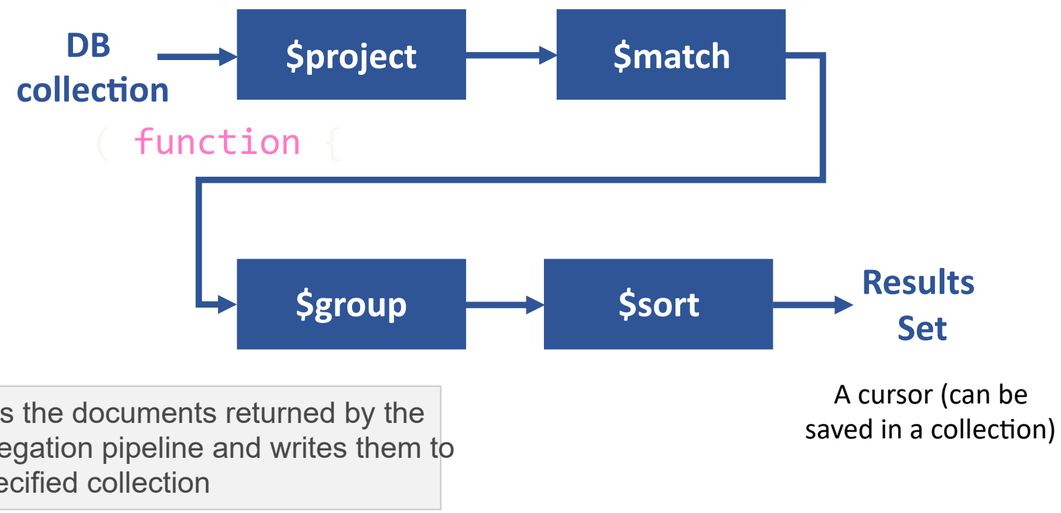
The aggregation framework is built on the concept of **data processing pipelines**. Documents enter a multi-stage pipeline that transforms the documents into an **aggregated result**.

- The pipeline provides efficient data aggregation using native operations within MongoDB.
- The aggregation pipeline can operate on a sharded collection.
- The aggregation pipeline can use indexes to improve its performance during some of its stages (only if it's done at the beginning of the aggregation pipeline).
- Every step can appear multiple times in the pipeline.

Aggregation Pipeline Stages

```
$group
$project
$match
$sort
$limit
$skip
$unwind
$out
$lookup
$redact
$geoNear
```

There is a 100 MB limit for any pipeline stage.

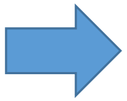


Takes the documents returned by the aggregation pipeline and writes them to a specified collection

A cursor (can be saved in a collection)

SQL vs Aggregate Example

id	name	category	manufacturer	price
1	iPad	Tablet	Apple	800
2	Nexus	Phone	Google	500
3	iPhone	Phone	Apple	600
4	iPadPro	Tablet	Apple	900



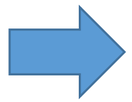
manufacturer	count
Apple	3
Google	1

```
select manufacturer, count(*) from products group by manufacturer
```

```
{ _id: 1, name: 'iPad', category: 'Tablet', manufacturer: 'Apple', price: 800 }
{ _id: 2, name: 'Nexus', category: 'Phone', manufacturer: 'Google', price: 500 }
{ _id: 3, name: 'iPhone', category: 'Phone', manufacturer: 'Apple', price: 600 }
{ _id: 4, name: 'iPadPro', category: 'Tablet', manufacturer: 'Apple', price: 900 }
```

```
db.products.aggregate([
        {$group: {
                _id:"$manufacturer",
                num_products:{$sum:1} }
        }
])
```

Every field must be an accumulator object

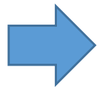


```
{ _id: 'Apple',  num_products: 3 }
{ _id: 'Google', num_products: 1 }
```

$group

Think of **`$group`** as an **UPSERT** stage, where it'll insert if **`_id`** if not found, or update when available.

```
db.products.aggregate([
        {$group: { _id:"$manufacturer",
                   num_products:{$sum:1} } }
])
```



```
{ _id: 'Apple', num_products: 3 }
{ _id: 'Google', num_products: 1 }
```

When we want to use a key as **DATA** in the right side to read its value we must use **$ sign** with the name. We don't need to do that when using it at the left side as it is just a label.

```
db.products.aggregate([
        {$group: { _id: {'manufacturer':"$manufacturer"},
                   num_products:{$sum:1} } }
])
```

```
{ "_id" : { "manufacturer" : "Google" }, "num_products" : 1 }
{ "_id" : { "manufacturer" : "Apple" }, "num_products" : 3 }
```

Compound Grouping

```
db.products.aggregate([
        {$group: { _id: { "manufacturer":"$manufacturer", "category" : "$category"},
                   num_products:{$sum:1} } }
])
```

```
{ "_id" : { "manufacturer" : "Google", "category" : "Phone" }, "num_products" : 1 }
{ "_id" : { "manufacturer" : "Apple", "category" : "Tablet" }, "num_products" : 2 }
{ "_id" : { "manufacturer" : "Apple", "category" : "Phone" }, "num_products" : 1 }
```

Primary Key

Aggregation Expressions with $group

$sum, $avg, $min, $max, $push, $addToSet, $first, $last

```
{ _id: 1, name: 'iPad', category: 'Tablet', manufacturer: 'Apple', price: 800 }
{ _id: 2, name: 'Nexus', category: 'Phone', manufacturer: 'Google', price: 500 }
{ _id: 3, name: 'iPhone', category: 'Phone', manufacturer: 'Apple', price: 600 }
{ _id: 4, name: 'iPadPro', category: 'Tablet', manufacturer: 'Apple', price: 900 }
```

```
db.products.aggregate([
        {$group: { _id: { "maker":"$manufacturer" },
                   sum_prices:{$sum:"$price"} } }
])
```

```
{ "_id" : { "maker" : "Google" }, "sum_prices" : 500 }
{ "_id" : { "maker" : "Apple" }, "sum_prices" : 2300 }
```

```
db.products.aggregate([
        {$group: { _id: { "category":"$category" },
                   avg_price:{$avg:"$price"} } }
])
```

```
{ "_id" : { "category" : "Phone" }, "avg_price" : 550 }
{ "_id" : { "category" : "Tablet" }, "avg_price" : 850 }
```

```
db.products.aggregate([
        {$group: { _id: { "maker":"$manufacturer" },
                   maxprice:{$max:"$price"} } }
])
```

```
{ "_id" : { "maker" : "Google" }, "maxprice" : 500 }
{ "_id" : { "maker" : "Apple" }, "maxprice" : 900 }
```

Aggregation Expressions with $group

```
{ _id: 1, name: 'iPad', category: 'Tablet', manufacturer: 'Apple', price: 800 }
{ _id: 2, name: 'Nexus', category: 'Phone', manufacturer: 'Google', price: 500 }
{ _id: 3, name: 'iPhone', category: 'Phone', manufacturer: 'Apple', price: 600 }
{ _id: 4, name: 'iPadPro', category: 'Tablet', manufacturer: 'Apple', price: 900 }
```

```
db.products.aggregate([
        {$group: { _id: { "maker":"$manufacturer" },
                   categories:{$addToSet:"$category"} } }
])
```

```
{ "_id" : { "maker" : "Google" }, "categories" : [ "Phone" ] }
{ "_id" : { "maker" : "Apple" }, "categories" : [ "Phone", "Tablet" ] }
```

```
db.products.aggregate([
        {$group: { _id: { "maker":"$manufacturer" },
                   categories:{$push:"$category"} } }
])
```

```
{ "_id" : { "maker" : "Google" }, "categories" : [ "Phone" ] }
{ "_id" : { "maker" : "Apple" }, "categories" : [ "Tablet", "Phone", "Tablet" ] }
```

$project

Use it to remove a key, add new key, rephrase a key or with some simple functions: **$toUpper** and **$toLower** for strings, **$add** and **$multiply** for numbers.

```
{ _id: 1, name: 'iPad', category: 'Tablet', manufacturer: 'Apple', price: 800 }
{ _id: 2, name: 'Nexus', category: 'Phone', manufacturer: 'Google', price: 500 }
{ _id: 3, name: 'iPhone', category: 'Phone', manufacturer: 'Apple', price: 600 }
{ _id: 4, name: 'iPadPro', category: 'Tablet', manufacturer: 'Apple', price: 900 }
```

```
db.products.aggregate([
        {$project: {  _id:0,
                        'maker': {$toLower:'$manufacturer'},
                        'details': {'category':'$category',
                                    'price':{$multiply:['$price',10]} },
                        'item':'$name' } } ])
```

```
{ maker: 'apple', details: { category: 'Tablet', price: 8000 }, item: 'iPad'}
{ maker: 'google', details: { category: 'Phone', price: 5000 }, item: 'Nexus'}
{ maker: 'apple', details: { category: 'Phone', price: 6000 }, item: 'iPhone'}
{ maker: 'apple', details: { category: 'Tablet', price: 9000 }, item: 'iPadPro'}
```

$match

Use it to **filter** the collection.

```
{ "_id" : "52556", "city" : "FAIRFIELD", "loc" : [ -91.957611, 41.003943 ], "pop" : 12147, "state" : "IA" }
{ "_id" : "52601", "city" : "BURLINGTON", "loc" : [ -91.116972, 40.808665 ], "pop" : 30564, "state" : "IA" }
{ "_id" : "52641", "city" : "MOUNT PLEASANT", "loc" : [ -91.56142699999999, 40.964573 ], "pop" : 11113, "state" : "IA" }
{ "_id" : "52241", "city" : "CORALVILLE", "loc" : [ -91.590608, 41.693666 ], "pop" : 12646, "state" : "IA" }
{ "_id" : "52240", "city" : "IOWA CITY", "loc" : [ -91.51119199999999, 41.654899 ], "pop" : 25049, "state" : "IA" }
{ "_id" : "52245", "city" : "IOWA CITY", "loc" : [ -91.51506999999999, 41.664916 ], "pop" : 21140, "state" : "IA" }
{ "_id" : "52246", "city" : "IOWA CITY", "loc" : [ -91.56688200000001, 41.643813 ], "pop" : 22869, "state" : "IA" }
```

```
db.zips.aggregate([
        {$match: { state:"IA" } },
        {$group: { _id: "$city",
                   population: {$sum:"$pop"},
                   zip_codes: {$addToSet: "$_id"} } },
        {$project: { _id: 0,
                    city: "$_id",
                    population: 1,
                    zip_codes:1 } }
])
```

1. Filter the zipcode collection and leave Iowa state entries
2. Group results by city, calculate the population, and add new field contains zipcode array for each city
3. Remove _id, project only city name, population and zipcodes.

```
{ "city" : "FAIRFIELD", "population" : 12147, "zip_codes" : ["52556"] }
{ "city" : "BURLINGTON", " population" : 30564, "zip_codes" : ["52601" ]}
{ "city" : "MOUNT PLEASANT", " population" : 11113, "zip_codes" : ["52641"] }
{ "city" : "CORALVILLE", " population" : 12646, "zip_codes" : ["52241"] }
{ "city" : "IOWA CITY", " population" : 69058, "zip_codes" : ["52240", "52245", "52246"] }
```

$sort, $skip and $limit

It's useful to use **$skip** and **$limit** after **$sort**, otherwise the result will be arbitrary.

```
db.zips.aggregate([
        {$match: { state:"IA" } },
        {$group: { _id: "$city",
                   population: {$sum:"$pop"} } },
        {$project: { _id: 0,
                     city: "$_id",
                     population: 1, } },
        {$sort: { population:-1 } },
        {$skip: 10},
        {$limit: 5}
])
```

Sorting can be done on Disk, or in Memory (default).

When sort is an early stage it may use indexes.

It can be used before/after grouping.

Note: The order of `.sort()`, `.skip()` and `.limit()` methods when applied to a collection cursor does not matter. While the order of `$sort`, `$skip` and `$limit` aggregation stages matters.

Exercise

Use the aggregation framework to get the largest city in every state using the Zipcode collection:


```
{ "_id" : "52556", "city" : "FAIRFIELD", "loc" : [ -91.957611, 41.003943 ], "pop" : 12147, "state" : "IA" }
{ "_id" : "52601", "city" : "BURLINGTON", "loc" : [ -91.116972, 40.808665 ], "pop" : 30564, "state" : "IA" }
{ "_id" : "52641", "city" : "MOUNT PLEASANT", "loc" : [ -91.56142699999999, 40.964573 ], "pop" : 11113, "state" : "IA" }
{ "_id" : "52241", "city" : "CORALVILLE", "loc" : [ -91.590608, 41.693666 ], "pop" : 12646, "state" : "IA" }
{ "_id" : "52240", "city" : "IOWA CITY", "loc" : [ -91.51119199999999, 41.654899 ], "pop" : 25049, "state" : "IA" }
{ "_id" : "52245", "city" : "IOWA CITY", "loc" : [ -91.51506999999999, 41.664916 ], "pop" : 21140, "state" : "IA" }
{ "_id" : "52246", "city" : "IOWA CITY", "loc" : [ -91.56688200000001, 41.643813 ], "pop" : 22869, "state" : "IA" }
```


```
db.zips.aggregate([
    {$group: { _id: {state:"$state", city:"$city"},
               population: {$sum:"$pop"}, } },
    {$sort: {"_id.state":1, "population":-1} },
    {$group: { _id:"$_id.state",
               city: {$first: "$_id.city"},
               population: {$first:"$population"} } },
    {$sort: {"_id":1} }
])
```

Get the largest city in every state:

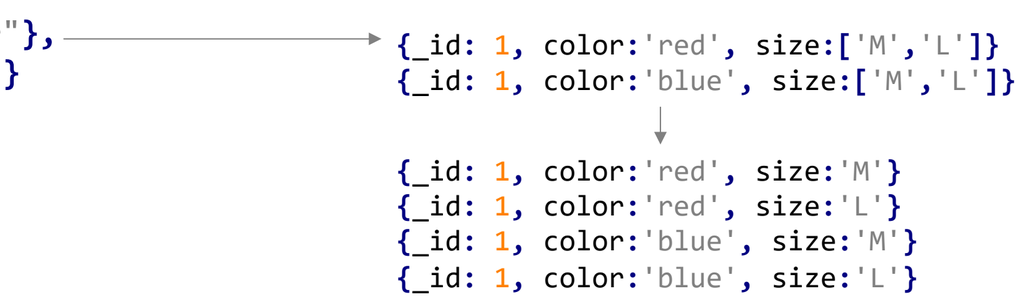
1. Get the population of every city in every state
2. Sort by state, population
3. Group by state, get the first item in each group
4. Now sort by state again (Group might change the order)

$unwind

Deconstructs an array field from the input documents to output a document for each element. Each output document is the input document with the value of the array field replaced by the element.

```
db.items.drop();
db.items.insert({ _id: 1,
                  'color':['red', 'blue'],
                  'size':['M', 'L'] });

db.items.aggregate([
     {$unwind:"$color"},
     {$unwind:"$size"}
]);
```



$push vs. $unwind

$push enables you to reverse the effects of an **$unwind**

```
db.items.aggregate([
     {$unwind:"$color"},
     {$group:{ '_id': "$_id"},
              'colors': {$push: "$color"}, }}
]);
```

```
{ _id: 1, 'color':['red', 'blue']}
```

```
{_id: 1, color:'red'}
{_id: 1, color:'blue'}
```

```
{ _id: 1, 'color':['red', 'blue']}
```

$lookup

Performs a **left outer join** to an unsharded collection.

```
//orders
{ "_id" : 1, "pk" : "abc", "price" : 12, "quantity" : 2 }
{ "_id" : 2, "pk" : "jkl", "price" : 20, "quantity" : 1 }
//inventory
{ "_id" : 1, "fk" : "abc", description: "product 1", "instock" : 120 }
{ "_id" : 2, "fk" : "def", description: "product 2", "instock" : 80 }
{ "_id" : 3, "fk" : "abc", description: "product 3", "instock" : 60 }
{ "_id" : 4, "fk" : "jkl", description: "product 4", "instock" : 70 }
```

```
db.orders.aggregate([
        { $lookup: {
                from: "inventory",
                localField: "pk",
                foreignField: "fk",
                as: "inventory_docs" }
        }
])
```

```
{
   "_id" : 1,
   "pk" : "abc",
   "price" : 12,
   "quantity" : 2,
   "inventory_docs" : [ { "_id" : 1, "fk" : "abc",
"description": "product 1", "instock" : 120 } ,
{ "_id" : 3, "fk" : "abc", "description": "product 3",
"instock" : 60 }
]
}
...
...
```


PREV NEXT

SQL to Aggregation Mapping Chart

WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum
join	$lookup

https://docs.mongodb.com/manual/reference/sql-aggregation-comparison/

Count all records from orders using aggregation framework:

```
SELECT COUNT(*) AS count FROM orders
```

```
db.orders.aggregate([
        { $group: { _id: null,
                    count: {$sum: 1}} }
])
```

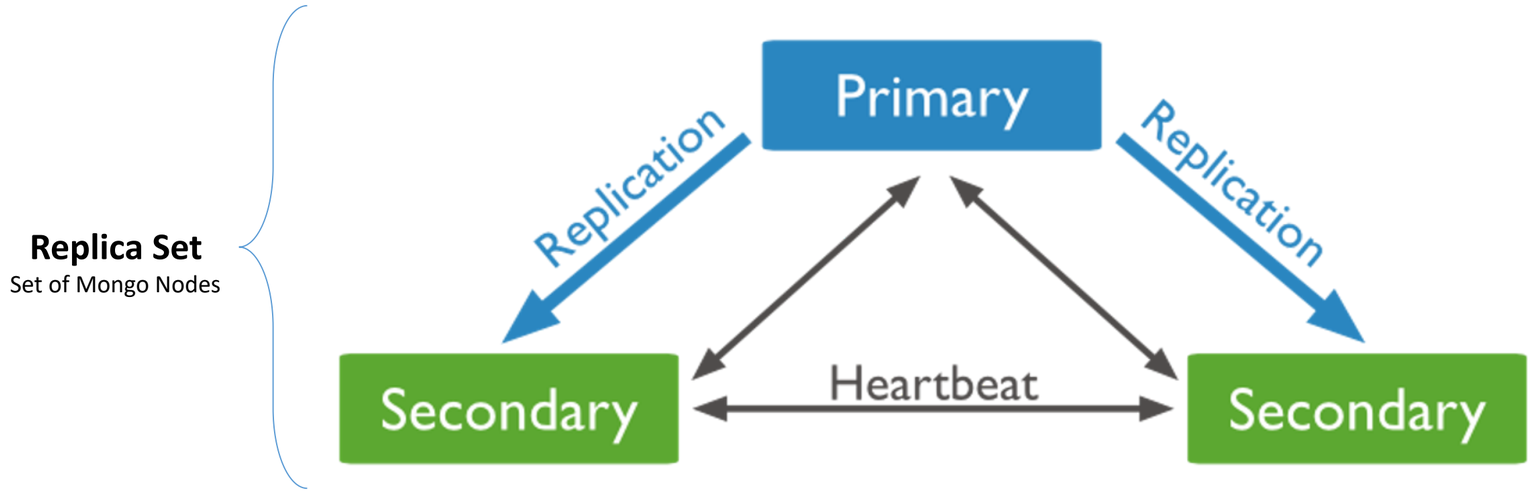
Aggregation Framework Implications

- There is 100 MB RAM limit for every stage in the pipeline, If you want to work with bigger data use Disk instead **`(allowDiskUse: true)`**
- If you want to save the results to a collection using **`$out`**, remember that there is 16 MB limit size for every document (it's not recommended, better to return a cursor and work with it)
- Aggregation will work fine in sharded environments and the results from **`$group`** and **`$sort`** will be sent back to the first sharded DB.

PREV NEXT

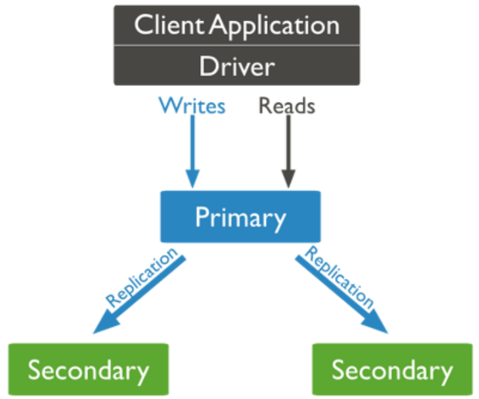
Replication

MongoDB has the capability to be **fail tolerance** with **high availability**. That's because of its advanced replication features which allow us to setup multiple nodes (one Primary and many Secondary) that will be automatically replicated asynchronously.



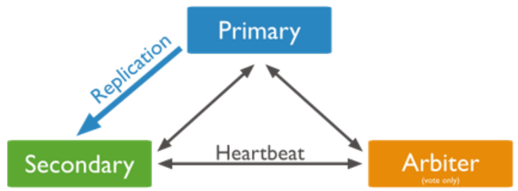
Replica Set

- Our application (Mongo Driver) always talk to the Primary Node. Perform writes operations to it or read operations from it.
- When the Primary node goes down, Secondary nodes will perform an election to elect a new Primary node based on the majority of original number of nodes (transparent to the application).
- We must at least have 3 nodes in a Replica Set.



Types of Replica Set nodes

- Regular Node (primary, secondary)
- Arbiter Node (for voting purposes, if we have two machines only)
- Delayed Node (disaster node, set a time behind other nodes)
- Hidden Node (for analytics)



All nodes can participate in the election and can vote but only regular nodes can become a Primary node.

25

PREV NEXT

OUTLINE

Search...

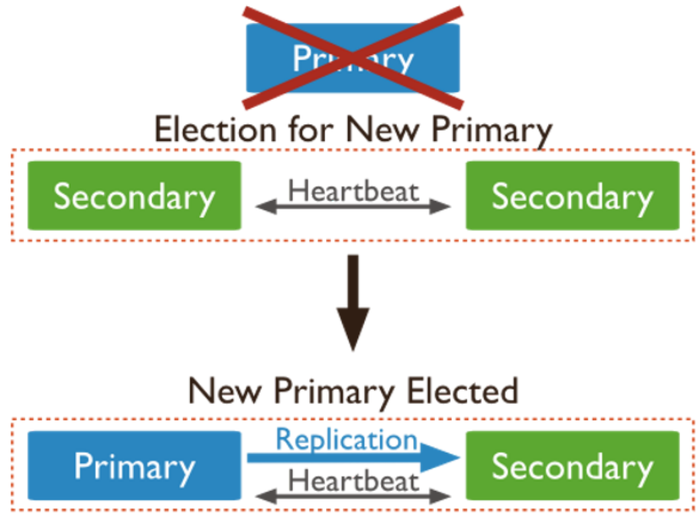
Replication Write Concerns

The MongoDB Driver will only **write** to the **Primary node**.

During the time of failover you cannot perform/complete a write because there is no Primary node and the new Primary node has not been elected yet. All write operations will be buffered in the Driver until one of the Secondary nodes is elected as Primary then the callback will be called and response will be received.

We can configure the write concerns to be:

- (w:0) which mean return success immediately
- (w:1) return success only until Primary successfully writes
- (w:2) return success only until Primary and one Secondary successfully writes

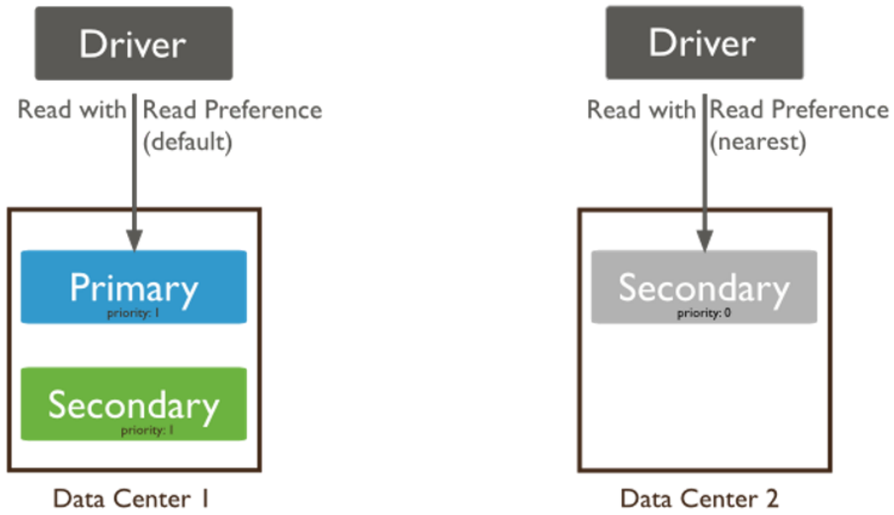


When the failed Primary node rejoin the Replica Set it will join as Secondary, It will sync itself with the existing new Primary. If there was any operations that has been applied to it before failure and wasn't synched to the other nodes they will be rolled back and put in file.

Replication Read Preferences

The MongoDB driver can be configured to **read** from Primary or Secondary or Nearest node.

Reading from the primary is more reliable and guarantee that applications will always read fresh data, while reading from Secondary can be stale as it might not be synched yet, but it could be faster when Primary is busy.

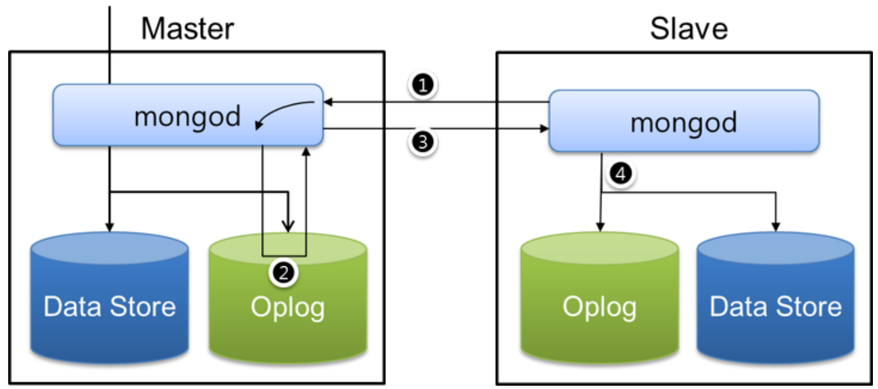


How Replication Really Works?

Oplog (operations log) is the heart of a MongoDB Replica Set. It is a special capped-collection that keeps a rolling record of all operations that modify the data stored in your databases.

MongoDB applies database operations on the primary and then records the operations on the primary's oplog.

All Secondary nodes will read and sync themselves with oplog collection.



citsoft

Capped collections *are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.*

PREV NEXT

Working with Replications in Node.js

```
const MongoClient = require('mongodb').MongoClient;
const client = new
MongoClient('mongodb://localhost:30001,localhost:30002,localhost:30003/testDB?w=1&
readPreference=secondary');

client.connect(function(err) {
   const db = client.db('myDB');
   const collection = db.collection('myCollection');
   ...
});
```

Set **Read Preferences**

Set **Write Concerns** to 1 return success only after Primary successfully writes

If you leave a Replica Set node out of the **Seed List** within the Driver, the missing node will be discovered as long as you list at least one valid node.

Implications of Replication

- Seed Lists

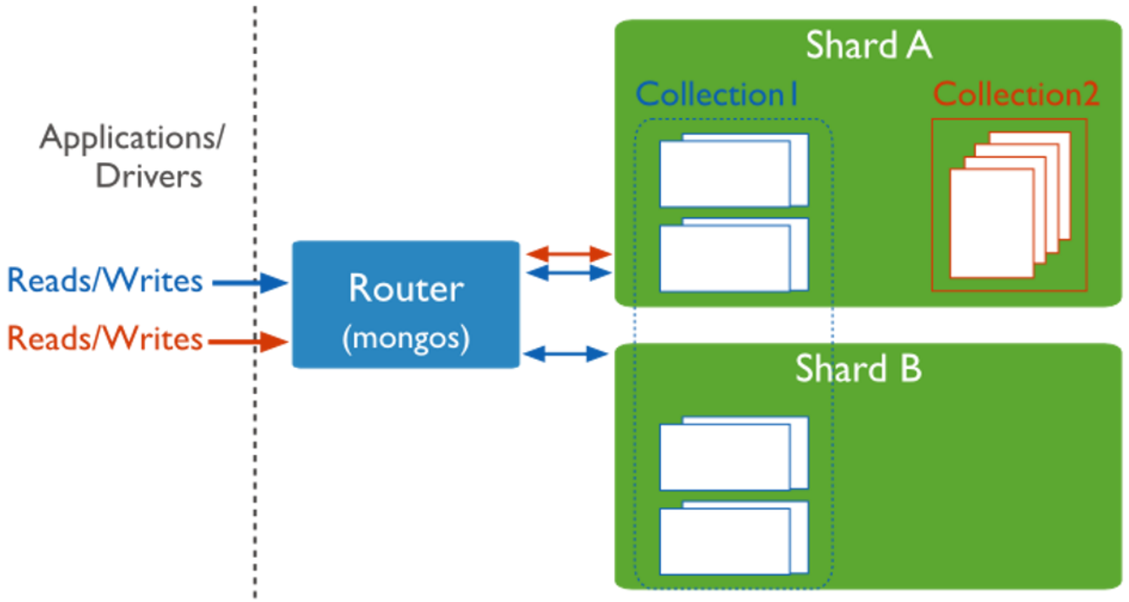
  The driver needs to know about the Replica Set nodes (at least one other node)

- Write concerns
- Read Preferences
- Errors can happen

  An operation might be written successfully in the Primary node and DB will send a success write feedback to the Driver but a TCP reset might occur and the client might not know about it (Always catch all error in your code)

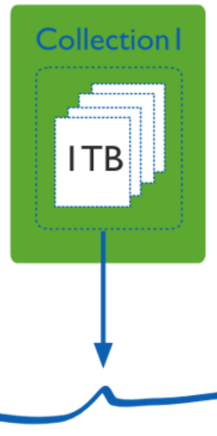
Sharding

Sharding is a method for storing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high output operations. *(provides high scalability)*

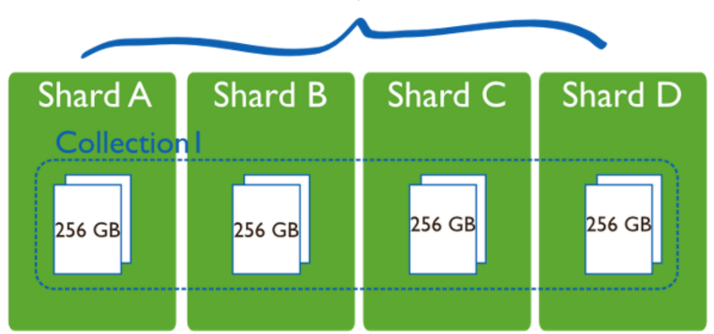


Vertical scaling vs Horizontal scaling

- **Vertical scaling** adds more CPU and storage resources to increase capacity. Scaling by adding capacity has limitations: high performance systems with large numbers of CPUs and large amount of RAM are *more expensive* than smaller systems.



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



PREV NEXT

Purpose of Sharding

- Sharding reduces the number of operations each shard handles. Each shard processes fewer operations as the cluster grows. As a result, a cluster can increase capacity and output *horizontally*.

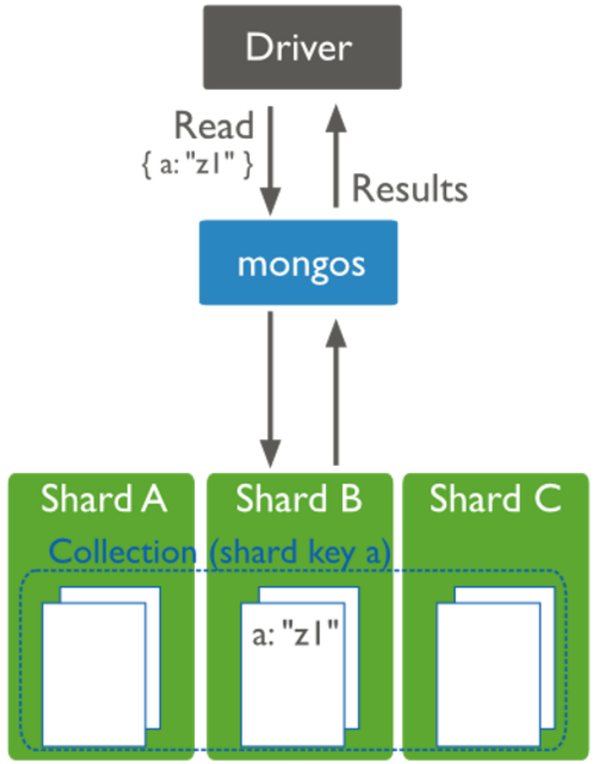
  For example, to insert/read data, the application only needs to access the shard responsible for that record.

- Sharding reduces the amount of data that each server needs to store. Each shard stores less data as the cluster grows.

  For example, if a database has a 1 terabyte data set, and there are 4 shards, then each shard might hold only 256GB of data. If there are 40 shards, then each shard might hold only 25GB of data.

Sharding in MongoDB

- **Query Routers**, or **`mongos`** instances, interface with client applications and direct operations to the appropriate shard or shards. *(we can have more than 1 mongos)*
- **Shards** store the data. To provide high availability and data consistency, **each shard can be a replica set**.



35

PREV NEXT

OUTLINE

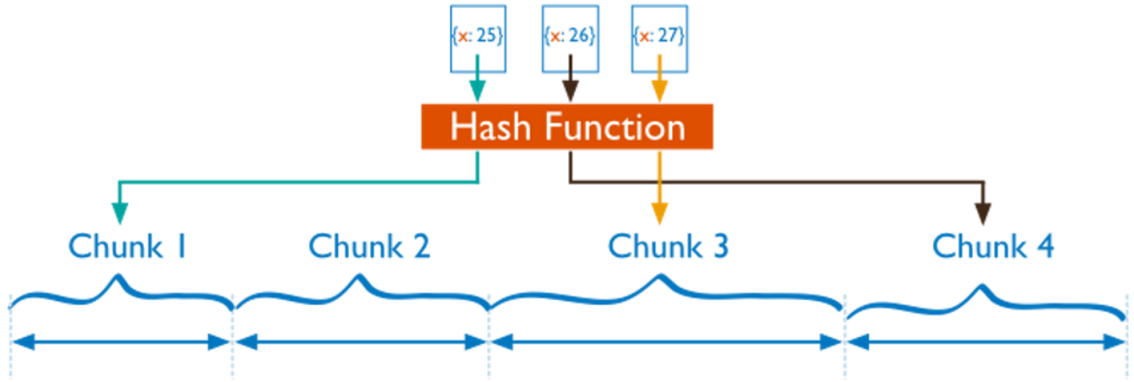
Search...

Data Partitioning

MongoDB distributes data, at the collection level. Sharding partitions a collection data by the **shard key**.

To shard a collection, you need to select a shard key. **A shard key is an Indexed Field that exists in every document** in the collection.

MongoDB divides the shard key values into **chunks** and distributes the chunks evenly across the shards.



MongoDB uses either **range based partitioning** or **hash based partitioning** to divide the shard key values into chunks.

PREV NEXT

Adding and Removing Shards from the Cluster

- Adding a shard to a cluster creates an imbalance since the new shard has no chunks. While MongoDB begins migrating data to the new shard immediately, it can take some time before the cluster balances.
- When removing a shard, the balancer migrates all chunks from a shard to other shards. After migrating all data and updating the meta data, the shard will be removed.