MongoDB

Document Database

Basics

- A document-oriented database
  - documents encapsulate and encode data (or information) in some standard formats or encodings
- NoSQL database
  - – highly optimized for retrieve and append operations
- Uses BSON format
- Schema-less
  - – Does not require configuring database columns with types
- No transactions
- No joins

Basics

- A MongoDB instance may have zero or more databases
- A database may have zero or more collections.
  - Can be thought of as the relation (table) in DBMS, but with many differences.
- A collection may have zero or more documents.
  - Documents in the same collection don’t need to have the same fields

Basics

- Documents are equivalent to records in RDBMS
- Documents can embed other documents
- Documents are addressed in the database via a unique key
- A document may have one or more fields.
- MongoDB Indexes is much like their RDBMS counterparts.

Basics

- Facilitate simple queries
- Facilitate easier and faster data integration
- Not suited for complex queries or heavy transaction applications

A Mongo Document

```
user = {
name: “DL",
occupation: "A Lecturer",
location: “Dublin"
}
```

A Mongo Document

```
{
name: ‘A Person',
address:
{
street: ‘A Dublin Street Name',
city: ‘Dublin'
}
}
```

A Mongo Collection

Create collection syntax:

```
db.createCollection(name, options)
```

Example:

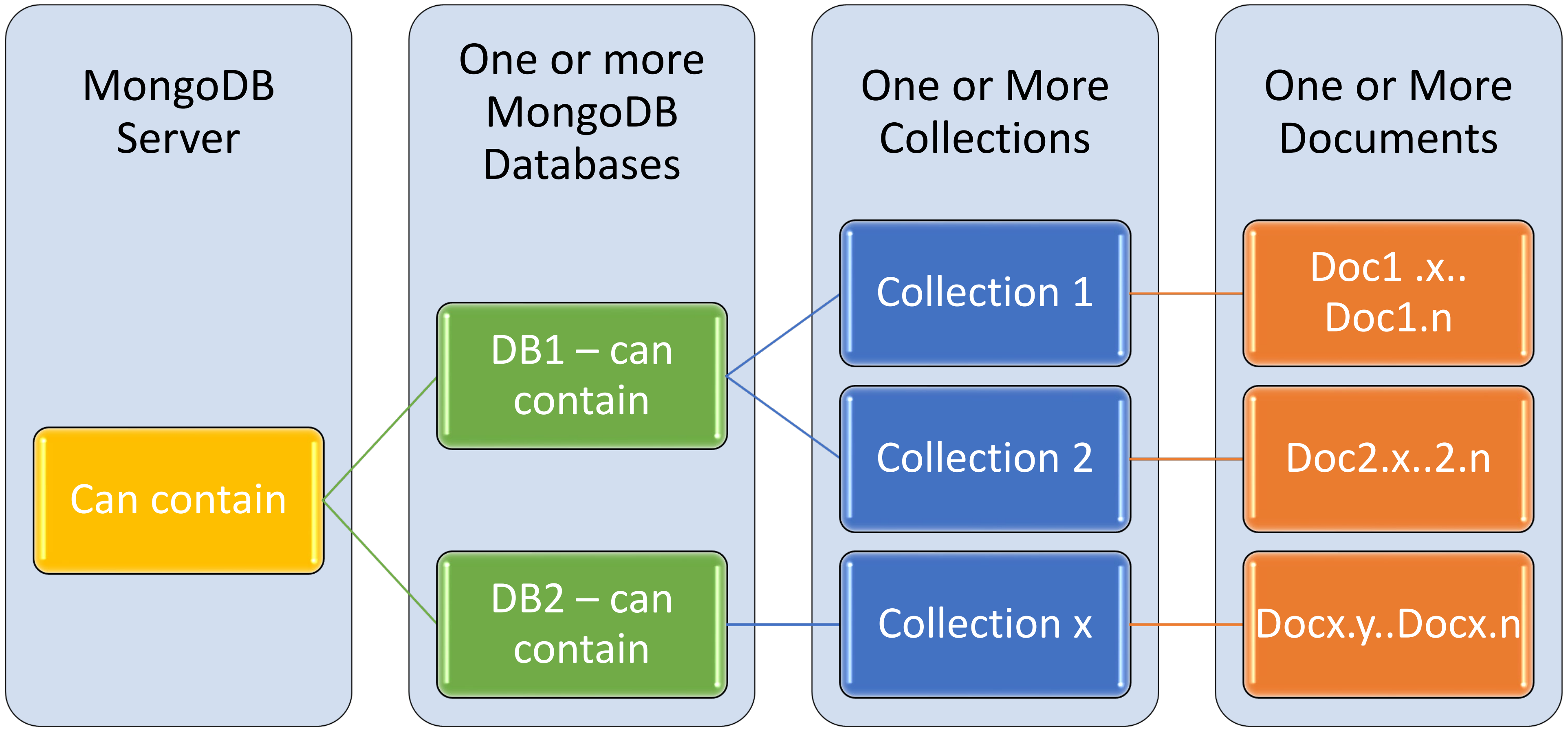
```
db.createCollection("mycollection")
{ "ok" : 1 }
```

A Mongo Collection

```
{"_id": ObjectId("4efa8d2b7d284dad101e4bc9"),
"Last Name": “Doe",
"First Name": "Jane",
"Date of Birth": "01-22-1963" },
{ "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
"Last Name": “Doe",
"First Name": “John",
"Date of Birth": "09-19-1983",
"Address": "1 Street 1",
"City": “Dublin" }
```

Object Ids are obligatory, and automatically generated by MongoDB

MongoDB Architecture



Creating a database

- To find out what databases you have

```
Show dbs
```

- To create a database simply

```
use dbame
```

e.g.

```
use test_mongo
```

- Once created you can switch database using the use command

Create a collection

- Collections are schemaless
- Collection name must start with an underscore or a character.
- Collection name cannot contain $, empty string, null character and does not begin with system. prefix.
- The maximum length of the collection name is 120 bytes(including the database name, dot separator, and the collection name).

Create a collection

- Create a collection by inserting documents into it
- Syntax

```
db.collection.insertOne(
   <document>,
   { writeConcern: <document>}
)
```

- Returns a document that contains the following fields:
  - acknowledged (boolean) - true if the insert executed with write concern; false if the write concern was disabled.
  - insertedId the value of _id field of the inserted document

Create a collection

- Create a collection by inserting documents into it
- Example

```
db.person.insertOne({ name:
'Jane Doe', address: '1 Street
1, Town 1’})
```

```
Response:
{
  acknowledged: true,
  insertedId:
ObjectId("637b87d0375e8adedd091132")
}
```

Create a collection

- Create a collection by inserting many documents into it
- Example

```
db.students.insertMany([
{
"student_name" : "Diarmuid",
"student_id" : 123456,
"student_age" : 22
},
{
"student_name" : "Fionn",
"student_id" : 012345,
"student_age" : 50
},
{
"student_name" : "Grainne",
"student_id" : 123458,
"student_age" : 23
}
])
```

Create a collection

- **Equivalent of select ***

```
db.students.find()
```

Pretty printed:

```
db.students.find().pretty()
```

- **Projection**

```
 db.students.find({}, {"_id": 0,
"student_name": 1})
```

Returns only student names

Creating documents -Example

- A Blog post
  - Has an author, some text, and many comments
- The comments are unique per post, but one author has many posts
- How would you design this in SQL?

Example – Bad Design

References a manually generated ID

```
post = {
        } id: 150,
        author: 100,
        text: 'This is a pretty awesome post.',
        comments: [100, 105, 112]


author = {
        id: 100,
        name: ‘A Blogger'
        posts: [150]
}
comment = {
        id: 105,
        text: ‘Rubbish.'
}
```

Example – Better Design

```
post = {
author: 'A Blogger',
text: 'This is a pretty awesome post.',
comments: [
‘Rubbish.',
'I agree!'
]
}
```

Embed comments, author name

Why is this better?

Embedded Objects

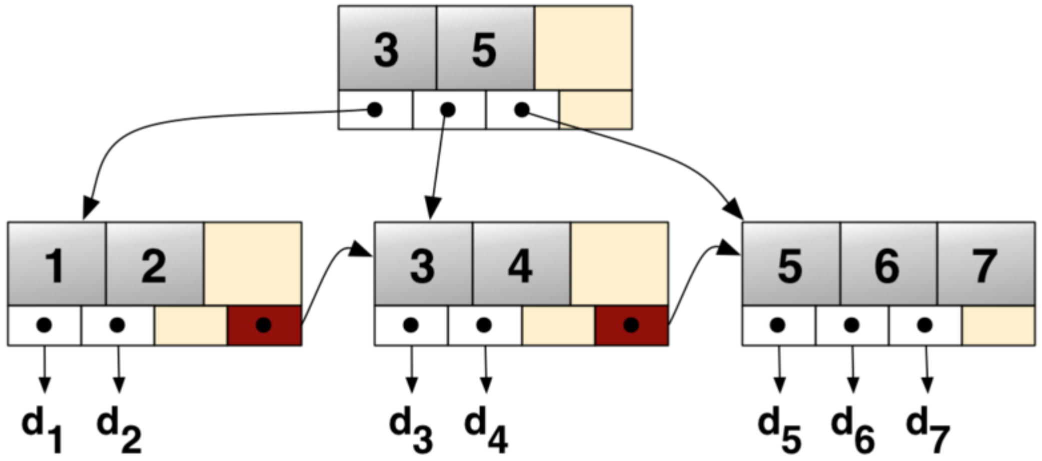
- Embedded objects are brought back in the same query as parent object
  - Requires only a single trip to the DB server required
- Objects in the same collection are generally stored contiguously on disk
  - Spatial locality implies a faster retrieval
- If the document model matches your domain well, it can be much easier to understand than using queries with joins

Queries

- Query expression objects indicate a pattern to match
  - db.users.find({last_name: ‘Doe'})
- Several query objects for advanced queries
  - db.users.find({age: {$gte: 23} })
  - db.users.find({age: {$in: [23,25]} })
- Exact match an entire embedded object
  - db.users.find({address: {street: ‘A Dublin street’, city: ‘Dublin'}})
- Dot-notation for a partial match
  - db.users.find({"address.city": ‘Dublin'})

Indexes

- Indexes in MongoDB are similar to indexes in RDBMS.
- MongoDB supports indexes on any field or sub-field contained in documents
- MongoDB defines indexes on a per collection level.
- All MongoDB indexes use a B-tree data structure.



Aggregation

- Aggregation operations exist to:
  - Group values from multiple documents together.
  - Perform operations on the grouped data to return a single result.
  - Analyze data changes over time.

How to perform aggregation?

Aggregation pipelines

- the preferred method for performing aggregations.

OR

Single purpose aggregation methods

- simple but lack the capabilities of an aggregation pipeline.

Aggregation Pipeline

- Can return results for groups of documents.
    - E.g. return the total, average, maximum, and minimum values.
- Consists of one or more stages that process documents
- Each stage performs an operation on the input documents.
    - E.g. a stage can filter documents, group documents, and calculate values.
- The documents that are output from a stage are passed to the next stage.

Create a collection of pizza orders

```
db.orders.insertMany( [
   { _id: 0, name: "Pepperoni", size: "small", price: 19,
     quantity: 10, date: ISODate( "2021-03-13T08:14:30Z" ) },
   { _id: 1, name: "Pepperoni", size: "medium", price: 20,
     quantity: 20, date : ISODate( "2021-03-13T09:13:24Z" ) },
   { _id: 2, name: "Pepperoni", size: "large", price: 21,
     quantity: 30, date : ISODate( "2021-03-17T09:22:12Z" ) },
   { _id: 3, name: "Cheese", size: "small", price: 12,
     quantity: 15, date : ISODate( "2021-03-13T11:21:39.736Z" ) },
   { _id: 4, name: "Cheese", size: "medium", price: 13,
     quantity:50, date : ISODate( "2022-01-12T21:23:13.331Z" ) },
   { _id: 5, name: "Cheese", size: "large", price: 14,
     quantity: 10, date : ISODate( "2022-01-12T05:08:13Z" ) },
   { _id: 6, name: "Vegan", size: "small", price: 17,
     quantity: 10, date : ISODate( "2021-01-13T05:08:13Z" ) },
   { _id: 7, name: "Vegan", size: "medium", price: 18,
     quantity: 10, date : ISODate( "2021-01-13T05:10:13Z" ) }
] )
```

Aggregation Stages

```
db.orders.aggregate( [
  // Stage 1: Filter pizza order documents by pizza size
  {
     $match: { size: "medium" }
  },
  // Stage 2: Group remaining documents by pizza name and calculate total quantity
  {
     $group: { _id: "$name", totalQuantity: { $sum: "$quantity" } }
  }
] )
```

- The $match stage: Filters out orders for medium size pizza and passes the remaining documents to the $group stage.
- The $group stage: Groups the remaining documents by pizza name and gives a quamtity per name.

Aggregation Stages

```
db.orders.aggregate( [
   // Stage 1: Filter pizza
order documents by pizza size
   {
      $match: { size: "medium" }
   },
   // Stage 2: Group remaining
documents by pizza name and
calculate total quantity
   {
      $group: { _id: "$name",
totalQuantity: { $sum:
"$quantity" } }
   }
] )
```

- Results:

```
[
 { _id: 'Vegan', totalQuantity: 10 },
 { _id: 'Pepperoni', totalQuantity: 20 },
 { _id: 'Cheese', totalQuantity: 50 }
]
```

```
db.orders.aggregate( [
   // Stage 1: Filter pizza order documents by
date range
   {
      $match:
      {
         "date": { $gte: new ISODate( "2020-
01-30" ), $lt: new ISODate( "2022-01-30" ) }
      }
   },
   // Stage 2: Group remaining documents by
date and calculate results
   {
      $group:
      {
         _id: { $dateToString: { format: "%Y-
%m-%d", date: "$date" } },
         totalOrderValue: { $sum: { $multiply:
[ "$price", "$quantity" ] } },
         averageOrderQuantity: { $avg:
"$quantity" }
      }
   },
   // Stage 3: Sort documents by
totalOrderValue in descending order
   {
      $sort: { totalOrderValue: -1 }
   }
 ] )
```

Aggregation

- The $match stage:
  - Filters the pizza order documents to those in a date range specified using $gte and $lt
- The $group stage:
  - Groups the documents by date using $dateToString
  - For each group, calculates:
  - Total order value using $sum and $multiply
  - Average order quantity using $avg
- The $sort stage:
  - Sorts the documents by the total order value for each group in descending order (-1).

Aggregation

- Total order value and average quantity between two dates:

```
db.orders.aggregate( [
   // Stage 1: Filter pizza order documents by
date range
   {
      $match:
      {
         "date": { $gte: new ISODate( "2020-
01-30" ), $lt: new ISODate( "2022-01-30" ) }
      }
   },
   // Stage 2: Group remaining documents by
date and calculate results
   {
      $group:
      {
         _id: { $dateToString: { format: "%Y-
%m-%d", date: "$date" } },
         totalOrderValue: { $sum: { $multiply:
[ "$price", "$quantity" ] } },
         averageOrderQuantity: { $avg:
"$quantity" }
      }
   },
   // Stage 3: Sort documents by
totalOrderValue in descending order
   {
      $sort: { totalOrderValue: -1 }
   }
 ] )
```

```
[
  { _id: '2022-01-12', totalOrderValue: 790,
averageOrderQuantity: 30 },
  { _id: '2021-03-13', totalOrderValue: 770,
averageOrderQuantity: 15 },
  { _id: '2021-03-17', totalOrderValue: 630,
averageOrderQuantity: 30 },
  { _id: '2021-01-13', totalOrderValue: 350,
averageOrderQuantity: 10 }
]
```

Aggregation Pipeline Stages

- $project - select certain fields from a collection
- $match - used in filtering operation and it can reduce the number of documents that are given as input to the next stage
- $group - groups all documents based on some keys.
- $sort – sorts all the documents.
- $skip & $limit – skip: can skip forward in the list of all documents for the given limit, limit: can limit the number of documents to look at by specifying the limit
- $first & $last - get the first and last values in each group of documents.
- $unwind - unwind for all documents, that are using arrays

Aggregation Expressions

- $sum
- $avg
- $max
- $min

Aggregation Expressions

$push	Inserts value into document
$addtoset	Inserts value into resultset
$first	First document according to group
$last	Last document according to group

Indexes and Exploring Query Performance

Indexes

- A Mongo index is a special data structure on which the index is created to hold the data of specific fields of documents.
- It stores a small portion of the data set that can be easily traversed later.
- A MongoDB index supports the efficient resolution of queries by storing the value of a specific field, ordered by the field's value as specified by the index.
- Why use an index?
  - Without an index when trying to retrieve data/information all documents will need to be scanned

Indexes

- A Mongo index is a special data structure on which the index is created to hold the data of specific fields of documents.
- It stores a small portion of the data set that can be easily traversed later.
- A MongoDB index supports the efficient resolution of queries by storing the value of a specific field, ordered by the field's value as specified by the index.

Indexes

- Create Index
- Syntax

```
db.COLLECTION_NAME.createIndex({KEY:1})
```

- Example

```
db.students.createIndex({student_name: 1})
```

Covered Query

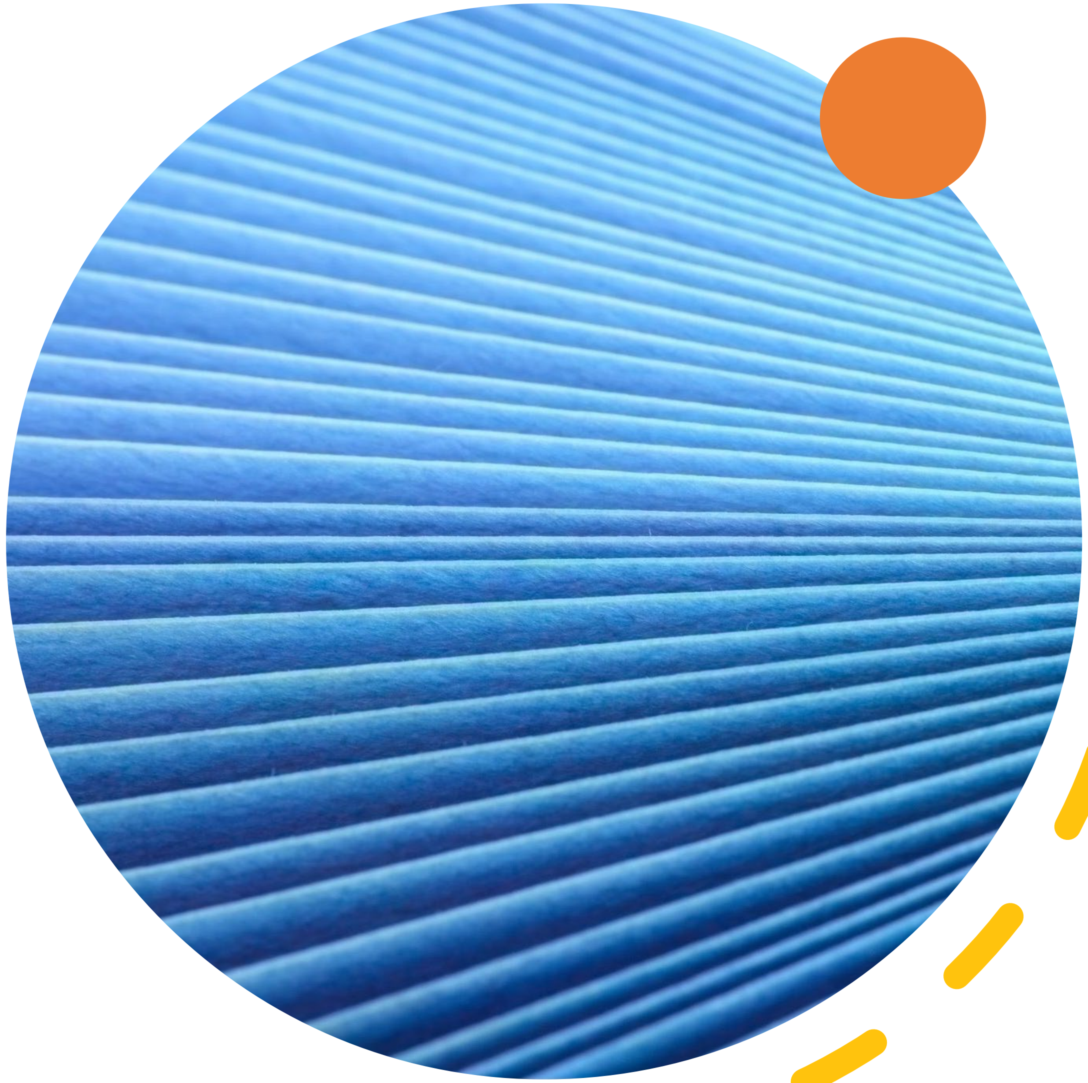
A query that can be satisfied entirely using an index and does not have to examine any documents

An index covers a query when all of the following apply:

all the fields in the query are part of an index, and

all the fields returned in the results are in the same index.

no fields in the query are equal to null (i.e. {"field" : null} or {"field" : {$eq : null}}).



Analysing Query Performance

Explain – equivalent of Explain Plan

Can identify

- how many documents were scanned
- how many documents were returned
- which index was used
- how long the query took to be executed
- which alternative execution plans were evaluated

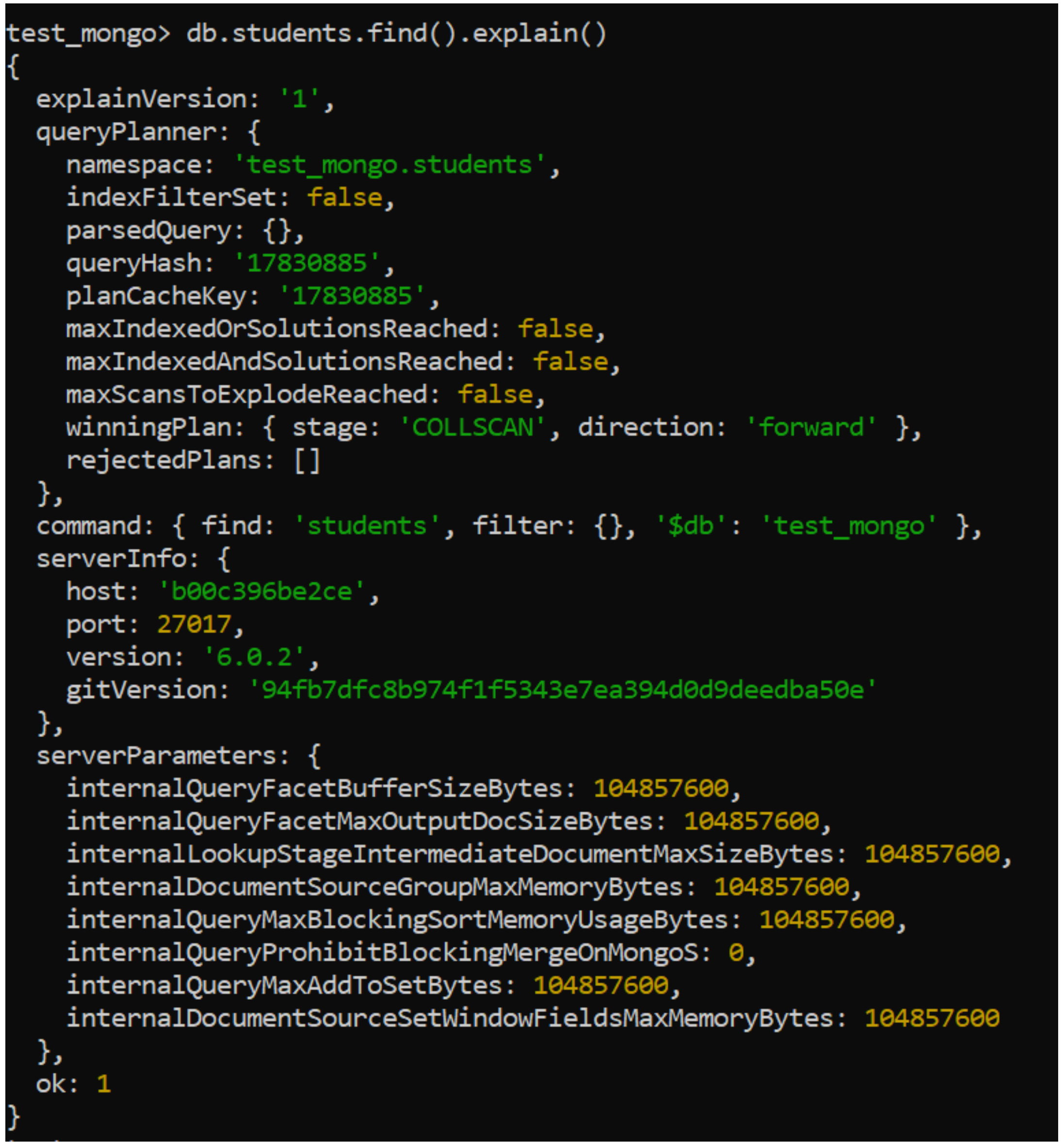
Explain

- For query to find all
- Syntax:

```
db.collection.find().explain()
```

- Example:

```
db.students.find().explain()
```

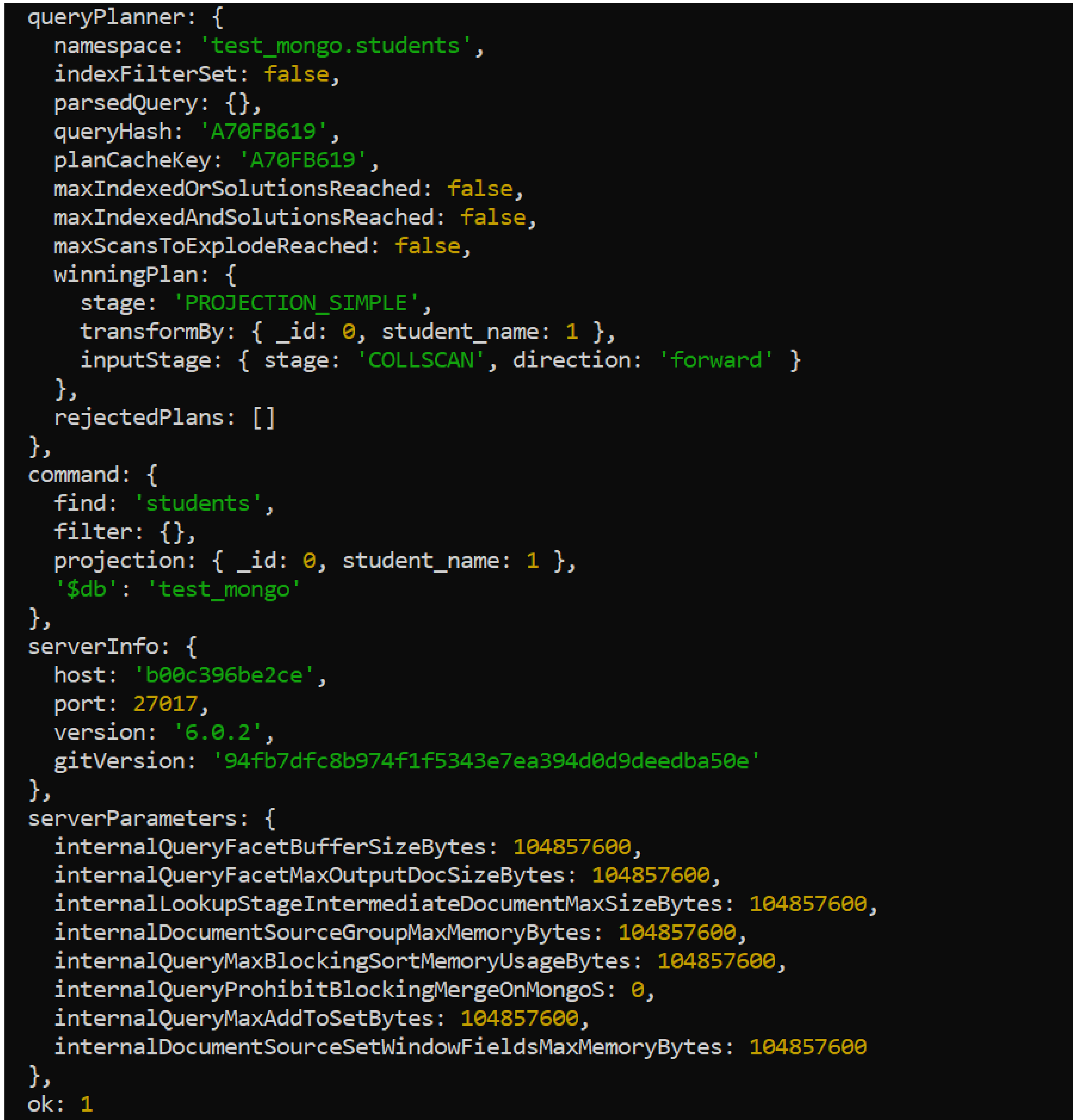


```
test_mongo> db.students.find().explain()
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'test_mongo.students',
    indexFilterSet: false,
    parsedQuery: {},
    queryHash: '17830885',
    planCacheKey: '17830885',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: { stage: 'COLLSCAN', direction: 'forward' },
    rejectedPlans: []
  },
  command: { find: 'students', filter: {}, '$db': 'test_mongo' },
  serverInfo: {
    host: 'b00c396be2ce',
    port: 27017,
    version: '6.0.2',
    gitVersion: '94fb7dfc8b974f1f5343e7ea394d0d9deedba50e'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1
}
```

Explain

- For projection
- Syntax:

```
db.students.find({}, {"_id":
0, "student_name":
1}).explain()
```

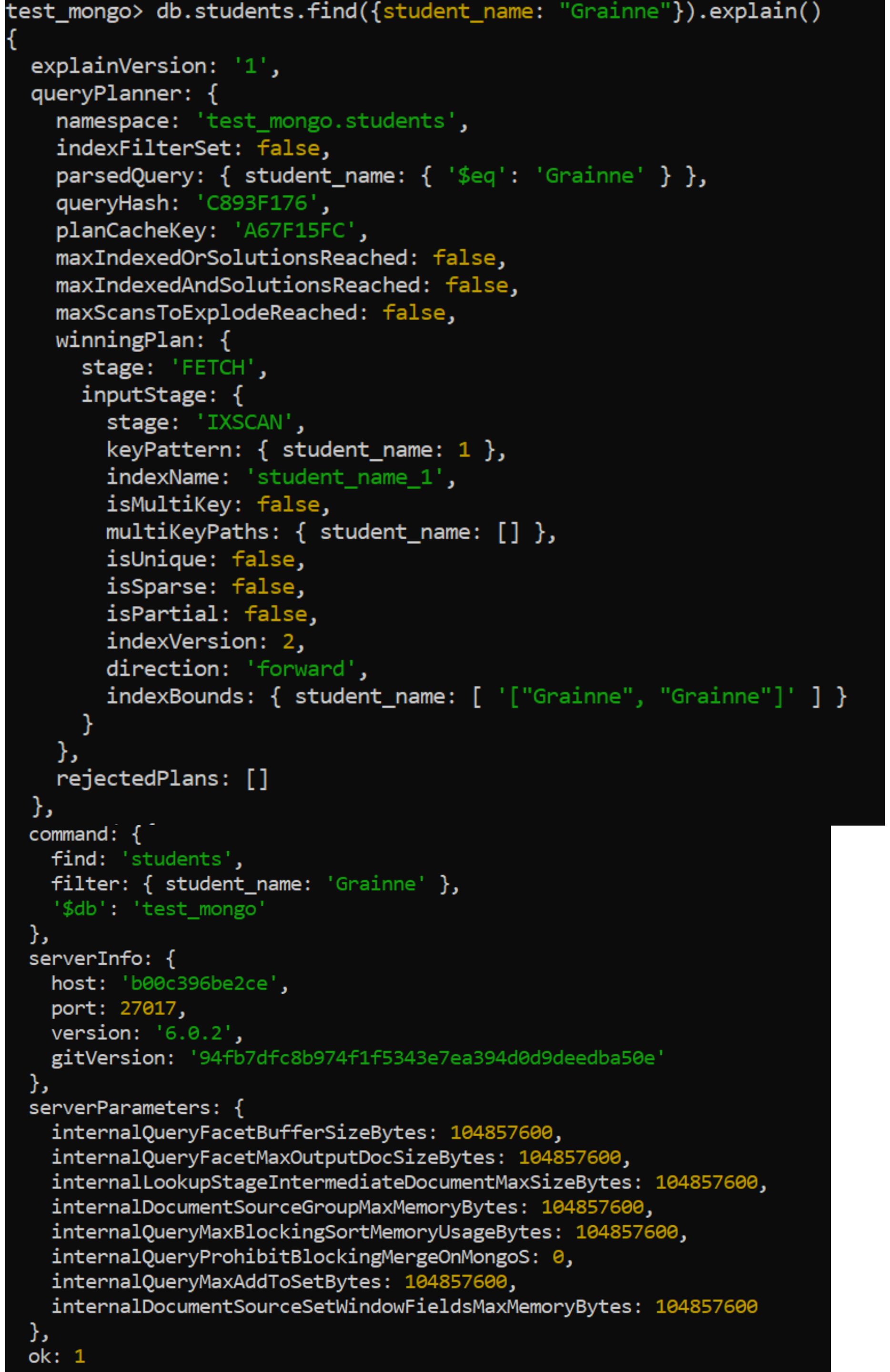


```
queryPlanner: {
  namespace: 'test_mongo.students',
  indexFilterSet: false,
  parsedQuery: {},
  queryHash: 'A70FB619',
  planCacheKey: 'A70FB619',
  maxIndexedOrSolutionsReached: false,
  maxIndexedAndSolutionsReached: false,
  maxScansToExplodeReached: false,
  winningPlan: {
    stage: 'PROJECTION_SIMPLE',
    transformBy: { _id: 0, student_name: 1 },
    inputStage: { stage: 'COLLSCAN', direction: 'forward' }
  },
  rejectedPlans: []
},
command: {
  find: 'students',
  filter: {},
  projection: { _id: 0, student_name: 1 },
  '$db': 'test_mongo'
},
serverInfo: {
  host: 'b00c396be2ce',
  port: 27017,
  version: '6.0.2',
  gitVersion: '94fb7dfc8b974f1f5343e7ea394d0d9deedba50e'
},
serverParameters: {
  internalQueryFacetBufferSizeBytes: 104857600,
  internalQueryFacetMaxOutputDocSizeBytes: 104857600,
  internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
  internalDocumentSourceGroupMaxMemoryBytes: 104857600,
  internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
  internalQueryProhibitBlockingMergeOnMongoS: 0,
  internalQueryMaxAddToSetBytes: 104857600,
  internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
},
ok: 1
```

Explain

- For projection where an index has been defined on the column
- Syntax:

```
db.students.find({student_name
: "Grainne"}).explain()
```



```
test_mongo> db.students.find({student_name: "Grainne"}).explain()
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'test_mongo.students',
    indexFilterSet: false,
    parsedQuery: { student_name: { '$eq': 'Grainne' } },
    queryHash: 'C893F176',
    planCacheKey: 'A67F15FC',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { student_name: 1 },
        indexName: 'student_name_1',
        isMultiKey: false,
        multiKeyPaths: { student_name: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { student_name: [ '["Grainne", "Grainne"]' ] }
      }
    },
    rejectedPlans: []
  },
  command: {
    find: 'students',
    filter: { student_name: 'Grainne' },
    '$db': 'test_mongo'
  },
  serverInfo: {
    host: 'b00c396be2ce',
    port: 27017,
    version: '6.0.2',
    gitVersion: '94fb7dfc8b974f1f5343e7ea394d0d9deedba50e'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1
```

Optimizing Aggregation Pipeline

- Reordering stages
- If you have a sequence with $sort followed by a $match, move the $match before the $sort to minimize the number of objects to sort.
- If you have you have a sequence with $skip followed by a $limit, move the $limit before the $skip.
- Similar recommendations for other stages

Optimizing Aggregation Pipeline

- An aggregation pipeline can use indexes from the input collection to improve performance.
- Using an index limits the amount of documents a stage processes.
- Ideally, an index can cover the stage query.
- A covered query has especially high performance, since the index returns all matching documents.
- To determine whether a pipeline uses indexes, review the query plan and look for IXSCAN or DISTINCT_SCAN plans.

What stages can benefit?

$match stage

$match can use an index to filter documents if it is the first stage in the pipeline, after any optimizations from the query planner.

$sort stage

$sort can benefit from an index as long as it is not preceded by a $project, $unwind, or $group stage.

$group stage

$group can use an index to find the first document in each group if it meets all of the following conditions:

- a $sort stage sorts the grouping field before $group
- an index exists that matches the sort order on the grouped field
- $first is the only accumulator in the $group stage

Example

- A pipeline that consists of $match, $sort, $group can benefit from indexes at every stage:
  - An index on the $match query field can efficiently identify the relevant data
  - An index on the sorting field can return data in sorted order for the $sort stage
  - An index on the grouping field that matches the $sort order can return all of the field values needed to execute the $group stage (a covered query)

Aggregation Pipeline Optimization

- If you know your query only requires a sub-set of documents –filter them first:
  - Use the $match, $limit, and $skip stages to restrict the documents that enter the pipeline.
  - When possible, put $match at the beginning of the pipeline to use indexes that scan the matching documents in a collection.
  - $match followed by $sort at the start of the pipeline is equivalent to a single query with a sort, and can use an index.

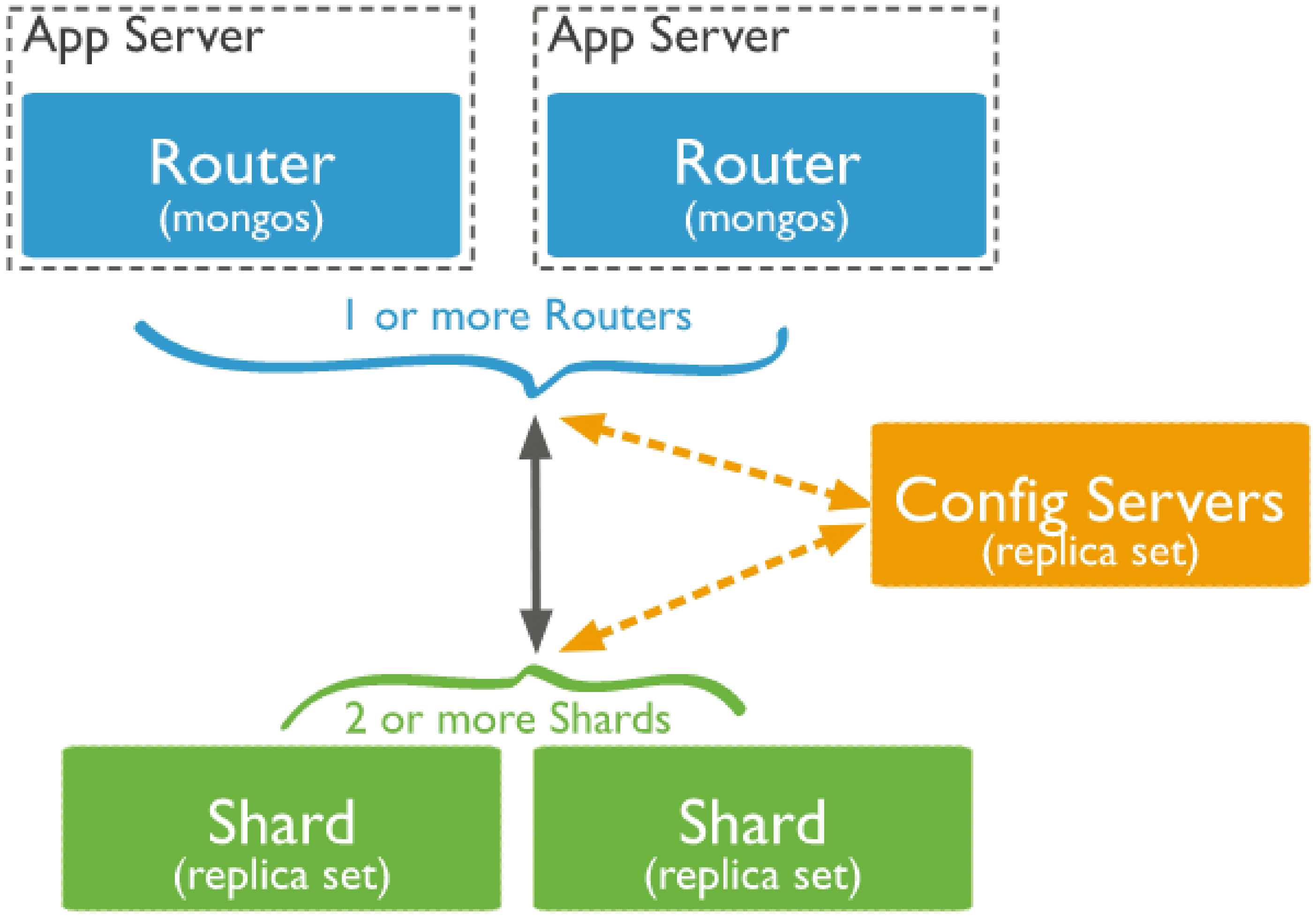
Sharding

Basics

- Horizontal partitioning approach.
- A method for distributing data across multiple machines.
- Use for very large data sets and high throughput operations.
- Shards in MongoDB are at **collection level**

Sharded Cluster

- Consists of:
  - Shards: each shard contains a subset of the sharded data.
    - each shard can be deployed as a replica set.
  - Mongos: act as a query router, providing an interface between client applications and the sharded cluster.
  - Config servers: Config servers store metadata and configuration settings for the cluster.



Shard Keys

- Shard keys are used to distribute the collection's documents across shards.
- The shard key consists of a field or multiple fields in the documents.
- The shard key when sharding a collection.
- A document's shard key value determines its distribution across the shards.

Shard Key Index

- To shard a populated collection, the collection must have an index that starts with the shard key.
- If you are sharding an empty collection, MongoDB creates the supporting index if the collection does not already have an appropriate index for the specified shard key.

Shard Key Strategy

- Choice of shard key affects the performance, efficiency, and scalability of a sharded cluster.
- A cluster with the best possible hardware and infrastructure can be bottlenecked by the choice of shard key.
- The choice of shard key and its backing index can also affect the sharding strategy that your cluster can use.

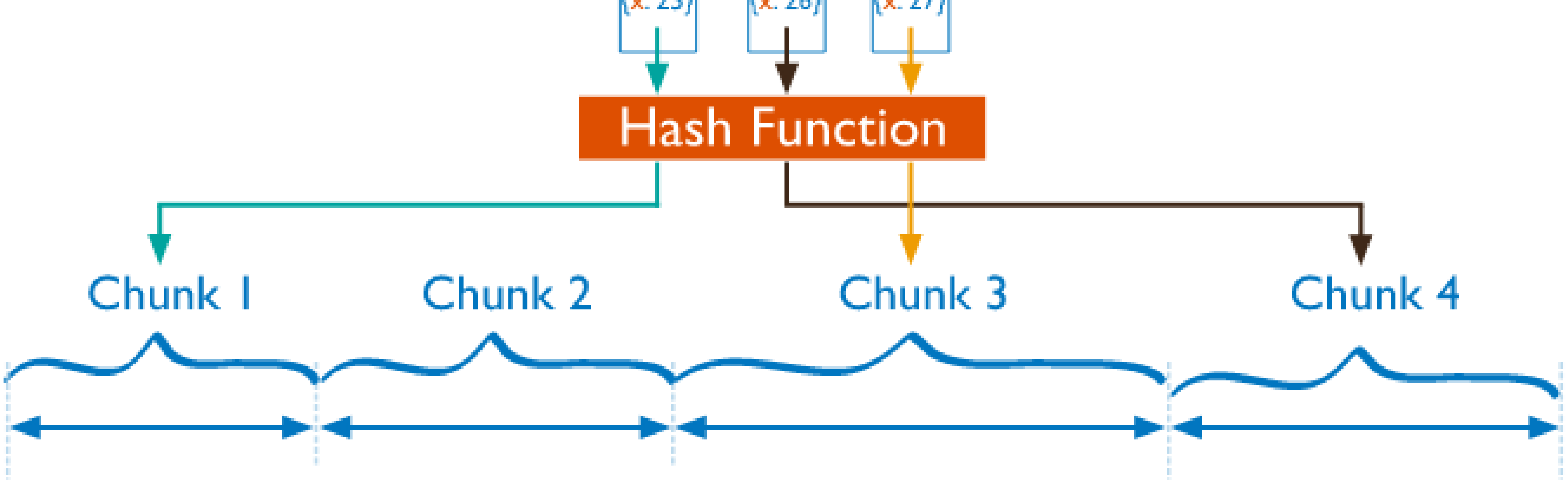
Chunks

MongoDB partitions sharded data into chunks.

Each chunk has an inclusive lower and exclusive upper range based on the shard key.

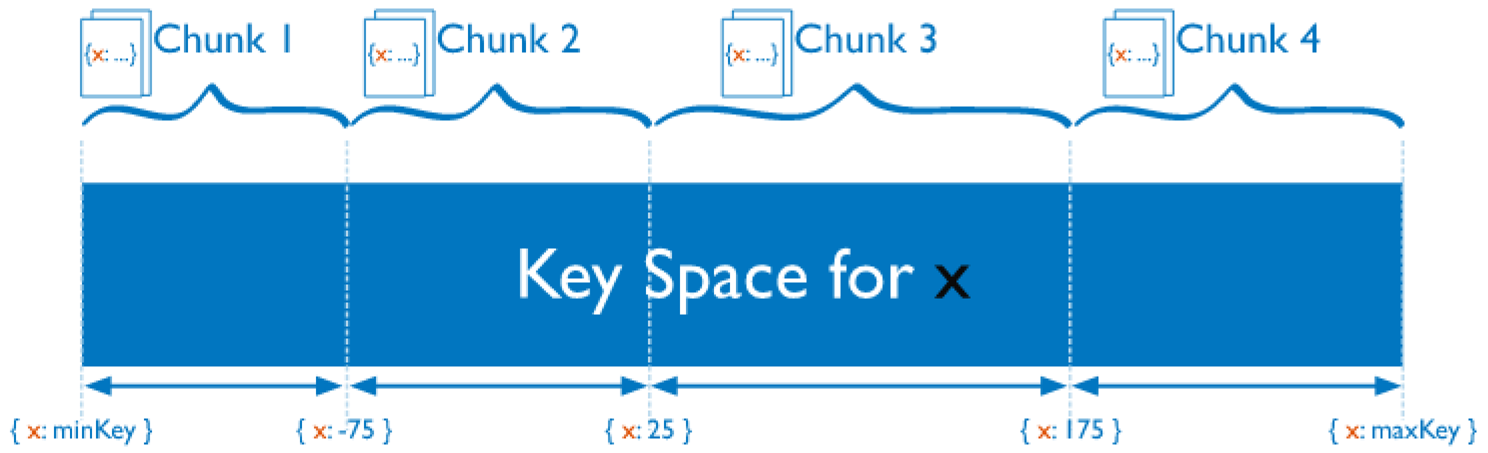
To try to achieve an even distribution of chunks across all shards in the cluster, a **balancer** runs in the background

- The balancer aims to migrate **chunks** across the **shards**.



Sharding Strategy

- Hashed
  - Involves computing a hash of the shard key field's value.
  - Each chunk is then assigned a range based on the hashed shard key values.
  - While a range of shard keys may be "close", their hashed values are unlikely to be on the same chunk.
  - Data distribution based on hashed values facilitates more even data distribution, especially in data sets where the shard key changes monotonically i.e. always increasing, never decreasing.



Sharding Strategy

- Ranged
  - Involves dividing data into ranges based on the shard key values.
  - Each chunk is then assigned a range based on the shard key values.
  - A range of shard keys whose values are "close" are more likely to reside on the same chunk.
  - This allows for targeted operations as a mongos can route the operations to only the shards that contain the required data.
  - Efficiency depends on choice of shard key

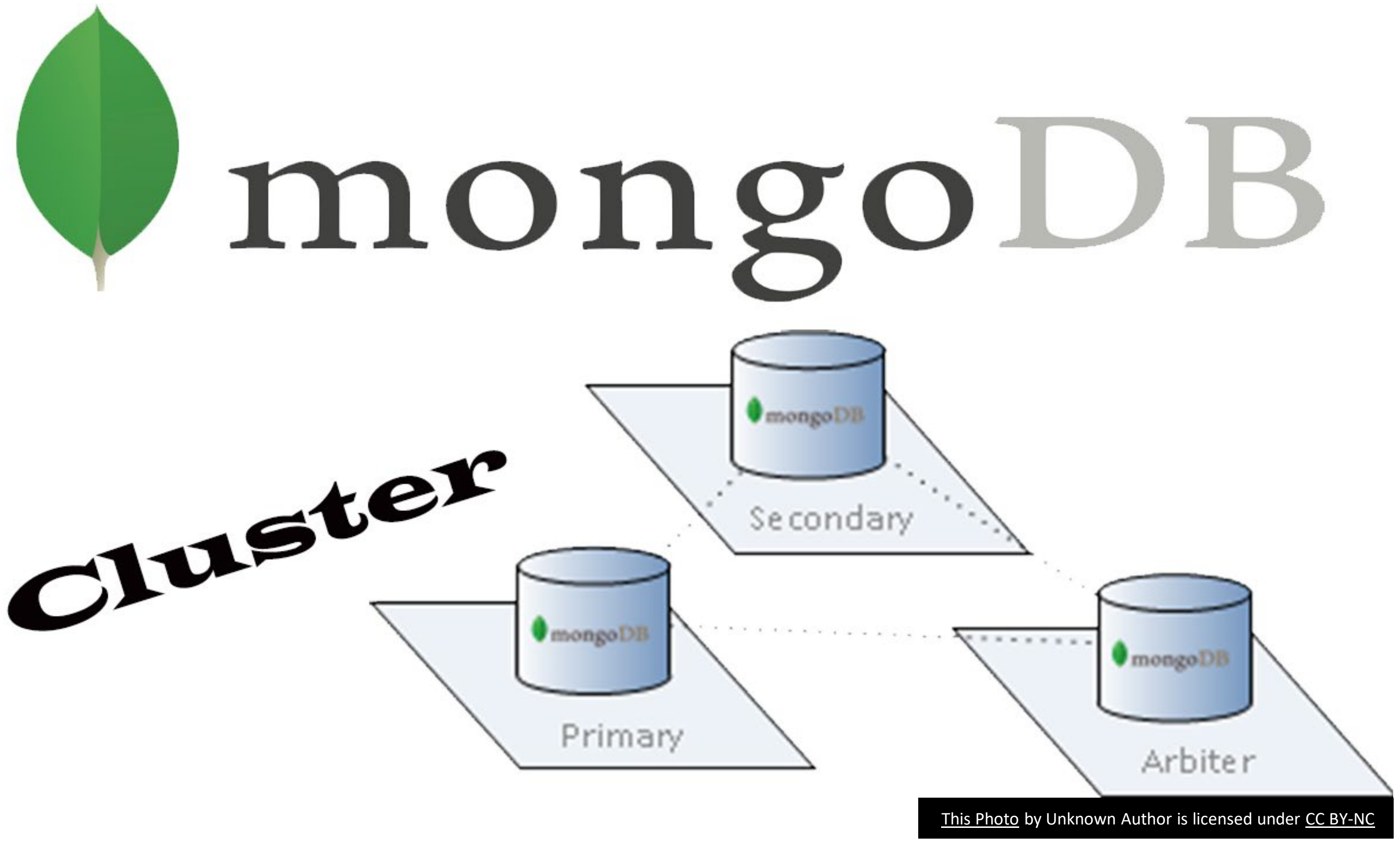
Before Sharding…

Sharded cluster infrastructure requirements and complexity require careful planning, execution, and maintenance.

Once a collection has been sharded, MongoDB provides no method to unshard a sharded collection.

You can reshard your collection later, but you need to carefully consider your shard key choice to avoid scalability and performance issues.

Connecting to a Sharded Cluster

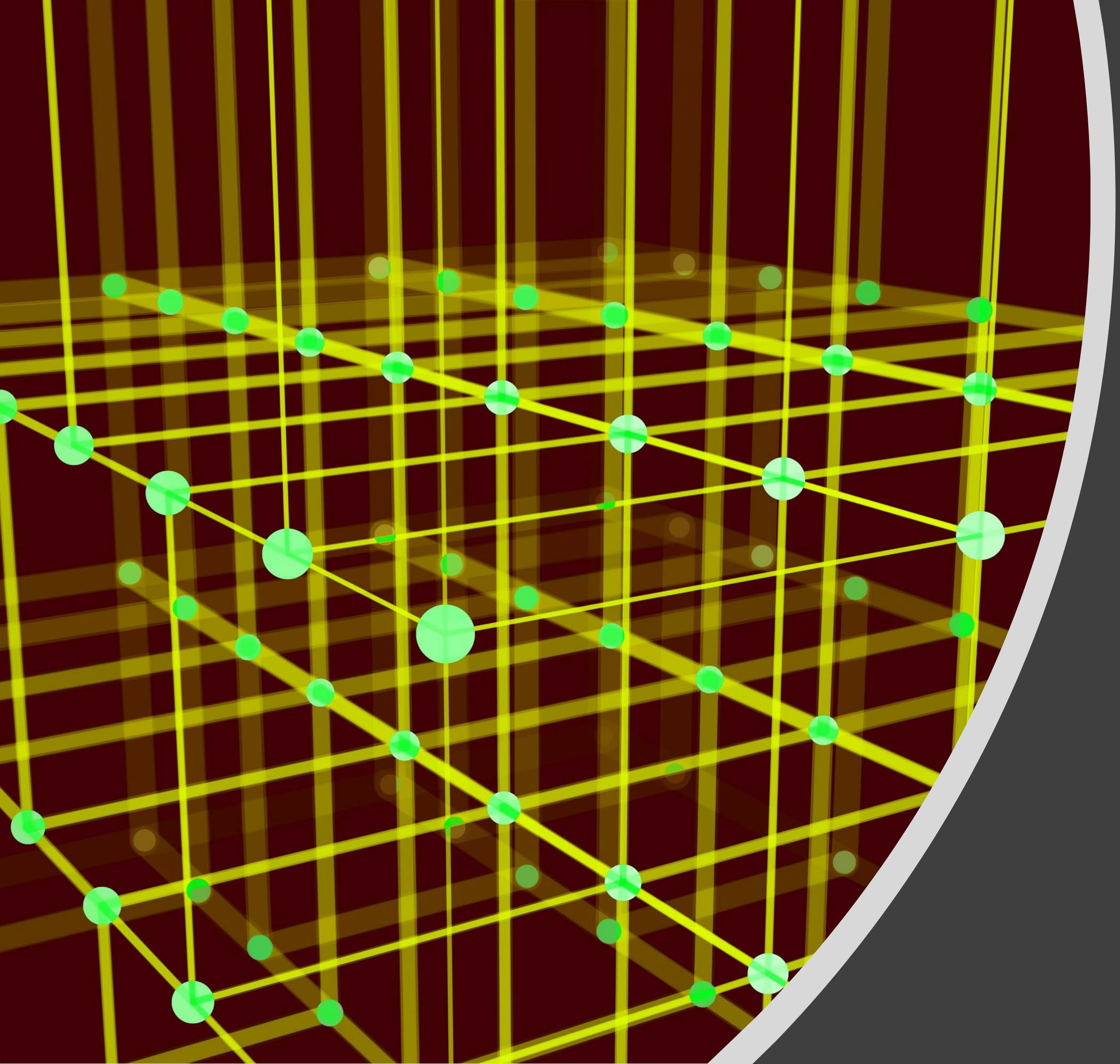


You must connect to a mongos (MongoDB Shard) router to interact with any collection in the sharded cluster.

This includes sharded and unsharded collections

- .

Clients should never connect to a single shard in order to perform read or write operations.



Replication

MongoDB Replication

Process of creating a copy of the same data set in more than one MongoDB server.

Can be achieved by using a Replica Set.

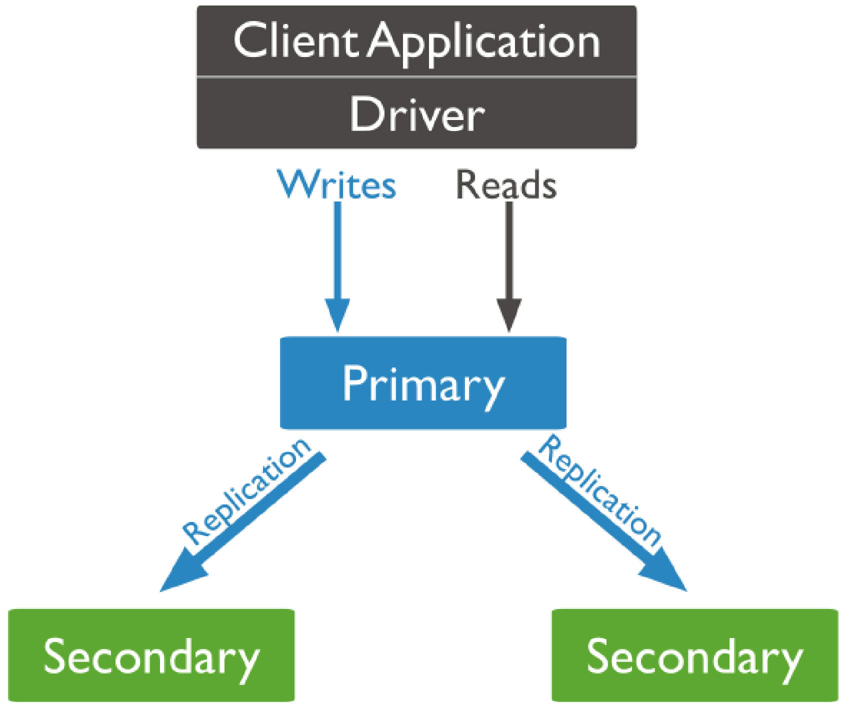
A replica set is a group of MongoDB instances that maintain the same data set and pertain to any mongodb process.

Why would you replicate?

- Provides:
  - Data redundancy
  - High availability of data
  - Increase fault tolerance
  - Improves performance
- Maintaining multiple MongoDB servers with the same data provides distributed access to the data while increasing the fault tolerance of the database by providing backups.
- Replication can also be used as a part of load balancing, where read and write operations can be distributed across all the instances depending on the use case.

How does it work?

- A Replica Set requires a minimum of three MongoDB nodes:
  - One of the nodes will be considered the primary node that receives all the write operations.
  - The others are considered secondary nodes.
    - These secondary nodes will replicate the data from the primary node.

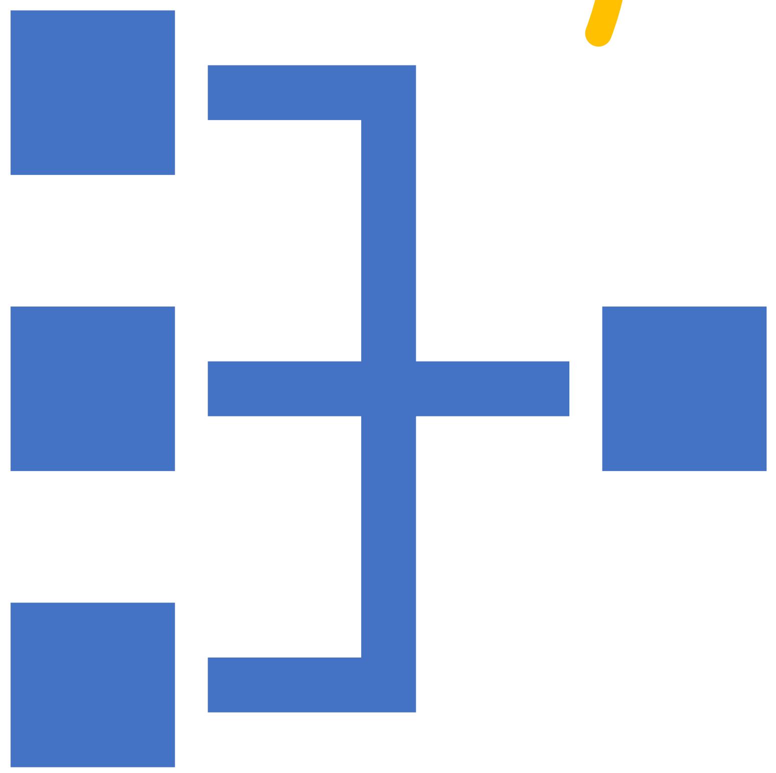


How does it work?

- The primary node is the only instance that accepts write operations
- Any other node within a replica set can accept read operations.
- If the primary node is unavailable or inoperable, a secondary node will take the primary node’s role to provide continuous availability of data.
  - The primary node selection is made through a process called Replica Set Elections, where the most suitable secondary node is selected as the new primary node.

Heartbeat Process

- Identifies the current status of a MongoDB node in a replica set.
    - The replica set nodes send pings to each other every two seconds (hence the name).
    - If any node doesn't ping back within 10 seconds, the other nodes in the replica set mark it as inaccessible.
- Needed for the failover in the event that the primary node is unreachable
    - If the secondary nodes do not receive a heartbeat from the primary nodes within the allocated time frame, MongoDB will automatically assign a secondary node to take over as the primary node.



Replica Set Elections

- Elections in replica sets are used to determine which MongoDB node should become the primary node.
- Can occur due to:
  - Loss of connectivity to the primary node (detected by heartbeats)
  - Initializing a replica set
  - Adding a new node to an existing replica set
  - Maintenance of a Replica set using stepDown or rs.reconfig methods

Replica Set Elections - Process

- A node will raise a flag requesting an election
- All the other nodes will vote to elect that node as the primary node
- Average time for an election process to complete is 12 seconds
- Network latency can cause delays in getting the replica set back to operation with the new primary node.

Replica Set Elections

- The replica set cannot process any write operations until the election is completed.
- However, read operations can be served if read queries are configured to be processed on secondary nodes.

MongoDB Replica Set vs MongoDB Sharded Cluster

- A replica set creates multiple copies of the same data set across the replica set nodes.
  - The basic objective of a replica set is to:
    - Increase data availability
    - Provide a built-in backup solution
- A Mongo Sharded cluster distributes the data across multiple nodes using a shard key.
  - The basic objective is to
    - Support extremely large data sets and high throughput operations by horizontally scaling the workload.
  - Breaks data into multiple pieces called shards and then copies each shard to a separate node.

MongoDB Replica Set vs MongoDB Sharded Cluster

- A replica set copies the data set as a whole.
- A cluster distributes the workload and stores pieces of data (shards) across multiple servers.

Demo

Consistency

Consistency

Strongly consistent by default

If there is a partition it chooses Consistency over Availability

Causal Consistency

- If an operation logically depends on a preceding operation, there is a causal relationship between the operations.
- A causally consistent session denotes that the associated sequence of read operations with "majority" read concern and write operations with "majority" write concern have a causal relationship that is reflected by their ordering.
- If some process updates a given object, all the processes that acknowledge the update on this object will consider the updated value.
- However, if some other process does not acknowledge the write operation, they will follow the eventual consistency model.
- Causal consistency is weaker than sequential consistency (RDBMS) but stronger than eventual consistency.

Working with Replication/Sharding - Read/Write Concern

Write Concern

- Allows you to set the level of acknowledgment for a desired write operation.

Read concern

- Allows you to control the consistency and isolation properties of the data read from your replica sets

Read Concern Level

Local

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)

Majority

Query returns the data that has been acknowledged by a majority of the replica set members.

documents returned by the read operation are durable, even in the event of failure.

Available

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) (not for causally consistent sessions)

Write Concern Level

<number>

requests acknowledgment that the write operation has propagated to a specified number of mongod instances or to mongod instances with specified tags

Majority

Requests acknowledgment that write operations have propagated to the

calculated majority

of the data-bearing voting members ({ w: "majority" } - the default write concern for most MongoDB deployments.

Default is 1+half the no. of replicas

<custom write concern name>

Requests acknowledgment that the write operations have propagated to tagged members that satisfy the custom write concern defined in settings.getLastErrorModes

MongoDB ACID

- MongoDB supports multi-document ACID transactions and distributed multi-document ACID transactions for the use cases that require them.
- To use a transaction, start a MongoDB session using an appropriate driver.
- Use that session to execute a group of database operations.
  - Can be operations across multiple documents, multiple collections, and multiple shards.