CHAPTER 2

MongoDB Architecture and Concepts

This chapter aims to equip you with an understanding of MongoDB architecture and internals referenced in subsequent chapters, which are necessary for MongoDB performance tuning.

A MongoDB tuning professional should be broadly familiar with these main areas of MongoDB technology:

- The MongoDB document model
- The way MongoDB applications interact with the MongoDB database server through the MongoDB API
- The MongoDB *optimizer*, which is the software layer concerned with maximizing the performance of MongoDB requests
- The MongoDB server architecture, which comprises the memory, processes, and files that interact to provide database services

Readers who feel thoroughly familiar with this material may wish to skim or skip this chapter. However, we will be assuming in subsequent chapters that you are familiar with the core concepts presented here.

The MongoDB Document Model

As you are no doubt aware, MongoDB is a *document database*. Document databases are a family of non-relational databases which store data as structured documents – usually in *JavaScript Object Notation* (*JSON*) format.

© Guy Harrison, Michael Harrison 2021
G. Harrison and M. Harrison, *MongoDB Performance Tuning*, https://doi.org/10.1007/978-1-4842-6879-7_2

JSON-based document databases like MongoDB have flourished over the past decade for many reasons. In particular, they address the conflict between object-oriented programming and the relational database model which had long frustrated software developers. The flexible document schema model supports agile development and DevOps paradigms and aligns closely with dominant programming models – especially those of modern, web-based applications.

JSON

MongoDB uses a variation of *JavaScript Object Notation* (JSON) as its data model, as well as for its communication protocol. JSON documents are constructed from a small set of elementary constructs – *values*, *objects*, and *arrays*:

- **Arrays** consist of lists of values enclosed by square brackets ("["and "]") and separated by commas (",").
- **Objects** consist of one or more name-value pairs in the format "name":"value", enclosed by braces ("{"and :}") and separated by commas (",").
- **Values** can be Unicode strings, standard format numbers (possibly including scientific notation), Booleans, arrays, or objects.

The last few words in the preceding definition are critical. Because values may include objects or arrays, which themselves contain values, a JSON structure can represent an arbitrarily complex and nested set of information. In particular, arrays can be used to represent repeating groups of documents which in a relational database would require a separate table.

Binary JSON (BSON)

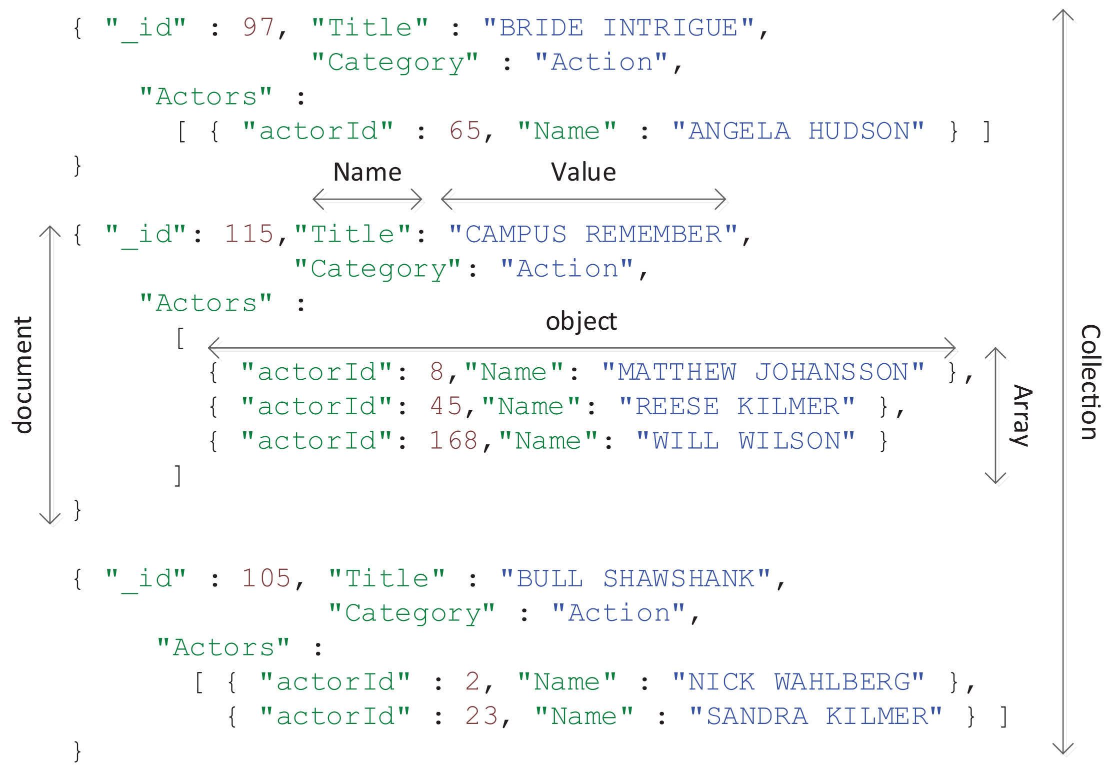
MongoDB stores JSON documents internally in the *Binary JSON* (*BSON*) format. BSON is designed to be a more compact and efficient representation of JSON data and uses more efficient encoding for numbers and other data types. For instance, BSON includes field length prefixes that allow scanning operations to "skip over" elements and hence improve efficiency.

BSON also provides a number of extra data types not supported in JSON. For example, a numeric value in JSON could be a Double, Int, Long, or Decimal128 in BSON. Additional types such as ObjectID, Date, and BinaryData are also commonly used. However, most of the time, the differences between JSON and BSON are unimportant.

Collections

MongoDB allows you to organize "similar" documents into *collections*. Collections are analogous to tables in a relational database. Usually, you'll store only documents with a similar structure or purpose within a specific collection, though by default the structure of the documents in a collection is not enforced.

Figure 2-1 shows the internal structure of JSON documents and how documents are organized into collections.



```
{ "_id" : 97, "Title" : "BRIDE INTRIGUE",
              "Category" : "Action",
    "Actors" :
      [ { "actorId" : 65, "Name" : "ANGELA HUDSON" } ]
}

{ "_id": 115,"Title": "CAMPUS REMEMBER",
             "Category": "Action",
    "Actors" :
      [
        { "actorId": 8,"Name": "MATTHEW JOHANSSON" },
        { "actorId": 45,"Name": "REESE KILMER" },
        { "actorId": 168,"Name": "WILL WILSON" }
      ]
}

{ "_id" : 105, "Title" : "BULL SHAWSHANK",
               "Category" : "Action",
     "Actors" :
       [ { "actorId" : 2, "Name" : "NICK WAHLBERG" },
         { "actorId" : 23, "Name" : "SANDRA KILMER" } ]
}
```

Figure 2-1. *JSON document structure*

MongoDB Schemas

The MongoDB document model allows for objects that would require many tables in a relational database to be stored within a single document.

Consider the following MongoDB document:

```
{
  _id: 1,
  name: 'Ron Swanson',
  address: 'Really not your concern',
  dob: ISODate('1971-04-15T01:03:48Z'),
  orders: [
    {
      orderDate: ISODate('2015-02-15T09:05:00Z'),
      items: [
        { productName: 'Meat damper', quantity: 999 },
        { productName: 'Meat sauce', quantity: 9 }
      ]
    },
    { otherorders  }
  ]
};
```

As in the preceding example, a document may contain another subdocument, and that subdocument may itself contain a subdocument and so on. Two limits will eventually stop this document nesting: a default limit of 100 levels of nesting and a 16MB size limit for a single document (including all its subdocuments).

In database parlance, a *schema* defines the structure of data within a database object. By default, a MongoDB database does not enforce a schema, so you can store whatever you like in a collection. However, it is possible to create a schema to enforce the document structure using the `validator` option of the `createCollection` method, as in the following example:

```
db.createCollection("customers", {
  "validator": {
      "$jsonSchema": {
          "bsonType": "object",
"additionalProperties": false,
"properties": {
    "_id": {
        "bsonType": "objectId"
    },
    "name": {
        "bsonType": "string"
    },
    "address": {
        "bsonType": "string"
    },
    "dob": {
        "bsonType": "date"
    },
    "orders": {
        "bsonType": "array",
        "uniqueItems": false,
        "items": {
            "bsonType": "object",
            "properties":  {
                  "orderDate": { "bsonType": "date"},
                  "items": {
                      "bsonType": "array",
                      "uniqueItems": false,
                      "items": {
                          "bsonType": "object",
                          "properties": {
                              "productName": {
                                  "bsonType": "string"
                              },
                              "quantity": {
                                  "bsonType": "int"
                              }
                          }
                      }
```

```
                              }
                        }
                    }
                }
            }
        }
    },
    "validationLevel": "strict",
    "validationAction": "warn"
});
```

The validator is in the *JSON schema* format – which is an open standard that allows for JSON documents to be annotated or validated. A JSON schema document will generate warnings or errors if a MongoDB command results in a document that does not match the schema definition. JSON schemas can be used to define mandatory attributes, restrict other attributes, and define the data types or data ranges that a document attribute can adopt.

The MongoDB Protocol

The MongoDB protocol defines the communication mechanism between the client and the server. Although the fine details of the protocol are outside the scope of our performance tuning efforts, it is important to understand the protocol, since many of the diagnostic tools will display data in the MongoDB protocol format.

Wire Protocol

The protocol for MongoDB is also known as the MongoDB *wire protocol*. This is the structure of the MongoDB packets which are sent to and received from the MongoDB server. The wire protocol runs over a TCP/IP connection – by default over port 27017.

The actual packet structure of the wire protocol is beyond our scope, but the essence of each packet is a JSON document containing a request or a response. For instance, if we send a command to MongoDB from the shell like this:

```
db.customers.find({FirstName:'MARY'},{Phone:1}).sort({Phone:1})
```

then the shell will send a request across the wire protocol that looks something like this:

```
{ "find" : "customers",
  "filter" : { "FirstName" : "MARY" },
  "sort" : { "Phone" : 1.0 },
  "projection" : { "Phone" : 1.0},
  "$db" : "mongoTuningBook",
  "$clusterTime" : { "clusterTime" : {
        "$timestamp" : { "t" : 1589596899, "i" : 1 } },
   "signature" : { "hash" : { "$binary" : { "base64" : ]
             "4RGjzZI5khOmM9BBWLz6y9xLZ9w=", "subType" : "00" } },
    "keyId" : 6826926447718825986 } },
    "lsid" : { "id" : { "$binary" : { "base64" :
    "JI3lUrOMRQmOY6Pr3iQ8EQ==", "subType" : "04" } } } }
```

MongoDB Drivers

A MongoDB driver translates requests from a programming language into wire protocol format. Each driver can have subtle syntax differences. For instance, in NodeJS the preceding MongoDB shell request is subtly different:

```
const docs = await db.collection('customers').
        find({'FirstName': 'MARY'},
             {'Phone': 1}).
        sort({Phone: 1}).toArray();
```

Because NodeJS is a JavaScript platform, the syntax is still similar to the MongoDB shell. But in other languages, the differences can be more marked. For instance, here is the same query in the Go language:

```
collection := client.Database("MongoDBTuningBook").
            Collection("customers")
filter := bson.D{{"FirstName", "MARY"}}
findOptions := options.Find()
findOptions.SetSort(map[string]int{"Phone": 1})
findOptions.SetProjection(map[string]int{"Phone": 1})
```

```
      cursor, err := collection.Find(ctx, filter, findOptions)
      var results []bson.M
      cursor.All(ctx, &results)
```

However, regardless of the syntax required by a MongoDB driver, the MongoDB server always receives packets which are in the standard wire protocol format.

MongoDB Commands

Logically MongoDB commands break down into the following categories:

- **Query commands**, such as `find()` and `aggregate()`, which return information from the databases
- **Data manipulation commands**, such as `insert()`, `update()`, and `delete()`, which modify data within the database
- **Data definition commands**, such as `createCollection()` and `createIndex()`, which define the structure of data in the database
- **Administration commands**, such as `createUser()` and `setParameter()`, which control the operations of the database

Database performance management is mainly concerned with the overhead and throughput of query and data manipulation statements. However, administration and data definition commands include some of the "tools of the trade" that we use to resolve performance problems (see Chapter 3).

The find Command

The find command is the workhorse of MongoDB data access. It has a quick and easy syntax and has a flexible and powerful filtering capability. The `find()` command has the following high-level syntax:

```
db.collection.find(
      {filter},
      {projection})
  sort({sortCondition}),
  skip(skipCount),
  limit(limitCount)
```

The preceding syntax is shown for the Mongo shell; the syntax for language-specific drivers can vary slightly.

The key parameters to the `find()` command are as follows:

- **Filter** is a JSON document that defines the documents to be returned.
- **Projection** defines the attributes from each document which will be returned.
- **Sort** defines the order in which documents will be returned.
- **Skip** allows some initial documents in the output to be skipped.
- **Limit** restricts the total number of documents to be returned.

In the wire protocol, a `find()` command returns just the first batch of documents (usually 1000), and subsequent batches are fetched by a `getMore` command. The MongoDB drivers generally handle `getMore` processing statements on your behalf, but you can vary the batch size to optimize performance in many cases (see Chapter 6).

The aggregate Command

`find()` can perform a wide variety of queries, but it lacks many of the capabilities of the relational database's SQL command. For instance, a `find()` operation cannot join data from multiple collections and cannot aggregate data. When you need more functionality than `find()`, you will generally turn to `aggregate()`.

At a high level, the syntax for aggregate is deceptively simple:

```
db.collection.aggregate([pipeline]);
```

where *`pipeline`* is an array of instructions to the aggregate command. Aggregate supports more than two dozen pipeline operators, and most are beyond the scope of this book. However, the most commonly used operators are

- **$match**, which filters documents within a pipeline using a syntax similar to the `find()` command
- **$group**, which aggregates multiple documents into a smaller aggregated set
- **$sort**, which sorts documents within the pipeline

- **$project**, which defines the attributes to be returned from each document
- **$unwind**, which returns one document for each element in an array
- **$limit**, which restricts the number of documents to be returned
- **$lookup**, which joins documents from another collection

Here's an example of aggregate that uses most of these operations to return a count of movie views by category:

```
db.customers.aggregate([
  { $unwind:  "$views" },
  { $project: {
         "filmId": "$views.filmId"
        }
  },
  { $group:{     _id:{ "filmId":"$filmId"  },
             "count":{$sum:1}
     }
  },
  { $lookup:
     { from:         "films",
       localField:   "_id.filmId",
       foreignField: "_id",
       as:           "filmDetails"
     }
  },
  { $group:{     _id:{
             "filmDetails_Category":"$filmDetails.Category"},
             "count":{$sum:1},
             "count-sum":{$sum:"$count"}
     }
  },
  { $project: {
          "category": "$_id.filmDetails_Category"  ,
          "count-sum": "$count-sum"
         }
  },
  { $sort:{  "count-sum":-1 }},
]);
```

Aggregation pipelines can be hard to write and hard to optimize. We'll look in detail at aggregation pipeline optimization in Chapter 7.

Data Manipulation Commands

`insert()`, `update()`, and `delete()` allow documents to be added, changed, or removed from a collection.

Both `update()` and `delete()` take a filter argument that defines the documents to be processed. The filter condition is identical to that from the `find()` command.

Optimization of the filter condition is usually the most important factor when optimizing updates and deletes. Their performance is also affected by the configuration of *write concern* (see the following section).

Here is an example of insert, update, and delete commands:

```
db.myCollection.insert({_id:1,name:'Guy',rating:9});
db.myCollection.update({_id:1},{$set:{rating:10}});
db.myCollection.deleteOne({_id:1});
```

We discuss the optimization of data manipulation statements in Chapter 8.

Consistency Mechanisms

All databases have to make trade-offs between consistency, availability, and performance. Relational databases like MySQL are regarded as *strongly consistent* databases because all users always see a consistent view of data. Non-relational databases such as Amazon Dynamo are often called *weakly consistent* or *eventually consistent* databases because users are not guaranteed to see such a consistent view.

MongoDB is – within limitations – strongly consistent by default, although it can be made to behave like an eventually consistent database through the configuration of *write concern* and *read preference*.

Read Preference and Write Concern

A MongoDB application has some control over the behavior of read and write operations, providing a degree of tunable consistency and availability.

- The **write concern** setting determines when MongoDB regards a write operation as having completed. By default, write operations complete once the primary has received the modification. Consequently, if the primary should fail irrecoverably, then data might be lost.
- However, if the write concern is set to "*majority*", then the database will not complete the write operation until a majority of secondaries receive the write. We can also set the write concern to wait until all secondaries or a specific number of secondaries receive the write operation.
- Write concern can also determine if write operations proceed to the on-disk *journal* before being acknowledged. This is true by default.
- The **read preference** determines where a client sends read requests. By default, read requests are sent to the primary. However, the client driver can be configured to send read requests to the secondary by default, to a secondary only if the primary is not available, or to whichever server is "nearest." The later setting is intended to favor low latency over consistency.

The default settings for the read preference and write concern result in MongoDB behaving as a strictly consistent system: everybody will see the same version of a document. Allowing reads to be satisfied from a secondary node results in a more eventually consistent behavior.

Read preference and write concern have definite performance impacts that we will discuss in Chapters 8 and 13.

Transactions

Although MongoDB started its life as a non-transactional database, since version 4.0 it has been possible to perform atomic transactions across multiple documents. For instance, in this example we atomically reduce the balance of one account by 100 and increment another account by the same amount:

```
session.startTransaction();
mycollection.update({userId:1},{$inc:{balance:100}});
mycollection.update({userId:2},{$inc:{balance:-100}});
session.commitTransaction();
```

The two updates will either both succeed or both fail.

In practice, coding transactions require some error handling logic, and the design of transactions can significantly affect performance. We discuss these considerations in Chapter 9.

Query Optimization

Like most databases, MongoDB commands represent a logical request for data, rather than a series of instructions for retrieving that data. For instance, a `find()` operation specifies the data that will be returned, but does not explicitly specify the indexes or other access methods to be employed in retrieving the data.

As a result, the MongoDB code must determine the most efficient way to process data requests. The *MongoDB optimizer* is the MongoDB code that makes these determinations. The decision that the optimizer makes for each command is referred to as the *query plan*.

When a new query or command is sent to MongoDB, the optimizer performs the following steps:

1. The optimizer looks for a matching query in the MongoDB *plan cache*. A matching query is one in which all of the filter and operation attributes match, even if the values do not. Such queries are said to have the same *query shape*. For instance, if you issue the same query against the `customers` collection for different customer names, MongoDB will consider these to have the same query shape.

2. If the optimizer cannot find a matching query, then the optimizer will consider all the possible ways of executing the query. The query that has the lowest number of *work units* will be successful. Work units are specific operations that MongoDB must perform – correlating mostly with the number of documents that must be processed.

3. MongoDB will select the plan that has the lowest number of work units, use that plan to execute the query, and store that query plan in the plan cache.

In practice, MongoDB tends to use index-based plans whenever possible and will usually choose the index that is the most *selective* (see Chapter 5).

MongoDB Architecture

You can do a lot of performance optimization without any reference to MongoDB architecture. However, if we do our job well and completely optimize the workload, eventually the limiting factor on performance will become the database server itself. At this point, we need to understand the MongoDB architecture if we want to optimize its internal efficiency.

Mongod

In a simple MongoDB implementation, a MongoDB client sends wire protocol messages to the MongoDB daemon process *mongod*. For instance, if you install MongoDB on your laptop, a single `mongod` process will respond to all of the MongoDB wire protocol requests.

Storage Engines

A *storage engine* abstracts database storage from the underlying storage medium and format. For instance, one storage engine might store data in memory, while another might be designed to store data in cloud object stores, while a third might store data on a local disk.

MongoDB can support multiple storage engines. Initially, MongoDB shipped with a relatively simple storage engine which stored data as memory-mapped files. This storage engine was known as the *MMAP* engine.

In 2014 MongoDB acquired the *WiredTiger* storage engine. WiredTiger has many advantages over MMAP and became the default storage engine from MongoDB 3.6. We'll be focusing predominantly on WiredTiger within this book.

WiredTiger provides MongoDB with a high-performance disk access layer which includes caching, consistency, and concurrency management and other modern data access facilities.

Figure 2-2 illustrates the architecture of a simple MongoDB deployment.

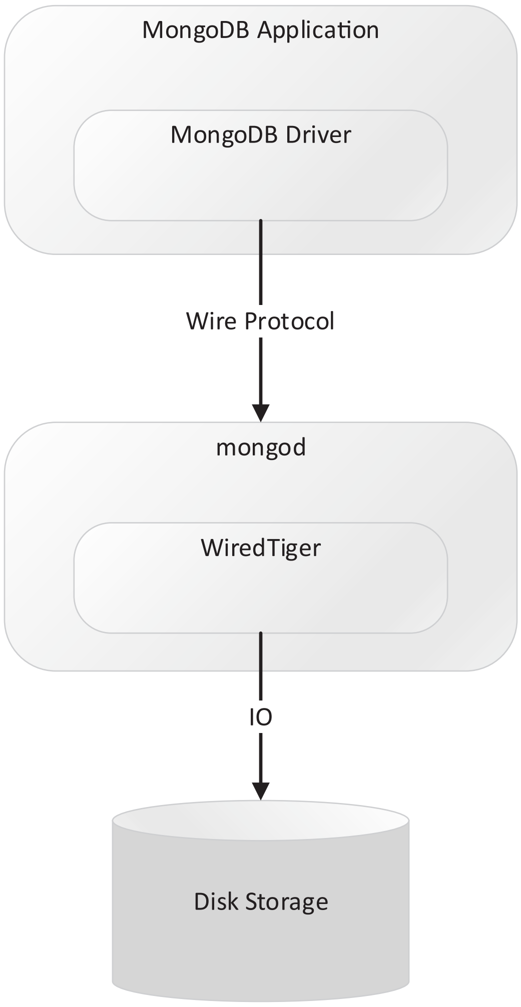


Figure 2-2. *Simple MongoDB deployment architecture*

Replica Sets

MongoDB achieves fault tolerance through the use of *replica sets*.

A replica set consists of a *primary node* together with two or more *secondary nodes*. The primary node accepts all write requests which are propagated synchronously or asynchronously to the secondary nodes.

The primary node is selected by an election involving all available nodes. To be eligible to become primary, a node must be able to contact more than half of the replica set. This approach ensures that if a network partition splits a replica set into two partitions, only one of the partitions will attempt to elect a primary. The *RAFT protocol*[1] is used to determine which node becomes the primary, with the objective of minimizing any data loss or inconsistencies following the failover.

The primary node stores information about document changes in a collection within its local database called the *Oplog*. The primary will continuously attempt to apply these changes to secondary instances.

Members within a replica set communicate frequently via *heartbeat* messages. If a primary finds it is unable to receive heartbeat messages from more than half of the secondaries, then it will renounce its primary status, and a new election will be called. Figure 2-3 illustrates a three-member replica set and shows how a network partition leads to a change of primary.

[1] https://en.wikipedia.org/wiki/Raft_(computer_science)

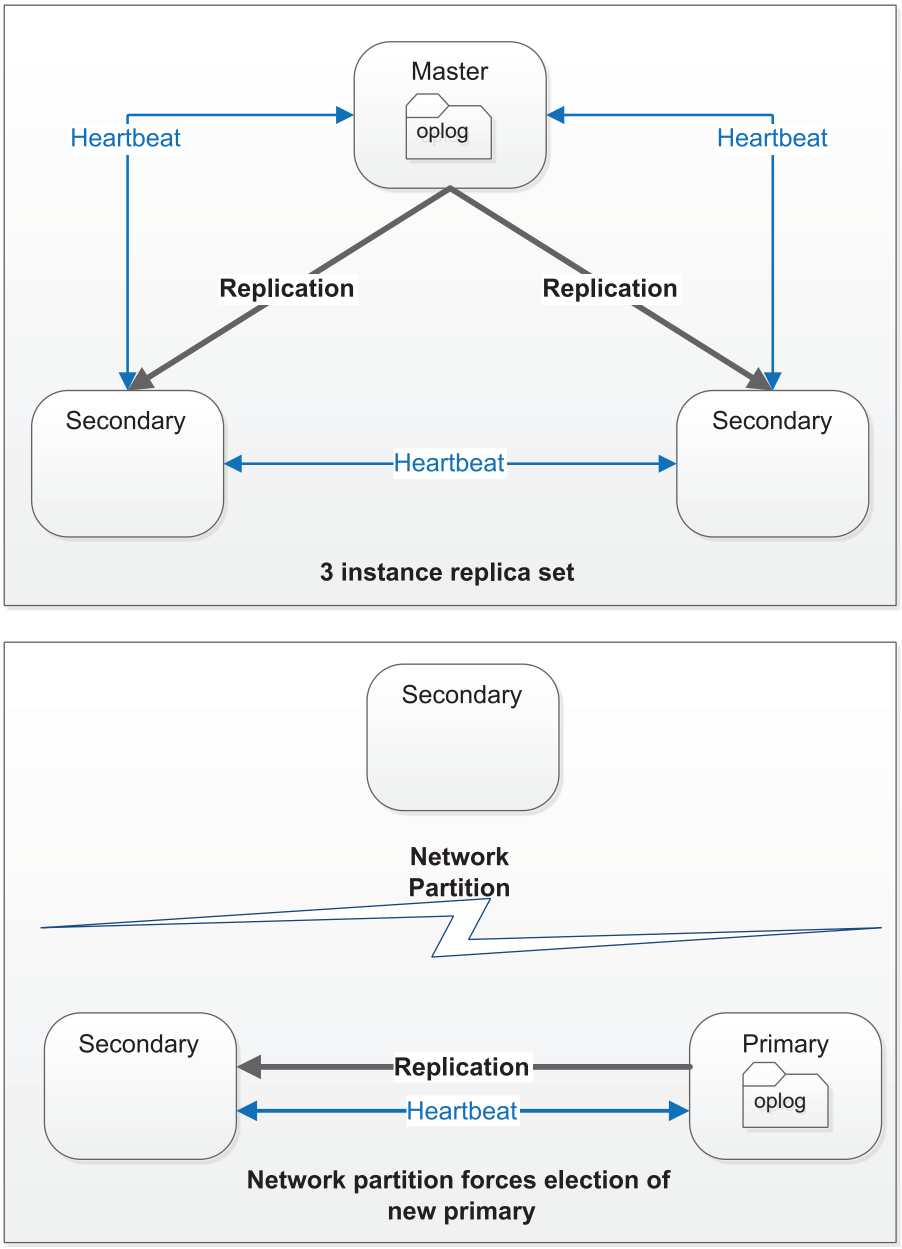


Figure 2-3. *MongoDB replica set election*

MongoDB replica sets primarily exist to support high availability – allowing a MongoDB cluster to survive a failure in an individual node. However, they can also provide performance advantages or disadvantages.

If the MongoDB *write concern* is greater than 1, then every MongoDB write operation (inserts, updates, and deletes) will need to be confirmed by more than one member of the cluster. This will result in a cluster which performs more slowly than a single node cluster. On the other hand, if the *read preference* is set to allow reads from secondary nodes, then read performance might be improved by spreading the read load across multiple servers. We'll discuss the performance impact of read preference and write concern in Chapter 13.

Sharding

While replica sets exist primarily to support high availability, MongoDB *sharding* is intended to provide scale-out capabilities. "Scaling out" allows us to increase database capacity by adding more nodes to a cluster.

In a sharded database cluster, selected collections are partitioned across multiple database instances. Each partition is referred to as a "shard." This partitioning is based on a *shard key* value; for instance, you could shard on a customer identifier, customer ZIP code or birth date. Selection of a particular shard key can have positive or negative impacts on your performance; in Chapter 14, we'll cover how to optimize shard keys. When operating on a particular document, the database determines which shard should contain the data and sends the data to the appropriate node.

A high-level representation of the MongoDB sharding architecture is shown in Figure 2-4. Each shard is implemented by a distinct MongoDB server, which in most respects is unaware of its role in the broader sharded server (1). A separate MongoDB server – the config server (2) – contains the metadata which is used to determine how data is distributed across shards. A router process (3) is responsible for routing client requests to the appropriate shard server.

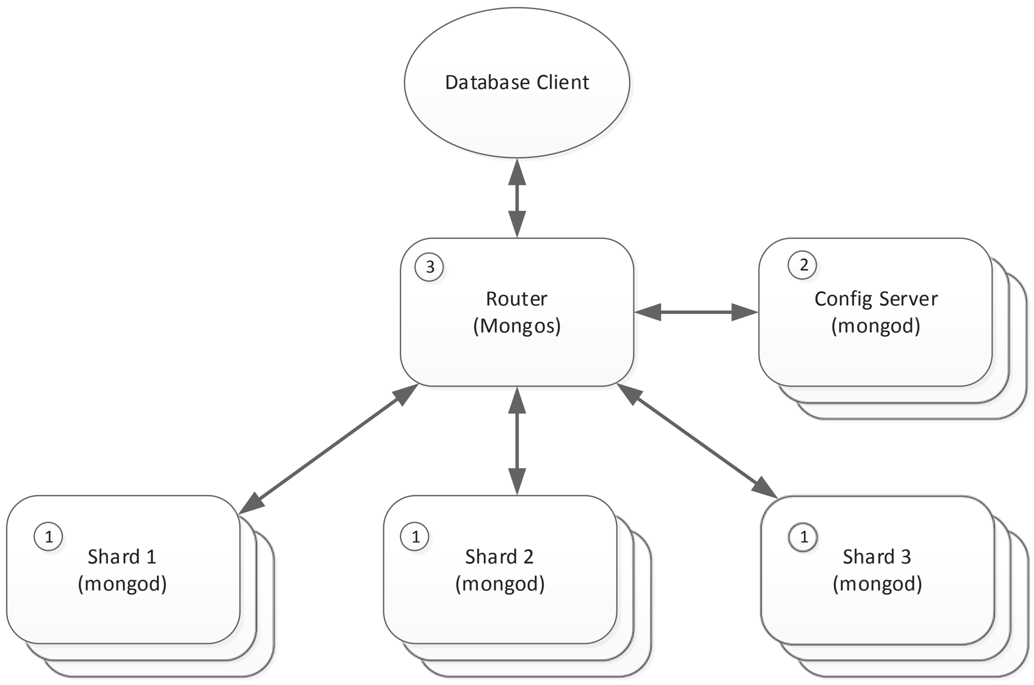


Figure 2-4. *MongoDB sharding*

To shard a collection, we choose a *shard key*, which are one or more indexed attributes that will be used to determine the distribution of documents across shards. Note that not all collections need be sharded. Traffic for unsharded collections will be directed to a single shard.

Sharding Mechanisms

Distribution of data across shards can be either *range-based* or *hash-based*. In range-based partitioning, each shard is allocated a specific range of shard key values. MongoDB consults the distribution of key values in the index to ensure that each shard is allocated approximately the same number of keys. In hash-based sharding, keys are distributed based on a hash function applied to the shard key.

See Chapter 14 for more details of range- and hash-based sharding.

Cluster Balancing

When hash-based sharding is implemented, the number of documents in each shard tends to remain balanced under most scenarios. However, in a range-based sharding configuration, it is very easy for the shards to become unbalanced, especially if the shard key is based on a continuously increasing value such as an auto-incrementing primary key ID.

For this reason, MongoDB will periodically assess the balance of shards across the cluster and perform rebalance operations if needed.

Conclusion

In this chapter, we've briefly reviewed the key architectural elements of MongoDB that are essential prerequisites for MongoDB performance tuning. Most readers will already be broadly familiar with the concepts covered in this chapter, but it's always good to be sure that you have the fundamentals of MongoDB covered.

The best place to learn more about any of these topics is the MongoDB documentation set – available online at `https://docs.mongodb.com/`.

In the next chapter, we'll deep dive into the essential tools provided with MongoDB that should be your constant companions during your tuning endeavors.