Replica Sets

This lesson introduces the theory behind replica sets, why and how they are used in MongoDB.

WE'LL COVER THE FOLLOWING

World of NoSQL

The NoSQL world has grown and blossomed in the past few years. It has given us more options in choosing databases than the past few decades did.

Now, we can think about the nature of our data and actually pick the type of database that best suits our needs. This is one of the coolest things about the NoSQL ecosystem, it contains many types of databases, and each type is best applied to different types of problems.

One of the dominant databases on the NoSQL market, at the moment, is certainly MongoDB. It is a documented NoSQL database, and thus, closer to the traditional relational databases than some other types of NoSQL databases. This probably explains its success. You can read about this and some other MongoDB basics in the previous chapter. Now, we are going to dive a little bit deeper into some of the MongoDB features regarding replication and scaling.

Introduction to Replica Sets

Introduction to Replica Sets

What is replication?

One of the reasons NoSQL databases evolved in the first place was the need for a different way of storing data in distributed systems-- one server was not good enough anymore.

Now, we have multiple servers with multiple NoSQL databases running on them. This process of *synchronizing* data across multiple servers in the whole system is called **replication**.

Why Use Replication?

The goal of replication is for all of the servers to have the same data so that the users who access different servers have the same information. This process ensures data *durability* and makes sure that no data is lost if one server fails, or if the network experiences problems for a certain amount of time.

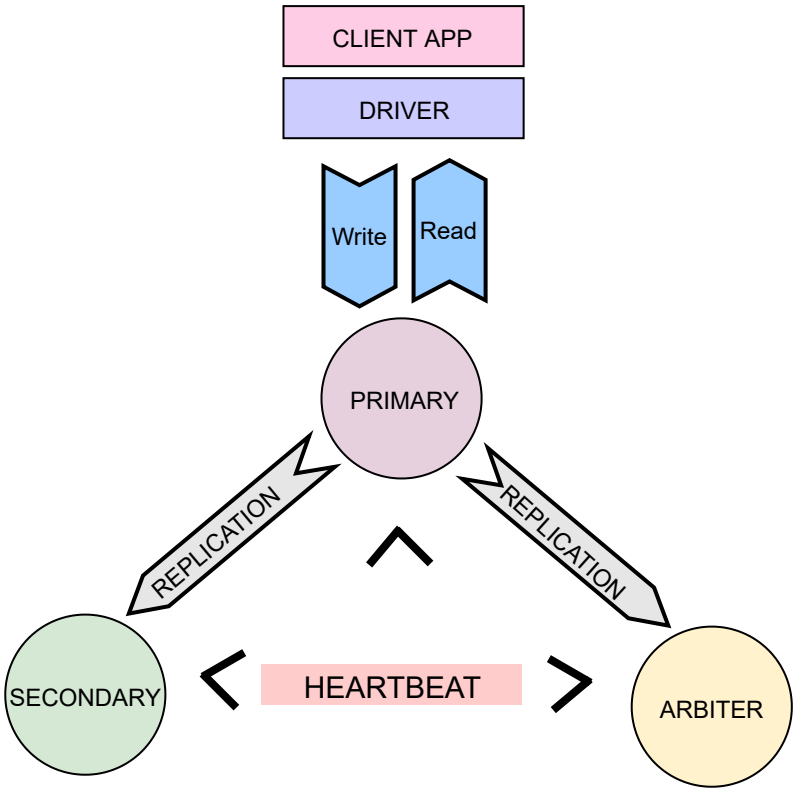
Also, this mechanism increases the general availability of data and eliminates maintenance downtime. Seems pretty cool, right?

Replication in MongoDB

In MongoDB, replication is implemented using **replica sets**. Replica sets are instances of MongoDB processes (`mongod`) that host the same data set. Each of these processes is considered as one node and that node can be a *data-bearing* node or an *arbiter node*.

Data bearing nodes, as the name implies, are nodes that host data. So a *replica set* is basically created from two or more of those nodes. Based on operations that are done on them, nodes can be either *primary* or *secondary*.

An *arbiter* node, on the other hand, does not maintain a data set nor can it be a *primary* node. The main purpose of an *arbiter* node is to add an odd number of votes in elections held to choose a primary node, and to respond to heartbeat requests by other members.



Primary Node

There can only be **one** *primary node* in a replica set, and that is the node on which all *write* operations are done. When a write operation is done on the primary, it records information about this operation in the *operation log* (*oplog*).

Secondary Nodes

Write operations never go to *secondary nodes*. Instead, they replicate the primary's *oplog* and then apply the recorded operations to their data set. This way, the secondary data set mirrors the primary one.

MongoDB client can be configured in a way that reads data from the secondary nodes.

Detecting a Failing Node

In general, these nodes maintain the heartbeat between them, as a mechanism for detecting a *failing* node. This way, when one of the nodes fails; other nodes are aware of that.

But what if a primary node fails?

In that case, *writes* cannot proceed until the remaining secondary nodes do an

In that case, *writes* cannot proceed until the remaining secondary nodes do an *election* among themselves to choose a *new* primary. The *read* queries, on the other hand, can continue to be served by the replica set as long as they are configured to run on secondary nodes while the primary is unavailable.

Elections also occur after a *replica set* is initiated. Once the *new* primary node is elected, the cluster is operational again and can receive write operations. If the old primary recovers, it will rejoin the replica set, but it will no longer be primary. Instead, it will re-sync data with the other nodes and will rejoin as a secondary node.

That's a lot of theory, but how does it look in practice? Let's find out in the next lesson!