Datenbank Implementationen

Das E-Book.

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

IT14C

April 2017

Inhaltsverzeichnis

1 Architecture

1.1 Core Processes

The MongoDB server package comes up with three main processes:

- mongod
- mongo
- mongos

mongod is the core database server. Once started, the *mongod* listens by default on port 27017 for requests. It also manages the data format and performs all background operations. For administration the *mondod* provides an HTTP interface witch can be reached at the localhost on port 28017 (1000 higher than the default port). The data directory *mongod* connects to is by default C:\data\db (or /data/db). This directory definitely has to exist and the default ports have to be free, otherwise the process fails to start (Sabharwal & Edward, 2015).
The *mongo* process is an interactive MongoDB shell. It gives the user the possibility to communicate with a running *mongod* process via a JavaScript like query language. If running on the same host it automatically connects to the *mongod* process and a preinstalled test database (Sabharwal & Edward, 2015).
The *mongos* process is working like a routing service and is the basis for MongoDBs sharding ability that will be described later. It holds the information about where the requested data is located and forwards the request from an application server to the right destination (Sabharwal & Edward, 2015).
By running a *mongod* process you already have a standalone deployment of MonogDB that can be accessed by a client. But in case of failure there is no redundancy or recovery, that prevents data loss, so its not recommended to use this in a production environment. To avoid this replication is used to guard against hardware failure or database corruption. It also gives the possibility to perform normally high-impact maintenance with little or no impact (Plugge, Hows, Membrey & Hawkins, 2015).

1.2 Replication

„MongoDB supports the replication of a database's contents to another server in real time or near real time“(Plugge et al., 2015). For that MongoDB provides two different replication methods. The traditional *Master/Slave Replication* and *Replication Sets*.

1.2.1 Master/Slave Replication

In a Master/Slave set up one *mongod instance* acts as a master the others declared as slaves. All write and read operations are requested to the master and the slaves replicate the data of the master, but can't be requested by a client. If a failure occurs that forces the master to go down, the hole system can't be reached anymore. The data, till the last replication to the slaves, is saved, but can't be accessed until the master comes up again. In MongoDB the master holds a capped collection called *oplog*. The *oplog* is an ordered history of all logical writes that are executed within a defined time period. The operations stored in the *oplog* in an idempotent way, so they can be performed multiple times without changing the result (Sabharwal & Edward, 2015). That's useful when a slave runs into failure while executing the operations onto its data. In that case the replication process can be simply restarted. If the slaves syncing process last to long or the slave was down for a longer time, the oplog data could be deleted before the slave was able to synchronize (Sabharwal & Edward, 2015). In that case the slave has to start a resync process. To avoid such a situation the oplog length should be chosen in consideration of the slaves performance.
The configuration of MongoDB with a *Master/Slave Replication* is only recommended for more than 50 nodes (Sabharwal & Edward, 2015). At that point the next described replication method the *Replica Set* is reaching its limits, because the communication overhead becomes to big.

1.2.2 Replica sets

In contrast to the *Master/Slave Replication* in a *Replica Set* no fixed master is defined. Instead the nodes are declared as primary or secondary. Each node in a *Replica Set* can become primary, but there is only one primary at a time, the others are secondaries. All write operations going through the primary, but read operations can also be performed by secondaries (Sabharwal & Edward, 2015). The replication process works just like it does in a *Master/Salve Replaction*, but if a primary goes down a new primary is elected out of the secondaries.

Communication

All nodes in a *Replica Set* communicate with each other. As life sign they are sending a heartbeat signal to each node and getting back status replies of each node. Those replies contain information about the node, such as is he primary or secondary and what type of node he is. Each node can be assigned a certain number of votes and a priority.
This results in a various types of nodes:

- **normal secondaries:** hold a copy of the primaries data, accept read requests and are primary candidates
- **priority-0-members:** secondaries that will never become primary
- **hidden-members:** priority-0-members that can't serve read request, because they are hidden for the client
- **delayed-members:** have a delay in synchronization to prevent human failure
- **arbiters:** don't hold data, they just solve ties in a election process
- **non-voting-members:** normal secondaries, but they can't vote

If the primary recognizes that the heartbeat of a secondary has stopped he has to check if he still can reach the majority of the set and if it can't he demotes itself to secondary and starts a election process. Also the election process is started if a secondary recognizes the primary is down. All voting nodes now collect the for the election required information form the primary candidates. The election of a primary depends on various parameters. Important is that the elected node has the most recent data of all nodes. The candidate with the most votes is promoted to primary. When the old primary comes up again he will be a new secondary (Sabharwal & Edward, 2015; Plugge et al., 2015).

Consitency

The replication in MongoDB works asynchronously. Writes are always routed to the primary, but reads can also requested to a secondary. „This behavior is characterized as eventual consistency, witch means that although the secondary's state is not consistent with the primary node state, it will become consistent over time“(Sabharwal & Edward, 2015). It is possible to configure the preferred reading node by primary, primaryPrefferd, secondary, secondaryPreffered or nearest (Plugge et al., 2015). The only way to ensure that a client always requests the up to date data is to set the read preference to primary. But in this situation we only get the traditional *Master/Slave* behaviour and we loose all availability features. Normally the read preference would be set to primaryPreffered, so

reads were just send to the secondaries if the primary is unavailable. Or if the system is widely distributed nearest would be chosen. To ensure a minimum number of secondaries is up to date it is possible to specify write concerns. With write concerns the client will get the success response only after the write operation was replicated to the specified number of secondaries (Sabharwal & Edward, 2015).

1.3 Sharding

If the amount of data exceeds the capacity of a single database server, partitioning is needed to distribute the data on multiple servers. For MongoDB this ability is even more important, because it uses memory mapped file I/O to access its underlining data storage(Plugge et al., 2015). For that MongoDB uses a horizontal partitioning mechanism called *sharding*.
The data collection gets divided and distributed onto multiple servers called shards. Every shard is an independent database managed by a *mongod* process. All the shards are combined to one logical database. The partitioning and routing are managed by the earlier mentioned *mongos* process. All write and read requests of an application are send to a *mongos* process, that holds the information where the requested data is stored and forwards the request to responsible *mongod* process. Also the partitioning decision is taken by the *mongos* process. The data is distributed based on a configured shard key and chunk size. The metadata of a sharded cluster is stored on special config servers, where the *mongos* processes can obtain the routing information.

1.4 Summary

Figure 1.1 describes one possible deployment architecture that contains all in this chapter mentioned artifacts. Clients can connect to a *mongos* process running on an application server. This process forwards the requests based on the information stored on the *config servers* to the right shard. A shard is an replica set containing several *mongod* processes.

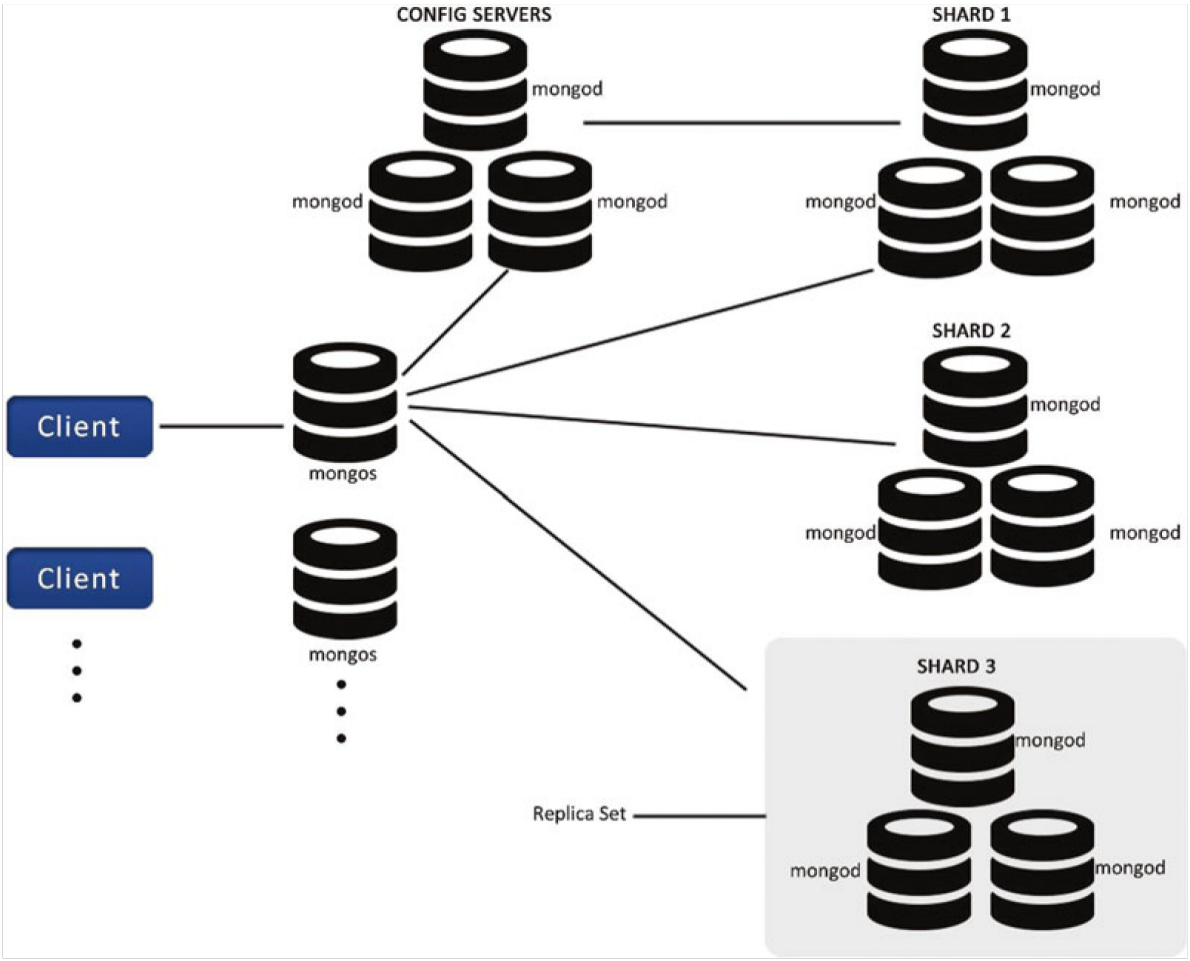

Abbildung 1.1: MongoDB Architecture Example

Literaturverzeichnis

Plugge, E., Hows, D., Membrey, P. & Hawkins, T. (2015). *The definitive guide to mongodb: A complete guide to dealing with big data using mongodb* (3rd Aufl.). Berkely, CA, USA: Apress.

Sabharwal, N. & Edward, S. G. (2015). *Practical mongodb: Architecting, developing, and administering mongodb.* Berkely, CA, USA: Apress.