# Video Script: Section #7 Replica Sets and Scaling

**Video Script: Video #1 Types of nodes**

|  |  |  |
| --- | --- | --- |
| No | Action on screen | Notes |
| 1 | Advance to section introduction slide | In this section we cover replica sets to achieve reliable and high performance deployments |
| 2 | Advance to in this section slide | Replica sets are made up of various node types. After reviewing these node types, a replica set will be built and then the failover verified. Write concern, which allows levels of data consistency, is then discussed followed by ReadPreference and how load can be distributed across nodes in a replica set. |
| 3 | Advance to Video 7.1 slide | This video focuses on a discussion of node types. |
| 4 | Advance to Replica set Concepts slide | Before discussing the node types, it’s important to understand that replica sets ensure multiple copies of data are available. Replica sets need to exist in odd number in order to ensure that a new primary can be elected. All writes go to a primary, while reads can be distributed. |
| Screens include: | | |
| 5 | Advance to types of Nodes slide | Nodes can be added to a replica set with a few purposes. Two main distinctions are regular and arbiter. Regular nodes function as secondary nodes and can take over the role of primary node in the event of a failure. In some cases, a regular node has restrictions introduced so it can fill a special purpose. One example is to create an active backup. One motivation for this is to provide a node that won’t respond to queries or be promoted to primary, but which contains a full copy of the data that can be used to create backups. Another special purpose use might be a system that is intended only to respond to queries, but not writes, so it can never become primary. |
| 6 | Advance to types of nodes table | When looking at the different node types, this table can help make decisions about system configuration. For example, a typical node will have high memory and disk requirements, plus high bandwidth to accommodate read and (potentiall) write traffic. An arbiter, on the other hand, requires minimal memory and disk and very little bandwidth. An active backup configuration may not have high memory requirements since it won’t actively be handling reads or writes. Bandwith requirements should be sufficient to replicate data from active nodes and sufficient disk space should be provided to hold all the data. |
| 7 |  |  |
| 8 |  |  |
| 12 |  |  |
| 13 | Advance to Summary Slide | Replica sets ensure multiple copies of data are available. They should exist in odd numbers so that elections for a new primary do not fail. All write traffic goes to the Primary, but read traffic can be distributed among all active nodes. Some nodes may have a special purpose or even have the role of arbiter. |
| 14 | Advance to next video slide | In the next video we will build a replica set |
|  |  |  |

**Video Script: Video #2 Build a replica set**

|  |  |  |  |
| --- | --- | --- | --- |
| No | | Action on screen | Narration |
| 1 | | Advance to video introduction slide | In this video we build a replica set. |
| 2 | | Advance to concepts slide | This process includes installing additional instances of MongoDB and subsequently adding those as nodes in the replica set. Finally all hosts are verified to show they are active. |
| Screens include: | | | |
| 3 | | Explain process for running multiple mongod processes on one machine | There is a utility on Linux called screen. The screen command makes it possible to run multiple shells from a single login. This is used throughout this tutorial to make it possible to start multiple mongod processes that will serve as nodes in the replica set.  This tutorial also makes use of the manual installation process to more easily run multiple mongod processes from a single shell. Start by downloading the latest distribution and unpackaging it in the current directory. |
| 4 | | Create data directories | Next create individual data directories for each node in the replica set, as well as a logs directory to hold all log files. |
| 5 | | Start the first mongod instance and verify | The first instance of mongod is started with this command. The combination of -d and -m tell screen to start this shell in detached mode. The -S specifies the label to give this screen, so that it’s easy to identify in the future. Next is the command that should be run in that shell. In this case, mongodb is being started with a number of commands that would normally be included in a configuration file. In this case it’s more useful to see them in the command call, so they are left as arguments to the call to mongod. Notice that the name of the replica set is given as rs0.  It is now possible to verify that the mongod process is running in a few different ways. First is with a ps aux command, which shows that mongod is running. Another way is to tail the log, which shows that the mongod process is running waiting for connections. Finally it’s possible to run screen -ls, which shows it running as a detached screen shell. |
| 6 | | Start three mongodb instances + arbiter | The remaining two nodes and the arbiter can be started quickly using similar commands. Notice that the primary difference between each command is the label given to the screen shell and the port assigned to the mongod instance. Naturally the data directory and log file name represent the different processes. |
| 7 | | Show them running in screen | The screen -ls command now shows all four instances running. |
| 8 | | Connect to the first | To setup the replica set, first connect to the first mongod instance in the mongo shell. This could be any of the nodes by changing the port argument, since they are running on distinct ports. |
| 9 | | Create rsconf document | Create a replica set configuration document as shown. This must reference the replicaset name that was identified when the mongod process was started. This was rs0.The members of the replica set are also given. Right now only a seed member is given as the current member. |
| 10 | | Initiate the replica set | This rsconf document can now be used to initiate the replica set, using this command. |
| 11 | | View configuration | The configuration can be verified by calling rs.conf. |
| 12 | | Add other nodes, including arbiter | Additional nodes can be added using the command rs.add, and providing the connection URL. The arbiter is added using rs.addArb and providing a connection URL. |
| 13 | | Show replica set status | The replica set status is shown using rs.status. Notice that… The replica set is setup and configured. |
| 14 | |  |  |
| 15 | | Advance to summary slide | Replica sets are easily setup by adding nodes, which are healthy instances of mongodb, to a configured replica set. The commands rs.conf and rs.status provide realtime details about the configuration and health of the replica set. |
| 16 | | Advance to next video slide | In the next video, failover will be verified. |
|  |  | |  |
|  |  | |  |

**Video Script: Video #3 Verify Failover**

|  |  |  |
| --- | --- | --- |
| No | Action on screen | Narration |
| 1 | Advance to video intro slide | In this video the failover of an existing replica which loses a member is verified |
| 2 | Advance to steps to add data slide | First the health of all nodes is verified. One node is then removed to simulate a system failure. Lastly confirm that if the Primary goes away, that a new primary is elected and the replica set remains responsive. |
| Screens include: | | |
| 3 | Connect to primary, show replica set status | This video assumes that a system is in the same state that it was at the end of Video 2 of this section. Use the mongo shell to connect to the first running instance of mongod. The screen command can be used to show all running instances like this. The command to connect should indicate the port 27017. It is now possible to get the current status of the replica set using rs.status() |
| 4 | Identify the primary screen | Based on the status output, the node \_\_\_\_ is the current primary. From the screen output the ID for that instance is \_\_\_\_. |
| 5 | Kill the mongoDB process | That instance of mongod can be destroyed using the screen command with the -X and -S commands and providing the ID for the instance that needs to be killed. This will simulate a server failure. |
| 6 | Show replica set status again | [discuss needing to connect to a still live instance, if the connection above was to the primary]. Showing the status again it is seen that the host previously assigned as primary now shows as down. The election occurred making node \_\_\_\_ the new primary. Note that this took only a few seconds for the system to notice the mongod instance went away and to compensate for it. |
| 7 | Confirm that writes are still possible | The fact that a new primary was elected already means that write traffic will continue to succeed, resulting in minimal downtime if any. |
| 8 | restart the MongoDB process in screen | Node \_\_\_\_ can now be started again using the screen command from before. A listing of screens shows that it is now running, and a call to ps aux|grep mongo confirms that there is a valid process. |
| 9 | Show replica set status again | The status of the replica set can now be checked again to confirm that the node is available. In now shows that the mongod process was reintegrated and that it is functioning as a secondary. |
| 10 | Show that the replica set recovers and reintegrates nodes that come back | Notice that when the mongod process was restarted, no additional intervention was required to reintegrate it into the replica set. The remaining nodes observed that it was available and the reintegration happened automatically. Depending on the volume of transactions that were missed, the node may show in a state of recovering while replication makes the node consistent with the rest of the replica set. |
| 11 |  |  |
| 12 | Advance to summary slide | In this video the replica set was examined to confirm that all nodes were active and health. One of the nodes was then removed and failover was verified. When the host was started again, it was successfully reintegrated into the replica set. |
| 13 | Advance to next video slide | The next video will cover write concern, which affects responsiveness and consistency. |
| 14 |  |  |

**Video Script: Video #4 Write concern**

|  |  |  |
| --- | --- | --- |
| No | Action on screen | Narration |
| 1 | Advance to Video intro slide | In this video, write concern is explored |
| 2 | Advance to steps slide | Write concern determines how MongoDB responds to writes. The choice of write concern levels depend on the desired level of consistency, redundancy and responsiveness. A few examples of write concern are then demonstrated. |
| Screens include: | | |
| 4 | Advance to slide about synchronicity | The concept of synchronicity is required to properly understand write concern. A synchronous process occurs in sequence. The actions must take place in a predefined order, preventing later steps from starting until earlier steps complete. Asynchronous, on the other hand, means that steps may occur in parallel. Responsiveness has to do with the user’s perception of how quickly the application responds to input. Highly synchronous application can feel less responsive because all steps must complete before the user can give another command. Asynchronous applications can feel more responsive because the previous command doesn’t have to complete before allowing the user to give another command. |
| 5 | Discuss how synchronicity maps on to write concern | Write concern tells mongodb whether steps must occur in sequence or if they must be done in parallel. For example, if data consistency (the level of assurance given that data will be exactly the same on all nodes in a replica set) is low, then write concern could lean toward asynchronous. If data consistency requirements are high then write concern will require leaning more toward synchronous behaviour. |
| 6 | Advance to slide about write concern levels | Unacknowledged is effectively asynchronous. This write concern tells the mongo client to send a write and immediately move on. The mongo client doesn’t even wait for the mongod process to respond that it received the request.  The default write concern is acknowledged. In this case, the mongo client will wait for the mongod process to respond that it received the request. However, that doesn’t mean the mongod process has done anything at all with it.  The Journaled write concern level instructs the mongod process to respond only after the write has been written to the journal (on disk) before responding to the client.  The next level, replica acknowledged, has various degrees. In this case, the mongod process is instructed to respond only after the write has been written to the primary and replicated to one or more nodes in the replica set. Tags can be associated with nodes in replica sets that may span rack, network and even datacenter boundaries to ensure the highest level of data consistency. |
| 7 | Show example queries | In this example, the document is to be written to a majority of the nodes and should timeout if this doesn’t occur in 3 seconds. |
| 8 |  | Other values for ‘w’ that are acceptable include 0, for unacknowledged, 1 for acknowledged (or a higher number to ensure that data is written to the specified number of nodes). It can also provide the name of a tag set to ensure that it is written to nodes in that tag set. |
| 9 |  |  |
| 10 | Advance to Summary slide | In this video, synchronicity was explained and applied to write concern. The various levels of write concern make it possible to balance application responsiveness with data consistency needs. An insert query was shown that required data to be written to a majority of nodes before returning. |
| 11 | Advance to Next video slide | The next video will cover ReadPreference and load distribution |
|  |  |  |
|  |  |  |
|  |  |  |

**Video Script: Video #5 ReadPreference and load distribution**

|  |  |  |
| --- | --- | --- |
| No | Action on screen | Narration |
| 1 | Advance to video intro slide | This video looks at ReadPreference, which makes it possible for MongoDB to distribute data between nodes in sophisticated ways. |
| 2 | Advance to secure intro slide | Proper use of ReadPreference requires understanding consistency and speed needs in order to identify data priorities. Finally a demonstration of ReadPreference is given. |
| Screens include: | | |
| 3 | Prerequisites | This section makes use of PHP and Python to show ReadPreference functionality. Additional details about using those languages with MongoDB is available in Section 6, videos 4 and 5. |
| 4 | Explain read preference | ReadPreference makes it possible to suggest to MongoDB how it should route read requests among nodes in a replica set. |
| 5 | Advance to readpreference modes slide | ReadPreference modes include primary, primarypreferred, secondary, secondarypreferred and nearest. If an application has high data consistency requirements, choosing a read preference of primary can reduce the risk of getting an inconsistent read, since all data is written first to the primary but replication to secondary nodes may lag. primaryPreferred allows an application to send a request to secondaries if the primary is not responsive enough. Secondary requires read requests to go to secondaries, which makes sense in scenarios where the primary needs to be 100% available for writes. secondarypreferred will still allow reads to go to the primary, but a secondary is top of the list.  Nearest nodes are determined based on latency. Depending on network, current load and other factors, this list may change and has no direct relationship to geographic or physical proximity. |
| 6 | Distributes reads | Nearest functions the most like a smart load balancer in that it continually tracks ping time to other nodes so that requests can be routed to the lowest latency nodes. |
| 7 | Provides some transaction throughput increase | It is possible to tune transaction throughput, for both reads and writes, by choosing an appropriate readpreference.. |
| 8 | Sharding preferred for scaling reads (counter indications) | While readpreference and the distribution of reads across multiple nodes does increase throughput somewhat, the preferred mechanism for scaling to very large datasets or for high volumes of transactions is sharding, which will be covered in the next section. |
| 9 | Show PHP example. | In PHP, the readpreference can be set on the instance of MongoClient, as shown here. After setting this once, all subsequent requests will be routed to the nearest node for processing, which may include the primary. |
| 10 | Show Python example. | In Python, the readpreference can be incorporated into the instantiation of the MongoClient object, as shown here. In this case, all read requests will now be routed to a secondary. |
| 11 |  |  |
| 12 |  |  |
| 13 | Advance to summary slide | This video covered ReadPreference as it related to data consistency and distribution of read requests among nodes in a replica set. This makes it possible to set data priorities. It concluded with two examples in PHP and Python to use readpreference. |
| 14 | Advance to section summary slide | This section started with a discussion of the types of nodes in a replica set. It then went on to build a replica set and verify that failover and recover functioned as expected. Write concern and readpreference make it possible to determine various aspects of application responsiveness and address data consistency needs. |
| 15 | Advance to next section slide | The next section covers some advanced topics for MongoDB |
| 16 |  |  |
|  |  |  |