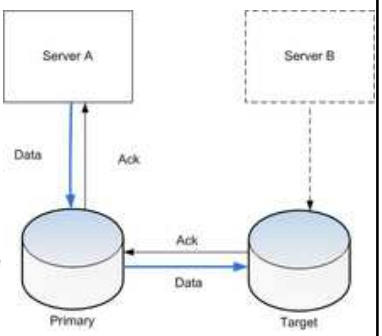
Replication

- What Is Replication
- Replication vs. Distributed Transactions
- MongoDB Replication
- Replica Set Fundamental Concepts
- Replica Set Failover and Recovery
- Replica Set in Multiple Data Centers
- Demo: Set up a Master/Slave Replication
- Demo: Set up a two-node Replica Set
- Summary
- Lab#4 Three-node Replication

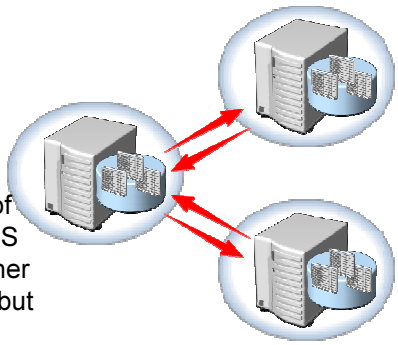
❑ What Is Replication

- **Replication** in computing involves sharing information so as to ensure ***consistency*** between ***redundant*** resources, such as software or hardware components, to improve **reliability**, **availability**, **fault-tolerance**, or **accessibility**.
  - **Active replication:** processing the same request at every replica.
  - **Passive replication:** processing each request on a single replica and then transferring its resultant state to the other replicas.
- **Disk storage replication**
  - **Synchronous replication:** guarantees "zero data loss" by the means of atomic write operation - write either completes on both sides or not at all.
  - **Asynchronous replication:** write is considered complete as soon as local storage acknowledges it. Remote storage is updated, but probably with a small lag.



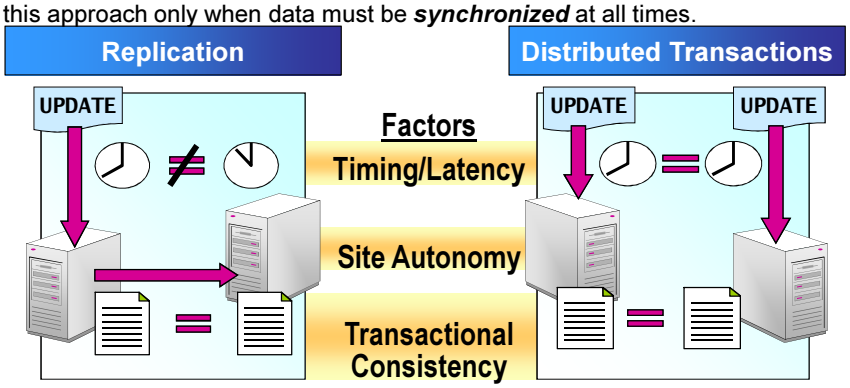
❑ What Is Replication?

- Replication is the mechanism for creating and maintaining **multiple copies** of the same data.
- Data is distributed in a loosely coupled fashion that gives sites more autonomy and ensures that updates are replicated within ***an acceptable timeframe***.
- **Benefits of replication**
  - Brings data closer to geographically distributed users.
  - Allows for autonomous sites that do not have to be continually connected.
  - Allows you to maintain separate online transaction processing (OLTP) and decision support system (DSS) copies of the data. By their nature, OLTP and DSS will cause locking conflicts with each other that can be resolved by using separate but replicated copies of the data.



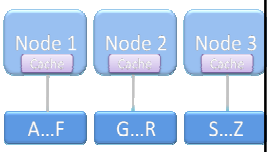
❑ Replication vs. Distributed Transactions

- **Replication:** Duplicates and distributes **recent copies** of data from a source database to a destination database, usually on a separate server. Autonomous sites are supported, allowing more scalability because sites ***can be online intermittently***.
- **Distributed Transaction:** **Guarantee** that all copies of your data have **the same values at the same time**. Each server that is included in a distributed transaction **must be online** and able to ***commit*** its part of the transaction. The failure of a transaction to commit successfully at any site, for any reason, means a failure to commit at all sites. This approach is less scalable than replication. You should use this approach only when data must be ***synchronized*** at all times.

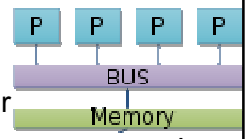


Shared-Nothing vs. Shared-Everything vs. Shared-Something

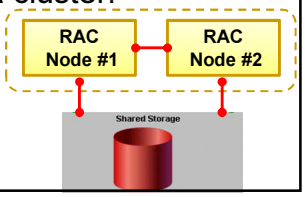
- **Shared-nothing architecture** is a distributed computing architecture in which each node is independent and self-sufficient.
  - None of the nodes share memory or storage.
  - Ex: SQL Server Cluster, MongoDB



- **Shared-everything architecture** is communal memory bus, shared I/O channel, etc.
  - Ex: Symmetric multiprocessing (SMP) cluster, Non-Uniform Memory Access (NUMA) cluster



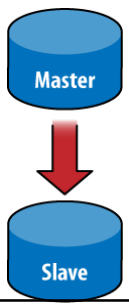
- **Shared-something architecture** is shared data access and some application-level memory sharing in a cluster.
  - Ex: Oracle Real Application Clusters (RAC)



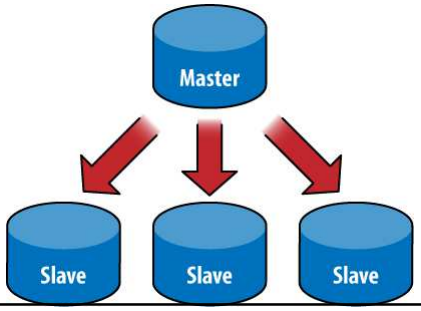
❑ MongoDB Replication

- **Master-slave** replication is the most general replication mode supported by MongoDB.
- This mode is very flexible and can be used for backup, failover, read scaling, and more.
- Deprecated since version 1.6: ***Replica sets*** replace ***master-slave*** replication. Use replica sets rather than master-slave replication for all new production deployments.

A master with one slave node
V 1.6

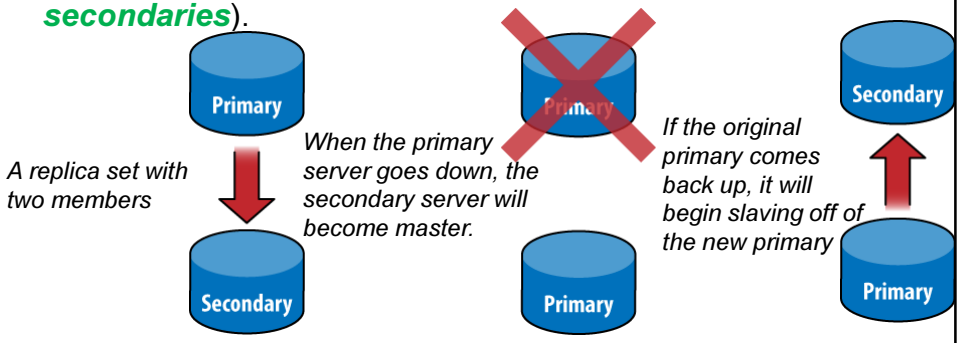


A master with multiple slave nodes
V 2+ / 3.x



Replica Sets

- A *replica set* is a master-slave cluster with **automatic failover**.
- The biggest difference between a master-slave cluster and a **replica set** is that a replica set does not have **a single master**: one is elected by the cluster and may change to another node if the current master goes down.
- They look very similar: a replica set always has a single master node (called a ***primary***) and one or more slaves (called ***secondaries***).



❑ Replica Set Fundamental Concepts

- A MongoDB ***replica set*** is a cluster of **mongod** instances that replicate amongst one another and ensure **automated failover**.
- Most replica sets consists of two or more **mongod** instances with at most one of these designated as the **primary** and the rest as **secondary** members.
- Clients direct all **writes to the primary**, while the secondary members replicate from the primary **asynchronously**.
- Database replication with MongoDB adds redundancy, helps to ensure high availability, simplifies certain administrative tasks such as backups, and may increase read capacity.
- Most **production** deployments use replication.
- MongoDB's replica sets provide automated failover. If a *primary* fails, the remaining members will automatically try to elect a new primary.
- In **MongoDB 3.x, replica sets can have up to 50 nodes**. Previous versions limited the maximum number of replica set members to 12.

Replication Functionality and Usage

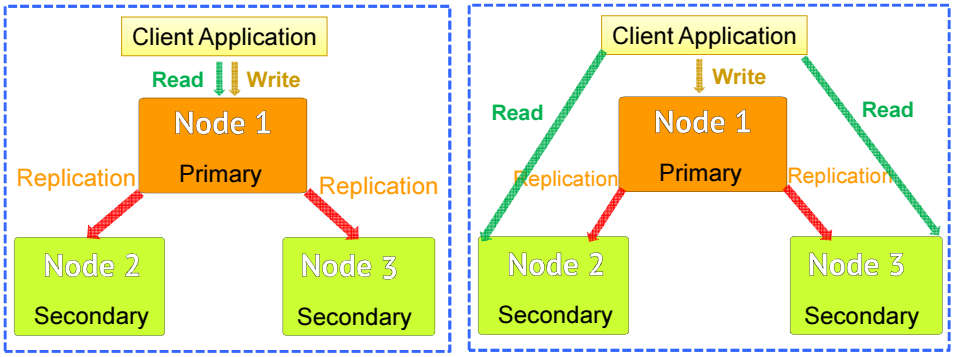
- Use of MongoDB's replication functionality is always recommended in **production** settings, especially since the current storage engine does not provide single-server durability.
- MongoDB supports **asynchronous replication** of data between servers for failover and redundancy.
- Only one server (in the set/shard) is active for **writes** (the **primary**, or **master**) at a given time – this is to allow **strong consistent** (atomic) operations.
- One can optionally send read operations to the **secondaries** when ***eventual consistency*** semantics are acceptable.
- Replicas can be used for:
  - Automated Failover / High Availability
  - Data Redundancy / Disaster recovery
  - Scaling out reads / Distributing read load
  - Taking hot backups
  - Offline batch processing

Application Concerns

Client applications are indifferent to the configuration and operation of replica sets. There are two major concepts that *are* important to consider when working with replica sets:

- **Write Concern** sends a MongoDB client a response from the server to confirm successful write operations. In replica sets you can configure ***replica acknowledged*** write concern to ensure that **secondary members** of the set have replicated operations before the write returns.
  - Specifies whether a write operation has succeeded. Write concern allows your application to detect insertion errors or unavailable **mongod instances**.
  - Create or modify data in the MongoDB instance is a write operation. Write operations target a single document at **collection level**.
- ***Read Preference:*** By default, drivers direct all reads to ***primary* for *strict consistency***. However, you may also direct reads to **secondaries** for ***eventually consistent*** reads.

Strong Consistency vs. Eventual Consistency



Strong Consistency

- *No stale reads*
- *Higher read latency*
- *Lower read throughput*

Eventual Consistency

- *Stale reads possible*
- *Lowest read latency*
- *Highest read throughput*

Since a consistent read may incur higher latency and lower read throughput it should only be used when an application scenario mandates returning the very latest value of a document. For all other scenarios an eventually consistent read will yield the best performance.

Read Preference

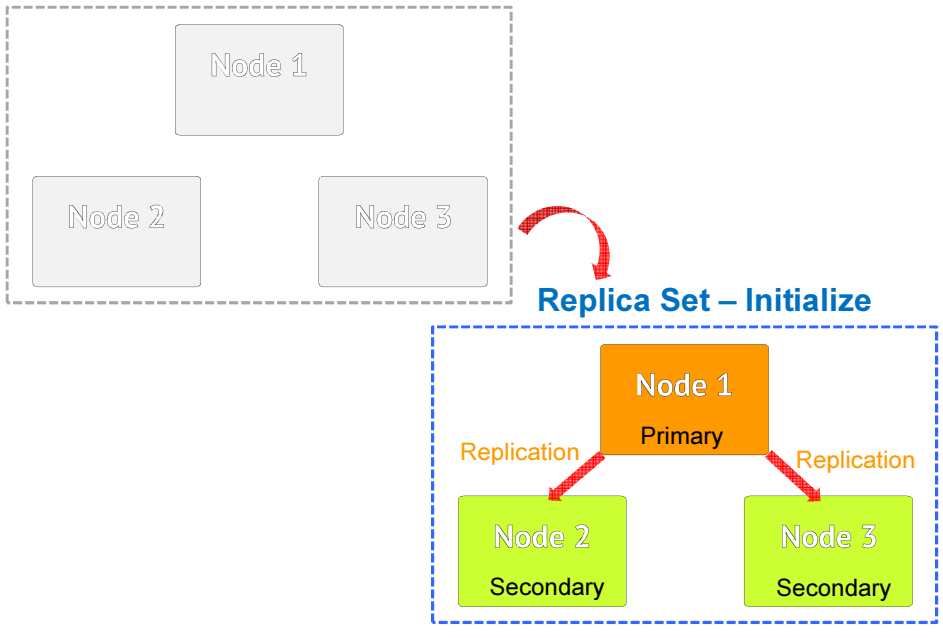
- Read preference describes how MongoDB clients route read operations to members of a ***replica set***.
- **By default**, an application directs its read operations to the ***primary*** member in a ***replica set***. Reading from the primary guarantees that read operations reflect the latest version of a document.
- Some applications may not require fully up-to-date data, you can improve read throughput, or reduce latency, by distributing some or all reads to **secondary members** of the replica set.
  - Running systems operations that do not affect the front-end application, operations such as backups and reports.
  - Providing low-latency queries for geographically distributed deployments. If one secondary is closer to an application server than the primary, better performance can be expected if you use secondary reads.
  - In ***failover*** situations where a set has *no* primary for 10 seconds or more. You can give the application the **primaryPreferred** read preference to prevent the application from performing reads if the set has no primary.

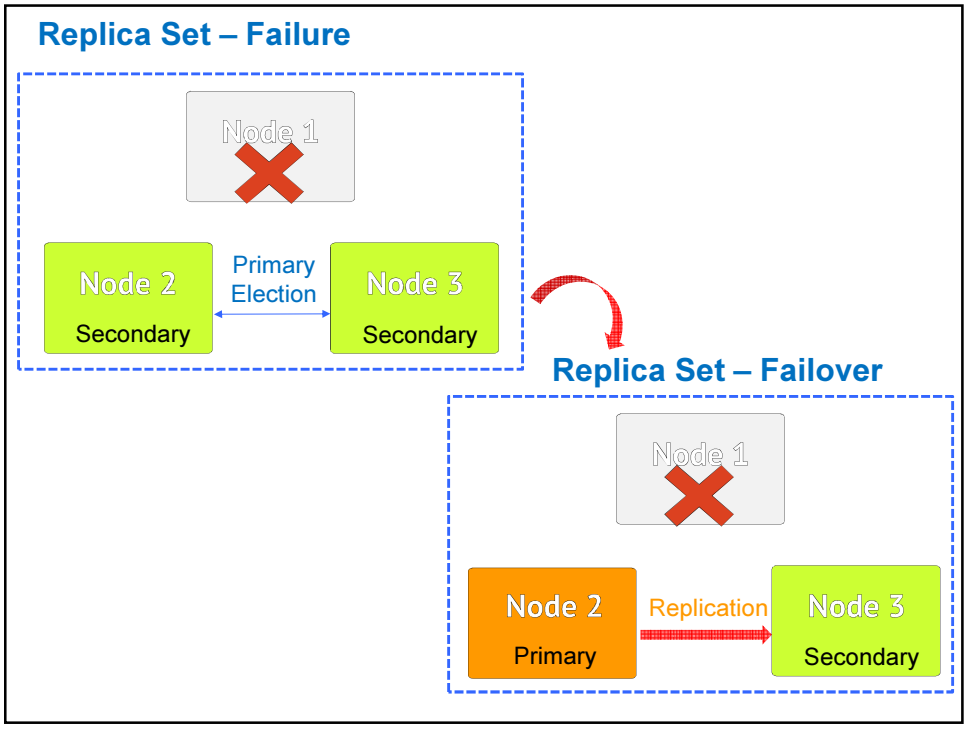
Read Preference Modes

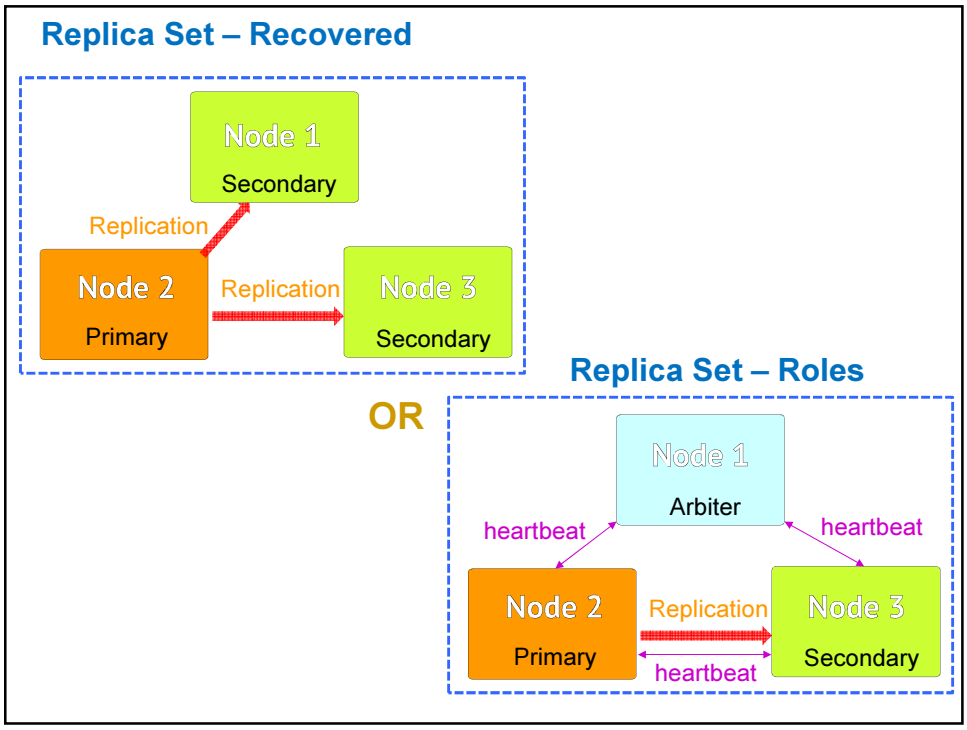
You can specify a read preference mode on connection objects, database object, collection object, or per-operation.

- **Primary** *(default):* All read operations use only the current replica set *primary*. If the primary is unavailable, read operations produce an error or throw an exception.
- **PrimaryPreferred**: If the primary is unavailable, as is the case during ***failover*** situations, operations read from secondary members.
- **Secondary**: Operations read *only* from the *secondary* members of the set. If no secondary is available, then this read operation produces an error or exception.
- **SecondaryPreferred**: If the set consists of a single *primary* (and no other members), the read operation will use the set's primary.
- **Nearest**: Reads in nearest mode may read from either primary or secondary.

❑ Replica Set – Creation




Replica Set – Failure
Node 1
Node 2
Secondary
Primary
Election
Node 3
Secondary
Replica Set – Failover
Node 1
Node 2
Primary
Replication
Node 3
Secondary


Replica Set – Recovered
Node 1
Secondary
Replication
Node 2
Primary
Replication
Node 3
Secondary
OR
Replica Set – Roles
Node 1
Arbiter
heartbeat
heartbeat
Node 2
Primary
Replication
heartbeat
Node 3
Secondary

Replica Set Member Types and Configuration Properties

- **Primary:** is the current ***master*** instance, which receives all write operations.
- **Secondary:** is the current ***slave*** instances that replicate the contents of the master database.
- **Arbiters**: These members have no data and exist solely to participate in ***elections***.
- Secondary members may handle read requests, but only the ***primary*** members can handle **write** operations.

Members of a replica set have the default proprieties.

- **Secondary-Only**: These members have data but cannot become primary under any circumstance.
- **Hidden**: These members are invisible to client applications.
- **Delayed**: These members apply operations from the primary's **oplog** after a specified delay. Oplog stores writes chronologically in MongoDB.
- **Non-Voting**: These members do not vote in elections.

Data Center Awareness

- **One primary data center, one disaster recovery site:** Set members at the main data center (sf) are eligible to become primary. In addition we have a member at a remote site that is never primary.

```
{ _id: 'myset',
    members: [ { _id:0, host: 'sf1', priority: 1 },
               { _id:1, host: 'sf2', priority: 1 },
               { _id:2, host: 'ny1', priority: 0 } ] }
```

- **Multi-site with local reads:** One set member in each of three data centers. At election time, **any** healthy up-to-date node, arbitrarily, **can become primary**. The others are then secondaries and can service queries locally if the client uses **slaveOk** read preference modes.

```
{ _id: 'myset',
    members: [ { _id:0, host: 'sf1', priority: 1 },
               { _id:1, host: 'sf2', priority: 1 },
               { _id:2, host: 'ny1', priority: 1 } ] }
```

Confirming propagation of writes with getLastError

- The **getLastError** command returns the error status of the last operation on the *current connection*.
- By default MongoDB does not provide a response to confirm the success or failure of a write operation, clients typically use **getLastError** in combination with write operations to ensure that the write succeeds.
- **w:** the number of servers to replicate to before returning.
  - A w value of **1** indicates the primary only.
  - A w value of **2** includes the primary and one secondary.
  - Set w to **majority** to indicate that the command should wait until the latest write propagates to a majority of replica set members.
- For example, if you had a three-member replica set, calling db.runCommand({getLastError: 1, **w: "majority"**}) would make sure the last write was propagated to at least 2 servers.

Tagging

Tagging gives you fine-grained control over where data is written.

- **Customizable:** you can express your architecture in terms of machines, racks, data centers, PDUs, IP addresses, etc.
- **Developer/DBA-friendly:** developers do not need to know about where servers are or changes in architecture.

- Each member of a replica set can be tagged with one or more physical or logical locations, e.g., {dc: "ny", rack: "rk1", ip: "192.168", server: "192.168.4.11"}. *Modes* can be defined that combine these tags into targets for **getLastError**'s ***w*** option.

```
{ _id : "MyDCSet", members : [
          {_id: 0, host: "A", tags: {dc: "ny"}},
          {_id: 1, host: "B", tags: {dc: "ny"}},
          {_id: 2, host: "C", tags: {dc: "sf"}},
          {_id: 3, host: "D", tags: {dc: "sf"}},
          {_id: 4, host: "E", tags: {dc: "cloud"}} ]
settings : { getLastErrorModes :
                    { veryImportant: {dc: 3},
                      someImportant: {dc: 2} } } }
```

Tagging Example

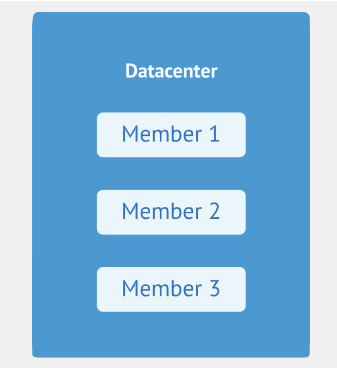
- When a developer calls **getLastError**, they can use any of the modes declared to ensure writes are propagated to the desired locations:

```
> db.foo.insert({x:1})
> db.runCommand({getLastError: 1, w: "verylmportant"})
```

- "**verylmportant**" makes sure that the write has made it to at least **3** tagged "regions", in this case, "ny", "sf", and "cloud". Once the write has been replicated to these regions, getLastError will return success. For example, if the write was present on *A*, *D*, and *E*, that would be a success condition.
- If "**somelmportant**" was used instead, **getLastError** would return success once the write had made it to two out of the three possible regions. Thus, *A* and *C* having the write or *D* and *E* having the write would both be "success." If *C* and *D* had the write, getLastError would continue waiting until a server in another region also had the write.

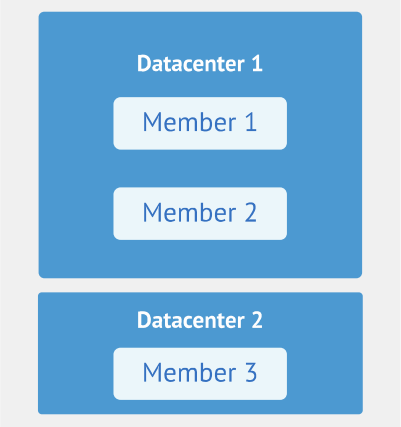
❑ Replica Set – 1 Data Center

- Single datacenter
- Single switch & power
- Points of failure:
  - Power
  - Network
  - Data center
  - Two node failure
- Automatic recovery of single node crash



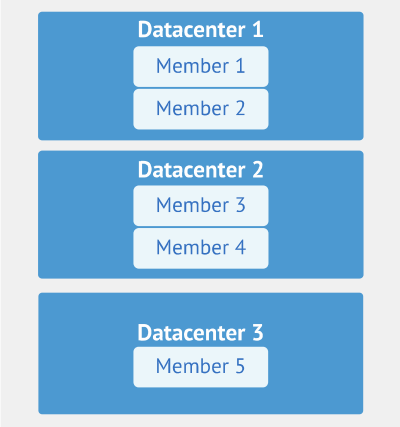
Replica Set – 2 Data Centers

- Multi data center
- DR node for safety
- Can't do multi data center durable write safely since only 1 node in distant DC



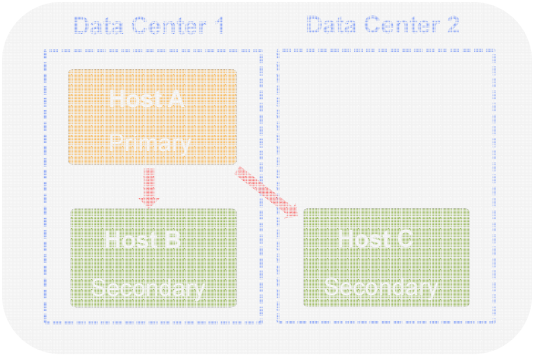
Replica Set – 3 Data Centers

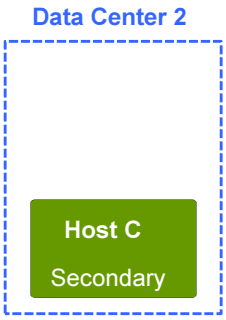
- Three data centers
- Can survive full data center loss
- Can do w= { dc: 2 } to guarantee write in 2 data centers (with tags)



Two Data Centers Scenario

- If hosts A & B crash, then the following will happen:
  - Host C will detect that the other two hosts have crashed
  - Host C will stay in secondary mode. This means that the cluster will be in ***read-only mode***.





Note 1: Assume Host C is designed for DR only and it will never become a Primary.

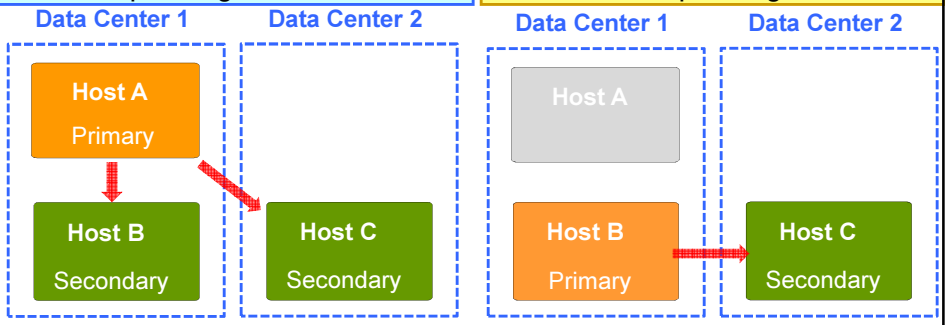
Note 2: Assume Host C is not designed for DR. However, it doesn't meet the majority requirement to become a primary.

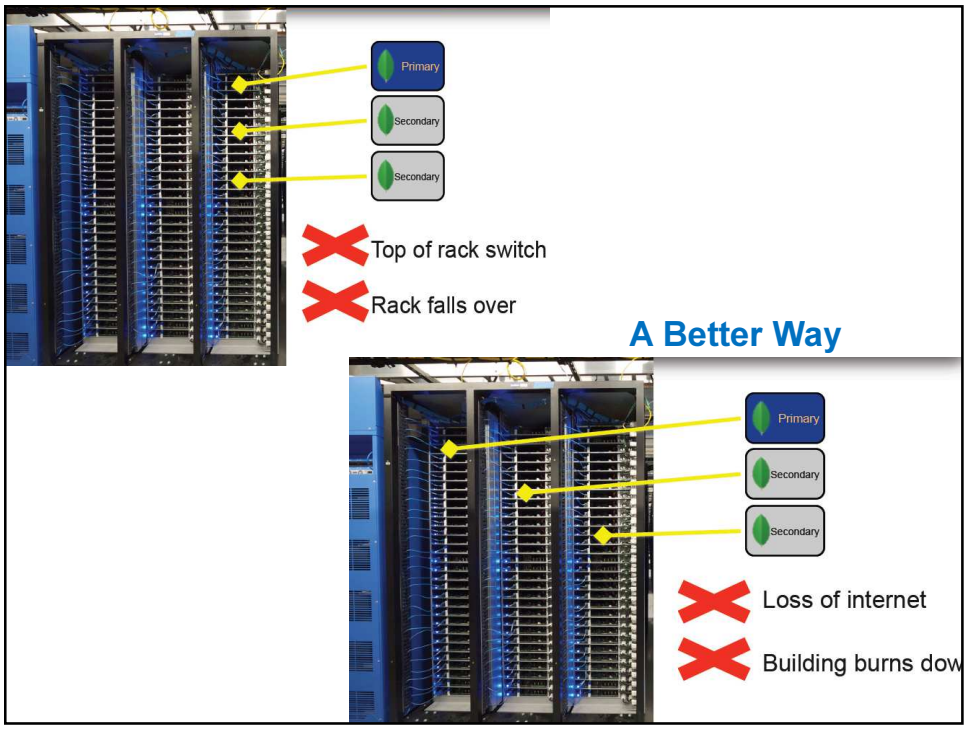
Two Data Centers Scenario

At some later time, host A (the old primary) comes back up. At this point, there are two voting hosts, and they can vote a majority, so the cluster can designate a new primary node and start receiving writes again.

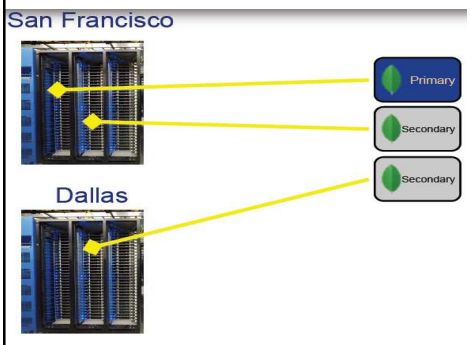
If there are writes that have been sent to host A but were <u>not</u> yet sent to host B, then the Replica Set code will detect that, and host A will become primary again. Once it does, host B will start replicating from host A.

If host B comes back up, then host B and C will compare and see who has the most recent writes: the one with the **<u>most recent writes</u>** will become the **primary** and the other one will start replicating from it.

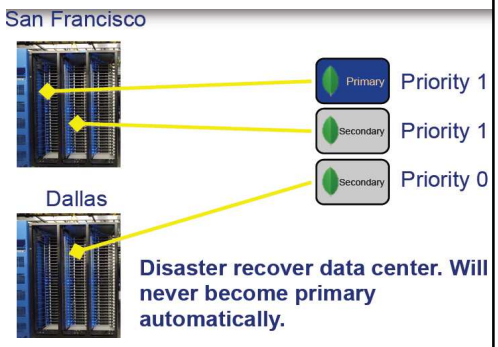




Two Data Center

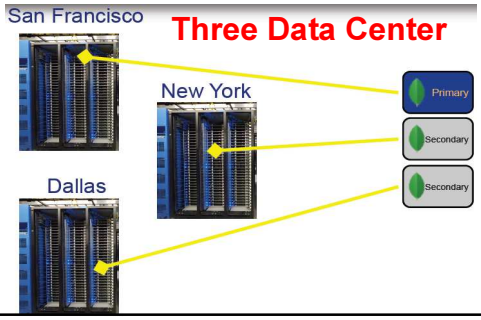


Set Priorities – Even Better



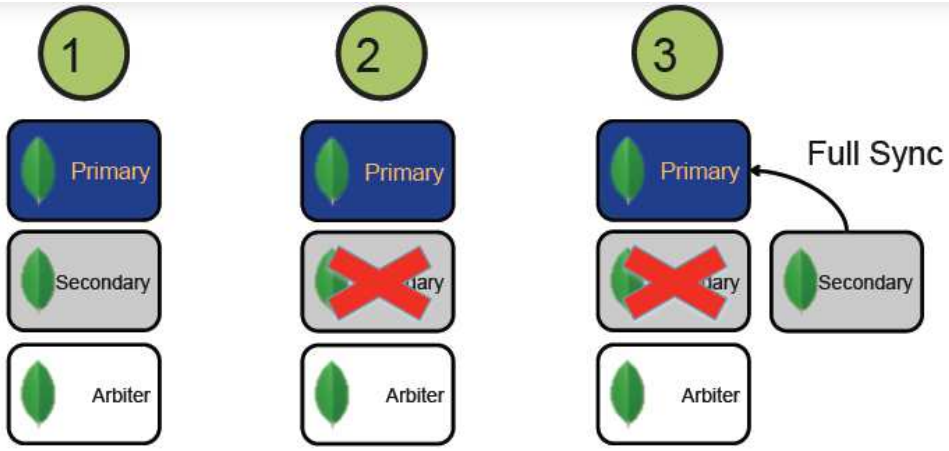
Disaster recover data center. Will never become primary automatically.

Three Data Center



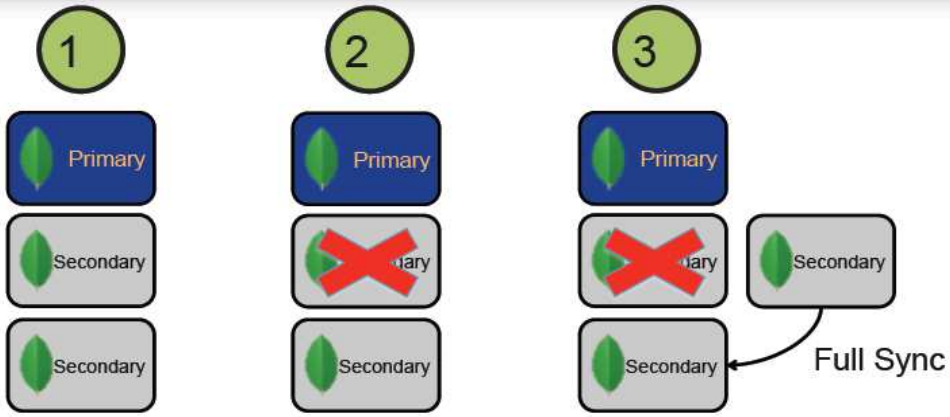
Fast Recovery

- *Two Replica + Arbiter*



Full Sync is going to use a lot of resources on the primary. So It may have degraded performance during downtime.

With 3 Replicas



Sync can happen from secondary, which will not impact traffic on Primary.

Nodes in a Replica Set

- ***Standard:*** A regular replica set node. It stores a full copy of the data being replicated, takes part in voting when a new primary is being elected, and is **capable of becoming the primary node** in the set.
- ***Passive:*** Passive nodes store a full copy of the data and participate in voting but will never become the **primary node** for the set.
- ***Arbiter:*** An arbiter node participates **only in voting**; it does not receive any of the data being replicated and cannot become the **primary node**.
- The difference between a standard node and a passive node is actually more of a sliding scale; each *participating node* (nonarbiter) has **a *priority* setting**.
  - A node with priority **0** is **passive** and will never be selected as primary.
  - Nodes with **nonzero** priority will be selected in order of decreasing priority, using freshness of data to break ties between nodes with the same priority.
  - In a set with two priority 1 nodes and a priority 0.5 node, the third node will be elected primary only if neither of the priority 1 nodes are available.

Example of modifying member priorities

- Member 0 to a priority of 0 so that it can never become ***primary***.
- Member 1 to a priority of 0.5, which makes it less likely to become primary than other members but doesn't prohibit the possibility.
- Member 2 to a priority of 1, which is the default value. Member 2 becomes primary if no member with a *higher* priority is eligible.
- Member 3 to a priority of 2. Member 3 becomes primary, if eligible, under most circumstances.

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[1].priority = 0.5
cfg.members[2].priority = 1
cfg.members[3].priority = 2
rs.reconfig(cfg)
```

Configuration Options

```
> conf = {
   _id : "mySet",
   members : [
      {_id: 0, host: "A", priority: 3},
      {_id: 1, host: "B", priority: 2},
      {_id: 2, host: "C"},
      {_id: 3, host: "D", hidden: true},
      {_id: 4, host: "E", hidden: true, slaveDelay: 3600}
       ]
   }

> rs.initiate(conf)
```

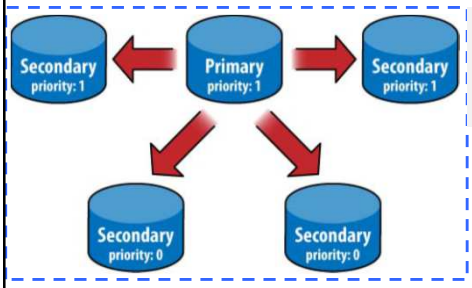
Primary DC

Secondary DC
Default priority = 1

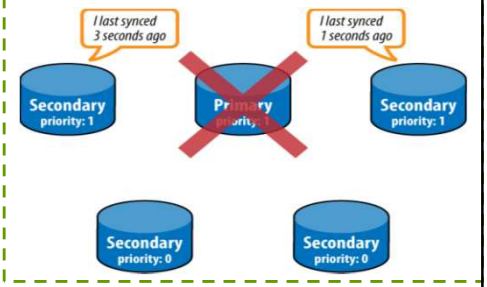
Reporting node

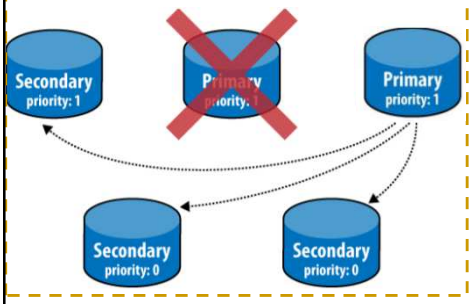
Backup node

1. A replica set can have several servers of different priority levels



2. If the primary goes down, the highest-priority servers will compare how up-to-date they are.



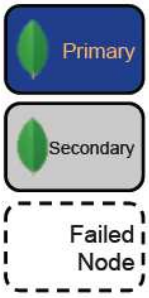


3. The **highest-priority most-up-to-date server** will become the new primary

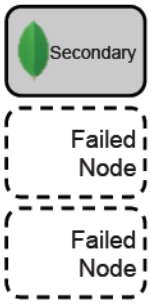
Failover

- As long as a replica set can see a majority (>50%) of the cluster, then it will elect a primary.

Simple Failure



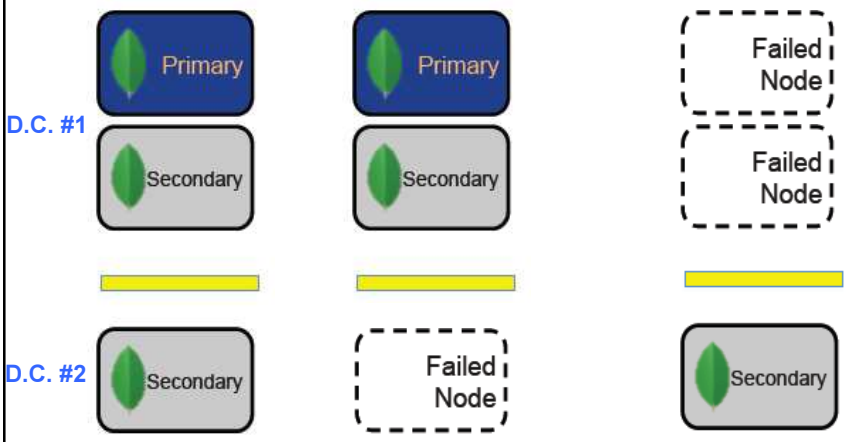
- 66% of cluster visible.
- Primary is elected



- 33% of cluster visible.
- Read only mode.

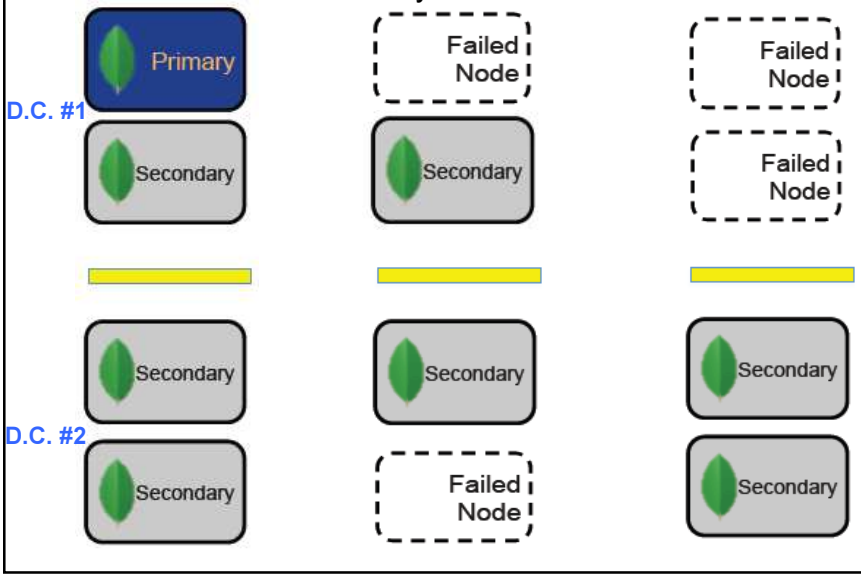
Network Partition

- 66% of cluster visible.
- Primary is elected
- 33% of cluster visible.
- Read only mode.



Even Cluster Size

- 50% of cluster visible.
- Read only mode.
- 50% of cluster visible.
- Read only mode.



❑ Replication Pair (Master/Slave) Setup

- The basic setup is to start a master node and one or more slave nodes, each of which knows the address of the master.
  - To start the master, run **mongod --master**
  - To start a slave, run **mongod --slave --source *master_address***, *where master_address is the address* of the master node that was just started.
- It is simple to try this on a single machine, although in production you would use multiple servers.
- Create two directories: C:\mongodb3.2\master & C:\mongodb3.2\slave
- For the master, create a directory to store data and choose a port: C:\mongodb3.2\master

```
mongod --dbpath master --port 10000 --master --rest
```

- For the slave, choosing a different data directory and port. You also need to tell it who its master is with the **--source** option:

C:\mongodb\slave

```
> mongod --dbpath slave --port 10001 --slave --rest --source localhost:10000
```

Replication setup (Mac)

- The basic setup is to start a master node and one or more slave nodes, each of which knows the address of the master.
  - To start the master, run mongod --master.
  - To start a slave, run mongod --slave --source *master_address, where master_address is the address* of the master node that was just started.
- It is simple to try this on a single machine, although in production you would use multiple servers.
- For the master, create a directory to store data and choose a port:

```
$ mkdir -p ~/dbs/master
$ ./mongod --dbpath ~/dbs/master --port 10000 --master
```

- For the slave, choosing a different data directory and port. You also need to tell it who its master is with the --source option:

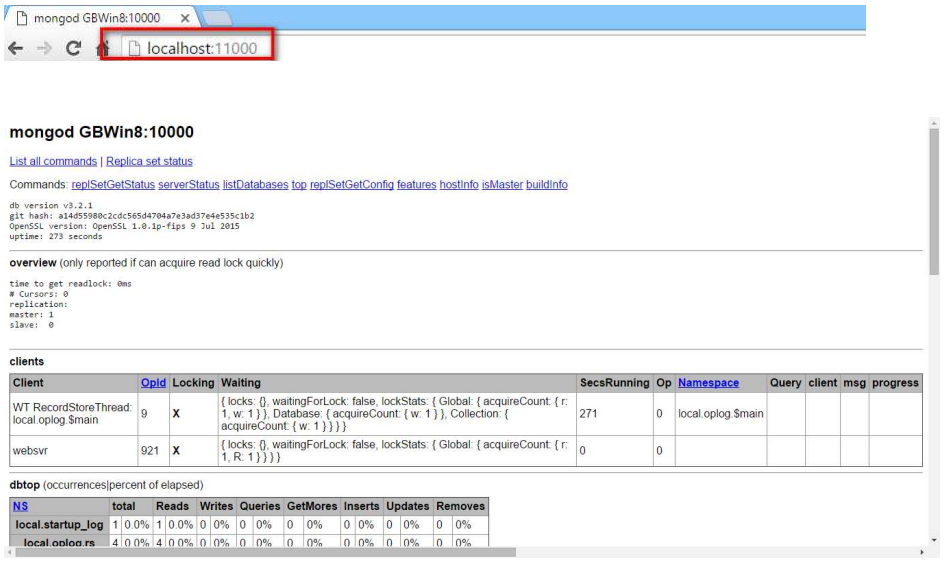
```
$ mkdir -p ~/dbs/slave
$ ./mongod --dbpath ~/dbs/slave --port 10001 --slave --source
localhost:10000
```

Master

mongod --dbpath master --port 10000 –master –rest --httpinterface

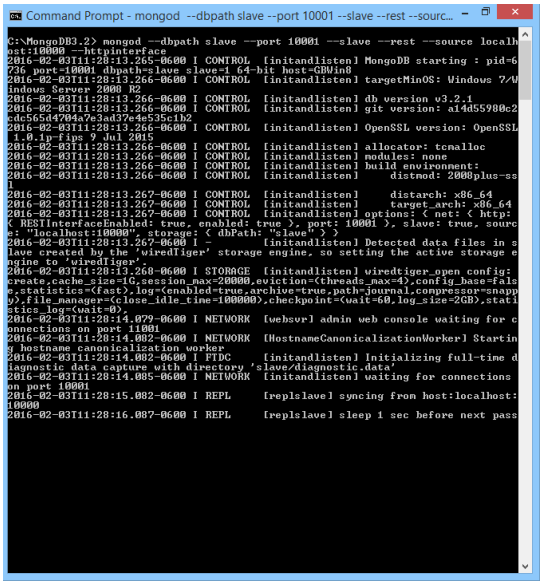
```
C:\MongoDB3.2> mongod --dbpath master --port 10000 --master --rest --httpinterfa
ce
2016-02-03T11:17:19.305-0600 I CONTROL  [initandlisten] MongoDB starting : pid=6
244 port=10000 dbpath=master master=1 64-bit host=GBWin8
2016-02-03T11:17:19.306-0600 I CONTROL  [initandlisten] targetMinOS: Windows 7/W
indows Server 2008 R2
2016-02-03T11:17:19.306-0600 I CONTROL  [initandlisten] db version v3.2.1
2016-02-03T11:17:19.306-0600 I CONTROL  [initandlisten] git version: a14d55980c2
cdc565d4704a7e3ad37e4e535c1b2
2016-02-03T11:17:19.307-0600 I CONTROL  [initandlisten] OpenSSL version: OpenSSL
 1.0.1p-fips 9 Jul 2015
2016-02-03T11:17:19.307-0600 I CONTROL  [initandlisten] allocator: tcmalloc
2016-02-03T11:17:19.307-0600 I CONTROL  [initandlisten] modules: none
2016-02-03T11:17:19.307-0600 I CONTROL  [initandlisten] build environment:
2016-02-03T11:17:19.307-0600 I CONTROL  [initandlisten]     distmod: 2008plus-ss
l
2016-02-03T11:17:19.307-0600 I CONTROL  [initandlisten]     distarch: x86_64
2016-02-03T11:17:19.307-0600 I CONTROL  [initandlisten]     target_arch: x86_64
2016-02-03T11:17:19.307-0600 I CONTROL  [initandlisten] options: { master: true,
 net: { http: { RESTInterfaceEnabled: true, enabled: true }, port: 10000 }, stor
age: { dbPath: "master" } }
2016-02-03T11:17:19.308-0600 I -        [initandlisten] Detected data files in m
aster created by the 'wiredTiger' storage engine, so setting the active storage
engine to 'wiredTiger'.
2016-02-03T11:17:19.309-0600 I STORAGE  [initandlisten] wiredtiger_open config:
create,cache_size=1G,session_max=20000,eviction=(threads_max=4),config_base=fals
e,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snapp
y),file_manager=(close_idle_time=100000),checkpoint=(wait=60,log_size=2GB),stati
stics_log=(wait=0),
2016-02-03T11:17:20.068-0600 I STORAGE  [initandlisten] Starting WiredTigerRecor
dStoreThread local.oplog.$main
2016-02-03T11:17:20.068-0600 I STORAGE  [initandlisten] The size storer reports
that the oplog contains 48 records totaling to 2976 bytes
2016-02-03T11:17:20.068-0600 I STORAGE  [initandlisten] Scanning the oplog to de
termine where to place markers for truncation
2016-02-03T11:17:20.152-0600 I NETWORK  [websvr] admin web console waiting for c
onnections on port 11000
2016-02-03T11:17:20.154-0600 I NETWORK  [HostnameCanonicalizationWorker] Startin
g hostname canonicalization worker
2016-02-03T11:17:20.155-0600 I FTDC     [initandlisten] Initializing full-time d
iagnostic data capture with directory "master/diagnostic.data"
2016-02-03T11:17:20.158-0600 I NETWORK  [initandlisten] waiting for connections
on port 10000
```

From the Web Browser…



Slave

> mongod --dbpath slave --port 10001 --slave –rest --source localhost:10000



Insert data in the Master

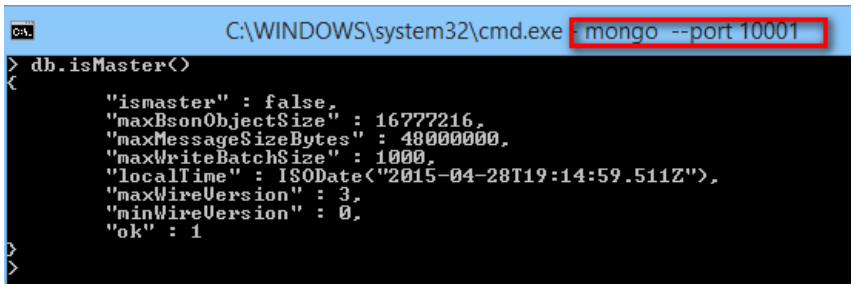
```
C:\MongoDB3.2>mongo --port 10000
MongoDB shell version: 3.2.1
connecting to: 127.0.0.1:10000/test
> db.fruits.insert({name:"Apple", price: 1.2})
WriteResult({ "nInserted" : 1 })
> db.fruits.insert({name:"Orange", price: 1.5})
WriteResult({ "nInserted" : 1 })
> db.fruits.find()
{ "_id" : ObjectId("56b239c8f9ef63224f0dee78"), "name" : "Apple", "price" : 1.2
}
{ "_id" : ObjectId("56b239ddf9ef63224f0dee79"), "name" : "Orange", "price" : 1.5
}
>
```

> db.isMaster()

```
> db.isMaster()
{
        "ismaster" : true,
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 1000,
        "localTime" : ISODate("2015-04-28T19:11:27.592Z"),
        "maxWireVersion" : 3,
        "minWireVersion" : 0,
        "ok" : 1
}
```

Verify from the Slave

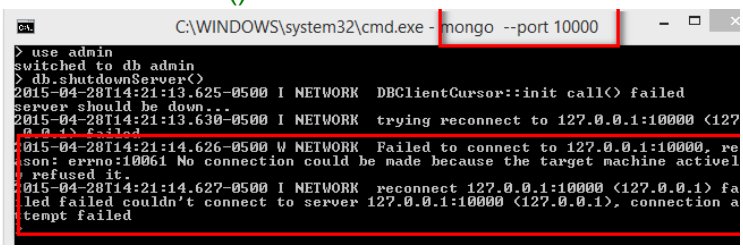
> db.isMaster()

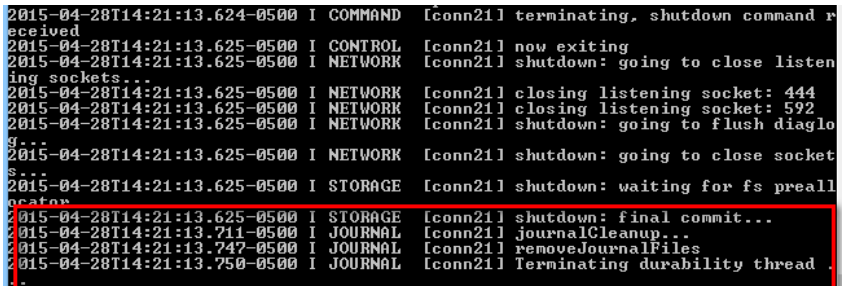


The Master fails

> use admin

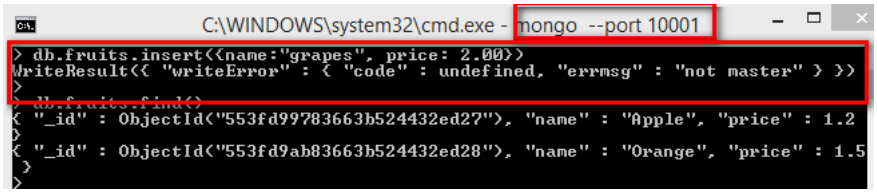
> db.shutdownServer()





On the Slave

- All slaves must be replicated from a master node.
- There is currently no mechanism for replicating from a slave (*daisy chaining), because slaves do not keep their own* ***oplog****.*
- There is no explicit limit on the number of slaves in a cluster, but having a thousand slaves querying a single master will likely overwhelm the master node.
- In practice, clusters with **less than a dozen slaves** tend to work well.
- Can't write data to slave. Can still read data.



master-slave replication options

- ***--only:*** Use on a slave node to specify only a single database to replicate. *(The default is to replicate all databases.)*
- ***--slavedelay:*** Use on a slave node to add a delay *(in seconds)* to be used when applying operations from the master.
  - This makes it easy to set up delayed slaves, which can be useful in case a user accidentally deletes important documents or inserts bad data.
  - By delaying the application of operations, you have a window in which recovery from the bad operation is possible.
- ***--fastsync:*** Start a slave from a snapshot of the master node. This option allows a slave to bootstrap much faster than doing a full sync, if its data directory is initialized with a snapshot of the master's data.
- ***--autoresync:*** Automatically perform a full resync if this slave gets out of sync with the master.
- ***--oplogSize****:* Size (in megabytes) for the master's oplog.

❑ Set Up A Two-node Replica Set

- **Specify a Data Directory:** If you want **mongod** to store data files at a path *other than C:***data\db** you can specify a **dbpath**. The **dbpath** **must exist** before you start **mongod**.
  - Create a directory for each node to store data **C:\mongodb3.2\node1**
- **Specify a TCP Port:** Only a single process can listen for connections on a network interface at a time. If you run multiple **mongod** processes on a single machine, you must assign each a different port to listen on for client connections.
  - To specify a port to **mongod**, use the **--port** option on the command line.
- The basic setup is to start a master node and one or more slave nodes, each of which knows the address of the master.
  - *Deploy a Replica Set*
  - *Add Members to a Replica Set*
  - *Test Failover*
  - *Primary Election*
  - *Check Status*
  - *Access Data After Role Switch*

mongod

mongod is the primary daemon process for the MongoDB system. It handles data requests, manages data format, and performs background management operations.

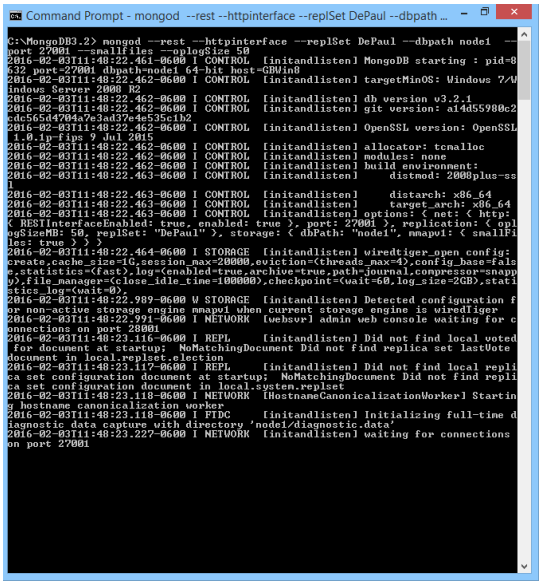
- **--smallfiles:** MongoDB uses a smaller default file size. It reduces data files from 2 GB to **512 MB** and journal files from 1 GB to **128 MB**.
  - Use **smallfiles** if you have a large number of databases that each holds a small quantity of data.
- **--oplogSize:** To record operations on the **master**.
  - The oplog is stored in a special database called ***local****, in the* ***oplog.$main collection***.
  - When starting the server, which allows you to specify the size of the oplog in megabytes. By default, 64-bit instances will use **5%** of available free space for the oplog.
- **--replSet:** Name of a replica set. All nodes must have the same set name.
- **--rest:** Enables the simple REST API allows HTTP clients to run commands against the MongoDB server.

```
mongod --rest --httpinterface --replSet DePaul --dbpath node1
--port 27001 --smallfiles --oplogSize 50
```

The operation log (Oplog)

- **oplog** stores only operations that ***change*** the state of the database.
  - A query, for example, would not be stored in the oplog.
  - oplog is a mechanism to keep data on slaves **in sync** with the master.
- As new operations are stored in the oplog, they will automatically replace the oldest operations. This guarantees that the oplog does not grow beyond a preset bound. That bound is configurable using the ***--oplogSize*** option.
- Each document in the oplog represents a single operation performed on the master server.
  - ***ts***: *Timestamp for the operation. The timestamp type is an internal type used to track* when operations are performed. It is composed of a 4-byte timestamp and a 4-byte incrementing counter.
  - ***op***: Type of operation performed as a 1-byte code (e.g., "i" for an insert).
  - ***ns***: Namespace (collection name) where the operation was performed.
  - ***o***: Document further specifying the operation to perform. For an insert, this would be the document to insert.

On the Master



Replica Commands

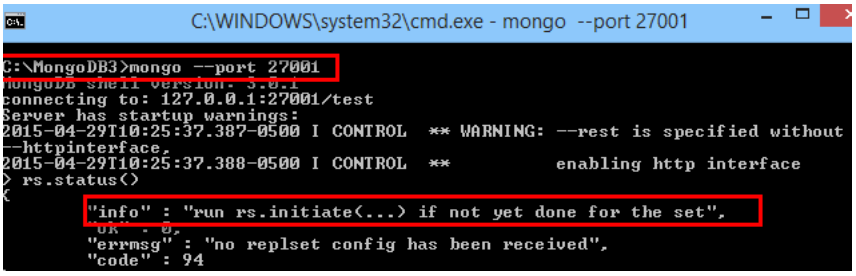
- **rs.status()** The output reflects the current status of the replica set, using data derived from the **heartbeat packets** sent by the other members of the replica set.
- **rs.initiate**(*configuration*) Initiates a replica set.
  - *Configuration* - Optionally takes a configuration argument in the form of a *document* that holds the configuration of a replica set.

```
{ _id: <setname>,
members: [  {_id: 0, host : <host0>},
            {_id: 1, host : <host1>},
            {_id: 2, host : <host2>}  ] }
```

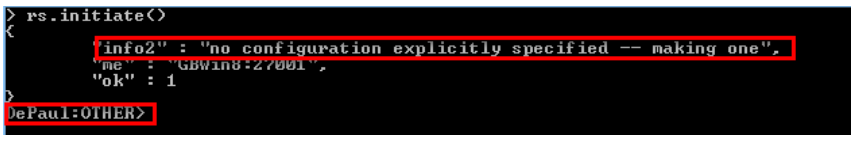
- **rs.add**(*hostspec*, *arbiterOnly*) Provides a simple method to add a member to an existing *replica set*.
  - **Host** – A hostname.
- **db.isMaster()** Returns a status document with fields that includes the **ismaster** field that reports if the current node is the ***primary*** node as well as a report of a subset of current replica set configuration.
  - **arbiterOnly** (*boolean*) – Optional. If **true**, this host is an arbiter.

Initiate the replica set and check status

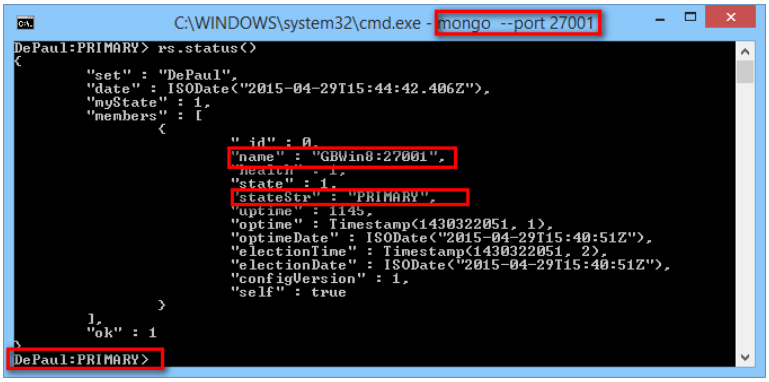
> rs.status()



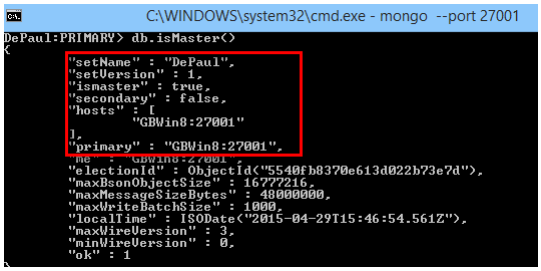
> rs.initiate()

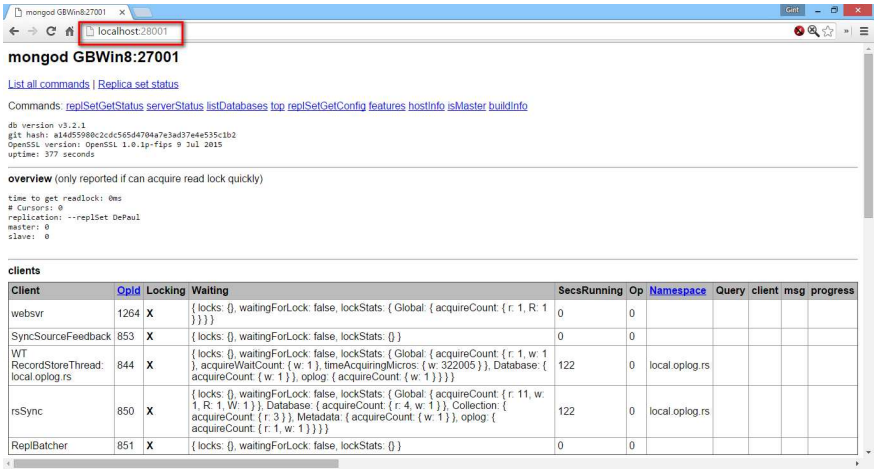


> rs.status()



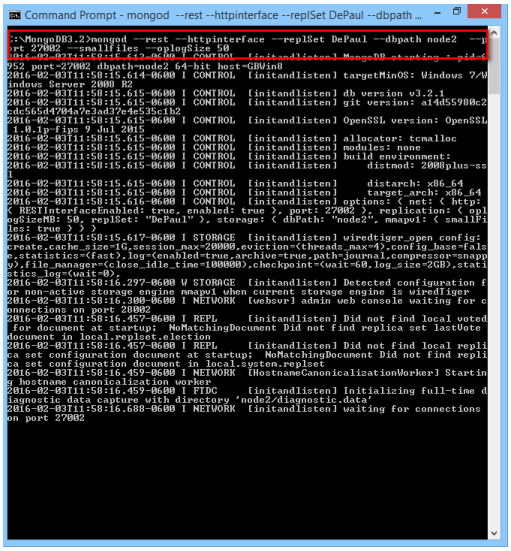
> db.isMaster()





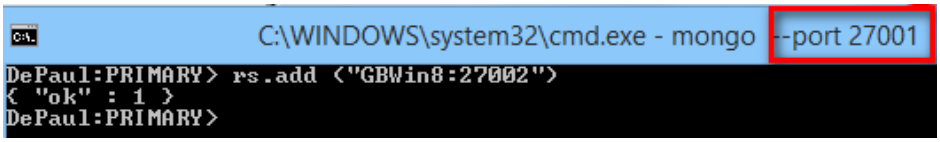
Add a second member to the set

```
mongod --rest --httpinterface --replSet DePaul --dbpath node2  --port 27002 --smallfiles --oplogSize 50
```

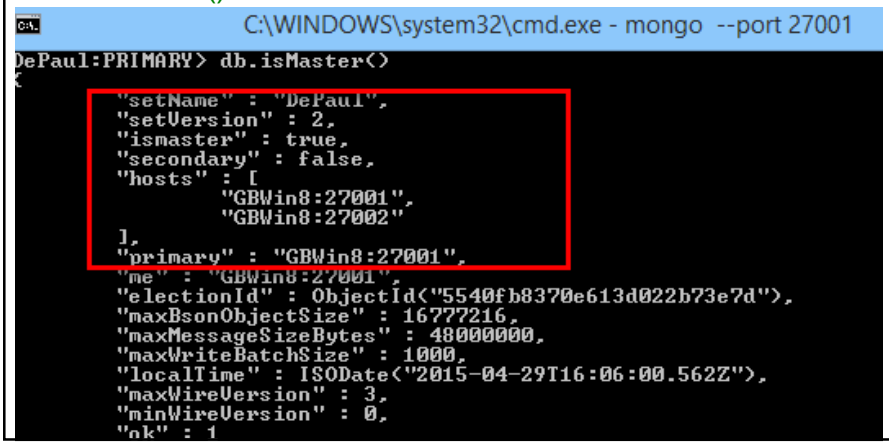


From node 1 mongo shell

```
> rs.add ("GBWin8:27002")
```

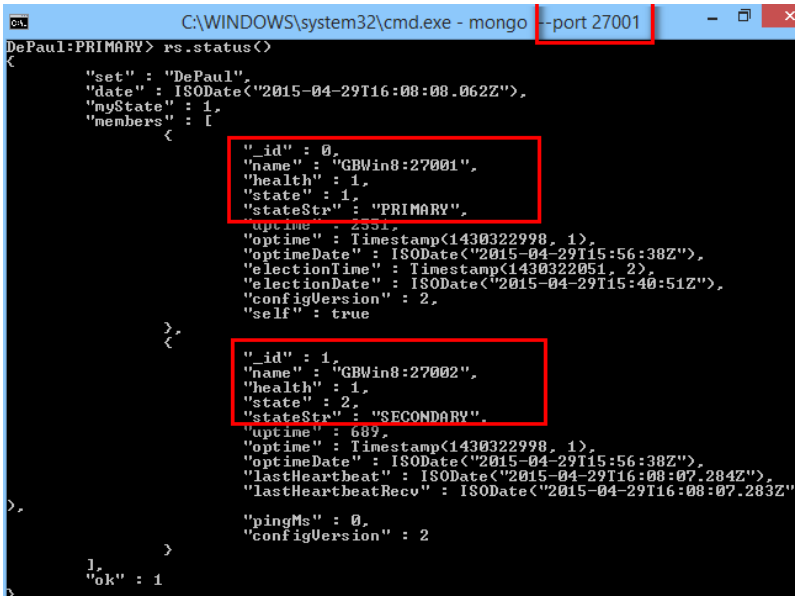



```
> db.isMaster()
```



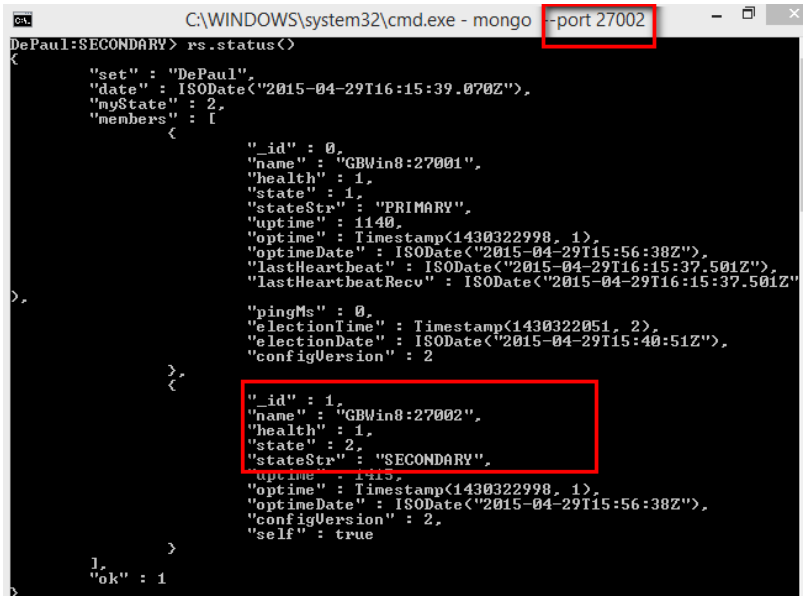
From node 1 mongo shell

> rs.status()



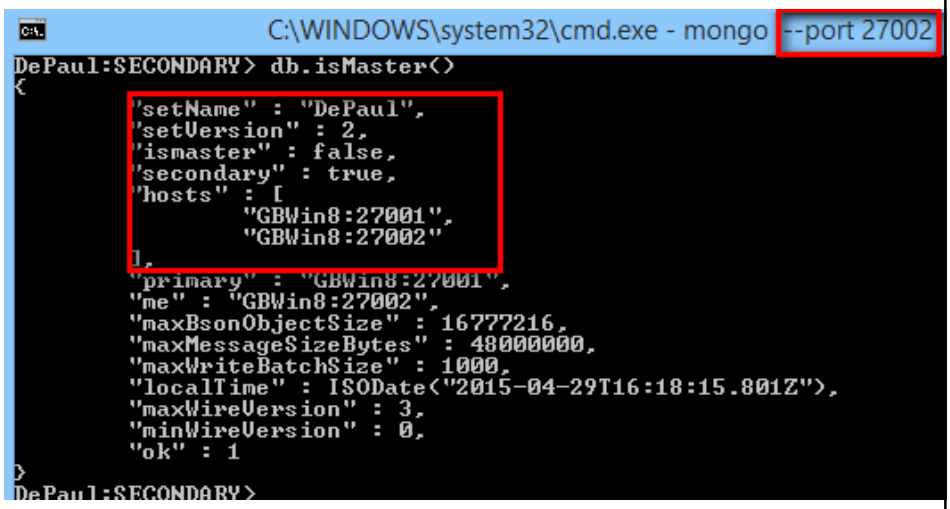
Start a second member mongo shell and check status

> rs.status()



Check replication status from web console

> db.isMaster()



On the Master - Insert/retrieve data

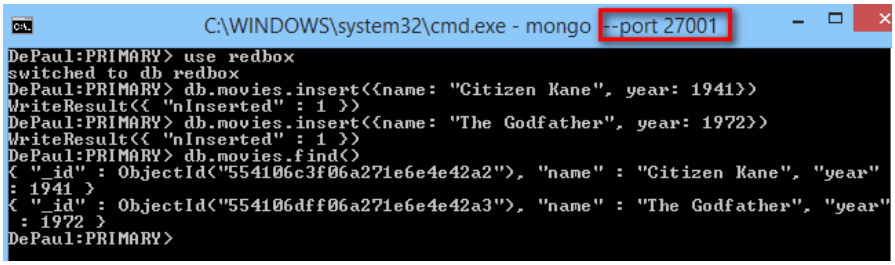
- Create a database called **redbox**

  > use redbox
- Insert two movies into movies collection

  > db.movies.insert({name: “Citizen Kane", year: 1941})

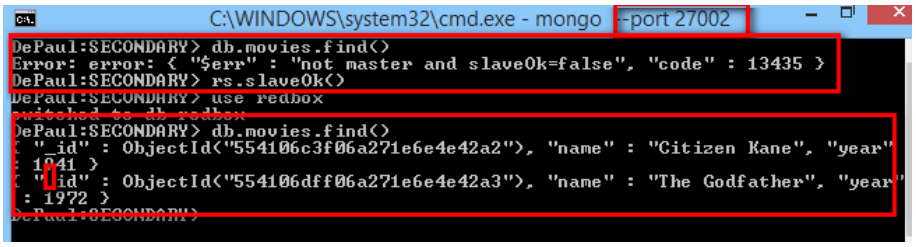
  > db.movies.insert({name: “The Godfather", year: 1972})
- Verify two movies in the collection

  > db.movies.find()



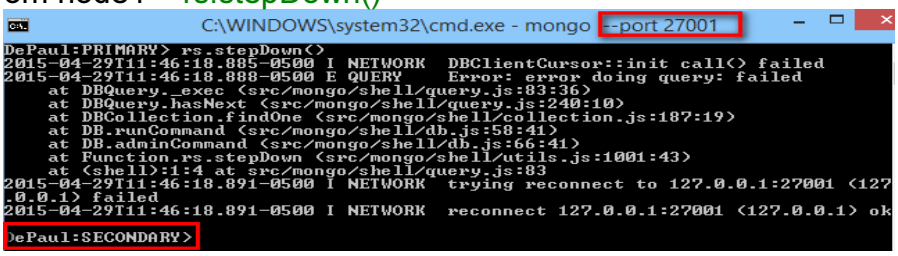
Retrieve movies from the secondary node

> rs.slaveOk() allows read operations to run on *secondary* nodes.

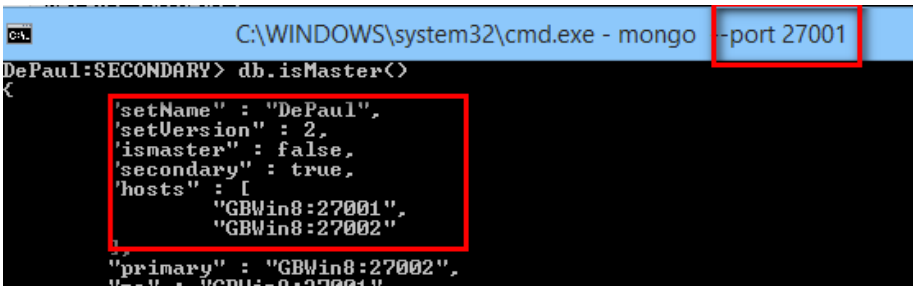


Fail over and check status on both nodes

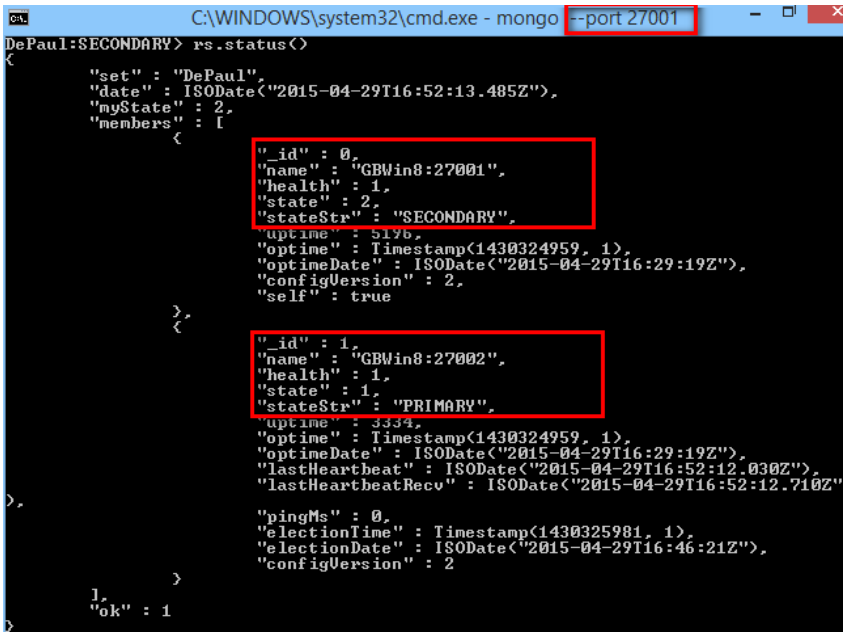
From node1 > rs.stepDown()



> db.isMaster()

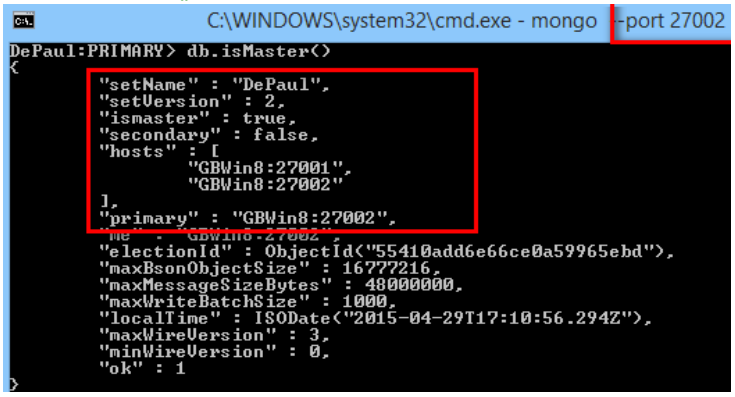


> rs.status()

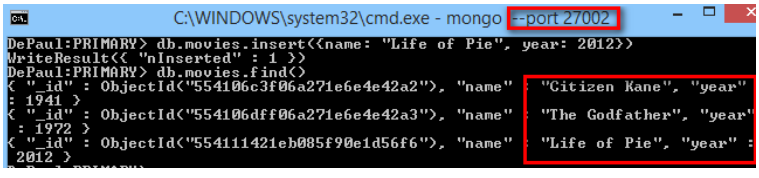


From node2 … Check status and insert another movie

> db.isMaster()

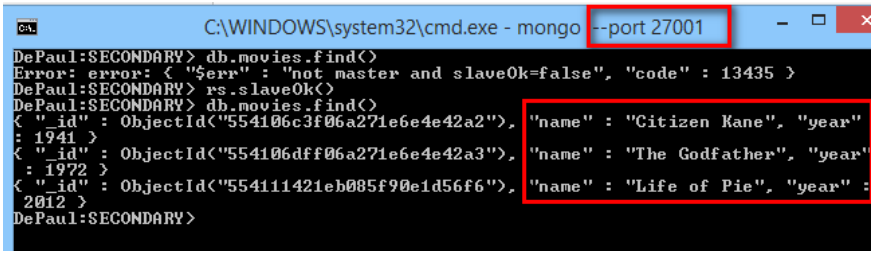


> db.movies.insert({name: "Life of Pie", year: 2012})



Verify movies on node 1 *(current Secondary)*

> rs.slaveOk()



Replication Commands

rs.help(): Returns a basic help text for all of the *replication* related shell functions

- rs.status() { replSetGetStatus : 1 } checks repl set status
- rs.initiate() { replSetInitiate : null } initiates set with default settings
- rs.initiate(cfg) { replSetInitiate : cfg } initiates set with configuration cfg
- rs.conf() get the current configuration object from local.system.replset
- rs.reconfig(cfg) updates the configuration of a running replica set with cfg
- rs.add(hostportstr) add a new member to the set with default attributes
- rs.add(membercfgobj) add a new member to the set with extra attributes
- rs.addArb(hostportstr) add a new member which is arbiterOnly:true
- rs.stepDown([secs]) step down as primary (momentarily)
- rs.syncFrom(hostportstr) make a secondary to sync from the given member
- rs.freeze(secs) make a node ineligible to become primary for the time specified
- rs.remove(hostportstr) remove a host from the replica set
- rs.slaveOk() shorthand for db.getMongo().setSlaveOk()
- db.isMaster() check who is primary reconfiguration helpers disconnect from the database so the shell will display an error, even if the command succeeds.

Summary

- **High Availability (auto-failover)**
- **Read Scaling (extra copies to read from)**
- **Backups**
  - Online, Delayed Copy (fat finger)
  - Point-In-Time (PIT) backups
- **Use (hidden) replica for secondary workload**
  - Analytics
  - Data-processing
  - Integration with external systems
- **Avoid single points of failure**
  - Separate racks
  - Separate data centers
- **Avoid long recovery downtime**
  - Use 3+ replicas
  - Use journaling
- **Keep your actives close**
  - Use priority to control where failovers happen