Replication and MongoDB

Release 2.6.7

MongoDB Documentation Project

January 13, 2015

Contents

A *replica set* in MongoDB is a group of `mongod` processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments. This section introduces replication in MongoDB as well as the components and architecture of replica sets. The section also provides tutorials for common tasks related to replica sets.

Replication Introduction **(page 3)** An introduction to replica sets, their behavior, operation, and use.

Replication Concepts **(page 5)** The core documentation of replica set operations, configurations, architectures and behaviors.

Replica Set Members **(page 7)** Introduces the components of replica sets.

Replica Set Deployment Architectures **(page 15)** Introduces architectural considerations related to replica sets deployment planning.

Replica Set High Availability **(page 22)** Presents the details of the automatic failover and recovery process with replica sets.

Replica Set Read and Write Semantics **(page 27)** Presents the semantics for targeting read and write operations to the replica set, with an awareness of location and set configuration.

Replica Set Tutorials **(page 42)** Tutorials for common tasks related to the use and maintenance of replica sets.

Replication Reference **(page 92)** Reference for functions and operations related to replica sets.

1 Replication Introduction

Replication is the process of synchronizing data across multiple servers.

1.1 Purpose of Replication

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

In some cases, you can use replication to increase read capacity. Clients have the ability to send read and write operations to different servers. You can also maintain copies in different data centers to increase the locality and availability of data for distributed applications.

1.2 Replication in MongoDB

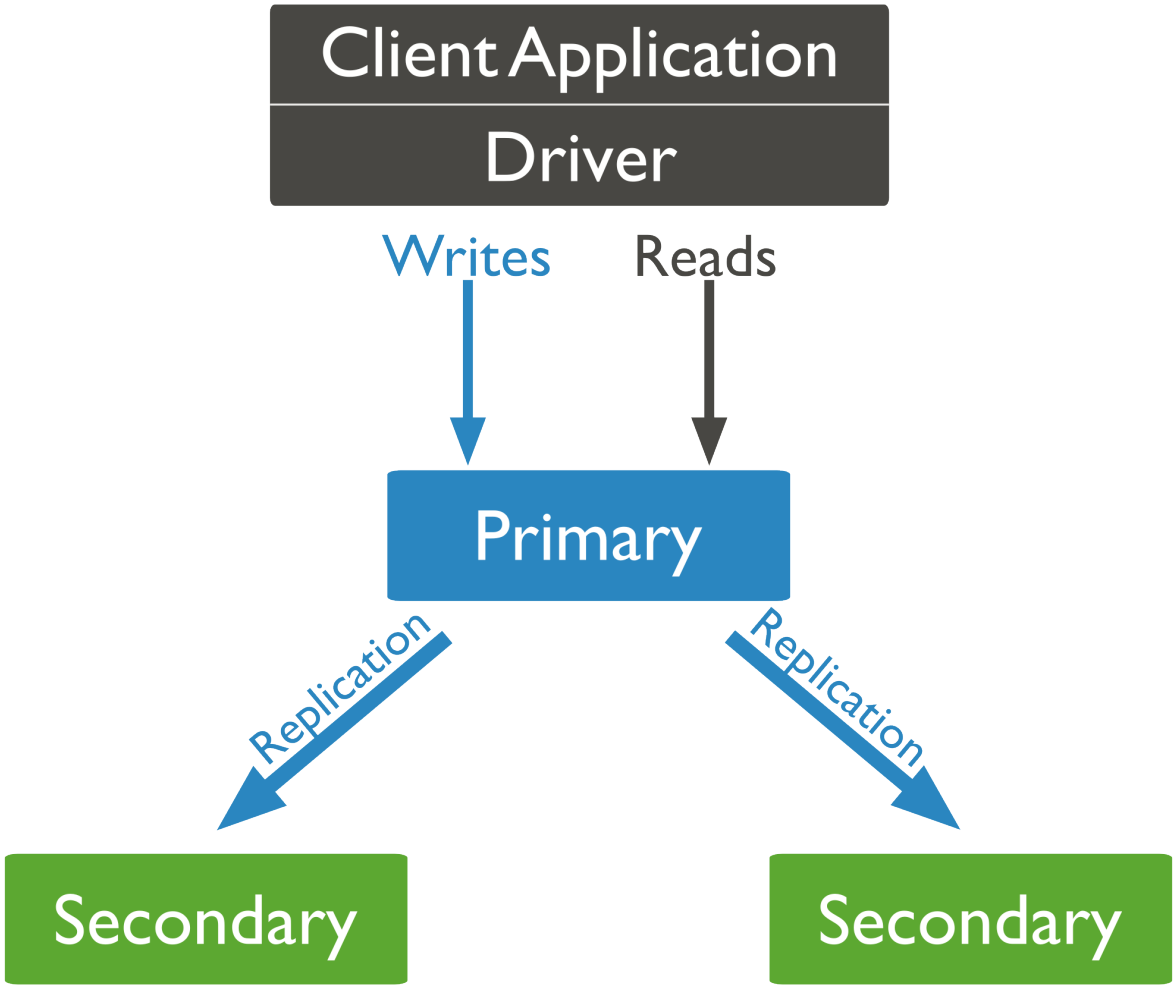
A replica set is a group of `mongod` instances that host the same data set. One `mongod`, the primary, receives all write operations. All other instances, secondaries, apply operations from the primary so that they have the same data set.

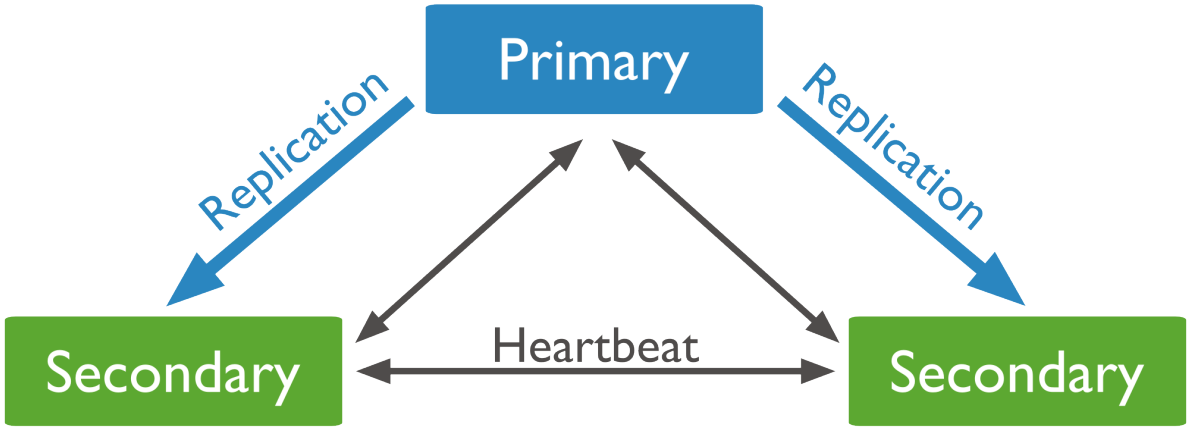
The **primary** accepts all write operations from clients. Replica set can have only one primary. Because only one member can accept write operations, replica sets provide **strict consistency** for all reads from the primary. To support replication, the primary logs all changes to its data sets in its *oplog* (page 34). See *primary* (page 7) for more information.

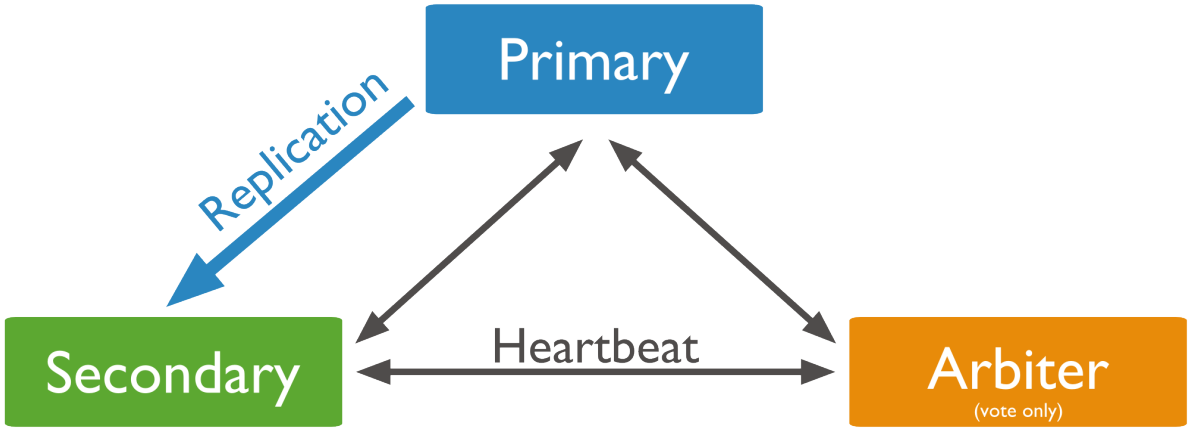
The **secondaries** replicate the primary's oplog and apply the operations to their data sets. Secondaries' data sets reflect the primary's data set. If the primary is unavailable, the replica set will elect a secondary to be primary. By default, clients read from the primary, however, clients can specify a *read preferences* (page 29) to send read operations to secondaries. Reads from secondaries may return data that does not reflect the state of the primary. See *secondaries* (page 9) for more information.

You may add an extra `mongod` instance to a replica set as an **arbiter**. Arbiters do not maintain a data set. Arbiters only exist to vote in elections. If your replica set has an even number of members, add an arbiter to obtain a majority of votes in an election for primary. Arbiters do not require dedicated hardware. See *arbiter* (page 14) for more information.

An **arbiter** will always be an arbiter. A **primary** may step down and become a **secondary**. A **secondary** may become the primary during an election.


Client Application
Driver
Writes
Reads
Primary
Replication
Replication
Secondary
Secondary


Primary
Replication
Replication
Heartbeat
Secondary
Secondary



Asynchronous Replication

Secondaries apply operations from the primary asynchronously. By applying operations after the primary, sets can continue to function without some members. However, as a result secondaries may not return the most current data to clients.

See *Replica Set Oplog* (page 34) and *Replica Set Data Synchronization* (page 35) for more information. See *Read Preference* (page 29) for more on read operations and secondaries.

Automatic Failover

When a primary does not communicate with the other members of the set for more than 10 seconds, the replica set will attempt to select another member to become the new primary. The first secondary that receives a majority of the votes becomes primary.

See *Replica Set Elections* (page 22) and *Rollbacks During Replica Set Failover* (page 26) for more information.

Additional Features

Replica sets provide a number of options to support application needs. For example, you may deploy a replica set with *members in multiple data centers* (page 21), or control the outcome of elections by adjusting the `priority` (page 95) of some members. Replica sets also support dedicated members for reporting, disaster recovery, or backup functions.

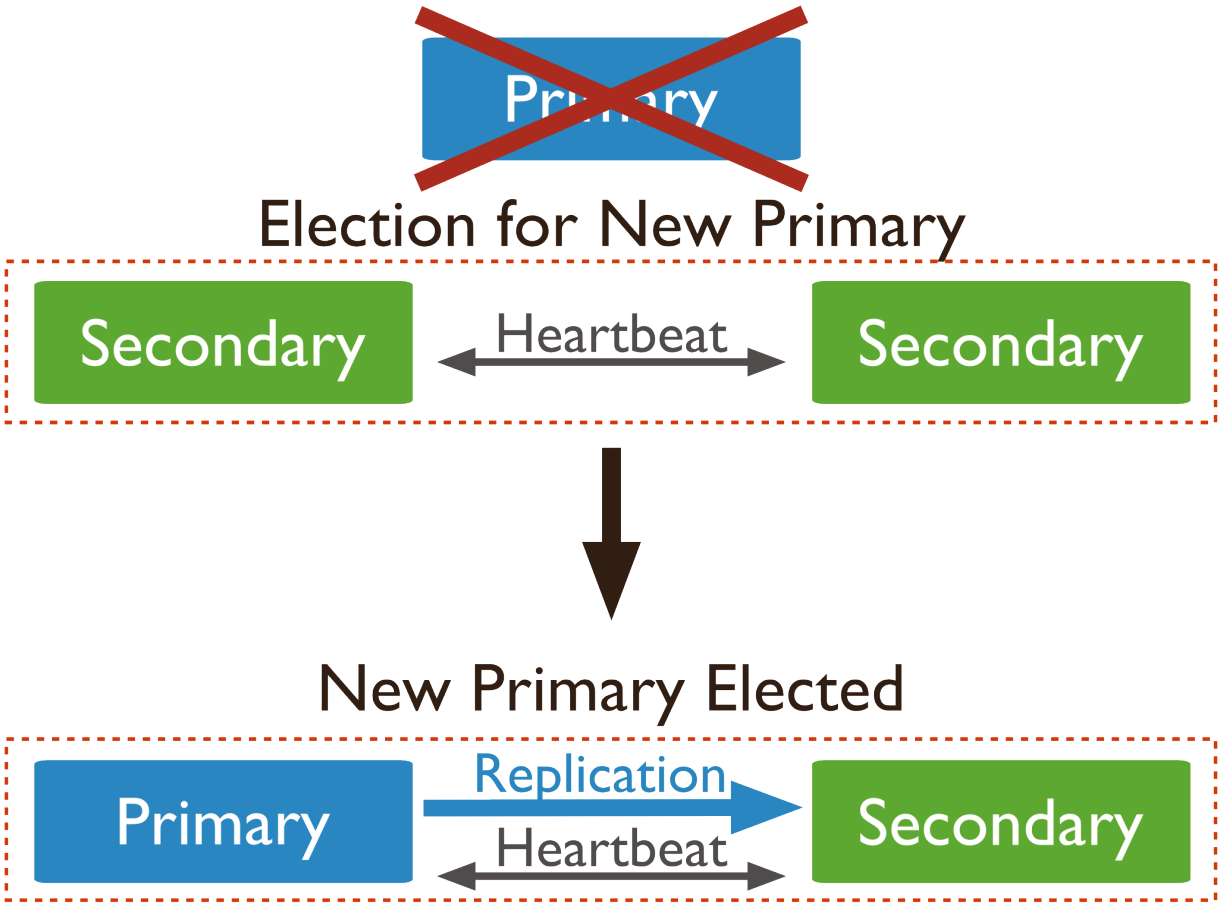
See *Priority 0 Replica Set Members* (page 11), *Hidden Replica Set Members* (page 12) and *Delayed Replica Set Members* (page 13) for more information.

2 Replication Concepts

These documents describe and provide examples of replica set operation, configuration, and behavior. For an overview of replication, see *Replication Introduction* (page 3). For documentation of the administration of replica sets, see *Replica Set Tutorials* (page 42). The *Replication Reference* (page 92) documents commands and operations specific to replica sets.

Replica Set Members **(page 7)** Introduces the components of replica sets.

> ***Replica Set Primary*** **(page 7)** The primary is the only member of a replica set that accepts write operations.


Primary
Election for New Primary
Secondary
Heartbeat
Secondary
New Primary Elected
Primary
Replication
Heartbeat
Secondary

Replica Set Secondary Members **(page 9)** Secondary members replicate the primary's data set and accept read operations. If the set has no primary, a secondary can become primary.

Priority 0 Replica Set Members **(page 11)** Priority 0 members are secondaries that cannot become the primary.

Hidden Replica Set Members **(page 12)** Hidden members are secondaries that are invisible to applications. These members support dedicated workloads, such as reporting or backup.

Replica Set Arbiter **(page 14)** An arbiter does not maintain a copy of the data set but participate in elections.

Replica Set Deployment Architectures **(page 15)** Introduces architectural considerations related to replica sets deployment planning.

Replica Set High Availability **(page 22)** Presents the details of the automatic failover and recovery process with replica sets.

Replica Set Elections **(page 22)** Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.

Read Preference **(page 29)** Applications specify *read preference* to control how drivers direct read operations to members of the replica set.

Replication Processes **(page 34)** Mechanics of the replication process and related topics.

Master Slave Replication **(page 37)** Master-slave replication provided redundancy in early versions of MongoDB. Replica sets replace master-slave for most use cases.

2.1 Replica Set Members

A *replica set* in MongoDB is a group of `mongod` processes that provide redundancy and high availability. The members of a replica set are:

***Primary* (page ??).** The *primary* receives all write operations.

***Secondaries* (page ??).** Secondaries replicate operations from the primary to maintain an identical data set. Secondaries may have additional configurations for special usage profiles. For example, secondaries may be *non-voting* (page 25) or *priority 0* (page 11).

You can also maintain an *arbiter* (page **??**) as part of a replica set. Arbiters do not keep a copy of the data. However, arbiters play a role in the elections that select a primary if the current primary is unavailable.

A replica set can have up to 12 members. [1] However, only 7 members can vote at a time.

The minimum requirements for a replica set are: A *primary* (page **??**), a *secondary* (page **??**), and an *arbiter* (page **??**). Most deployments, however, will keep three members that store data: A *primary* (page **??**) and two *secondary members* (page **??**).

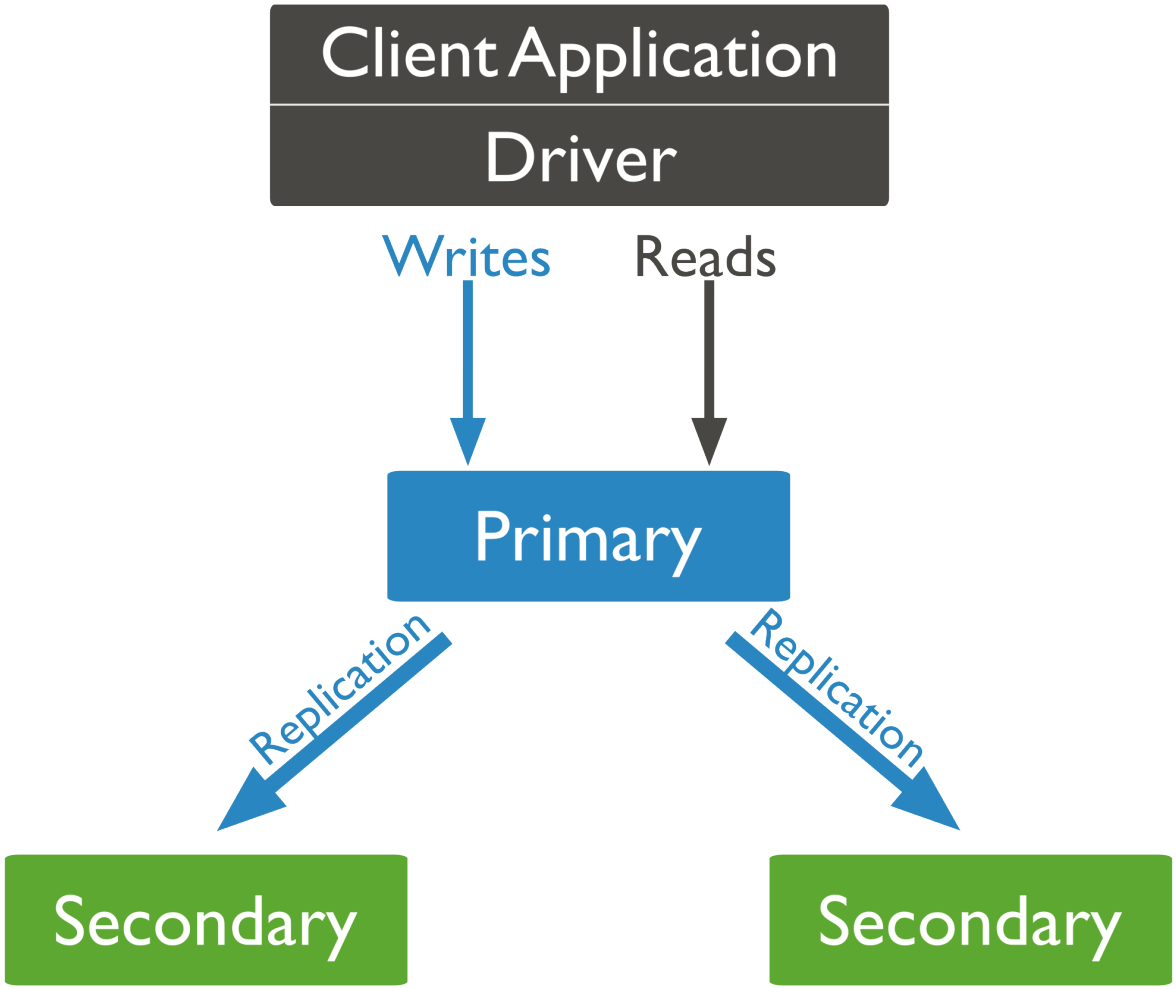
Replica Set Primary

The primary is the only member in the replica set that receives write operations. MongoDB applies write operations on the *primary* and then records the operations on the primary's *oplog* (page 34). *Secondary* (page **??**) members replicate this log and apply the operations to their data sets.

In the following three-member replica set, the primary accepts all write operations. Then the secondaries replicate the oplog to apply to their data sets.

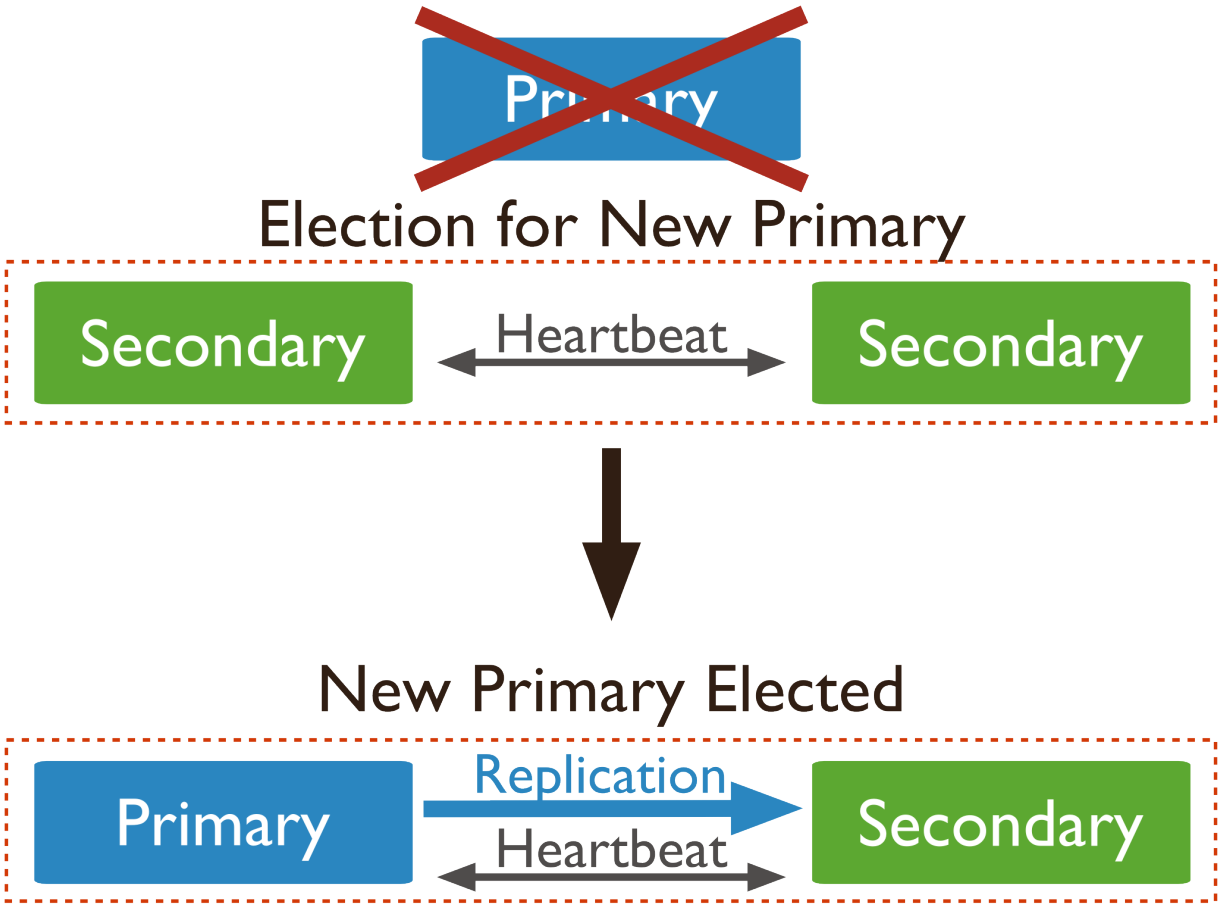
All members of the replica set can accept read operations. However, by default, an application directs its read operations to the primary member. See *Read Preference* (page 29) for details on changing the default read behavior.

[1] While replica sets are the recommended solution for production, a replica set can support only 12 members in total. If your deployment requires more than 12 members, you'll need to use *master-slave* (page 37) replication. Master-slave replication lacks the automatic failover capabilities.


Client Application
Driver
Writes
Reads
Primary
Replication
Replication
Secondary
Secondary

The replica set can have at most one primary. If the current primary becomes unavailable, an election determines the new primary. See *Replica Set Elections* (page 22) for more details.

In the following 3-member replica set, the primary becomes unavailable. This triggers an election which selects one of the remaining secondaries as the new primary.



Replica Set Secondary Members

A secondary maintains a copy of the *primary's* data set. To replicate data, a secondary applies operations from the primary's *oplog* (page 34) to its own data set in an asynchronous process. A replica set can have one or more secondaries.

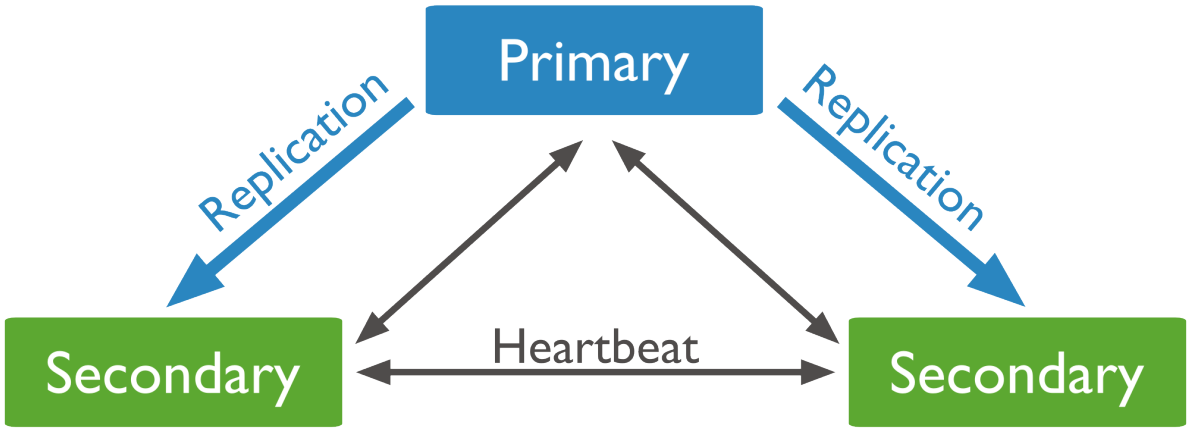
The following three-member replica set has two secondary members. The secondaries replicate the primary's oplog and apply the operations to their data sets.

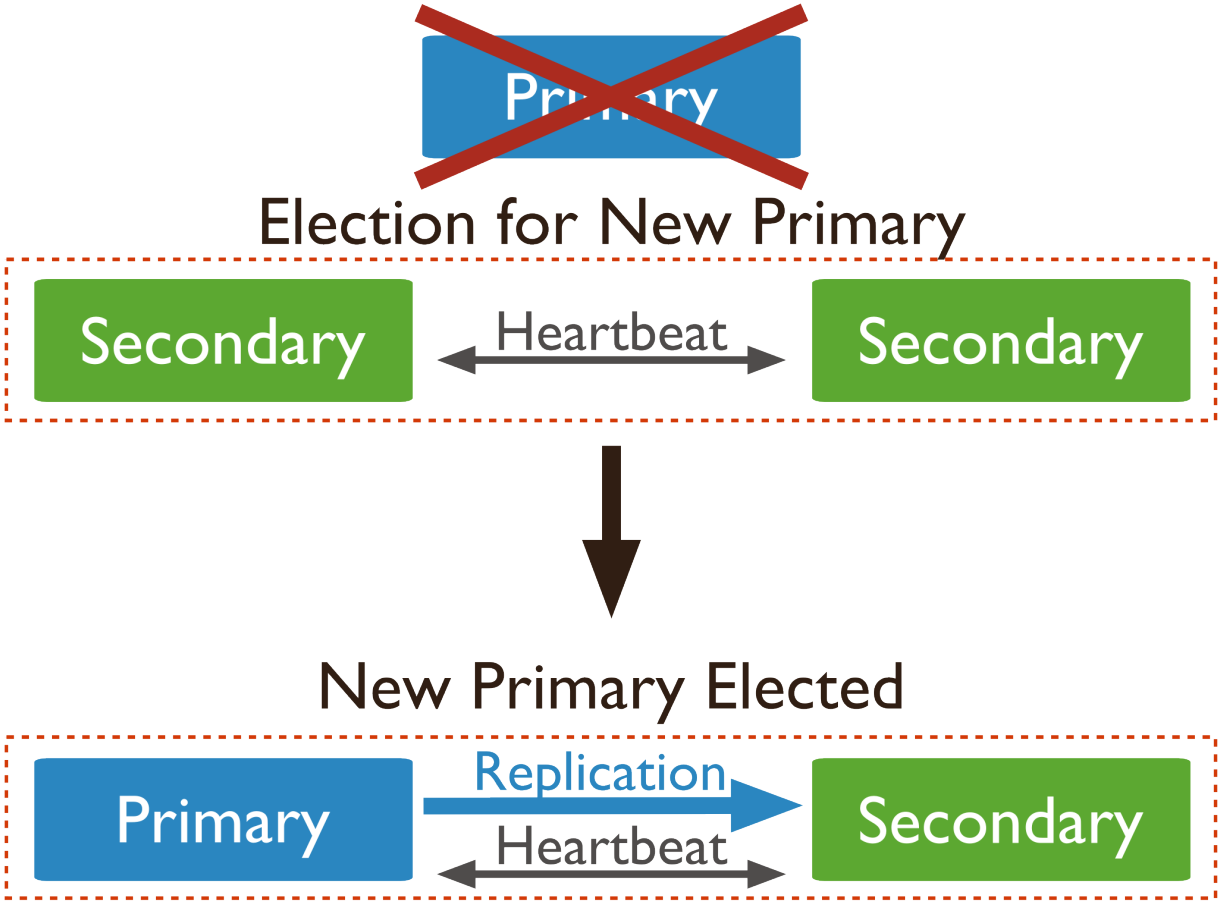
Although clients cannot write data to secondaries, clients can read data from secondary members. See *Read Preference* (page 29) for more information on how clients direct read operations to replica sets.

A secondary can become a primary. If the current primary becomes unavailable, the replica set holds an *election* to choose which of the secondaries becomes the new primary.

In the following three-member replica set, the primary becomes unavailable. This triggers an election where one of the remaining secondaries becomes the new primary.

See *Replica Set Elections* (page 22) for more details.


Primary
Replication
Replication
Secondary
Heartbeat
Secondary


Election for New Primary
Secondary
Heartbeat
Secondary
New Primary Elected
Primary
Replication
Heartbeat
Secondary

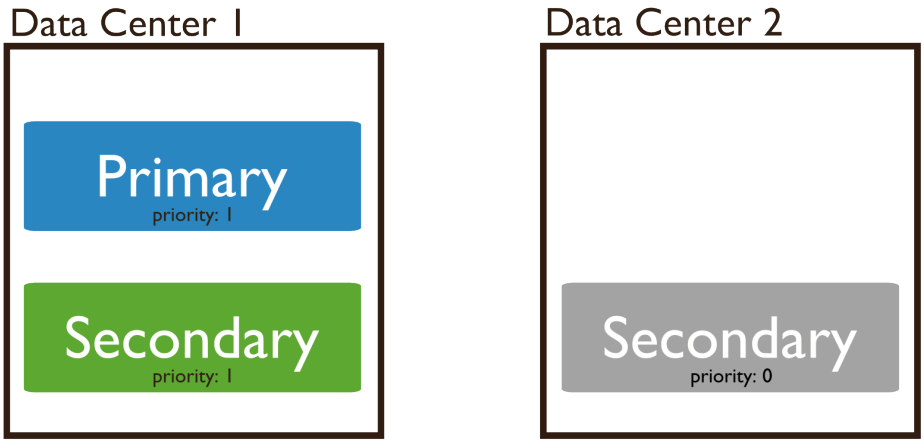
You can configure a secondary member for a specific purpose. You can configure a secondary to:

- Prevent it from becoming a primary in an election, which allows it to reside in a secondary data center or to serve as a cold standby. See *Priority 0 Replica Set Members* (page 11).
- Prevent applications from reading from it, which allows it to run applications that require separation from normal traffic. See *Hidden Replica Set Members* (page 12).
- Keep a running "historical" snapshot for use in recovery from certain errors, such as unintentionally deleted databases. See *Delayed Replica Set Members* (page 13).

Priority 0 Replica Set Members

A *priority 0* member is a secondary that **cannot** become *primary*. *Priority 0* members cannot *trigger elections*. Otherwise these members function as normal secondaries. A *priority 0* member maintains a copy of the data set, accepts read operations, and votes in elections. Configure a *priority 0* member to prevent *secondaries* from becoming primary, which is particularly useful in multi-data center deployments.

In a three-member replica set, in one data center hosts the primary and a secondary. A second data center hosts one *priority 0* member that cannot become primary.



Priority 0 Members as Standbys A *priority 0* member can function as a standby. In some replica sets, it might not be possible to add a new member in a reasonable amount of time. A standby member keeps a current copy of the data to be able to replace an unavailable member.

In many cases, you need not set standby to *priority 0*. However, in sets with varied hardware or *geographic distribution* (page 21), a *priority 0* standby ensures that only qualified members become primary.

A *priority 0* standby may also be valuable for some members of a set with different hardware or workload profiles. In these cases, deploy a member with *priority 0* so it can't become primary. Also consider using an *hidden member* (page 12) for this purpose.

If your set already has seven voting members, also configure the member as *non-voting* (page 25).

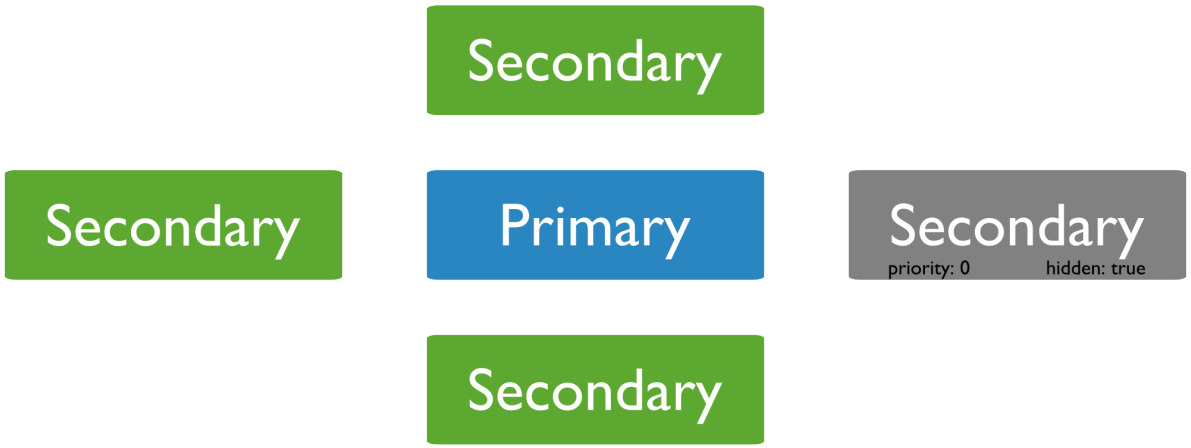
Priority 0 Members and Failover When configuring a *priority 0* member, consider potential failover patterns, including all possible network partitions. Always ensure that your main data center contains both a quorum of voting members and contains members that are eligible to be primary.

Configuration To configure a *priority 0* member, see *Prevent Secondary from Becoming Primary* (page 62).

Hidden Replica Set Members

A hidden member maintains a copy of the *primary's* data set but is **invisible** to client applications. Hidden members are good for workloads with different usage patterns from the other members in the *replica set*. Hidden members must always be *priority 0 members* (page 11) and so **cannot become primary**. The `db.isMaster()` method does not display hidden members. Hidden members, however, **do vote** in *elections* (page 22).

In the following five-member replica set, all four secondary members have copies of the primary's data set, but one of the secondary members is hidden.



Behavior

Read Operations Clients will not distribute reads with the appropriate *read preference* (page 29) to hidden members. As a result, these members receive no traffic other than basic replication. Use hidden members for dedicated tasks such as reporting and backups. *Delayed members* (page 13) should be hidden.

In a sharded cluster, `mongos` do not interact with hidden members.

Voting Hidden members *do* vote in replica set elections. If you stop a hidden member, ensure that the set has an active majority or the *primary* will step down.

For the purposes of backups, you can avoid stopping a hidden member with the `db.fsyncLock()` and `db.fsyncUnlock()` operations to flush all writes and lock the `mongod` instance for the duration of the backup operation.

Further Reading For more information about backing up MongoDB databases, see `http://docs.mongodb.org/manual/core/backups`. To configure a hidden member, see *Configure a Hidden Replica Set Member* (page 64).

Delayed Replica Set Members

Delayed members contain copies of a *replica set's* data set. However, a delayed member's data set reflects an earlier, or delayed, state of the set. For example, if the current time is 09:52 and a member has a delay of an hour, the delayed member has no operation more recent than 08:52.

Because delayed members are a "rolling backup" or a running "historical" snapshot of the data set, they may help you recover from various kinds of human error. For example, a delayed member can make it possible to recover from unsuccessful application upgrades and operator errors including dropped databases and collections.

Considerations

Requirements Delayed members:

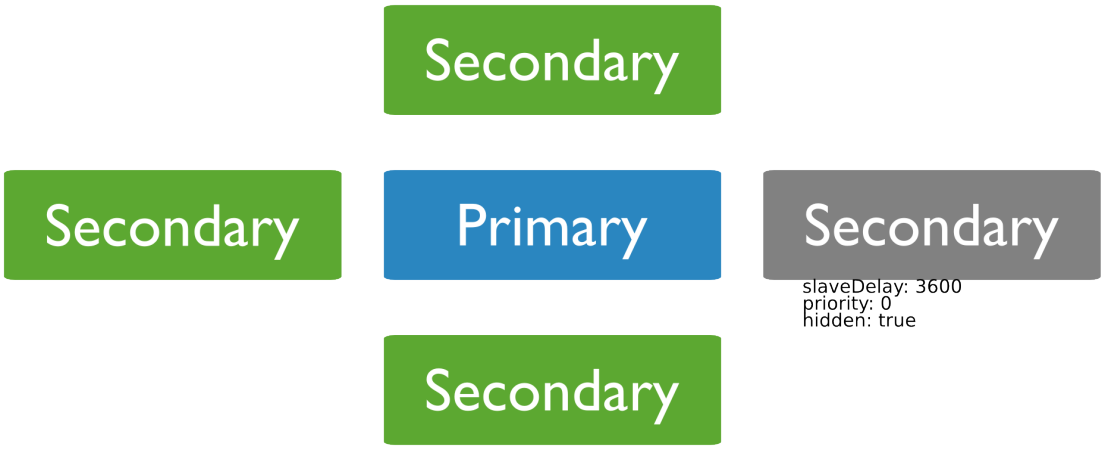
- **Must be** *priority 0* (page 11) members. Set the priority to 0 to prevent a delayed member from becoming primary.
- **Should be** *hidden* (page 12) members. Always prevent applications from seeing and querying delayed members.
- *do* vote in *elections* for primary.

Behavior Delayed members apply operations from the *oplog* on a delay. When choosing the amount of delay, consider that the amount of delay:

- must be is equal to or greater than your maintenance windows.
- must be *smaller* than the capacity of the oplog. For more information on oplog size, see *Oplog Size* (page 34).

Sharding In sharded clusters, delayed members have limited utility when the *balancer* is enabled. Because delayed members replicate chunk migrations with a delay, the state of delayed members in a sharded cluster are not useful for recovering to a previous state of the sharded cluster if any migrations occur during the delay window.

Example In the following 5-member replica set, the primary and all secondaries have copies of the data set. One member applies operations with a delay of 3600 seconds, or an hour. This delayed member is also *hidden* and is a *priority 0 member*.



Configuration A delayed member has its `priority` (page 95) equal to `0`, `hidden` (page 95) equal to `true`, and its `slaveDelay` (page 96) equal to the number of seconds of delay:

```
{
   "_id" : <num>,
   "host" : <hostname:port>,
   "priority" : 0,
   "slaveDelay" : <seconds>,
   "hidden" : true
}
```

To configure a delayed member, see *Configure a Delayed Replica Set Member* (page 65).

Replica Set Arbiter

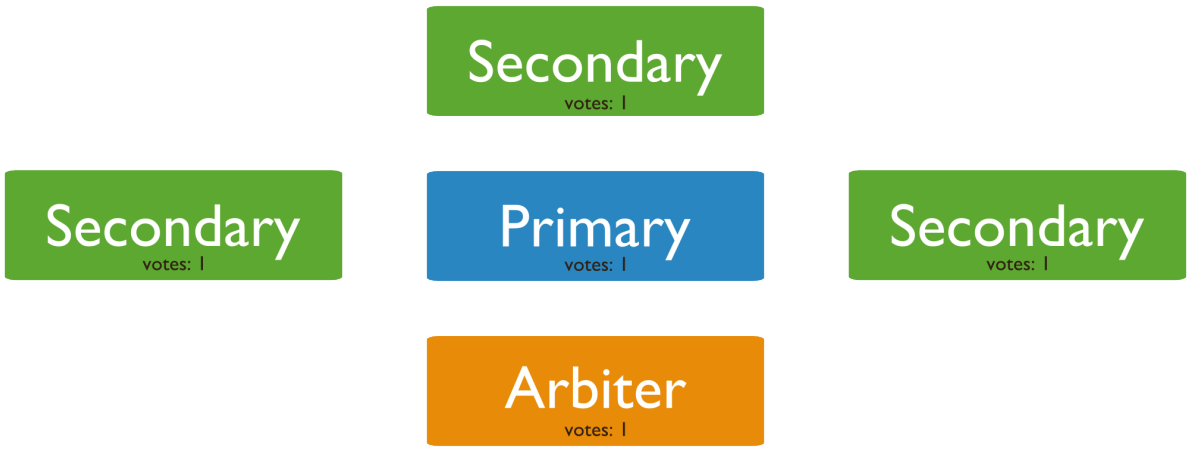
An arbiter does **not** have a copy of data set and **cannot** become a primary. Replica sets may have arbiters to add a vote in *elections of for primary* (page 22). Arbiters allow replica sets to have an uneven number of members, without the overhead of a member that replicates data.

Important: Do not run an arbiter on systems that also host the primary or the secondary members of the replica set.

Only add an arbiter to sets with even numbers of members. If you add an arbiter to a set with an odd number of members, the set may suffer from tied *elections*. To add an arbiter, see *Add an Arbiter to Replica Set* (page 55).

Example

For example, in the following replica set, an arbiter allows the set to have an odd number of votes for elections:



Security

Authentication When running with `authorization`, arbiters exchange credentials with other members of the set to authenticate. MongoDB encrypts the authentication process. The MongoDB authentication exchange is cryptographically secure.

Arbiters use `keyfiles` to authenticate to the replica set.

Communication The only communication between arbiters and other set members are: votes during elections, heartbeats, and configuration data. These exchanges are not encrypted.

However, if your MongoDB deployment uses SSL, MongoDB will encrypt *all* communication between replica set members. See `http://docs.mongodb.org/manual/tutorial/configure-ssl` for more information.

As with all MongoDB components, run arbiters in trusted network environments.

2.2 Replica Set Deployment Architectures

The architecture of a *replica set* affects the set's capacity and capability. This document provides strategies for replica set deployments and describes common architectures.

The standard replica set deployment for production system is a three-member replica set. These sets provide redundancy and fault tolerance. Avoid complexity when possible, but let your application requirements dictate the architecture.

Strategies

Determine the Number of Members

Add members in a replica set according to these strategies.

Deploy an Odd Number of Members An odd number of members ensures that the replica set is always able to elect a primary. If you have an even number of members, add an arbiter to get an odd number. *Arbiters* do not store a copy of the data and require fewer resources. As a result, you may run an arbiter on an application server or other shared process.

Consider Fault Tolerance *Fault tolerance* for a replica set is the number of members that can become unavailable and still leave enough members in the set to elect a primary. In other words, it is the difference between the number of members in the set and the majority needed to elect a primary. Without a primary, a replica set cannot accept write operations. Fault tolerance is an effect of replica set size, but the relationship is not direct. See the following table:

Number of Members.	Majority Required to Elect a New Primary.	Fault Tolerance.
3	2	1
4	3	1
5	3	2
6	4	2

Adding a member to the replica set does not *always* increase the fault tolerance. However, in these cases, additional members can provide support for dedicated functions, such as backups or reporting.

Use Hidden and Delayed Members for Dedicated Functions Add *hidden* (page 12) or *delayed* (page 13) members to support dedicated functions, such as backup or reporting.

Load Balance on Read-Heavy Deployments In a deployment with *very* high read traffic, you can improve read throughput by distributing reads to secondary members. As your deployment grows, add or move members to alternate data centers to improve redundancy and availability.

Always ensure that the main facility is able to elect a primary.

Add Capacity Ahead of Demand The existing members of a replica set must have spare capacity to support adding a new member. Always add new members before the current demand saturates the capacity of the set.

Determine the Distribution of Members

Distribute Members Geographically To protect your data if your main data center fails, keep at least one member in an alternate data center. Set these members' `priority` (page 95) to 0 to prevent them from becoming primary.

Keep a Majority of Members in One Location When a replica set has members in multiple data centers, network partitions can prevent communication between data centers. To replicate data, members must be able to communicate to other members.

In an election, members must see each other to create a majority. To ensure that the replica set members can confirm a majority and elect a primary, keep a majority of the set's members in one location.

Target Operations with Tags

Use *replica set tags* (page 76) to ensure that operations replicate to specific data centers. Tags also support targeting read operations to specific machines.

See also:

`/data-center-awareness` and `http://docs.mongodb.org/manual/core/operational-segregation`.

Use Journaling to Protect Against Power Failures

Enable journaling to protect data against service interruptions. Without journaling MongoDB cannot recover data after unexpected shutdowns, including power failures and unexpected reboots.

All 64-bit versions of MongoDB after version 2.0 have journaling enabled by default.

Replica Set Naming

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

Deployment Patterns

The following documents describe common replica set deployment patterns. Other patterns are possible and effective depending on the application's requirements. If needed, combine features of each architecture in your own deployment:

Three Member Replica Sets **(page 17)** Three-member replica sets provide the minimum recommended architecture for a replica set.

Replica Sets with Four or More Members **(page 17)** Four or more member replica sets provide greater redundancy and can support greater distribution of read operations and dedicated functionality.

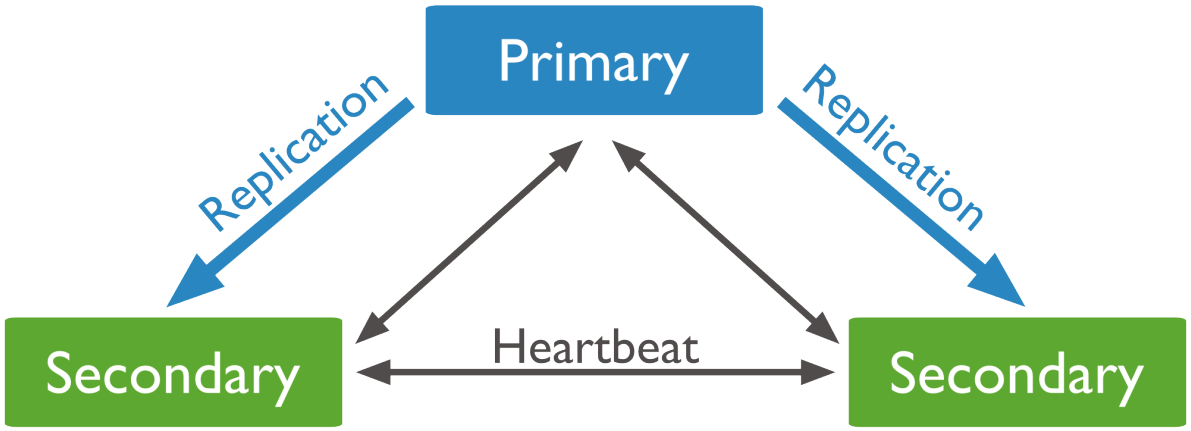
Geographically Distributed Replica Sets **(page 21)** Geographically distributed sets include members in multiple locations to protect against facility-specific failures, such as power outages.

Three Member Replica Sets

The minimum architecture of a replica set has three members. A three member replica set can have either three members that hold data, or two members that hold data and an arbiter.

Primary with Two Secondary Members A replica set with three members that store data has:

- One *primary* (page 7).
- Two *secondary* (page 9) members. Both secondaries can become the primary in an *election* (page 22).



These deployments provide two complete copies of the data set at all times in addition to the primary. These replica sets provide additional fault tolerance and *high availability* (page 22). If the primary is unavailable, the replica set elects a secondary to be primary and continues normal operation. The old primary rejoins the set when available.

Primary with a Secondary and an Arbiter A three member replica set with a two members that store data has:

- One *primary* (page 7).
- One *secondary* (page 9) member. The secondary can become primary in an *election* (page 22).
- One *arbiter* (page 14). The arbiter only votes in elections.

Since the arbiter does not hold a copy of the data, these deployments provides only one complete copy of the data. Arbiters require fewer resources, at the expense of more limited redundancy and fault tolerance.

However, a deployment with a primary, secondary, and an arbiter ensures that a replica set remains available if the primary *or* the secondary is unavailable. If the primary is unavailable, the replica set will elect the secondary to be primary.

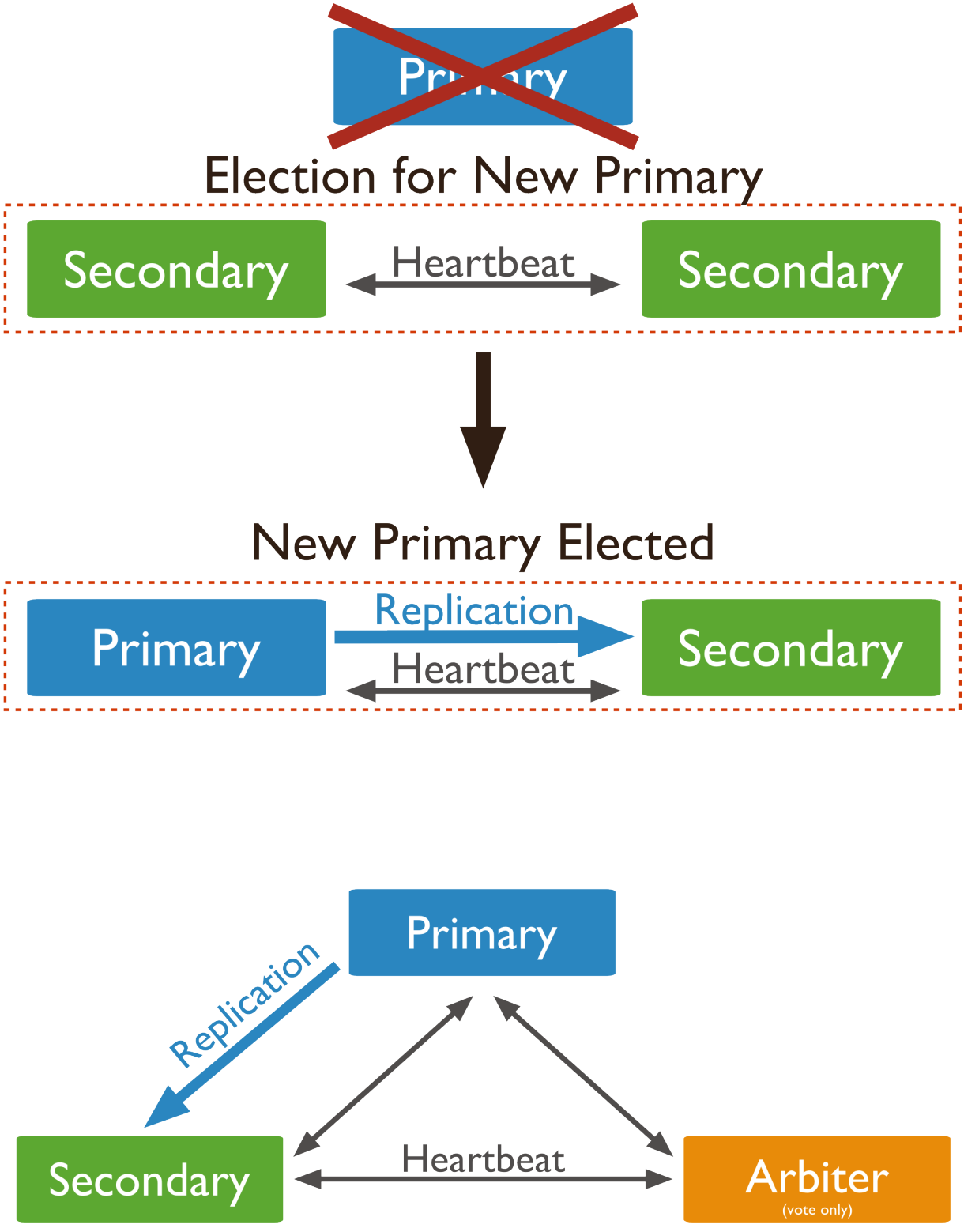
See also:

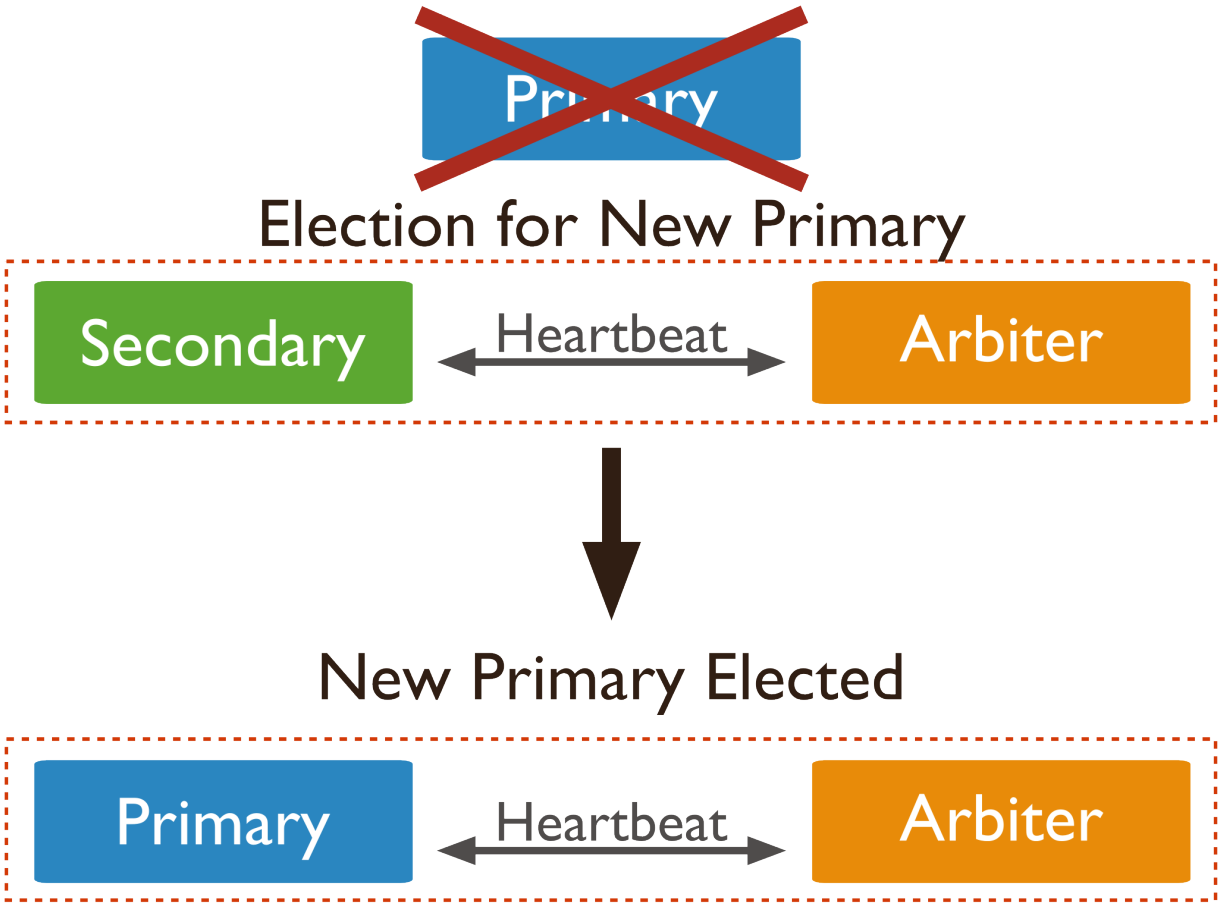
Deploy a Replica Set (page 44).

Replica Sets with Four or More Members

Although the standard replica set configuration has three members you can deploy larger sets. Add additional members to a set to increase redundancy or to add capacity for distributing secondary read operations.

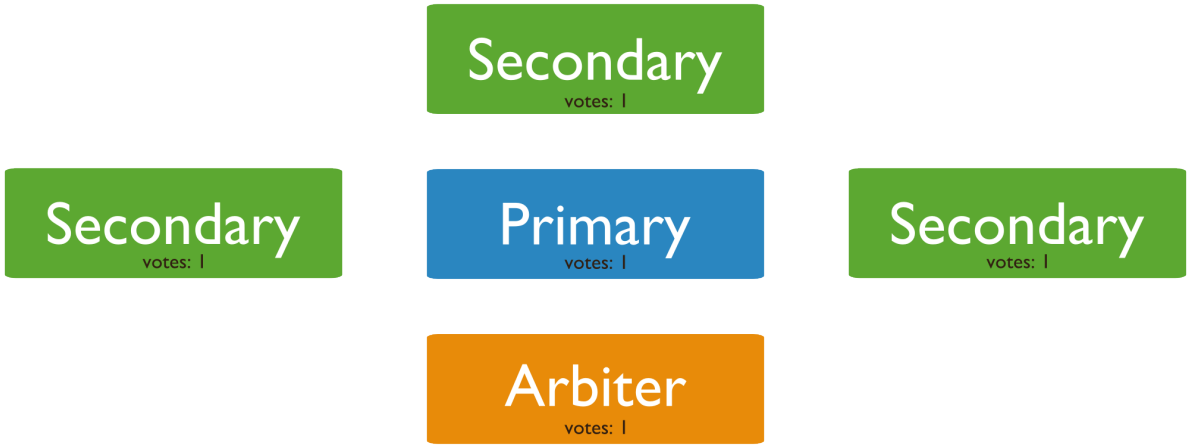
When adding members, ensure that:


Election for New Primary
Secondary
Heartbeat
Secondary
New Primary Elected
Primary
Replication
Heartbeat
Secondary
Primary
Replication
Secondary
Heartbeat
Arbiter
(vote only)


Election for New Primary
Secondary
Heartbeat
Arbiter
New Primary Elected
Primary
Heartbeat
Arbiter

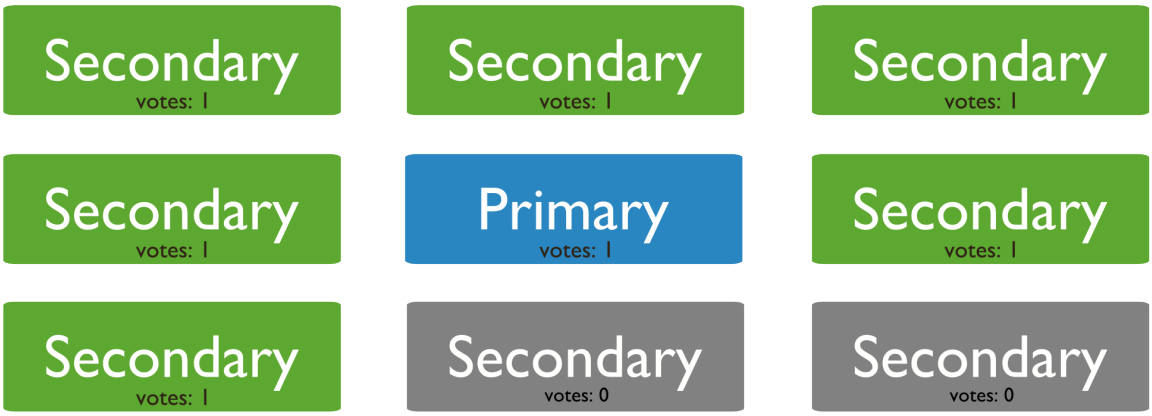
- The set has an odd number of voting members. If you have an *even* number of voting members, deploy an *arbiter* (page **??**) so that the set has an odd number.

  The following replica set needs an arbiter to have an odd number of voting members.



- A replica set can have up to 12 members, [2] but only 7 voting members. See *non-voting members* (page 25) for more information.

  The following 9 member replica set has 7 voting members and 2 non-voting members.



- Members that cannot become primary in a *failover* have *priority 0 configuration* (page 11).

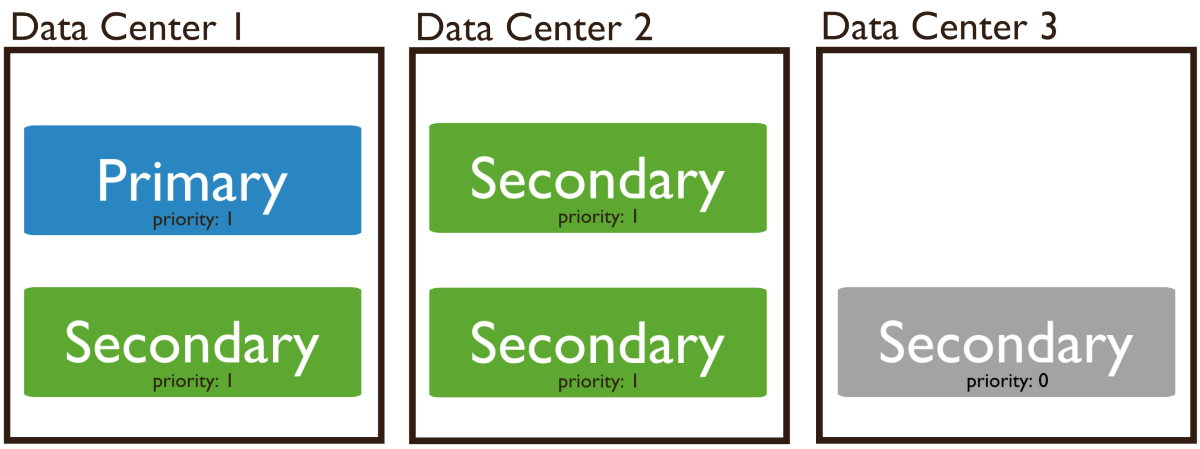
  For instance, some members that have limited resources or networking constraints and should never be able to become primary. Configure members that should not become primary to have *priority 0* (page 11). In following replica set, the secondary member in the third data center has a priority of 0:

- A majority of the set's members should be in your applications main data center.

See also:

Deploy a Replica Set (page 44), *Add an Arbiter to Replica Set* (page 55), and *Add Members to a Replica Set* (page 57).

[2] While replica sets are the recommended solution for production, a replica set can support only 12 members in total. If your deployment requires more than 12 members, you'll need to use *master-slave* (page 37) replication. Master-slave replication lacks the automatic failover capabilities.



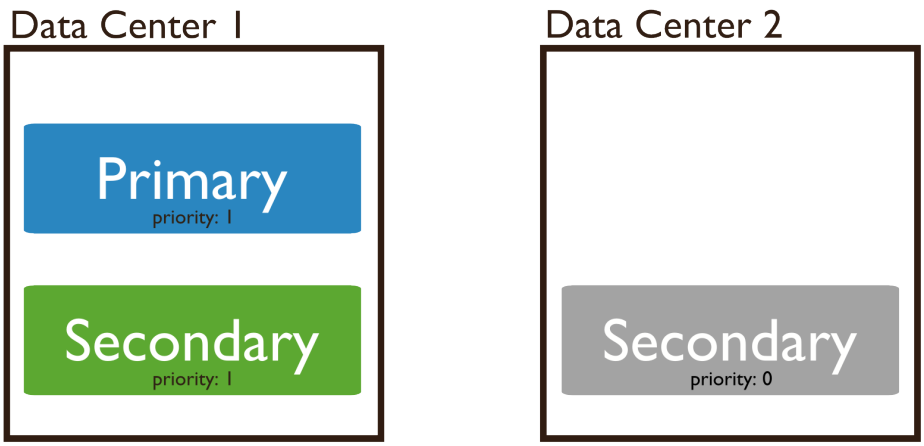
Geographically Distributed Replica Sets

Adding members to a replica set in multiple data centers adds redundancy and provides fault tolerance if one data center is unavailable. Members in additional data centers should have a *priority of 0* (page 11) to prevent them from becoming primary.

For example: the architecture of a geographically distributed replica set may be:

- One *primary* in the main data center.
- One *secondary* member in the main data center. This member can become primary at any time.
- One *priority 0* (page 11) member in a second data center. This member cannot become primary.

In the following replica set, the primary and one secondary are in *Data Center 1*, while *Data Center 2* has a *priority 0* (page 11) secondary that cannot become a primary.



If the primary is unavailable, the replica set will elect a new primary from *Data Center 1*. If the data centers cannot connect to each other, the member in *Data Center 2* will not become the primary.

If *Data Center 1* becomes unavailable, you can manually recover the data set from *Data Center 2* with minimal downtime. With sufficient *write concern*, there will be no data loss.

To facilitate elections, the main data center should hold a majority of members. Also ensure that the set has an odd

number of members. If adding a member in another data center results in a set with an even number of members, deploy an *arbiter* (page **??**). For more information on elections, see *Replica Set Elections* (page 22).

See also:

Deploy a Geographically Redundant Replica Set (page 49).

2.3 Replica Set High Availability

Replica sets provide high availability using automatic *failover*. Failover allows a *secondary* member to become *primary* if primary is unavailable. Failover, in most situations does not require manual intervention.

Replica set members keep the same data set but are otherwise independent. If the primary becomes unavailable, the replica set holds an *election* (page 22) to select a new primary. In some situations, the failover process may require a *rollback* (page 26). [3]

The deployment of a replica set affects the outcome of failover situations. To support effective failover, ensure that one facility can elect a primary if needed. Choose the facility that hosts the core application systems to host the majority of the replica set. Place a majority of voting members and all the members that can become primary in this facility. Otherwise, network partitions could prevent the set from being able to form a majority.

Failover Processes

The replica set recovers from the loss of a primary by holding an election. Consider the following:

Replica Set Elections **(page 22)** Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.

Rollbacks During Replica Set Failover **(page 26)** A rollback reverts write operations on a former primary when the member rejoins the replica set after a failover.

Replica Set Elections

Replica sets use elections to determine which set member will become *primary*. Elections occur after initiating a replica set, and also any time the primary becomes unavailable. The primary is the only member in the set that can accept write operations. If a primary becomes unavailable, elections allow the set to recover normal operations without manual intervention. Elections are part of the *failover process* (page 22).

In the following three-member replica set, the primary is unavailable. The remaining secondaries hold an election to choose a new primary.

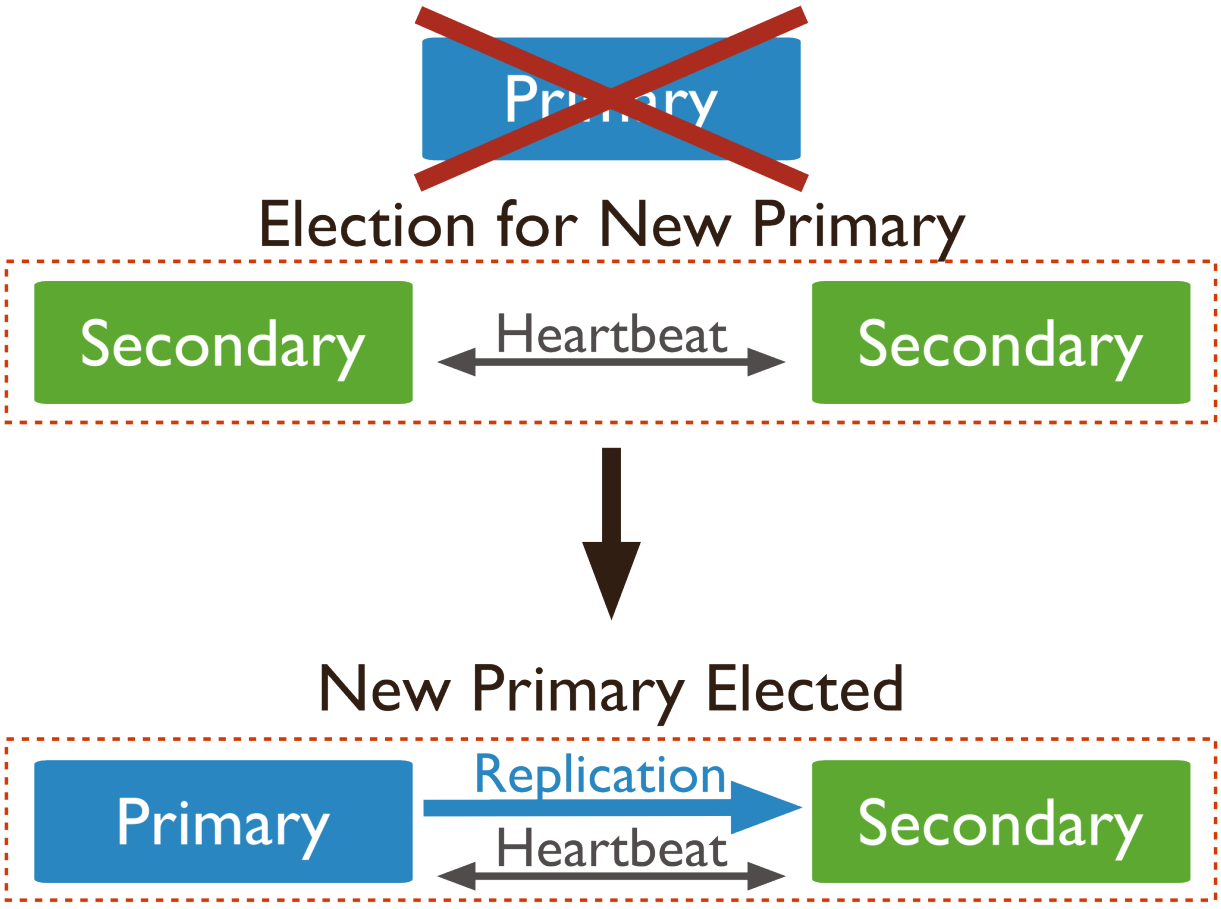
Behavior Elections are essential for independent operation of a replica set; however, elections take time to complete. While an election is in process, the replica set has no primary and cannot accept writes and all remaining members become read-only. MongoDB avoids elections unless necessary.

If a majority of the replica set is inaccessible or unavailable, the replica set cannot accept writes and all remaining members become read-only.

Factors and Conditions that Affect Elections

Heartbeats Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds, the other members mark the delinquent member as inaccessible.

[3] Replica sets remove "rollback" data when needed without intervention. Administrators must apply or discard rollback data manually.


Primary
Election for New Primary
Secondary
Heartbeat
Secondary
New Primary Elected
Primary
Replication
Heartbeat
Secondary

Priority Comparisons The `priority` (page 95) setting affects elections. Members will prefer to vote for members with the highest priority value.

Members with a priority value of `0` cannot become primary and do not seek election. For details, see *Priority 0 Replica Set Members* (page 11).

A replica set does *not* hold an election as long as the current primary has the highest priority value or no secondary with higher priority is within 10 seconds of the latest *oplog* entry in the set.

If a higher-priority member catches up to within 10 seconds of the latest oplog entry of the current primary, the set holds an election in order to provide the higher-priority node a chance to become primary.

Optime The `optime` is the timestamp of the last operation that a member applied from the oplog. A replica set member cannot become primary unless it has the highest (i.e. most recent) `optime` of any visible member in the set.

Connections A replica set member cannot become primary unless it can connect to a majority of the members in the replica set. For the purposes of elections, a majority refers to the total number of *votes*, rather than the total number of members.

If you have a three-member replica set, where every member has one vote, the set can elect a primary as long as two members can connect to each other. If two members are unavailable, the remaining member remains a *secondary* because it cannot connect to a majority of the set's members. If the remaining member is a *primary* and two members become unavailable, the primary steps down and becomes and secondary.

Network Partitions Network partitions affect the formation of a majority for an election. If a primary steps down and neither portion of the replica set has a majority the set will **not** elect a new primary. The replica set becomes read-only.

To avoid this situation, place a majority of instances in one data center and a minority of instances in any other data centers combined.

Election Mechanics

Election Triggering Events Replica sets hold an election any time there is no primary. Specifically, the following:

- the initiation of a new replica set.
- a secondary loses contact with a primary. Secondaries call for elections when they cannot see a primary.
- a primary steps down.

Note: *Priority 0 members* (page 11), do not trigger elections, even when they cannot connect to the primary.

A primary will step down:

- after receiving the `replSetStepDown` command.
- if one of the current secondaries is eligible for election *and* has a higher priority.
- if primary cannot contact a majority of the members of the replica set.

In some cases, modifying a replica set's configuration will trigger an election by modifying the set so that the primary must step down.

Important: When a primary steps down, it closes all open client connections, so that clients don't attempt to write data to a secondary. This helps clients maintain an accurate view of the replica set and helps prevent *rollbacks*.

Participation in Elections Every replica set member has a *priority* that helps determine its eligibility to become a *primary*. In an election, the replica set elects an eligible member with the highest `priority` (page 95) value as primary. By default, all members have a priority of `1` and have an equal chance of becoming primary. In the default, all members also can trigger an election.

You can set the `priority` (page 95) value to weight the election in favor of a particular member or group of members. For example, if you have a *geographically distributed replica set* (page 21), you can adjust priorities so that only members in a specific data center can become primary.

The first member to receive the majority of votes becomes primary. By default, all members have a single vote, unless you modify the `votes` (page 96) setting. *Non-voting members* (page 66) have `votes` (page 96) value of `0`. All other members have `1` vote.

Note: Deprecated since version 2.6: `votes` (page 96) values greater than `1`.

Earlier versions of MongoDB allowed a member to have more than `1` vote by setting `votes` (page 96) to a value greater than `1`. Setting `votes` (page 96) to value greater than `1` now produces a warning message.

The `state` of a member also affects its eligibility to vote. Only members in the following states can vote: `PRIMARY`, `SECONDARY`, `RECOVERING`, `ARBITER`, and `ROLLBACK`.

Important: Do not alter the number of votes in a replica set to control the outcome of an election. Instead, modify the `priority` (page 95) value.

Vetoes in Elections All members of a replica set can veto an election, including *non-voting members* (page 25). A member will veto an election:

- If the member seeking an election is not a member of the voter's set.
- If the member seeking an election is not up-to-date with the most recent operation accessible in the replica set.
- If the member seeking an election has a lower priority than another member in the set that is also eligible for election.
- If a *priority 0 member* (page 11) [4] is the most current member at the time of the election. In this case, another eligible member of the set will catch up to the state of this secondary member and then attempt to become primary.
- If the current primary has more recent operations (i.e. a higher `optime`) than the member seeking election, from the perspective of the voting member.
- If the current primary has the same or more recent operations (i.e. a higher or equal `optime`) than the member seeking election.

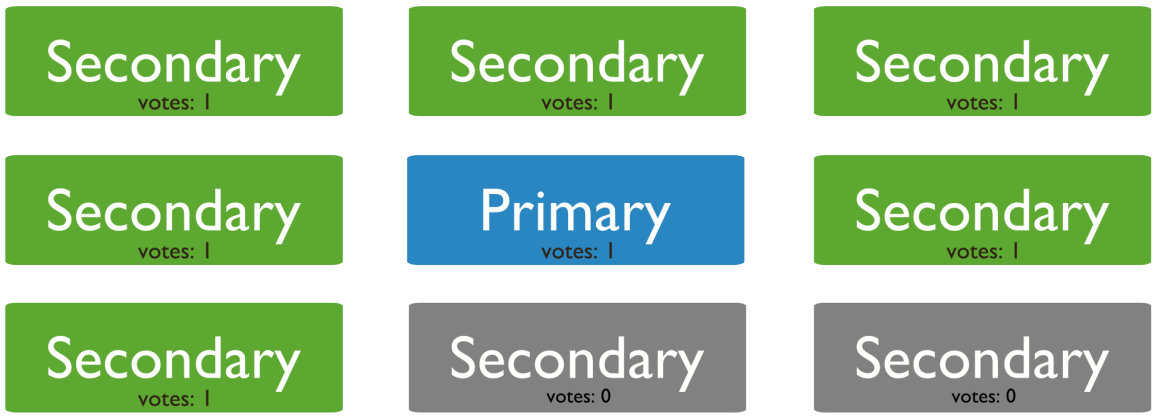
Non-Voting Members Non-voting members hold copies of the replica set's data and can accept read operations from client applications. Non-voting members do not vote in elections, but **can** *veto* (page 25) an election and become primary.

Because a replica set can have up to 12 members but only up to seven voting members, non-voting members allow a replica set to have more than seven members.

For instance, the following nine-member replica set has seven voting members and two non-voting members.

A non-voting member has a `votes` (page 96) setting equal to `0` in its member configuration:

[4] Remember that *hidden* (page 12) and *delayed* (page 13) imply *priority 0* (page 11) configuration.




```
{
  "_id" : <num>
  "host" : <hostname:port>,
  "votes" : 0
}
```

Important: Do **not** alter the number of votes to control which members will become primary. Instead, modify the `priority` (page 95) option. *Only* alter the number of votes in exceptional cases. For example, to permit more than seven members.

When possible, all members should have one vote. Changing the number of votes can cause the wrong members to become primary.

To configure a non-voting member, see *Configure Non-Voting Replica Set Member* (page 66).

Rollbacks During Replica Set Failover

A rollback reverts write operations on a former *primary* when the member rejoins its *replica set* after a *failover*. A rollback is necessary only if the primary had accepted write operations that the *secondaries* had **not** successfully replicated before the primary stepped down. When the primary rejoins the set as a secondary, it reverts, or "rolls back," its write operations to maintain database consistency with the other members.

MongoDB attempts to avoid rollbacks, which should be rare. When a rollback does occur, it is often the result of a network partition. Secondaries that can not keep up with the throughput of operations on the former primary, increase the size and impact of the rollback.

A rollback does *not* occur if the write operations replicate to another member of the replica set before the primary steps down *and* if that member remains available and accessible to a majority of the replica set.

Collect Rollback Data When a rollback does occur, administrators must decide whether to apply or ignore the rollback data. MongoDB writes the rollback data to *BSON* files in the `rollback/` folder under the database's `dbPath` directory. The names of rollback files have the following form:

```
<database>.<collection>.<timestamp>.bson
```

For example:

```
records.accounts.2011-05-09T18-10-04.0.bson
```

Administrators must apply rollback data manually after the member completes the rollback and returns to secondary status. Use `bsondump` to read the contents of the rollback files. Then use `mongorestore` to apply the changes to the new primary.

Avoid Replica Set Rollbacks To prevent rollbacks, use *replica acknowledged write concern* to guarantee that the write operations propagate to the members of a replica set.

Rollback Limitations A `mongod` instance will not rollback more than 300 megabytes of data. If your system must rollback more than 300 megabytes, you must manually intervene to recover the data. If this is the case, the following line will appear in your `mongod` log:

```
[replica set sync] replSet syncThread: 13410 replSet too much data to roll back
```

In this situation, save the data directly or force the member to perform an initial sync. To force initial sync, sync from a "current" member of the set by deleting the content of the `dbPath` directory for the member that requires a larger rollback.

See also:

Replica Set High Availability (page 22) and *Replica Set Elections* (page 22).

2.4 Replica Set Read and Write Semantics

From the perspective of a client application, whether a MongoDB instance is running as a single server (i.e. "standalone") or a *replica set* is transparent.

By default, in MongoDB, read operations to a replica set return results from the *primary* (page 7) and are *consistent* with the last write operation.

Users may configure *read preference* on a per-connection basis to prefer that the read operations return on the *secondary* members. If clients configure the *read preference* to permit secondary reads, read operations can return from *secondary* members that have not replicated more recent updates or operations. When reading from a secondary, a query may return data that reflects a previous state.

This behavior is sometimes characterized as *eventual consistency* because the secondary member's state will *eventually* reflect the primary's state and MongoDB cannot guarantee *strict consistency* for read operations from secondary members.

To guarantee consistency for reads from secondary members, you can configure the *client* and *driver* to ensure that write operations succeed on all members before completing successfully. See `http://docs.mongodb.org/manual/core/write-concern` for more information. Additionally, such configuration can help prevent *Rollbacks During Replica Set Failover* (page 26) during a failover.

Note: *Sharded clusters* where the shards are also replica sets provide the same operational semantics with regards to write and read operations.

Write Concern for Replica Sets **(page 28)** Write concern is the guarantee an application requires from MongoDB to consider a write operation successful.

Read Preference **(page 29)** Applications specify *read preference* to control how drivers direct read operations to members of the replica set.

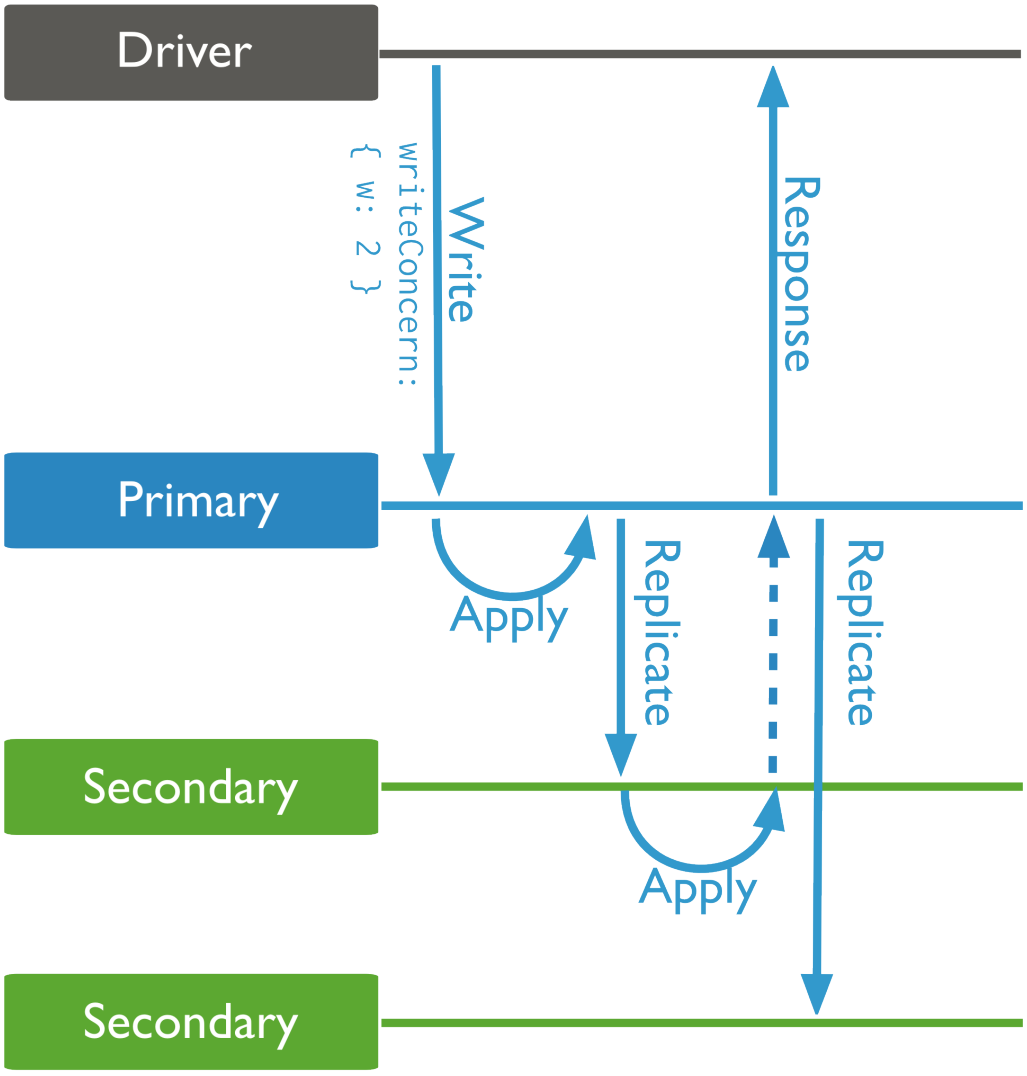
Read Preference Processes **(page 32)** With replica sets, read operations may have additional semantics and behavior.

Write Concern for Replica Sets

From the perspective of a client application, whether a MongoDB instance is running as a single server (i.e. "standalone") or a *replica set* is transparent. However, replica sets offer some configuration options for write. [5]

Verify Write Operations to Replica Sets

For a replica set, the default `write concern` confirms write operations only on the primary. You can, however, override this default write concern, such as to confirm write operations on a specified number of the replica set members.



To override the default write concern, specify a write concern with each write operation. For example, the following method includes a write concern that specifies that the method return only after the write propagates to the primary and at least one secondary or the method times out after 5 seconds.

[5] *Sharded clusters* where the shards are also replica sets provide the same configuration options with regards to write and read operations.

```
db.products.insert(
   { item: "envelopes", qty : 100, type: "Clasp" },
   { writeConcern: { w: 2, wtimeout: 5000 } }
)
```

You can include a timeout threshold for a write concern. This prevents write operations from blocking indefinitely if the write concern is unachievable. For example, if the write concern requires acknowledgement from 4 members of the replica set and the replica set has only available 3 members, the operation blocks until those members become available. See *wc-wtimeout*.

See also:

write-methods-incompatibility

Modify Default Write Concern

You can modify the default write concern for a replica set by setting the `getLastErrorDefaults` (page 97) setting in the *replica set configuration* (page 93). The following sequence of commands creates a configuration that waits for the write operation to complete on a majority of the set members before returning:

```
cfg = rs.conf()
cfg.settings = {}
cfg.settings.getLastErrorDefaults = { w: "majority", wtimeout: 5000 }
rs.reconfig(cfg)
```

If you issue a write operation with a specific write concern, the write operation uses its own write concern instead of the default.

Note: Use of insufficient write concern can lead to *rollbacks* (page 26) in the case of *replica set failover* (page 22). Always ensure that your operations have specified the required write concern for your application.

See also:

write-operations-write-concern and *connections-write-concern*

Custom Write Concerns

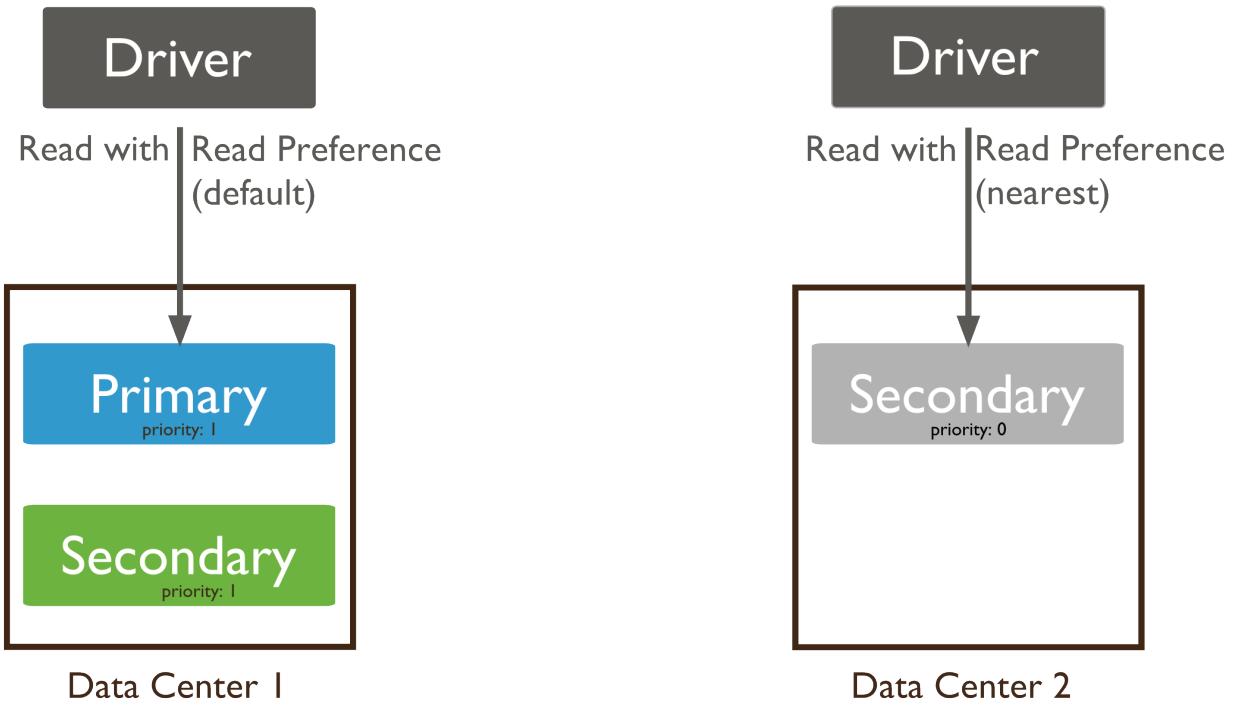
You can *tag* (page 76) the members of replica sets and use the tags to create custom write concerns. See *Configure Replica Set Tag Sets* (page 76) for information on configuring custom write concerns using tag sets.

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. However, by distributing some or all reads to secondary members of the replica set, you can improve read throughput or reduce latency for an application that does not require fully up-to-date data.

Important: You must exercise care when specifying read preferences: modes other than `primary` (page 102) can *and will* return stale data because the secondary queries will not include the most recent write operations to the replica set's *primary*.



Use Cases

Indications The following are common use cases for using non-`primary` (page 102) read preference modes:

- Running systems operations that do not affect the front-end application.

  Note: Read preferences aren't relevant to direct connections to a single `mongod` instance. However, in order to perform read operations on a direct connection to a secondary member of a replica set, you must set a read preference, such as *secondary*.

- Providing local reads for geographically distributed applications.

  If you have application servers in multiple data centers, you may consider having a *geographically distributed replica set* (page 21) and using a non primary read preference or the `nearest` (page 103). This allows the client to read from the lowest-latency members, rather than always reading from the primary.

- Maintaining availability during a failover.

  Use `primaryPreferred` (page 102) if you want an application to read from the primary under normal circumstances, but to allow stale reads from secondaries in an emergency. This provides a "read-only mode" for your application during a failover.

Counter-Indications In general, do *not* use `secondary` (page 102) and `secondaryPreferred` (page 103) to provide extra capacity for reads, because:

- All members of a replica have roughly equivalent write traffic, as a result secondaries will service reads at roughly the same rate as the primary.
- Because replication is asynchronous and there is some amount of delay between a successful write operation and its replication to secondaries, reading from a secondary can return out-of-date data.

- Distributing read operations to secondaries can compromise availability if *any* members of the set are unavailable because the remaining members of the set will need to be able to handle all application requests.
- For queries of sharded collections, for clusters with the *balancer* active, secondaries may return stale results with missing or duplicated data because of incomplete or terminated migrations.

`Sharding` increases read and write capacity by distributing read and write operations across a group of machines, and is often a better strategy for adding capacity.

See *Read Preference Processes* (page 32) for more information about the internal application of read preferences.

Read Preference Modes

New in version 2.2.

Important: All read preference modes except `primary` (page 102) may return stale data because *secondaries* replicate operations from the primary with some delay. Ensure that your application can tolerate stale data if you choose to use a non-`primary` (page 102) mode.

MongoDB `drivers` support five read preference modes.

Read Preference Mode	Description
`primary` (page 102)	Default mode. All operations read from the current replica set *primary*.
`primaryPreferred` (page 102)	In most situations, operations read from the *primary* but if it is unavailable, operations read from *secondary* members.
`secondary` (page 102)	All operations read from the *secondary* members of the replica set.
`secondaryPreferred` (page 103)	In most situations, operations read from *secondary* members but if no *secondary* members are available, operations read from the *primary*.
`nearest` (page 103)	Operations read from member of the *replica set* with the least network latency, irrespective of the member's type.

The syntax for specifying the read preference mode is specific to the driver and to the idioms of the host language[6].

Read preference modes are also available to clients connecting to a *sharded cluster* through a `mongos`. The `mongos` instance obeys specified read preferences when connecting to the *replica set* that provides each *shard* in the cluster.

In the `mongo` shell, the `readPref()` cursor method provides access to read preferences.

For more information, see *read preference background* (page 29) and *read preference behavior* (page 32). See also the documentation for your driver[7].

Tag Sets

Tag sets allow you to target read operations to specific members of a replica set.

Custom read preferences and write concerns evaluate tags sets in different ways. Read preferences consider the value of a tag when selecting a member to read from. Write concerns ignore the value of a tag to when selecting a member, *except* to consider whether or not the value is unique.

You can specify tag sets with the following read preference modes:

- `primaryPreferred` (page 102)
- `secondary` (page 102)

[6] http://api.mongodb.org/
[7] http://api.mongodb.org/

- `secondaryPreferred` (page 103)
- `nearest` (page 103)

Tags are not compatible with mode `primary` (page 102) and, in general, only apply when *selecting* (page 32) a *secondary* member of a set for a read operation. However, the `nearest` (page 103) read mode, when combined with a tag set, selects the matching member with the lowest network latency. This member may be a primary or secondary.

All interfaces use the same *member selection logic* (page 32) to choose the member to which to direct read operations, basing the choice on read preference mode and tag sets.

For information on configuring tag sets, see the *Configure Replica Set Tag Sets* (page 76) tutorial.

For more information on how read preference *modes* (page 102) interact with tag sets, see the *documentation for each read preference mode* (page 102).

Read Preference Processes

Changed in version 2.2.

MongoDB drivers use the following procedures to direct operations to replica sets and sharded clusters. To determine how to route their operations, applications periodically update their view of the replica set's state, identifying which members are up or down, which member is *primary*, and verifying the latency to each `mongod` instance.

Member Selection

Clients, by way of their drivers, and `mongos` instances for sharded clusters, periodically update their view of the replica set's state.

When you select non-`primary` (page 102) read preference, the driver will determine which member to target using the following process:

1. Assembles a list of suitable members, taking into account member type (i.e. secondary, primary, or all members).
2. Excludes members not matching the tag sets, if specified.
3. Determines which suitable member is the closest to the client in absolute terms.
4. Builds a list of members that are within a defined ping distance (in milliseconds) of the "absolute nearest" member.

   Applications can configure the threshold used in this stage. The default "acceptable latency" is 15 milliseconds, which you can override in the drivers with their own `secondaryAcceptableLatencyMS` option. For `mongos` you can use the *`--localThreshold`* or `localPingThresholdMs` runtime options to set this value.
5. Selects a member from these hosts at random. The member receives the read operation.

Drivers can then associate the thread or connection with the selected member. This *request association* (page 32) is configurable by the application. See your `driver` documentation about request association configuration and default behavior.

Request Association

Important: *Request association* is configurable by the application. See your `driver` documentation about request association configuration and default behavior.

Because *secondary* members of a *replica set* may lag behind the current *primary* by different amounts, reads for *secondary* members may reflect data at different points in time. To prevent sequential reads from jumping around in time, the driver **can** associate application threads to a specific member of the set after the first read, thereby preventing reads from other members. The thread will continue to read from the same member until:

- The application performs a read with a different read preference,
- The thread terminates, or
- The client receives a socket exception, as is the case when there's a network error or when the `mongod` closes connections during a *failover*. This triggers a *retry* (page 33), which may be transparent to the application.

When using request association, if the client detects that the set has elected a new *primary*, the driver will discard all associations between threads and members.

Auto-Retry

Connections between MongoDB drivers and `mongod` instances in a *replica set* must balance two concerns:

1. The client should attempt to prefer current results, and any connection should read from the same member of the replica set as much as possible. Requests should prefer *request association* (page 32) (e.g. *pinning*).
2. The client should minimize the amount of time that the database is inaccessible as the result of a connection issue, networking problem, or *failover* in a replica set.

As a result, MongoDB drivers:

- Reuse a connection to a specific `mongod` for as long as possible after establishing a connection to that instance. This connection is *pinned* to this `mongod`.
- Attempt to reconnect to a new member, obeying existing *read preference modes* (page 102), if the connection to `mongod` is lost.

  Reconnections are transparent to the application itself. If the connection permits reads from *secondary* members, after reconnecting, the application can receive two sequential reads returning from different secondaries. Depending on the state of the individual secondary member's replication, the documents can reflect the state of your database at different moments.
- Return an error *only* after attempting to connect to three members of the set that match the *read preference mode* (page 102) and *tag set* (page 31). If there are fewer than three members of the set, the client will error after connecting to all existing members of the set.

  After this error, the driver selects a new member using the specified read preference mode. In the absence of a specified read preference, the driver uses `primary` (page 102).
- After detecting a failover situation, [8] the driver attempts to refresh the state of the replica set as quickly as possible.

Read Preference in Sharded Clusters

Changed in version 2.2: Before version 2.2, `mongos` did not support the *read preference mode semantics* (page 102).

In most *sharded clusters*, each shard consists of a *replica set*. As such, read preferences are also applicable. With regard to read preference, read operations in a sharded cluster are identical to unsharded replica sets.

[8] When a *failover* occurs, all members of the set close all client connections that produce a socket error in the driver. This behavior prevents or minimizes *rollback*.

Unlike simple replica sets, in sharded clusters, all interactions with the shards pass from the clients to the `mongos` instances that are actually connected to the set members. `mongos` is then responsible for the application of read preferences, which is transparent to applications.

There are no configuration changes required for full support of read preference modes in sharded environments, as long as the `mongos` is at least version 2.2. All `mongos` maintain their own connection pool to the replica set members. As a result:

- A request without a specified preference has `primary` (page 102), the default, unless, the `mongos` reuses an existing connection that has a different mode set.

  To prevent confusion, always explicitly set your read preference mode.

- All `nearest` (page 103) and latency calculations reflect the connection between the `mongos` and the `mongod` instances, not the client and the `mongod` instances.

  This produces the desired result, because all results must pass through the `mongos` before returning to the client.

2.5 Replication Processes

Members of a *replica set* replicate data continuously. First, a member uses *initial sync* to capture the data set. Then the member continuously records and applies every operation that modifies the data set. Every member records operations in its *oplog* (page 34), which is a *capped collection*.

Replica Set Oplog **(page 34)** The oplog records all operations that modify the data in the replica set.

Replica Set Data Synchronization **(page 35)** Secondaries must replicate all changes accepted by the primary. This process is the basis of replica set operations.

Replica Set Oplog

The *oplog* (operations log) is a special *capped collection* that keeps a rolling record of all operations that modify the data stored in your databases. MongoDB applies database operations on the *primary* and then records the operations on the primary's oplog. The *secondary* members then copy and apply these operations in an asynchronous process. All replica set members contain a copy of the oplog, in the `local.oplog.rs` (page 99) collection, which allows them to maintain the current state of the database.

To facilitate replication, all replica set members send heartbeats (pings) to all other members. Any member can import oplog entries from any other member.

Whether applied once or multiple times to the target dataset, each operation in the oplog produces the same results, i.e. each operation in the oplog is *idempotent*. For proper replication operations, entries in the oplog must be idempotent:

- initial sync
- post-rollback catch-up
- sharding chunk migrations

Oplog Size

When you start a replica set member for the first time, MongoDB creates an oplog of a default size. The size depends on the architectural details of your operating system.

In most cases, the default oplog size is sufficient. For example, if an oplog is 5% of free disk space and fills up in 24 hours of operations, then secondaries can stop copying entries from the oplog for up to 24 hours without becoming

too stale to continue replicating. However, most replica sets have much lower operation volumes, and their oplogs can hold much higher numbers of operations.

Before `mongod` creates an oplog, you can specify its size with the `oplogSizeMB` option. However, after you have started a replica set member for the first time, you can only change the size of the oplog using the *Change the Size of the Oplog* (page 69) procedure.

By default, the size of the oplog is as follows:

- For 64-bit Linux, Solaris, FreeBSD, and Windows systems, MongoDB allocates 5% of the available free disk space, but will always allocate at least 1 gigabyte and never more than 50 gigabytes.
- For 64-bit OS X systems, MongoDB allocates 183 megabytes of space to the oplog.
- For 32-bit systems, MongoDB allocates about 48 megabytes of space to the oplog.

Workloads that Might Require a Larger Oplog Size

If you can predict your replica set's workload to resemble one of the following patterns, then you might want to create an oplog that is larger than the default. Conversely, if your application predominantly performs reads with a minimal amount of write operations, a smaller oplog may be sufficient.

The following workloads might require a larger oplog size.

Updates to Multiple Documents at Once The oplog must translate multi-updates into individual operations in order to maintain *idempotency*. This can use a great deal of oplog space without a corresponding increase in data size or disk use.

Deletions Equal the Same Amount of Data as Inserts If you delete roughly the same amount of data as you insert, the database will not grow significantly in disk use, but the size of the operation log can be quite large.

Significant Number of In-Place Updates If a significant portion of the workload is in-place updates, the database records a large number of operations but does not change the quantity of data on disk.

Oplog Status

To view oplog status, including the size and the time range of operations, issue the `rs.printReplicationInfo()` method. For more information on oplog status, see *Check the Size of the Oplog* (page 90).

Under various exceptional situations, updates to a *secondary's* oplog might lag behind the desired performance time. Use `db.getReplicationInfo()` from a secondary member and the `replication status` output to assess the current state of replication and determine if there is any unintended replication delay.

See *Replication Lag* (page 88) for more information.

Replica Set Data Synchronization

In order to maintain up-to-date copies of the shared data set, members of a replica set *sync* or replicate data from other members. MongoDB uses two forms of data synchronization: *initial sync* (page 36) to populate new members with the full data set, and replication to apply ongoing changes to the entire data set.

Initial Sync

Initial sync copies all the data from one member of the replica set to another member. A member uses initial sync when the member has no data, such as when the member is new, or when the member has data but is missing a history of the set's replication.

When you perform an initial sync, MongoDB does the following:

1. Clones all databases. To clone, the `mongod` queries every collection in each source database and inserts all data into its own copies of these collections. At this time, _id indexes are also built.
2. Applies all changes to the data set. Using the oplog from the source, the `mongod` updates its data set to reflect the current state of the replica set.
3. Builds all indexes on all collections (except _id indexes, which were already completed).

   When the `mongod` finishes building all index builds, the member can transition to a normal state, i.e. *secondary*.

To perform an initial sync, see *Resync a Member of a Replica Set* (page 75).

Replication

Replica set members replicate data continuously after the initial sync. This process keeps the members up to date with all changes to the replica set's data. In most cases, secondaries synchronize from the primary. Secondaries may automatically change their *sync targets* if needed based on changes in the ping time and state of other members' replication.

For a member to sync from another, the `buildIndexes` (page 95) setting for both members must have the same value/ `buildIndexes` (page 95) must be either `true` or `false` for both members.

Beginning in version 2.2, secondaries avoid syncing from *delayed members* (page 13) and *hidden members* (page 12).

Validity and Durability

In a replica set, only the primary can accept write operations. Writing only to the primary provides *strict consistency* among members.

Journaling provides single-instance write durability. Without journaling, if a MongoDB instance terminates ungracefully, you must assume that the database is in an invalid state.

Multithreaded Replication

MongoDB applies write operations in batches using multiple threads to improve concurrency. MongoDB groups batches by namespace and applies operations using a group of threads, but always applies the write operations to a namespace in order.

While applying a batch, MongoDB blocks all reads. As a result, secondaries can never return data that reflects a state that never existed on the primary.

Pre-Fetching Indexes to Improve Replication Throughput

To help improve the performance of applying oplog entries, MongoDB fetches memory pages that hold affected data and indexes. This *pre-fetch* stage minimizes the amount of time MongoDB holds the write lock while applying oplog entries. By default, secondaries will pre-fetch all *indexes*.

Optionally, you can disable all pre-fetching or only pre-fetch the index on the `_id` field. See the `secondaryIndexPrefetch` setting for more information.

2.6 Master Slave Replication

Important: *Replica sets* (page 5) replace *master-slave* replication for most use cases. If possible, use replica sets rather than master-slave replication for all new production deployments. This documentation remains to support legacy deployments and for archival purposes only.

In addition to providing all the functionality of master-slave deployments, replica sets are also more robust for production use. Master-slave replication preceded replica sets and made it possible to have a large number of non-master (i.e. slave) nodes, as well as to restrict replicated operations to only a single database; however, master-slave replication provides less redundancy and does not automate failover. See *Deploy Master-Slave Equivalent using Replica Sets* (page 39) for a replica set configuration that is equivalent to master-slave replication. If you wish to convert an existing master-slave deployment to a replica set, see *Convert a Master-Slave Deployment to a Replica Set* (page 39).

Fundamental Operations

Initial Deployment

To configure a *master-slave* deployment, start two `mongod` instances: one in master mode, and the other in slave mode.

To start a `mongod` instance in master mode, invoke `mongod` as follows:

```
mongod --master --dbpath /data/masterdb/
```

With the *`--master`* option, the `mongod` will create a `local.oplog.$main` (page 100) collection, which the "operation log" that queues operations that the slaves will apply to replicate operations from the master. The *`--dbpath`* is optional.

To start a `mongod` instance in slave mode, invoke `mongod` as follows:

```
mongod --slave --source <masterhostname><:<port>> --dbpath /data/slavedb/
```

Specify the hostname and port of the master instance to the *`--source`* argument. The *`--dbpath`* is optional.

For slave instances, MongoDB stores data about the source server in the `local.sources` (page 100) collection.

Configuration Options for Master-Slave Deployments

As an alternative to specifying the *`--source`* run-time option, can add a document to `local.sources` (page 100) specifying the master instance, as in the following operation in the `mongo` shell:

```
1 use local
2 db.sources.find()
3 db.sources.insert( { host: <masterhostname> <,only: databasename> } );
```

In line 1, you switch context to the `local` database. In line 2, the `find()` operation should return no documents, to ensure that there are no documents in the `sources` collection. Finally, line 3 uses `db.collection.insert()` to insert the source document into the `local.sources` (page 100) collection. The model of the `local.sources` (page 100) document is as follows:

host
: The host field specifies the master`mongod` instance, and holds a resolvable hostname, i.e. IP address, or a name from a `host` file, or preferably a fully qualified domain name.

  You can append `<:port>` to the host name if the `mongod` is not running on the default `27017` port.

only
: Optional. Specify a name of a database. When specified, MongoDB will only replicate the indicated database.

Operational Considerations for Replication with Master Slave Deployments

Master instances store operations in an *oplog* which is a `capped collection`. As a result, if a slave falls too far behind the state of the master, it cannot "catchup" and must re-sync from scratch. Slave may become out of sync with a master if:

- The slave falls far behind the data updates available from that master.
- The slave stops (i.e. shuts down) and restarts later after the master has overwritten the relevant operations from the master.

When slaves are out of sync, replication stops. Administrators must intervene manually to restart replication. Use the `resync` command. Alternatively, the *`--autoresync`* allows a slave to restart replication automatically, after ten second pause, when the slave falls out of sync with the master. With *`--autoresync`* specified, the slave will only attempt to re-sync once in a ten minute period.

To prevent these situations you should specify a larger oplog when you start the `master` instance, by adding the *`--oplogSize`* option when starting `mongod`. If you do not specify *`--oplogSize`*, `mongod` will allocate 5% of available disk space on start up to the oplog, with a minimum of 1GB for 64bit machines and 50MB for 32bit machines.

Run time Master-Slave Configuration

MongoDB provides a number of command line options for `mongod` instances in *master-slave* deployments. See the *Master-Slave Replication Command Line Options* for options.

Diagnostics

On a *master* instance, issue the following operation in the `mongo` shell to return replication status from the perspective of the master:

```
rs.printReplicationInfo()
```

New in version 2.6: `rs.printReplicationInfo()`. For previous versions, use `db.printReplicationInfo()`.

On a *slave* instance, use the following operation in the `mongo` shell to return the replication status from the perspective of the slave:

```
rs.printSlaveReplicationInfo()
```

New in version 2.6: `rs.printSlaveReplicationInfo()`. For previous versions, use `db.printSlaveReplicationInfo()`.

Use the `serverStatus` as in the following operation, to return status of the replication:

```
db.serverStatus()
```

See *server status repl fields* for documentation of the relevant section of output.

Security

When running with `authorization` enabled, in *master-slave* deployments configure a `keyFile` so that slave `mongod` instances can authenticate and communicate with the master `mongod` instance.

To enable authentication and configure the `keyFile` add the following option to your configuration file:

```
keyFile = /srv/mongodb/keyfile
```

Note: You may chose to set these run-time configuration options using the *`--keyFile`* option on the command line.

Setting `keyFile` enables authentication and specifies a key file for the `mongod` instances to use when authenticating to each other. The content of the key file is arbitrary but must be the same on all members of the deployment can connect to each other.

The key file must be less one kilobyte in size and may only contain characters in the base64 set. The key file must not have group or "world" permissions on UNIX systems. Use the following command to use the OpenSSL package to generate "random" content for use in a key file:

```
openssl rand -base64 741
```

See also:

`http://docs.mongodb.org/manual/security` for more information about security in MongoDB

Ongoing Administration and Operation of Master-Slave Deployments

Deploy Master-Slave Equivalent using Replica Sets

If you want a replication configuration that resembles *master-slave* replication, using *replica sets* replica sets, consider the following replica configuration document. In this deployment hosts `<master>` and `<slave>` [9] provide replication that is roughly equivalent to a two-instance master-slave deployment:

```
{
  _id : 'setName',
  members : [
    { _id : 0, host : "<master>", priority : 1 },
    { _id : 1, host : "<slave>", priority : 0, votes : 0 }
  ]
}
```

See *Replica Set Configuration* (page 93) for more information about replica set configurations.

Convert a Master-Slave Deployment to a Replica Set

To convert a master-slave deployment to a replica set, restart the current master as a one-member replica set. Then remove the data directories from previous secondaries and add them as new secondaries to the new replica set.

1. To confirm that the current instance is master, run:

   ```
   db.isMaster()
   ```

   This should return a document that resembles the following:

[9] In replica set configurations, the `host` (page 94) field must hold a resolvable hostname.

```
{
        "ismaster" : true,
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "localTime" : ISODate("2013-07-08T20:15:13.664Z"),
        "ok" : 1
}
```

2. Shut down the `mongod` processes on the master and all slave(s), using the following command while connected to each instance:

   ```
   db.adminCommand({shutdown : 1, force : true})
   ```

3. Back up your `/data/db` directories, in case you need to revert to the master-slave deployment.
4. Start the former master with the *`--replSet`* option, as in the following:

   ```
   mongod --replSet <setname>
   ```

5. Connect to the `mongod` with the `mongo` shell, and initiate the replica set with the following command:

   ```
   rs.initiate()
   ```

   When the command returns, you will have successfully deployed a one-member replica set. You can check the status of your replica set at any time by running the following command:

   ```
   rs.status()
   ```

You can now follow the *convert a standalone to a replica set* (page 55) tutorial to deploy your replica set, picking up from the *Expand the Replica Set* (page 56) section.

Failing over to a Slave (Promotion)

To permanently failover from a unavailable or damaged *master* (`A` in the following example) to a *slave* (`B`):

1. Shut down `A`.
2. Stop `mongod` on `B`.
3. Back up and move all data files that begin with `local` on `B` from the `dbPath`.

   Warning: Removing `local.*` is irrevocable and cannot be undone. Perform this step with extreme caution.

4. Restart `mongod` on `B` with the *`--master`* option.

Note: This is a one time operation, and is not reversible. `A` cannot become a slave of `B` until it completes a full resync.

Inverting Master and Slave

If you have a *master* (`A`) and a *slave* (`B`) and you would like to reverse their roles, follow this procedure. The procedure assumes `A` is healthy, up-to-date and available.

If `A` is not healthy but the hardware is okay (power outage, server crash, etc.), skip steps 1 and 2 and in step 8 replace all of `A`'s files with `B`'s files in step 8.

If A is not healthy and the hardware is not okay, replace A with a new machine. Also follow the instructions in the previous paragraph.

To invert the master and slave in a deployment:

1. Halt writes on A using the *fsync* command.
2. Make sure B is up to date with the state of A.
3. Shut down B.
4. Back up and move all data files that begin with local on B from the dbPath to remove the existing local.sources data.

   Warning: Removing local.* is irrevocable and cannot be undone. Perform this step with extreme caution.

5. Start B with the *--master* option.
6. Do a write on B, which primes the *oplog* to provide a new sync start point.
7. Shut down B. B will now have a new set of data files that start with local.
8. Shut down A and replace all files in the dbPath of A that start with local with a copy of the files in the dbPath of B that begin with local.

   Considering compressing the local files from B while you copy them, as they may be quite large.

9. Start B with the *--master* option.
10. Start A with all the usual slave options, but include *fastsync*.

Creating a Slave from an Existing Master's Disk Image

If you can stop write operations to the *master* for an indefinite period, you can copy the data files from the master to the new *slave* and then start the slave with *--fastsync*.

Warning: Be careful with *--fastsync*. If the data on both instances is **not** identical, a discrepancy will exist forever.

fastsync is a way to start a slave by starting with an existing master disk image/backup. This option declares that the administrator guarantees the image is correct and completely up-to-date with that of the master. If you have a full and complete copy of data from a master you can use this option to avoid a full synchronization upon starting the slave.

Creating a Slave from an Existing Slave's Disk Image

You can just copy the other *slave's* data file snapshot without any special options. Only take data snapshots when a mongod process is down or locked using db.fsyncLock().

Resyncing a Slave that is too Stale to Recover

Slaves asynchronously apply write operations from the *master* that the slaves poll from the master's *oplog*. The oplog is finite in length, and if a slave is too far behind, a full resync will be necessary. To resync the slave, connect to a slave using the mongo and issue the resync command:

```
use admin
db.runCommand( { resync: 1 } )
```

This forces a full resync of all data (which will be very slow on a large database). You can achieve the same effect by stopping `mongod` on the slave, deleting the entire content of the `dbPath` on the slave, and restarting the `mongod`.

Slave Chaining

Slaves cannot be "chained." They must all connect to the *master* directly.

If a slave attempts "slave from" another slave you will see the following line in the `mongod` long of the shell:

```
assertion 13051 tailable cursor requested on non capped collection ns:local.oplog.$main
```

Correcting a Slave's Source

To change a *slave's* source, manually modify the slave's `local.sources` (page 100) collection.

Example

Consider the following: If you accidentally set an incorrect hostname for the slave's *source*, as in the following example:

```
mongod --slave --source prod.mississippi
```

You can correct this, by restarting the slave without the *--slave* and *--source* arguments:

```
mongod
```

Connect to this `mongod` instance using the `mongo` shell and update the `local.sources` (page 100) collection, with the following operation sequence:

```
use local

db.sources.update( { host : "prod.mississippi" },
                   { $set : { host : "prod.mississippi.example.net" } } )
```

Restart the slave with the correct command line arguments or with no *--source* option. After configuring `local.sources` (page 100) the first time, the *--source* will have no subsequent effect. Therefore, both of the following invocations are correct:

```
mongod --slave --source prod.mississippi.example.net
```

or

```
mongod --slave
```

The slave now polls data from the correct *master*.

---

3 Replica Set Tutorials

The administration of *replica sets* includes the initial deployment of the set, adding and removing members to a set, and configuring the operational parameters and properties of the set. Administrators generally need not intervene in failover or replication processes as MongoDB automates these functions. In the exceptional situations that require

manual interventions, the tutorials in these sections describe processes such as resyncing a member. The tutorials in this section form the basis for all replica set administration.

Replica Set Deployment Tutorials **(page 43)** Instructions for deploying replica sets, as well as adding and removing members from an existing replica set.

Deploy a Replica Set **(page 44)** Configure a three-member replica set for production systems.

Convert a Standalone to a Replica Set **(page 55)** Convert an existing standalone `mongod` instance into a three-member replica set.

Add Members to a Replica Set **(page 57)** Add a new member to an existing replica set.

Remove Members from Replica Set **(page 59)** Remove a member from a replica set.

Continue reading from *Replica Set Deployment Tutorials* (page 43) for additional tutorials of related to setting up replica set deployments.

Member Configuration Tutorials **(page 61)** Tutorials that describe the process for configuring replica set members.

Adjust Priority for Replica Set Member **(page 61)** Change the precedence given to a replica set members in an election for primary.

Prevent Secondary from Becoming Primary **(page 62)** Make a secondary member ineligible for election as primary.

Configure a Hidden Replica Set Member **(page 64)** Configure a secondary member to be invisible to applications in order to support significantly different usage, such as a dedicated backups.

Continue reading from *Member Configuration Tutorials* (page 61) for more tutorials that describe replica set configuration.

Replica Set Maintenance Tutorials **(page 69)** Procedures and tasks for common operations on active replica set deployments.

Change the Size of the Oplog **(page 69)** Increase the size of the *oplog* which logs operations. In most cases, the default oplog size is sufficient.

Resync a Member of a Replica Set **(page 75)** Sync the data on a member. Either perform initial sync on a new member or resync the data on an existing member that has fallen too far behind to catch up by way of normal replication.

Force a Member to Become Primary **(page 73)** Force a replica set member to become primary.

Change Hostnames in a Replica Set **(page 83)** Update the replica set configuration to reflect changes in members' hostnames.

Continue reading from *Replica Set Maintenance Tutorials* (page 69) for descriptions of additional replica set maintenance procedures.

Troubleshoot Replica Sets **(page 88)** Describes common issues and operational challenges for replica sets. For additional diagnostic information, see `http://docs.mongodb.org/manual/faq/diagnostics`.

3.1 Replica Set Deployment Tutorials

The following tutorials provide information in deploying replica sets.

See also:

`http://docs.mongodb.org/manual/administration/security-deployment` for additional related tutorials.

Deploy a Replica Set **(page 44)** Configure a three-member replica set for production systems.

Deploy a Replica Set for Testing and Development **(page 46)** Configure a three-member replica set for either development or testing systems.

Deploy a Geographically Redundant Replica Set **(page 49)** Create a geographically redundant replica set to protect against location-centered availability limitations (e.g. network and power interruptions).

Add an Arbiter to Replica Set **(page 55)** Add an arbiter give a replica set an odd number of voting members to prevent election ties.

Convert a Standalone to a Replica Set **(page 55)** Convert an existing standalone `mongod` instance into a three-member replica set.

Add Members to a Replica Set **(page 57)** Add a new member to an existing replica set.

Remove Members from Replica Set **(page 59)** Remove a member from a replica set.

Replace a Replica Set Member **(page 61)** Update the replica set configuration when the hostname of a member's corresponding `mongod` instance has changed.

Deploy a Replica Set

This tutorial describes how to create a three-member *replica set* from three existing `mongod` instances running with `access control` disabled.

To deploy a replica set with enabled `access control`, see `http://docs.mongodb.org/manual/tutorial/deploy-rep`
If you wish to deploy a replica set from a single MongoDB instance, see *Convert a Standalone to a Replica Set* (page 55). For more information on replica set deployments, see the *Replication* (page 2) and *Replica Set Deployment Architectures* (page 15) documentation.

Overview

Three member *replica sets* provide enough redundancy to survive most network partitions and other system failures. These sets also have sufficient capacity for many distributed read operations. Replica sets should always have an odd number of members. This ensures that *elections* (page 22) will proceed smoothly. For more about designing replica sets, see *the Replication overview* (page 3).

The basic procedure is to start the `mongod` instances that will become members of the replica set, configure the replica set itself, and then add the `mongod` instances to it.

Requirements

For production deployments, you should maintain as much separation between members as possible by hosting the `mongod` instances on separate machines. When using virtual machines for production deployments, you should place each `mongod` instance on a separate host server serviced by redundant power circuits and redundant network paths.

Before you can deploy a replica set, you must install MongoDB on each system that will be part of your *replica set*. If you have not already installed MongoDB, see the *installation tutorials*.

Before creating your replica set, you should verify that your network configuration allows all possible connections between each member. For a successful replica set deployment, every member must be able to connect to every other member. For instructions on how to check your connection, see *Test Connections Between all Members* (page 89).

Considerations When Deploying a Replica Set

Architecture In a production, deploy each member of the replica set to its own machine and if possible bind to the standard MongoDB port of `27017`. Use the `bind_ip` option to ensure that MongoDB listens for connections from applications on configured addresses.

For a geographically distributed replica sets, ensure that the majority of the set's `mongod` instances reside in the primary site.

See *Replica Set Deployment Architectures* (page 15) for more information.

Connectivity Ensure that network traffic can pass between all members of the set and all clients in the network securely and efficiently. Consider the following:

- Establish a virtual private network. Ensure that your network topology routes all traffic between members within a single site over the local area network.
- Configure access control to prevent connections from unknown clients to the replica set.
- Configure networking and firewall rules so that incoming and outgoing packets are permitted only on the default MongoDB port and only from within your deployment.

Finally ensure that each member of a replica set is accessible by way of resolvable DNS or hostnames. You should either configure your DNS names appropriately or set up your systems' `/etc/hosts` file to reflect this configuration.

Configuration Specify the run time configuration on each system in a `configuration file` stored in `/etc/mongodb.conf` or a related location. Create the directory where MongoDB stores data files before deploying MongoDB.

For more information about the run time options used above and other configuration options, see `http://docs.mongodb.org/manual/reference/configuration-options`.

Procedure

The following procedure outlines the steps to deploy a replica set when access control is disabled.

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod` and specify the replica set name through the `replSet` option. Specify any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies the replica set name through the *`--replSet`* command-line option:

```
mongod --replSet "rs0"
```

You can also specify the `replica set name` in the `configuration file`. To start `mongod` with a configuration file, specify the file with the *`--config`* option:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *control script* to manage this process. Control scripts are beyond the scope of this document.

Step 2: Connect a mongo shell to a replica set member. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 4: Verify the initial replica set configuration. Use `rs.conf()` to display the *replica set configuration object* (page 93):

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 5: Add the remaining members to the replica set. Add the remaining members with the `rs.add()` method.

The following example adds two members:

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
```

When complete, you have a fully functional replica set. The new replica set will elect a *primary*.

Step 6: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

See also:

`http://docs.mongodb.org/manual/tutorial/deploy-replica-set-with-auth`

Deploy a Replica Set for Testing and Development

This procedure describes deploying a replica set in a development or test environment. For a production deployment, refer to the *Deploy a Replica Set* (page 44) tutorial.

This tutorial describes how to create a three-member *replica set* from three existing `mongod` instances running with `access control` disabled.

To deploy a replica set with enabled `access control`, see `http://docs.mongodb.org/manual/tutorial/deploy-rep`
If you wish to deploy a replica set from a single MongoDB instance, see *Convert a Standalone to a Replica Set*

(page 55). For more information on replica set deployments, see the *Replication* (page 2) and *Replica Set Deployment Architectures* (page 15) documentation.

Overview

Three member *replica sets* provide enough redundancy to survive most network partitions and other system failures. These sets also have sufficient capacity for many distributed read operations. Replica sets should always have an odd number of members. This ensures that *elections* (page 22) will proceed smoothly. For more about designing replica sets, see *the Replication overview* (page 3).

The basic procedure is to start the `mongod` instances that will become members of the replica set, configure the replica set itself, and then add the `mongod` instances to it.

Requirements

For test and development systems, you can run your `mongod` instances on a local system, or within a virtual instance.

Before you can deploy a replica set, you must install MongoDB on each system that will be part of your *replica set*. If you have not already installed MongoDB, see the *installation tutorials*.

Before creating your replica set, you should verify that your network configuration allows all possible connections between each member. For a successful replica set deployment, every member must be able to connect to every other member. For instructions on how to check your connection, see *Test Connections Between all Members* (page 89).

Considerations

Replica Set Naming

Important: These instructions should only be used for test or development deployments.

The examples in this procedure create a new replica set named `rs0`.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

You will begin by starting three `mongod` instances as members of a replica set named `rs0`.

Procedure

1. Create the necessary data directories for each member by issuing a command similar to the following:

   ```
   mkdir -p /srv/mongodb/rs0-0 /srv/mongodb/rs0-1 /srv/mongodb/rs0-2
   ```

   This will create directories called “rs0-0”, “rs0-1”, and “rs0-2”, which will contain the instances’ database files.

2. Start your `mongod` instances in their own shell windows by issuing the following commands:

   First member:

   ```
   mongod --port 27017 --dbpath /srv/mongodb/rs0-0 --replSet rs0 --smallfiles --oplogSize 128
   ```

   Second member:

   ```
   mongod --port 27018 --dbpath /srv/mongodb/rs0-1 --replSet rs0 --smallfiles --oplogSize 128
   ```

   Third member:

```
mongod --port 27019 --dbpath /srv/mongodb/rs0-2 --replSet rs0 --smallfiles --oplogSize 128
```

This starts each instance as a member of a replica set named `rs0`, each running on a distinct port, and specifies the path to your data directory with the *`--dbpath`* setting. If you are already using the suggested ports, select different ports.

The *`--smallfiles`* and *`--oplogSize`* settings reduce the disk space that each `mongod` instance uses. This is ideal for testing and development deployments as it prevents overloading your machine. For more information on these and other configuration options, see `http://docs.mongodb.org/manual/reference/configuration-options`.

3. Connect to one of your `mongod` instances through the `mongo` shell. You will need to indicate which instance by specifying its port number. For the sake of simplicity and clarity, you may want to choose the first one, as in the following command;

   ```
   mongo --port 27017
   ```

4. In the `mongo` shell, use `rs.initiate()` to initiate the replica set. You can create a replica set configuration object in the `mongo` shell environment, as in the following example:

   ```
   rsconf = {
              _id: "rs0",
              members: [
                         {
                          _id: 0,
                          host: "<hostname>:27017"
                         }
                        ]
            }
   ```

   replacing `<hostname>` with your system's hostname, and then pass the `rsconf` file to `rs.initiate()` as follows:

   ```
   rs.initiate( rsconf )
   ```

5. Display the current *replica configuration* (page 93) by issuing the following command:

   ```
   rs.conf()
   ```

   The replica set configuration object resembles the following

   ```
   {
      "_id" : "rs0",
      "version" : 4,
      "members" : [
         {
            "_id" : 1,
            "host" : "localhost:27017"
         }
      ]
   }
   ```

6. In the `mongo` shell connected to the *primary*, add the second and third `mongod` instances to the replica set using the `rs.add()` method. Replace `<hostname>` with your system's hostname in the following examples:

   ```
   rs.add("<hostname>:27018")
   rs.add("<hostname>:27019")
   ```

   When complete, you should have a fully functional replica set. The new replica set will elect a *primary*.

Check the status of your replica set at any time with the `rs.status()` operation.

See also:

The documentation of the following shell functions for more information:

- `rs.initiate()`
- `rs.conf()`
- `rs.reconfig()`
- `rs.add()`

You may also consider the simple setup script[10] as an example of a basic automatically-configured replica set.

Refer to *Replica Set Read and Write Semantics* (page 27) for a detailed explanation of read and write semantics in MongoDB.

Deploy a Geographically Redundant Replica Set

Overview

This tutorial outlines the process for deploying a *replica set* with members in multiple locations. The tutorial addresses three-member sets, four-member sets, and sets with more than four members.

For appropriate background, see *Replication* (page 2) and *Replica Set Deployment Architectures* (page 15). For related tutorials, see *Deploy a Replica Set* (page 44) and *Add Members to a Replica Set* (page 57).

Considerations

While *replica sets* provide basic protection against single-instance failure, replica sets whose members are all located in a single facility are susceptible to errors in that facility. Power outages, network interruptions, and natural disasters are all issues that can affect replica sets whose members are colocated. To protect against these classes of failures, deploy a replica set with one or more members in a geographically distinct facility or data center to provide redundancy.

Prerequisites

In general, the requirements for any geographically redundant replica set are as follows:

- Ensure that a majority of the *voting members* (page 25) are within a primary facility, "Site A". This includes *priority 0 members* (page 11) and *arbiters* (page 14). Deploy other members in secondary facilities, "Site B", "Site C", etc., to provide additional copies of the data. See *Determine the Distribution of Members* (page 16) for more information on the voting requirements for geographically redundant replica sets.
- If you deploy a replica set with an even number of members, deploy an *arbiter* (page 14) on Site A. The arbiter must be on site A to keep the majority there.

For instance, for a three-member replica set you need two instances in a Site A, and one member in a secondary facility, Site B. Site A should be the same facility or very close to your primary application infrastructure (i.e. application servers, caching layer, users, etc.)

A four-member replica set should have at least two members in Site A, with the remaining members in one or more secondary sites, as well as a single *arbiter* in Site A.

[10] https://github.com/mongodb/mongo-snippets/blob/master/replication/simple-setup.py

For all configurations in this tutorial, deploy each replica set member on a separate system. Although you may deploy more than one replica set member on a single system, doing so reduces the redundancy and capacity of the replica set. Such deployments are typically for testing purposes and beyond the scope of this tutorial.

This tutorial assumes you have installed MongoDB on each system that will be part of your replica set. If you have not already installed MongoDB, see the *installation tutorials*.

Procedures

General Considerations

Architecture In a production, deploy each member of the replica set to its own machine and if possible bind to the standard MongoDB port of `27017`. Use the `bind_ip` option to ensure that MongoDB listens for connections from applications on configured addresses.

For a geographically distributed replica sets, ensure that the majority of the set's `mongod` instances reside in the primary site.

See *Replica Set Deployment Architectures* (page 15) for more information.

Connectivity Ensure that network traffic can pass between all members of the set and all clients in the network securely and efficiently. Consider the following:

- Establish a virtual private network. Ensure that your network topology routes all traffic between members within a single site over the local area network.
- Configure access control to prevent connections from unknown clients to the replica set.
- Configure networking and firewall rules so that incoming and outgoing packets are permitted only on the default MongoDB port and only from within your deployment.

Finally ensure that each member of a replica set is accessible by way of resolvable DNS or hostnames. You should either configure your DNS names appropriately or set up your systems' `/etc/hosts` file to reflect this configuration.

Configuration Specify the run time configuration on each system in a `configuration file` stored in `/etc/mongodb.conf` or a related location. Create the directory where MongoDB stores data files before deploying MongoDB.

For more information about the run time options used above and other configuration options, see `http://docs.mongodb.org/manual/reference/configuration-options`.

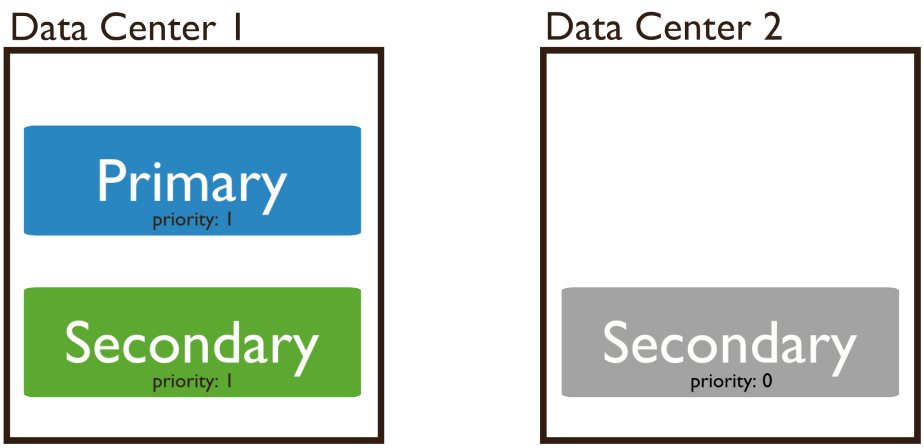
Deploy a Geographically Redundant Three-Member Replica Set

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod` and specify the replica set name through the `replSet` option. Specify any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies the replica set name through the *`--replSet`* command-line option:

```
mongod --replSet "rs0"
```



You can also specify the `replica set name` in the `configuration file`. To start `mongod` with a configuration file, specify the file with the *`--config`* option:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *control script* to manage this process. Control scripts are beyond the scope of this document.

Step 2: Connect a `mongo` shell to a replica set member. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 4: Verify the initial replica set configuration. Use `rs.conf()` to display the *replica set configuration object* (page 93):

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 5: Add the remaining members to the replica set. Add the remaining members with the `rs.add()` method.

The following example adds two members:

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
```

When complete, you have a fully functional replica set. The new replica set will elect a *primary*.

Step 7: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

Step 6: Configure the outside member as *priority 0 members*. Configure the member located in Site B (in this example, `mongodb2.example.net`) as a *priority 0 member* (page 11).

1. View the replica set configuration to determine the `members` (page 94) array position for the member. Keep in mind the array position is not the same as the `_id`:

   ```
   rs.conf()
   ```

2. Copy the replica set configuration object to a variable (to `cfg` in the example below). Then, in the variable, set the correct priority for the member. Then pass the variable to `rs.reconfig()` to update the replica set configuration.

   For example, to set priority for the third member in the array (i.e., the member at position 2), issue the following sequence of commands:

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

   Note: The `rs.reconfig()` shell method can force the current primary to step down, causing an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While most elections complete within a minute, always make sure any replica configuration changes occur during scheduled maintenance periods.

After these commands return, you have a geographically redundant three-member replica set.

Deploy a Geographically Redundant Four-Member Replica Set A geographically redundant four-member deployment has two additional considerations:

- One host (e.g. `mongodb4.example.net`) must be an *arbiter*. This host can run on a system that is also used for an application server or on the same machine as another MongoDB process.
- You must decide how to distribute your systems. There are three possible architectures for the four-member replica set:
  - Three members in Site A, one *priority 0 member* (page 11) in Site B, and an arbiter in Site A.
  - Two members in Site A, two *priority 0 members* (page 11) in Site B, and an arbiter in Site A.
  - Two members in Site A, one priority 0 member in Site B, one priority 0 member in Site C, and an arbiter in site A.

  In most cases, the first architecture is preferable because it is the least complex.

To deploy a geographically redundant four-member set:

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod` and specify the replica set name through the `replSet` option. Specify any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies the replica set name through the *`--replSet`* command-line option:

```
mongod --replSet "rs0"
```

You can also specify the `replica set name` in the `configuration file`. To start `mongod` with a configuration file, specify the file with the *`--config`* option:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *control script* to manage this process. Control scripts are beyond the scope of this document.

Step 2: Connect a `mongo` shell to a replica set member. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 4: Verify the initial replica set configuration. Use `rs.conf()` to display the *replica set configuration object* (page 93):

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 8: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

Step 5: Add the remaining members to the replica set. Use `rs.add()` in a `mongo` shell connected to the current primary. The commands should resemble the following:

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
rs.add("mongodb3.example.net")
```

When complete, you should have a fully functional replica set. The new replica set will elect a *primary*.

Step 6: Add the arbiter. In the same shell session, issue the following command to add the arbiter (e.g. `mongodb4.example.net`):

```
rs.addArb("mongodb4.example.net")
```

Step 7: Configure outside members as *priority 0 members*. Configure each member located outside of Site A (e.g. `mongodb3.example.net`) as a *priority 0 member* (page 11).

1. View the replica set configuration to determine the `members` (page 94) array position for the member. Keep in mind the array position is not the same as the `_id`:

   ```
   rs.conf()
   ```

2. Copy the replica set configuration object to a variable (to `cfg` in the example below). Then, in the variable, set the correct priority for the member. Then pass the variable to `rs.reconfig()` to update the replica set configuration.

   For example, to set priority for the third member in the array (i.e., the member at position 2), issue the following sequence of commands:

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

   Note: The `rs.reconfig()` shell method can force the current primary to step down, causing an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While most elections complete within a minute, always make sure any replica configuration changes occur during scheduled maintenance periods.

After these commands return, you have a geographically redundant four-member replica set.

Deploy a Geographically Redundant Set with More than Four Members The above procedures detail the steps necessary for deploying a geographically redundant replica set. Larger replica set deployments follow the same steps, but have additional considerations:

- Never deploy more than seven voting members.
- If you have an even number of members, use *the procedure for a four-member set* (page 52)). Ensure that a single facility, "Site A", always has a majority of the members by deploying the *arbiter* in that site. For example, if a set has six members, deploy at least three voting members in addition to the arbiter in Site A, and the remaining members in alternate sites.
- If you have an odd number of members, use *the procedure for a three-member set* (page 50). Ensure that a single facility, "Site A" always has a majority of the members of the set. For example, if a set has five members, deploy three members within Site A and two members in other facilities.
- If you have a majority of the members of the set *outside* of Site A and the network partitions to prevent communication between sites, the current primary in Site A will step down, even if none of the members outside of Site A are eligible to become primary.

Add an Arbiter to Replica Set

Arbiters are `mongod` instances that are part of a *replica set* but do not hold data. Arbiters participate in *elections* (page 22) in order to break ties. If a replica set has an even number of members, add an arbiter.

Arbiters have minimal resource requirements and do not require dedicated hardware. You can deploy an arbiter on an application server or a monitoring host.

Important: Do not run an arbiter on the same system as a member of the replica set.

Considerations

An arbiter does not store data, but until the arbiter's `mongod` process is added to the replica set, the arbiter will act like any other `mongod` process and start up with a set of data files and with a full-sized *journal*.

To minimize the default creation of data, set the following in the arbiter's `configuration file`:

- `journal.enabled` to `false`

  Warning: Never set `journal.enabled` to `false` on a data-bearing node.

- `smallFiles` to `true`

These settings are specific to arbiters. Do not set `journal.enabled` to `false` on a data-bearing node. Similarly, do not set `smallFiles` unless specifically indicated.

Add an Arbiter

1. Create a data directory (e.g. `dbPath`) for the arbiter. The `mongod` instance uses the directory for configuration data. The directory *will not* hold the data set. For example, create the `/data/arb` directory:

   ```
   mkdir /data/arb
   ```

2. Start the arbiter. Specify the data directory and the replica set name. The following, starts an arbiter using the `/data/arb` `dbPath` for the `rs` replica set:

   ```
   mongod --port 30000 --dbpath /data/arb --replSet rs
   ```

3. Connect to the primary and add the arbiter to the replica set. Use the `rs.addArb()` method, as in the following example:

   ```
   rs.addArb("m1.example.net:30000")
   ```

   This operation adds the arbiter running on port `30000` on the `m1.example.net` host.

Convert a Standalone to a Replica Set

This tutorial describes the process for converting a *standalone* `mongod` instance into a three-member *replica set*. Use standalone instances for testing and development, but always use replica sets in production. To install a standalone instance, see the *installation tutorials*.

To deploy a replica set without using a pre-existing `mongod` instance, see *Deploy a Replica Set* (page 44).

Procedure

1. Shut down the *standalone* `mongod` instance.
2. Restart the instance. Use the `--replSet` option to specify the name of the new replica set.

   For example, the following command starts a standalone instance as a member of a new replica set named `rs0`. The command uses the standalone's existing database path of `/srv/mongodb/db0`:

   ```
   mongod --port 27017 --dbpath /srv/mongodb/db0 --replSet rs0
   ```

   If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

   For more information on configuration options, see `http://docs.mongodb.org/manual/reference/configurati` and the `mongod` manual page.
3. Connect to the `mongod` instance.
4. Use `rs.initiate()` to initiate the new replica set:

   ```
   rs.initiate()
   ```

   The replica set is now operational.

   To view the replica set configuration, use `rs.conf()`. To check the status of the replica set, use `rs.status()`.

Expand the Replica Set Add additional replica set members by doing the following:

1. On two distinct systems, start two new standalone `mongod` instances. For information on starting a standalone instance, see the *installation tutorial* specific to your environment.
2. On your connection to the original `mongod` instance (the former standalone instance), issue a command in the following form for each new instance to add to the replica set:

   ```
   rs.add("<hostname><:port>")
   ```

   Replace `<hostname>` and `<port>` with the resolvable hostname and port of the `mongod` instance to add to the set. For more information on adding a host to a replica set, see *Add Members to a Replica Set* (page 57).

Sharding Considerations If the new replica set is part of a *sharded cluster*, change the shard host information in the *config database* by doing the following:

1. Connect to one of the sharded cluster's `mongos` instances and issue a command in the following form:

   ```
   db.getSiblingDB("config").shards.save( {_id: "<name>", host: "<replica-set>/<member,><member,><.
   ```

   Replace `<name>` with the name of the shard. Replace `<replica-set>` with the name of the replica set. Replace `<member,><member,><>` with the list of the members of the replica set.
2. Restart all `mongos` instances. If possible, restart all components of the replica sets (i.e., all `mongos` and all shard `mongod` instances).

Add Members to a Replica Set

Overview

This tutorial explains how to add an additional member to an existing *replica set*. For background on replication deployment patterns, see the *Replica Set Deployment Architectures* (page 15) document.

Maximum Voting Members A replica set can have a maximum of seven *voting members* (page 22). To add a member to a replica set that already has seven votes, you must either add the member as a *non-voting member* (page 25) or remove a vote from an `existing member` (page 96).

Control Scripts In production deployments you can configure a *control script* to manage member processes.

Existing Members You can use these procedures to add new members to an existing set. You can also use the same procedure to "re-add" a removed member. If the removed member's data is still relatively recent, it can recover and catch up easily.

Data Files If you have a backup or snapshot of an existing member, you can move the data files (e.g. the `dbPath` directory) to a new system and use them to quickly initiate a new member. The files must be:

- A valid copy of the data files from a member of the same replica set. See `http://docs.mongodb.org/manual/tutorial/backup-with-filesystem-snapshots` document for more information.

  Important: Always use filesystem snapshots to create a copy of a member of the existing replica set. **Do not** use `mongodump` and `mongorestore` to seed a new replica set member.

- More recent than the oldest operation in the *primary's oplog*. The new member must be able to become current by applying operations from the primary's oplog.

Requirements

1. An active replica set.
2. A new MongoDB system capable of supporting your data set, accessible by the active replica set through the network.

Otherwise, use the MongoDB *installation tutorial* and the *Deploy a Replica Set* (page 44) tutorials.

Procedures

Prepare the Data Directory Before adding a new member to an existing *replica set*, prepare the new member's *data directory* using one of the following strategies:

- Make sure the new member's data directory *does not* contain data. The new member will copy the data from an existing member.

  If the new member is in a *recovering* state, it must exit and become a *secondary* before MongoDB can copy all data as part of the replication process. This process takes time but does not require administrator intervention.

- Manually copy the data directory from an existing member. The new member becomes a secondary member and will catch up to the current state of the replica set. Copying the data over may shorten the amount of time for the new member to become current.

  Ensure that you can copy the data directory to the new member and begin replication within the *window allowed by the oplog* (page 34). Otherwise, the new instance will have to perform an initial sync, which completely resynchronizes the data, as described in *Resync a Member of a Replica Set* (page 75).

  Use `rs.printReplicationInfo()` to check the current state of replica set members with regards to the oplog.

For background on replication deployment patterns, see the *Replica Set Deployment Architectures* (page 15) document.

Add a Member to an Existing Replica Set

1. Start the new `mongod` instance. Specify the data directory and the replica set name. The following example specifies the `/srv/mongodb/db0` data directory and the `rs0` replica set:

   ```
   mongod --dbpath /srv/mongodb/db0 --replSet rs0
   ```

   Take note of the host name and port information for the new `mongod` instance.

   For more information on configuration options, see the `mongod` manual page.

   Optional

   You can specify the data directory and replica set in the `mongo.conf configuration file`, and start the `mongod` with the following command:

   ```
   mongod --config /etc/mongodb.conf
   ```

   ---

2. Connect to the replica set's primary.

   You can only add members while connected to the primary. If you do not know which member is the primary, log into any member of the replica set and issue the `db.isMaster()` command.

3. Use `rs.add()` to add the new member to the replica set. For example, to add a member at host `mongodb3.example.net`, issue the following command:

   ```
   rs.add("mongodb3.example.net")
   ```

   You can include the port number, depending on your setup:

   ```
   rs.add("mongodb3.example.net:27017")
   ```

4. Verify that the member is now part of the replica set. Call the `rs.conf()` method, which displays the *replica set configuration* (page 93):

   ```
   rs.conf()
   ```

   To view replica set status, issue the `rs.status()` method. For a description of the status fields, see `http://docs.mongodb.org/manual/reference/command/replSetGetStatus`.

Configure and Add a Member You can add a member to a replica set by passing to the `rs.add()` method a `members` (page 94) document. The document must be in the form of a `local.system.replset.members` (page 94) document. These documents define a replica set member in the same form as the *replica set configuration document* (page 93).

Important: Specify a value for the `_id` field of the `members` (page 94) document. MongoDB does not automatically

populate the _id field in this case. Finally, the members (page 94) document must declare the host value. All other fields are optional.

Example

To add a member with the following configuration:

- an _id of 1.
- a hostname and port number (page 94) of mongodb3.example.net:27017.
- a priority (page 95) value within the replica set of 0.
- a configuration as hidden (page 95),

Issue the following:

```
rs.add({_id: 1, host: "mongodb3.example.net:27017", priority: 0, hidden: true})
```

Remove Members from Replica Set

To remove a member of a *replica set* use either of the following procedures.

Remove a Member Using rs.remove()

1. Shut down the mongod instance for the member you wish to remove. To shut down the instance, connect using the mongo shell and the db.shutdownServer() method.
2. Connect to the replica set's current *primary*. To determine the current primary, use db.isMaster() while connected to any member of the replica set.
3. Use rs.remove() in either of the following forms to remove the member:

   ```
   rs.remove("mongod3.example.net:27017")
   rs.remove("mongod3.example.net")
   ```

   MongoDB disconnects the shell briefly as the replica set elects a new primary. The shell then automatically reconnects. The shell displays a DBClientCursor::init call() failed error even though the command succeeds.

Remove a Member Using rs.reconfig()

To remove a member you can manually edit the *replica set configuration document* (page 93), as described here.

1. Shut down the mongod instance for the member you wish to remove. To shut down the instance, connect using the mongo shell and the db.shutdownServer() method.
2. Connect to the replica set's current *primary*. To determine the current primary, use db.isMaster() while connected to any member of the replica set.
3. Issue the rs.conf() method to view the current configuration document and determine the position in the members array of the member to remove:

   Example

   mongod_C.example.net is in position 2 of the following configuration file:

```
{
    "_id" : "rs",
    "version" : 7,
    "members" : [
        {
            "_id" : 0,
            "host" : "mongod_A.example.net:27017"
        },
        {
            "_id" : 1,
            "host" : "mongod_B.example.net:27017"
        },
        {
            "_id" : 2,
            "host" : "mongod_C.example.net:27017"
        }
    ]
}
```

4. Assign the current configuration document to the variable `cfg`:

   ```
   cfg = rs.conf()
   ```

5. Modify the `cfg` object to remove the member.

   Example

   To remove `mongod_C.example.net:27017` use the following JavaScript operation:

   ```
   cfg.members.splice(2,1)
   ```

6. Overwrite the replica set configuration document with the new configuration by issuing the following:

   ```
   rs.reconfig(cfg)
   ```

   As a result of `rs.reconfig()` the shell will disconnect while the replica set renegotiates which member is primary. The shell displays a `DBClientCursor::init call() failed` error even though the command succeeds, and will automatically reconnected.

7. To confirm the new configuration, issue `rs.conf()`.

   For the example above the output would be:

   ```
   {
       "_id" : "rs",
       "version" : 8,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongod_A.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongod_B.example.net:27017"
           }
       ]
   }
   ```

Replace a Replica Set Member

If you need to change the hostname of a replica set member without changing the configuration of that member or the set, you can use the operation outlined in this tutorial. For example if you must re-provision systems or rename hosts, you can use this pattern to minimize the scope of that change.

Operation

To change the hostname for a replica set member modify the `host` (page 94) field. The value of `_id` (page 94) field will not change when you reconfigure the set.

See *Replica Set Configuration* (page 93) and `rs.reconfig()` for more information.

Note: Any replica set configuration change can trigger the current *primary* to step down, which forces an *election* (page 22). During the election, the current shell session and clients connected to this replica set disconnect, which produces an error even when the operation succeeds.

Example

To change the hostname to `mongo2.example.net` for the replica set member configured at `members[0]`, issue the following sequence of commands:

```
cfg = rs.conf()
cfg.members[0].host = "mongo2.example.net"
rs.reconfig(cfg)
```

3.2 Member Configuration Tutorials

The following tutorials provide information in configuring replica set members to support specific operations, such as to provide dedicated backups, to support reporting, or to act as a cold standby.

Adjust Priority for Replica Set Member **(page 61)** Change the precedence given to a replica set members in an election for primary.

Prevent Secondary from Becoming Primary **(page 62)** Make a secondary member ineligible for election as primary.

Configure a Hidden Replica Set Member **(page 64)** Configure a secondary member to be invisible to applications in order to support significantly different usage, such as a dedicated backups.

Configure a Delayed Replica Set Member **(page 65)** Configure a secondary member to keep a delayed copy of the data set in order to provide a rolling backup.

Configure Non-Voting Replica Set Member **(page 66)** Create a secondary member that keeps a copy of the data set but does not vote in an election.

Convert a Secondary to an Arbiter **(page 67)** Convert a secondary to an arbiter.

Adjust Priority for Replica Set Member

Overview

The priority settings of replica set members affect the outcomes of *elections* (page 22) for primary. Use this setting to ensure that some members are more likely to become primary and that others can never become primary.

The value of the member's priority (page 95) setting determines the member's priority in elections. The higher the number, the higher the priority.

Considerations

To modify priorities, you update the members (page 94) array in the replica configuration object. The array index begins with 0. Do **not** confuse this index value with the value of the replica set member's _id (page 94) field in the array.

The value of priority (page 95) can be any floating point (i.e. decimal) number between 0 and 1000. The default value for the priority (page 95) field is 1.

To block a member from seeking election as primary, assign it a priority of 0. *Hidden members* (page 12), *delayed members* (page 13), and *arbiters* (page **??**) all have priority (page 95) set to 0.

Adjust priority during a scheduled maintenance window. Reconfiguring priority can force the current primary to step down, leading to an election. Before an election the primary closes all open *client* connections.

Procedure

Step 1: Copy the replica set configuration to a variable. In the mongo shell, use rs.conf() to retrieve the replica set configuration and assign it to a variable. For example:

```
cfg = rs.conf()
```

Step 2: Change each member's priority value. Change each member's priority (page 95) value, as configured in the members (page 94) array.

```
cfg.members[0].priority = 0.5
cfg.members[1].priority = 2
cfg.members[2].priority = 2
```

This sequence of operations modifies the value of cfg to set the priority for the first three members defined in the members (page 94) array.

Step 3: Assign the replica set the new configuration. Use rs.reconfig() to apply the new configuration.

```
rs.reconfig(cfg)
```

This operation updates the configuration of the replica set using the configuration defined by the value of cfg.

Prevent Secondary from Becoming Primary

Overview

In a replica set, by default all *secondary* members are eligible to become primary through the election process. You can use the priority (page 95) to affect the outcome of these elections by making some members more likely to become primary and other members less likely or unable to become primary.

Secondaries that cannot become primary are also unable to trigger elections. In all other respects these secondaries are identical to other secondaries.

To prevent a *secondary* member from ever becoming a *primary* in a *failover*, assign the secondary a priority of `0`, as described here. For a detailed description of secondary-only members and their purposes, see *Priority 0 Replica Set Members* (page 11).

Considerations

When updating the replica configuration object, access the replica set members in the `members` (page 94) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 94) field in each document in the `members` (page 94) array.

Note: MongoDB does not permit the current *primary* to have a priority of `0`. To prevent the current primary from again becoming a primary, you must first step down the current primary using `rs.stepDown()`.

Procedure

This tutorial uses a sample replica set with 5 members.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 22). When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 55) to ensure that members can quickly obtain a majority of votes in an election for primary.

Step 1: Retrieve the current replica set configuration. The `rs.conf()` method returns a *replica set configuration document* (page 93) that contains the current configuration for a replica set.

In a `mongo` shell connected to a primary, run the `rs.conf()` method and assign the result to a variable:

```
cfg = rs.conf()
```

The returned document contains a `members` (page 94) field which contains an array of member configuration documents, one document for each member of the replica set.

Step 2: Assign priority value of `0`. To prevent a secondary member from becoming a primary, update the secondary member's `priority` (page 95) to `0`.

To assign a priority value to a member of the replica set, access the member configuration document using the array index. In this tutorial, the secondary member to change corresponds to the configuration document found at position `2` of the `members` (page 94) array.

```
cfg.members[2].priority = 0
```

The configuration change does not take effect until you reconfigure the replica set.

Step 3: Reconfigure the replica set. Use `rs.reconfig()` method to reconfigure the replica set with the updated replica set configuration document.

Pass the `cfg` variable to the `rs.reconfig()` method:

```
rs.reconfig(cfg)
```

Related Documents

- `priority` (page 95)
- *Adjust Priority for Replica Set Member* (page 61)
- *Replica Set Reconfiguration*
- *Replica Set Elections* (page 22)

Configure a Hidden Replica Set Member

Hidden members are part of a *replica set* but cannot become *primary* and are invisible to client applications. Hidden members do vote in *elections* (page 22). For a more information on hidden members and their uses, see *Hidden Replica Set Members* (page 12).

Considerations

The most common use of hidden nodes is to support *delayed members* (page 13). If you only need to prevent a member from becoming primary, configure a *priority 0 member* (page 11).

If the `chainingAllowed` (page 97) setting allows secondary members to sync from other secondaries, MongoDB by default prefers non-hidden members over hidden members when selecting a sync target. MongoDB will only choose hidden members as a last resort. If you want a secondary to sync from a hidden member, use the `replSetSyncFrom` database command to override the default sync target. See the documentation for `replSetSyncFrom` before using the command.

See also:

Manage Chained Replication (page 82)

Changed in version 2.0: For *sharded clusters* running with replica sets before 2.0, if you reconfigured a member as hidden, you *had* to restart `mongos` to prevent queries from reaching the hidden member.

Examples

Member Configuration Document To configure a secondary member as hidden, set its `priority` (page 95) value to `0` and set its `hidden` (page 95) value to `true` in its member configuration:

```
{
  "_id" : <num>
  "host" : <hostname:port>,
  "priority" : 0,
  "hidden" : true
}
```

Configuration Procedure The following example hides the secondary member currently at the index `0` in the `members` (page 94) array. To configure a *hidden member*, use the following sequence of operations in a `mongo` shell connected to the primary, specifying the member to configure by its array index in the `members` (page 94) array:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
rs.reconfig(cfg)
```

After re-configuring the set, this secondary member has a priority of `0` so that it cannot become primary and is hidden. The other members in the set will not advertise the hidden member in the `isMaster` or `db.isMaster()` output.

When updating the replica configuration object, access the replica set members in the `members` (page 94) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 94) field in each document in the `members` (page 94) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 22). When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 55) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- *Replica Set Reconfiguration*
- *Replica Set Elections* (page 22)
- *Read Preference* (page 29)

Configure a Delayed Replica Set Member

To configure a delayed secondary member, set its `priority` (page 95) value to `0`, its `hidden` (page 95) value to `true`, and its `slaveDelay` (page 96) value to the number of seconds to delay.

Important: The length of the secondary `slaveDelay` (page 96) must fit within the window of the oplog. If the oplog is shorter than the `slaveDelay` (page 96) window, the delayed member cannot successfully replicate operations.

When you configure a delayed member, the delay applies both to replication and to the member's *oplog*. For details on delayed members and their uses, see *Delayed Replica Set Members* (page 13).

Example

The following example sets a 1-hour delay on a secondary member currently at the index `0` in the `members` (page 94) array. To set the delay, issue the following sequence of operations in a `mongo` shell connected to the primary:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].slaveDelay = 3600
rs.reconfig(cfg)
```

After the replica set reconfigures, the delayed secondary member cannot become *primary* and is hidden from applications. The `slaveDelay` (page 96) value delays both replication and the member's *oplog* by 3600 seconds (1 hour).

When updating the replica configuration object, access the replica set members in the `members` (page 94) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 94) field in each document in the `members` (page 94) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 22). When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 55) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- `slaveDelay` (page 96)
- *Replica Set Reconfiguration*
- *Oplog Size* (page 34)
- *Change the Size of the Oplog* (page 69) tutorial
- *Replica Set Elections* (page 22)

Configure Non-Voting Replica Set Member

Non-voting members allow you to add additional members for read distribution beyond the maximum seven voting members. To configure a member as non-voting, set its `votes` (page 96) value to `0`.

Example

To disable the ability to vote in elections for the fourth, fifth, and sixth replica set members, use the following command sequence in the `mongo` shell connected to the primary. You identify each replica set member by its array index in the `members` (page 94) array:

```
cfg = rs.conf()
cfg.members[3].votes = 0
cfg.members[4].votes = 0
cfg.members[5].votes = 0
rs.reconfig(cfg)
```

This sequence gives `0` votes to the fourth, fifth, and sixth members of the set according to the order of the `members` (page 94) array in the output of `rs.conf()`. This setting allows the set to elect these members as *primary* but does not allow them to vote in elections. Place voting members so that your designated primary or primaries can reach a majority of votes in the event of a network partition.

When updating the replica configuration object, access the replica set members in the `members` (page 94) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 94) field in each document in the `members` (page 94) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 22). When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 55) to ensure that members can quickly obtain a majority of votes in an election for primary.

In general and when possible, all members should have only 1 vote. This prevents intermittent ties, deadlocks, or the wrong members from becoming primary. Use `priority` (page 95) to control which members are more likely to become primary.

Related Documents

- `votes` (page 96)
- *Replica Set Reconfiguration*
- *Replica Set Elections* (page 22)

Convert a Secondary to an Arbiter

If you have a *secondary* in a *replica set* that no longer needs to hold data but that needs to remain in the set to ensure that the set can *elect a primary* (page 22), you may convert the secondary to an *arbiter* (page **??**) using either procedure in this tutorial. Both procedures are operationally equivalent:

- You may operate the arbiter on the same port as the former secondary. In this procedure, you must shut down the secondary and remove its data before restarting and reconfiguring it as an arbiter.

  For this procedure, see *Convert Secondary to Arbiter and Reuse the Port Number* (page 67).

- Run the arbiter on a new port. In this procedure, you can reconfigure the server as an arbiter before shutting down the instance running as a secondary.

  For this procedure, see *Convert Secondary to Arbiter Running on a New Port Number* (page 68).

Convert Secondary to Arbiter and Reuse the Port Number

1. If your application is connecting directly to the secondary, modify the application so that MongoDB queries don't reach the secondary.
2. Shut down the secondary.
3. Remove the *secondary* from the *replica set* by calling the `rs.remove()` method. Perform this operation while connected to the current *primary* in the `mongo` shell:

   ```
   rs.remove("<hostname><:port>")
   ```

4. Verify that the replica set no longer includes the secondary by calling the `rs.conf()` method in the `mongo` shell:

   ```
   rs.conf()
   ```

5. Move the secondary's data directory to an archive folder. For example:

```
mv /data/db /data/db-old
```

Optional

You may remove the data instead.

---

6. Create a new, empty data directory to point to when restarting the `mongod` instance. You can reuse the previous name. For example:

   ```
   mkdir /data/db
   ```

7. Restart the `mongod` instance for the secondary, specifying the port number, the empty data directory, and the replica set. You can use the same port number you used before. Issue a command similar to the following:

   ```
   mongod --port 27021 --dbpath /data/db --replSet rs
   ```

8. In the `mongo` shell convert the secondary to an arbiter using the `rs.addArb()` method:

   ```
   rs.addArb("<hostname><:port>")
   ```

9. Verify the arbiter belongs to the replica set by calling the `rs.conf()` method in the `mongo` shell.

   ```
   rs.conf()
   ```

   The arbiter member should include the following:

   ```
   "arbiterOnly" : true
   ```

Convert Secondary to Arbiter Running on a New Port Number

1. If your application is connecting directly to the secondary or has a connection string referencing the secondary, modify the application so that MongoDB queries don't reach the secondary.
2. Create a new, empty data directory to be used with the new port number. For example:

   ```
   mkdir /data/db-temp
   ```

3. Start a new `mongod` instance on the new port number, specifying the new data directory and the existing replica set. Issue a command similar to the following:

   ```
   mongod --port 27021 --dbpath /data/db-temp --replSet rs
   ```

4. In the `mongo` shell connected to the current primary, convert the new `mongod` instance to an arbiter using the `rs.addArb()` method:

   ```
   rs.addArb("<hostname><:port>")
   ```

5. Verify the arbiter has been added to the replica set by calling the `rs.conf()` method in the `mongo` shell.

   ```
   rs.conf()
   ```

   The arbiter member should include the following:

   ```
   "arbiterOnly" : true
   ```

6. Shut down the secondary.
7. Remove the *secondary* from the *replica set* by calling the `rs.remove()` method in the `mongo` shell:

```
rs.remove("<hostname><:port>")
```

8. Verify that the replica set no longer includes the old secondary by calling the `rs.conf()` method in the `mongo` shell:

   ```
   rs.conf()
   ```

9. Move the secondary's data directory to an archive folder. For example:

   ```
   mv /data/db /data/db-old
   ```

Optional

You may remove the data instead.

3.3 Replica Set Maintenance Tutorials

The following tutorials provide information in maintaining existing replica sets.

Change the Size of the Oplog **(page 69)** Increase the size of the *oplog* which logs operations. In most cases, the default oplog size is sufficient.

Perform Maintenance on Replica Set Members **(page 71)** Perform maintenance on a member of a replica set while minimizing downtime.

Force a Member to Become Primary **(page 73)** Force a replica set member to become primary.

Resync a Member of a Replica Set **(page 75)** Sync the data on a member. Either perform initial sync on a new member or resync the data on an existing member that has fallen too far behind to catch up by way of normal replication.

Configure Replica Set Tag Sets **(page 76)** Assign tags to replica set members for use in targeting read and write operations to specific members.

Reconfigure a Replica Set with Unavailable Members **(page 80)** Reconfigure a replica set when a majority of replica set members are down or unreachable.

Manage Chained Replication **(page 82)** Disable or enable chained replication. Chained replication occurs when a secondary replicates from another secondary instead of the primary.

Change Hostnames in a Replica Set **(page 83)** Update the replica set configuration to reflect changes in members' hostnames.

Configure a Secondary's Sync Target **(page 87)** Specify the member that a secondary member synchronizes from.

Change the Size of the Oplog

The *oplog* exists internally as a *capped collection*, so you cannot modify its size in the course of normal operations. In most cases the *default oplog size* (page 34) is an acceptable size; however, in some situations you may need a larger or smaller oplog. For example, you might need to change the oplog size if your applications perform large numbers of multi-updates or deletes in short periods of time.

This tutorial describes how to resize the oplog. For a detailed explanation of oplog sizing, see *Oplog Size* (page 34). For details how oplog size affects *delayed members* and affects *replication lag*, see *Delayed Replica Set Members* (page 13).

Overview

To change the size of the oplog, you must perform maintenance on each member of the replica set in turn. The procedure requires: stopping the `mongod` instance and starting as a standalone instance, modifying the oplog size, and restarting the member.

Important: Always start rolling replica set maintenance with the secondaries, and finish with the maintenance on primary member.

Procedure

- Restart the member in standalone mode.

  Tip

  Always use `rs.stepDown()` to force the primary to become a secondary, before stopping the server. This facilitates a more efficient election process.

- Recreate the oplog with the new size and with an old oplog entry as a seed.
- Restart the `mongod` instance as a member of the replica set.

Restart a Secondary in Standalone Mode on a Different Port Shut down the `mongod` instance for one of the non-primary members of your replica set. For example, to shut down, use the `db.shutdownServer()` method:

```
db.shutdownServer()
```

Restart this `mongod` as a standalone instance running on a different port and *without* the *`--replSet`* parameter. Use a command similar to the following:

```
mongod --port 37017 --dbpath /srv/mongodb
```

Create a Backup of the Oplog (Optional) Optionally, backup the existing oplog on the standalone instance, as in the following example:

```
mongodump --db local --collection 'oplog.rs' --port 37017
```

Recreate the Oplog with a New Size and a Seed Entry Save the last entry from the oplog. For example, connect to the instance using the `mongo` shell, and enter the following command to switch to the `local` database:

```
use local
```

In `mongo` shell scripts you can use the following operation to set the `db` object:

```
db = db.getSiblingDB('local')
```

Ensure that the `temp` temporary collection is empty by dropping the collection:

```
db.temp.drop()
```

Use the `db.collection.save()` method and a sort on reverse *natural order* to find the last entry and save it to a temporary collection:

```
db.temp.save( db.oplog.rs.find( { }, { ts: 1, h: 1 } ).sort( {$natural : -1} ).limit(1).next() )
```

To see this oplog entry, use the following operation:

```
db.temp.find()
```

Remove the Existing Oplog Collection Drop the old `oplog.rs` collection in the `local` database. Use the following command:

```
db = db.getSiblingDB('local')
db.oplog.rs.drop()
```

This returns `true` in the shell.

Create a New Oplog Use the `create` command to create a new oplog of a different size. Specify the `size` argument in bytes. A value of `2 * 1024 * 1024 * 1024` will create a new oplog that's 2 gigabytes:

```
db.runCommand( { create: "oplog.rs", capped: true, size: (2 * 1024 * 1024 * 1024) } )
```

Upon success, this command returns the following status:

```
{ "ok" : 1 }
```

Insert the Last Entry of the Old Oplog into the New Oplog Insert the previously saved last entry from the old oplog into the new oplog. For example:

```
db.oplog.rs.save( db.temp.findOne() )
```

To confirm the entry is in the new oplog, use the following operation:

```
db.oplog.rs.find()
```

Restart the Member Restart the `mongod` as a member of the replica set on its usual port. For example:

```
db.shutdownServer()
mongod --replSet rs0 --dbpath /srv/mongodb
```

The replica set member will recover and "catch up" before it is eligible for election to primary.

Repeat Process for all Members that may become Primary Repeat this procedure for all members you want to change the size of the oplog. Repeat the procedure for the primary as part of the following step.

Change the Size of the Oplog on the Primary To finish the rolling maintenance operation, step down the primary with the `rs.stepDown()` method and repeat the oplog resizing procedure above.

Perform Maintenance on Replica Set Members

Overview

Replica sets allow a MongoDB deployment to remain available during the majority of a maintenance window.

This document outlines the basic procedure for performing maintenance on each of the members of a replica set. Furthermore, this particular sequence strives to minimize the amount of time that the *primary* is unavailable and controlling the impact on the entire deployment.

Use these steps as the basis for common replica set operations, particularly for procedures such as `upgrading to the latest version of MongoDB` and *changing the size of the oplog* (page 69).

Procedure

For each member of a replica set, starting with a secondary member, perform the following sequence of events, ending with the primary:

- Restart the `mongod` instance as a standalone.
- Perform the task on the standalone instance.
- Restart the `mongod` instance as a member of the replica set.

Step 1: Stop a secondary. In the `mongo` shell, shut down the `mongod` instance:

```
db.shutdownServer()
```

Step 2: Restart the secondary as a standalone on a different port. At the operating system shell prompt, restart `mongod` as a standalone instance running on a different port and *without* the `--replSet` parameter:

```
mongod --port 37017 --dbpath /srv/mongodb
```

Always start `mongod` with the same user, even when restarting a replica set member as a standalone instance.

Step 3: Perform maintenance operations on the secondary. While the member is a standalone, use the `mongo` shell to perform maintenance:

```
mongo --port 37017
```

Step 4: Restart `mongod` as a member of the replica set. After performing all maintenance tasks, use the following procedure to restart the `mongod` as a member of the replica set on its usual port.

From the `mongo` shell, shut down the standalone server after completing the maintenance:

```
db.shutdownServer()
```

Restart the `mongod` instance as a member of the replica set using its normal command-line arguments or configuration file.

The secondary takes time to *catch up to the primary* (page 35). From the `mongo` shell, use the following command to verify that the member has caught up from the `RECOVERING` (page 101) state to the `SECONDARY` (page 100) state.

```
rs.status()
```

Step 5: Perform maintenance on the primary last. To perform maintenance on the primary after completing maintenance tasks on all secondaries, use `rs.stepDown()` in the `mongo` shell to step down the primary and allow one of the secondaries to be elected the new primary. Specify a 300 second waiting period to prevent the member from being elected primary again for five minutes:

```
rs.stepDown(300)
```

After the primary steps down, the replica set will elect a new primary. See *Replica Set Elections* (page 22) for more information about replica set elections.

Force a Member to Become Primary

Synopsis

You can force a *replica set* member to become *primary* by giving it a higher `priority` (page 95) value than any other member in the set.

Optionally, you also can force a member never to become primary by setting its `priority` (page 95) value to `0`, which means the member can never seek *election* (page 22) as primary. For more information, see *Priority 0 Replica Set Members* (page 11).

Procedures

Force a Member to be Primary by Setting its Priority High Changed in version 2.0.

For more information on priorities, see `priority` (page 95).

This procedure assumes your current *primary* is `m1.example.net` and that you'd like to instead make `m3.example.net` primary. The procedure also assumes you have a three-member *replica set* with the configuration below. For more information on configurations, see *Replica Set Configuration Use*.

This procedure assumes this configuration:

```
{
    "_id" : "rs",
    "version" : 7,
    "members" : [
        {
            "_id" : 0,
            "host" : "m1.example.net:27017"
        },
        {
            "_id" : 1,
            "host" : "m2.example.net:27017"
        },
        {
            "_id" : 2,
            "host" : "m3.example.net:27017"
        }
    ]
}
```

1. In the `mongo` shell, use the following sequence of operations to make `m3.example.net` the primary:

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

   This sets `m3.example.net` to have a higher `local.system.replset.members[n].priority` (page 95) value than the other `mongod` instances.

The following sequence of events occur:

- `m3.example.net` and `m2.example.net` sync with `m1.example.net` (typically within 10 seconds).
- `m1.example.net` sees that it no longer has highest priority and, in most cases, steps down. `m1.example.net` *does not* step down if `m3.example.net`'s sync is far behind. In that case, `m1.example.net` waits until `m3.example.net` is within 10 seconds of its optime and then steps down. This minimizes the amount of time with no primary following failover.
- The step down forces on election in which `m3.example.net` becomes primary based on its `priority` (page 95) setting.

2. Optionally, if `m3.example.net` is more than 10 seconds behind `m1.example.net`'s optime, and if you don't need to have a primary designated within 10 seconds, you can force `m1.example.net` to step down by running:

   ```
   db.adminCommand({replSetStepDown: 86400, force: 1})
   ```

   This prevents `m1.example.net` from being primary for 86,400 seconds (24 hours), even if there is no other member that can become primary. When `m3.example.net` catches up with `m1.example.net` it will become primary.

   If you later want to make `m1.example.net` primary again while it waits for `m3.example.net` to catch up, issue the following command to make `m1.example.net` seek election again:

   ```
   rs.freeze()
   ```

   The `rs.freeze()` provides a wrapper around the `replSetFreeze` database command.

Force a Member to be Primary Using Database Commands Changed in version 1.8.

Consider a *replica set* with the following members:

- `mdb0.example.net` - the current *primary*.
- `mdb1.example.net` - a *secondary*.
- `mdb2.example.net` - a secondary .

To force a member to become primary use the following procedure:

1. In a `mongo` shell, run `rs.status()` to ensure your replica set is running as expected.
2. In a `mongo` shell connected to the `mongod` instance running on `mdb2.example.net`, freeze `mdb2.example.net` so that it does not attempt to become primary for 120 seconds.

   ```
   rs.freeze(120)
   ```

3. In a `mongo` shell connected the `mongod` running on `mdb0.example.net`, step down this instance that the `mongod` is not eligible to become primary for 120 seconds:

   ```
   rs.stepDown(120)
   ```

   `mdb1.example.net` becomes primary.

   Note: During the transition, there is a short window where the set does not have a primary.

For more information, consider the `rs.freeze()` and `rs.stepDown()` methods that wrap the `replSetFreeze` and `replSetStepDown` commands.

Resync a Member of a Replica Set

A *replica set* member becomes "stale" when its replication process falls so far behind that the *primary* overwrites oplog entries the member has not yet replicated. The member cannot catch up and becomes "stale." When this occurs, you must completely resynchronize the member by removing its data and performing an *initial sync* (page 36).

This tutorial addressed both resyncing a stale member and to creating a new member using seed data from another member. When syncing a member, choose a time when the system has the bandwidth to move a large amount of data. Schedule the synchronization during a time of low usage or during a maintenance window.

MongoDB provides two options for performing an initial sync:

- Restart the `mongod` with an empty data directory and let MongoDB's normal initial syncing feature restore the data. This is the more simple option but may take longer to replace the data.

  See *Procedures* (page 75).

- Restart the machine with a copy of a recent data directory from another member in the replica set. This procedure can replace the data more quickly but requires more manual steps.

  See *Sync by Copying Data Files from Another Member* (page 75).

Procedures

Automatically Sync a Member

Warning: During initial sync, `mongod` will remove the content of the `dbPath`.

This procedure relies on MongoDB's regular process for *initial sync* (page 36). This will store the current data on the member. For an overview of MongoDB initial sync process, see the *Replication Processes* (page 34) section.

If the instance has no data, you can simply follow the *Add Members to a Replica Set* (page 57) or *Replace a Replica Set Member* (page 61) procedure to add a new member to a replica set.

You can also force a `mongod` that is already a member of the set to to perform an initial sync by restarting the instance without the content of the `dbPath` as follows:

1. Stop the member's `mongod` instance. To ensure a clean shutdown, use the `db.shutdownServer()` method from the `mongo` shell or on Linux systems, the *`mongod --shutdown`* option.
2. Delete all data and sub-directories from the member's data directory. By removing the data `dbPath`, MongoDB will perform a complete resync. Consider making a backup first.

At this point, the `mongod` will perform an initial sync. The length of the initial sync process depends on the size of the database and network connection between members of the replica set.

Initial sync operations can impact the other members of the set and create additional traffic to the primary and can only occur if another member of the set is accessible and up to date.

Sync by Copying Data Files from Another Member This approach "seeds" a new or stale member using the data files from an existing member of the replica set. The data files **must** be sufficiently recent to allow the new member to catch up with the *oplog*. Otherwise the member would need to perform an initial sync.

Copy the Data Files You can capture the data files as either a snapshot or a direct copy. However, in most cases you cannot copy data files from a running `mongod` instance to another because the data files will change during the file copy operation.

Important: If copying data files, you must copy the content of the `local` database.

You *cannot* use a `mongodump` backup for the data files, **only a snapshot backup**. For approaches to capturing a consistent snapshot of a running `mongod` instance, see the `http://docs.mongodb.org/manual/core/backups` documentation.

Sync the Member After you have copied the data files from the "seed" source, start the `mongod` instance and allow it to apply all operations from the oplog until it reflects the current state of the replica set.

Configure Replica Set Tag Sets

Tag sets let you customize *write concern* and *read preferences* for a *replica set*. MongoDB stores tag sets in the replica set configuration object, which is the document returned by `rs.conf()`, in the `members[n].tags` (page 96) sub-document.

This section introduces the configuration of tag sets. For an overview on tag sets and their use, see *Replica Set Write Concern* and *Tag Sets* (page 31).

Differences Between Read Preferences and Write Concerns

Custom read preferences and write concerns evaluate tags sets in different ways:

- Read preferences consider the value of a tag when selecting a member to read from.
- Write concerns do not use the value of a tag to select a member except to consider whether or not the value is unique.

For example, a tag set for a read operation may resemble the following document:

```
{ "disk": "ssd", "use": "reporting" }
```

To fulfill such a read operation, a member would need to have both of these tags. Any of the following tag sets would satisfy this requirement:

```
{ "disk": "ssd", "use": "reporting" }
{ "disk": "ssd", "use": "reporting", "rack": "a" }
{ "disk": "ssd", "use": "reporting", "rack": "d" }
{ "disk": "ssd", "use": "reporting", "mem": "r"}
```

The following tag sets would *not* be able to fulfill this query:

```
{ "disk": "ssd" }
{ "use": "reporting" }
{ "disk": "ssd", "use": "production" }
{ "disk": "ssd", "use": "production", "rack": "k" }
{ "disk": "spinning", "use": "reporting", "mem": "32" }
```

Add Tag Sets to a Replica Set

Given the following replica set configuration:

```
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [
            {
                    "_id" : 0,
```

```
                "host" : "mongodb0.example.net:27017"
        },
        {
                "_id" : 1,
                "host" : "mongodb1.example.net:27017"
        },
        {
                "_id" : 2,
                "host" : "mongodb2.example.net:27017"
        }
    ]
}
```

You could add tag sets to the members of this replica set with the following command sequence in the `mongo` shell:

```
conf = rs.conf()
conf.members[0].tags = { "dc": "east", "use": "production"  }
conf.members[1].tags = { "dc": "east", "use": "reporting"  }
conf.members[2].tags = { "use": "production"  }
rs.reconfig(conf)
```

After this operation the output of `rs.conf()` would resemble the following:

```
{
    "_id" : "rs0",
    "version" : 2,
    "members" : [
        {
                "_id" : 0,
                "host" : "mongodb0.example.net:27017",
                "tags" : {
                        "dc": "east",
                        "use": "production"
                }
        },
        {
                "_id" : 1,
                "host" : "mongodb1.example.net:27017",
                "tags" : {
                        "dc": "east",
                        "use": "reporting"
                }
        },
        {
                "_id" : 2,
                "host" : "mongodb2.example.net:27017",
                "tags" : {
                        "use": "production"
                }
        }
    ]
}
```

Important: In tag sets, all tag values must be strings.

Custom Multi-Datacenter Write Concerns

Given a five member replica set with members in two data centers:

1. a facility `VA` tagged `dc.va`
2. a facility `GTO` tagged `dc.gto`

Create a custom write concern to require confirmation from two data centers using replica set tags, using the following sequence of operations in the `mongo` shell:

1. Create a replica set configuration JavaScript object `conf`:

   ```
   conf = rs.conf()
   ```

2. Add tags to the replica set members reflecting their locations:

   ```
   conf.members[0].tags = { "dc.va": "rack1"}
   conf.members[1].tags = { "dc.va": "rack2"}
   conf.members[2].tags = { "dc.gto": "rack1"}
   conf.members[3].tags = { "dc.gto": "rack2"}
   conf.members[4].tags = { "dc.va": "rack1"}
   rs.reconfig(conf)
   ```

3. Create a custom `getLastErrorModes` (page 97) setting to ensure that the write operation will propagate to at least one member of each facility:

   ```
   conf.settings = { getLastErrorModes: { MultipleDC : { "dc.va": 1, "dc.gto": 1}}
   ```

4. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

To ensure that a write operation propagates to at least one member of the set in both data centers, use the `MultipleDC` write concern mode as follows:

```
db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )
```

Alternatively, if you want to ensure that each write operation propagates to at least 2 racks in each facility, reconfigure the replica set as follows in the `mongo` shell:

1. Create a replica set configuration object `conf`:

   ```
   conf = rs.conf()
   ```

2. Redefine the `getLastErrorModes` (page 97) value to require two different values of both `dc.va` and `dc.gto`:

   ```
   conf.settings = { getLastErrorModes: { MultipleDC : { "dc.va": 2, "dc.gto": 2}}
   ```

3. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

Now, the following write operation will only return after the write operation propagates to at least two different racks in the each facility:

Changed in version 2.6: A new protocol for *write operations* integrates write concerns with the write operations. Previous versions used the `getLastError` command to specify the write concerns.

```
db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )
```

Configure Tag Sets for Functional Segregation of Read and Write Operations

Given a replica set with tag sets that reflect:

- data center facility,
- physical rack location of instance, and
- storage system (i.e. disk) type.

Where each member of the set has a tag set that resembles one of the following: [11]

```
{"dc.va": "rack1", disk:"ssd", ssd: "installed" }
{"dc.va": "rack2", disk:"raid"}
{"dc.gto": "rack1", disk:"ssd", ssd: "installed" }
{"dc.gto": "rack2", disk:"raid"}
{"dc.va": "rack1", disk:"ssd", ssd: "installed" }
```

To target a read operation to a member of the replica set with a disk type of `ssd`, you could use the following tag set:

```
{ disk: "ssd" }
```

However, to create comparable write concern modes, you would specify a different set of `getLastErrorModes` (page 97) configuration. Consider the following sequence of operations in the `mongo` shell:

1. Create a replica set configuration object `conf`:

   ```
   conf = rs.conf()
   ```

2. Redefine the `getLastErrorModes` (page 97) value to configure two write concern modes:

   ```
   conf.settings = {
                    "getLastErrorModes" : {
                            "ssd" : {
                                        "ssd" : 1
                            },
                            "MultipleDC" : {
                                    "dc.va" : 1,
                                   "dc.gto" : 1
                            }
                    }
                  }
   ```

3. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

Now you can specify the `MultipleDC` write concern mode, as in the following, to ensure that a write operation propagates to each data center.

Changed in version 2.6: A new protocol for *write operations* integrates write concerns with the write operations. Previous versions used the `getLastError` command to specify the write concerns.

```
db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )
```

Additionally, you can specify the `ssd` write concern mode to ensure that a write operation propagates to at least one instance with an SSD.

[11] Since read preferences and write concerns use the value of fields in tag sets differently, larger deployments may have some redundancy.

Reconfigure a Replica Set with Unavailable Members

To reconfigure a *replica set* when a **majority** of members are available, use the `rs.reconfig()` operation on the current *primary*, following the example in the *Replica Set Reconfiguration Procedure*.

This document provides the following options for re-configuring a replica set when *only* a **minority** of members are accessible:

- *Reconfigure by Forcing the Reconfiguration* (page 80)
- *Reconfigure by Replacing the Replica Set* (page 81)

You may need to use one of these procedures, for example, in a geographically distributed replica set, where *no* local group of members can reach a majority. See *Replica Set Elections* (page 22) for more information on this situation.

Reconfigure by Forcing the Reconfiguration

Changed in version 2.0.

This procedure lets you recover while a majority of *replica set* members are down or unreachable. You connect to any surviving member and use the `force` option to the `rs.reconfig()` method.

The `force` option forces a new configuration onto the member. Use this procedure only to recover from catastrophic interruptions. Do not use `force` every time you reconfigure. Also, do not use the `force` option in any automatic scripts and do not use `force` when there is still a *primary*.

To force reconfiguration:

1. Back up a surviving member.
2. Connect to a surviving member and save the current configuration. Consider the following example commands for saving the configuration:

   ```
   cfg = rs.conf()

   printjson(cfg)
   ```

3. On the same member, remove the down and unreachable members of the replica set from the `members` (page 94) array by setting the array equal to the surviving members alone. Consider the following example, which uses the `cfg` variable created in the previous step:

   ```
   cfg.members = [cfg.members[0] , cfg.members[4] , cfg.members[7]]
   ```

4. On the same member, reconfigure the set by using the `rs.reconfig()` command with the `force` option set to `true`:

   ```
   rs.reconfig(cfg, {force : true})
   ```

   This operation forces the secondary to use the new configuration. The configuration is then propagated to all the surviving members listed in the `members` array. The replica set then elects a new primary.

   Note: When you use `force : true`, the version number in the replica set configuration increases significantly, by tens or hundreds of thousands. This is normal and designed to prevent set version collisions if you accidentally force re-configurations on both sides of a network partition and then the network partitioning ends.

   ---

5. If the failure or partition was only temporary, shut down or decommission the removed members as soon as possible.

Reconfigure by Replacing the Replica Set

Use the following procedure **only** for versions of MongoDB prior to version 2.0. If you're running MongoDB 2.0 or later, use the above procedure, *Reconfigure by Forcing the Reconfiguration* (page 80).

These procedures are for situations where a *majority* of the *replica set* members are down or unreachable. If a majority is *running*, then skip these procedures and instead use the `rs.reconfig()` command according to the examples in *replica-set-reconfiguration-usage*.

If you run a pre-2.0 version and a majority of your replica set is down, you have the two options described here. Both involve replacing the replica set.

Reconfigure by Turning Off Replication This option replaces the *replica set* with a *standalone* server.

1. Stop the surviving `mongod` instances. To ensure a clean shutdown, use an existing *control script* or use the `db.shutdownServer()` method.

   For example, to use the `db.shutdownServer()` method, connect to the server using the `mongo` shell and issue the following sequence of commands:

   ```
   use admin
   db.shutdownServer()
   ```

2. Create a backup of the data directory (i.e. `dbPath`) of the surviving members of the set.

   Optional

   If you have a backup of the database you may instead remove this data.

   ---

3. Restart one of the `mongod` instances *without* the `--replSet` parameter.

   The data is now accessible and provided by a single server that is not a replica set member. Clients can use this server for both reads and writes.

When possible, re-deploy a replica set to provide redundancy and to protect your deployment from operational interruption.

Reconfigure by "Breaking the Mirror" This option selects a surviving *replica set* member to be the new *primary* and to "seed" a new replica set. In the following procedure, the new primary is `db0.example.net`. MongoDB copies the data from `db0.example.net` to all the other members.

1. Stop the surviving `mongod` instances. To ensure a clean shutdown, use an existing *control script* or use the `db.shutdownServer()` method.

   For example, to use the `db.shutdownServer()` method, connect to the server using the `mongo` shell and issue the following sequence of commands:

   ```
   use admin
   db.shutdownServer()
   ```

2. Move the data directories (i.e. `dbPath`) for all the members except `db0.example.net`, so that all the members except `db0.example.net` have empty data directories. For example:

   ```
   mv /data/db /data/db-old
   ```

3. Move the data files for `local` database (i.e. `local.*`) so that `db0.example.net` has no local database. For example

```
mkdir /data/local-old
mv /data/db/local* /data/local-old/
```

4. Start each member of the replica set normally.
5. Connect to `db0.example.net` in a `mongo` shell and run `rs.initiate()` to initiate the replica set.
6. Add the other set members using `rs.add()`. For example, to add a member running on `db1.example.net` at port `27017`, issue the following command:

   ```
   rs.add("db1.example.net:27017")
   ```

   MongoDB performs an initial sync on the added members by copying all data from `db0.example.net` to the added members.

See also:

Resync a Member of a Replica Set (page 75)

Manage Chained Replication

Starting in version 2.0, MongoDB supports chained replication. A chained replication occurs when a *secondary* member replicates from another secondary member instead of from the *primary*. This might be the case, for example, if a secondary selects its replication target based on ping time and if the closest member is another secondary.

Chained replication can reduce load on the primary. But chained replication can also result in increased replication lag, depending on the topology of the network.

New in version 2.2.2.

You can use the `chainingAllowed` (page 97) setting in *Replica Set Configuration* (page 93) to disable chained replication for situations where chained replication is causing lag.

MongoDB enables chained replication by default. This procedure describes how to disable it and how to re-enable it.

Note: If chained replication is disabled, you still can use `replSetSyncFrom` to specify that a secondary replicates from another secondary. But that configuration will last only until the secondary recalculates which member to sync from.

Disable Chained Replication

To disable chained replication, set the `chainingAllowed` (page 97) field in *Replica Set Configuration* (page 93) to `false`.

You can use the following sequence of commands to set `chainingAllowed` (page 97) to `false`:

1. Copy the configuration settings into the `cfg` object:

   ```
   cfg = rs.config()
   ```

2. Take note of whether the current configuration settings contain the `settings` sub-document. If they do, skip this step.

   Warning: To avoid data loss, skip this step if the configuration settings contain the `settings` sub-document.

   If the current configuration settings **do not** contain the `settings` sub-document, create the sub-document by issuing the following command:

```
cfg.settings = { }
```

3. Issue the following sequence of commands to set `chainingAllowed` (page 97) to `false`:

   ```
   cfg.settings.chainingAllowed = false
   rs.reconfig(cfg)
   ```

Re-enable Chained Replication

To re-enable chained replication, set `chainingAllowed` (page 97) to `true`. You can use the following sequence of commands:

```
cfg = rs.config()
cfg.settings.chainingAllowed = true
rs.reconfig(cfg)
```

Change Hostnames in a Replica Set

For most *replica sets*, the hostnames in the `host` (page 94) field never change. However, if organizational needs change, you might need to migrate some or all host names.

Note: Always use resolvable hostnames for the value of the `host` (page 94) field in the replica set configuration to avoid confusion and complexity.

Overview

This document provides two separate procedures for changing the hostnames in the `host` (page 94) field. Use either of the following approaches:

- *Change hostnames without disrupting availability* (page 84). This approach ensures your applications will always be able to read and write data to the replica set, but the approach can take a long time and may incur downtime at the application layer.

  If you use the first procedure, you must configure your applications to connect to the replica set at both the old and new locations, which often requires a restart and reconfiguration at the application layer and which may affect the availability of your applications. Re-configuring applications is beyond the scope of this document.

- *Stop all members running on the old hostnames at once* (page 85). This approach has a shorter maintenance window, but the replica set will be unavailable during the operation.

See also:

Replica Set Reconfiguration Process, *Deploy a Replica Set* (page 44), and *Add Members to a Replica Set* (page 57).

Assumptions

Given a *replica set* with three members:

- `database0.example.com:27017` (the *primary*)
- `database1.example.com:27017`
- `database2.example.com:27017`

And with the following `rs.conf()` output:

```
{
    "_id" : "rs",
    "version" : 3,
    "members" : [
        {
            "_id" : 0,
            "host" : "database0.example.com:27017"
        },
        {
            "_id" : 1,
            "host" : "database1.example.com:27017"
        },
        {
            "_id" : 2,
            "host" : "database2.example.com:27017"
        }
    ]
}
```

The following procedures change the members' hostnames as follows:

- `mongodb0.example.net:27017` (the primary)
- `mongodb1.example.net:27017`
- `mongodb2.example.net:27017`

Use the most appropriate procedure for your deployment.

Change Hostnames while Maintaining Replica Set Availability

This procedure uses the above *assumptions* (page 83).

1. For each *secondary* in the replica set, perform the following sequence of operations:
   (a) Stop the secondary.
   (b) Restart the secondary at the new location.
   (c) Open a `mongo` shell connected to the replica set's primary. In our example, the primary runs on port `27017` so you would issue the following command:

   ```
   mongo --port 27017
   ```

   (d) Use `rs.reconfig()` to update the *replica set configuration document* (page 93) with the new hostname.

   For example, the following sequence of commands updates the hostname for the secondary at the array index `1` of the `members` array (i.e. `members[1]`) in the replica set configuration document:

   ```
   cfg = rs.conf()
   cfg.members[1].host = "mongodb1.example.net:27017"
   rs.reconfig(cfg)
   ```

   For more information on updating the configuration document, see *replica-set-reconfiguration-usage*.
   (e) Make sure your client applications are able to access the set at the new location and that the secondary has a chance to catch up with the other members of the set.

   Repeat the above steps for each non-primary member of the set.

2. Open a `mongo` shell connected to the primary and step down the primary using the `rs.stepDown()` method:

   ```
   rs.stepDown()
   ```

   The replica set elects another member to the become primary.

3. When the step down succeeds, shut down the old primary.

4. Start the `mongod` instance that will become the new primary in the new location.

5. Connect to the current primary, which was just elected, and update the *replica set configuration document* (page 93) with the hostname of the node that is to become the new primary.

   For example, if the old primary was at position `0` and the new primary's hostname is `mongodb0.example.net:27017`, you would run:

   ```
   cfg = rs.conf()
   cfg.members[0].host = "mongodb0.example.net:27017"
   rs.reconfig(cfg)
   ```

6. Open a `mongo` shell connected to the new primary.

7. To confirm the new configuration, call `rs.conf()` in the `mongo` shell.

   Your output should resemble:

   ```
   {
       "_id" : "rs",
       "version" : 4,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongodb0.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongodb1.example.net:27017"
           },
           {
               "_id" : 2,
               "host" : "mongodb2.example.net:27017"
           }
       ]
   }
   ```

Change All Hostnames at the Same Time

This procedure uses the above *assumptions* (page 83).

1. Stop all members in the *replica set*.

2. Restart each member *on a different port* and *without* using the *`--replSet`* run-time option. Changing the port number during maintenance prevents clients from connecting to this host while you perform maintenance. Use the member's usual *`--dbpath`*, which in this example is `/data/db1`. Use a command that resembles the following:

   ```
   mongod --dbpath /data/db1/ --port 37017
   ```

3. For each member of the replica set, perform the following sequence of operations:

(a) Open a `mongo` shell connected to the `mongod` running on the new, temporary port. For example, for a member running on a temporary port of `37017`, you would issue this command:

```
mongo --port 37017
```

(b) Edit the replica set configuration manually. The replica set configuration is the only document in the `system.replset` collection in the `local` database. Edit the replica set configuration with the new hostnames and correct ports for all the members of the replica set. Consider the following sequence of commands to change the hostnames in a three-member set:

```
use local

cfg = db.system.replset.findOne( { "_id": "rs" } )

cfg.members[0].host = "mongodb0.example.net:27017"

cfg.members[1].host = "mongodb1.example.net:27017"

cfg.members[2].host = "mongodb2.example.net:27017"

db.system.replset.update( { "_id": "rs" } , cfg )
```

(c) Stop the `mongod` process on the member.

4. After re-configuring all members of the set, start each `mongod` instance in the normal way: use the usual port number and use the *`--replSet`* option. For example:

   ```
   mongod --dbpath /data/db1/ --port 27017 --replSet rs
   ```

5. Connect to one of the `mongod` instances using the `mongo` shell. For example:

   ```
   mongo --port 27017
   ```

6. To confirm the new configuration, call `rs.conf()` in the `mongo` shell.

   Your output should resemble:

   ```
   {
       "_id" : "rs",
       "version" : 4,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongodb0.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongodb1.example.net:27017"
           },
           {
               "_id" : 2,
               "host" : "mongodb2.example.net:27017"
           }
       ]
   }
   ```

Configure a Secondary's Sync Target

Overview

Secondaries capture data from the primary member to maintain an up to date copy of the sets' data. However, by default secondaries may automatically change their sync targets to secondary members based on changes in the ping time between members and the state of other members' replication. See *Replica Set Data Synchronization* (page 35) and *Manage Chained Replication* (page 82) for more information.

For some deployments, implementing a custom replication sync topology may be more effective than the default sync target selection logic. MongoDB provides the ability to specify a host to use as a sync target.

To override the default sync target selection logic, you may manually configure a *secondary* member's sync target to temporarily pull *oplog* entries. The following provide access to this functionality:

- `replSetSyncFrom` command, or
- `rs.syncFrom()` helper in the `mongo` shell

Considerations

Sync Logic Only modify the default sync logic as needed, and always exercise caution. `rs.syncFrom()` will not affect an in-progress initial sync operation. To affect the sync target for the initial sync, run `rs.syncFrom()` operation *before* initial sync.

If you run `rs.syncFrom()` during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Persistence `replSetSyncFrom` and `rs.syncFrom()` provide a temporary override of default behavior. `mongod` will revert to the default sync behavior in the following situations:

- The `mongod` instance restarts.
- The connection between the `mongod` and the sync target closes.

Changed in version 2.4: The sync target falls more than 30 seconds behind another member of the replica set; the `mongod` will revert to the default sync target.

Target The member to sync from must be a valid source for data in the set. To sync from a member, the member must:

- Have data. It cannot be an arbiter, in startup or recovering mode, and must be able to answer data queries.
- Be accessible.
- Be a member of the same set in the replica set configuration.
- Build indexes with the `buildIndexes` (page 95) setting.
- A different member of the set, to prevent syncing from itself.

If you attempt to replicate from a member that is more than 10 seconds behind the current member, `mongod` will log a warning but will still replicate from the lagging member.

If you run `replSetSyncFrom` during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Procedure

To use the `replSetSyncFrom` command in the `mongo` shell:

```
db.adminCommand( { replSetSyncFrom: "hostname<:port>" } );
```

To use the `rs.syncFrom()` helper in the `mongo` shell:

```
rs.syncFrom("hostname<:port>");
```

3.4 Troubleshoot Replica Sets

This section describes common strategies for troubleshooting *replica set* deployments.

Check Replica Set Status

To display the current state of the replica set and current state of each member, run the `rs.status()` method in a `mongo` shell connected to the replica set's *primary*. For descriptions of the information displayed by `rs.status()`, see `http://docs.mongodb.org/manual/reference/command/replSetGetStatus`.

Note: The `rs.status()` method is a wrapper that runs the `replSetGetStatus` database command.

Check the Replication Lag

Replication lag is a delay between an operation on the *primary* and the application of that operation from the *oplog* to the *secondary*. Replication lag can be a significant issue and can seriously affect MongoDB *replica set* deployments. Excessive replication lag makes "lagged" members ineligible to quickly become primary and increases the possibility that distributed read operations will be inconsistent.

To check the current length of replication lag:

- In a `mongo` shell connected to the primary, call the `rs.printSlaveReplicationInfo()` method.

  Returns the `syncedTo` value for each member, which shows the time when the last oplog entry was written to the secondary, as shown in the following example:

  ```
  source: m1.example.net:27017
      syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
      0 secs (0 hrs) behind the primary
  source: m2.example.net:27017
      syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
      0 secs (0 hrs) behind the primary
  ```

  A *delayed member* (page 13) may show as `0` seconds behind the primary when the inactivity period on the primary is greater than the `slaveDelay` (page 96) value.

  Note: The `rs.status()` method is a wrapper around the `replSetGetStatus` database command.

- Monitor the rate of replication by watching the oplog time in the "replica" graph in the MongoDB Management Service[12]. For more information see the documentation for MMS[13].

Possible causes of replication lag include:

[12] https://mms.mongodb.com/
[13] https://docs.mms.mongodb.com/

- **Network Latency**

  Check the network routes between the members of your set to ensure that there is no packet loss or network routing issue.

  Use tools including `ping` to test latency between set members and `traceroute` to expose the routing of packets network endpoints.

- **Disk Throughput**

  If the file system and disk device on the secondary is unable to flush data to disk as quickly as the primary, then the secondary will have difficulty keeping state. Disk-related issues are incredibly prevalent on multi-tenant systems, including virtualized instances, and can be transient if the system accesses disk devices over an IP network (as is the case with Amazon's EBS system.)

  Use system-level tools to assess disk status, including `iostat` or `vmstat`.

- **Concurrency**

  In some cases, long-running operations on the primary can block replication on secondaries. For best results, configure *write concern* to require confirmation of replication to secondaries, as described in *replica set write concern*. This prevents write operations from returning if replication cannot keep up with the write load.

  Use the *database profiler* to see if there are slow queries or long-running operations that correspond to the incidences of lag.

- **Appropriate Write Concern**

  If you are performing a large data ingestion or bulk load operation that requires a large number of writes to the primary, particularly with *unacknowledged write concern*, the secondaries will not be able to read the oplog fast enough to keep up with changes.

  To prevent this, require *write acknowledgment or journaled write concern* after every 100, 1,000, or an another interval to provide an opportunity for secondaries to catch up with the primary.

  For more information see:

  – *Replica Acknowledge Write Concern*

  – *Replica Set Write Concern*

  – *Oplog Size* (page 34)

Test Connections Between all Members

All members of a *replica set* must be able to connect to every other member of the set to support replication. Always verify connections in both "directions." Networking topologies and firewall configurations prevent normal and required connectivity, which can block replication.

Consider the following example of a bidirectional test of networking:

Example

Given a replica set with three members running on three separate hosts:

- `m1.example.net`
- `m2.example.net`
- `m3.example.net`

1. Test the connection from `m1.example.net` to the other hosts with the following operation set `m1.example.net`:

```
mongo --host m2.example.net --port 27017

mongo --host m3.example.net --port 27017
```

2. Test the connection from `m2.example.net` to the other two hosts with the following operation set from `m2.example.net`, as in:

   ```
   mongo --host m1.example.net --port 27017

   mongo --host m3.example.net --port 27017
   ```

   You have now tested the connection between `m2.example.net` and `m1.example.net` in both directions.

3. Test the connection from `m3.example.net` to the other two hosts with the following operation set from the `m3.example.net` host, as in:

   ```
   mongo --host m1.example.net --port 27017

   mongo --host m2.example.net --port 27017
   ```

If any connection, in any direction fails, check your networking and firewall configuration and reconfigure your environment to allow these connections.

Socket Exceptions when Rebooting More than One Secondary

When you reboot members of a replica set, ensure that the set is able to elect a primary during the maintenance. This means ensuring that a majority of the set's '`votes` (page 96) are available.

When a set's active members can no longer form a majority, the set's *primary* steps down and becomes a *secondary*. The former primary closes all open connections to client applications. Clients attempting to write to the former primary receive socket exceptions and *Connection reset* errors until the set can elect a primary.

Example

Given a three-member replica set where every member has one vote, the set can elect a primary only as long as two members can connect to each other. If two you reboot the two secondaries once, the primary steps down and becomes a secondary. Until the at least one secondary becomes available, the set has no primary and cannot elect a new primary.

For more information on votes, see *Replica Set Elections* (page 22). For related information on connection errors, see *faq-keepalive*.

Check the Size of the Oplog

A larger *oplog* can give a replica set a greater tolerance for lag, and make the set more resilient.

To check the size of the oplog for a given *replica set* member, connect to the member in a `mongo` shell and run the `rs.printReplicationInfo()` method.

The output displays the size of the oplog and the date ranges of the operations contained in the oplog. In the following example, the oplog is about 10MB and is able to fit about 26 hours (94400 seconds) of operations:

```
configured oplog size:   10.10546875MB
log length start to end: 94400 (26.22hrs)
oplog first event time:  Mon Mar 19 2012 13:50:38 GMT-0400 (EDT)
oplog last event time:   Wed Oct 03 2012 14:59:10 GMT-0400 (EDT)
now:                     Wed Oct 03 2012 15:00:21 GMT-0400 (EDT)
```

The oplog should be long enough to hold all transactions for the longest downtime you expect on a secondary. At a minimum, an oplog should be able to hold minimum 24 hours of operations; however, many users prefer to have 72 hours or even a week's work of operations.

For more information on how oplog size affects operations, see:

- *Oplog Size* (page 34),
- *Delayed Replica Set Members* (page 13), and
- *Check the Replication Lag* (page 88).

Note: You normally want the oplog to be the same size on all members. If you resize the oplog, resize it on all members.

To change oplog size, see the *Change the Size of the Oplog* (page 69) tutorial.

Oplog Entry Timestamp Error

Consider the following error in `mongod` output and logs:

```
replSet error fatal couldn't query the local local.oplog.rs collection.  Terminating mongod after 30
<timestamp> [rsStart] bad replSet oplog entry?
```

Often, an incorrectly typed value in the `ts` field in the last *oplog* entry causes this error. The correct data type is Timestamp.

Check the type of the `ts` value using the following two queries against the oplog collection:

```
db = db.getSiblingDB("local")
db.oplog.rs.find().sort({$natural:-1}).limit(1)
db.oplog.rs.find({ts:{$type:17}}).sort({$natural:-1}).limit(1)
```

The first query returns the last document in the oplog, while the second returns the last document in the oplog where the `ts` value is a Timestamp. The `$type` operator allows you to select *BSON type* 17, is the Timestamp data type.

If the queries don't return the same document, then the last document in the oplog has the wrong data type in the `ts` field.

Example

If the first query returns this as the last oplog entry:

```
{ "ts" : {t: 1347982456000, i: 1},
  "h" : NumberLong("8191276672478122996"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 4 } }
```

And the second query returns this as the last entry where `ts` has the `Timestamp` type:

```
{ "ts" : Timestamp(1347982454000, 1),
  "h" : NumberLong("6188469075153256465"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 3 } }
```

Then the value for the `ts` field in the last oplog entry is of the wrong data type.

To set the proper type for this value and resolve this issue, use an update operation that resembles the following:

```
db.oplog.rs.update( { ts: { t:1347982456000, i:1 } },
                    { $set: { ts: new Timestamp(1347982456000, 1)}})
```

Modify the timestamp values as needed based on your oplog entry. This operation may take some period to complete because the update must scan and pull the entire oplog into memory.

Duplicate Key Error on `local.slaves`

The *duplicate key on local.slaves* error, occurs when a *secondary* or *slave* changes its hostname and the *primary* or *master* tries to update its `local.slaves` collection with the new name. The update fails because it contains the same `_id` value as the document containing the previous hostname. The error itself will resemble the following.

```
exception: E11000 duplicate key error index: local.slaves.$_id_  dup key: { : ObjectId('<object ID>')
```

This is a benign error and does not affect replication operations on the *secondary* or *slave*.

To prevent the error from appearing, drop the `local.slaves` collection from the *primary* or *master*, with the following sequence of operations in the `mongo` shell:

```
use local
db.slaves.drop()
```

The next time a *secondary* or *slave* polls the *primary* or *master*, the *primary* or *master* recreates the `local.slaves` collection.

4 Replication Reference

4.1 Replication Methods in the `mongo` Shell

Name	Description
`rs.add()`	Adds a member to a replica set.
`rs.addArb()`	Adds an *arbiter* to a replica set.
`rs.conf()`	Returns the replica set configuration document.
`rs.freeze()`	Prevents the current member from seeking election as primary for a period of time.
`rs.help()`	Returns basic help text for *replica set* functions.
`rs.initiate()`	Initializes a new replica set.
`rs.printReplicationInfo()`	Prints a report of the status of the replica set from the perspective of the primary.
`rs.printSlaveReplicationInfo()`	Prints a report of the status of the replica set from the perspective of the secondaries.
`rs.reconfig()`	Re-configures a replica set by applying a new replica set configuration object.
`rs.remove()`	Remove a member from a replica set.
`rs.slaveOk()`	Sets the `slaveOk` property for the current connection. Deprecated. Use `readPref()` and `Mongo.setReadPref()` to set *read preference*.
`rs.status()`	Returns a document with information about the state of the replica set.
`rs.stepDown()`	Causes the current *primary* to become a secondary which forces an *election*.
`rs.syncFrom()`	Sets the member that this replica set member will sync from, overriding the default sync target selection logic.

4.2 Replication Database Commands

Name	Description
`applyOps`	Internal command that applies *oplog* entries to the current data set.
`getoptime`	Internal command to support replication, returns the optime.
`isMaster`	Displays information about this member's role in the replica set, including whether it is the master.
`replSetFreeze`	Prevents the current member from seeking election as *primary* for a period of time.
`replSetGetStatus`	Returns a document that reports on the status of the replica set.
`replSetInitiate`	Initializes a new replica set.
`replSetMaintenance`	Enables or disables a maintenance mode, which puts a *secondary* node in a `RECOVERING` state.
`replSetReconfig`	Applies a new configuration to an existing replica set.
`replSetStepDown`	Forces the current *primary* to *step down* and become a *secondary*, forcing an election.
`replSetSyncFrom`	Explicitly override the default logic for selecting a member to replicate from.
`resync`	Forces a `mongod` to re-synchronize from the *master*. For master-slave replication only.

4.3 Replica Set Reference Documentation

Replica Set Configuration **(page 93)** Complete documentation of the *replica set* configuration object returned by `rs.conf()`.

The local Database **(page 98)** Complete documentation of the content of the `local` database that `mongod` instances use to support replication.

Replica Set Member States **(page 100)** Reference for the replica set member states.

Read Preference Reference **(page 102)** Complete documentation of the five read preference modes that the MongoDB drivers support.

Replica Set Configuration

The configuration for a replica set is stored as a document in the `system.replset` (page 99) collection in the *local database* (page 98).

Replica Set Configuration Document

The following document provides a representation of a replica set configuration document. The configuration of your replica set may include only a subset of these settings:

```
{
  _id: <string>,
  version: <int>,
  members: [
    {
      _id: <int>,
      host: <string>,
      arbiterOnly: <boolean>,
      buildIndexes: <boolean>,
      hidden: <boolean>,
      priority: <number>,
      tags: <document>,
      slaveDelay: <int>,
      votes: <number>
```

```
    },
    ...
  ],
  settings: {
    getLastErrorDefaults : <document>,
    chainingAllowed : <boolean>,
    getLastErrorModes : <document>,
    heartbeatTimeoutSecs: <int>
  }
}
```

Configuration Settings

`local.system.replset._id`
: *Type*: string

  The name of the replica set. Once set, you cannot change the name of a replica set.

  See

  `replSetName` or *`--replSet`* for information on setting the replica set name.

`local.system.replset.version`
: An incrementing number used to distinguish revisions of the replica set configuration object from previous iterations of the configuration.

`replset.members`
`local.system.replset.members`
: *Type*: array

  An array of member configuration documents, one for each member of the replica set. The `members` (page 94) array is a zero-indexed array.

  Each member-specific configuration document can contain the following fields:

  `local.system.replset.members[n]._id`
  : *Type*: integer

    A numeric identifier of every member in the replica set. Once set, you cannot change the `_id` (page 94) of a member.

    Note: When updating the replica configuration object, access the replica set members in the `members` (page 94) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 94) field in each document in the `members` (page 94) array.

  `local.system.replset.members[n].host`
  : *Type*: string

    The hostname and, if specified, the port number, of the set member.

    The hostname name must be resolvable for every host in the replica set.

    Warning: `host` (page 94) cannot hold a value that resolves to `localhost` or the local interface unless *all* members of the set are on hosts that resolve to `localhost`.

local.system.replset.members[n].**arbiterOnly**
Optional.

Type: boolean

Default: false

A boolean that identifies an arbiter. A value of `true` indicates that the member is an arbiter.

When using the `rs.addArb()` method to add an arbiter, the method automatically sets `arbiterOnly` (page 94) to `true` for the added member.

local.system.replset.members[n].**buildIndexes**
Optional.

Type: boolean

Default: true

A boolean that indicates whether the `mongod` builds *indexes* on this member. You can only set this value when adding a member to a replica set. You cannot change `buildIndexes` (page 95) field after the member has been added to the set. To add a member, see `rs.add()` and `rs.reconfig()`.

Do not set to `false` for `mongod` instances that receive queries from clients.

Setting `buildIndexes` to `false` may be useful if **all** the following conditions are true:

- you are only using this instance to perform backups using `mongodump`, *and*
- this member will receive no queries, *and*
- index creation and maintenance overburdens the host system.

Even if set to `false`, secondaries *will* build indexes on the `_id` field in order to facilitate operations required for replication.

> **Warning:** If you set `buildIndexes` (page 95) to `false`, you must also set `priority` (page 95) to `0`. If `priority` (page 95) is not `0`, MongoDB will return an error when attempting to add a member with `buildIndexes` (page 95) equal to `false`.
> To ensure the member receives no queries, you should make all instances that do not build indexes hidden.
> Other secondaries cannot replicate from a member where `buildIndexes` (page 95) is false.

local.system.replset.members[n].**hidden**
Optional.

Type: boolean

Default: false

When this value is `true`, the replica set hides this instance and does not include the member in the output of `db.isMaster()` or `isMaster`. This prevents read operations (i.e. queries) from ever reaching this host by way of secondary *read preference*.

See also:

Hidden Replica Set Members (page 12)

local.system.replset.members[n].**priority**
Optional.

Type: Number, between 0 and 1000.

Default: 1.0

A number that indicates the relative eligibility of a member to become a *primary*.

Specify higher values to make a member *more* eligible to become *primary*, and lower values to make the member *less* eligible. Priorities are only used in comparison to each other. Members of the set will veto election requests from members when another eligible member has a higher priority value. Changing the balance of priority in a replica set will trigger an election.

A `priority` (page 95) of `0` makes it impossible for a member to become primary.

See also:

Replica Set Elections (page 22).

`local.system.replset.members[n].`**`tags`**

Optional.

Type: document

Default: none

A document that contains arbitrary field and value pairs for describing or *tagging* members in order to extend `write concern` and *read preference* (page 102) and thereby allowing configurable data center awareness.

Use `tags` to configure write concerns in conjunction with `getLastErrorModes` (page 97) and `getLastErrorDefaults` (page 97).

Important: In tag sets, all tag values must be strings.

For more information on configuring tag sets for read preference and write concern, see *Configure Replica Set Tag Sets* (page 76).

`local.system.replset.members[n].`**`slaveDelay`**

Optional.

Type: integer

Default: 0

The number of seconds "behind" the primary that this replica set member should "lag".

Use this option to create *delayed members* (page 13). Delayed members maintain a copy of the data that reflects the state of the data at some time in the past.

See also:

Delayed Replica Set Members (page 13)

`local.system.replset.members[n].`**`votes`**

Optional.

Type: integer

Default: 1

The number of votes a server will cast in a *replica set election* (page 22). The number of votes each member has can be either `1` or `0`.

A replica set can have up to 12 members, but can have at most only 7 *voting* members. If you need more than 7 members in one replica set, set `votes` (page 96) to `0` for the additional non-voting members.

Note: Deprecated since version 2.6: `votes` (page 96) values greater than `1`.

Earlier versions of MongoDB allowed a member to have more than `1` vote by setting `votes` (page 96) to a value greater than `1`. Setting `votes` (page 96) to value greater than `1` now produces a warning message.

replset.settings
local.system.replset.**settings**

Optional.

Type: document

A document that contains configuration options that apply to the whole replica set.

The settings (page 97) document contain the following fields:

local.system.replset.settings.**chainingAllowed**

New in version 2.2.4.

Optional.

Type: boolean

Default: true

When chainingAllowed (page 97) is true, the replica set allows *secondary* members to replicate from other secondary members. When chainingAllowed (page 97) is false, secondaries can replicate only from the *primary*.

When you run rs.conf() to view a replica set's configuration, the chainingAllowed (page 97) field appears only when set to false. If not set, chainingAllowed (page 97) is true.

See also:

Manage Chained Replication (page 82)

local.system.replset.settings.**getLastErrorDefaults**

Optional.

Type: document

A document that specifies the *write concern* (page 28) for the replica set. The replica set will use this write concern only when *write operations* or getLastError specify no other write concern.

If getLastErrorDefaults (page 97) is not set, the default write concern for the replica set only requires confirmation from the primary.

local.system.replset.settings.**getLastErrorModes**

Optional.

Type: document

A document used to define an extended *write concern* through the use of tags (page 96). The extended *write concern* can provide *data-center awareness*.

For example, the following document defines an extended write concern named eastCoast and associates with a write to a member that has the east tag.

```
{ getLastErrorModes: { eastCoast: { "east": 1 } } }
```

Write operations to the replica set can use the extended write concern, e.g. `{ w: "eastCoast" }`.

See *Configure Replica Set Tag Sets* (page 76) for more information and example.

local.system.replset.settings.**heartbeatTimeoutSecs**

Optional.

Type: int

Default: 10

Number of seconds that the replica set members wait for a successful heartbeat from each other. If a member does not respond in time, other members mark the delinquent member as inaccessible.

View Replica Set Configuration

To view the current configuration for a replica set, use the `rs.conf()` method. See `rs.conf()` for more information.

Modify Replica Set Configuration

To modify the configuration for a replica set, use the `rs.reconfig()` method, passing a configuration document to the method. See `rs.reconfig()` for more information.

The `local` Database

Overview

Every `mongod` instance has its own `local` database, which stores data used in the replication process, and other instance-specific data. The `local` database is invisible to replication: collections in the `local` database are not replicated.

In replication, the `local` database store stores internal replication data for each member of a *replica set*. The `local` stores the following collections:

Changed in version 2.4: When running with authentication (i.e. `authorization`), authenticating to the `local` database is **not** equivalent to authenticating to the `admin` database. In previous versions, authenticating to the `local` database provided access to all databases.

Collection on all `mongod` Instances

`local.startup_log`

On startup, each `mongod` instance inserts a document into `startup_log` (page 98) with diagnostic information about the `mongod` instance itself and host information. `startup_log` (page 98) is a capped collection. This information is primarily useful for diagnostic purposes.

Example

Consider the following prototype of a document from the `startup_log` (page 98) collection:

```
{
  "_id" : "<string>",
  "hostname" : "<string>",
  "startTime" : ISODate("<date>"),
  "startTimeLocal" : "<string>",
  "cmdLine" : {
        "dbpath" : "<path>",
        "<option>" : <value>
  },
  "pid" : <number>,
  "buildinfo" : {
        "version" : "<string>",
        "gitVersion" : "<string>",
        "sysInfo" : "<string>",
```

```
            "loaderFlags" : "<string>",
            "compilerFlags" : "<string>",
            "allocator" : "<string>",
            "versionArray" : [ <num>, <num>, <...> ],
            "javascriptEngine" : "<string>",
            "bits" : <number>,
            "debug" : <boolean>,
            "maxBsonObjectSize" : <number>
        }
    }
```

Documents in the `startup_log` (page 98) collection contain the following fields:

`local.startup_log.`**`_id`**
: Includes the system hostname and a millisecond epoch value.

`local.startup_log.`**`hostname`**
: The system's hostname.

`local.startup_log.`**`startTime`**
: A UTC *ISODate* value that reflects when the server started.

`local.startup_log.`**`startTimeLocal`**
: A string that reports the `startTime` (page 99) in the system's local time zone.

`local.startup_log.`**`cmdLine`**
: A sub-document that reports the `mongod` runtime options and their values.

`local.startup_log.`**`pid`**
: The process identifier for this process.

`local.startup_log.`**`buildinfo`**
: A sub-document that reports information about the build environment and settings used to compile this `mongod`. This is the same output as `buildInfo`. See `buildInfo`.

Collections on Replica Set Members

`local.system.`**`replset`**
: `local.system.replset` (page 99) holds the replica set's configuration object as its single document. To view the object's configuration information, issue `rs.conf()` from the `mongo` shell. You can also query this collection directly.

`local.oplog.`**`rs`**
: `local.oplog.rs` (page 99) is the capped collection that holds the *oplog*. You set its size at creation using the `oplogSizeMB` setting. To resize the oplog after replica set initiation, use the *Change the Size of the Oplog* (page 69) procedure. For additional information, see the *Oplog Size* (page 34) section.

`local.replset.`**`minvalid`**
: This contains an object used internally by replica sets to track replication status.

`local.`**`slaves`**
: This contains information about each member of the set and the latest point in time that this member has synced to. If this collection becomes out of date, you can refresh it by dropping the collection and allowing MongoDB to automatically refresh it during normal replication:

```
db.getSiblingDB("local").slaves.drop()
```

Collections used in Master/Slave Replication

In *master/slave* replication, the `local` database contains the following collections:

- On the master:

  `local.oplog.$main`
  : This is the oplog for the master-slave configuration.

  `local.slaves`
  : This contains information about each slave.

- On each slave:

  `local.sources`
  : This contains information about the slave's master server.

Replica Set Member States

Each member of a replica set has a state that reflects its disposition within the set.

Number	Name	State Description
0	STARTUP (page 101)	Not yet an active member of any set. All members start up in this state. The `mongod` parses the *replica set configuration document* (page 61) while in STARTUP (page 101).
1	PRIMARY (page 100)	The member in state *primary* (page 7) is the only member that can accept write operations.
2	SECONDARY (page 100)	A member in state *secondary* (page 9) is replicating the data store. Data is available for reads, although they may be stale.
3	RECOVERING (page 101)	A member in this state is replicating the data store but does not yet have a consistent view of the data. Data is not available for reads until the member transitions to state *secondary* (page 9).
5	STARTUP2 (page 101)	The member has joined the set and is running an initial sync.
6	UNKNOWN (page 101)	The member's state, as seen from another member of the set, is not yet known.
7	ARBITER (page 101)	*Arbiters* (page **??**) do not replicate data and exist solely to participate in elections.
8	DOWN (page 101)	The member, as seen from another member of the set, is unreachable.
9	ROLLBACK (page 101)	This member is actively performing a *rollback* (page 26). Data is not available for reads.
10	REMOVED (page 101)	This member was once in a replica set but was subsequently removed.

States

Core States

PRIMARY
: Members in PRIMARY (page 100) state accept write operations. A replica set has at most one primary at a time. A SECONDARY (page 100) member becomes primary after an *election* (page 22). Members in the PRIMARY (page 100) state are eligible to vote.

SECONDARY
: Members in SECONDARY (page 100) state replicate the primary's data set and can be configured to accept read

operations. Secondaries are eligible to vote in elections, and may be elected to the PRIMARY (page 100) state if the primary becomes unavailable.

ARBITER

Members in ARBITER (page 101) state do not replicate data or accept write operations. They are eligible to vote, and exist solely to break a tie during elections. Replica sets should only have a member in the ARBITER (page 101) state if the set would otherwise have an even number of members, and could suffer from tied elections. There should only be at most one arbiter configured in any replica set.

See *Replica Set Members* (page 7) for more information on core states.

Other States

STARTUP

Each member of a replica set starts up in STARTUP (page 101) state. `mongod` then loads that member's replica set configuration, and transitions the member's state to STARTUP2 (page 101). Members in STARTUP (page 101) are not eligible to vote, as they are not yet a recognized member of any replica set.

STARTUP2

Each member of a replica set enters the STARTUP2 (page 101) state as soon as `mongod` finishes loading that member's configuration, at which time it becomes an active member of the replica set. The member then decides whether or not to undertake an initial sync. If a member begins an initial sync, the member remains in STARTUP2 (page 101) until all data is copied and all indexes are built. Afterwards, the member transitions to RECOVERING (page 101).

RECOVERING

A member of a replica set enters RECOVERING (page 101) state when it is not ready to accept reads. The RECOVERING (page 101) state can occur during normal operation, and doesn't necessarily reflect an error condition. Members in the RECOVERING (page 101) state are eligible to vote in elections, but are not eligible to enter the PRIMARY (page 100) state.

A member transitions from RECOVERING (page 101) to SECONDARY (page 100) after replicating enough data to guarantee a consistent view of the data for client reads. The only difference between RECOVERING (page 101) and SECONDARY (page 100) states is that RECOVERING (page 101) prohibits client reads and SECONDARY (page 100) permits them. SECONDARY (page 100) state does not guarantee anything about the staleness of the data with respect to the primary.

Due to overload, a *secondary* may fall far enough behind the other members of the replica set such that it may need to *resync* (page 75) with the rest of the set. When this happens, the member enters the RECOVERING (page 101) state and requires manual intervention.

Members in SECONDARY (page 100) state can be forced into state RECOVERING (page 101) to prevent client reads by using _maintenance **mode_**. You can toggle maintenance mode with the `replSetMaintenance` command. Maintenance mode is also enabled automatically by the `compact` and `touch` commands. While maintenance mode is on, a member will remain in RECOVERING (page 101) state and will reject client reads.

UNKNOWN

Members that have never communicated status information to the replica set are in the UNKNOWN (page 101) state.

DOWN

Members that lose their connection to the replica set are seen as DOWN (page 101) by the remaining members of the set.

REMOVED

Members that are removed from the replica set enter the REMOVED (page 101) state. When members enter the REMOVED (page 101) state, the logs will mark this event with a `replSet REMOVED` message entry.

ROLLBACK

Whenever the replica set replaces a *primary* in an election, the old primary may contain documents that did not

replicate to the *secondary* members. In this case, the old primary member reverts those writes. During *rollback* (page 26), the member will have ROLLBACK (page 101) state.

FATAL
A member in FATAL (page 102) encountered an unrecoverable error. The member must be shut down and restarted; a resync may be required as well.

Read Preference Reference

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. However, by distributing some or all reads to secondary members of the replica set, you can improve read throughput or reduce latency for an application that does not require fully up-to-date data.

Read Preference Mode	Description
primary (page 102)	Default mode. All operations read from the current replica set *primary*.
primaryPreferred (page 102)	In most situations, operations read from the *primary* but if it is unavailable, operations read from *secondary* members.
secondary (page 102)	All operations read from the *secondary* members of the replica set.
secondaryPreferred (page 103)	In most situations, operations read from *secondary* members but if no *secondary* members are available, operations read from the *primary*.
nearest (page 103)	Operations read from member of the *replica set* with the least network latency, irrespective of the member's type.

Read Preference Modes

primary
All read operations use only the current replica set *primary*. This is the default. If the primary is unavailable, read operations produce an error or throw an exception.

The primary (page 102) read preference mode is not compatible with read preference modes that use *tag sets* (page 31). If you specify a tag set with primary (page 102), the driver will produce an error.

primaryPreferred
In most situations, operations read from the *primary* member of the set. However, if the primary is unavailable, as is the case during *failover* situations, operations read from secondary members.

When the read preference includes a *tag set* (page 31), the client reads first from the primary, if available, and then from *secondaries* that match the specified tags. If no secondaries have matching tags, the read operation produces an error.

Since the application may receive data from a secondary, read operations using the primaryPreferred (page 102) mode may return stale data in some situations.

> **Warning:** Changed in version 2.2: mongos added full support for read preferences.
> When connecting to a mongos instance older than 2.2, using a client that supports read preference modes, primaryPreferred (page 102) will send queries to secondaries.

secondary
Operations read *only* from the *secondary* members of the set. If no secondaries are available, then this read operation produces an error or exception.

Most sets have at least one secondary, but there are situations where there may be no available secondary. For example, a set with a primary, a secondary, and an *arbiter* may not have any secondaries if a member is in recovering state or unavailable.

When the read preference includes a *tag set* (page 31), the client attempts to find secondary members that match the specified tag set and directs reads to a random secondary from among the *nearest group* (page 32). If no secondaries have matching tags, the read operation produces an error. [14]

Read operations using the `secondary` (page 102) mode may return stale data.

`secondaryPreferred`

In most situations, operations read from *secondary* members, but in situations where the set consists of a single *primary* (and no other members), the read operation will use the set's primary.

When the read preference includes a *tag set* (page 31), the client attempts to find a secondary member that matches the specified tag set and directs reads to a random secondary from among the *nearest group* (page 32). If no secondaries have matching tags, the client ignores tags and reads from the primary.

Read operations using the `secondaryPreferred` (page 103) mode may return stale data.

`nearest`

The driver reads from the *nearest* member of the *set* according to the *member selection* (page 32) process. Reads in the `nearest` (page 103) mode do not consider the member's *type*. Reads in `nearest` (page 103) mode may read from both primaries and secondaries.

Set this mode to minimize the effect of network latency on read operations without preference for current or stale data.

If you specify a *tag set* (page 31), the client attempts to find a replica set member that matches the specified tag set and directs reads to an arbitrary member from among the *nearest group* (page 32).

Read operations using the `nearest` (page 103) mode may return stale data.

Note: All operations read from a member of the nearest group of the replica set that matches the specified read preference mode. The `nearest` (page 103) mode prefers low latency reads over a member's *primary* or *secondary* status.

For `nearest` (page 103), the client assembles a list of acceptable hosts based on tag set and then narrows that list to the host with the shortest ping time and all other members of the set that are within the "local threshold," or acceptable latency. See *Member Selection* (page 32) for more information.

Use Cases

Depending on the requirements of an application, you can configure different applications to use different read preferences, or use different read preferences for different queries in the same application. Consider the following applications for different read preference strategies.

Maximize Consistency To avoid *stale* reads under all circumstances, use `primary` (page 102). This prevents all queries when the set has no *primary*, which happens during elections, or when a majority of the replica set is not accessible.

[14] If your set has more than one secondary, and you use the `secondary` (page 102) read preference mode, consider the following effect. If you have a *three member replica set* (page 17) with a primary and two secondaries, and if one secondary becomes unavailable, all `secondary` (page 102) queries must target the remaining secondary. This will double the load on this secondary. Plan and provide capacity to support this as needed.

Maximize Availability To permit read operations when possible, Use `primaryPreferred` (page 102). When there's a primary you will get consistent reads, but if there is no primary you can still query *secondaries*.

Minimize Latency To always read from a low-latency node, use `nearest` (page 103). The driver or `mongos` will read from the nearest member and those no more than 15 milliseconds [15] further away than the nearest member.

`nearest` (page 103) does *not* guarantee consistency. If the nearest member to your application server is a secondary with some replication lag, queries could return stale data. `nearest` (page 103) only reflects network distance and does not reflect I/O or CPU load.

Query From Geographically Distributed Members If the members of a replica set are geographically distributed, you can create replica tags based that reflect the location of the instance and then configure your application to query the members nearby.

For example, if members in "east" and "west" data centers are *tagged* (page 76) `{'dc': 'east'}` and `{'dc': 'west'}`, your application servers in the east data center can read from nearby members with the following read preference:

```
db.collection.find().readPref( { mode: 'nearest',
                                 tags: [ {'dc': 'east'} ] } )
```

Although `nearest` (page 103) already favors members with low network latency, including the tag makes the choice more predictable.

Reduce load on the primary To shift read load from the primary, use mode `secondary` (page 102). Although `secondaryPreferred` (page 103) is tempting for this use case, it carries some risk: if all secondaries are unavailable and your set has enough *arbiters* to prevent the primary from stepping down, then the primary will receive all traffic from clients. If the primary is unable to handle this load, queries will compete with writes. For this reason, use `secondary` (page 102) to distribute read load to replica sets, not `secondaryPreferred` (page 103).

Read Preferences for Database Commands

Because some *database commands* read and return data from the database, all of the official drivers support full *read preference mode semantics* (page 102) for the following commands:

- `group`
- `mapReduce` [16]
- `aggregate` [17]
- `collStats`
- `dbStats`
- `count`
- `distinct`
- `geoNear`
- `geoSearch`
- `geoWalk`

[15] This threshold is configurable. See `localPingThresholdMs` for `mongos` or your driver documentation for the appropriate setting.

[16] Only "inline" `mapReduce` operations that do not write data support read preference, otherwise these operations must run on the *primary* members.

[17] Using the `$out` pipeline operator forces the aggregation pipeline to run on the primary.

- `parallelCollectionScan`

New in version 2.4: `mongos` adds support for routing commands to shards using read preferences. Previously `mongos` sent all commands to shards' primaries.