{ blog: mongolab }

- home
- archive

Replication Lag & The Facts of Life

By dampier on 2013/03/18 in mongodb

So you're checking in on your latest awesome application one day — it's really getting traction! You're proud of its uptime record, thanks in part to the MongoDB replica set underneath it. But now ... something's wrong. Users are complaining that some of their data has gone missing. Others are noticing stuff they deleted has suddenly reappeared. What's going on?!?

Don't worry... we'll get to the bottom of this! In doing so, we'll examine a source of risk that's easy to overlook in a MongoDB application: *replication lag* — what it means, why it happens, and what you can do about it.

Here's what we're going to cover:

Continuing this cautionary tale... Seriously, wtf?! You were doing everything right!

Using MongoDB with a well-designed schema and lovingly-tuned indexes, your application back-end has been handling thousands of transactions per second without breaking a sweat. You've got multiple nodes arranged in a replica set with no single point of failure. Your application tier's Mongo driver connections are aware of the replica set and can follow changes in the PRIMARY node during failover. All critical writes are "safe" writes. Your app has been up without interruption for almost six months now! *How could this have happened?*

This unsettling situation has the hallmarks of an insidious foe in realm of high-availability data stewardship: **unchecked replication lag**.

Closely monitoring a MongoDB replica set for replication lag is critical.

What is replication lag?

As you probably know, like many data stores MongoDB relies on *replication* — making redundant copies of data — to meet design goals around availability.

 In a perfect world, data replication would be instantaneous; but in reality, thanks to pesky laws of physics, some delay is inevitable — it's a **fact of life**. We need to be able to reason about how it affects us so as to manage around the phenomenon appropriately. Let's start with definitions...

For a given secondary node, ***replication lag*** is the delay between the time an operation occurs on the primary and the time that same operation gets applied on the secondary.

For the replica set as a whole, replication lag is (for most purposes) the smallest replication lag found among all its secondary nodes.

In a smoothly running replica set, all secondaries closely follow changes on the primary, fetching each group of operations from its oplog and replaying them approximately as fast as they occur. That is, replication lag remains as close to zero as possible. Reads from any node are then reasonably consistent; and, should the current primary become unavailable, the secondary that assumes the PRIMARY role will be able to serve to clients a dataset that is almost identical to the original.

For a variety of reasons, however, secondaries may fall behind. Sometimes elevated replication lag is transient and will remedy itself without intervention. Other times, replication lag remains high or continues to rise, indicating a systemic problem that needs to be addressed. In either case, the larger the replication lag grows and the longer it remains that way, the more exposure your database has to the associated risks.

Why is lag problematic?

Significant replication lag creates failure modes that can be problematic for a MongoDB database deployment that is meant to be highly available. Here's why:

- If your replica set fails over to a secondary that is significantly behind the primary, a lot of un-replicated data may be on the original primary that will need to be manually reconciled. This will be painful or impossible if the original primary is unrecoverable.
- If the failed primary cannot be recovered quickly, you may be forced to run on a node whose data is not up-to-date, or forced to take down your database altogether until the primary can be recovered.
- If you have only one secondary, and it falls farther behind than the earliest history retained in the primary's oplog, your secondary will require a full resynchronization from the primary.
    - During the resync, your cluster will lack the redundancy of a valid secondary; the cluster will not return to high availability until the entire data set is copied.
    - If you only take backups from your secondary (which we highly recommend), backups must be suspended for the duration of the resync.
- Replication lag makes it more likely that results of any read operations distributed across secondaries will be inconsistent.
- A "safe" write with 'w' > 1 — i.e., requiring multiple nodes acknowledge the write before it returns — will incur latency proportional to the current replication lag, and/or may time out.

Strictly speaking, the problem of replication lag is distinct from the problem of data durability. But as the last point above regarding multi-node write concern illustrates, the two concepts are most certainly linked. Data that has not yet been replicated is not completely protected from single-node failure; and

client writes specified to be safe from single-node failure must block until replication catches up to them.

What causes a secondary to fall behind?

In general, a secondary falls behind on replication any time it cannot keep up with the rate at which the primary is writing data. Some common causes:

Secondary is weak

To have the best chance of keeping up, a secondary host should match the primary host's specs for CPU, disk IOPS, and network I/O. If it's outmatched by the primary on any of these specs, a secondary may fall behind during periods of sustained write activity. Depending on load this will, at best, create brief excursions in replication lag and, at worst, cause the secondary to fall irretrievably behind.

Bursty writes

In the wake of a burst of write activity on the primary, a secondary may not be able to fetch and apply the ops quickly enough. If the secondary is underpowered, this effect can be quite dramatic. But even when the nodes have evenly matched specs, such a situation is possible. For example, a command like:

```
1 db.coll.update({x: 7}, {$set: {y: 42}}, {multi: true})
```

can place an untold number of separate “update” ops in the primary's oplog. To keep up, a secondary must fetch those ops (max 4MB at a time for each `getMore` command!), read into RAM any index and data pages necessary to satisfy each `_id` lookup (remember: each oplog entry references a single target document by `_id`; the original query about “x” is never directly reflected the oplog), and finally perform the update op, altering the document and placing the corresponding entry into its oplog; and it must do all this in the same amount of time that the primary does merely the last step. Multiplied by a large enough number of ops, that disparity can amount to a noticeable lag.

Map/reduce output

A specific type of the extreme write burst scenario might be a command like:

```
1 db.coll.mapReduce( ... { out: other_coll ... })
```

From the point of view of the oplog, the entire output collection basically materializes at once, from which point the replication to the secondary plays out as above.

Index build

It may surprise you to learn that, even if you build an index in the background on the primary, it will be built in the foreground on each secondary. There is currently no way to build indexes in the background on secondary nodes (cf. SERVER-2771). Therefore, whenever a secondary builds an index, it will **block all other operations**, including replication, for the duration. If the index builds quickly, this may not be a problem; but long-running index builds can swiftly manifest as significant replication lag.

Secondary is locked for backup

One of the suggested methods for backing up data in a replica set involves explicitly locking a secondary against changes while the backup is taken. Assuming the primary is still conducting business as usual, of course replication lag will climb until the backup is complete and the lock is released.

Secondary is offline

Similarly, if the secondary is not running or cannot reach the primary for whatever reason, it cannot make progress against the replication backlog. When it rejoins the replica set, the replication lag will naturally reflect the time spent away.

How do I measure lag?

Run the `db.printSlaveReplicationInfo()` command

To determine the current replication lag of your replica set, you can use the mongo shell and run the **`db.printSlaveReplicationInfo()`** command.

```
rs-ds046297:PRIMARY db.printSlaveReplicationInfo()

source: ds046297-a1.mongolab.com:46297
syncedTo: Tue Mar 05 2013 07:48:19 GMT-0800 (PST)
      = 7475 secs ago (2.08hrs)
source: ds046297-a2.mongolab.com:46297
syncedTo: Tue Mar 05 2013 07:48:19 GMT-0800 (PST)
      = 7475 secs ago (2.08hrs)
```

More than 2 hours — whoa, isn't that a lot? Maybe!

See, those "syncedTo" times don't have much to do with the clock on the wall; they're just the timestamp on the last operation that the replica has copied over from the PRIMARY. If the last write operation on the PRIMARY happened 5 minutes ago, then yes: 2 hours is a lot. On the other hand, if the last op was 2.08 hours ago, then this is golden!

To fill in that missing piece of the story, we can use the **`db.printReplicationInfo()`** command.

```
rs-ds046297:PRIMARY db.printReplicationInfo()

configured oplog size:   1024MB
log length start to end: 5589secs (1.55hrs)
oplog first event time:  Tue Mar 05 2013 06:15:19 GMT-0800 (PST)
oplog last event time:   Tue Mar 05 2013 07:48:19 GMT-0800 (PST)
now:                     Tue Mar 05 2013 09:53:07 GMT-0800 (PST)
```

Let's see ... PRIMARY's "oplog last event time" – SECONDARY's "syncedTo" = 0.0. Yay.

As fun as that subtraction may be, it's seldom called for. If there is a steady flow of write operations, the last op on the PRIMARY will usually have been quite recent. Thus, a figure like "2.08 hours" should probably raise eyebrows; you would expect to see a nice low number there instead — perhaps as high as a few seconds. And, having seen a low number, there would be no need to qualify its context with the second command.

Examine the "repl lag" graph in MMS

You can also view recent and historical replication lag using the MongoDB Monitoring Service (MMS) from 10gen. On the Status tab of each SECONDARY node, you'll find the **repl lag** graph:

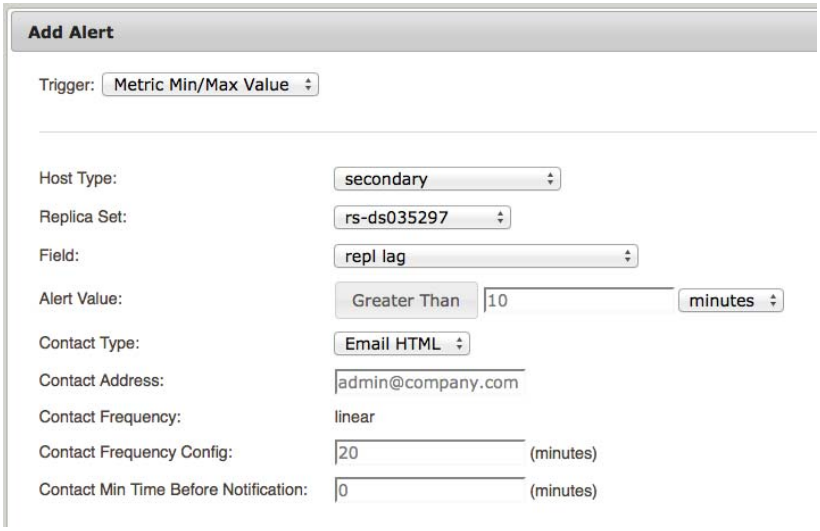
How do I monitor for lag?

It is critical that the replication lag of your replica set(s) be monitored continuously. Since you have to sleep occasionally, this is a job best done by robots. It is essential that these robots be reliable, and that they notify you promptly whenever a replica set is lagging too far behind.

Here are a couple ways you can make sure this is taken care of:

- If MongoLab is hosting your replica set, relax! For any multi-node, highly-available replica set we host for you, you can monitor replication lag in our UI and by default you will receive automated alerts whenever the replication lag exceeds 10 minutes.

- You can also set up an alert using the MMS system. Its exciting new features allow you to configure a replication lag alert:



What can I do to minimize lag?

Out of courtesy (for them or for ourselves), we would like to make those lag-monitoring automata's lives as boring as possible. Here are some tips:

Tip #1: Make sure your secondary has enough horsepower

It's not uncommon for people to run under-powered secondaries to save money — this can be fine if the write load is light. But in scenarios where the write load is heavy, the secondary might not be able to keep up with the primary. To avoid this, you should beef up your secondary so that it's as powerful as your primary.

Specifically, a SECONDARY node should have enough network bandwidth that it can retrieve ops from the PRIMARY's oplog at roughly the rate they're created and also enough storage throughput that it can apply the ops — i.e., read any affected documents and their index entries into RAM, and commit the altered documents back to disk — at that same rate. CPU rarely becomes a bottleneck, but it may need to be considered if there are many index keys to compute and insert for the documents that are being added or changed.

Tip #2: Consider adjusting your write concern

Your secondary may be lagging simply because your primary's oplog is filling up faster than it can be replicated. Even with an equally-brawny SECONDARY node, the PRIMARY will always be capable of depositing 4MB in its memory-mapped oplog in a fraction of the time those same 4MB will need to make it across a TCP/IP connection.

One viable way to apply some back-pressure to the primary might be to adjust your write concern.

If you are currently using a write concern that does not acknowledge writes (aka "fire-and-forget" mode), you can change your write concern to require an acknowledgement from the primary (`w:1`) and/or a write to the primary's journal (`j:true`). Doing so will slow down the rate at which the concerned connection can generate new ops needing replication.

Other times it may be appropriate to use a 'w' > 1 or a 'w' set to "`majority`" to ensure that each write to the cluster is replicated to more than one node before the command returns. Requiring confirmation that a write has replicated to secondaries will effectively guarantee that those secondaries have caught up (at least up to the timestamp of this write) before the next command on the same connection can produce more ops in the backlog.

As previously alluded to, choosing the most appropriate write concern for the data durability requirements of your application — or for particular critical write operations within the application — is something you must give thought to irrespective of the replication lag issue we're focusing on here. But you should be aware of the interrelationship: just as the durability guarantee of w>1 can be used as a means of forcing a periodic "checkpoint" on replication, excessive replication lag can show up as a surprisingly high latency (or timeout) for that very occasional critical write operation where you've used "`w: majority`" to make sure it's truly committed.

Adjust to taste

Having servers acknowledge every write can be a big hit to system throughput. If it makes sense for your application, you can amortize that penalty by doing inserts in batches, requiring acknowledgement only at the end of each batch. The smaller the batch, the greater the back-pressure on PRIMARY data creation rate, and correspondingly greater potential adverse impact to overall throughput.

Don't overdo it

Using a large value for 'w' can itself be problematic. It represents a demand that *w* nodes finish working through their existing backlog before the command returns. So, if replication lag is high (in the sense of there being a large volume of data waiting to copy over) when the write command is issued, the command execution time will suffer a proportionally high latency. Also, if enough nodes go offline such that 'w' cannot be satisfied, you have effectively locked up your database. This is basically the opposite of "high availability."

Tip #3: Plan for index builds

As mentioned earlier, an index build on a secondary is a foreground, blocking operation. If you're going to create an index that is sizeable, perhaps you can arrange to do it during a period of low write activity on the primary. Alternately, if you have more than one secondary, you can follow the steps here to minimize the impact of building large indexes.

Tip #4: Take backups without blocking

Earlier we discussed the technique of locking the secondary to do a backup. There are other alternatives to consider here, including filesystem snapshots and "point-in-time" backups using the "`--oplog`" option of `mongodump` without locking. These are preferable to locking the secondary during a period of active writes if there's any chance you'll use the secondary for anything other than backups.

Tip #5: Be sure capped collections have an `_id` field & a unique index

Reliable replication is not possible unless there is a unique index on the `_id` field. Before MongoDB version 2.2, capped collections did not have an `_id` field or index by default. If you have a collection like this, you should create an index on the `_id` field, specifying `unique: true`. Failing to do this can, in certain situations, cause replication to **halt entirely**. So ... this should not be regarded as optional.

Tip #6: Check for replication errors

If you see that replication lag is only increasing (and never falling), your replica set could be experiencing replication errors. To check for errors, run `rs.status()` and look at the `errmsg` field in the result. Additionally, check the log file of your secondary and look for error messages there.

One specific example: if you see "`RS102 too stale to catch up`" in the secondary's `mongodb.log` or in the `errmsg` field when running `rs.status()`, it means that secondary has fallen so far behind that there is not enough history retained by the primary (its "oplog size") to bring it up to date. In this case, your secondary will require a full resynchronization from the primary.

In general, though, what you do in response to an error depends on the error. Sometimes you can simply restart the `mongod` process for your secondary; but the majority of the time you will need to understand

the root cause of the error before you can fix the problem.

Don’t let replication lag take you by surprise.

At the end of the day, replication lag is just one more source of risk in any high-availability system that we need to understand and design around. Striking the right balance between performance and “safety” of write operations is an exercise in risk management — the “right” balance will be different in different situations. For an application on a tight budget with occasional spikes in write volume, for example, you might decide that a large replication lag in the wake of those spikes is acceptable given the goals of the application, and so an underpowered secondary makes sense. At the opposite extreme, for an application where every write is precious and sacred, the required “majority” write concern will mean you have essentially no tolerance for replication lag above the very minimum possible. The good news is that MongoDB makes this all very configurable, even on an operation by operation basis.

We hope this article has given you some insight into the phenomenon of replication lag that will enable you to reason about the risk it poses for a high-availability MongoDB application, and armed you with some tools for managing it. As always, let us know if we can help!

Related Posts:

- No related posts found

← Object Modeling in Node.js with Mongoose
MongoLab at Overdriver.com →

5 Comments MongoLab's Blog Login

Sort by Best Recommend Share

Join the discussion…

Manan Shah • a year ago
Is there a way to check the lag in milliseconds of time? Currently MMS show the lag in seconds.
• Reply • Share ›

Max Hodges ➔ Manan Shah • 9 days ago
is milliseconds necessary?
• Reply • Share ›

Tim Hawkins • 2 years ago
One thing that folks often fail to take into account is replication network bandwidth, once you get beyond a certain size of cluster (about 6-7 machines), your replication traffic on a system that is handling a lot of writes, is going to be an appreciable percentage of the overall available bandwidth of your internal network. Latency can build up soon, and your primary is going to be sending out a lot of oplog copies to each secondary, which can result in saturation of its nic. If you lump application traffic in on top of that, then the bandwidth requirements can become extreem.
• Reply • Share ›

jean • 2 years ago
Hello! this is a very useful post. I have a replica set of 3 members.

currently I have problems with 1 member, It has a laggin and felt out behind the master.

The other secondary is ok and always uptime. The cluster doesn't have traffic, only I did a mongorestore of 26 GB and this replica can't replicate the data quickly

My master is ssd and secondaries is sata.

The oplog is by default. And I saw a lot of read operation when I did mongotop over the slow replica
• Reply • Share ›

Max Hodges ➔ jean • 9 days ago
>My master is ssd and secondaries is sata.

To have the best chance of keeping up, a secondary host should match the primary host's specs for CPU, disk IOPS, and network I/O. If it's outmatched by the primary on any of these specs, a secondary may fall behind during periods of sustained write activity. Depending on load this will, at best, create brief excursions in replication lag and, at worst, cause the secondary to fall irretrievably behind.
• Reply • Share ›

ALSO ON MONGOLAB'S BLOG WHAT'S THIS?

Respondly explains why devs love Meteor and MongoDB
8 comments • 4 months ago
Chris Chang — Hi EJ,Great feedback. I'll reach out to Tim to see if he has anything to share.Thanks for reading!

Build your own lead capture page with Meteor and MongoDB in minutes
1 comment • 2 years ago
Toussaint Gouthier — this was a very good tip on how to create lead capture pagecheck out my website …

A Primer on Geospatial Data and MongoDB
2 comments • 6 months ago
wsmoak — I just ran into a similar thing -- you can insert arbitrary values for a Point,

Harnessing the power of Twitter and MongoDB
9 comments • 2 years ago
Ben — I just finished a quick fix on my fork https://github.com/BenCoDev/tw...

About

This blog is brought to you by MongoLab, the fully-managed MongoDB Database-as-a-Service (DBaaS) platform that automates the operational aspects of running MongoDB in the cloud.

We hope you enjoy it!

Welcome • Plans & Features • Docs & Support

Search

Search...

Recent Posts

- Fully managed Dedicated MongoDB plans on Heroku
- Respondly explains why devs love Meteor and MongoDB
- Run SQL Queries on MongoLab
- Fully managed MongoDB replica sets for $15
- Custom firewalls for your MongoDB deployment(s)

Featured partners

- Amazon (AWS)
- Microsoft Azure
- Google Cloud Platform (GCP)

You might also like

- Indexing tips at the June 19 2012 SF MongoDB User Group
- Round Two: Node Knockout 2013!
- Fully managed Dedicated MongoDB plans on Heroku
- Run SQL Queries on MongoLab