Parse

Tuesday, December 4, 12

Hi! My name is Charity Majors, and I am a systems engineer at Parse.

Parse is a platform for mobile developers.

You can use our apis to build apps for iOS, Android, and Windows phones. We take care of all of the provisioning and scaling for backend services, so you can focus on building your app and user experience.

Replica sets

- Always use replica sets
- Distribute across Availability Zones
- Avoid situations where you have even # voters
- More voters are better than fewer

Parse

Tuesday, December 4, 12

First, the basics.

* Always run with replica sets. Never run with a single node, unless you really hate your data. And always distribute your replica set members across as many different regions as possible. If you have three nodes, use three regions. Do not put two nodes in one region and one node in a second region. Remember, you need at least two nodes to form a quorum in case of network split. And an even number of nodes can leave you stuck in a situation where they can't elect a master. If you need to run with an even number of nodes temporarily, either assign more votes to some nodes or add an arbiter. But always, always think about how to protect yourself from situations where you can't elect a master. Go for more votes rather than fewer, because it's easier to subtract if you have too many than to add if you have too few.

** Remember, if you get in to a situation where you have only one node, you have a situation where you have no way to add another node to the replica set. There was one time very early on when we were still figuring mongo out, and we had to recover from an outage by bringing up a node from snapshot with the same hostname so it would be recognized as a member of the same replica set. Bottom line, you just really don't want to be in this situation. Spread your eggs around in lots of baskets.

Snapshots

- Snapshot often
- Lock Mongo
- Set snapshot node to priority = 0
- Always warm up a snapshot before promoting
- Warm up both indexes and data

Parse

Tuesday, December 4, 12

Snapshots

* Snapshot regularly. We snapshot every 30 minutes. EBS snapshot actually does a differential backup, so subsequent snapshots will be faster the more frequently you do them.

* Make sure you use a snapshot script that locks mongo. It's not enough to just use ec2-create-snapshot on the RAID volumes, you also need to lock mongo beforehand and unlock it after. We use a script called ec2-consistent-snapshot, though I think we may have modified it to add mongo support.

* Always set your snapshot node to config priority = 0. This will prevent it from ever getting elected master. You really, really do not want your snapshotting host to ever become master, or your site will go down. We also like to set our primary priority to 3, and all non-snapshot secondaries to 2, because priority 1 isn't always visible from rs.conf(). That's just a preference of ours.

* Never, ever switch primary over to a newly restored snapshot. Something a lot of people don't seem to realize is that EBS blocks are actually lazy-loaded off S3. You need to warm your fresh secondaries up. I mean, you think loading data into RAM from disk is bad, try loading into RAM from S3. There's just a *tiny* bit of latency there.

Warming up

Lots of people seem to do this in different ways, and it kind of depends on how much data you have. If you have less data than you have RAM, you can just use dd or vmtouch to load entire databases into memory. If you have more data than RAM, it's a little bit trickier.

The way we do it is, first we run a script on the primary. It gets the current ops every quarter of a second or so for an hour, then sorts by most frequently accessed collections. Then we take that list of collections and feed it into a warmup script on the secondary, which reads all the collections and indexes into memory. The script is parallelized, but it still takes several hours to complete. You can also read collections into memory by doing a full table scan, or a natural sort.

God, what I wouldn't give for block-level replication like Amazon's RDS.

Chef everything

- Role attributes for backup volumes, cluster names
- Nodes are disposable
- Delete volumes and aws attributes, run chef-client to reprovision

Parse

Tuesday, December 4, 12

Chef

Moving along … chef! Everything we have is fully chef'd. It only takes us like 5 minutes to bring up a new node from snapshot. We use the opscode MongoDB and AWS cookbooks, with some local modifications so they can handle PIOPS and the ebs_optimized dedicated NICs. We haven't open sourced these changes, but we probably can, if there's any demand for them. It looks like this:

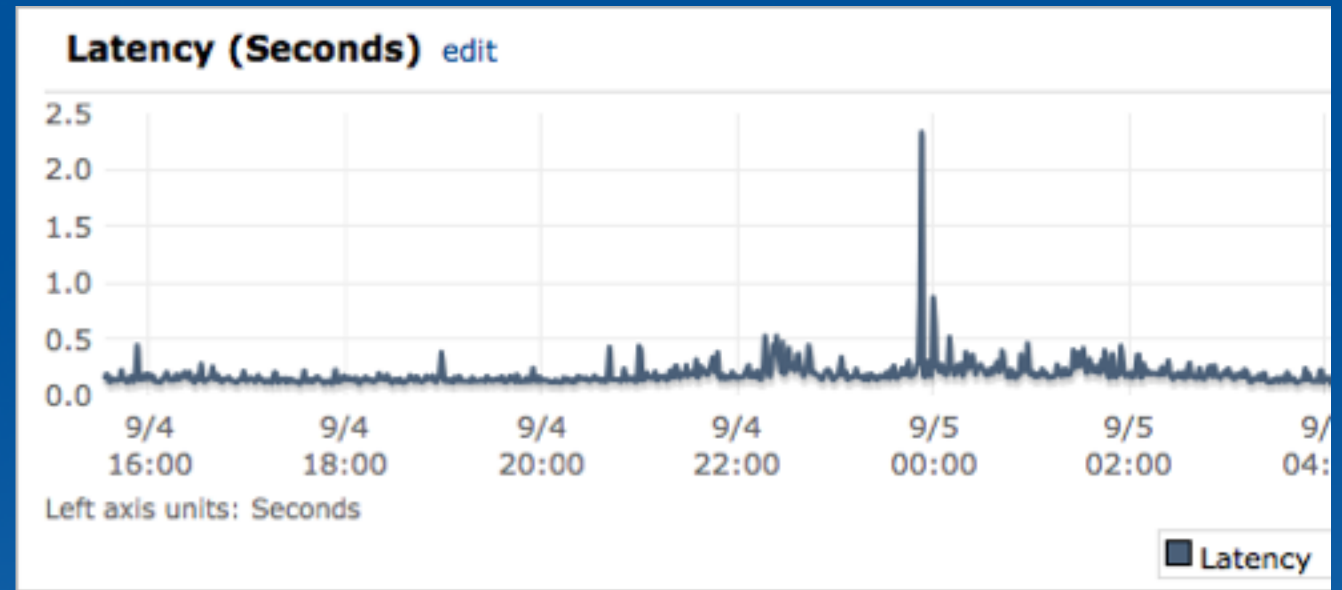
```
$ knife ec2 server create -r "role[mongo-replset1-iops]" -f m2.4xlarge -G db -x ubuntu --node-name db36 -I ami-xxxxxxxx -Z us-east-1d -E production
```

There are some neat things in the mongo cookbook. You can create a role attribute to define the cluster name, so it automatically comes up and joins the cluster. The backup volumes for a cluster are also just attributes for the role. So it's easy to create a mongo backups role that automatically backs up whatever volumes are pointed to by that attribute.

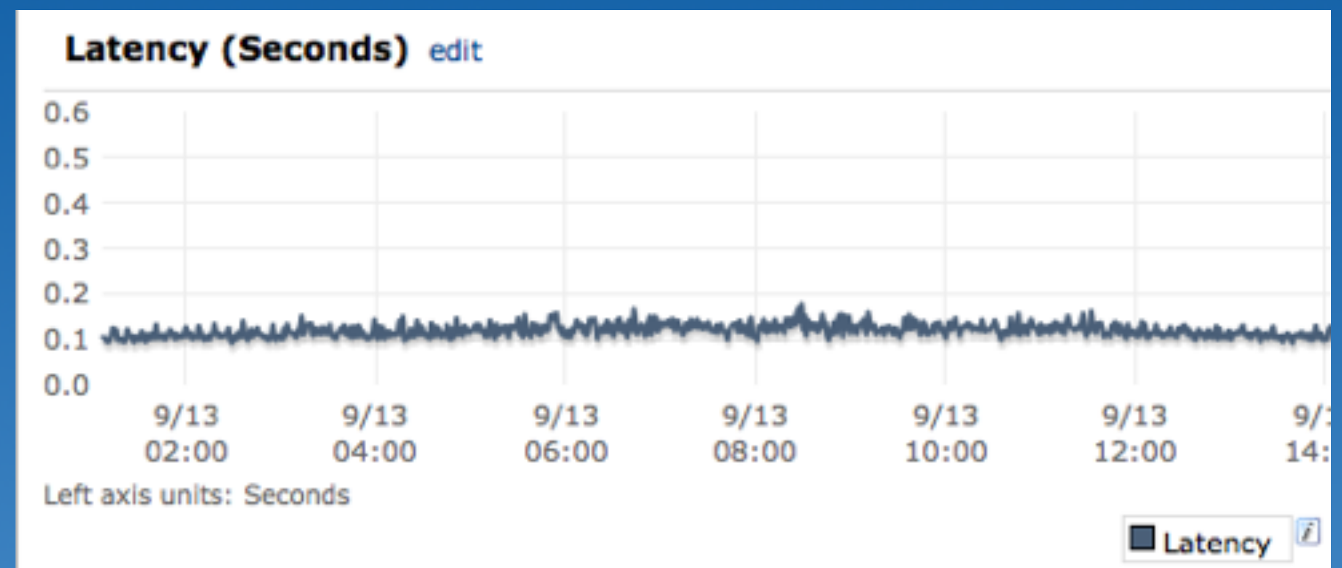
We use the m2.4xlarge flavor, which has like 68 gigs of memory. We have about a terabyte of data per replica set, so 68 gigs is just barely enough for the working set to fit into memory.

We used to use four EBS volumes RAID 10'd, but we don't even bother with RAID 10 anymore, we just stripe PIOPS volumes. It's faster for us to reprovision a replica set member than repairing the RAID array. If an EBS volume dies, or the secondary falls too far behind, or whatever, we just delete the volumes, remove the AWS attributes for the node in the chef node description, and re-run chef-client. It reprovisions new volumes for us from the latest snapshot in a matter of minutes. For most problems, it's faster for us to destroy and rebuild than attempt any sort of repair.

Before PIOPS:



After PIOPS:



Parse

Tuesday, December 4, 12

P-IOPS

And … we use PIOPS. We switched to Provisioned IOPS literally as soon as it was available. As you can see from this graph, it made a *huge* difference for us.

These are end-to-end latency graphs in Cloudwatch, from the point a request enters the ELB til the response goes back out. Note the different Y-axis! order of magnitude difference. The top Y-axis goes up to 2.5, the bottom one goes up to 0.6.

EBS is awful. It's bursty, and flaky, and just generally everything you DON'T want in your database hardware. As you can see here in the top graph, using 4 EBS volumes raid 10'd, we had ebs spikes all the time. Any time one of the four ebs volumes had any sort of availability event, our end to end latency took a hit. With PIOPS, our average latency dropped in half and went almost completely flat around 100 milliseconds.

So yes. Use PIOPS. Until recently you could only provision 1k iops per volume, but you can now provision volumes with up to 2000 iops per volume. And they guarantee a variability of less than .1%, which is exactly what you want in your database hardware.

Filesystem & misc

- Use ext4
- Raise file descriptor limits (cat /proc/<pid>/limits to verify)
- Sharding. Eventually you must shard.

Parse

Tuesday, December 4, 12

Misc

Some small, miscellaneous details:

* Remember to raise your file descriptor limits. And test that they are actually getting applied. The best way to do this is find the pid of your mongodb process, and type “cat /proc/<pid>/limits. We had a hard time getting sysvinit scripts to properly apply the increased limits, so we converted to use upstart and have had no issues. I don’t know if ubuntu no longer supports sysvinit very well, or what.

* We use ext4. Supposedly either ext4 or xfs will work, but I have been scarred by xfs file corruption way too many times to ever consider that. They say it’s fixed, but I have like xfs PTSD or something.

* Sharding -- at some point you have to shard your data. The mongo built-in sharding didn’t work for us for a variety of reasons I won’t go into here. We’re doing sharding at the app layer, the goal is to

Parse runs on MongoDB

- DDoS protection and query profiling
- Billing and logging analytics
- User data

Parse

Tuesday, December 4, 12

In summary, we are very excited about MongoDB. We love the fact that it fails over seamlessly between Availability Zones during an AZ event. And we value the fact that its flexibility allows us to build our expertise and tribal knowledge around one primary database product, instead of a dozen different ones.

In fact, we actually use MongoDB in at least three or four distinct ways. We use it for a high-writes DDoS and query analyzer cluster, where we process a few hundred thousand writes per minute and expire the data every 10 minutes. We use it for our logging and analytics cluster, where we analyze all our logs from S3 and generate billing data. And we use it to store all the app data for all of our users and their mobile apps.

Something like Parse wouldn't even be possible without a nosql product as flexible and reliable as Mongo is. We've built our business around it, and we're very excited about its future.

Also, we're hiring. See me if you're interested. :)

Thank you! Any questions?

Parse