written by Adam Horwich on 5 March 2013 in devops, MetaBroadcast

hadoop in the cloud

Now, I know, you're going to say "but Adam, there's a reason Elastic Map Reduce (EMR) exists," but there's a couple of reasons why I'm talking about Hadoop best practices in the cloud. Today I'm going to talk about the virtues of managing your own cluster, and 3 tips on how you can integrate with AWS to enhance and optimise your cluster.

better the devil you know

Oh Hadoop, how you infuriate me with your spurious failures and endless bugs, but how fantastic you can actually be when it comes down to it. I've been fighting with Hadoop a lot this past year, from a Region Server domino apocalypse, to the seemingly impossible job of duplicating a cluster. I'll preface ALL of these complaints with the clarification that we're currently running Cloudera's CDH3u0 release of Hadoop (don't worry, we're changing that, and I'm sure I'll be blog-raging about it soon enough). But to make the most of what you've got, I've been researching better ways of using resources available. There's, of course, always been the option of using Amazon's EMR service, but we originally built our cluster before that existed as a product, and have built our services around a standardised Hadoop cluster, with local DataNodes. This blog post will be about adding in some nice EMR style features to your dedicated Hadoop cluster running in AWS.

availability-zone know thyself

Hadoop clusters can be expensive beasts, because to mitigate against Amazon Availability Zone (AZ) outages, you really ought to be spanning your cluster over multiple AZs. While prices have significantly improved as of late, there's a cost incurred with transferring data between Zones, and there's also added latency and contention, as the data is no longer datacentre local. As a result it's a good idea to add in 'rack awareness' to your cluster. Brad Helund put together a fantastic article on Hadoop Cluster behaviour. This will mean that the TaskTrackers running your map and reduce tasks will favour data local to their 'rack.' In Amazon's model, we have no privileged information about physical rack layouts, but we can extend that model to Regions and Availability Zones instead. There's a catch though. Unhelpfully documentation is thin, and contrary to some Hadoop documentation (unless this is something that's been revised in future releases), when a TaskTracker registers with the JobTracker, or the DataNode registers with the NameNode, it sends its IP and port to the master. Amazon IPs are terribly unhelpful, because there's no convenient reverse DNS lookup available. If you really want that, you have to currently email them! No, so instead it's a good idea to utilise AWS CLI tools and query the IP you've been sent with your known infrastructure. As node registration is an infrequent act, the speed of the script doesn't have to be lightning (and with AWS

tools, it's never really an option!):

Above gives us the major region (eu-west) and zone within 1c, relevant for our account. This script could be extended to multiple regions, though you may want to tweak it to support things like us-west-1 and us-west-2.

In terms of plumbing, you just need to set up an IAM user with permission to describe instances, and configure your hdfs-site.xml with "topology.script.file.name" and the location of the script above. Despite being in the HDFS-SITE configuration file, it will also be used by the JobTracker when new TaskTrackers report in for duty.

my job's more important than your job

As schedulers go, Hadoop's pretty damn basic out of the box. There's two main contenders for schedulers if you don't want to use the basic one: the fair scheduler, and the capacity scheduler (our release of Hadoop does not have support for other schedulers). The capacity scheduler will try and provide rules for clusters shared between users. It's nice, but it's not great for the kinds of important jobs we have, and fundamentally, it won't preempt jobs if you have higher priorities. We need something that will kill tasks and let your super important time-critical job take the slots. No, the fair scheduler is where it's at. Ironic? Sure!:

That preemption flag will allow tasks to be terminated (no, the job won't fail, it's ok!) if a higher priority job is submitted. We do this by having a 'priority' queue that our jobs can submit to:

The rules here will kick in 5 minutes after a high priority job has been submitted. They will take necessary resources and not get stuck behind slow jobs.

spot me a node would'ya

A more exciting development of late has been the integration of the lucrative Spot Instance market offered by AWS. This allows you to bid on under-utilised resources in AWS for much cheaper prices than a standard on-demand instance. Integrating this with custom metrics and Auto Scaling groups offers some very nice capacity planning options for when clusters get overloaded with jobs. We have some pretty big jobs, that take a while, so we're keen to ensure they complete in a reasonable amount of time. To the extent that we're happy to launch dedicated TaskTrackers without any local data to help speed the job along. Our tests have shown that the impact of non-local data is less than an order of magnitude, and so acceptable when the cluster is overcapacity. Since we are using Spot Instances, and would want to scale down once there are quieter periods in the cluster, it makes sense to only extend the cluster with TaskTrackers. Launching new DataNodes is very risky, when replication comes into play. It's possible to lose data entirely when you're at the mercy of the Spot Instance pricing market.

pushing metrics for the win

Before we can scale our cluster, we need a good metric to work with. We decided on the longest running job time. It's not infallible, but it's pretty good with limits

imposed on the Auto Scaling group. To do this, I enhanced a simple Sensu check which looks at the current running jobs and finds the one which has been running the longest (here's the snippet we care about):

The bulk of this python script is used, based on a supplied threshold, to see if a job has been running too long for our tastes. By using the wonderful boto library, we can set up a very simple IAM user in our AWS account that only has the most basic of permissions:

```
{ "Statement": [ { "Action": [ "cloudwatch:ListMetrics", "cloudwatch:PutMetricData" ], "Effect": "Allow", "Resource": [ "*" ] } ] }
```

This allows our Sensu integrated check to push metric values for slow jobs into AWS, and allow the inbuilt auto-scaling mechanisms for monitoring CloudWatch metrics to make capacity decisions for us.

scaling like a boss

The last part of the puzzle is the Auto Scaling configuration. Setting up a spot instance based Auto Scaling group is as simple as including `--spot-price "0.30"` into your launch configuration. Amazon takes care of the rest. You can then use your Auto Scaling group configuration to set maximum and desired capacities for your task tracker nodes. The important thing here too is that you can allocate EC2 tags in the config. This is something missing from the standard Spot Instance Request API. Because it's an asynchronous request (you make a bid, and then you find out if it's accepted), you can't define tags as part of it. A bit of a flaw if you ask me! With this configuration in place, AWS will automatically be monitoring the slow jobs, and spinning up instances to add capacity, then turning them off once the longest running job time drops back down below your thresholds. Simple?

nothing's easy in the amazon

There are a few flaws. Firstly, the Spot Instance market fluctuates, a lot. And each AZ has different prices based on capacity. With Hadoop nodes in multiple zones, you want to ensure you don't over-bias one zone with your launch configuration so it's best to configure the Auto Scaling group configuration for multiple zones. Drawback here is that it seems the logic for determining which AZ your Spot Instance Request goes to is not very clear, or very good. In principle it should be submitted to any AZ which is below the Spot Instance price you've requested, in practice we've seen our requests get stuck requesting to the same AZ, which has a higher Spot Instance price than our request, despite there being another AZ which we could have launched in. Another issue is that the Hadoop reducers generally spend more time running than mappers. This makes them less ideal to be run on Spot Instance TaskTrackers, as you're more likely to interrupt them. One solution would be to configure the spot nodes to just run mappers, the other to configure your jobs so that they are spread over many reduce tasks.

Overall though, minor gripes aside, these tweaks we've made have helped us streamline the use of our Hadoop cluster, to scale in a more cost effective manner, and to minimise unnecessary noise between nodes.