Database Benchmarking

Tested Cassandra And PostgreSQL

Performance notes.

6/10/19 With batch processing alone, I managed to get POSTGRES to seed, with real time data generation, JSONB values, in 37 minutes.
Real time data generation takes 15 minutes. I could refactor my datagen script to lessen it, but at the expense of randomness of data, and it matters very little.
The batch size for this process was 1000 rows.

6/11/19
Cassandra seeds, with real time data generation. However, I'm getting errors regarding excessive concurrency from my node instance. I just adjusted batch size to 13 from 14. Lets see if it can finish seeding. Over the last day I've had to reinstall cassandra as well, due to connection problems.

UPDATE: 6/11/19 6:18PM.
Batching is a bad idea to increase efficiency in Cassandra. It works in other databases, but not in Cassandra. Made cassandra code asynchronous. Now cassandra seeds in 33 minutes

UPDATE: 6/11/19 11:00PM
Batching and asynchronous concurrency implemented in postgresql seeding script. Settings are Batchsize: 1000 Concurrency limit: 120. Still cannot get below 36 minutes, including real time data generation.

UPDATE 6/12/19 Cassandra

- Getting **Server timeout during write query at consistency LOCAL_ONE** when attempting cassandra seeding. Decreased concurrency limit to 20, and increased write timeout limit to 5 seconds from 2 seconds.
- Dropped concurrency to 10. Still getting error with 10 million dataset.
- Dropped concurrency to 5. Still getting error.
- Increased write timeout to 15 seconds from 5 seconds. Still getting error.
- Reinstalled cassandra. Still getting error. Now using default settings.

UPDATE 612/19 6:00pm Adjusted postgres shared_buffers from 128mb to 1024mb. Projected seed time is 2600seconds.

- Adjusted wal_buffer from 16 to 256MB. Projected seed time is 2458 seconds.
- Adjusted shared_buffers to 4GB, 25% of available system memory. Takes over 40 minutes now.

UPDATE 1/12/19 11:00pm- 1/13/12:00PM..

- -Fixed cassandra issue. Needed to kill background processes. Cassandra config survived system restart this time. Added rejected insert handling. Need to minimize rejections, or eliminate. Cassandra seeds in 44 minutes now. Concurrency size was 50 when I got that result.
- Reduced Concurrency size to 30. Reduced failed insertions to 11 when seeding 1 million.
- Changed all TEXT entries in schema to ASCII. Still takes 3 minutes for 1 million. 31 failed.
- Changed replication factor from 2 to 1. 56 failed and a little over 3 minutes taken.
- Changed Write timeout to 5 seconds in cassandra.yaml.Still getting the same Server timeout error. 24 failed.
- Changed Max heap and new heap size to 6gb and 800 mb respectively in cassandra-env.sh. 55 failed and it takes 202 seconds.
- Trying concurrency of 2. Lets see what happens. Only 5 failed, but it took 248 seconds for 1 million records.
- Specified idempotence level as true. 3 failures, and it took 4 minutes.
- Added a retry policy. Doesn't seem to be activating. 2 failures, and took 252 seconds.
- Set prepare to be true. No meaningful changes.
- Added a recursive call to seed() again, whenever exception is caught by insert query. Retries the insert. 10 fails. Got a new type of fail. **Represents a client side error that is raised when the client did not hear back from the server within socketOptions.readTimeout.**
- The datastax library retry method is calling upon insert fail, and so is my recursive seed() call. Removing my recursive seed() call.
- 1 million entries seeded with 5 retries, and in 240 seconds.
- Changed concurrency back to 30. That's strange. I seem to have no fails, and finished in 188 seconds.
- Ran it again. Fails are back. 267 rejections, and it ran in 209 seconds.
- Apparently, every time I drop the database at the end of my script, it snapshots the database and saves it to hard drive. I manually deleted 120 gb of snapshots. Performance is a little bit better. Purged and reinstalled.
- Changed commitlog_sync to batch from periodic in yaml. Changed autosnapshot to false in yaml as well.
- Changed mem_table_heap_space_in_mb to 5120. Left Offspace at 2048. Changed write timeout from 2000miliseconds to 5000.

UPDATE 1/13/19 1:30

- Postgress can perform a select based off of item id in 1.7ms. That's rather fast.

The Following is a list of all changes that need to be made to cassandra upon install for the purpose of this project's seeding, and assumes that you are testing your hardware's capacity with concurrency.

- Change autosnapshot to false in yaml.
- Set `memtable_allocation_type: offheap_objects` This keeps large amounts of data out of the JVM heap.
- Avoid adjusting the heap size in cassandra-env.sh. Any more than 8 will cause Garbage collection to slow down, less will cause a bottleneck. Default should be good.
- Set `memtable_heap_space_in_mb` to rather low. Again, this assumes you are trying to incorporate concurrency. 32 -512 might be ideal. Experiment. Smaller space can create smaller, more frequent flushes, which would be good for concurrency.
- `Compaction_throughput_mb_per_sec` compaction can create a lot of garbage. Investigate and experiment with this.
- Adjust concurrency to quite low, and work up slowly.
- Commit log and sstables should be on different drives. Commit log can go on a Disk drive, while the sstables should go on an ssd.
- Sparingly adjust write_request_timeout_in_ms to be higher, if necessary. This is useful for testing purposes, but a bad practice in general.

Reinstalling and applying some of these changes. Starting with 100,000 entries. No fails.

- Switching to 1 million entries for larger dataset. Current concurrency is set to 5. 23 fails and time was 218 seconds.
- Set memtable_allocation type. Tried again. 18 fails and runtime of 223 seconds.
- Switched memtable_heap_space_in_mb to 512 from 2048. Got 12 fails and it took 213 seconds.
- Switched memtable_offheap_space_in_mb to 512. Got 3 fails and it took 207 seconds. I like what I see.
- Switched metable_offheap_space and heap_space to 256 each. 7 failed and 213 seconds.
- Ran tests again to test variance. 41 failed and 220 seconds, 7 failed and 215 seconds.
- Ran nodetool repair, cleanup, and flush. Ran again, got 233 seconds and 24 fails.
- Disabled garbage compaction. Lowered both memtable sizes to 64. 301 seconds and 94 fails. They seem to be more frequent now.
- Increased the write_request_timeout_in_ms from 2000 to 3000. 250 seconds and 20 fails.
- Raised timeout to 4000. 31 fails and 290 seconds. Tested again for variance and I got 21 fails and 276 seconds.
- Lowered both memtables to 32mb. 290 seconds and 36 fails.

6/14/19 2:19PM Cassandra.

- I am now using datastax executeConcurrent function instead of the async library's function to time the queries. The 'async' library function I now use is timesSeries to time each batch of concurrent queries.

- I switched the memtable and off heap memtable back to their defaults. I adjusted write request timeout to 15 seconds, and I disabled readTimeout in my client options object. I was able to seed 4 million without any issue in 16.6 minutes.
- I reconciled the fact that my memtable flushes are going to be frequent and big, and switched those settings back to 512 each, which I determined was the sweet spot. I adjusted distance.local client pools from 2 to 4.
- I was able to seed, with only two fails, which were retried, in 2774seconds. Cassandra is not really meant to run in a single node setup.

Update 6/15 Cassandra

- Enabled trickle_fsync in yaml. Raised concurrency to 40. Lowered write timeouts to 7 seconds from 15 seconds. Was able to seed 5 million in 971 seconds. No timeouts.
- Lowered write timeout limit to 5 seconds from 7, install libjemalloc2. 4 failed, and 1005 seconds to seed 5 million. Re enabled autocompaction. Its a bad idea to have it off. Running it again.
- 962 seconds with 2 failures for 5 million.
- Reset local and remote coreconnectionsperhost settings to 4. Running again.
- 900 seconds and 0 failed. Setting to 5. Running again 918 seconds and 0 failed. Setting both to 6, and running again. 969seconds and 3 failed. Setting both back to 5 and rerunning to account for variance. 950 and 7 failed.
- I should increase compaction_throughput_mb_per_sec. Increased from 16 to 32. Running again. 923 seconds, and 0 failed. Pooling is still at 5, and concurrency is still at 40. Very nice.
- Set memtable_flush_writers to 2. Default is 1. This is will let memtable_cleanup_threshold to default to the new value of 1 / (memtable_flush_writers+1), so basically, from .5 to .33.Running again. 908 seconds to seed 5 million, and 0 failed.
- 2100 seconds to seed 10 million, and 4 failed. Compare to yesterday's 2774 and 2, and consider that yesterday the write timeout was set to 15 second, and today, it was set to 5 seconds.

So to have deduced for the sake of our project, it seems that, for Cassandra:

- Cpu/Memory/Memory timings dictate concurrency and concurrency batch sizes. Pool size should increase with concurrency as well.
- Concurrency determines memflush timings. More concurrency need faster, more frequent memflushes, or else timeouts will occur.
- More memflushes require faster compaction rate.
- The easy way to deal with timeouts is simply increase the timeout length, but that does not address the issue and is not a good idea.
- I am curious as to the stochastic aspect of database design? Should I minimize the number of timeouts to as low as possible, or optimize for time complexity at the expense of more timeouts and simply handle the timeouts well?

I am proceeding to the next phase of this project with PostGreSQL because the query times are simply much faster.

So, we are deploying to AWS t2 microinstances, and they only have a gigabyte of ram. I was able to seed postgres in 33 minutes on my workstation. It's seeding 23gb of data, compressed to 18gb. It's going to take alot long on a microinstance.

- Getting out of memory errors when seeding at current script settings of batch size 1000 and concurrency of 80.
- Reduced concurrency to 50 and batch size to 150. Its seeding now without out of memory errors.
- 150 doesn't work. Getting the same issue. Switched to batch size of 100. This is going to take forever.
- I have to mess with the wal_buffer.
- Disabled full page writes.
- Tried some other stuff, and none of that works, and I've come to realize this effort is utterly futile. I'm basically juryrigging this database into working in a way it was never intended by removing every built in safety. I'm just going to let it run all night and concede that I am writing 18 gigabytes of data and that it can only be optimized so much.
- Commented out two memory adjustments and now I am going with postgres's default autodetected settings. These settings were shared_buffers and wal_buffers.
- It seeds in 3 hours. I consider that a victory.

Stress Testing

- I finally chose postgresql for my project because query times were substantially lower, at 1.5ms versus Cassandra's 9ms average.
- I chose k6 as my testing tool, and New Relic as my metrics tool.
- Utilizing k6's http.get() I was able to locally query my database. My script queries a random item from the database.

Update:6/18/19

- I got some metrics, and then cleaned up some console logs from my code. I disabled k6 from running any tests on queries. No reason for New Relic and k6 to check the same thing, and I think New Relic's metrics are more experimentally derived, and therefore, more useful. I disabled an IF block in my database script. It wasn't doing anything. The original author of this code used it to add additional functionality to the /items route.
- I was able to get 41k requests through at a 1.5 ms latency, and a 0 error rate. K6 ran 920 virtual users. I can't push k6 harder than that without getting errors indicating that too many files were opened. It seems that whenever a socket is opened, Linux creates a file for it to mount to. I knew that hardware works this way in linux. Did not know that network sockets work this way.

- Raised the limit for open files. I was able to raise the virtual users to 1000. I can't seem to get any meaningful metrics from New Relic regarding the change from a 10 minute test. K6 is reporting that instead of 1.4gb received, 48mb sent, and an average req duration of 57ms, I am now getting 1.5gb received, 53mb sent, and an average req duration of 58.6ms. I should really run a longer test.
- Running a 100 minute test at 1000vu. K6 uses 5.3 gb of my systems RAM for the test. It's actually going up as I am writing this. Now it's at 5.9gb. I wonder how high it will go.
- Update. It's been 4 minutes. It's at 6.4gb. Ram usage seems to not be rising any more. I think for some reason, it might just not be updating in ksysguard. It could still be rising, and might just crash my system.
- Update. 6 minutes from start. Ksysguard is updating again, but at slower intervals. I wonder what's up. Ram usage is hovering around 6.4
- Update. 35 minutes in. Ram usage is at 6.55 gb. I have 16gb of ram in my system and it's clocked pretty fast. Ram usage is rising, but at a slow rate. I wonder if this process will push the memory addressing to the limit and crash the system. There's probably some logging that I could disable in k6. I have not enabled the postgresql integration in New Relic yet.
- So I killed the test in k6. It ran for 76 minutes, and that's long enough. I am getting some more accurate metrics now, I think. Average latency is 55ms. 55700rpm. 0% error rate.
- Activated the postgresql New Relic integration. Now I have stats on my databases performance.

Update 6/21/2019

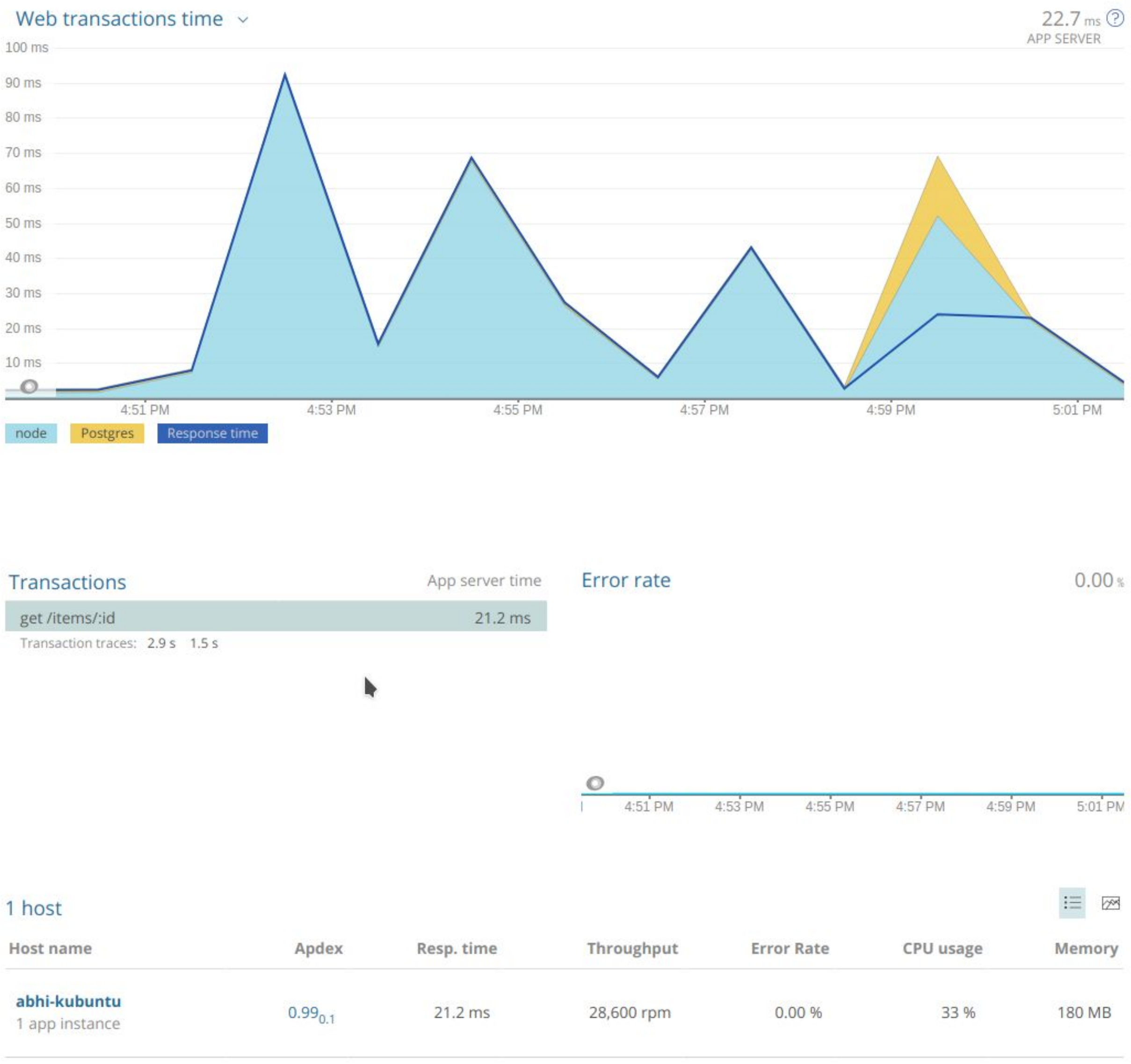
- I'm waiting for postgres to seed on AWS.

Update 6/26/2019

- Database is seeded, and service works on AWS.
- Proxy works as well.

The following are screenshots of my stress tests both on my workstation and on AWS. The monitoring tool I used was New Relic and the stress testing tool I used was loader.io. The free tier only allowed me to test for a few minutes. Because of the nature of the free tier, I believe that my statistics underreport, and further optimization would allow for lower latency and higher RPS capacity.

Service running locally on my workstation.



Transactions — App server time

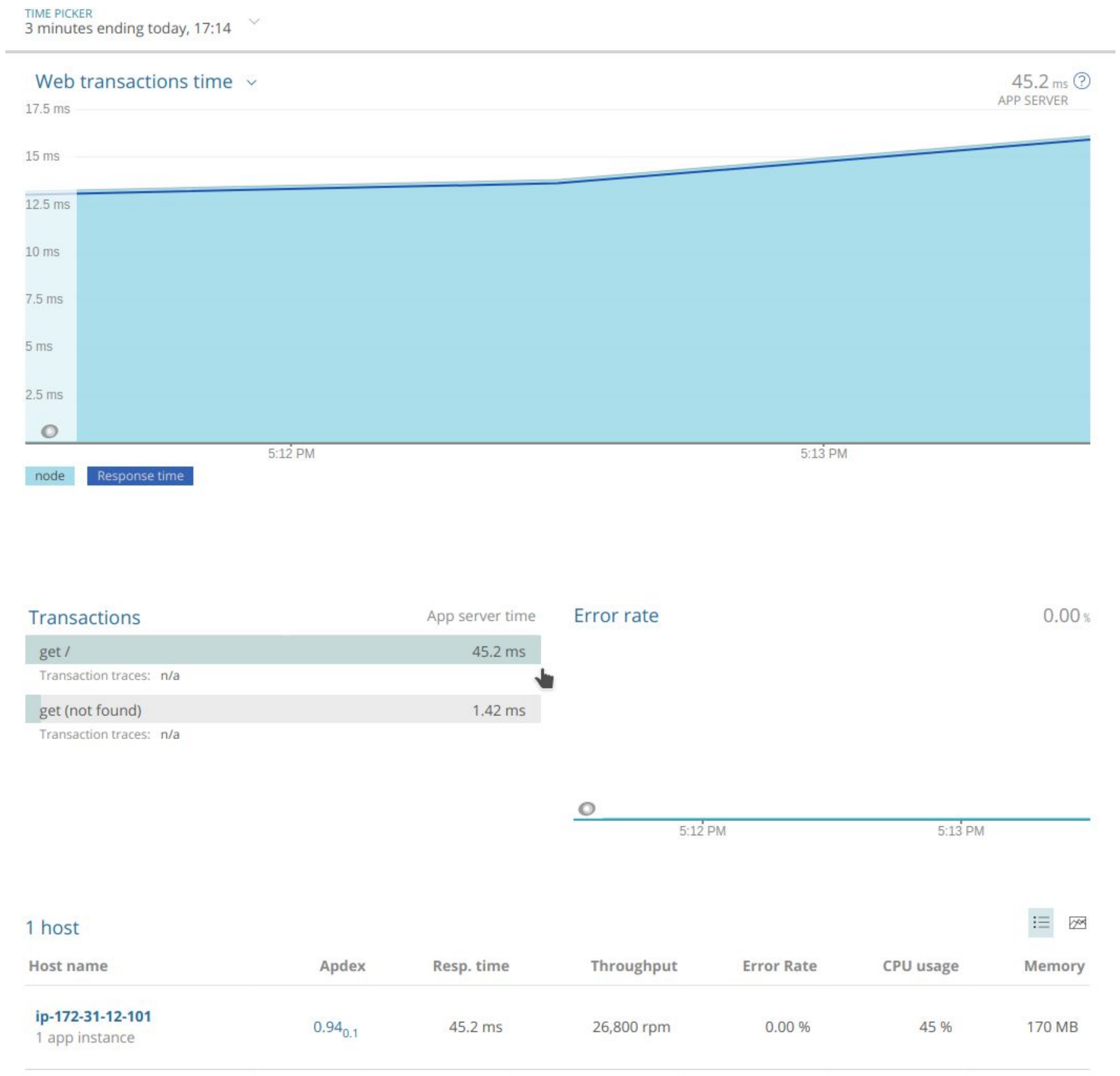
get /items/:id 21.2 ms

Transaction traces: 2.9 s 1.5 s

1 host

Host name	Apdex	Resp. time	Throughput	Error Rate	CPU usage	Memory
abhi-kubuntu 1 app instance	$0.99_{0.1}$	21.2 ms	28,600 rpm	0.00 %	33 %	180 MB

Service running off of a t2 microinstance.



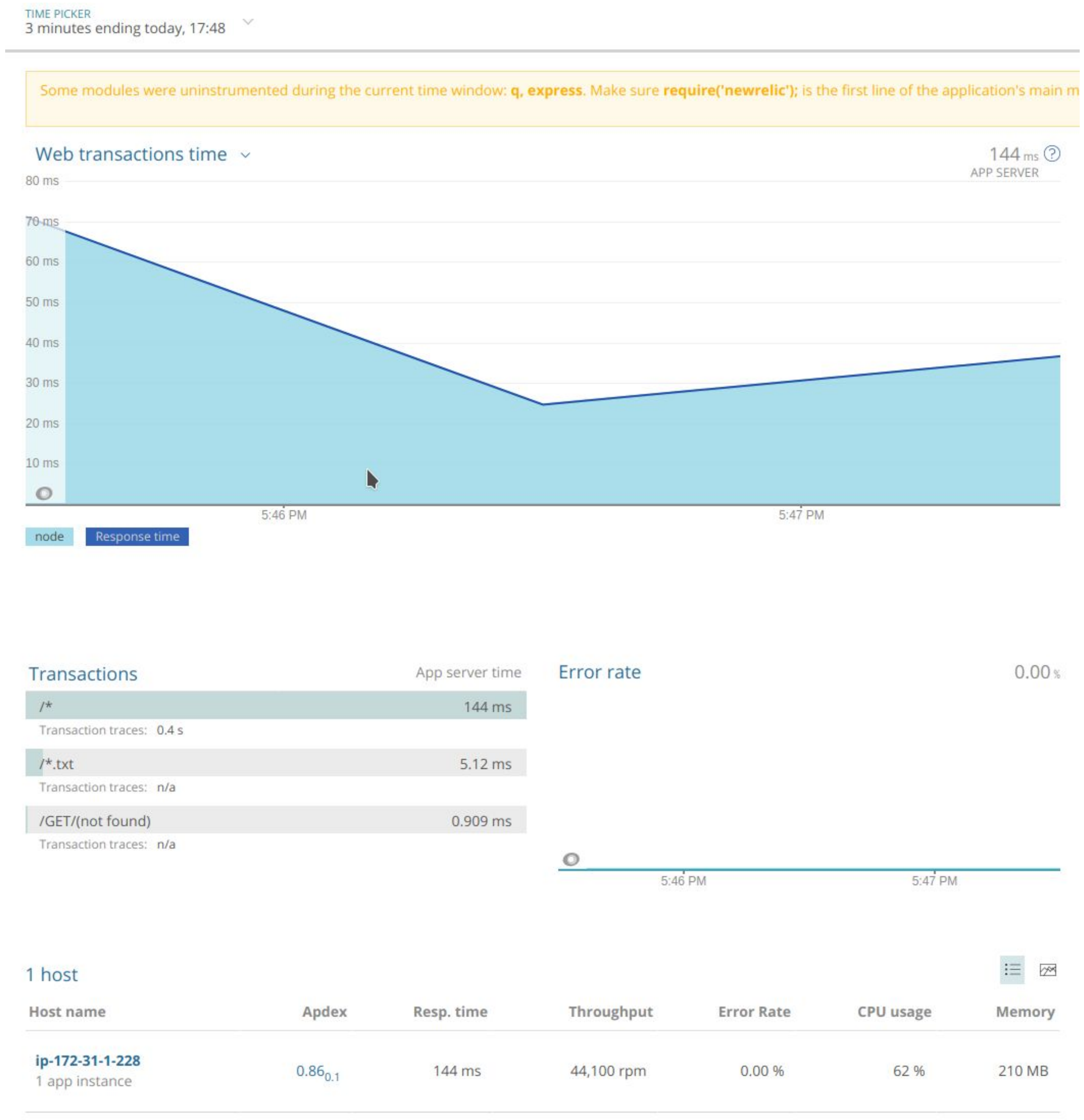
Transactions — App server time

get /	45.2 ms
Transaction traces: n/a	
get (not found)	1.42 ms
Transaction traces: n/a	

1 host

Host name	Apdex	Resp. time	Throughput	Error Rate	CPU usage	Memory
ip-172-31-12-101 1 app instance	$0.94_{0.1}$	45.2 ms	26,800 rpm	0.00 %	45 %	170 MB

Proxy running off of a t2 microinstance.


TIME PICKER
3 minutes ending today, 17:48

Some modules were uninstrumented during the current time window: **q, express**. Make sure **require('newrelic');** is the first line of the application's main m

Transactions — App server time

/*	144 ms
Transaction traces: 0.4 s	
/*.txt	5.12 ms
Transaction traces: n/a	
/GET/(not found)	0.909 ms
Transaction traces: n/a	

1 host

Host name	Apdex	Resp. time	Throughput	Error Rate	CPU usage	Memory
ip-172-31-1-228 1 app instance	$0.86_{0.1}$	144 ms	44,100 rpm	0.00 %	62 %	210 MB

Resuming development.

Update: 7/20/2019

I have, since last update, implemented server side rending. The app loaded much faster, but I need further investigation into the proper method of implementing it and the resulting stress tests indicated that my tiny t2.microinstance was buckling under the cpu load. For this reason, I didn't bother taking screenshots of the response times but rest assured, my server was under substantially more load. I don't think microinstances were meant for that sort of thing.

I reverted my server back to client side rendering, so I could continue to investigate methods of scaling.

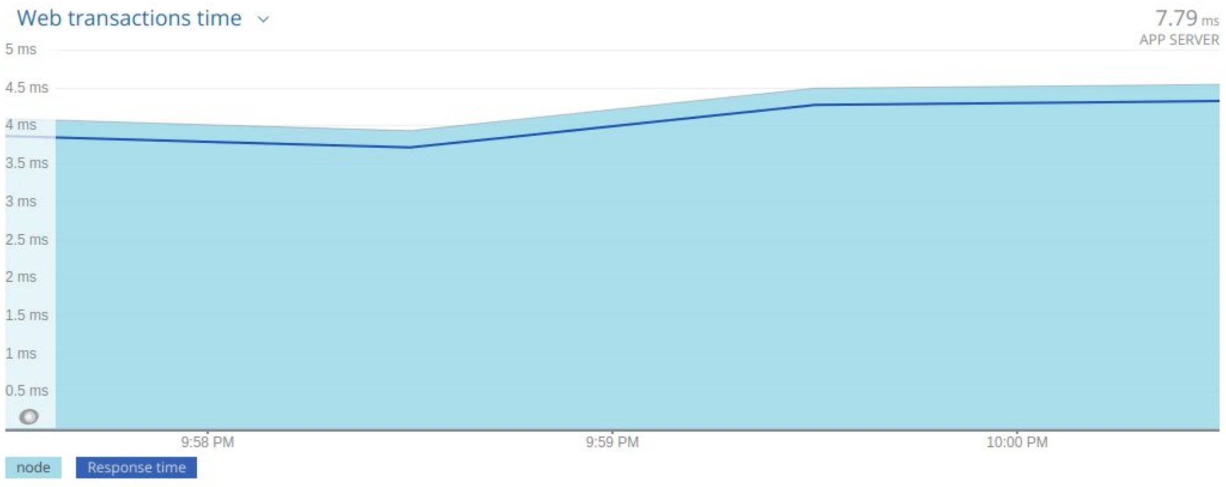
I successfully wrote the script that allowed my app to be run as a service and restart whenever the server reboots.

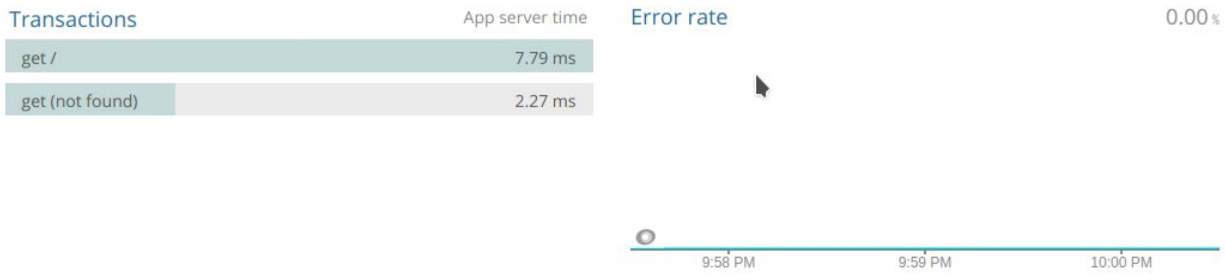
Now I'm going to implement an AWS load balancer.   My goal is to always remain at below a 100ms response time. I'm eliminating the proxy for the remainder of these experiments, and I will be load balancing the service alone.

But first, I'm going to increase the work_mem for my postgres database from 4 to 8mb.
It did nothing.
I'm going to increase from 8 to 16mb.
Did nothing

I think the service is the bottleneck right now. Lets proceed to load balancing.

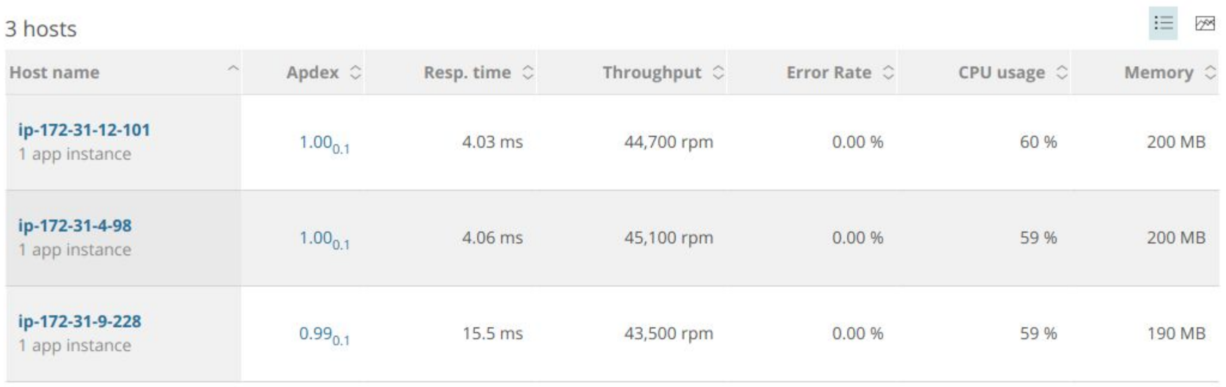
Ok. So I implemented a load balancer with 3 identical instances of my service, and i am able to get over 2000 rps with a latency of below 5ms on two instances and 15ms on one of the instances.





Transactions

	App server time
get /	7.79 ms
get (not found)	2.27 ms


3 hosts

Host name	Apdex	Resp. time	Throughput	Error Rate	CPU usage	Memory
ip-172-31-12-101 1 app instance	$1.00_{0.1}$	4.03 ms	44,700 rpm	0.00 %	60 %	200 MB
ip-172-31-4-98 1 app instance	$1.00_{0.1}$	4.06 ms	45,100 rpm	0.00 %	59 %	200 MB
ip-172-31-9-228 1 app instance	$0.99_{0.1}$	15.5 ms	43,500 rpm	0.00 %	59 %	190 MB

This is awesome, but there's a bottleneck somewhere, and that's what's causing the discrepancy between the two instances and the one. In addition, when I raise the concurrent clients per second on www.loader.io from 2500 to 2700, the response time rises to above 300ms. That's another bottleneck. CPU is being fully utilized on each of the services so that could be the bottleneck at this point. I don't know how well the database is being utilized.

Horizontally scaling my database is beyond the scope of this project, and I think I should move on. Also, the more I think about it, the more I feel like the dataset I'm using right now is not good for the educational purpose of database optimization, because it consists of a number index and a massive json object in each row. Also, there's no read/write load mix. Populating the database was a write heavy load, and reading from the database during the load tests is a load consisting of literally only reads. So those are not interesting engineering considerations. I think I need to move on from any more postgresql investigations.

I'm going to implement an autoscaling group, and then move on.

Autoscaling implemented. I think we're done here. I'm nuking the instances.