Possible performance optimisations of Airflow CI build by using self-hosted High-Mem Spot instances

07.02.2020

This document describes improvements that can be achieved by introducing High-Memory spot instances for the Airflow CI system.

This analysis shows that it is possible to achieve 8x speedup in the elapsed time of executing most of Apache Airflow tests, at the same time lowering the cost of those jobs by half by employing high-memory spot instances EC2 (with 64GB RAM) and running tests in parallel on those instances - when paired with auto-scaling

Context

For Apache projects, starting December 2020 we are experiencing a high strain of GitHub Actions jobs. All Apache projects are sharing 180 jobs and as more projects are using GitHub Actions the job queue becomes a serious bottleneck. For several weeks we are experiencing 4-5 hours delays between the jobs put in the queue and starting to execute. This is highly unacceptable and while several attempts were made to discuss it with Apache Infrastructure and Github Developer Relation team, it seems that there is no solution that could be applied soon. Most of it is documented here.

The conclusion is that the only way to tackle this issue is to employ Self-Hosted Runners for each project (providing that there are funds that the project can spend on those). Apache Airflow has funds donated by the Amazon via (AWS promotional credits for Open-Source Projects) , from their open-source programme (4000 USD/year as well as matching donation from Astronomer), The problem is, that the Self-hosted running is strongly discouraged for Public Projects by GitHub due to security concerns.

Current state

Ash Berlin Taylor and Astronomer had worked hard over the last few weeks to implement a custom security layer on top of the existing runners. This uses https://github.com/actions/runner/pull/783 to not have un-trusted users run code (security is based on the actors of the commit - commiter's PRs and direct pushes are allowed to run builds on self-hosted runners) on our hosts, and then a combination of a Github Application, AWS Lambda and an AWS Auto-Scaling Group.

The infrastructure is in-place but we are running some last optimisations and tests to see how to use the infrastructure in an optimal way.

Optimisation Hypothesis

Currently we are using small memory machines with Local SSDs to run the tests however due to the nature of CI (ephemeral, clean state needed for every job, no need to store the results locally as cache) and based of previous experiences of Jarek Potiuk with similar CI systems a hypothesis was made that using far bigger machines with lots of memory using "tmpfs" in-memory system might bring significant benefits and performance improvements. The hypothesis was that when docker builds, software builds and running tests run entirely in-memory without touching the local SSD, it will significantly increase test execution time without impacting the cost (actually decreasing the cost as bigger machines will be far cheaper when using "spot" instances from AWS and they will be needed for far shorter time.

This document explores this hypothesis.

Airflow CI build anatomy

The CI build uses the following steps:

1) Common steps to build Docker images which are stored as Cache in Github Packages/Github Container registry (PrOD/CI builds - separate job per python version) - **6 jobs**
2) Static check jobs that perform static checks for the code of Airflow (**one job**)
3) Documentation building job that verifies and builds documentation (**one job**)
4) Test Jobs - pytests for various matrix combinations of Python version, Database version - **10 jobs**
5) Kubernetes tests jobs that run for multiple versions of Kubernetes/Python versions - **9 jobs**
6) Additional less significant jobs

Typical execution times for the jobs:

- 1) Docker image building. Depends on stage of the image vs. cached image
    - ~ 240s : 4:00 minutes for only source code change

    - ~ 600s : 10 minutes when dependencies change
    - ~ 960s : 16 minutes when image is rebuild from scratch (when Dockerfile changes significantly
- 2) Static checks: ~ 15 mins for full set of tests for all files
- 3) Docs: ~ 28 minutes for full documentation build
- 4) PyTests: ~ 1h12 minutes for all types of tests (Always, Core, Other, API, CLI, Providers, WWW, Integration, Heisentest)
- 5) Kubernetes tests: ~ 18 minutes for full set of tests

Performance tests

The set of performance tests were executed on 8 CPU machine with 32 GB RAM with in-memory (“tmpfs”) filesystem. Note that for most of the test only 1 CPU was actually used which opens up the possibility for parallelisation (as mentioned in the tests).

Disk-based:

All tests executed on a machine with Local SSD NVM storage (results were far worse on machines with General Purpose SSD storage).

Docker build:

```
Build image from cache: pull + rebuild all scratch: (0:16:30)
Build image from cache: pull + deps: (0:10:10)
Build image from cache: pull + only airflow sources changed: (0:04:10)
```

Docs build

```
Docs build:  (0:28:50)
```

Tests:

```
Always: 49.42s (0:00:49)
```

```
Core: 390.05s (0:06:30)
Other: 408.38s (0:06:48)
API: 773.32s (0:12:53)
CLI: 309.71s (0:05:09)
Providers: 407.60s (0:06:47)
WWW: 778.65s (0:12:58)
Integration: 269.43s (0:04:29)
Heisentests: 244.54s (0:04:04)
```

Total tests: **1:16:00**

```
Docs build:  (0:28:50)
```

Kubernetes tests

```
Prepare Kind Cluster and deploy Airflow: (0:07:00)
Run Kubernetes tests: (0:11:00)
```

Static checks:

Total time: **14:53**

Memory - based:

Docker build

```
Build image from cache: pull + rebuild all scratch: (0:07:32)
Build image from cache: pull + deps: (0:02:20)
Build image from cache: pull + only airflow sources changed: (0:01:10)
```

Docs build

```
Docs build:  (0:27:55)
```

Also an attempt was made to run the build with -j 8 switch (parallel builds)- but it seems that the way the build is done now, the builds does not use the parallel build anyway. There are some improvements that can be made to only build selected parts of the documentation and maybe enable the parallelism of build supported by sphinx.

Tests:

```
Always: 22.25s (0:22:25)
Core:  175.57s (0:02:55)
Other: 272.05s (0:04:32)
API: 495.20s (0:08:15)
CLI: 157.02s (0:02:37)
Providers: 302.18s (0:05:02)
WWW: 526.76s (0:08:46)
Integration: 139.50s (0:02:19)
Heisentests: 137.19s (0:02:17)
```

Total tests: **0:40:45**

Static checks:

Total time: **6:00**

Kubernetes tests

Not completed due to lack of time/ memory - but preliminary comparison had shown that 2x improvements are possible

Cost comparison:

Typical cost of comparable machines for AWS:

Minimum disk based instance (needs Local SSD)

- c5d.xlarge - 2 vCPUs, 8GB RAM, Local SSD.
  140 USD/Month (base price) - 80% spot ratio = 28 USD/Month

Minimum memory based instance (needs lots of RAM)

- r5.2xlarge 8 VCPUs. 64 GB RAM, no SSD needed
  235 USD/Month (base price) - 80% spot rato = 47 USD month

The disk based instance is roughly 1.5 x cheaper than the memory one. This means that we can achieve cost savings if the bigger machines are used for ~2x shorter time for sure.

Summary

Docker builds

Docker builds are 3x-4x times faster for in-memory machines than the disk ones. We can save from 3 to 10 minutes of time for each of the 6 Docker image builds (depending on the state of the image). Also this 3x -10x minutes of elapsed time for each build.

Tests:

It looks like (as expected), due to the databases being used for many tests, the tests are quite seriously I/O Bound. By switching to in-memory filesystem we can speed up test execution almost 2x using a single CPU.

Overall speedup for test in-memory vs. disk based: **40:45** vs **1:16:00**. Which equals to 48% improvement (almost 2x faster)

Also running it on bigger machines with more RAM and CPUS, it opens up the possibility of parallelising test execution (because the tests are using 1 CPU and we have 8 CPUs). Those are test groups we can run in parallel:

```
* API (9:00)
* WWW (9:00)
* Core + Other (7:30)
* CLI+Providers (7:40)
* Integration + Heisentests( 4:40)
```

This means that all tests would finish under 9 minutes.

Note that we cannot easily run tests in parallel on the Disk- based instances because they lack the CPUs (we have 2) and memory (we have 16GB) needed. Running tests in parallel on disk-based instances also increases the load on the SSD connected and since the tests have already shown that they are quite significantly I/O- Bound, there is high risk of contentions, race conditions increase, and not-linear scalability

Seems that we can easily achieve **8x speedup** and elapsed time < **10 minutes** if we parallelise the tests as described above. If we include that we have auto-scaling instances that are only used when needed (including some overhead) this means that we can get **8x speedup of CI builds with 2x lower price to run the builds if we utilise properly the memory-heavy instances (!)**.

This means that we can get not only far faster but also far cheaper builds when we switch to the memory-heavy instances.

Docs:

It looks like the docs build are not I/O bound. Also they cannot be easily improved by parallelism. This is the weakest point of the CI build. If we manage to optimise the tests to 9 minutes, doc build running 28 minutes and utilising only a single CPU will be a serious bottleneck and it will impact significantly the elapsed time of the build. Some rework to increase the amount of parallelism or limiting the build scope is needed.

Kubernetes

Not executed due to memory limits (tests were run with lower amount of tmpfs - 17GB that would be available on the biggest machines), but some preliminary runs have shown at least 2.5 speedup for those tests as well.