AIP-15 Support Multiple-Schedulers for HA & Better Scheduling Performance

Status

State	Completed
Discussion Thread	Multiple Schedulers - "scheduler_lock" A Naive Multi-Scheduler Architecture Experiment of Airflow [PROPOSAL][AIP-15 Support Multiple-Schedulers for HA & Better Scheduling Performance]
Github issue	https://github.com/apache/airflow/issues/9630
Created	$action.dateFormatter.formatGivenString("yyyy-MM-dd", $content.getCreationDate())
In Release	2.0.0

Background

Original document (archived now) is in AIP-15 Scalable Scheduler [ARCHIVED]

The purpose of this document is to serve as a functional and high-level technical design for the Airflow Scheduler high-availability effort.

Airflow has excellent support for task execution ranging from the basic Local Executor for running tasks on the local machine, to Celery-based distributed execution on a dedicated set of nodes, to Kubernetes-based distributed execution on an as-needed, dynamically scalable set of nodes. The Airflow scheduler reads DAGs, analyses task dependencies, schedules and tracks task execution across these distributed nodes, handles task failures with retries as appropriate, and reports errors.

As a result, task execution is now significantly scalable within Airflow with the improvement in executors. However, in the evolution, the Airflow Scheduler itself has become a bottleneck for future scalability and worryingly the ‘single point of failure’ which is an anti-pattern in today's computing architectures.

It is clear that the Airflow Scheduler does far more than ‘just scheduling’ and is analogous to a hypervisor, but the naming discussion is a topic for another time.

Motivation

By default, users launch one scheduler instance for Airflow. This brings up a few concerns, including

- **High Availability**: what if the single scheduler is down.
- **Scheduling Performance**: the scheduling latency for each DAG may be long if there are many DAGs.

In today’s world, the single point of failure does come up as a blocking issue in some users' adoption of Airflow. This is a well-known limitation with Airflow today and some SIs have written blogs about how enterprises could address this in their environments. For example, here is a link from Clairvoyant about making Airflow highly available.

It would be ideal for Airflow to support multiple schedulers, to address these concerns.

There was a "hacky" method to start multiple schedulers and let each handle a specific set of DAGs. It does improve scheduling performance, but doesn't address HA concern, and was never "officially" supported by Airflow, and required manual book-keeping.

Another issue that has started coming up with AI-centric users is the need for massively parallel task execution - a recent example of this required thousands of DAG runs to be initiated within a short time frame and run in parallel as part of an ML pipeline, and Airflow didn't really cope and the performance (i.e. time to schedule tasks) wasn't acceptable.

There are also still occasional reports of the scheduler "stalling" and no longer scheduling tasks.

Goals

Based on the above, the goal of this project is two-fold:

1. Make the Airflow Scheduler highly available to address reliability concerns, and
2. Make the Airflow Scheduler scalable beyond the current single-node, multi-thread/process model to serve the increased scalability demands of users.

Approach

The high-level approach outlined below is intended to be contained within Airflow and infrastructure-neutral, since it needs to work in the vast majority of enterprise and cloud deployments without *needing* infrastructure components such as Kubernetes.

The high level design approach is to leverage a 'active / active' model using the existing shared database for locking and a not require any direct communication between schedulers. Specifically, this entails the following enhancements to Airflow:

- Multiple schedulers can now be run within Airflow. For a high availability situation (any production deployment), this should be at least two, three is better.
- The running schedulers will all use the same shared relational database (the 'metadata database') and regularly heartbeat their own availability so that external monitoring systems can perform health checks easily. This 'external' data availability will be performed by making 'operational data' available through the existing statsd interface.
- Each scheduler will be fully 'active'. This includes 'non-scheduling' work as well such as task execution monitoring, task error handling, sending task emails, and clean-up.
- The configuration for all the schedulers will be identical. Differing configurations for different scheduler instances will NOT be supported. This is actually incomplete at this time, since it is intended to cover 'application configuration', not necessarily 'infrastructure configuration' detailing where these scheduler instances could be run. There are no current plans to *ensure* each scheduler is configured the same
- This approach relies on DAG serialization becoming an integral part of the scheduler. In practice, this means the following:
    - As part of DAG file parsing, the DAG file will be parsed and serialized sufficiently that the serialized version can be used for scheduling. This serialized version will be stored in the database.
    - To reduce lock contention in this process, each scheduler will sort the DAGs to be processed in a random order before starting the parsing steps.
- The scheduler will use the database as the 'shared queue' for operations to be performed and lock at the appropriate object.
    - The scheduler 'shared queue' will probably be at the 'DAG run' level. This needs to be validated during the detailed design, since this may be 'necessary, but not sufficient', since certain operations such as 'managing task SLAs' may require 'task-level locking' though it seems possible to refactor those into the 'DAG' level locking model.
    - In practice, this means that one of the schedulers picks 'the next DAG run' in the list and 'locks it' in the database to start working on it. While it is working on it, it periodically updates the lock as a heartbeat that it is alive and working on it.
    - The current thinking is to use the database locking by `SELECT FOR UPDATE` on this row as the locking mechanism as opposed to using a separate field in the table for the lock. If a scheduler dies while working on a DAG, the lock would be expired by the database. This needs to be validated and optimized since there have been reports of slowdowns observed while trialing this approach.
- The workers will be enhanced to implement a fast-follow mechanism to reduce the activity of the scheduler. Specifically, this includes:
    - Workers will look for the next task in the same DAG to be 'ready' at the point of task completion and if it is ready, put the next task into the QUEUED state.
    - This would reduce the scheduler activity for follow-on task execution, but the communication between the worker and the executor about the identification of the 'next task ready' and 'task being worked on' needs to be clarified.
    - A config flag which triggers this option may be of value, since this does change the existing behavior and has a possibility of 'DAG starvation'.

More sophisticated approaches can be layered onto this approach for specific cloud / infrastructure capabilities thereby increasing both scalability and the handling of infrastructure failures.

The probability of schedulers competing on the same DAG is easy to calculate since it's a typical Birthday Problem, and it is reasonably low if # of DAGs/ # of schedulers is not too low (the probability that there are schedulers competing on the same DAG is **1-m!/((m-n)! * (m^n))** , **m** is the number of DAGs and **n** is the number of schedulers).

Let's say we have 200 DAGs and we start 2 schedulers. At any moment, the probability that there is schedulers competing on the same DAG is only 0.5%. If we run 2 schedulers against 300 DAGs, this probability is only 0.33%.(https://lists.apache.org/thread.html/389287b628786c6144c0b8e6abf74a040890cd9410a5abe6e968eb55@%3Cdev.airflow.apache.org%3E)

There is currently a `scheduler_lock` column on the DagModel table, but it's not used in current implementation of Airflow (as of now, end of Feb 2020). If the SELECT FOR UPDATE approach doesn't work this may be a suitable fallback.

Acceptance Criteria

The following test criteria need to be met:

1. The simplest test of the high availability criteria is to:
    a. Start multiple schedulers with a set of long running DAGs,
    b. Externally kill one of the schedulers during the processing of a DAG,
    c. Validate that all the DAG runs are successfully executed to completion.
2. The same test as the above, but with a modification such that one of the schedulers is run in with a different machine 'localtime' timezone to the others (The default_timezone in airflow.cfg must be the same across all schedulers.). If this is not supportable in the initial release, this should be explicitly noted and ideally prevented from the configuration.
3. The scalability test criteria is more difficult to define upfront, but benchmarks need to be published before and after about the volume of tasks which can be launched and managed by a single scheduler instance at this time and how this can be scaled with a 'multiple active' model.

Known Limitations

The approach detailed above makes Airflow itself a more robust and scalable platform. However, it does not leverage scalability to a general multi-region set of clusters and with a true 'quorum-style' capabilities for distributed processing at scale without a 'shared source of truth' such as an ACID compliant database like PostgreSQL (or MySQL).

It is possible to extend the above approach towards a multi-region, shared-nothing approach from a software perspective, or to potentially leverage a distributed ACID database for the same from an infrastructure perspective, but at this time, those have not been seriously considered, as we believe we can achieve the goals without the extra complexity these technologies introduce.

Deployment Models

1. Airflow scheduler HA deployment in a set of Virtual Machines (no Kubernetes)

blocked URL

2. Airflow scheduler HA deployment in a Kubernetes environment

   blocked URL

Flow diagram

Updated flow diagram of the scheduler with respect to DAG parsing:

blocked URL

To avoid the "contention" between schedulers, we may want to consider random sort list of DAG files before it's passed to scheduler process (https://lists.apache.org/thread.html/e21d028944092b588295112acb9a3e203c4aea7fae50978f288c2af1@%3Cdev.airflow.apache.org%3E)

Another method to avoid schedulers competing with each other is to let scheduler look select the DAG that's not been processed for the longest time that is not locked (https://lists.apache.org/thread.html/6021f5f8324dd7e7790b0b1903e3034d2325e21feba5aef15084eb17@%3Cdev.airflow.apache.org%3E).

Elements considered and rejected or deferred

This section is primarily for completeness and contains elements considered in the technical approach and rejected during the high level design process.

- Staring up and restarting of schedulers when they fail will be left out of Airflow and will be handled by the managing infrastructure. There the following elements are now discarded:
  - This includes a configuration change to specify the number of schedulers to be run.
  - The running schedulers will register themselves within the shared PostgreSQL (or other) database and regularly heartbeat their own availability. When this falls below that number, a new scheduler will be started.
  - Each scheduler will be fully 'active', including the ability to 'launch' a complete new scheduler if the active count in the database is below the configured value.

- Scheduler locking DAG runs using an additional field in the database and dealing with lock expiration as below.
  - If a scheduler dies while working on a DAG, the lock would be expired by one of the schedulers as they look for the next DAG to work on. It is probably best to use the same approach of '2n + 1' for lock expiration similar to task heartbeat. A cross-check for the same is with the scheduler registry (and heartbeat) discussed above. It is arguable that only of these is needed and not both, but that can be nailed down during the detailed design and implementation