Scheduler

The Airflow scheduler monitors all tasks and DAGs, then triggers the task instances once their dependencies are complete. Behind the scenes, the scheduler spins up a subprocess, which monitors and stays in sync with all DAGs in the specified DAG directory. Once per minute, by default, the scheduler collects DAG parsing results and checks whether any active tasks can be triggered.

The Airflow scheduler is designed to run as a persistent service in an Airflow production environment. To kick it off, all you need to do is execute the `airflow scheduler` command. It uses the configuration specified in `airflow.cfg`.

The scheduler uses the configured Executor to run tasks that are ready.

To start a scheduler, simply run the command:

```
airflow scheduler
```

Your DAGs will start executing once the scheduler is running successfully.

Note

The first DAG Run is created based on the minimum `start_date` for the tasks in your DAG. Subsequent DAG Runs are created by the scheduler process, based on your DAG's `schedule_interval`, sequentially.

The scheduler won't trigger your tasks until the period it covers has ended e.g., A job with `schedule_interval` set as `@daily` runs after the day has ended. This technique makes sure that whatever data is required for that period is fully available before the dag is executed. In the UI, it appears as if Airflow is running your tasks a day **late**

Note

If you run a DAG on a `schedule_interval` of one day, the run with `execution_date` `2019-11-21` triggers soon after `2019-11-21T23:59`.

Let's Repeat That, the scheduler runs your job one `schedule_interval` AFTER the start date, at the END of the period.

You should refer to DAG Runs for details on scheduling a DAG.

Triggering DAG with Future Date

If you want to use 'external trigger' to run future-dated execution dates, set `allow_trigger_in_future = True` in `scheduler` section in `airflow.cfg`. This only has effect if your DAG has no `schedule_interval`. If you keep default `allow_trigger_in_future = False` and try 'external trigger' to run future-dated execution dates, the scheduler won't execute it now but the scheduler will execute it in the future once the current date rolls over to the execution date.

Running More Than One Scheduler

Airflow supports running more than one scheduler concurrently – both for performance reasons and for resiliency.

Overview

The HA scheduler is designed to take advantage of the existing metadata database. This was primarily done for operational simplicity: every component already has to speak to this DB, and by not using direct communication or consensus algorithm between schedulers (Raft, Paxos, etc.) nor another consensus tool (Apache Zookeeper, or Consul for instance) we have kept the "operational surface area" to a minimum.

The scheduler now uses the serialized DAG representation to make its scheduling decisions and the rough outline of the scheduling loop is:

- Check for any DAGs needing a new DagRun, and create them
- Examine a batch of DagRuns for schedulable TaskInstances or complete DagRuns
- Select schedulable TaskInstances, and whilst respecting Pool limits and other concurrency limits, enqueue them for execution

This does however place some requirements on the Database.

Database Requirements

The short version is that users of PostgreSQL 9.6+ or MySQL 8+ are all ready to go – you can start running as many copies of the scheduler as you like – there is no further set up or config options needed. If you are using a different database please read on.

To maintain performance and throughput there is one part of the scheduling loop that does a number of calculations in memory (because having to round-trip to the DB for each TaskInstance would be too slow) so we need to ensure that only a single scheduler is in this critical section at once - otherwise limits would not be correctly respected. To achieve this we use database row-level locks (using `SELECT ... FOR UPDATE`).

This critical section is where TaskInstances go from scheduled state and are enqueued to the executor, whilst ensuring the various concurrency and pool limits are respected. The critical section is obtained by asking for a row-level write lock on every row of the Pool table (roughly equivalent to `SELECT * FROM slot_pool FOR UPDATE NOWAIT` but the exact query is slightly different).

The following databases are fully supported and provide an "optimal" experience:

- PostgreSQL 9.6+
- MySQL 8+

Warning

MariaDB does not implement the `SKIP LOCKED` or `NOWAIT` SQL clauses (see MDEV-13115). Without these features running multiple schedulers is not supported and deadlock errors have been reported.

Warning

MySQL 5.x also does not support `SKIP LOCKED` or `NOWAIT`, and additionally is more prone to deciding queries are deadlocked, so running with more than a single scheduler on MySQL 5.x is not supported or recommended.

Note

Microsoft SQLServer has not been tested with HA.

Scheduler Tuneables

The following config settings can be used to control aspects of the Scheduler HA loop.

- max_dagruns_to_create_per_loop

  This changes the number of dags that are locked by each scheduler when creating dag runs. One possible reason for setting this lower is if you have huge dags and are running multiple schedules, you won't want one scheduler to do all the work.
- max_dagruns_per_loop_to_schedule

  How many DagRuns should a scheduler examine (and lock) when scheduling and queuing tasks. Increasing this limit will allow more throughput for smaller DAGs but will likely slow down throughput for larger (>500 tasks for example) DAGs. Setting this too high when using multiple schedulers could also lead to one scheduler taking all the dag runs leaving no work for the others.
- use_row_level_locking

  Should the scheduler issue `SELECT ... FOR UPDATE` in relevant queries. If this is set to False then you should not run more than a single scheduler at once
- pool_metrics_interval

  How often (in seconds) should pool usage stats be sent to statsd (if statsd_on is enabled). This is a *relatively* expensive query to compute this, so this should be set to match the same period as your statsd roll-up period.
- clean_tis_without_dagrun_interval

  How often should each scheduler run a check to "clean up" TaskInstance rows that are found to no longer have a matching DagRun row.

  In normal operation the scheduler won't do this, it is only possible to do this by deleting rows via the UI, or directly in the DB. You can set this lower if this check is not important to you – tasks will be left in what ever state they are until the cleanup happens, at which point they will be set to failed.
- orphaned_tasks_check_interval

  How often (in seconds) should the scheduler check for orphaned tasks or dead SchedulerJobs.

  This setting controls how a dead scheduler will be noticed and the tasks it was "supervising" get picked up by another scheduler. (The tasks will stay running, so there is no harm in not detecting this for a while.)

  When a SchedulerJob is detected as "dead" (as determined by scheduler_health_check_threshold) any running or queued tasks that were launched by the dead process will be "adopted" and monitored by this scheduler instead.

Was this entry helpful?