


# DagRun Refactor (Scheduler 2.0) [ARCHIVED]

## Tracking:

 AIRFLOW-14 - Jira project doesn't exist or you don't have permission to view it.

<https://github.com/airbnb/airflow/compare/master...jlowin:dagrun-refactor>

## Introduction

- [Tracking:](#)
- [Introduction](#)
- [Description of New Workflow](#)
  - [Issues](#)
- [Classes](#)
  - [DagRun](#)
    - [Description](#)
    - [Methods](#)
  - [DAG](#)
    - [Methods](#)
  - [DagRunJob](#)
    - [Description](#)
    - [Methods](#)
  - [SchedulerJob](#)
    - [Description](#)
    - [Methods](#)
  - [BackfillJob](#)
    - [Description](#)
    - [Methods](#)

The primary issue with DAG execution is that there are two completely separate execution avenues in Airflow: `SchedulerJob` and `BackfillJob`. `DagRuns` were recently added to track DAG execution state but they are used inconsistently, adding to the confusion. To put it briefly, there are three competing issues:

1. Scheduler uses `DagRun` and runs any tasks in `RUNNING` `DagRuns`, but also runs any queued tasks regardless of whether they are in an active `DagRun` or not. `Backfill` does not use `DagRuns` and loops over the tasks in its DAG, brute forcing them into the executor until they finish or fail. Therefore they interfere with each other. Scheduler takes a lock on the DAG it's running, but `Backfill` ignores locks. It might be ok if we could guarantee that Scheduler and `Backfill` weren't run simultaneously (even though we can't!), but operators like `SubDagOperator` use `Backfill` internally, meaning they must play nicely together.
2. `DagRuns` were originally intended to track the state of a DAG's execution, just as `TaskInstances` do for tasks. However, the implementation is not complete. `DagRuns` must be unique for each combination of `(DAG, execution_date)`, but nonetheless allow arbitrary attributes like `run_id` and `external_trigger`. If there can only be one `DagRun` per execution date, and that `DagRun` is marked as `external_trigger`, then the scheduler won't be able to create a non-`external_trigger` `DagRun` for the same date! (I'm actually surprised this hasn't bitten anyone yet... probably because `DagRuns` are not heavily used outside the Scheduler.)

What's needed is to formalize the concept of a `DagRun` and make it the canonical way to execute a DAG and track the state of that DAG's execution, just as we do for `TaskInstances`.

This refactor does just that:

- Promote `DagRun` as a first-class Airflow object
  - Remove attributes that didn't make sense, like `external_trigger` -- in a pragmatic sense, we don't really care, and the uniqueness constraint on `(DAG, execution_date)` means this is irrelevant anyway
  - `DagRuns` can manage their own state
  - `DagRuns` can be locked or unlocked by a specific `lock_id`, which guarantees that only one job is trying to execute the `DagRun` at a time
- Create a `DagRunJob` which is used to manage the execution of one or more `DagRuns`
- Implement both `SchedulerJob` and `BackfillJob` as subclasses of `DagRunJob`
  - The only difference is how they acquire `DagRuns` (`Scheduler` queries for all active `DagRuns`, `BackfillJob` works through a pre-specified list)
- Move much of the existing logic for figuring out the next scheduled run from the Scheduler to the DAG, where it belongs.
  - The Scheduler is just a special type of `DagRunJob`. Users may want to implement their own "Schedulers" or their own scheduling logic, and by moving it to the DAG that becomes possible.

## Description of New Workflow

DagRuns represent the state of a DAG at a certain point in time (**perhaps they should be called DagInstances?**). To run a DAG – or to manage the execution of a DAG – a DagRun must first be created. This can be done manually (simply by creating a DagRun object) or automatically, using methods like `dag.schedule_dag()`. Therefore, both scheduling new runs OR introducing ad-hoc runs can be done by any process at any time, simply by creating the appropriate object.

Just creating a DagRun is not enough to actually run the DAG (just as creating a TaskInstance is not the same as actually running a task). We need a Job for that. The DagRunJob is fairly simple in structure. It maintains a set of DagRuns that it is tasked with executing, and loops over that set until all the DagRuns either succeed or fail. New DagRuns can be passed to the job explicitly via `DagRunJob.submit_dagruns()` or by defining its `DagRunJob.collect_dagruns()` method, which is called during each loop. When the DagRunJob is executing a specific DagRun, it locks it. Other DagRunJobs will not try to execute locked DagRuns. This way, many DagRunJobs can run simultaneously in either a local or distributed setting, and can even be pointed at the same DagRuns, without worrying about collisions or interference.

The basic DagRunJob loop works like this:

1. refresh dags
2. collect new dagruns
3. process dagruns (including updating dagrun states for success/failure)
4. call executor/own heartbeat

By tweaking the DagRunJob, we can easily recreate the behavior of the current SchedulerJob and BackfillJob. The Scheduler simply runs forever and picks up ALL active DagRuns in `collect_dagruns()`; Backfill generates DagRuns corresponding to the requested start/end dates and submits them to itself prior to initiating its loop.

## Changes

- (major API changes are tracked in [UPDATING.md](#))
- DagRuns
  - A DagRun represents the execution of a certain DAG on a date, just as a TaskInstance represents the execution of a certain task on a date. As such, we don't want multiple DagRuns for the same dag/date, because they would all just point at the same taskinstances and therefore have no additional value.
    - This appears to already be enforced in Airflow via unique constraint but needs to be fleshed out. Right now it doesn't create any issues because only the Scheduler creates DagRuns, and only in sequence, but if Backfill also created a DagRun it could create a conflict with an existing scheduled DagRun.

If a DagRun uniquely identifies a dag/date, then a few current DagRun attributes become meaningless, in particular `id`, `run_id`, and `external_trigger`. For example, we don't need a `run_id` if (`dag_id`, `execution_date`) is sufficient to identify a DagRun. These fields would be helpful if more than one DagRun could point at a certain day, but that has never been allowed and is simply more formally enforced in Scheduler 2.0.

- \*\* The `conf` parameter should probably be rethought as well, for the same reason. It's tied to the method of execution. \*\*
- DagRuns can be locked (and unlocked). DagRunJob locks DagRuns when it's trying to execute them, and won't try to execute DagRuns if they are locked. This means multiple DagRunJobs can all run at the same time without stepping on each other. In particular, a BackfillJob might create DagRuns that are actually executed by the Scheduler. This sort of cooperation is ok -- DagRunJobs don't care how their DagRuns get executed, just that they do.
- DagRuns have a "run" method which loads up all the TaskInstances they cover and tries to execute them (one time).
- DAGs
  - `schedule_dag()` The logic for scheduling a DAG moves from the Scheduler to the DAG itself, where it can be more easily reused.
  - `update_dagrun_states()` The DAG can review its outstanding DagRuns and update their states (Pending -> Active, Active -> Success/Failed)
- Scheduler behavior
  - Scheduling logic
    1. if `schedule_interval == @once` and has never been scheduled, run the dag now
    2. PREVIOUSLY: if the DAG has never been scheduled, see if it's ever been run at all. If it has run, the first scheduled date is `4 * schedule_intervals` prior to that run. If it's never run, use the earliest task start date (but not the dag start date?)
 

NOW: if the DAG has never been scheduled, figure out the first date that SHOULD have been scheduled (probably `dag.start_date + schedule_interval`)
    3. if the DAG has been scheduled, add `schedule_interval` to the last scheduled date
    4. make sure the next run date is `>=` the dag's `start_date`
    5. make sure the next run date PLUS `schedule_interval` is `<=` now, indicating that the entire period has passed
  - In addition to scheduling new DagRuns, Scheduler tries to run ANY active DagRun (possibly restricted to specific `dag_ids`). This means Scheduler will try to run backfills and subdags, if they have created DagRuns.
  - Scheduler used to prioritize ALL queued tasks. Now it only prioritizes tasks that correspond to its active DagRuns (which could potentially mean ALL, but not necessarily)

## Issues

- Can't get DagRunModelView to create new DagRuns (mostly because I don't know flask well...)

# Classes

Notable additions and changes.

## DagRun

### Description

DagRun has been updated a lot. The new table looks like this:

```
__tablename__ = "dag_run"

dag_id = Column(String(ID_LEN), primary_key=True)
execution_date = Column(DateTime, default=func.now(), primary_key=True)
start_date = Column(DateTime, default=func.now())
end_date = Column(DateTime)
state = Column(String(50))
conf = Column(PickleType)
lock_id = Column(Integer)

__table_args__ = (
    Index('dr_dag_date', dag_id, execution_date, unique=True),
)
```

### Methods

- comparison operators (`==`, `<`, ...) for sorting and inclusion in sets
- `refresh_from_db`
  - Analogous to `TaskInstance.refresh_from_db()`
- `set_state()`
  - Logic for changing the DagRun's state. For example, moving from PENDING -> RUNNING sets the `start_date`; moving from RUNNING -> SUCCESS sets the `end_date`
- `set_conf()`
  - Update the conf object. Conf may need to be rethought -- it seems at odds with the uniqueness constraint.
- `lock()`
  - Marks the DagRun as locked by a certain `lock_id`. Used by `DagRunJob`.
- `unlock()`
  - Marks the DagRun as unlocked
- `run()`
  - Runs the DagRun by submitting all eligible tasks to the executor. Does NOT loop or make sure the DagRun completes; this is just one pass through the tasks. Returns a progress object with detailed information but what tasks were able to run.

## DAG

### Methods

- `on_schedule()`
  - checks if a date is on the DAG's schedule or not
- `schedule_dag()`
  - Formerly a method of `SchedulerJob`, this method creates new DagRuns according to the DAG's schedule.
- `update_dagrun_states()`
  - Formerly `get_active_dagruns()`, this method used to BOTH update `dagrun_states` AND return DagRuns for the Scheduler. Now it just updates states.

## DagRunJob

### Description

DagRunJob is a Job that has methods for executing and managing DagRuns. It has the following `_execute` structure:

```

def _execute(self):
    self.executor.start()

    i = 0
    while self.dagruns:
        self.refresh_dags(full_refresh=(i % self.refresh_dags_every == 0))
        self.collect_dagruns()
        self.process_dagruns()
        self.executor.heartbeat()
        self.heartbeat()
        i += 1

    self.executor.end()

```

## Methods

- submit\_dagruns()
  - Submit DagRuns for execution by placing them in self.dagruns
- collect\_dagruns()
  - Collects DagRuns from the database and submits them for execution. The base DagRunJob collects any unfinished DagRuns that have been locked by its own id. Under most circumstances, no DagRuns will meet that criteria -- but it provides an automated mechanism for jobs to pass DagRuns around.
  - The Scheduler uses this method to collect ALL unfinished DagRuns.
- refresh\_dags()
  - Reloads the DagBag
- process\_dagruns()
  - Attempts to run any DagRuns in self.dagruns. While DagRuns are being executed, the DagRunJob locks them so no other DagRunJob will try to run them. DagRuns that are FAILED or SUCCESS are removed from self.dagruns. After running all the DagRuns, calls prioritize\_queued() and dag.update\_dagrun\_states() for every dag in the dagbag.
- prioritize\_queued()
  - Collects all queued tasks across all DagRuns in self.dagruns and attempt to run them in the order implied by their global priority weights.
- manage\_slas()
  - Manage SLAs and send notification emails

## SchedulerJob

### Description

SchedulerJob is a subclass of DagRunJob with the following \_execute. Note that the only real difference from DagRunJob is the loop criteria and the call to schedule\_dags(). The Scheduler runs in a loop, just as the current Scheduler does, exiting when num\_runs is hit or possibly never.

```

self.executor.start()

i = 0
while not self.num_runs or self.num_runs > i:
    try:
        loop_start_dttm = datetime.now()
        self.logger.info('Starting scheduler loop...')
        try:
            self.refresh_dags(
                full_refresh=(i % self.refresh_dags_every == 0))
            self.schedule_dagruns()
            self.collect_dagruns()
            self.process_dagruns()
            self.manage_slas()

        except Exception as e:
            self.logger.exception(e)

        self.logger.info('Done scheduling, calling heartbeat.')

        self.executor.heartbeat()
        self.heartbeat()
    except Exception as e:
        self.logger.exception(e)

    i += 1

self.executor.end()

```

## Methods

- `schedule_dags()`
  - Examine all Dags (either in a specific list or in the `DAGS_FOLDER`) and schedule new dagruns.
- `collect_dagruns()`
  - The same as the basic `DagRunJob`, but includes all active `DagRuns` rather than only ones assigned to the job. In other words the Scheduler greedily tries to run all `DagRunJobs`. However, the locking mechanism means that it won't interfere with any other Jobs and if a `DagRun` is marked finished, it will stop trying to run it.

## BackfillJob

### Description

`BackfillJob` is a subclass of `DagRunJob` with this `_execute` structure:

```
self.heartbeat()
self.executor.start()

runs = [
    DagRun(dag_id=self.dag.dag_id, execution_date=dtm)
    for dtm in self.dag.date_range(
        start_date=self.bf_start_date, end_date=self.bf_end_date)]

self.submit_dagruns(runs)
self.target_runs = runs

while self.dagruns:
    self.collect_dagruns()
    self.process_dagruns()
    self.executor.heartbeat()
    self.heartbeat()

    progress = self.get_progress()
    self.logger.info(' | '.join([
        '[backfill progress: {pct_complete:.1%}]',
        'total dagruns: {total_dagruns}',
        'total tasks: {total_tasks}',
        'finished: {finished}',
        'succeeded: {succeeded}',
        'skipped: {skipped}',
        'failed: {failed}',
    ]).format(**progress))

self.executor.end()
```

`BackfillJob` prints progress, like this:

```
[2016-04-28 18:26:00,011] [jobs.py:912] INFO - [backfill progress: 0.0%] | total dagruns: 1 | total tasks: 2 | finished: 0 | succeeded: 0 | skipped: 0 | failed: 0
```

### Methods

The `BackfillJob` adds no new methods; its only difference from `DagRunJob` is that it generates and submits a list of `DagRuns` to itself.