Investigating the many loops of the Airflow Scheduler

Philippe Gagnon

Philippe Gagnon

Your speaker today

Solutions Architecture at Astronomer, inc.

Based in Montreal, Canada

Works on data platform architecture and implementation in heavily regulated industries since 2017, mostly around open-source

What is covered

1. Airflow task scheduling/execution components and their role
2. The scheduler initialization process
3. Task scheduling framework at a high level
4. The scheduler timers and what they do
5. How DagRuns are created
6. The scheduler ”critical section” and TaskInstance handling
7. How do executors pick up TaskInstances? CeleryExec and KubeExec
8. How does the task actually ”run”? CeleryExec and KubeExec

Schedulers… DAG Processors..? Executors… Workers ?!

Scheduler: Responsible for adding jobs to the queue when their dependencies are met and triggering the execution of tasks.

DAG Processor: Parses, processes and serializes the DAG files. It can either run as part of the scheduler, or standalone.

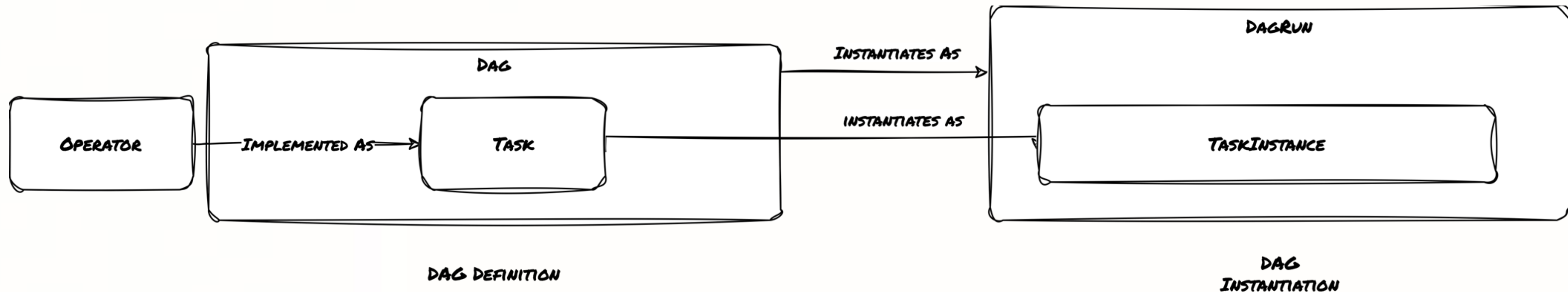
Executor: Component that actually runs or submits a task for execution. It runs as part of the scheduler.

Worker: Component that actually executes the tasks' payload. It runs an Operator's `execute` method.

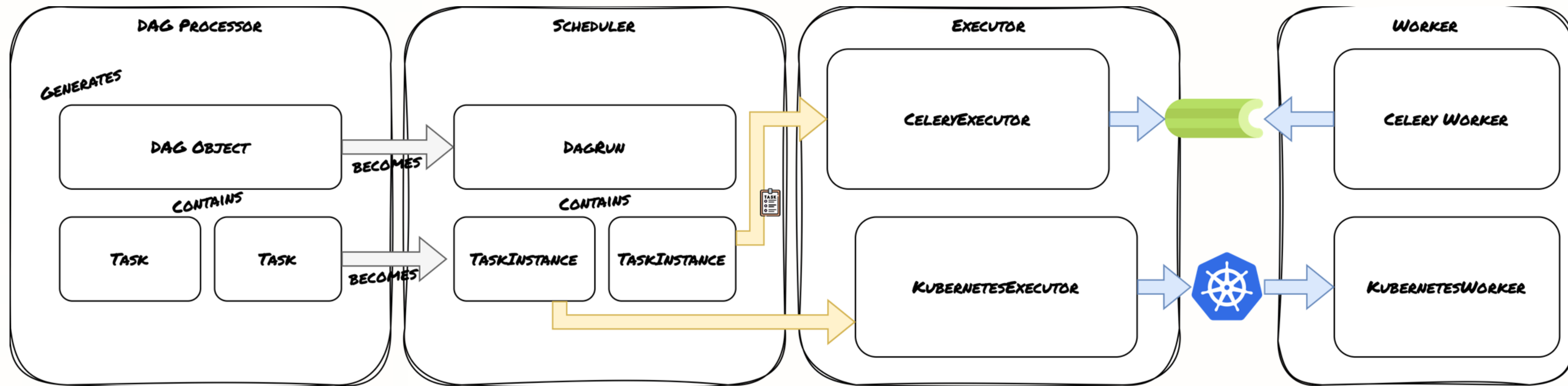
Triggerer: Runs and checks Triggers, which are asynchronous coroutines that monitor conditions after a task is deferred in order to resume it.

Tl;dr: DAGs, DagRuns, Tasks, TaskInstances

- A **DAG** is a DAG.
- A **task** is the implementation of an operator. It belongs to a **DAG**.
- A DagRun is the instantiation of a DAG, at runtime.
- A **TaskInstance** is the instantiation of a task, at runtime. It belongs to a **DagRun**.



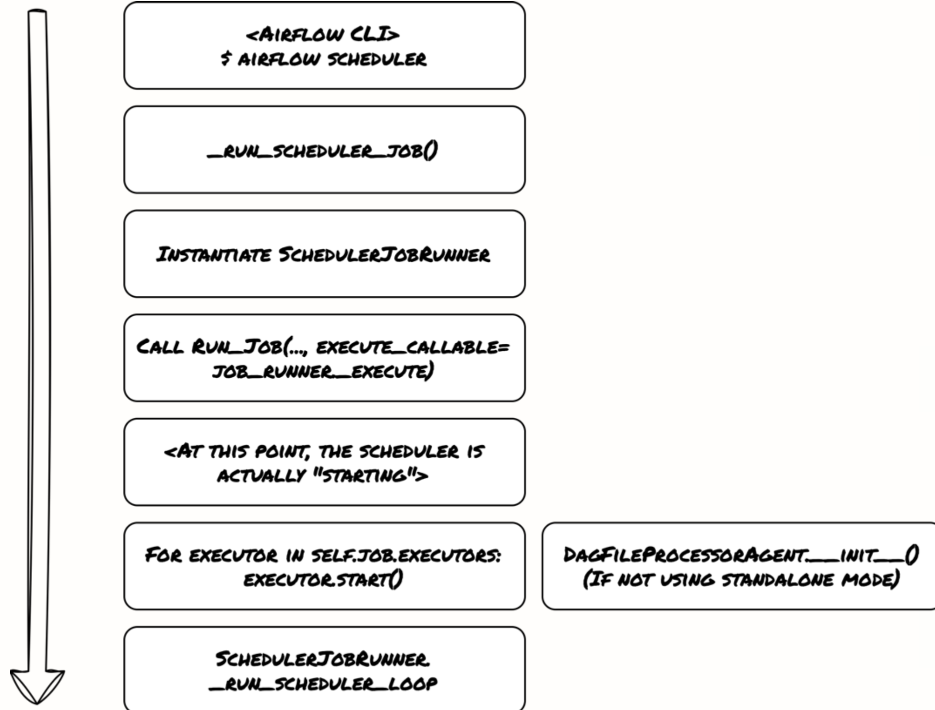
Task scheduling and execution framework



ASTRONOMER

The scheduler initialization process

- The first step needed to schedule tasks is to start the scheduler.
- At a high level, the CLI command will invoke `_run_scheduler_job`, which will instantiate a `SchedulerJobRunner`.
- The job runner will (1) run the main scheduler loop and (2) start the executors.



ASTRONOMER

SchedulerJobRunner._run_scheduler_loop

INITIALIZE TIMERS
(EVENTSCHEDULER)

FOR LOOP_COUNT IN ITERTOOLS.COUNT(START=1):

SELF._DO_SCHEDULING(...)

FOR EXECUTOR IN SELF.JOB.EXECUTORS:

EXECUTOR.HEARTBEAT(...)

PERFORM_HEARTBEAT(JOB=SELF.JOB)

ONLY IF NECESSARY!

TIMERS.RUN(...)

RUN ANY PENDING TIMED EVENTS

- At this point, we're ready to start the real scheduling loop.
- The core TaskInstance and DagRun scheduling logic is in the `_do_scheduling` method, and the executor logic is in the `heartbeat` method.
- We also run maintenance operations periodically.

ASTRONOMER

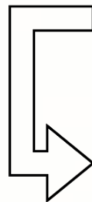
The loop timers

Method	Configuration Setting	Default
`adopt_or_reset_orphaned_tasks`	orphaned_tasks_check_interval	300
`check_trigger_timeouts`	trigger_timeout_check_interval	15
`_emit_pool_metrics`	pool_metrics_interval	5
`_find_zombies`	zombie_detection_interval	10
`_update_dag_run_state_for_paused_dags`	None!	60
`_fail_tasks_stuck_in_queued`	task_queued_timeout_check_interval	None!
`_orphan_unreferenced_datasets`	parsing_cleanup_interval	None!
`_cleanup_stale_dags`	parsing_cleanup_interval	None!

ASTRONOMER

SchedulerJobRunner._do_scheduling(...)

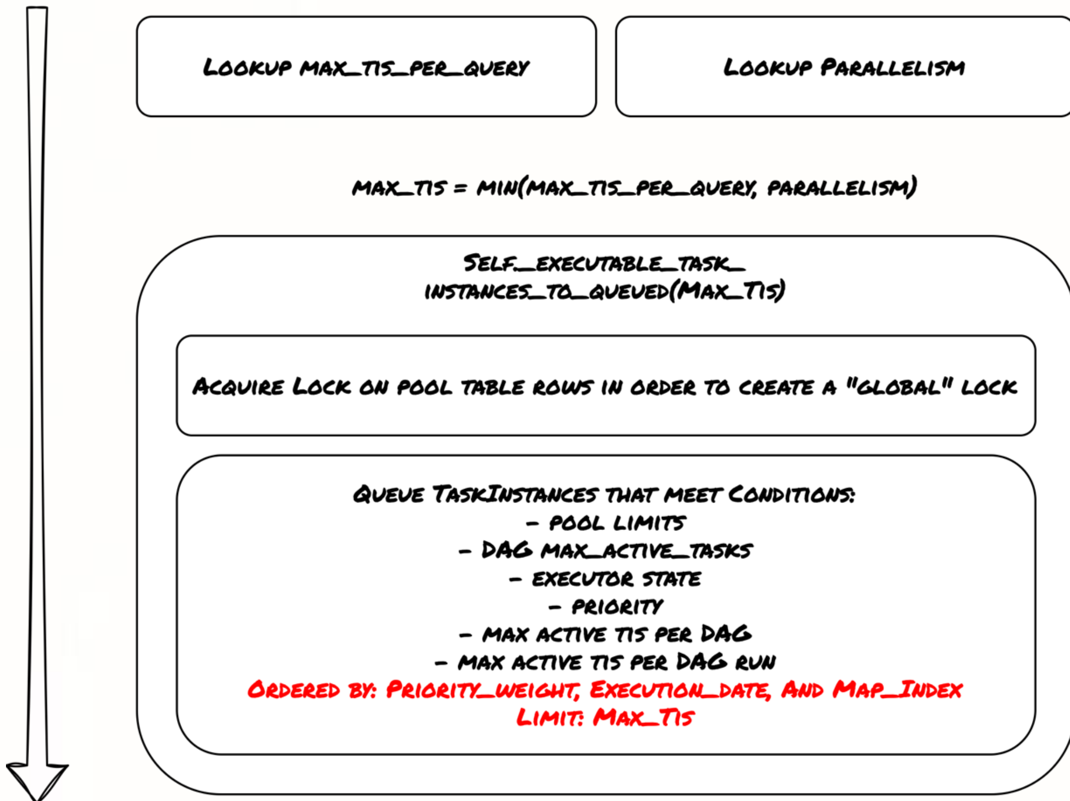
self._create_dagruns_for_dags(...)	Limit: scheduler.max_dagruns_to_create_per_loop (default: 10) Ordered By next_dagrun_create_after
self._start_queued_dagruns(...)	Limit: max_dagruns_per_loop_to_schedule (default: 20)
self._get_next_dagruns_to_examine(RUNNING)	Limit: max_dagruns_per_loop_to_schedule (default: 20)
self._schedule_all_dag_runs(...)	Schedule TaskInstances for Running DagRuns and Update next_dagrun_create_after
self._critical_section_enqueue_task_instances(...)	Selects TaskInstances in Scheduled State and Queues Them (which effectively makes them visible to the executor)



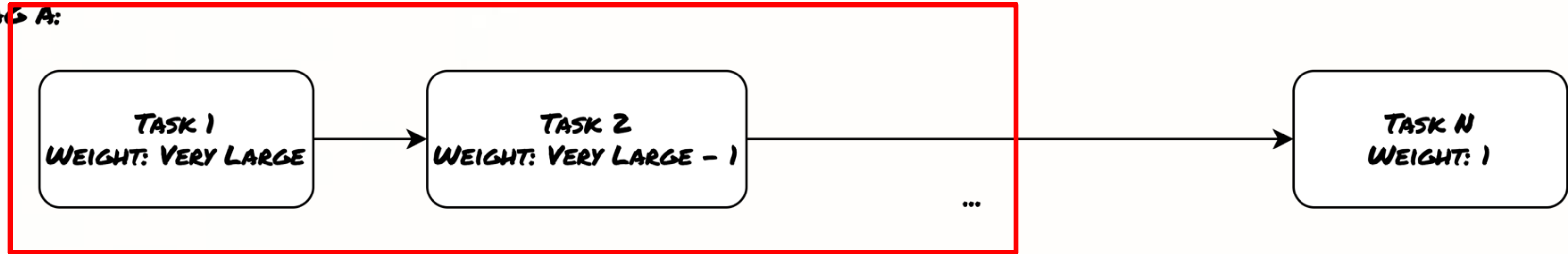
Bulk Fetch For

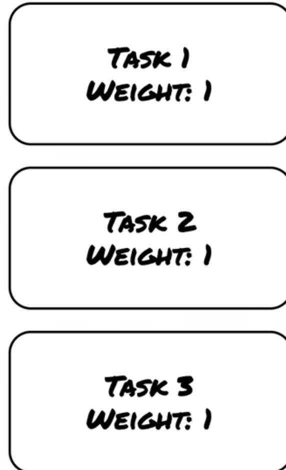
ASTRONOMER

The critical section

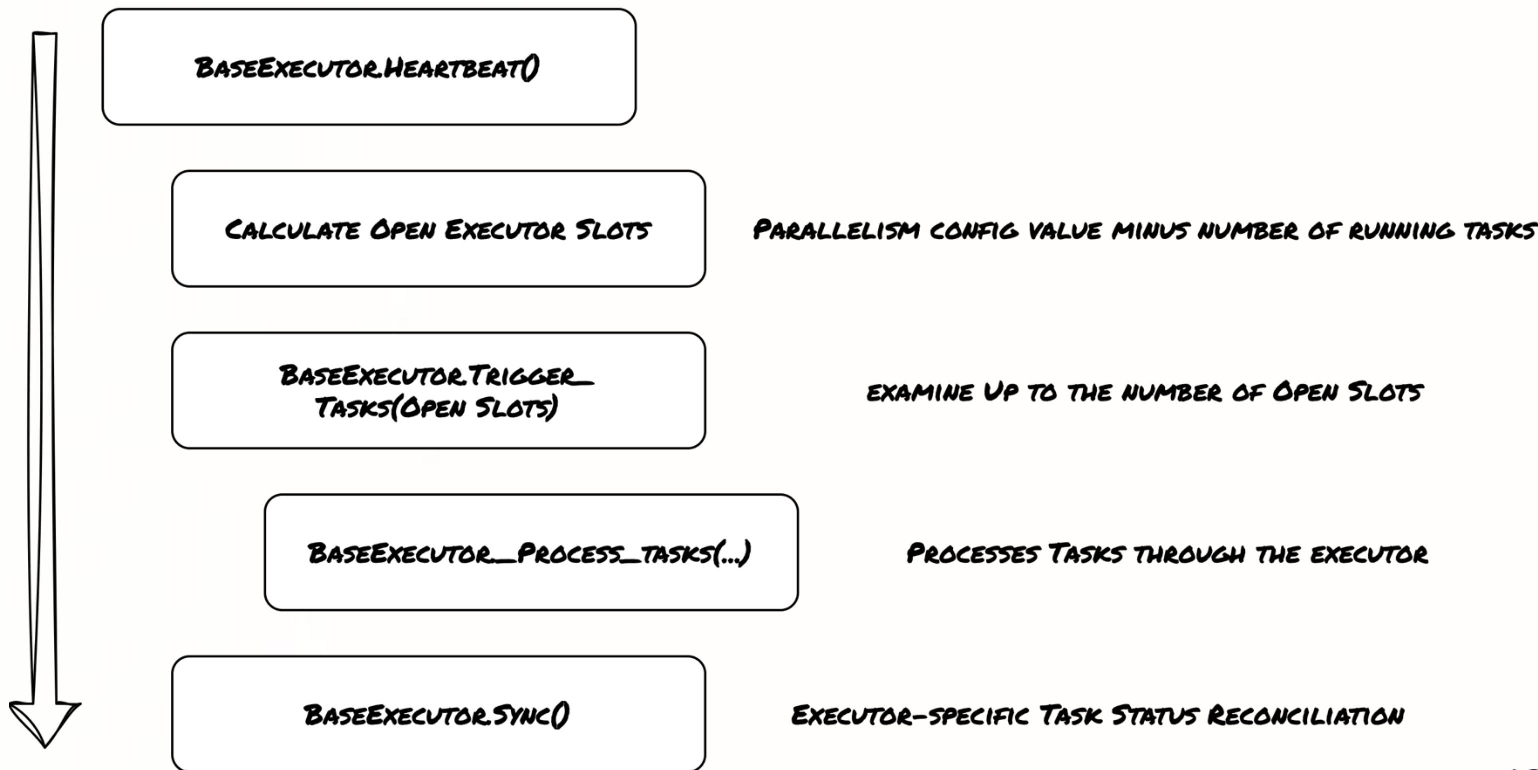


Impact of priority weight





The Executor Heartbeat Process



With CeleryExecutor (initialization)

- The CeleryExecutor initialization process is relatively simple.
- Everything is defined by the __init__ method.
- Two important objects are initialized:
  - BulkStateFetcher
  - Tasks map
- Of note, the start() method in CeleryExecutor does not do anything.
- It's important to note that with CeleryExecutor, we also need to start workers!

With CeleryExecutor

CeleryExecutor._process_tasks(...)

CeleryExecutor overrides the _process_tasks method

CeleryExecutor._send_tasks_to_celery(...)

with ProcessPoolExecutor(...):

Uses a process pool Executor to send tasks to celery in parallel (Up to Sync_Parallelism)

CeleryExecutor.send_task_to_executor(...)

task_to_run.apply_async(...)

Celery Library Method to run celery tasks

Return Value is Added to self.tasks collection for state tracking

At this point, the task should be queued in the celery backend

From scheduler's _do_scheduling method

CeleryExecutor.sync()

CeleryExecutor.update_all_task_states()

bulk_state_fetcher.get_many()

Get State for tasks using the best available method (depending on result backend)

For task in <result from above>:

If State:

CeleryExecutor.update_task_state(...)

Success

Failure

ASTRONOMER

CeleryWorker

Workers will prefetch "worker_prefetch_multiplier" times the number of celery processes number of tasks to run

CeleryExecutor

post tasks

Celery Broker

fetch results

Result Backend

Fetch task

post result

Airflow Celery worker

$ airflow tasks run ...

Airflow Celery worker

Airflow Celery worker

Airflow Celery worker

Each worker will process up to "worker_concurrency" tasks at the same time

With KubernetesExecutor (initialization)

- The KubernetesExecutor initialization process on the other hand is a lot more complex. This is because it tracks a lot more state than the CeleryExecutor.
- The main subcomponents we instantiate are (1) a task queue, (2) a result queue, (3) an AirflowKubernetesScheduler, (4) a Kube client, and (5) an event scheduler.
- Since some of these components are relatively complex to instantiate, we make use of the start() method for the actual instantiation.

With KubernetesExecutor (initialization)

Job.executors()

KubernetesExecutor.__init__(...)

Task_queue

Result_queue

SchedulerJobRunner._execute(...)

KubernetesExecutor.start()

kube_client

AirflowKubernetesScheduler

EventScheduler

AirflowKubernetes
Scheduler.__init__()

AirflowKubernetes
Scheduler.kube_watchers=
self._make_kube_watchers()

For namespace:

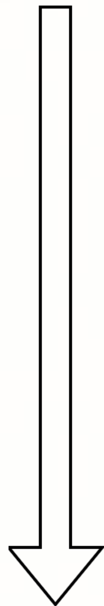
AirflowKubernetesScheduler.
make_job_watcher(watcher_queue)

Pass AirflowKubernetes
Scheduler.watcher_queue

KubernetesJobWatcher.start()

Multiprocessing.process that
watches events in Kube
namespaces

With KubernetesExecutor (execute_async)



BaseExecutor._process_tasks(...)

Gets a TaskInstance from BaseExecutor.queued_tasks

KubernetesExecutor.execute_async(...)

Generates Pod Specs and Command

KubernetesExecutor.task_queue.put(Task)

Puts (key, command, kube_executor_config, pod_template_file) in the queue.

With KubernetesExecutor (sync method)

BaseExecutor.heartbeat()

KubernetesExecutor.sync()

AirflowKubernetesExecutor.sync()

For all items in watcher_queue:

AirflowKubernetesExecutor.process_watcher_task(task)

AirflowKubernetesExecutor.process_watcher_task(task)

If done:

result_queue.put(<task tuple>)

For all items in result_queue:

KubernetesExecutor._change_state(item, state)

For i in range(worker_pods_creation_batch_size):

task = self.task_queue.get_nowait()

AirflowKubernetesScheduler.run_next(task) → PodGenerator.construct_pod(task) -> Pod(..., args=<command>) → AirflowKubernetesScheduler.run_pod_async(pod) → self.kube_client.create_namespaced_pod(pod)

KubernetesJobWatcher is continuously putting events in the queue. It runs in a subprocess.

while True:

KubernetesJobWatcher._run(...)

KubernetesJobWatcher.process_status(...)

KubernetesJobWatcher.watcher_queue.put(...)

We create up to worker_pods_creation_batch_size pods per iteration

ASTRONOMER

Conclusions and Takeaways

- The scheduling process follows the same steps at each iteration:
  - We create DagRuns
  - Queue DagRuns
  - Queue TaskInstances
  - Create new TaskInstances
  - Run the executor
- A task always “travels” from the scheduler to the executor to the worker.
- Configuration parameters are numerous and need to be tuned carefully according to your workload patterns.

Questions?

www.linkedin.com/in/pfgagnon