Deep dive into Airflow's Scheduler

Ash Berlin-Taylor,
PMC member @ Apache Airflow
Director of Airflow Engineering @ astronomer.io
Scheduler: The load-bearing infinite loop of Apache Airflow
Thank you for coming to my talk
Responsibilities of the scheduler

Start tasks on schedule
Check dependencies between tasks
Manage retries
Ensure task is actually still running
Deal with DST transitions
Be highly-available
SLAs

Trigger success/failure callbacks
Cope with changing DAG structure
Enforce concurrency limits
Emit metrics
Support trigger rules (one success, any failed etc.) including custom ones
Respect differing start_dates for tasks
Scheduler components

SchedulerJob ← State Machine for tasks and dag runs
Executor ← Handles actual task execution
DagFileProcessor ← Parses DAGs into serialized_dags table
"The" Scheduler

`airflow.jobs.scheduler_job`
Never load DAG code in to a long-running process
Scheduling decisions are only made upon serialized DAG representation
_do_scheduling()

processor_agent.heartbeat()

heartbeat()

timed_events.run()
SchedulerJob._do_scheduling()

self._create_dagruns_for_dags()

self._start_queued_dagruns()

dag_runs = self._get_next_dagruns_to_examine(State.RUNNING)
for dag_run in dag_runs:
    self._schedule_dag_run(dag_run)

num_queued_tis = self._critical_section_execute_task_instances()}
SchedulerJob._do_scheduling()

```
self._create_dagruns_for_dags()

self._start_queued_dagruns()

dag_runs = self._get_next_dagruns_to_examine(State.RUNNING)

for dag_run in dag_runs:
    self._schedule_dag_run(dag_run)

num_queued_tis = self._critical_section_execute_task_instances()
```

For each DAG* needing a DagRun to be created (next_dagrun_create_after < NOW()):
- Create the dag run from the serialized representation
- Update next DagRun info columns on DAG table (next_dagrun, next_dagrun_create_after)
self._create_dagruns_for_dags()

self._start_queued_dagruns()

dag_runs = self._get_next_dagruns_to_examine(State.RUNNING)
for dag_run in dag_runs:
    self._schedule_dag_run(dag_run)

num_queued_tis = self._critical_section_execute_task_instances()

**For each DAG in 'queued' state:**
- Check number of already running DagRuns against `dag.max_active_runs`
- If below limit set state to 'running'
def do_scheduling(self):
    self._create_dagruns_for_dags()
    self._start_queued_dagruns()
    dag_runs = self._get_next_dagruns_to_examine(State.RUNNING)
    for dag_run in dag_runs:
        self._schedule_dag_run(dag_run)
    num_queued_tis = self._critical_section_execute_task_instances()
SchedulerJob._do_scheduling()

self._create_dagruns()

self._start_queued_dagruns()

dag_runs = self._get_next_dagruns_to_examine(State.RUNNING)

for dag_run in dag_runs:

    self._schedule_dag_run(dag_run)

num_queued_tis = self._critical_section_execute_task_instances()

Check DagRun timeouts

Check if DAG structure (tasks) has changed

Compute which TaskInstances can now be 'scheduled' (via the currently-misnamed DagRun.update_state method)

Pass pending callbacks to DagFileProcessorManager
SchedulerJob._do_scheduling()

self._create_dagruns_for_dags()

self._start_queued_dagruns()

dag_runs = self._get_next_dagruns_to_examine(State.RUNNING)
for dag_run in dag_runs:
    self._schedule_dag_run(dag_run)

num_queued_tis = self._critical_section_execute_task_instances()

Check concurrency limits, and send as many tasks as possible to the executor
Before enqueueing a TaskInstance

Checks that must pass:

- Enough open pool slots available for task (can be >1 slot per task)
- Per DAG max_active_tasks limit
- Per (DAG, Task) task_concurrency limit
- Executor slots available (parallelism)

Everything else (task state, upstream etc) is checked before TaskInstance is put in to "scheduled" state
Executor
Send TaskInstance to runner to *actually* execute
Executor interface

(Interface/responsibilities between Scheduler and Executor needs clarification)

Tasks report their own status directly back to DB

Executor responsible for watching when tasks *don't* do this

State kept *in memory*
DAG parsing
airflow.dag_processing

Sole place where user DAG code is loaded

Previously split across airflow.job.scheduler_job and airflow.utils.dag_processing
DagFileProcessorManager

Subprocess of main airflow scheduler command

Infinite loop.

Maintains a pool of subprocess that:

- Parse a DAG file into serialized_dag table
- Execute any pending DAG level callbacks

Periodically checks for new DAG files being added
DagFileProcessorManager._run_parsing_loop

- Callback request from Scheduler
- _collect_results_from_processor
- start_new_processes
- Periodically: send heartbeat
- Periodically: _refresh_dag_dir
- "parse" dag file
- write DAGs to DB tables

Parsing process
High Availability
Use the existing metadata DB for synchronisation
SELECT * FROM task_instance
LIMIT 2

TaskInstance 1
TaskInstance 2
TaskInstance 3
TaskInstance 4

SELECT * FROM task_instance
LIMIT 2
Scheduler 1

SELECT * FROM task_instance
LIMIT 2

TaskInstance 1
TaskInstance 2
TaskInstance 3
TaskInstance 4

Scheduler 2

SELECT * FROM task_instance
LIMIT 2
SELECT * FROM task_instance
LIMIT 2 FOR UPDATE

TaskInstance 1
TaskInstance 2
TaskInstance 3
TaskInstance 4

Scheduler 1

Scheduler 2

SELECT * FROM task_instance
LIMIT 2 FOR UPDATE
SELECT * FROM task_instance
LIMIT 2 FOR UPDATE SKIP LOCKED
SchedulerJob._do_scheduling()

```
self._create_dagruns_for_dags()

self._start_queued_dagruns()

dag_runs = self._get_next_dagruns_to_examine(State.RUNNING)
for dag_run in dag_runs:
    self._schedule_dag_run(dag_run)

num_queued_tis = self._critical_section_execute_task_instances()
```
`SchedulerJob._do_scheduling()`

```python
with prohibit_commit(session) as guard:
    self._create_dagruns_for_dags(guard)

self._start_queued_dagruns(session)
guard.commit()
dag_runs = self._get_next_dagruns_to_examine(State.RUNNING, session)
for dag_run in dag_runs:
    self._schedule_dag_run(dag_run)
guard.commit()
num_queued_tis = self._critical_section_execute_task_instances()
```
critical_section_execute_task_instances
SELECT * FROM pool FOR UPDATE NOWAIT;
If we can't lock any rows, abort rather than wait

```sql
SELECT * FROM pool FOR UPDATE NOWAIT;
```
Adopting tasks

Periodically detect dead schedulers

"Adopt" tasks from dead executors

Means a scheduler/executer can go away (or partition) at any point

Active-active model.
Other responsibilities

Detecting dead schedulers
"Adopting" tasks from dead schedulers
Detecting zombie tasks
Managing SLAs
Optimization: check downstream states after task completion

After a Task executes, we have all the info to check it's downstream tasks.

Only goes as far as 'scheduled'

If "a" just finished, we can possibly schedule tasks b and c

Happens in the worker!