AIP-40: Deferrable ("Async") Operators

Status

State	Completed
Discussion Thread	[DISCUSS][AIP-40] Deferrable ("Async") Operators
Created	$action.dateFormatter.formatGivenString("yyyy-MM-dd", $content.getCreationDate())
In Release	2.2.0

Motivation

Smart Sensors solve an obvious need - making it a lot less resource-intensive to run many mostly-idle Sensors at once - but they are a relatively limited feature, and not very extensible.

We need a way to bring these sorts of efficiency gains to any Operator/Task that has a large amount of idle time, and on top of this, an API extensible enough that it can be extended out to future event-driven styles of operation.

Considerations

What change do you propose to make?

A new core concept is added to Airflow, called a **Trigger**. A Trigger is a small workload that is distinct from a Task; it is designed to be run alongside many other Triggers in an asynchronous (asyncio) environment. They fire *events* when their conditions are met.

The ability will be added for any Task/Operator to “defer” itself to keep running at a later time, providing a Trigger as part of the deferral.

Once a task requests to be deferred, Airflow stores a small amount of state describing how it should be resumed (a method to re-enter at, and optional keyword arguments), and then puts the task in a new “deferred” state, with it no longer taking up a worker/executor slot.

When the trigger fires an event, Airflow re-schedules the task to run again, passing the state it provided when it deferred itself, plus the event that woke it up.

Triggers run in a new process called a *triggerer* (analogous to the scheduler), which fetches and runs the triggers inside itself (all at once, because they are required to be asynchronous). When a Task requests deferral based on a Trigger, the trigger is dynamically loaded into the *triggerer* and run; once there are no tasks left that are waiting on it, the trigger is stopped and unloaded.

Triggers

Triggers have a relatively simple contract:

- They must provide an async-compatible `run` method that yields control when they are idle. This allows them to coexist with thousands of other Triggers within one process.
- They must be serializable to and from the database. This allows them to be run in a separate process from where they are declared, and is achieved via a standard “serialize” method which returns the classpath and keyword arguments to use to re-instantiate the Trigger.
- They must emit *events* (via a *yield* from their run function) whenever their conditions are met. Some triggers will emit one event and then exit; some may emit multiple events over time. In this first version, only the first event emitted will ever be used, but future plans (see below) will make use of multi-event triggers.
- They must be able to coexist with copies of themselves, so they can be run in a redundant fashion. Events must each have a distinct *payload* based on what triggered the event so they can be de-duplicated.

Here’s an example of a basic, single-shot datetime trigger:

```
class DateTimeTrigger(BaseTrigger):

    def __init__(self, moment: datetime.datetime):
        super().__init__()
        self.moment = moment

    def serialize(self) -> Tuple[str, Dict[str, Any]]:
        return ("airflow.triggers.temporal.DateTimeTrigger", {"moment": self.moment})

    async def run(self):
        while self.moment > timezone.utcnow():
            await asyncio.sleep(1)
        yield TriggerEvent(self.moment)
```

This has a `run()` method that sleeps (yielding control with `await`) until its time has been reached, and then fires a single event.

Triggers are stored in a new database table, `trigger`, with the following schema:

- `id` (bigint) - Autoincrementing single-column ID
- `classpath` (varchar) - The period-separated path to the Trigger class inside a module
- `kwargs` (json) - JSON-encoded keyword arguments to pass to the trigger's constructor
- `created_date` (datetime) - When the trigger was created

In the first version, one instance of a trigger will be made per deferred task, but the schema and API contract is designed so that, in future, deduplication of triggers could be performed.

Triggers *must* yield back control via await or yield quickly, as otherwise the nature of Python's async implementation means they will block the whole event loop. We can ship a coarse detector inside Airflow that ensures this situation isn't happening, but we unfortunately can't tell exactly *which* trigger is doing it if it happens - Python's slow-async-callback detection is not customisable or accessible, so we will either have to manually force asyncio's debug mode to on (so it alerts users to long-running callbacks), or advise our users how to do this.

Triggerer

A separate *triggerer* process runs at all times, and would form a new continuously-running-process part of an Airflow installation. It contains an asyncio event loop where it can run thousands of triggers at once efficiently.

This process monitors the `trigger` table and makes sure that it is always running all the triggers that are defined in there - adding new ones that appear to the event loop, and removing ones whose tasks are completed - as well as monitoring the triggers it is running continuously.

When an event fires from a trigger, the triggerer looks up which tasks depend on that trigger, and moves those tasks to the *scheduled* state, as well as adding the event's payload to their kwargs. It then disables the trigger and removes it from the event loop if that was the only task depending on it.

The triggerer is designed to be run in a highly-available (HA) manner; it should coexist with other instances of itself, running the same triggers, and deduplicating events when they fire off. In future, it should also be designed to run in a sharded/partitioned architecture, where the set of triggers is divided across replica sets of triggerers.

Task Deferral

In order for a Task/Operator to be deferred, it should throw a new exception, `TaskDeferred`, from somewhere within its `execute()` method. This exception takes four arguments:

- The trigger instance to base the deferral upon (the task will be resumed when it fires an event)
- The method name to call when the task is resumed, rather than `execute()`. This allows Operators to separate their logic into several steps, rather than cramming it all into one method and using lots of conditionals.
- The keyword arguments to pass to that method when the task is resumed. This allows the operator to persist a small amount of state between its invocations.
- An optional timeout, after which the task will be marked as `failed` if its trigger has not fired.

There will also be a convenience method provided on BaseOperator called `defer()`, so you could write the following code inside an Operator:

```
def execute(self, context):
    # This fires the TaskDeferred exception and suspends execution at this point
    self.defer(trigger=DateTimeTrigger(moment=self.target_time), method_name="execute_complete")

def execute_complete(self, context, event):
    # The Operator comes back here when the trigger fires due to "method_name"
    return
```

The state lifecycle of a deferred task is:

- *Running*, until the task encounters the deferral point
- *Deferred*, where it is blocked on a trigger
- *Scheduled*, once the trigger has fired and it is ready to be run again
- *Running*, once it resumes execution

In order to implement deferral, the `_execute_task()` wrapper on TaskInstance will catch the exception and handle the persisting of state to the database as well as setting up the trigger to run. It will also look at the new columns when a task resumes to see if it needs to use a different method to enter the operator, as well as what keyword arguments to pass.

These changes mean three columns need to be added to the `task_instance` table:

- `trigger_id` (bigint), a reference to the trigger table
- `trigger_timeout` (datetime), the optional UTC time when this deferral expires and the task instance should fail
- `next_method` (varchar), the name of the method to run on the Operator if it is not execute
- `next_kwargs` (json), the keyword arguments to pass to that method

Additionally, a new possible value of `state` is added, "deferred". The scheduler will be updated to understand this as a pending type of task state, and not consider a DAG deadlocked when its tasks are all deferred.

What problem does it solve?

It solves the problem where both Sensors and externally-dependent Operators spend most of their time idle, but occupying a worker slot.

Implementing this AIP will result in very significant efficiency gains in any Airflow installation that contains these.

Why is it needed?

Smart Sensors attempt to solve this problem but require specific implementation per Sensor and cannot be run in a highly-available manner. The "reschedule" mode of Sensors also tries to do this, but in a time-driven matter rather than an event-driven manner, which leads to inefficiencies if you want a relatively low-latency response. Additionally, both of these are Sensor-specific and will not work with more general Operators/Tasks.

Are there any downsides to this change?

It adds one new process that needs to continuously run on any Airflow installation that uses deferrable operators.

It adds an implementation that is a superset of Smart Sensors and so duplicates some functionality.

Which users are affected by the change?

Users who wish to use deferrable operators will have to run a new process. Nobody else will be affected in the short term apart from a database migration.

In the long term, if Airflow changes some of its core Sensors to be deferrable, all users will have to run the *triggerer* process, but any DAGs that use those sensors will get an immediate performance boost with no code changes.

Any existing Operator or Sensor could be ported to be deferrable underneath with no change to its DAG-facing API, which would allow relatively seamless upgrades.

How are users affected by the change? (e.g. DB upgrade required?)

A database migration is required for all users, and running a new long-running process is required for users who wish to use deferrable operators.

Other considerations?

Prototype Implementation

In order to validate the ideas presented, this has already been implemented in a prototype form to ensure all the changes are possible and co-exist well with Airflow.

The prototype branch can be seen in draft pull request 15389.

It provides one new async sensor, `AsyncTimeSensor`, which is a drop-in replacement for the existing `TimeSensor`, but that doesn't take up a worker slot while it's waiting.

Note that the prototype is rough, and would receive significant further work before it was landed to complete this AIP.

Future Plans

This proposal forms the basis of an ongoing ability to modify the core of Airflow to be more event-driven. Some potential changes this opens up in future - that are not part of the scope of this AIP, but which are worth considering in terms of general direction - are:

- Deprecating the `reschedule` mode on Sensors. This could now be achieved using a time-based trigger instead, and remove the need for sensors to have execution modes.
- Allowing a similar deferral operation in the TaskFlow API. This would be more complex to make "clean" as users will expect state to persist, but could be provided relatively easily with some decent safeguards.
- Allowing DAGs to mix in Triggers directly as elements of the graph. A basic Sensor wrapper could be added that auto-wraps any declared triggers with enough of a Task that it functions as users expect.
- Re-using Triggers to also trigger DAGs as well as un-defer Tasks. This would be a large lift, but sets the stage for a more streaming-oriented Airflow, as well as allowing us to add more diverse ways of triggering DAGs (e.g. webhooks).

Excluded Alternatives

Some alternate approaches were considered but ultimately excluded from consideration for this AIP:

- Having a separate "async task" type that only ran on async workers. This would have involved a lot more API changes and touched every Executor, and ultimately for a less flexible outcome.
- Making Triggers a Task subset, like Sensors. This would have meant a much more "open" contract that was harder to optimise, and runs into many of the same problems that the "async task" idea does.
- Running Triggers inside DAGs rather than inside a new process. This is how Smart Sensors work, and while it seems attractive due to the lack of new processes needed, DAGs are not designed to run for a very long time or in parallel, so it would harm reliability.
- Not using asyncio, and instead using another event loop/our own cooperative multitasking. This makes no sense due to the very strong library ecosystem around asyncio.
- Persisting all of the state of an Operator from its class body when it is suspended. This was excluded due to it being mostly unnecessary and hard to get right (we'd need a list of included fields, and not all values can be JSON-ified, even with Airflow's extended JSON serializer)

What defines this AIP as "done"?

When the deferral functionality is merged into the master branch.