Airflow Summit 2022

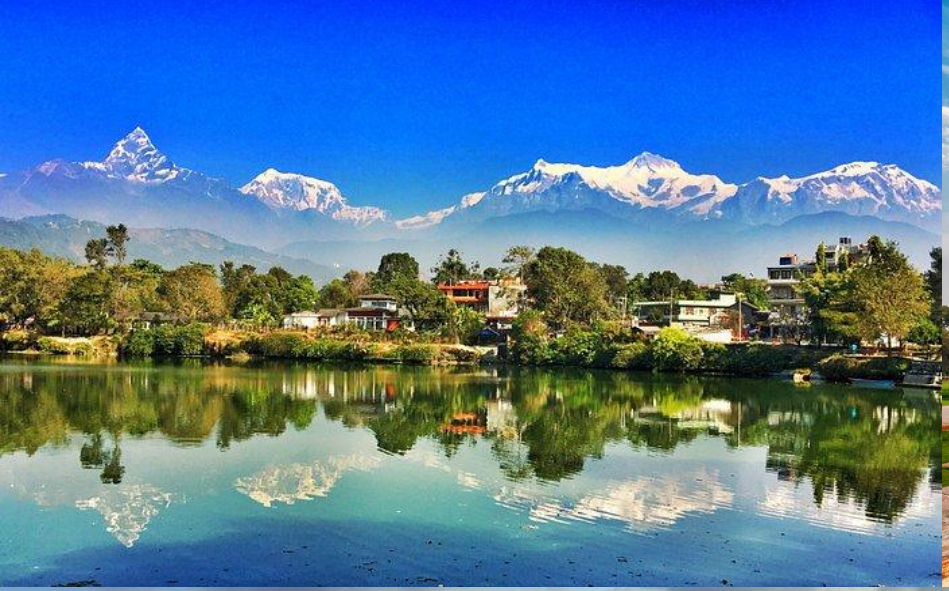
Love for writing deferrable operators. Why and how to defer?

Ankit Chaurasia

Hi, I'm

Ankit Chaurasia

- Senior Software Engineer at Astronomer
- Using airflow since April 2021




Airflow
Summit 2022

Airflow Summit 2022

1. **Standard Operators**
   Issues around it
2. **Motivation and high level concepts**
   Need for Deferrable Operator
3. **Deep Dive**
   Deferrable Operator
4. **Implement real world Deferrable Operator**
   Code your own Deferrable Operator
5. **Pitfalls and learning**

Standard Operators

Standard **Operators** and **Sensors** take up a full worker slot for the entire time they are running (even if they are idle).

Sensors have two types of **mode**: poke and reschedule.

poke

The sensor takes up a worker slot for its entire runtime; this mode is best if you expect a short runtime for the sensor.

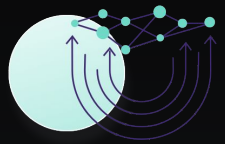
reschedule

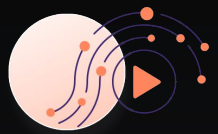
The sensor takes up a worker slot only when it is checking, and sleeps for a set duration between checks; this mode is best if you expect a long runtime for the sensor, because it is less resource intensive and frees up workers for other tasks.

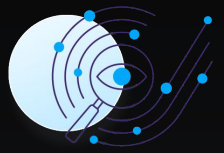
Traditional way

Introduction to deferrable operator

Operator with ability to **suspend itself** and **free up the worker** when it knows it has to wait, and hand off the job of resuming it to **a Trigger**







Why - Deferrable Operator and sensors?

- (Save your dollars :p) Massive resource savings on long-running operators (e.g. Spark, S3 Sensors)
- Deferrable Operator framework released in Airflow 2.2

Conducted an experiment to measure impact of using deferred/async sensors via a triggerer compared to regular sensors that occupy a slot while running.

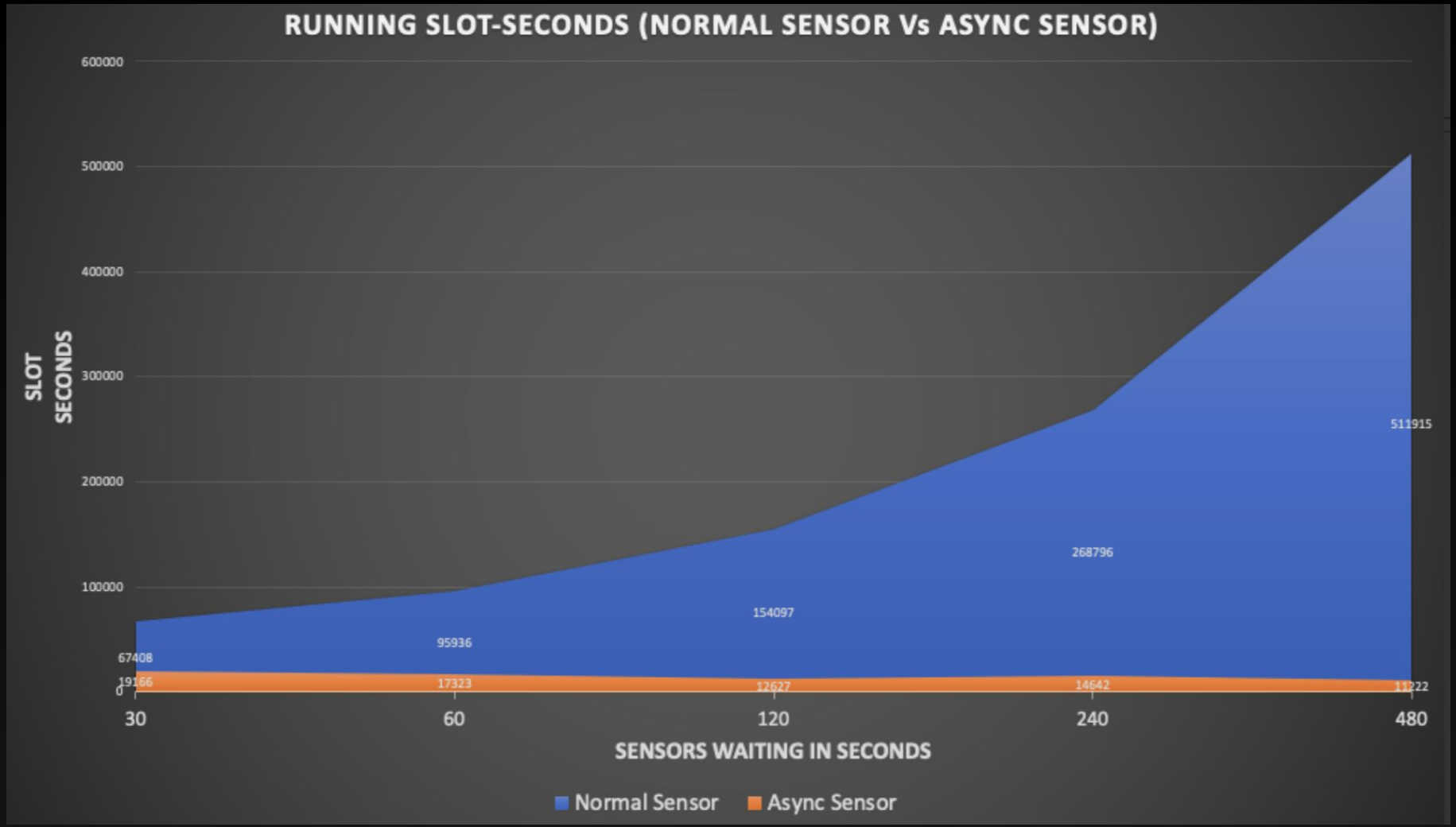
BenchMark

Conducted an experiment to measure the impact of using deferred/async sensors via a triggerer compared to regular sensors that occupy a slot while running. Our hypothesis was that we will observe lower slot consumption on deferred/async sensors.

Metrics used: The total number of minutes a task spends running on a worker, summed over all tasks (slot usage seconds)

Slots-seconds for a 1000-task DAG run

sleep before create	running	running (async)	improvement	occupied	occupied (async)	improvement
30s	67408	19166	71.6%	92500	61798	33.2%
1m	95936	17323	81.9%	124752	55741	55.3%
2m	154097	12627	91.8%	186224	41135	77.9%
4m	268796	14642	94.6%	300157	45061	85.0%
8m	511915	11222	97.8%	542793	35890	93.4%


RUNNING SLOT-SECONDS (NORMAL SENSOR Vs ASYNC SENSOR)
600000
500000
400000
300000
200000
100000
0
SLOT SECONDS
67408
95936
154097
268796
511915
19166
17323
12627
14642
11222
30
60
120
240
480
SENSORS WAITING IN SECONDS
Normal Sensor
Async Sensor

The Trigger and the Triggerer

The Trigger

- Small, asynchronous Python function that quickly and continuously evaluates a given condition.
- Thousands of Triggers can be run at once in a single process(Triggerer).

The Triggerer

- This is a new airflow service (like a scheduler or a worker) that runs an asyncio event loop in your Airflow environment. Running a triggerer is essential for using deferrable operators.
- Responsible for running Triggers and signaling tasks to resume when their conditions have been met.
- Designed to be highly-available.

New terms

asyncio

- This Python library is used as a foundation for multiple asynchronous frameworks.
- It’s core to deferrable operator's functionality, and is used when writing triggers.

Deferred

- This is a new Airflow task state (medium purple color) introduced to indicate that a task has paused its execution, released the worker slot, and submitted a trigger to be picked up by the triggerer process.

How it works

The process for running a task using a deferrable operator:

Running state

The Task is picked up by a Worker

Deferred state

The task defines a Trigger and defers the function of checking on some condition to the Triggerer

The Triggerer

The Triggerer runs the task's Trigger periodically to check whether the condition has been met

Queued state

Once the Trigger condition succeeds, the task is again queued by the Scheduler

How it works


Submit Job to Spark Cluster
Poll Spark Cluster for Job Status
Receive Terminal Status for Job on Spark Cluster
Worker Slot Allocated
Triggerer Process
Worker Slot Allocated

Writing deferrable operator

Writing a deferrable operator takes a bit more work. There are some main points to consider:

- Deferrable Operators & Triggers rely on more recent asyncio features, and as a result only work on Python 3.7 or higher.
- Ensure your Airflow installation is running at least one triggerer process, as well as the normal scheduler.
- Your Operator must defer itself with a Trigger. If there is a Trigger in core Airflow you can use, great; otherwise, you will have to write one.
- Your Operator will be stopped and removed from its worker while deferred, and no state will persist automatically. You can persist state by asking Airflow to resume you at a certain method or pass certain kwargs, but that's it.
- You can defer multiple times, and you can defer before/after your Operator does significant work, or only defer if certain conditions are met (e.g. a system does not have an immediate answer). Deferral is entirely under your control.
- Any Operator can defer; no special marking on its class is needed, and it's not limited to Sensors.
- No sync operations inside the triggers
- Writing a async hook that the operator will use

Structure of Deferrable Operator

```python
class MyOperator(BaseOperator):
    def __init__(self,target_time, **kwargs):
        super().__init__(**kwargs)
        self.target_time = target_time

    def execute(self, context):
        # This fires the Task Deferred 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
```

Writing Trigger

Airflow Summit 2022

A Trigger is written as a class that inherits from *BaseTrigger*, and implements three methods:

- *__init__*, to receive arguments from Operators instantiating it
- *run*, an asynchronous method that runs its logic and yields one or more *TriggerEvent* instances as an asynchronous generator
- *serialize*, which returns the information needed to re-construct this trigger, as a tuple of the classpath, and keyword arguments to pass to *__init__*

```python
class DateTimeTrigger(BaseTrigger):

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

    def serialize(self):
        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)
```

Hooks

- Operators rely heavily upon Hooks to interact with all of their source and destination systems.
- Hooks provide an interface in which to interact with an external system, but do not contain the logic for how that system is interacted with.
- Writing a async hook that the operator will use

```python
class S3HookAsync(AwsBaseHookAsync):
    """Interact with AWS S3, using the aiobotocore library."""

    conn_type = "s3"
    hook_name = "S3"

    def __init__(self, *args: Any, **kwargs: Any) -> None:
        kwargs["client_type"] = "s3"
        kwargs["resource_type"] = "s3"
        super().__init__(*args, **kwargs)

    @staticmethod
    async def _check_exact_key(client: AioBaseClient, bucket: str, key: str) -> bool:
        """
        Checks if a key exists in a bucket asynchronously

        :param client: aiobotocore client
        :param bucket: Name of the bucket in which the file is stored
        :param key: S3 key that will point to the file
        :return: True if the key exists and False if not.
        """
        try:
            await client.head_object(Bucket=bucket, Key=key)
            return True
        except ClientError as e:
            if e.response["ResponseMetadata"]["HTTPStatusCode"] == 404:
                return False
            else:
                raise e
```

Demo of deferrable S3KeySensorAsync

Considerations while writing Async or Deferrable Operator

- Deferrable Operators & Triggers rely on more recent asyncio features, and as a result only work on Python 3.7 or higher.
- Any Deferrable Operator implementation needs the API used to give you a unique identifier in order to poll for the status in the Trigger. This does not affect creating an async Sensor as "sensors" are just poll-based whereas "Operators" are "Submit + Poll" operation. For example in the below code snippet, the Google BigQuery API returns a job_id using which we can track the status of the job execution from the Trigger.

```
job = self._submit_job(hook, configuration=configuration)
self.job_id = job.job_id
self.defer(
    timeout=self.execution_timeout,
    trigger=BigQueryGetDataTrigger(
        conn_id=self.gcp_conn_id,
        job_id=self.job_id,
        dataset_id=self.dataset_id,
        table_id=self.table_id,
        project_id=hook.project_id,
    ),
    method_name="execute_complete",
)
```

Considerations while writing Async or Deferrable Operator [continued]

- See if the official library supports async, if not find a third-party library that supports async calls. For example, *pip install apache-airflow-providers-snowflake* also installs s*nowflake-connector-python* which officially support async calls to execute the queries. So it is used directly to implement deferrable operators for Snowflake. But many providers don't come with official support for async like Amazon. If not some research to find the right third-party library that support calls is important. In case of Amazon, we use aiobotocore for Async client for amazon services using botocore and aiohttp/asyncio.
- Inheriting the sync version of the operator wherever possible so boilerplate code can be avoided while keeping consistency. And then replacing the logic of the execute method.
- Logging: Passing the Status of the task from Trigger to the Operator or Sensors so the logs show up in the Task Logs since Triggerer logs don’t make it to Task Logs

Some common Pitfalls

- From Operator we cannot pass a class object to Trigger because serialize method will only support JSON-serializable values. It will be serialized into the database.
- At times the async implementation might require to call the synchronous function. We use asgiref sync_to_async function wrappers for this. sync_to_async lets async code call a synchronous function, which is run in a threadpool and control returned to the async coroutine when the synchronous function completes. For example

```
async def service_file_as_context(self) -> Any:  # noqa: D102
    sync_hook = await self.get_sync_hook()
    return await sync_to_async(sync_hook.provide_gcp_credential_file_as_context)()
```

- While implementing trigger serialize method, it's important to use the correct class name.

```
def serialize(self) -> Tuple[str, Dict[str, Any]]:
    """Serialize S3KeyTrigger arguments and classpath."""
    return (
        "astronomer.providers.amazon.aws.triggers.s3.S3KeyTrigger",
        {
            "bucket_name": self.bucket_name,
            "bucket_key": self.bucket_key,
            "wildcard_match": self.wildcard_match,
            "aws_conn_id": self.aws_conn_id,
            "hook_params": self.hook_params,
        },
    )
```

The benefits

Reduced resource consumption

Reduction in the number of workers needed to run tasks during periods of high concurrency.

Resiliency against restarts

Deferred tasks will not be set to a failure state if a triggerer needs to be restarted due to a deployment or infrastructure issue.

Event-based DAGs

The presence of asyncio in core Airflow is a potential foundation for event-triggered DAGs.

When to use deferrable operators

We can create Async operators for the "sync-version" of operators that do some level of polling (take more than a few seconds to complete)

For example, we won't create an async Operator for a BigQueryCreateEmptyTableOperator but will create one for BigQuerylnsertJobOperator that actually runs queries and can take hours in the worst case for task completion

- File system based operations
- Network-backed operations
- Time taking task which can be executed async
- DB based operations like executing long running query in async operations
- Poke operations
- Any I/O bound task

Astronomer's Deferrable Operators

The operators are drop-in replacements for non-Deferrable operators, meaning that you only have to change the import statements in your DAGs to begin using them.

```
from astronomer.providers.amazon.aws.sensors.s3 import S3PrefixSensorAsync as S3PrefixSensor
```

Package **Astronomer Providers** contains **24 Async Operators/Sensors** that are publicly available. For more information, see the CHANGELOG in astronomer-providers.

Guide to implement your deferrable operator with astronomer-providers repository as an example to start with.

References for useful concepts and libraries: Concurrency, Python asyncio

- https://threadreaderapp.com/thread/1484237419494838272.html
- https://redis.com/blog/async-await-programming-basics-python-examples/
- https://realpython.com/async-io-python/

Questions?

Thank you :)