Reliable Airflow DAG Design when building a Time-series Data Lakehouse

Sung Yun
Enterprise Data Lake, Bloomberg
# Time-series Data

<table>
<thead>
<tr>
<th>Date</th>
<th>Company ID</th>
<th>Price</th>
<th>P/E Ratio</th>
<th>Industry Sector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1992-09-16</td>
<td>Company A</td>
<td>15000</td>
<td>5</td>
<td>Tech</td>
</tr>
<tr>
<td>1992-09-16</td>
<td>Company B</td>
<td>5000</td>
<td>10</td>
<td>Tech</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2023-09-18</td>
<td>Company A</td>
<td>23600</td>
<td>19</td>
<td>Tech</td>
</tr>
<tr>
<td>2023-09-18</td>
<td>Company B</td>
<td>19000</td>
<td>11</td>
<td>Tech</td>
</tr>
<tr>
<td>2023-09-19</td>
<td>Company A</td>
<td>23500</td>
<td>20</td>
<td>Tech</td>
</tr>
<tr>
<td>2023-09-19</td>
<td>Company B</td>
<td>21000</td>
<td>10</td>
<td>Tech</td>
</tr>
</tbody>
</table>

Disclaimer: dummy data
Human Expectations

“Daily starting Tuesday, September 19th, 2023, between 5:30 PM and 7:30 PM EDT”

Recurrence Pattern
Start Date and Time
Expected Delivery Window
## Time-series Analysis

“Daily starting Tuesday, September 19th, 2023, between 5:30 PM and 7:30 PM EDT”

<table>
<thead>
<tr>
<th>Date</th>
<th>Company ID</th>
<th>Price</th>
<th>P/E Ratio</th>
<th>Industry Sector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1992-09-16</td>
<td>Company B</td>
<td>5000</td>
<td>10</td>
<td>Tech</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2023-09-18</td>
<td>Company A</td>
<td>23600</td>
<td>19</td>
<td>Tech</td>
</tr>
<tr>
<td>2023-09-18</td>
<td>Company B</td>
<td>19000</td>
<td>11</td>
<td>Tech</td>
</tr>
</tbody>
</table>

Disclaimer: dummy data
Designing for Reliability

- Recoverability
- Scalability
- Failure and Delay Detection (SLA Miss Detection)
Recoverability

Making tasks safe to re-run
Example DAG Run

Source Data Sensor

Spark Transform & Ingest to Lakehouse

Quality Control
Making Each Task Safe to Re-run

Make each write consistent on retries

1. Use `data_interval_start` or `data_interval_end` to query source data
2. And also as the write partition in your open table format
Making Each Task Safe to Re-run

Make each write atomic

1. Use snapshot isolation to ensure no in-between states are exposed to other writers or readers
2. Use a write-optimized lakehouse partition strategy
3. Utilize operations that atomically replace an entire partition
Retry on Failure

Enable Retries

1. Enable automatic task level retries with ‘retries’ parameter
2. Invoke a retry manually by ‘clearing’ a task

**FAILED** → **QUEUED**

**SUCCESS** → **QUEUED**

```python
with DAG(
    dag_id="my_dag",
    start_date=pendulum.datetime(2016, 1, 1),
    schedule="@daily",
    default_args={"retries": 2},
)
:
    op = BashOperator(
        task_id="hello_world",
        bash_command="Hello World!"
    )

    op2 = BashOperator(
        task_id="good_bye_cruel_world",
        bash_command="Good Bye!"
    )

    op >> op2
```
Scalability

Optimizing resource utilization
Scalability

DAG start

Source data expected delivery window

ETL

Expected DAG finish
Hogging resources, only when we need them

DAG start

Source data expected delivery window

Sensor → ETL

Expected DAG finish
import time
from datetime import timedelta
from typing import Any

from airflow.configuration import conf
from airflow.sensors.base import BaseSensorOperator
from airflow.triggers.temporal import TimeDeltaTrigger
from airflow.utils.context import Context

class WaitOneHourSensor(BaseSensorOperator):
    def __init__(self, deferrable: bool, **kwargs) -> None:
        super().__init__(**kwargs)
        self.deferrable = deferrable

    def execute(self, context: Context) -> None:
        if self.deferrable:
            self.defer(
                trigger=TimeDeltaTrigger(timedelta(hours=1)),
                method_name="execute_complete",
            )
        else:
            time.sleep(3600)

    def execute_complete(self, self, context: Context, event: dict[str, Any] | None = None,):
        # We have no more work to do here. Mark as complete.
        return None

Asynchronous Sensors (Deferrable)

```python
import time
from datetime import timedelta
from typing import Any
from airflow.configuration import conf
from airflow.sensors.base import BaseSensorOperator
from airflow.triggers.temporal import TimeDeltaTrigger
from airflow.utils.context import Context

class WaitOneHourSensor(BaseSensorOperator):
    def __init__(self, deferrable: bool, **kwargs)
        super().__init__(**kwargs)
        self.deferrable = deferrable

    def execute(self, context: Context) -> None:
        if self.deferrable:
            self.defer(
                trigger=TimeDeltaTrigger(timedelta(hours=1)),
                method_name="execute_complete",
            )
        else:
            time.sleep(3600)

    def execute_complete(self, context: Context, event:
        dict[str, Any] | None = None) -> None:
        # We have no more work to do here. Mark as complete.
        return
```

Failure Detection

Failed:
- S3 client failure - rate limit
- Credentials issue

Delay Detection:
- Source File is not available in expected window

Failed:
- S3 client failure - rate limit
- Credentials issue
- Spark SQL transformation failure

Defined hanging Kubernetes Spark Job

Accountable Owner

Source Data Sensor

Kubernetes Spark Operator

dag.dagrun_timeout: timedelta
task.execution_timeout: timedelta

task.on_failure_callback: Callable
SLAs in Airflow

(expected time of completion)

sort of works... with a lot of confusion... and with a lot of flaws

if start_date + sla < timezone.utcnow():
    sla_missed = True

Using SLAs causes DagFileProcessorManager timeouts and prevents deleted dags from being recreated

#15596 by argbiss was closed on Mar 16

SlaMiss Records Never Created for Packaged DAGs

#33410 opened 2 weeks ago by tseruga 1 of 2 tasks

Are SLAs usable? Are others using them?

notatalishaw-gts asked on Nov 9, 2022 in Q&A - Unanswered

TechAtBloomberg.com

© 2023 Bloomberg Finance L.P. All rights reserved.
DAG or Task-level Feature?

Callback defined at DAG level, but evaluated for each task

```python
@dag(
    schedule="*/2 * * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    sla_miss_callback=sla_callback,
    default_args={"email": "email@example.com"},
)
def example_sla_dag():
    @task(sla=datetime.timedelta(seconds=10))
    def sleep_20():
        """Sleep for 20 seconds""
        time.sleep(20)

    @task
    def sleep_30():
        """Sleep for 30 seconds""
        time.sleep(30)

    sleep_20() >> sleep_30()

example_dag = example_sla_dag()
```


Callback defined at DAG level, but evaluated for each task
Different Function Signature from Other Callbacks

**Others**

```python
def task_failure_alert(context):
    print(f"Task has failed, "
          "ti_key_str: {context['task_instance_key_str']}"
    )

def dag_success_alert(context):
    print(f"DAG has succeeded, "
          "run_id: {context['run_id']}"
    )
```

**SLA**

```python
def sla_callback(
    dag,
    task_list,
    blocking_task_list,
    slas,
    blocking_tis
):

    print("
The callback arguments are: ",
    {
        "dag": dag,
        "task_list": task_list,
        "blocking_task_list": blocking_task_list,
        "slas": slas,
        "blocking_tis": blocking_tis,
    },
)
```


Custom SLAs

malthe commented on Sep 30, 2021

Seems like this could use the new trigger service. Essentially, it is like branching out and having a suspended task with a trigger that activates at the deadline.

malthe commented on Oct 4, 2021

@yuqian90 what I'm referring to are the new deferrable operators.

There's a framework in there which allows us to set up future actions such as reacting to a “missed deadline”. It might need a little reworking in order to implement SLAs but I think it’s pretty close since you could also just branch out and use the new DateTimeSensorAsync.

Note that this framework is available only from Airflow 2.2 onwards.
Custom Operator: SLAMonitor

```python
from datetime import timedelta
from typing import Callable
from airflow.models.baseoperator import BaseOperator
from airflow.triggers.temporal import DateTimeTrigger
from airflow.utils.state import TaskInstanceState

class SlaMonitor(BaseOperator):
    def __init__(self, sla: timedelta, target_task_id: str, callback: Callable, **kwargs):
        super().__init__(**kwargs)
        self.sla = sla
        self.target_task_id = target_task_id
    def execute(self, context):
        deadline = context['data_interval_end'] + self.sla
        self.defer(trigger=DateTimeTrigger(deadline), method_name='execute_complete')
    def execute_complete(self, context, event=None):
        ti = context['dagrun'].get_task_instance(self.target_task_id)
        if ti and ti.state == TaskInstanceState.SUCCESS:
            self.log.info('SUCCEEDED')
        else:
            self.log.error('Sla was just missed!!')
            self.callback(context)
```

Sensor A ➔ Task B ➔ Task C ➔ Task D

Monitors SLA

Deferred for 1 hour

SlaMonitor A

Executes callback ➔ "Sla was just missed!!"
Custom Operator: SLAMonitor

[18:30:05 EDT] {task_command.py:410} INFO - Running <TaskInstance: market_close_pricing_v3.source_file_sensor_sla>
[18:30:06 EDT] {local_task_job_runner.py:222} INFO - Task exited with return code 100 (task deferral)
[18:30:06 EDT] {taskinstance.py:1415} INFO - Pausing task as DEFERRED
[19:29:37 EDT] {temporal.py:64} INFO - sleeping 1 second...
[19:29:47 EDT] {temporal.py:69} INFO - sleeping 1 second...
[19:30:03 EDT] {taskinstance.py:1306} INFO - Resuming after deferral
[19:30:03 EDT] {taskinstance.py:1327} INFO - Executing <Task(SlaMonitor): source_file_sensor_sla> on 2023-08-04 22:30:00+00:00
[19:31:03 EDT] {taskinstance.py:1345} INFO - Marking task as SUCCESS. dag_id=market_close_pricing_v3, task_id=source_file_sensor_sla ...
from datetime import timedelta
from typing import Callable

from airflow.models.baseoperator import BaseOperator
from airflow.triggers.temporal import DateTimeTrigger
from airflow.utils.state import TaskInstanceState

class SlaMonitor(BaseOperator):
    def __init__(self, sla: timedelta, target_task_id: str, callback: Callable, **kwargs):
        super().__init__(**kwargs)
        self.sla = sla
        self.target_task_id = target_task_id

    def execute(self, context):
        deadline = context['data_interval_end'] + self.sla
        self.defer(trigger=DateTimeTrigger(deadline), method_name='execute_complete')

    def execute_complete(self, context, event=None):
        ti = context['dagrun'].get_task_instance(self.target_task_id)
        if ti and ti.state == TaskInstanceState.SUCCESS:
            self.log.info('SUCCEEDED')
        else:
            self.log.error('Sla was just missed!!')
            self.callback(context)
            return

Sensor A → Task B → Task C → Task D → Teardown

SlaMonitor A → SlaMonitor D

Monitors SLA

Marks SUCCESS

sensor_callback
generic_callback
Questions?

https://www.linkedin.com/in/sung-yun-33451688
Future of SLAs?

DAG start

DAG SLA measured within a DagRun

Task 1

Task 1 SLA

Task 2

Task 2 SLA

Task 2 start

DAG SLA measured within a Task Instance

Task SLA measured within a Task Instance

DAG start

Task 1 SLA

Task 2 SLA

Other definitions of SLAs as Custom Operators:
Costly to manage in Airflow Core