# Dataclasses as Pipeline Definitions

Madison Swain -Bowden

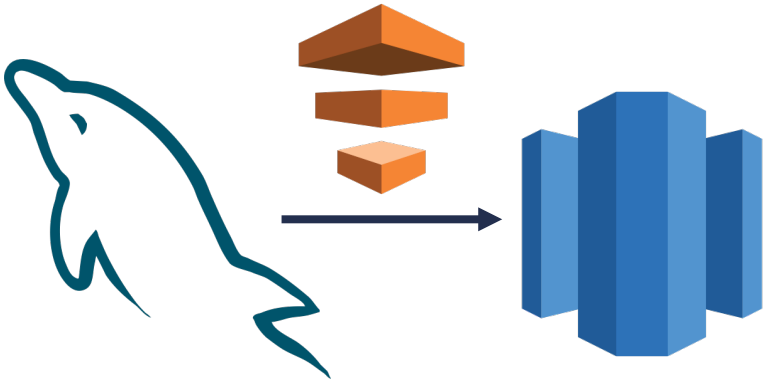
# Outline

OOKLA

- Problem space
- Dataclasses intro
- Dataclasses for Pipelines
- Validation
- Testing
- Alerts
- Documentation
- Airflow 2.0 performance improvements

# Our ETL Setup

- MySQL to Redshift replication
- Three pipeline "types"
  - Full - entire table is copied every interval
  - Incremental - only new records are copied
  - Rotating - large chunks of the table are replaced each interval, iterating through the entire table over time



- AWS Data Pipelines pain-points
  - Updating was cumbersome
  - Alerts were unintuitive
  - Configuration drift
- Perfect Airflow use-case!

# Dataclasses

OOKLA

- “NamedTuples with batteries included”
- New in python 3.7, PEP 557
- Features:
  - Methods auto-generated
  - Easy to provide defaults (including mutable ones)
  - Explicit typing
  - Attribute access (over key access)

# Dataclasses

```python
@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

```python
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0)
-> None:
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand

def __repr__(self):
    return f'InventoryItem(name={self.name!r}, unit_price=
{self.unit_price!r}, quantity_on_hand={self.quantity_on_hand!r})'

def __eq__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) ==
(other.name, other.unit_price, other.quantity_on_hand)
    return NotImplemented

def __ne__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) !=
(other.name, other.unit_price, other.quantity_on_hand)
    return NotImplemented

def __lt__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) <
(other.name, other.unit_price, other.quantity_on_hand)
    return NotImplemented

def __le__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) <=
(other.name, other.unit_price, other.quantity_on_hand)
    return NotImplemented

def __gt__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) >
(other.name, other.unit_price, other.quantity_on_hand)
    return NotImplemented

def __ge__(self, other):
    if other.__class__ is self.__class__:
        return (self.name, self.unit_price, self.quantity_on_hand) >=
(other.name, other.unit_price, other.quantity_on_hand)
    return NotImplemented
```

`__init__`, `__repr__`, `==`, `!=`, `>`, `>=`, `<`, `<=`

# Applying dataclasses

```python
@dataclass
class PipelineTemplate:
    """
    Base pipeline template class for defining new pipelines.
    """

    schema: str
    table: str

    source_schema: str = ""
    source_table: str = ""

    id_column: str = ""

    dag_id: str = ""
    conn_id: str = ""
    queue: str = "default"
    pool: str = ""
    debug_mode: bool = field(init=False)

    dataload_state: str = field(init=False)

    clusters: Sequence[str] = Clusters.CI
    schedule_interval: Optional[str] = None
    priority_weight: int = 0

    columns: Sequence[str] = ("*",)
    truncate_columns: bool = False
    accept_inv_chars: Union[bool, str] = False

    source: str = field(init=False)
    target: str = field(init=False)

    pipeline_type: str = "base"
    tags: Sequence[str] = ()
```

- JSON in UI -> pure Python in git
- Subclass by pipeline type
- Logic directly in class definition

```python
SpeedtestnetUserAgents = IncrementalPipelineTemplate(
    schema="speedtestnet",
    source_schema="web",
    table="user_agents",
    schedule_interval="0 */4 * * *",
    id_column="user_agent_id",
)
SpeedtestnetResults = IncrementalPipelineTemplate(
    schema="speedtestnet",
    table="results",
    schedule_interval="@hourly",
    id_column="result_id",
    select_limit=10_000_000,
    columns=[
        "result_id",
        "server_id",
        "result_date",
        "ip_address",
        "user_agent_id",
        "CASE WHEN download_kbps > 2147483647 THEN 0 ELSE download_kbps END AS download_kbps",
        "CASE WHEN upload_kbps > 2147483647 THEN 0 ELSE upload_kbps END AS upload_kbps",
        "latency",
        "test_type",
        "server_owner_id",
        "hash_key_id",
        "is_spoofed",
        "inet6_ntoa(ip_packed) as ip_packed",
    ],
)
```

# Applying dataclasses (cont'd)

```python
def __post_init__(self):
    self.source_schema = self.source_schema or self.schema
    self.source_table = self.source_table or self.table
    self.source = f"{self.source_schema}.{self.source_table}"
    self.dag_id = (
        self.dag_id
        or f"Pipeline_{self.schema}__{self.table}__{self.pipeline_type.upper()}"
    )

    # Use a set to combine these since module/schema may be the same
    self.tags = tuple({self.pipeline_type, self.schema, *self.tags})

    # Use the following criteria to determine if the table data should be
    # directed to a "debug" table rather than the production table:
    # - Global debug flag (on)
    # - Per-pipeline debug flag (on)
    # - Per-pipeline production flag (off)
    # If any of the above criteria are met, the table will be in debug mode
    pipeline_debug = self._get_setting("PIPELINE_DEBUG", default=False)

    pipeline_prod_enabled = self._get_setting("PROD_ENABLE", default=False)

    self.debug_mode = (GLOBAL_DEBUG or pipeline_debug) and not pipeline_prod_enabled

    if not self.debug_mode:
        self.target = f"{self.schema}.{self.table}"
        self.dataload_state = DATALOAD_STATE
    else:
        self.target = f"dev_dataeng.debug_{self.schema}__{self.table}"
        self.dataload_state = "dev_dataeng.airflow_dataload_state"
```

```python
### __post_init__ continued... ###
self.conn_id = self.conn_id or SCHEMA_TO_CONNECTION_MAPPING[self.source_schema]

# Pool is either overridden or a combination of the base connection ID
# and the queue. This allows separate pools for different worker queues so
# resource usage can be maximized.
self.pool = self.pool or self._get_setting(
    setting_name="POOL", default=f"{self.conn_id}_{self.queue}_read"
)

# If priority weight is overridden, use that. Otherwise, calculate it
# from the schedule interval
self.priority_weight = self.priority_weight or self._get_setting(
    setting_name="PRIORITY_WEIGHT",
    default=self._calculate_priority(self.schedule_interval),
)

# Also allow queue to be overridden
self.queue = self._get_setting("QUEUE", self.queue)
```

# Applying dataclasses (cont'd)

```python
@dataclass
class FullPipelineTemplate(PipelineTemplate):
    """
    Pipeline template class for defining full pipelines

    Column notes:
        - No additional columns beyond the base class are needed
    """

    pipeline_type: str = "full"

    def make_dag_docstring(self) -> str:
        source_type = "Redshift" if "redshift" in self.conn_id else "MySQL"
        return textwrap.dedent(
            f"""
            ## FULL: `{self.source}` -> `{self.target}`

            This pipeline copies all data from `{self.source}` in {source_type}
            to `{self.target}` in Redshift via a Parquet file in S3.

            **Clusters**: {', '.join(c.upper() for c in self.clusters)}

            **Pool**: `{self.pool}`

            **Queue**: `{self.queue}`

            **Priority Weight**: `{self.priority_weight}`
            """
        )
```

# Dataclasses + Dynamic DAGs

OOKLA

- 180 pipeline definitions → 3 definitions!
- Shared queries/ steps
- Shared alerting +all other Airflow goodies!

# Dataclasses + Dynamic DAGs

OOKLA

```python
mysql_to_s3 = SourceToS3ParquetOperator(
    task_id="mysql_to_s3",
    conn_id=pipeline.conn_id,
    pool=pipeline.pool,
    queue=pipeline.queue,
    params={
        "columns": pipeline.columns,
        "source": pipeline.source,
        "schema": pipeline.schema,
        "table": pipeline.table,
    },
    key=pipeline_utils.KEY,
    # Intentionally set this to an empty string, we'll provide the whole key
    dag_key_prefix="",
    query="queries/full_table_select.j2.sql",
    push_columns=True,
    source_schema=pipeline.source_schema,
    source_table=pipeline.source_table,
    target_schema=pipeline.schema,
    target_table=pipeline.table,
    target_conn_id=f"redshift_{pipeline.clusters[0]}",
    truncate_columns=pipeline.truncate_columns,
    accept_inv_chars=pipeline.accept_inv_chars,
)
```

```python
for name, query in pipeline.additional_limits.items():
    task_id = f"get_limit_{name}"
    task = SummaryOperator(
        task_id=task_id,
        conn_id=pipeline.conn_id,
        pool=pipeline.pool,
        single_value=True,
        process_func=pipeline_utils.df_single_value,
        params=dataclasses.asdict(pipeline),
        query=query,
    )

    limits[name] = task_id
    limit_tasks.append(task)
```

# Validation

Type validation

```
SpeedtestnetUserAgents = IncrementalPipelineTemplate(
    schema="speedtestnet",
    source_schema="web",
    table="user_agents",
    schedule_interval="0 */4 * * *",
    id_column="user_agent_id",
    clusters=12,
)
```

Expected type 'Sequence[str]', got 'int' instead

Pre-PR validation

▼ PIPELINE VALIDATION OUTPUT

**Pipeline:** `speedtestnet.servers`
**Type:** Full
**Status:** ✅

**MySQL:**

- [x] table exists
- [x] all referenced columns exist

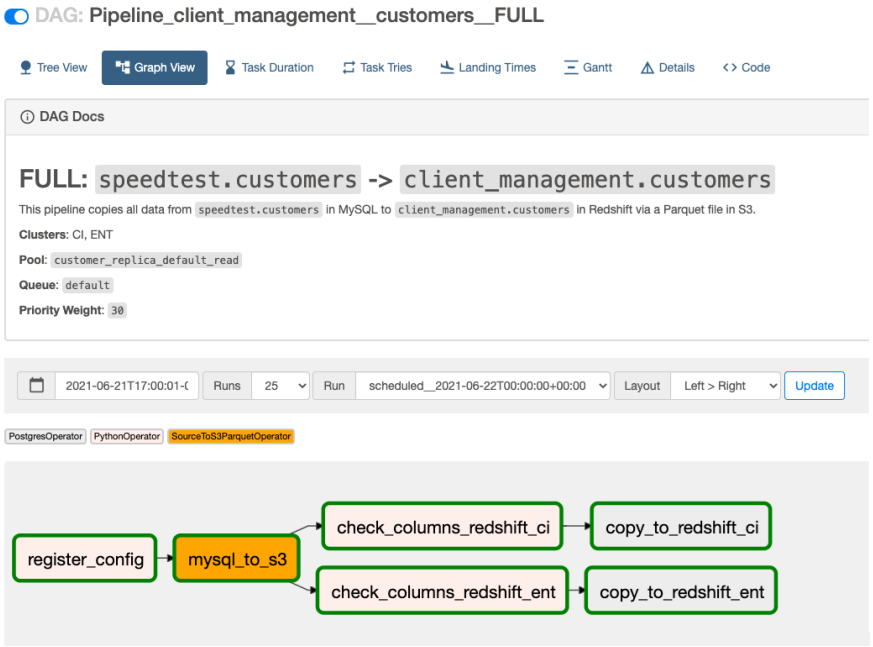
**Redshift CI:**

- [x] table exists
- [x] all referenced columns exist

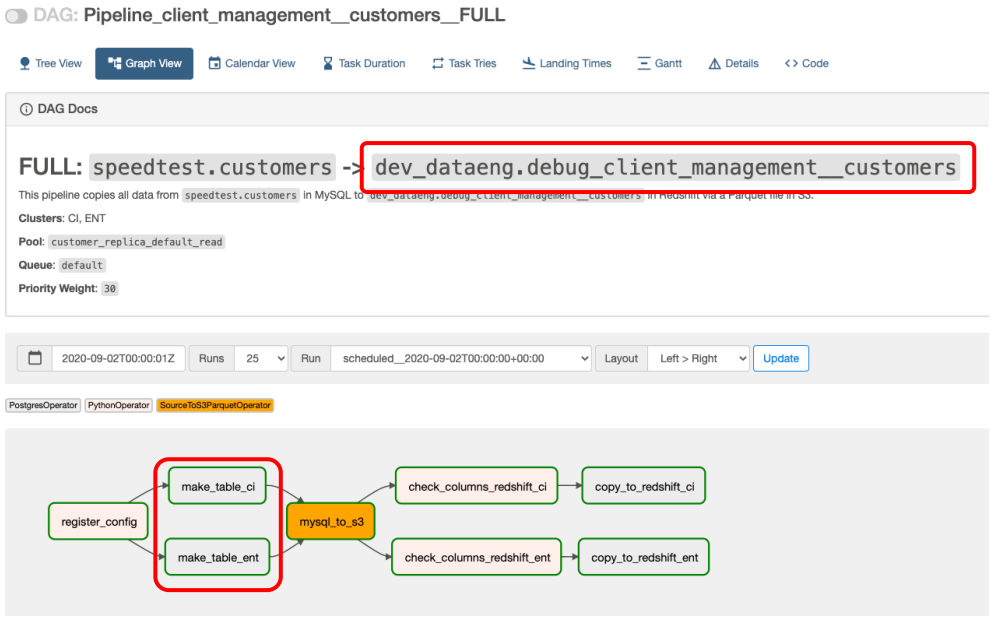
**Redshift ENT:**

- [x] table exists
- [x] all referenced columns exist

# Testing




Production




Development

# Alerts

OOKLA

Yesterday

**Opsgenie** APP 1:08 PM

#18265: Airflow pipeline failure

**DAG:** `Pipeline_speedtestnet__servers__FULL`

**Task:** `check_columns_redshift_ci`

**Execution Date:** 2021-06-22T16:00:00Z

**Exception:** Column mismatch for data from speedtestnet.servers on connection redshift_ci: Extra columns in destination table but not in return result: ['forceping_isp_id']

**Exception Type:** `airflow.exceptions.AirflowException`

**Log:**

Log Link

==Pipeline details==

**Source:** `speedtestnet.servers`

**Target:** `speedtestnet.servers`

**Conn ID:** `stnet_replica`

**S3 Copy Options:** `None`

Show less

**Priority**
P2

**Tags**
Airflow-Pipelines

**Routed Teams**
Speedtest_DataEng

Madison Swain-Bowden acknowledged alert #18265 "Airflow pipeline failure"

# Documentation

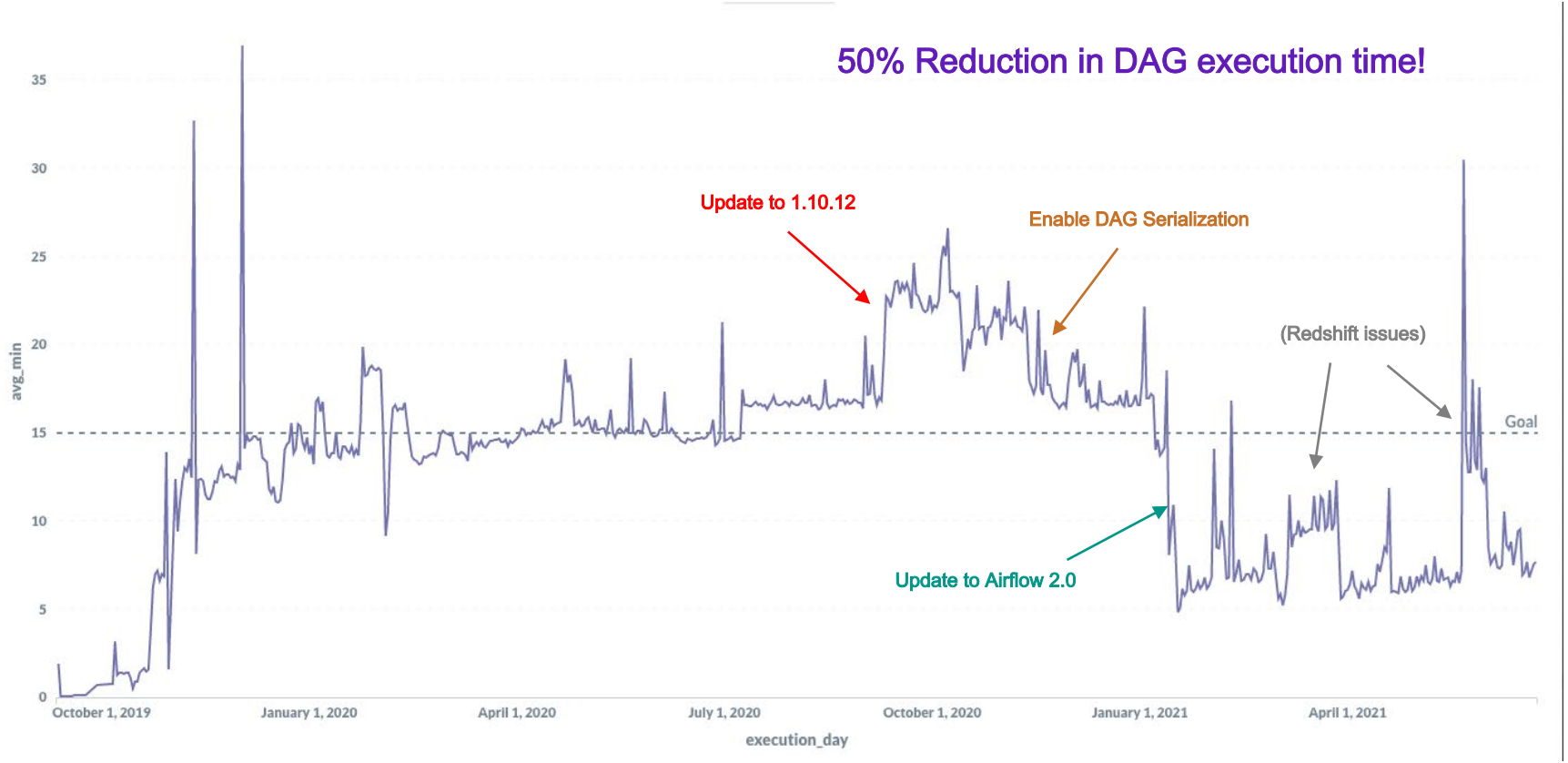
Pipelines:

- Full Pipelines
- Chunked Pipelines
- Incremental Pipelines
- Rotating Pipelines

## Full Pipelines

| Target Table | Source Table | Schedule Interval | Clusters |
|---|---|---|---|
| client_management.customers | speedtest.customers | @daily | CI, ENT |
| [redacted] | [redacted] | @weekly | CI |
| | | @weekly | CI |
| | | @daily | CI |
| | | @daily | CI |
| | | @daily | CI |
| | --- | 0 */4 * * * | CI |
| | --- | @daily | CI |
| | --- | @daily | CI |
| | --- | @daily | ENT |
| | --- | 0 */4 * * * | CI, BG |
| | --- | 0 */4 * * * | CI, BG |
| | --- | 0 */4 * * * | CI, BG |
| | --- | @weekly | CI, BG |
| | --- | @weekly | CI, BG |
| | --- | @weekly | CI |

# Airflow 2.0 Upgrade



# Takeaways

OOKLA

- Dataclasses can be used for pipeline definitions (in lieu of YAML/ JSON/ etc)
- Using native Python objects can make it easier to maintain pipelines (validation, testing, alerts, etc.)
- Upgrade to Airflow 2.0!

# Thank you!