Airflow in an On-premise Data Mesh Setup
Speaker

Jorrick Sleijster
Data Engineer
Data Platform Development
Tennis & Padel tennis
Adyen
We provide a single payment platform globally to accept payments and grow revenue online, on mobile, and at the point of sale
The Workflow of the Big Data Platform

**Fully on-premise**
We have all servers in-house. We manage everything from hardware to software. Different clusters per env: beta, test and live.

**Weekly releases**
Each week we start with a new release on beta. Two days later deployed to test, and finally a week later to live.

**Teams / Streams**
We have 20+ streams and 100+ data scientists on the big data platform. Streams are responsible for their own DAGs.
Airflow Cluster Setup at Adyen

- Airflow 2.2
- Celery workers
- Yarn Queue
- Postgres DB

**adyen extensions**
Models, views, database listeners
Why Data Mesh

- What is a Data Mesh?
  - Monolithic data infrastructure
  - Distributed domains
  - Domain responsible for their own ETLs & data

- Product teams(domains) with a clear focus
  - Front-enders, back-enders, data scientists

- No central ETL team

- Self service
Data Mesh Teams

- Product teams
- Tooling teams
- Infrastructure teams
Creating a (new) Data Pipeline

- Create an Airflow DAG
- Create the ETL code
Creating a (new) Data Pipeline

- Create an Airflow DAG
- Create the ETL code

- We need to:
  - Define the schema
  - Have enough resources for the ETL
  - Possibly handle updates of the schema
  - Retention, we don’t have infinite storage
  - Remove the table
  - Don’t want to (accidentally?) modify other teams resources.
Table Schemas
How can we enable schema evolution?

Tuning Resources
How can we handle data outgrowing the predefined ETL resources?

Retention Period
How can we save HDFS from being filled with data we don’t need anymore?

User Permissions
How can we give users access to only the parts that they need?
Table Schemas

- We are managing an ETL pipeline
- How & where do we define the schema?
- How do we update the schema?
- How do we delete tables we no longer need?
Table Schemas Library

- Each domain has their own database.
- Each database has multiple scopes:
  - pii - Requires white-listed access.
  - private - Access by domain members only.
  - public - Access by everyone.
- Each database scope can have multiple tables.
- Each table has at least one change file.
- Change types: new, (in)compatible, remove

Tool to create schemas from a Spark DataFrame.
Process Flow

1. Update the schema file (.ddl)
2. Introduce the schema change file (.ddl)
3. Create an MR (merge request)
4. Teammate reviews and approves MR
5. MR is merged
Process Flow

1. Update the schema file (.ddl)
2. Introduce the schema change file (.ddl)
3. Create an MR (merge request)
4. Teammate reviews and approves MR
5. MR is merged
6. Release a new version on the cluster
7. Duty rolls out all schema change files for release
8. (Optional) User can also perform ad-hoc table changes like redefining the table with the latest schema.
not-so-live demo
### Roll out Table Changes

Please select the changes which you want to roll out from the table below.

Filter on release version  
Select a release  
Filter on other properties  
Filter changes  
Reset selection  
Roll Out Changes

<table>
<thead>
<tr>
<th>RC</th>
<th>Change Type</th>
<th>Database</th>
<th>Table Privacy</th>
<th>Table Name</th>
<th>Requested by</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1_12</td>
<td>REMOVE</td>
<td>monitoring</td>
<td>public</td>
<td>api_performance</td>
<td>bart</td>
<td></td>
</tr>
<tr>
<td>V1_12</td>
<td>NEW</td>
<td>analytics,core</td>
<td>public</td>
<td>tablemetadata</td>
<td>merge</td>
<td></td>
</tr>
<tr>
<td>V1_12</td>
<td>INCOMPATIBLE</td>
<td>cdi</td>
<td>pii</td>
<td>pii_table</td>
<td>lisa</td>
<td></td>
</tr>
<tr>
<td>V1_12</td>
<td>COMPATIBLE</td>
<td>analytics,core</td>
<td>public</td>
<td>sparkpushedfilters</td>
<td>merge</td>
<td></td>
</tr>
</tbody>
</table>
Tuning Resources

• As we are on-premise, we have finite resources
• We hardcode the resources in our code base
• Now imagine:
  • A DAG that has been running for 100 days
  • On day 101 you get error code 143
  • What do you do?
The Usual Workflow

- You google the issue

- Solution: Set a higher Spark driver and/or worker memory limit
The Usual Workflow

- You google the issue
- Solution: Set a higher Spark driver and/or worker memory limit
- But we hardcoded our resources in the code base, so that would require:
  - Create an MR
  - Request for an approval
  - Creating an official patch request
  - Request for an approval
  - Patching
- How can we prevent patching?
not-so-live demo
Retirement Period

- As we are on-premise, we have finite storage.
- If we are close to storage limits, there are two options:
  - Buy more servers
  - Get rid of some of our data
- We should prevent getting close to the limit
Governance library

- Each stream has a Stream file in this library.
- They define all their tables there and their corresponding retention period.
- Disclaimer: This is much more complex when you work with tables that are not partitioned by date.

```python
class AnalyticsCore(Stream):
    def get_all_tables(self) -> List[HDFSTable]:
        return [
            HDFSTable(
                HiveDatabase.ANALYTICS_CORE, "parquetfilesize", RetentionPeriod.THIRD_MONTHS
            ),
            HDFSTable(
                HiveDatabase.ANALYTICS_CORE, "parquetmetadatainfo", RetentionPeriod.ONE_MONTH
            ),
            HDFSTable(
                HiveDatabase.ANALYTICS_CORE, "sparkpushedfilters", RetentionPeriod.ONE_YEAR
            ),
            HDFSTable(
                HiveDatabase.ANALYTICS_CORE, "tablemetadata", RetentionPeriod.THREE_YEARS
            ),
        ]
```
<table>
<thead>
<tr>
<th>Stream</th>
<th>Hive Database</th>
<th>Table Name</th>
<th>Planned Removal Date</th>
<th>Status</th>
<th>Partitions Start Date</th>
<th>Partitions End Date</th>
<th>Forbid Undo Dag</th>
</tr>
</thead>
<tbody>
<tr>
<td>analytics_core</td>
<td>analytics_core</td>
<td>sparkpushedfilters</td>
<td>2022-05-18</td>
<td>SUCCEEDED</td>
<td>2021-05-11</td>
<td>2021-05-17</td>
<td>False</td>
</tr>
<tr>
<td>analytics_core</td>
<td>analytics_core</td>
<td>sparkpushedfilters</td>
<td>2022-05-25</td>
<td>STAGED</td>
<td>2021-05-18</td>
<td>2021-05-24</td>
<td>False</td>
</tr>
<tr>
<td>analytics_core</td>
<td>analytics_core</td>
<td>tablemetadata</td>
<td>2022-05-18</td>
<td>SUCCEEDED</td>
<td>2022-05-05</td>
<td>2022-05-11</td>
<td>False</td>
</tr>
<tr>
<td>analytics_core</td>
<td>analytics_core</td>
<td>tablemetadata</td>
<td>2022-05-25</td>
<td>STAGED</td>
<td>2022-05-12</td>
<td>2022-05-18</td>
<td>False</td>
</tr>
<tr>
<td>Table Name</td>
<td>Planned Removal Date</td>
<td>Status</td>
<td>Partitions Start Date</td>
<td>Partitions End Date</td>
<td>Forbid Undo Dag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------</td>
<td>------------</td>
<td>-----------------------</td>
<td>---------------------</td>
<td>-----------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sparkpushedfilters</td>
<td>2022-05-18</td>
<td>SUCCEEDED</td>
<td>2021-05-11</td>
<td>2021-05-17</td>
<td>False</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analytics_core</td>
<td>2022-05-25</td>
<td>STAGED</td>
<td>2021-05-18</td>
<td>2021-05-24</td>
<td>False</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analytics_core</td>
<td>2022-05-25</td>
<td>STAGED</td>
<td>2022-05-12</td>
<td>2022-05-18</td>
<td>False</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
User Permissions

- We want to empower the streams so it truly becomes self-service
- Yet, they should only be able to modify their own resources.
User Groups

• We always have one on-duty Admin

• For teams we have two access groups:
  • Stream admin
  • Standard user

• Team admins are able to:
  • Manage all the DAGs of the teams
  • Manage all the tables of their database
  • Manage the Spark resources of their tasks
Flask Appbuilder Permissions

- Why are you talking about Flask Appbuilder?

- In terms of permissions on these views:
  - Each DAG has two POVs:
    - my_dag.can_read
    - my_dag.can_edit
  - Each function in a view has a POV:
    - spark_configuration.can_add
    - spark_configuration.can_edit

- For roles we create:
  - One basic role for all users
  - One role for each stream admin group
We Implemented

- On each release, all permissions per role are updated.
- Created decorators to indicate who has access.
- Validators that raise exceptions when someone tries to modify a task that it has no access on.
- Modified our views to only show runnable parts.
not-so-live demo
### List Users

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>User Name</th>
<th>Email</th>
<th>Is Active?</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>ac</td>
<td>ac</td>
<td><a href="mailto:ac@adyen.com">ac@adyen.com</a></td>
<td>True</td>
<td>[User, stream-analytics-core]</td>
</tr>
<tr>
<td>adyen</td>
<td>adyen</td>
<td>adyen</td>
<td>adyen@adyen</td>
<td>True</td>
<td>[Admin]</td>
</tr>
</tbody>
</table>

Record Count: 2
Mapping Users to Roles

• Now that we have the roles defined, this is the final thing left to do.

• Security team requires us to use LDAP for managing the roles.

• Service that runs each hour.
Wrap up

Table Schemas
How one can enable schema evolution over time and abstract away complexities

Tuning Resources
Preventing patches for data overflowing its ETL resource limits

Retention Period
Work with limited storage

User Permissions
Each domain has their own database.
Before I forget…

NL - Amsterdam
Head office

US - Chicago
Tech hub

Spain - Madrid
Tech hub
Thank you
Slides for questions
Code Example: How we Define Permissions

class ConfigurationView(AdyenProtectedModelView):
    datamodel = CustomSQLAInterface(ConfigModel)

    extra_allowed_user_functions = ["list"]
    extra_allowed_stream_admin_functions = ["add", "delete", "edit"]

    @action("export", "Export the Spark Configurations", ",", single=False)
    @visible_to_users(is_fab_action=True, action_name="export")
    def export_configs(self, configs: List[ConfigModel]) -> Response:
        pass

    @action("drop", "Drop Configurations completely", "You sure?", single=False)
    @visible_to_stream_admins(is_fab_action=True, action_name="drop")
    def drop_completely(self, configs: List[ConfigModel]) -> Response:
        pass
Code example: Implementation of Decorators

```python
def visible_to_stream_admins(is_fab_action: bool, action_name: Optional[str] = None):
    def inner_function(func: Callable):
        @wraps(func)
        def wrapper(*args: Any, **kwargs: Any) -> Any:
            return func(*args, **kwargs)

        if is_fab_action and (action_name is None or not any(action_name)):
            raise ValueError(f"If {is_fab_action=} you must also set {action_name=}.")

        wrapper.visible_to_stream_admins = True
        wrapper.is_fab_action = is_fab_action
        wrapper.fab_action_name = action_name

        return wrapper

    return inner_function
```
Operations on Tables

For any operation you wish to perform on a table, you can categorise them in one (or multiple) of these 5 operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>NEW</th>
<th>COMPATIBLE</th>
<th>INCOMPATIBLE</th>
<th>REMOVE</th>
<th>MOVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defines hive table definition</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Delete hive table definition</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Delete table data</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>MSCK Repair</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>