Customizing Xcom for data sharing between tasks

Vikram Koka and Ephraim Anierobi
Introductions

Vikram Koka
Apache Airflow Committer
Senior Vice President Engineering at Astronomer
Silicon Valley

Ephraim Anierobi
Apache Airflow Committer
Software Engineer, Open Source at Astronomer
Nigeria
Xcom in a nutshell

Task 1

xcom_data
(metadatabase)

Task 2
Xcom Overview

Cross communication between tasks
- Pass parameters from one task to another
- Supports multiple parameters
- Identified by key
- Intended for use within a single DAG

Usage:
- “push” and “pull”

Uses the Airflow metadatabase (Postgres / MySQL)

```python
xcom_push(
    key = 'return_value',
    value = 'my value'
)

value = xcom_pull(
    task_ids='pushing_task',
    key='return_value'
)
```
Xcom with TaskFlow API

Greater Abstraction

- Return values implicitly use xcom
- Focused on the most common pattern
- Supports python native types including `dict`

Pythonic functional use

```python
def extract:
    ...
    return order_data
    ...

order_data = extract()
order_summary = transform(order_data)
```
Xcom limitations

Data types

As it stands, only the following datatypes are supported in Airflow 2.0

- Python native: dict, list, tuple, str, int, long, float, True, False, None
- Future: Airflow supported objects such as numpy objects, datetime, date, etc
- For security, pickling is no longer recommended
Github issues

Sample questions / problems

- Unable to store xcom because of MySQL Blob type limitation 65,535
- Data too long when pushing to XCOM
- Raise do_xcom_push size limit
- Lambda to transform response before xcom push
- Provide shared storage between task via pluggable storage providers akin to S3 remote logging
Custom XCom backends

Persistence class
- Python class specified in config
- Read at Airflow start up,
  Class needs to be in Airflow path

Methods needed:
- serialize_value
- deserialize_value
  Used for storing and restoring data
- orm_deserialize_value
  Used to display XCom data in UI
Custom XCom for Local Execution

- Write / read local file system
- Essential for development
- Local Executor

Not for distributed deployments with celery and Kubernetes Executors
Xcom stored in local file system

Task 1

Custom Xcom backend class

xcom_index (metadatabase)

xcom_data (file system)

Task 2
Code walk through and demo
Airflow distributed execution
Airflow distributed execution - Kubernetes
Custom XCom for Distributed Execution

- Write / read cloud storage
- Accessible from any configured node
- Can be used with Celery and Kubernetes Executors

Higher latency, so could cause delays when used with short running tasks

More expensive than other options
Custom Xcom for Distributed Execution

- Write / read from Redis
- Accessible from any configured node
- Can be used with Celery and Kubernetes Executors
- Already part of the Airflow stack

Size limit of 512MB, so ideal for smaller dataset between short running tasks.

Another caveat that Redis keeps everything in memory.
Xcom stored in Redis or Cloud Storage

Task 1

Custom Xcom backend class

xcom_index (metadatabase)

Task 2

xcom_data
Redis or Cloud Storage
Code walk through and demo
Clutter

- Clean-up of old data in cloud storage or elsewhere
- As data gets larger, data cleanup becomes more important
- System performance can degrade
Clean-up DAG

- Maintenance DAG to clean-up old Xcom data
- Deletes data from metadatabase and external locations
- Not tied to DAG lifecycle - needs to be configured carefully
- Downside if trying to rerun old tasks
Code walk through and demo
def clean_fs_xcom(self, fs_xcom):
    for filename in glob.iglob("**/*.*"):
        print(f"Deleting file {filename}"),
        os.remove(filename)

def clean_redis_xcom(self):
    keys = self.redis().keys(pattern='*XCOM*')
    print(f"Redis keys found: {keys}"),
    self.redis().delete(keys)

def clean_db_xcom(self, session=None):
    with session:
        session.query(XCOM).delete()
Success

We have addressed the core questions raised

- Handling of non-native objects such as Dataframes
- Large data sets between tasks
- Leveraging cloud storage
- Maintenance and cleanup
Limitations

Not tied to DAG life cycle management

- Data sharing across DAGs is difficult
- Maintenance DAGs for clean-up is a kludge
- Should be cleanly handled by Airflow when DAG is done
Future: Top level data object in Airflow

- Result of DAGs from one team is data
- Can be used by DAGs from other teams
- Key for cross-DAG dependencies
- Availability can be used to trigger follow-on DAGs

Integrated with DAG life cycle management and with Event driven DAGs

Airflow Improvement Proposal upcoming
Jobs at Astronomer

We are hiring Airflowers all over the world!

https://careers.astronomer.io/

https://linkedin.com/vikramkoka

Contact us: We would love to hear from you!
Questions?