Modern ETL-ing

PyCon DE 2017 - Tamara Mendt

What to expect from this talk?

- What is ETL-ing and why it’s (still) relevant?
- How Airflow can help?
- What role does Spark play here?

… and a few Memes

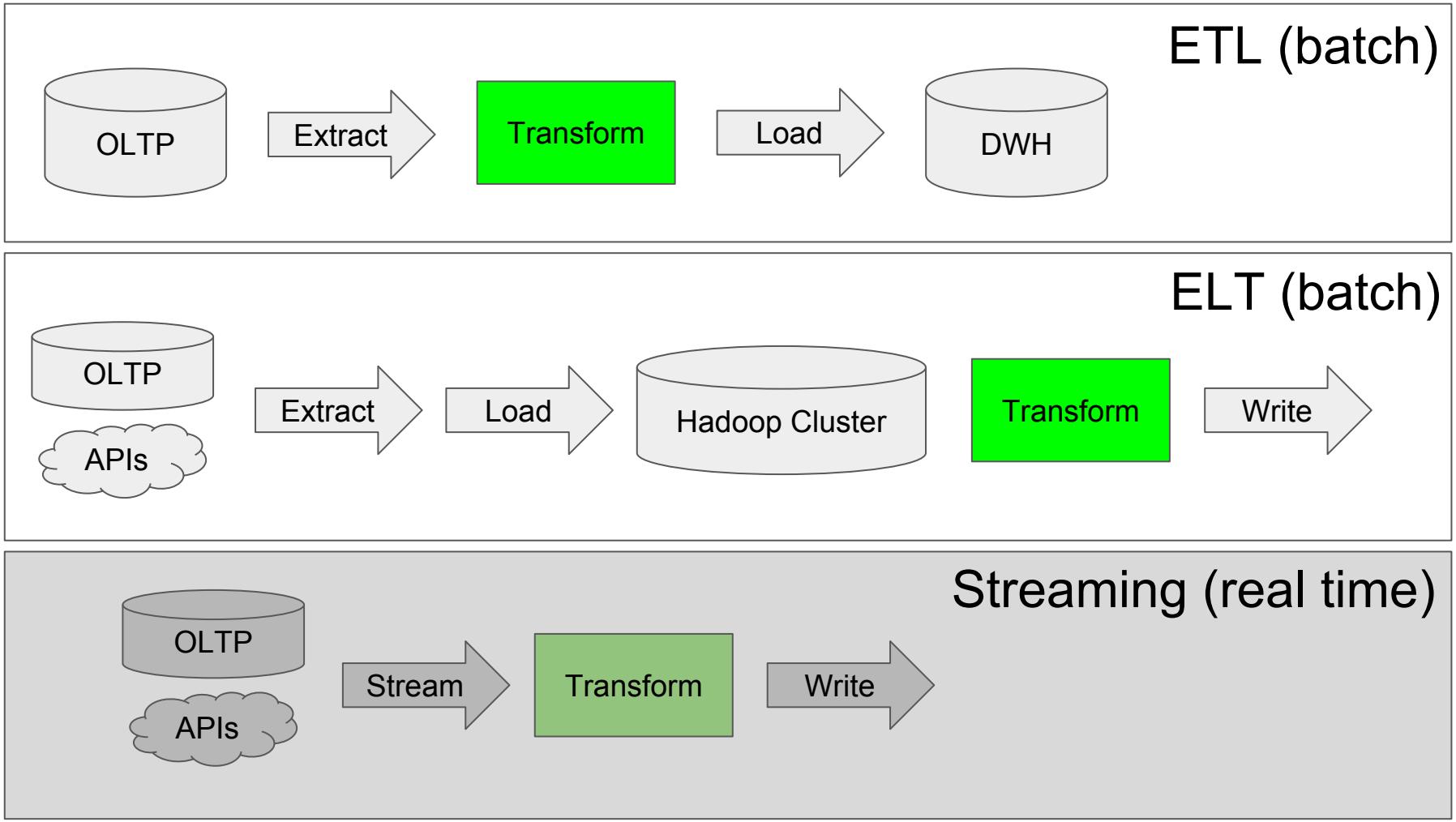
A little of my experience

- Worked at HelloFresh for 2 years as a Data Engineer
  - Built DWH up from scratch
  - Lots of ETLs running on a Production environment using Airflow

- Studied a very specific Master called “IT for Business Intelligence”
  - Specialized in database technologies and data warehousing

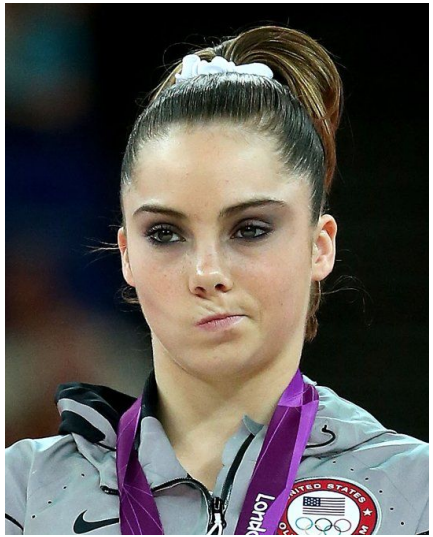
What the hell is ETL-ing?

- Stands for:
  - **EXTRACT, TRANSFORM, LOAD**
- Transforming data for analytical purposes
  - **Integrating** data from multiple sources
  - **Pre-aggregating** data to speed up queries
  - Computing features for input to ML models
- Usually implies **batch** processing


ETL (batch)
OLTP
Extract
Transform
Load
DWH
ELT (batch)
OLTP
APIs
Extract
Load
Hadoop Cluster
Transform
Write
Streaming (real time)
OLTP
APIs
Stream
Transform
Write

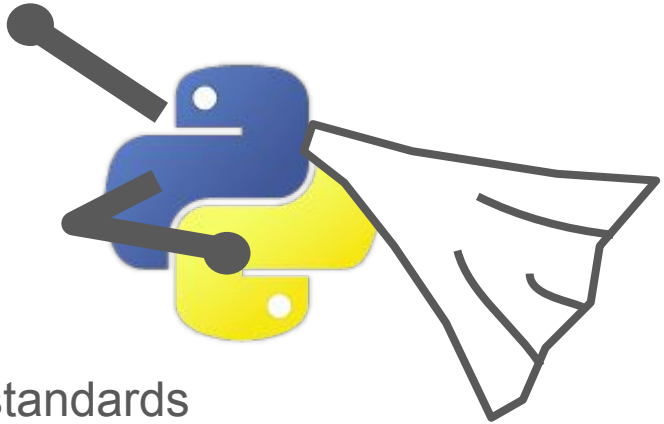
The problem with ETL tools?

- Mostly designed to deal with well structured data
- Biggest problem now is **variety** of sources and formats
- Mostly not open source
  - It's hard to see why operators are not behaving as you expect
  - Often expensive
  - Not easy to extend
- You're limited to the functionality provided by the tool



How about using Python instead?

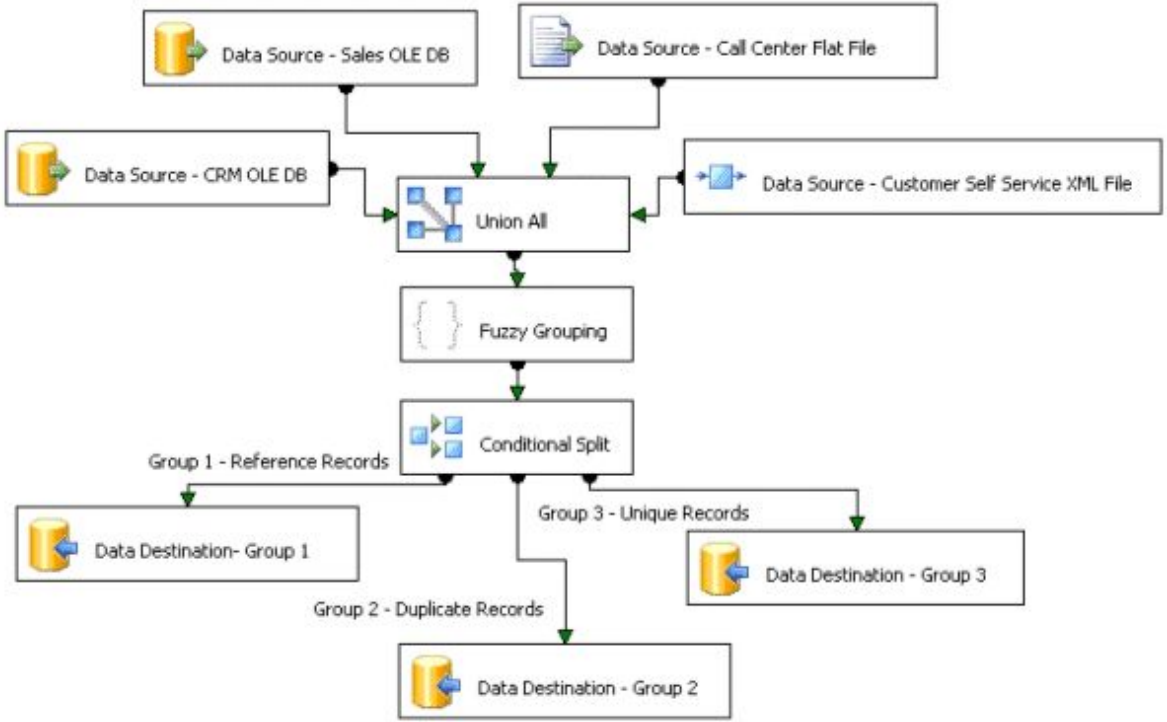
- It’s easy!
- Way more flexible, you can code whatever you like
- Easy to re-use logic and create meaningful abstractions
- You can test the logic!
  - Less risk of introducing breaking changes
- Versioning and collaboration
- Its free



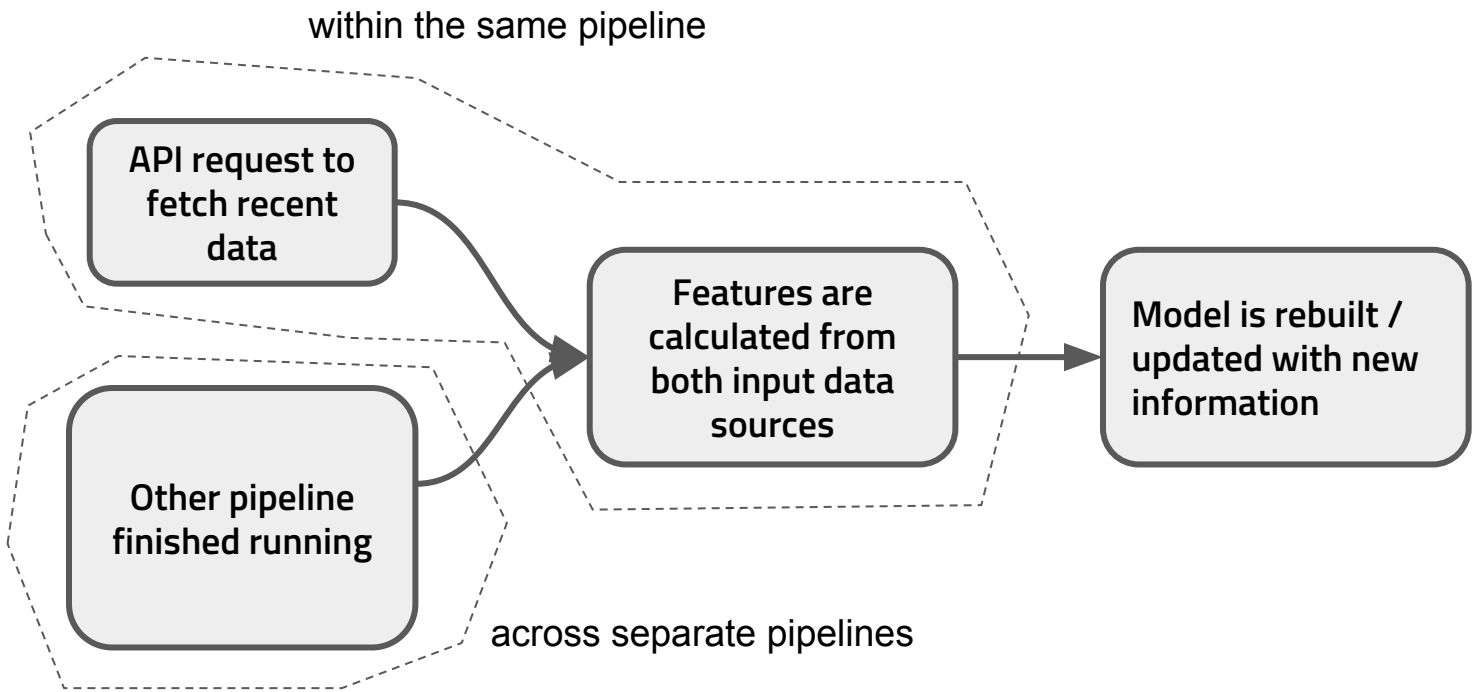
- Note: this is subject to organization culture and standards

You know, ETL tools have some nice features though

Explicit documentation, data lineage



Defining dependencies between tasks



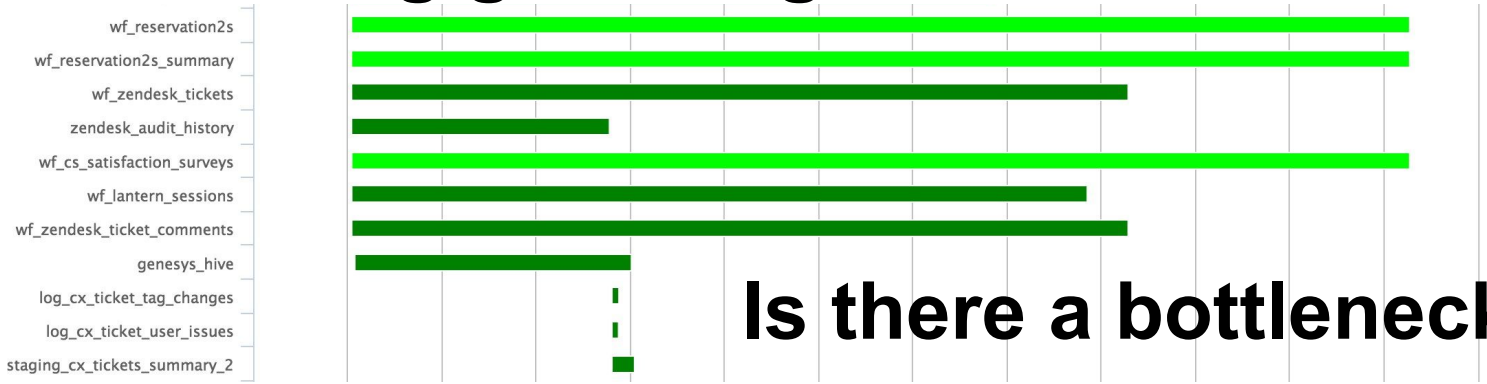
Centralized Scheduling (and configuration)

- In hand with defining dependencies
- Schedule automatic retries
- Configure data sources only once and reuse

Alerting and monitoring

		DAG	Schedule	Owner	Recent Statuses
	On	example_bash_operator	0 0 * * *	airflow	
	On	example_branch_operator	@daily	airflow	5
	On	example_http_operator	1 day, 0:00:00	airflow	1 1 4
	On	example_passing_params_via_test_command	*/1 * * * *	me	1 1
	On	example_python_operator	None	airflow	

Did something go wrong?



Is there a bottleneck?

That's what we have Airflow for

"Airflow is a platform to programmatically author, schedule and monitor workflows."

PLUS:

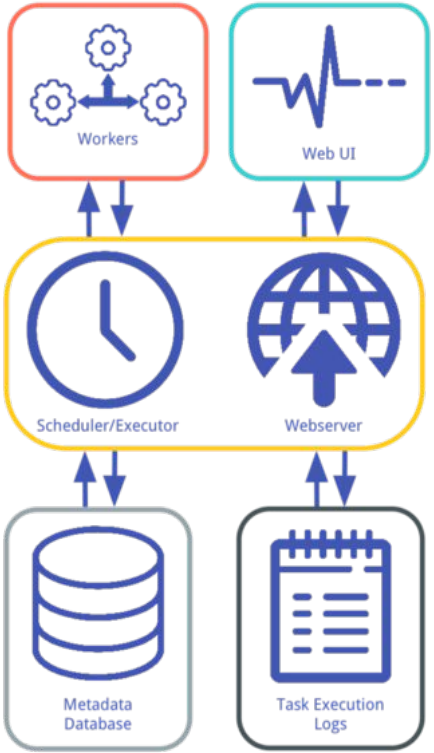
- It's Python!
  - Workflow (DAG) definitions
  - Execution engine
- It's open source
  - You can always see what's going on
  - You can extend it to your needs
  - Apache incubator
  - Good community
- Dynamic pipeline and task creation

```
from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.bash_operator import BashOperator

default_args = {
  'owner': 'airflow',
  'depends_on_past': True,
}

dag = DAG("my_first_dag",
          schedule_interval='*10 * * * *',
          default_args=default_args,
          start_date=datetime(2017, 10, 25))

task_1 = DummyOperator(task_id='dummy_task', dag=dag)
task_2 = BashOperator(task_id='bash_task', dag=dag,
                      bash_command="echo 'HelloWorld' ")

task_1.set_downstream(task_2)
```

Airflow basics: Architecture

- Sequential Executor
- Local Executor
- Celery Executor

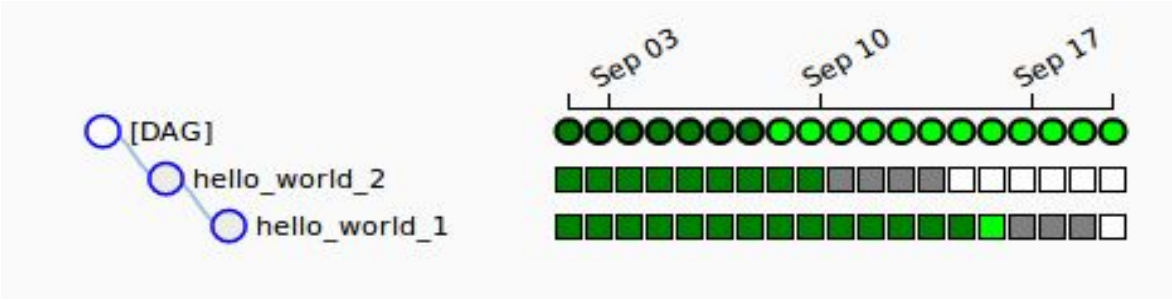
- DAGs are identified by their ids



source: https://medium.com/@dustinstansbury/understanding-apache-airflows-key-concepts-a96efed52b1a

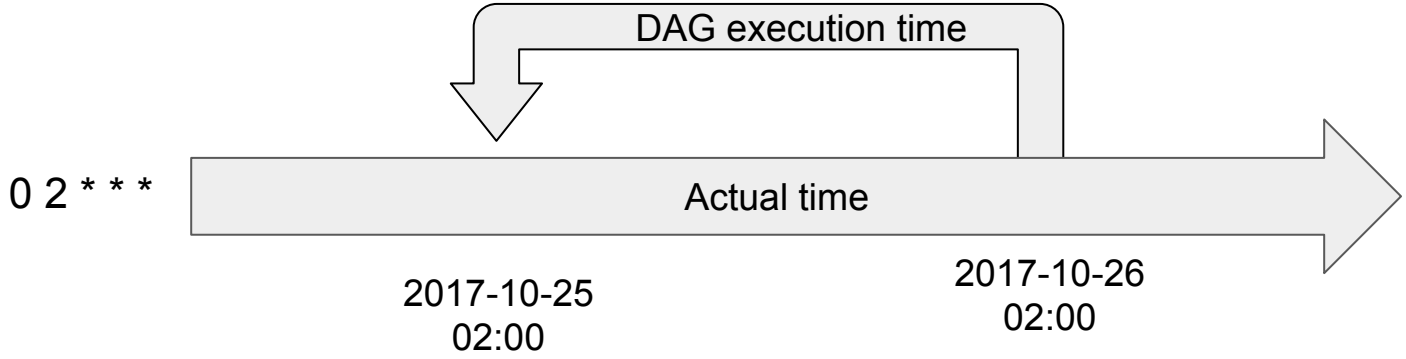
Airflow basics: Scheduler

- Each DAG has a **schedule** and **start date**
  - 0 */5 * * *→every 5 hours
  - datetime(2017, 10, 15) → since this date
- A DAGRun is created for every schedule period between start and now
  - DAG + execution time
  - Task instances are created for every DagRun
  - In order for tasks to be backfilled, the status of the DagRun must be cleared as well.
  - Be aware of the ***depends_on_past*** property



Airflow basics: Scheduler

- Preventing the Scheduler from “catching up” or “backfilling” past executions
  - **LatestOperator**: will avoid all downstream tasks that are not the latest execution from running
  - Set DAG **catchup** property to False
- The DagRun is triggered when the period it covers has ended!
  - The intuition behind this is that the DAG processes the data generated in the previous period



Incremental vs. Historical

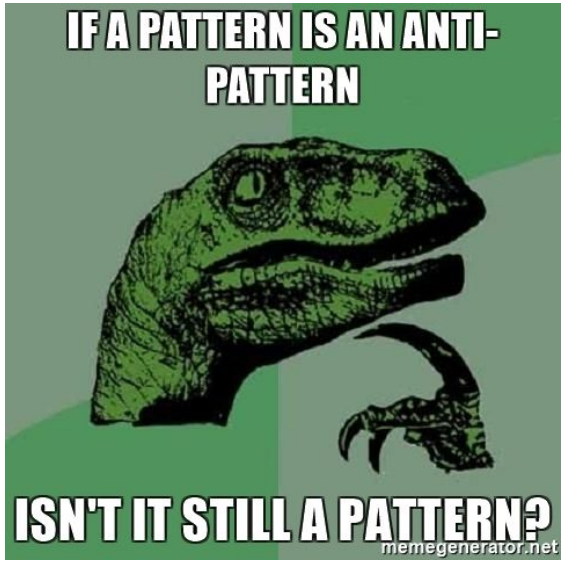
- Historical, e.g. first run, errors corrected in source, change in logic, ..
- Data should be **immutable** and all transformations should be **reproducible**
- Use jinja templating {{ ds }} {{ macros.ds_add(ds, 7) }}
- Two very similar DAGs parameterized differently

	On	my_dag_historical	@once
	On	my_dag_weekly	0 5 * * Mon

Passing data from one task to another

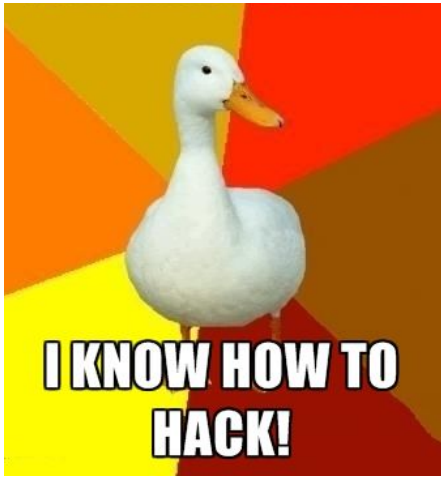
- Airflow anti-pattern
  - **NO PIPELINING BETWEEN TASKS**
  - Airflow is designed to run on multiple workers
  - Each task must read from and write to systems accessible to all workers
  - Explicit documentation is not entirely true
- Xcom is used for sharing small bits of information
  - Tasks can push or pull variables
  - Some operators include xcom_push as a property
  - In others you must explicitly push the variable and pull using jinja templating

```
{{ task_instance.xcom_pull(task_ids='my_task') }}
```



Organizing DAG definitions

This will allow you to load DAGs from other directories




```
from airflow.models import DagBag

for directory in AIRFLOW_DAG_DIRECTORIES:
    dag_bag = DagBag(directory)

    if dag_bag:
        for dag_id, dag in dag_bag.dags.items():
            globals()[dag_id] = dag
```

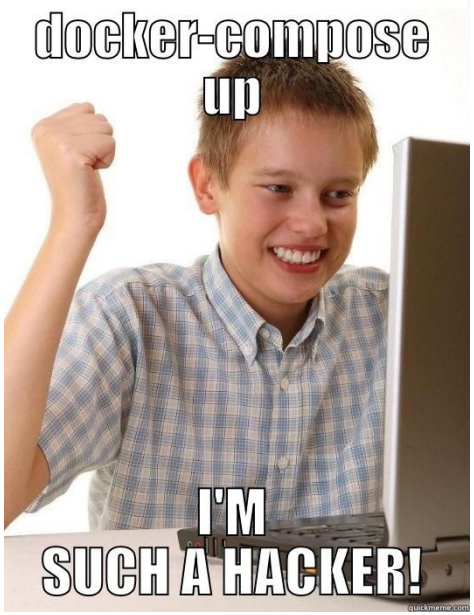
One important Gotcha: Airflow will only find DAGs if the files containing the DAG definition import an airflow module (even if it's an unused import)

Using the command line

- Most of the time more convenient than using the GUI
  - airflow clear <dag-id>
  - airflow backfill -s <start-date> -e <end-date> -m <dag-id>
  - airflow test <dag-id> <task-id> <execution-date>

How can you start?

- Easiest way to start is using Docker
  - https://github.com/puckel/docker-airflow
  - docker-compose -f docker-compose-LocalExecutor.yml up -d
  - Write your dag definitions in the /dags directory
- For deployment we use Ansible
  - Many ansible roles out there
    - https://github.com/LREN-CHUV/ansible-airflow
  - webserver and scheduler run by Supervisor



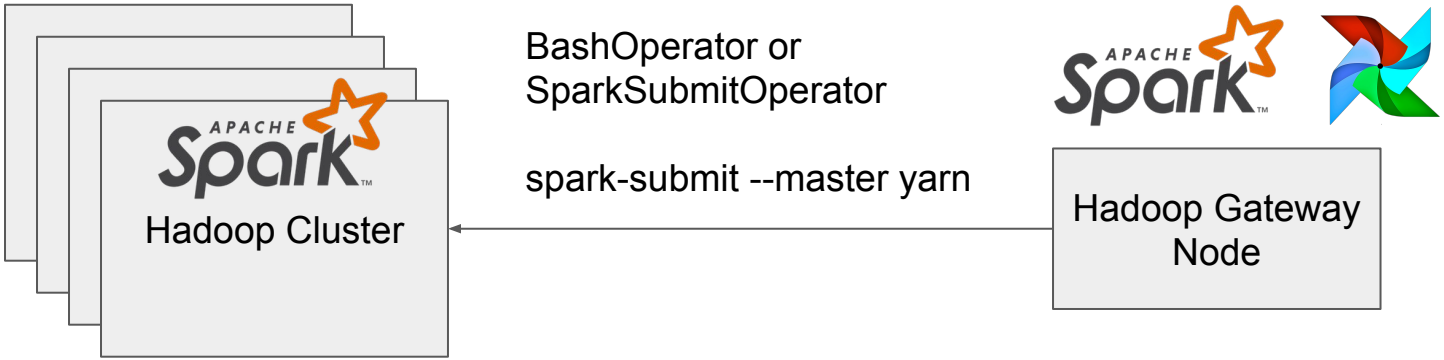

Where does
Spark fit in?

What's Spark (very briefly)?

- Open source framework for distributed computation
- Most commonly used together with Hadoop YARN (Resource Manager)
- Written in JAVA, but has a very nice Python API
- Very nice SQL API

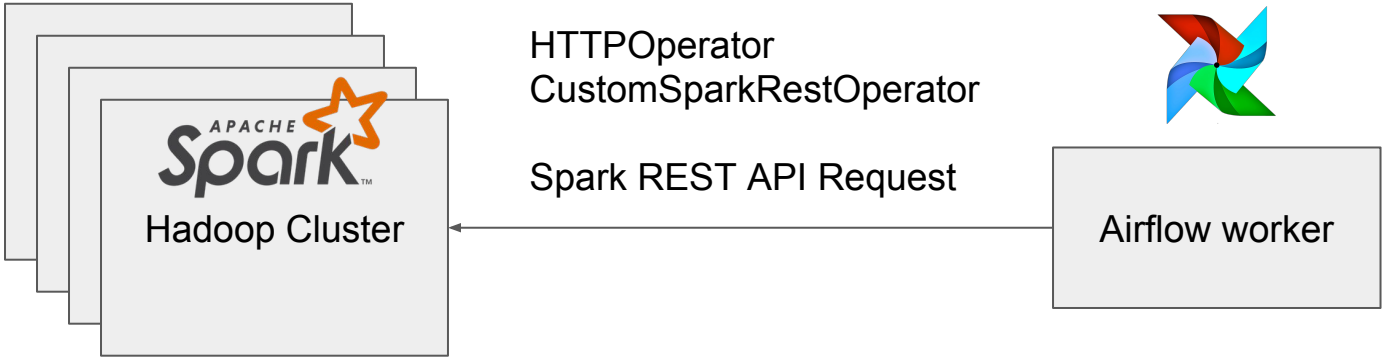
```
sc = SparkContext(appName="My App")
sqlc = SQLContext(sc)

data_frame = sqlc.read.json("my-file.txt")

my_udf = udf(lambda x: x * x, returnType=types.IntegerType())

data_frame = data_frame.where("field > 100")
                        .withColumn("new_field", my_udf(data_frame.field))

data_frame.write.parquet("processed-file.txt")
```

Using Spark and Airflow



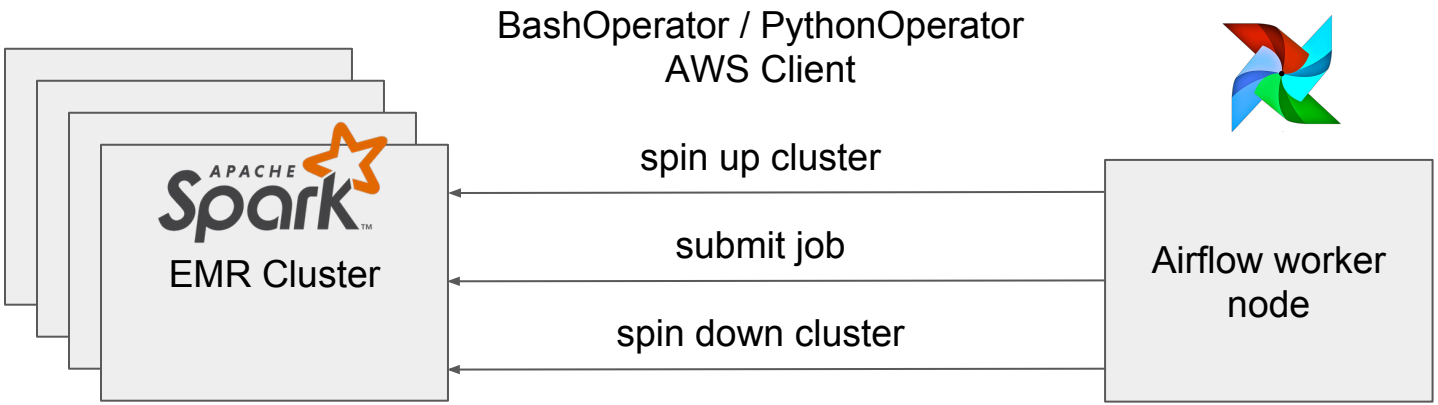
- Client mode, SparkDriver is initialized on the client machine
- Cluster mode is preferable, but:
  - No logs recorded on Airflow
  - How can Airflow know when the task finishes?
    - FileSensor

Using Spark and Airflow



- The Operator should submit the job and then poll the cluster through API requests to check if it the job is done.
- Similar approach taken by DatabricksSubmitRunOperator

Using Spark and Airflow



- There are EMRCreateJobFlowOperator and EMRAddStepsOperator to help, but don't seem to be any operators to spin up and provision and EMR cluster.
- For reference: https://www.agari.com/automated-model-building-emr-spark-airflow/


So in summary

So in summary

- Batch data processing is still used widely, and in whichever form it takes, the principles of ETL-ing are applicable
- Python makes it possible for people who are not software developers to code batch data processing jobs
- Airflow is great tool to have the benefits of GUI ETL-ing tools, with the flexibility of coding
- Airflow can run distributed (CeleryExecutor), but perhaps you won’t need this, if you have a separate distributed architecture to run your jobs, e.g. a separate Hadoop cluster running MapReduce or Spark jobs.

Questions?