UNIVERSITAT DE BARCELONA

Master in Fundamental Principles of Data Science

Dr Rohit Kumar

UNIVERSITAT DE BARCELONA

Data Pipelines

ETL

Extracted – copied from the source system to a staging area
Transformed – reformatted for the warehouse with business calculations applied
Loaded – copied from the staging area into the warehouse

ELT

UNIVERSITAT DE BARCELONA

Data Pipeline

Main Types Of Data Passing Through A Data Pipeline

- **Structured Data:**
- **Unstructured Data:**

The Elements Of A Data Pipeline

- **Sources**: where your data originates. This can include data from CRM systems, third-party vendors, flat files, ERP, RDBMS, email lists, and SaaS applications.
- **Extraction**: data collection from all disparate, siloed sources.
- **Denormalization or standardization**: discarding extraneous data and prioritizing read-oriented over write-oriented data. The process of denormalization/transformation standardizes the format of all extracted data.
- **Loading**: after de-normalizing, the data is loaded into a data lake or data warehouse.
- **Analytics**: data is analyzed using descriptive and predictive analytics. Next, insights are extracted from it. In descriptive analytics, we ask what happened in the past and determine what we can learn. Meanwhile, in predictive analytics, we question what will happen in the future and decide how to adjust our business strategies accordingly.

UNIVERSITAT DE BARCELONA

Stages of Data Pipeline

- **Event frameworks** help you capture events from your applications more easily, creating an event log that can then be processed for use.
- **Message bus** is hardware or software that ensures that data sent between clusters of machines is properly queued and received
- **Data persistence** stores your data in files or other non-volatile storage so that it can be processed in batches, rather than all at once, simultaneously.
- **Workflow management** structures the tasks (or processes) in your data pipeline, and makes it easier to supervise and manage them.
- **Serialization frameworks** convert data into more compact formats for storage and transmission.

UNIVERSITAT DE BARCELONA

Different Options

talend

StreamSets

Apache Airflow

pentaho®

APACHE nifi

AWS Data Pipeline

Azure Data Factory

UNIVERSITAT DE BARCELONA

How to choose?

- **Easy Data Replication –** The tool you choose should allow you to intuitively build a pipeline and set up your infrastructure in minimal time.
- **Maintenance Overhead –** The tool should have minimal maintenance overhead and should work pretty much out of the box.
- **Data Sources Supported –** It should allow you to connect to numerous and various data sources. You should also consider support for those sources you may need in the future.
- **Data Reliability –** It should transfer and load data without error or dropped packet.
- **Realtime Data Availability –** Depending on your use case, decide if you need data real-time or in batches will be just fine.
- **Customer Support –** Any issue while using the tool should be solved quickly and for that choose the one offering most responsive and knowledgeable customer sources

The Evolution of Data Pipelines

Flat File Era: Data is saved locally on servers

The Evolution of Data Pipelines

Database Era: Data is staged in flat files and then loaded into a database

The Evolution of Data Pipelines

Data Lake Era: Data is stored in Hadoop/S3 and then loaded into a DB

The Evolution of Data Pipelines

Serverless Era : Managed services are used for storage and querying

Airflow

A typical data pipeline

1. download data from source
2. send data somewhere else to process
3. Monitor when the process is completed
4. Get the result and generate the report
5. Send the report out by email

A traditional ETL approach

Example of a naive approach:

- Writing a script to pull data from database and send it to HDFS to process.
- Schedule the script as a cronjob.

UNIVERSITAT DE BARCELONA

Problems

- **Failures**:
  - retry if failure happens (how many times? how often?)
- **Monitoring**:
  - success or failure status, how long does the process runs?
- **Dependencies**:
  - Data dependencies: upstream data is missing.
  - Execution dependencies: job 2 runs after job 1 is finished.
- **Scalability**:
  - there is no centralized scheduler between different cron machines.
- **Deployment**:
  - deploy new changes constantly
- **Process historic data**:
  - backfill/rerun historical data

UNIVERSITAT DE BARCELONA

Apache Airflow

- The project joined the Apache Software Foundation's incubation program in 2016.
- A workflow (data-pipeline) management system developed by Airbnb
  - A framework to define tasks & dependencies in python
  - Executing, scheduling, distributing tasks accross worker nodes.
  - View of present and past runs, logging feature
  - Extensible through plugins
  - Nice UI, possibility to define REST interface
  - Interact well with database
- Used by more than 200 companies: Airbnb, Yahoo, Paypal, Intel, Stripe,…

UNIVERSITAT DE BARCELONA

Apache Airflow

Airflow is a platform to programmaticaly author, schedule and monitor workflows or data pipelines.

UNIVERSITAT DE BARCELONA

Airflow Workflow

- It is sequence of task
- Started on a Schedule or triggered by an event
- Frequently used to handle big data processing pipelines.

An Airflow workflow is designed as a directed acyclic graph (DAG). That means, that when authoring a workflow, you should think how it could be divided into tasks which can be executed independently. You can then merge these tasks into a logical whole by combining them into a graph.

UNIVERSITAT DE BARCELONA

What makes Airflow great?

- Can handle upstream/downstream dependencies gracefully (Example: upstream missing tables)
- Easy to reprocess historical jobs by date, or re-run for specific intervals
- Jobs can pass parameters to other jobs downstream
- Handle errors and failures gracefully. Automatically retry when a task fails.
- Ease of deployment of workflow changes (continuous integration)
- Integrations with a lot of infrastructure (Hive, Presto, Druid, AWS, Google cloud, etc)

UNIVERSITAT DE BARCELONA

What makes Airflow great?

- Data sensors to trigger a DAG when data arrives
- Job testing through airflow itself
- Accessibility of log files and other meta-data through the web GUI
- Implement trigger rules for tasks
- Monitoring all jobs status in real time + Email alerts
- Community support

UNIVERSITAT DE BARCELONA

Airflow applications

- **Data warehousing**: cleanse, organize, data quality check, and publish/stream data into our growing data warehouse
- **Machine Learning**: automate machine learning workflows
- **Growth analytics**: compute metrics around guest and host engagement as well as growth accounting
- **Experimentation**: compute A/B testing experimentation frameworks logic and aggregates
- **Email targeting**: apply rules to target and engage users through email campaigns
- **Sessionization**: compute clickstream and time spent datasets
- **Search**: compute search ranking related metrics
- **Data infrastructure maintenance**: database scrapes, folder cleanup, applying data retention policies, …

UNIVERSITAT DE BARCELONA

Airflow DAG

- Workflow as a Directed Acyclic Graph (DAG) with multiple tasks which can be executed independently.
- A DAG is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.

DAG

Acyclic graph: a graph has no cycles.

Cyclic Graph

Cyclic graph: a graph has cycles.A cycle in a directed graph is a path that starts and ends at the same node.

For example, the path (V5,V2,V3,V5) is a cycle. This is a loop.

UNIVERSITAT DE BARCELONA

Airflow DAG

Directed Acyclic Graph is a graph that has **no cycles** and the data in each node flows forward in only **one direction**.

It is useful to represent a complex data flows using a graph.

- Each node in the graph is a task
- The edges represent dependencies amongst tasks.
- These graphs are called computation graphs or data flow graphs and it transform the data as it flow through the graph and enable very complex numeric computations.
- Given that data only needs to be computed once on a given task and the computation then carries forward, the graph is directed and acyclic.

Airflow DAG

start
section-1-task-1
section-1-task-2
section-1-task-3
section-1-task-4
section-1-task-5
some-other-task
section-2-task-1
section-2-task-2
section-2-task-3
section-2-task-4
section-2-task-5
end

Other DAG in ML

TensorFlow uses a dataflow graph to represent your computation in terms of the dependencies between individual operations.

UNIVERSITAT DE BARCELONA

Operators, and Tasks

- **DAGs** do not perform any actual computation. Instead, **Operators** determine what actually gets done.

- **Task**: Once an operator is instantiated, it is referred to as a “task”. An operator describes a single task in a workflow.
  - Instantiating a task requires providing a unique task_id and DAG container

- A **DAG** is a container that is used to organize tasks and set their execution context.

UNIVERSITAT DE BARCELONA

Operator

```
# t1, t2 are examples of tasks created by instantiating operato
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)

t2 = BashOperator(
    task_id='sleep',
    depends_on_past=False,
    bash_command='sleep 5',
    dag=dag,
)
```

UNIVERSITAT DE BARCELONA

Operators categories

Sensors: a certain type of operator that will keep running until a certain criteria is met. Example include waiting for a certain time, external file, or upstream data source.

- HdfsSensor: Waits for a file or folder to land in HDFS
- NamedHivePartitionSensor: check whether the most recent partition of a Hive table is available for downstream processing.

UNIVERSITAT DE BARCELONA

Operators categories

Operators: triggers a certain action (e.g. run a bash command, execute a python function, or execute a Hive query, etc)

- BashOperator: executes a bash command
- PythonOperator: calls an arbitrary Python function
- HiveOperator: executes hql code or hive script in a specific Hive database.
- BigQueryOperator: executes Google BigQuery SQL queries in a specific BigQuery database

UNIVERSITAT DE BARCELONA

Operators categories

Transfers: moves data from one location to another.

- MySqlToHiveTransfer: Moves data from MySql to Hive.
- S3ToRedshiftTransfer: load files from s3 to Redshift

UNIVERSITAT DE BARCELONA

Working with Operators

- Airflow provides prebuilt operators for many common tasks.
- There are more operators being added by the community. You can just go to the Airflow official Github repo, specifically in the airflow/contrib/ directory to look for the community added operators.
- All operators are derived from BaseOperator and acquire much functionality through inheritance. Contributors can extend BaseOperator class to create custom operators as they see fit.

UNIVERSITAT DE BARCELONA

Operators

```
class HiveOperator(BaseOperator):
    """
    HiveOperator inherits from BaseOperator
    """
```

UNIVERSITAT DE BARCELONA

DAG Assignment

There are three ways to assisgn operators to DAG

```
dag = DAG('my_dag', start_date=datetime(2016, 1, 1))

# sets the DAG explicitly
explicit_op = DummyOperator(task_id='op1', dag=dag)

# deferred DAG assignment
deferred_op = DummyOperator(task_id='op2')
deferred_op.dag = dag

# inferred DAG assignment (linked operators must be in the same DAG)
inferred_op = DummyOperator(task_id='op3')
inferred_op.set_upstream(deferred_op)
```

UNIVERSITAT DE BARCELONA

Common Operators

- BashOperator - executes a bash command
- PythonOperator - calls an arbitrary Python function
- EmailOperator - sends an email
- SimpleHttpOperator - sends an HTTP request
- MySqlOperator, SqliteOperator, PostgresOperator, MsSqlOperator, OracleOperator, JdbcOperator, etc. - executes a SQL command
- In addition to these basic building blocks, there are many more specific operators: DockerOperator, HiveOperator, S3FileTransformOperator, PrestoToMySqlTransfer, SlackAPIOperator

UNIVERSITAT DE BARCELONA

TASK

- A Task defines a unit of work within a DAG.
- It is represented as a node in the DAG graph, and it is written in Python.
- Each task is an implementation of an Operator, for example a PythonOperator to execute some Python code, or a BashOperator to run a Bash command.
- Each task is an implementation of an Operator, for example a PythonOperator to execute some Python code, or a BashOperator to run a Bash command.

UNIVERSITAT DE BARCELONA

Defining Task Dependencies

After defining a DAG, and instantiate all the tasks, you can then set the dependencies or the order in which the tasks should be executed.

Task dependencies are set using:

- the set_upstream and set_downstream operators.
- the bitshift operators << and >>

```
# This means that t2 will depend on t1
# running successfully to run.
t1.set_downstream(t2)

# bit shift operator
# t1 >> t2
```

UNIVERSITAT DE BARCELONA

Bitshift Composition

```
op1 >> op2
op1.set_downstream(op2)

op2 << op1
op2.set_upstream(op1)
```

op1 runs first and op2 runs second.

Bitshift Composition

Bitshift can also be used with lists. For example:

```
op1 >> [op2, op3] >> op4
```

UNIVERSITAT DE BARCELONA

Relationship Builders

chain and **cross_downstream** function provide easier ways to set relationships between operators in specific situation. *Moved in Airflow 2.0*

When setting a relationship between two lists, if we want all operators in one list to be upstream to all operators in the other, we cannot use a single bitshift composition. Instead we have to split one of the lists:

```
[op1, op2, op3] >> op4
[op1, op2, op3] >> op5
[op1, op2, op3] >> op6
```

Relationship Builders

cross_downstream could handle list relationships easier.

```
cross_downstream([op1, op2, op3], [op4, op5, op6])
```

Equivalent to

```
[op1, op2, op3] >> op4
[op1, op2, op3] >> op5
[op1, op2, op3] >> op6
```

Relationship Builders

When setting single direction relationships to many operators, we could concat them with bitshift composition.

```
op1 >> op2 >> op3 >> op4 >> op5
```

is equivalent to:

```
chain(op1, op2, op3, op4, op5)
```

UNIVERSITAT DE BARCELONA

DagRuns

A key concept in Airflow is the **execution_time**. The execution times begin at the DAG's **start_date** and repeat every **schedule_interval**.

For this example the scheduled execution times would be ("2018–12–01 00:00:00", "2018–12–02 00:00:00", …).

For each execution_time, a DagRun is created and operates under the context of that execution time.

A DagRun is simply a DAG that has a specific execution time.

UNIVERSITAT DE BARCELONA

DagRuns

```
default_args = {
    'owner': 'airflow',
    'start_date': datetime(2018, 12, 01),
    # 'end_date': datetime(2018, 12, 30),
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    }

dag = DAG(
    'tutorial',
    default_args=default_args,
    description='A simple tutorial DAG',
    # Continue to run DAG once per day
    schedule_interval=timedelta(days=1),
)
```

DagRuns are DAGs that runs at a certain time

UNIVERSITAT DE BARCELONA

TaskInstances

TaskInstances are the task belongs to that **DagRuns**.

Each **DagRun** and **TaskInstance** is associated with an entry in Airflow's metadata database that logs their state (e.g. "queued", "running", "failed", "skipped", "up for retry").

Task Lifecycle

A task goes through various stages from start to completion. In the Airflow UI (graph and tree views), these stages are displayed by a color representing each stage:

Task lifecycle

The complete lifecycle of the task looks like this:

UNIVERSITAT DE BARCELONA

Task lifecycle

The happy flow consists of the following stages:

- No status (scheduler created empty task instance)
- Scheduled (scheduler determined task instance needs to run)
- Queued (scheduler sent task to executor to run on the queue)
- Running (worker picked up a task and is now running it)
- Success (task completed)

Task lifecycle

There is also visual difference between scheduled and manually triggered DAGs/tasks:

The DAGs/tasks with a black border are scheduled runs, whereas the non-bordered DAGs/tasks are manually triggered, i.e. by airflow dags trigger

UNIVERSITAT DE BARCELONA

Default Arguments

If a dictionary of default_args is passed to a DAG, it will apply them to any of its operators. This makes it easy to apply a common parameter to many operators without having to type it many times.

UNIVERSITAT DE BARCELONA

Airflow Variables

- Variables are key-value stores in Airflow's metadata database.
- It is used to store and retrieve arbitrary content or settings from the metadata database.

UNIVERSITAT DE BARCELONA

When to use Variables

- Variables are mostly used to store static values like:
  - config variables
  - a configuration file
  - list of tables
  - list of IDs to dynamically generate tasks from
- Separate the constants and variables from pipeline code:
  - It is useful to have some variables or configuration items accessible and modifiable through the UI.

UNIVERSITAT DE BARCELONA

Working with Variables

- Variables can be listed, created, updated and deleted from the UI (Admin -> Variables).
- In addition, json settings files can be bulk uploaded through the UI.
  - Example:

```
{
        "example_variables_config": {
                "var1": "value1",
                "var2": [1, 2, 3],
                "var3": {
                        "k": "value3"
                }
        }
}
```

You have to keep inside Airflow home folder/dags/config

UNIVERSITAT DE BARCELONA

Airflow Variables

Variables

Choose File No file chosen | Import Variables

List (4) | Create | Add Filter | With selected | Search

		Key	Val
		example_variables_config	{"var1": "value1", "var2": [1, 2, 3], "var3": {"k": "value3"}}
		var1	value1
		var2	[1, 2, 3]
		var3	{'k': 'value3'}

UNIVERSITAT DE BARCELONA

Accessing Variables

```
var1 = Variable.get("var1")
var2 = Variable.get("var2")
var3 = Variable.get("var3")
```

!! Problem 3 DB connection.

Recommendation:

```
dag_config = Variable.get("example_variables_config",
deserialize_json=True)
var1 = dag_config["var1"]
var2 = dag_config["var2"]
var3 = dag_config["var3"]
```

UNIVERSITAT DE BARCELONA

Accessing Variables

You can directly use a variable from a jinja template

1) {{ var.value.<variable_name> }}

```
t2 = BashOperator( task_id="get_variable_value",
    bash_command='echo {{ var.value.var3 }})
```

2) {{ var.json.<variable_name> }}

```
t3 = BashOperator( task_id="get_variable_json",
bash_command='echo {{
var.json.example_variables_config.var3 }} ')
```

UNIVERSITAT DE BARCELONA

Carefull with Variables!!

Since Airflow Variables are stored in Metadata Database, so any call to variables would mean a connection to Metadata DB.

Instead of storing a large number of variable in your DAG, which may end up saturating the number of allowed connections to your database.

It is recommended you store all your DAG configuration inside a single Airflow variable with JSON value.

UNIVERSITAT DE BARCELONA

Steps to write an Airflow DAG

A DAG file, which is basically just a Python script, is a configuration file specifying the DAG's structure as code.

There are only 5 steps you need to remember to write an Airflow DAG or workflow:

Step 1: Importing modules

Step 2: Default Arguments

Step 3: Instantiate a DAG

Step 4: Tasks

Step 5: Setting up Dependencies

UNIVERSITAT DE BARCELONA

Step 1: Importing modules

Import Python dependencies needed for the workflow

```
from datetime import timedelta

import airflow
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
```

UNIVERSITAT DE BARCELONA

Step 2: Default Arguments

Define default and DAG-specific arguments

```
default_args = {
    'owner': 'airflow',
    'start_date': airflow.utils.dates.days_ago(2),
    # 'end_date': datetime(2018, 12, 30),
    'depends_on_past': False,
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    # If a task fails, retry it once after waiting
    # at least 5 minutes
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    }
```

UNIVERSITAT DE BARCELONA

Step 3: Instantiate a DAG

Give the DAG name, configure the schedule, and set the DAG settings

```
dag = DAG(
    'tutorial',
    default_args=default_args,
    description='A simple tutorial DAG',
    # Continue to run DAG once per day
    schedule_interval=timedelta(days=1),
)
```

UNIVERSITAT DE BARCELONA

Step 3: Instantiate a DAG

You can choose to use some preset argument or cron-like argument:

preset	meaning	cron
`None`	Don't schedule, use for exclusively "externally triggered" DAGs	
`@once`	Schedule once and only once	
`@hourly`	Run once an hour at the beginning of the hour	`0 * * * *`
`@daily`	Run once a day at midnight	`0 0 * * *`
`@weekly`	Run once a week at midnight on Sunday morning	`0 0 * * 0`
`@monthly`	Run once a month at midnight of the first day of the month	`0 0 1 * *`
`@yearly`	Run once a year at midnight of January 1	`0 0 1 1 *`

Example of setting schedule

Daily schedule:

schedule_interval='@daily'

schedule_interval='0 0 * * *’

https://cronreader.com/

1 4 * * * => At 04:01 Everyday

1 4 * * 2 => At 04:01, On Tuesday

UNIVERSITAT DE BARCELONA

Step 4: Tasks

The next step is to lay out all the tasks in the workflow.

```
# t1, t2 and t3 are examples of tasks created by instantiating
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)

t2 = BashOperator(
    task_id='sleep',
    depends_on_past=False,
    bash_command='sleep 5',
    dag=dag,
)
```

UNIVERSITAT DE BARCELONA

Templating with Jinja

Airflow leverages the power of Jinja Templating and provides the pipeline author with a set of built-in parameters and macros. Airflow also provides hooks for the pipeline author to define their own parameters, macros and templates.

{{ ds }} (today's "date stamp")

```
templated_command = """
{% for i in range(5) %}
    echo "{{ ds }}"
    echo "{{ macros.ds_add(ds, 7)}}"
    echo "{{ params.my_param }}"
{% endfor %}
"""


t3 = BashOperator(
    task_id='templated',
    depends_on_past=False,
    bash_command=templated_command,
    params={'my_param': 'Parameter I passed in'},
    dag=dag,
)
```

Notice that the templated_command contains code logic in {% %} blocks, references parameters like {{ ds }}, calls a function as in {{ macros.ds_add(ds, 7)}}, and references a user-defined parameter in {{ params.my_param }}

UNIVERSITAT DE BARCELONA

Step 5: Setting up Dependencies

Set the dependencies or the order in which the tasks should be executed.

```
# This means that t2 will depend on t1
# running successfully to run.
t1.set_downstream(t2)

# similar to above where t3 will depend on t1
t3.set_upstream(t1)
```

Step 5: Setting up Dependencies

```
# The bit shift operator can also be
# used to chain operations:
t1 >> t2

# And the upstream dependency with the
# bit shift operator:
t2 << t1
```

Step 5: Setting up Dependencies

```
# A list of tasks can also be set as
# dependencies. These operations
# all have the same effect:
t1.set_downstream([t2, t3])
t1 >> [t2, t3]
[t2, t3] << t1
```

UNIVERSITAT DE BARCELONA

Step 6: Running the Script

The default location for your DAGs is ~/airflow/dags

You should copy your Python file there.

You can change the location in airflow.cfg file.

UNIVERSITAT DE BARCELONA

Step 6: Running the Script

Command Line Metadata Validation

```
# print the list of active DAGs

$ airflow list_dags

# prints the list of tasks the "tutorial" dag_id

$ airflow list_tasks tutorial

# prints the hierarchy of tasks in the tutorial DAG

$ airflow list_tasks tutorial --tree
```

UNIVERSITAT DE BARCELONA

Testing

```
# command layout: command subcommand dag_id task_id date

# testing print_date
airflow test tutorial print_date 2015-06-01

# testing sleep
airflow test tutorial sleep 2015-06-01
```

But lets test from front end ☺

localhost:8080

UNIVERSITAT DE BARCELONA

References

- https://airflow.apache.org/docs/stable/tutorial.html
- http://michal.karzynski.pl/blog/2017/03/19/developing-workflows-with-apache-airflow/
- https://www.polidea.com/blog/apache-airflow-tutorial-and-beginners-guide/
- https://towardsdatascience.com/getting-started-with-apache-airflow-df1aa77d7b1b