Universitat de Barcelona

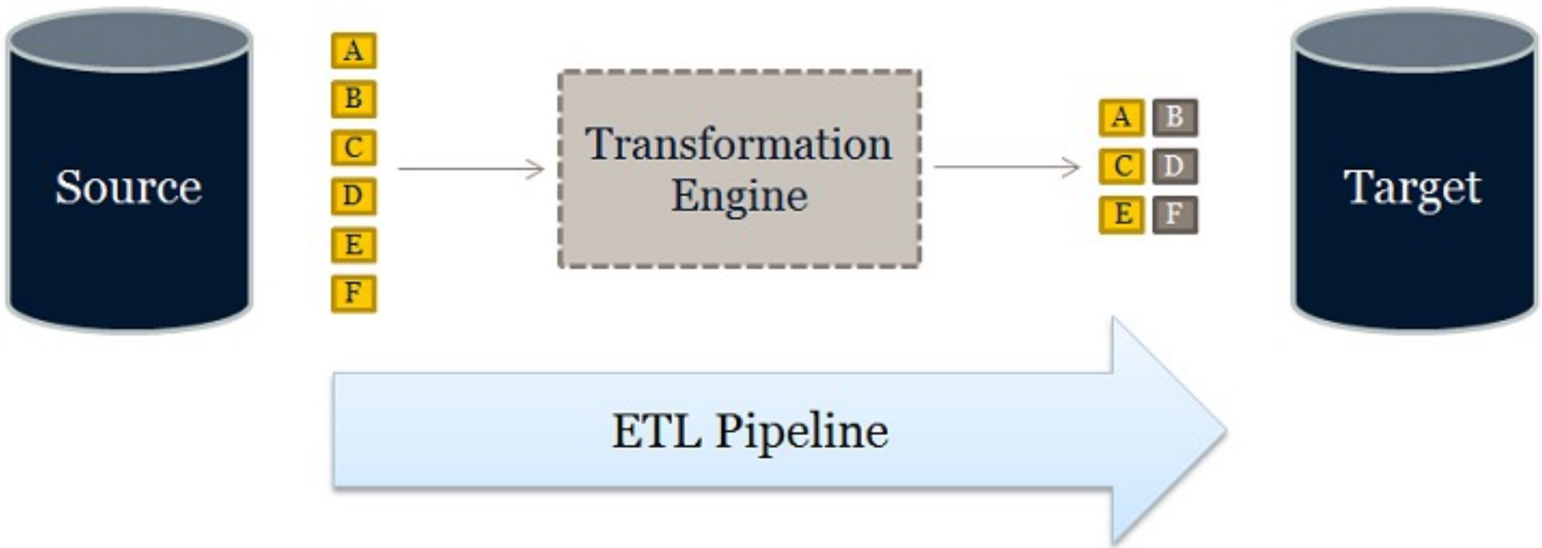
Master in Fundamental Principles of Data Science

Dr Rohit Kumar

UNIVERSITAT DE BARCELONA

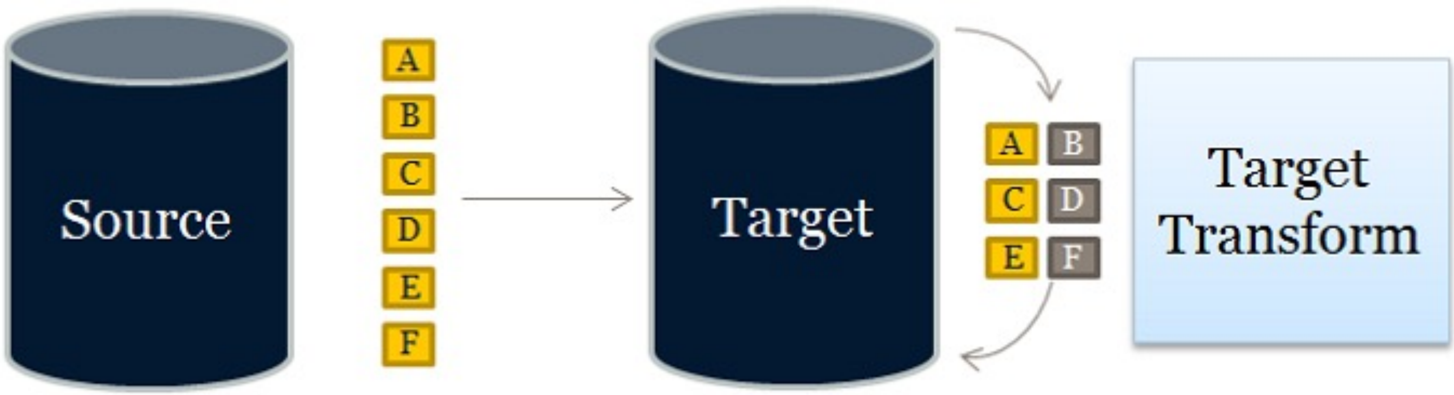
Data Pipelines

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:**

Different Options

talend

StreamSets

pentaho®

Apache Airflow

APACHE nifi

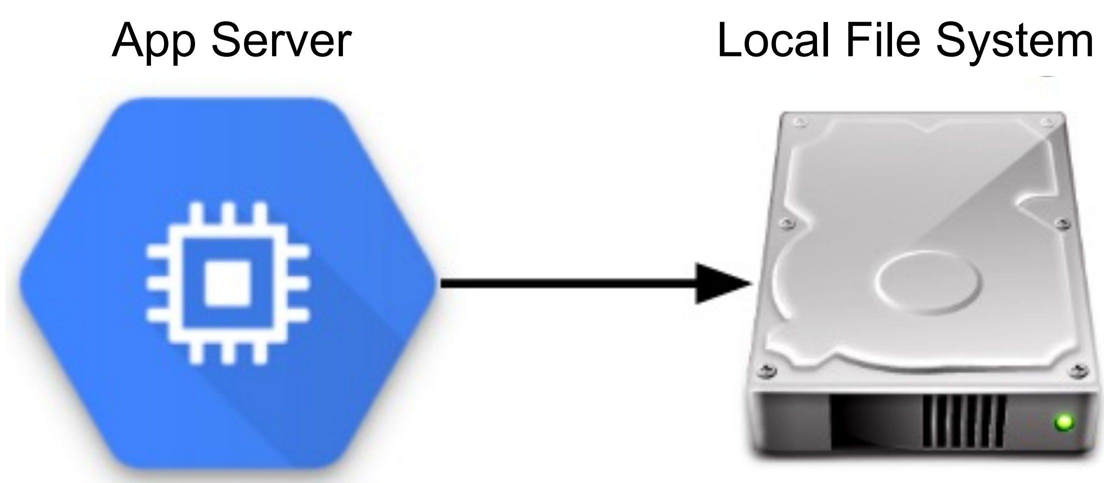
AWS Data Pipeline

Azure Data Factory

UNIVERSITAT DE BARCELONA

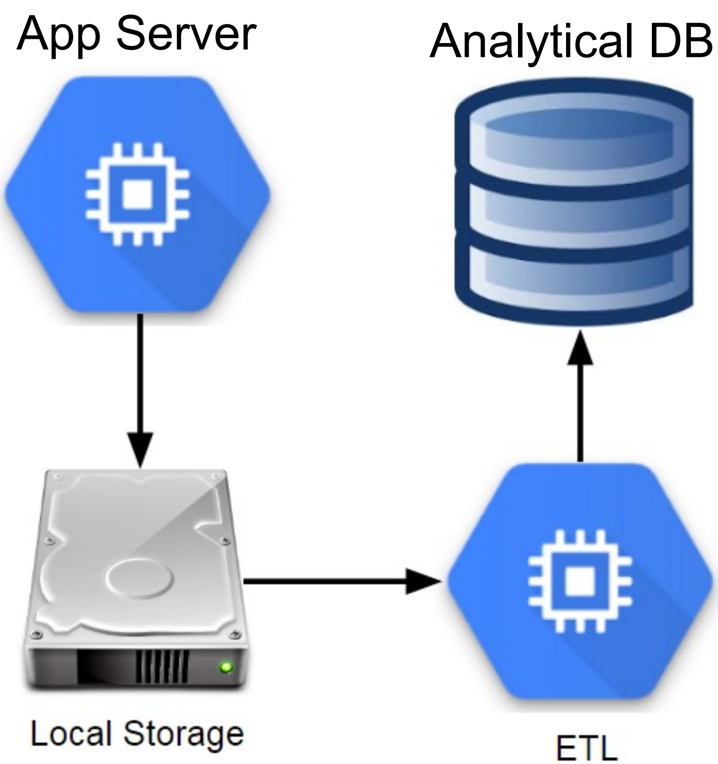
The Evolution of Data Pipelines

Flat File Era: Data is saved locally on servers



The Evolution of Data Pipelines

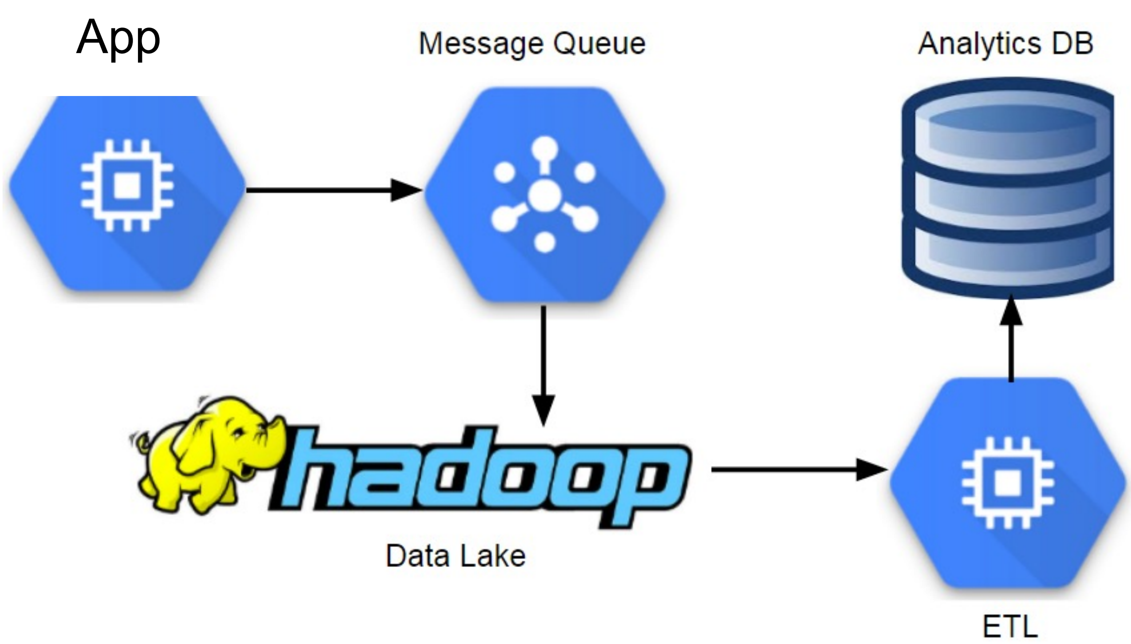
Database Era: Data is staged in flat files and then loaded into a database



UNIVERSITAT DE BARCELONA

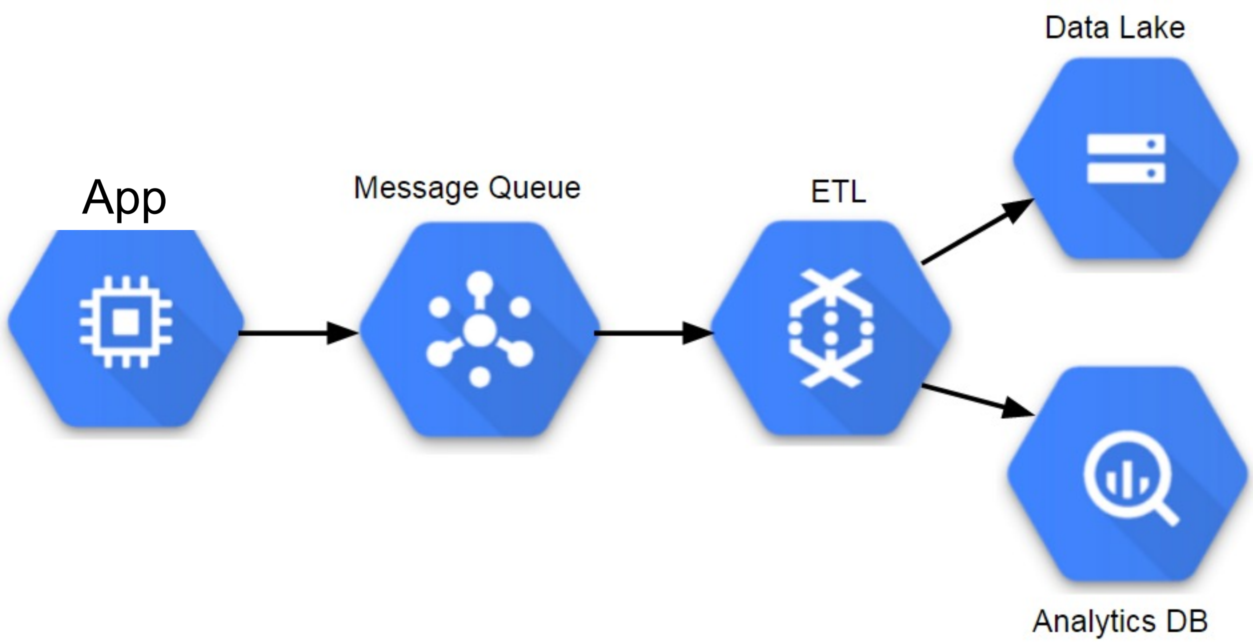
The Evolution of Data Pipelines

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

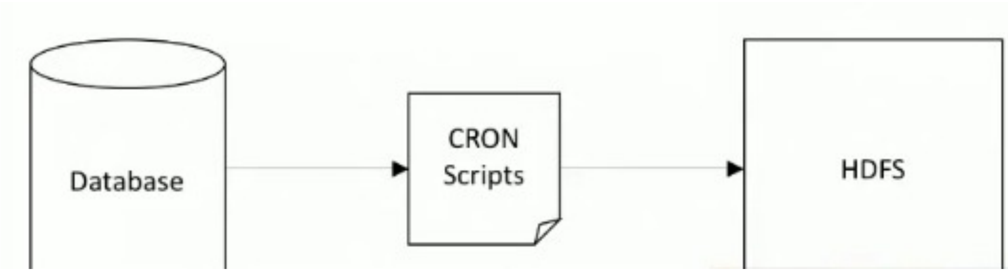


Airflow

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.

Lets see how to do cron Jobs!!!

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,…

Apache Airflow

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

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.

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)

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

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, …

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.

Airflow DAG

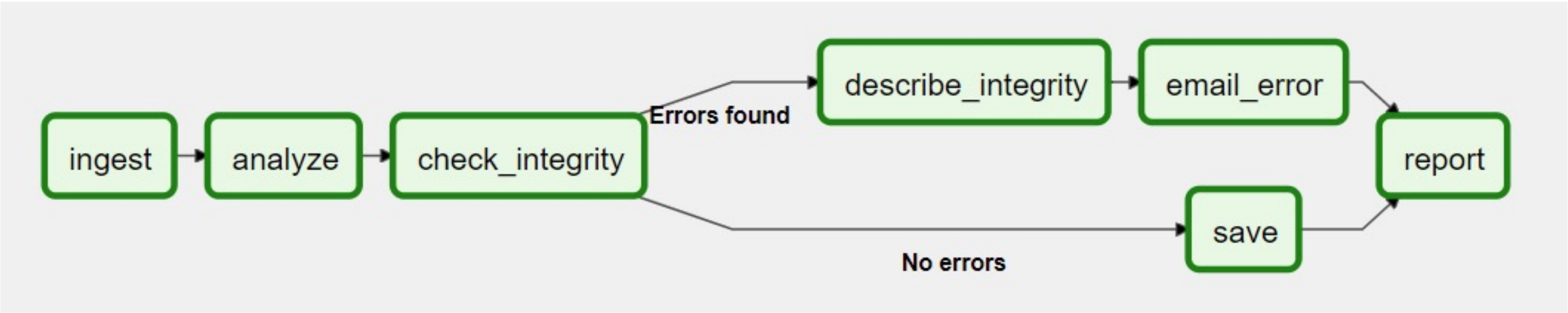
Directed Acyclic Graph is a graph that has **no cycles** and the data in each node flows forward in only **one direction**.

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

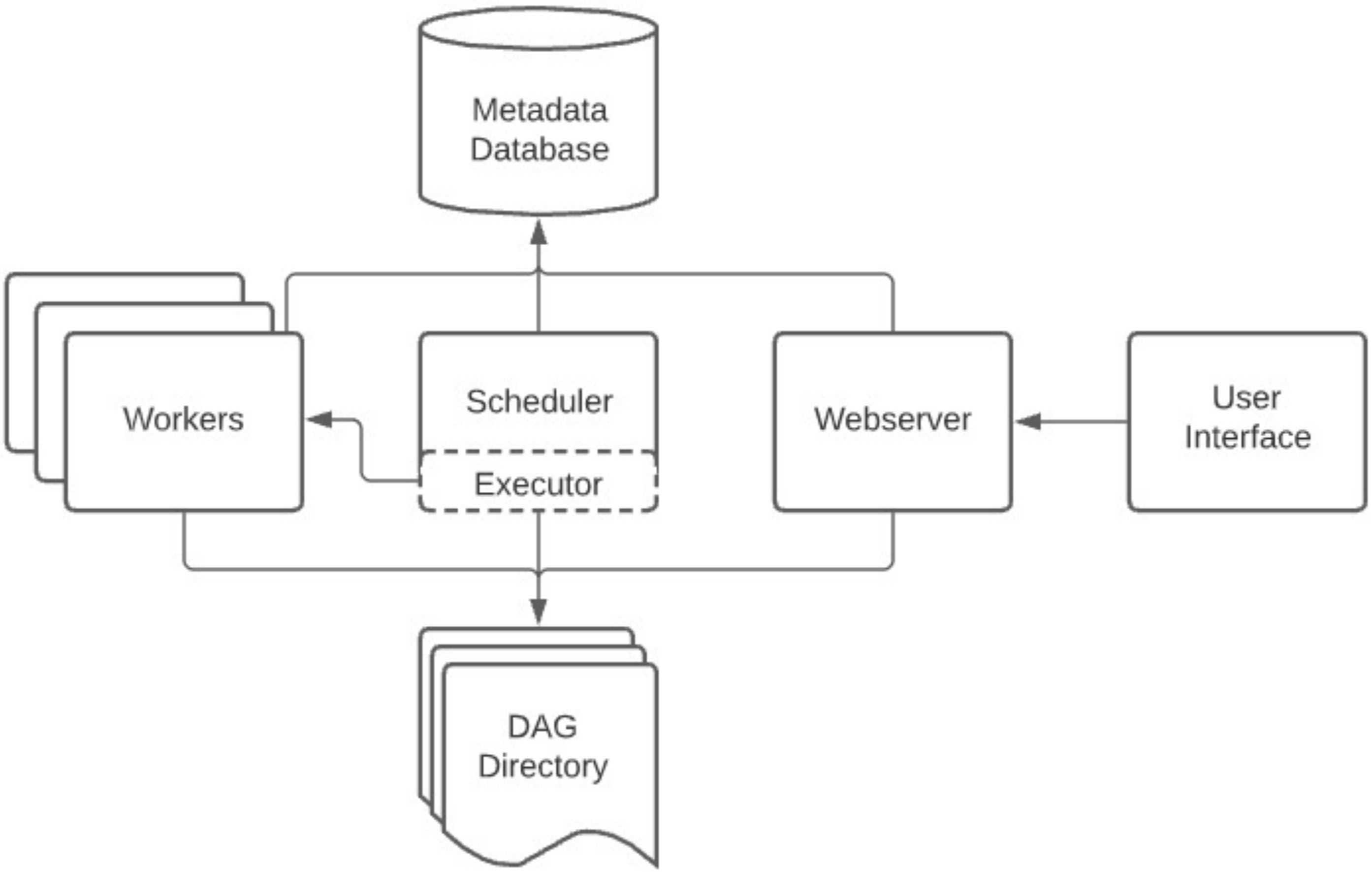
- Each node in the graph is a task
- The edges represent dependencies amongst tasks.
- These graphs are called computation graphs or data flow graphs and it transform the data as it flow through the graph and enable very complex numeric computations.
- Given that data only needs to be computed once on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

UNIVERSITAT DE BARCELONA

Airflow DAG



Architecture Overview



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.

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

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.

Operators

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

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)
```

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.

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
```

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
```

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)
```

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

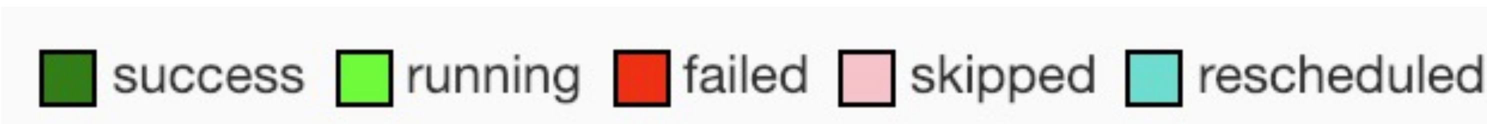
TaskInstances

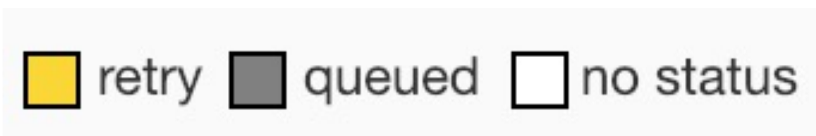
TaskInstances are the task belongs to that **DagRuns**.

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

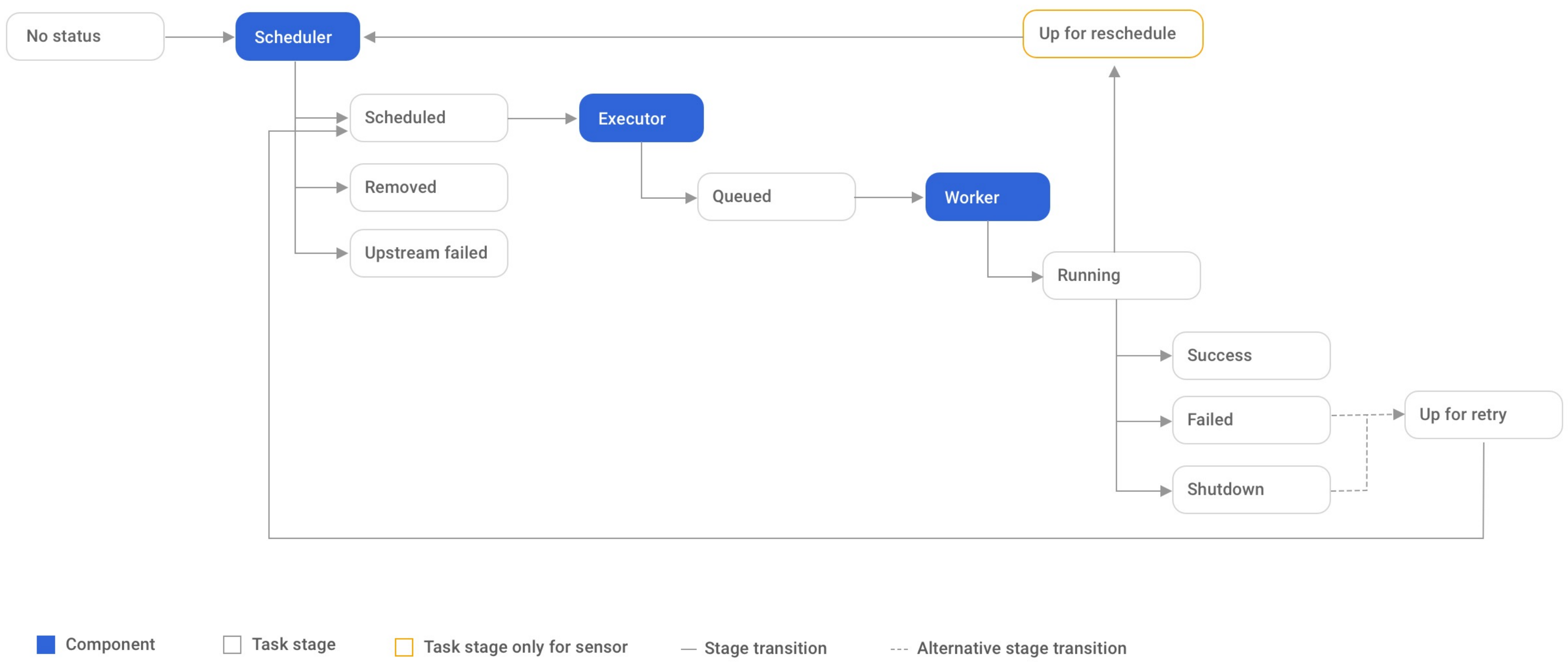
A task goes through various stages from start to completion. In the Airflow UI (graph and tree views), these stages are displayed by a color representing each stage:





Task lifecycle

The complete lifecycle of the task looks like this:



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

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.

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 0: Install airflow Python library

Step 1: Importing modules

Step 2: Default Arguments

Step 3: Instantiate a DAG

Step 4: Tasks

Step 5: Setting up Dependencies

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
```

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),
    }
```

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),
)
```

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 * * *'
schedule_interval=timedelat(days=1)

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,
)
```

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 }}

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)
```

UNIVERSITAT DE BARCELONA

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
```

Step 6: Running the Script

Use Docker to setup airflow cluster

https://airflow.apache.org/docs/apache-airflow/stable/start/docker.html

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.

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