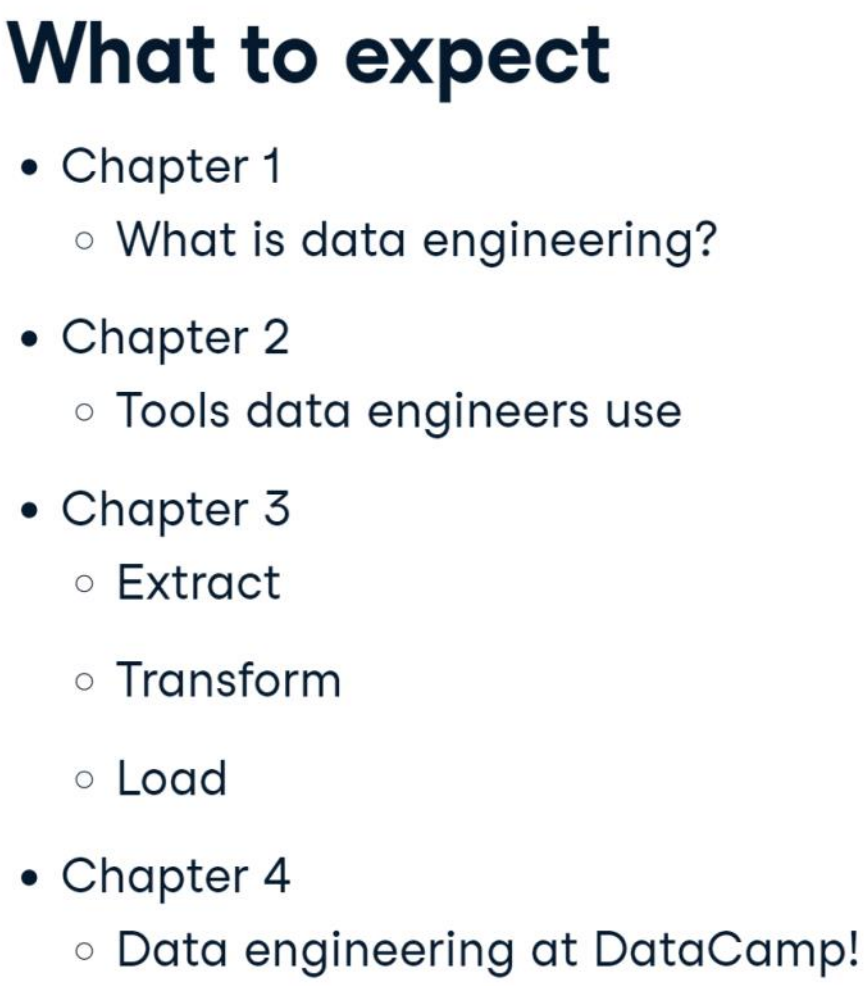
Introduction to Data Engineering

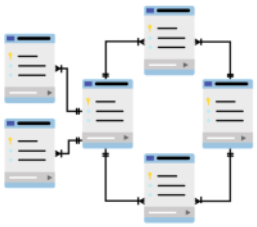
Friday, 20 November 2020 2:11 PM

Introduction to Data Engineering



In comes the data engineer

- Data is scattered
- Not optimized for analyses
- Legacy code is causing corrupt data

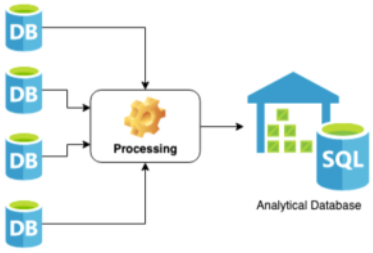


Data engineer to the rescue!

Data engineers: making your life easier

- Gather data from different sources
- Optimized database for analyses
- Removed corrupt data

Data scientist's life got way easier!



Definition of the job

> An engineer that develops, constructs, tests, and maintains architectures such as databases and large-scale processing systems

- Processing large amounts of data
- Use of clusters of machines

Data Engineer vs Data Scientist

Data Engineer

- Develop scalable data architecture
- Streamline data acquisition
- Set up processes to bring together data
- Clean corrupt data
- Well versed in cloud technology

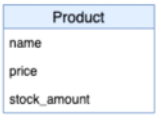
Data Scientist

- Mining data for patterns
- Statistical modeling
- Predictive models using machine learning
- Monitor business processes
- Clean outliers in data

Databases

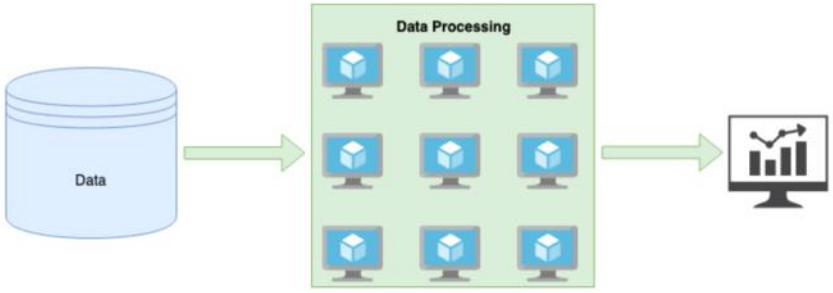
- Hold large amounts of data
- Support application

- Other databases are used for analyses



Processing

- Clean data
- Aggregate data
- Join data



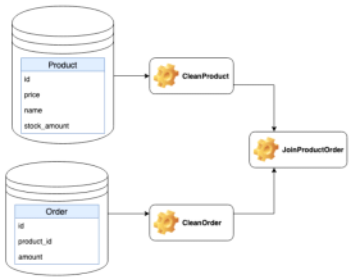
Processing: an example

```
df = spark.read.parquet("users.parquet")

outliers = df.filter(df["age"] > 100)

print(outliers.count())
```

Data engineer understands the abstractions.

Scheduling

- Plan jobs with specific intervals
- Resolve dependency requirements of jobs



`JoinProductOrder` needs to run after `CleanProduct` and `CleanOrder`

Existing tools

Databases

MySQL

PostgreSQL

Scheduling

Airflow

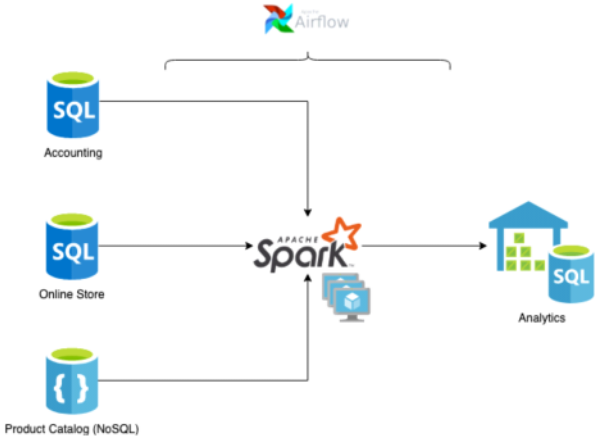
OOZIE

Processing

Spark

HIVE

A data pipeline



Data processing in the cloud

Clusters of machines required

Problem: self-host data-center

- Cover electrical and maintenance costs
- Peaks vs. quiet moments: hard to optimize

Solution: use the cloud

Data storage in the cloud

Reliability is required

Problem: self-host data-center

- Disaster will strike
- Need different geographical locations

Solution: use the cloud

The big three: AWS, Azure and Google

32% market share in 2018

17% market share in 2018

10% market share in 2018

- Storage
- Computation
- Databases.

Storage

Upload files, e.g. storing product images

Services

- AWS S3
- Azure Blob Storage
- Google Cloud Storage

Computation

Perform calculations, e.g. hosting a web server

Services

- AWS EC2
- Azure Virtual Machines
- Google Compute Engine

Databases

Hold structured information

Services

- AWS RDS
- Azure SQL Database
- Google Cloud SQL

Data engineering toolbox

What are databases?

- Holds data
- Organizes data
- Retrieve/Search data through DBMS

> *A usually large collection of data organized especially for rapid search and retrieval.*

Databases and file storage

Databases

- Very organized
- Functionality like search, replication, ...

File systems

- Less organized
- Simple, less added functionality

Structured and unstructured data

Structured: database schema

- Relational database

Semi-structured

```
{ "key": "value"}
```

- JSON

Unstructured: schemaless, more like files

- Videos, photos

SQL and NoSQL

SQL

- Tables
- Database schema
- Relational databases

MySQL

PostgreSQL

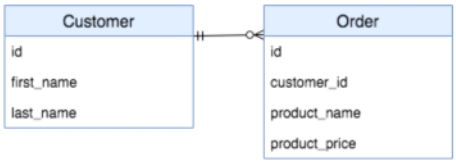
NoSQL

- Non-relational databases
- Structured or unstructured
- Key-value stores (e.g. caching)
- Document DB (e.g. JSON objects)

redis mongoDB.

SQL: The database schema

```sql
-- Create Customer Table
CREATE TABLE "Customer" (
  "id" SERIAL NOT NULL,
  "first_name" varchar,
  "last_name" varchar,
  PRIMARY KEY ("id")
);

-- Create Order Table
CREATE TABLE "Order" (
  "id" SERIAL NOT NULL,
  "customer_id" integer REFERENCES "Customer",
  "product_name" varchar,
  "product_price" integer,
  PRIMARY KEY ("id")
);
```

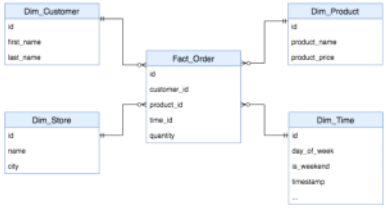



```sql
-- Join both tables on foreign key
SELECT * FROM "Customer"
INNER JOIN "Order"
ON "customer_id" = "Customer"."id";
```

```
id | first_name | ... | product_price
 1 | Vincent    | ... |            10
```

SQL: Star schema

> *The star schema consists of one or more fact tables referencing any number of dimension tables.*



- **Facts**: things that happened (eg. Product Orders)
- **Dimensions**: information on the world (eg. Customer Information)

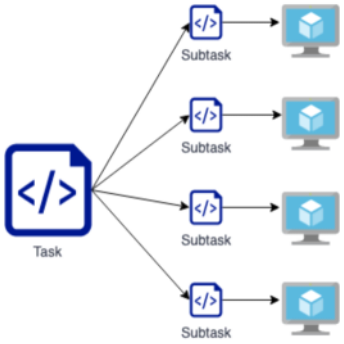
Idea behind parallel computing

Basis of modern data processing tools

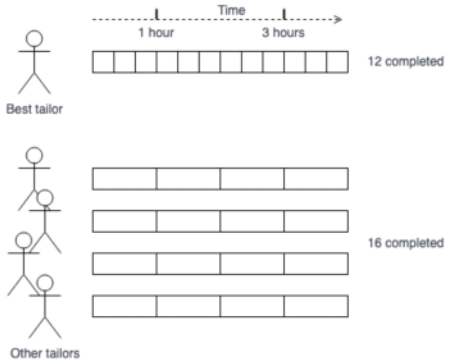
- Memory
- Processing power

Idea

- Split task into subtasks
- Distribute subtasks over several computers
- Work together to finish task



The tailor shop



Running a tailor shop

Goal: 100 shirts

- Best tailor finishes shirt / 20 minutes
- Other tailors do shirt / 1 hour

Multiple tailors working together > best tailor

Benefits of parallel computing

- Processing power
- Memory: partition the dataset

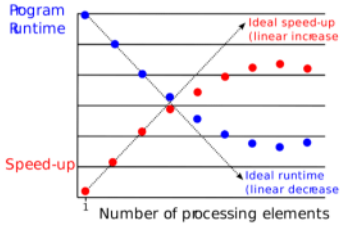
RAM memory chip:

Risks of parallel computing

Overhead due to communication

- Task needs to be large
- Need several processing units

Parallel slowdown:



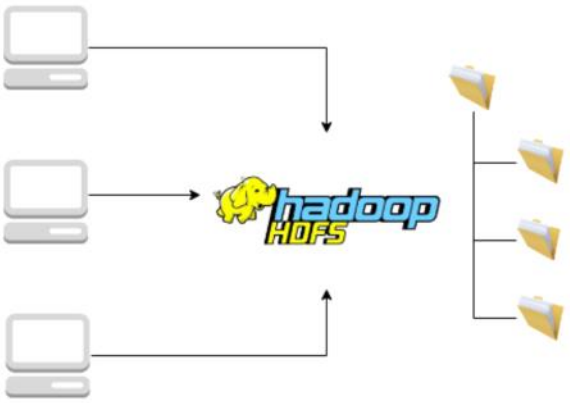
`multiprocessing.Pool`

```
from multiprocessing import Pool

def take_mean_age(year_and_group):
    year, group = year_and_group
    return pd.DataFrame({"Age": group["Age"].mean()}, index=[year])


with Pool(4) as p:
    results = p.map(take_mean_age, athlete_events.groupby("Year"))


result_df = pd.concat(results)
```

`dask`

```
import dask.dataframe as dd

# Partition dataframe into 4
athlete_events_dask = dd.from_pandas(athlete_events, npartitions = 4)

# Run parallel computations on each partition
result_df = athlete_events_dask.groupby('Year').Age.mean().compute()
```

HDFS



Hive

- Runs on Hadoop
- Structured Query Language: Hive SQL
- Initially MapReduce, now other tools

Hive: an example

```
SELECT year, AVG(age)
FROM views.athlete_events
GROUP BY year
```

- Avoid disk writes
- Maintained by Apache Software Foundation

Resilient distributed datasets (RDD)

- Spark relies on them
- Similar to list of tuples
- Transformations: `.map()` or `.filter()`
- Actions: `.count()` or `.first()`

PySpark

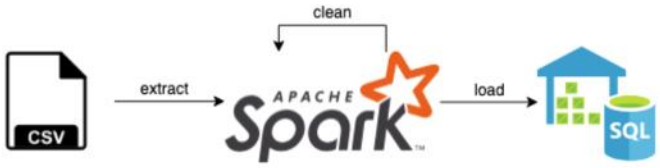
- Python interface to Spark
- DataFrame abstraction
- Looks similar to Pandas

PySpark: an example

```
# Load the dataset into athlete_events_spark first

(athlete_events_spark
  .groupBy('Year')
  .mean('Age')
  .show())
```

```
SELECT year, AVG(age)
FROM views.athlete_events
GROUP BY year
```

An example pipeline



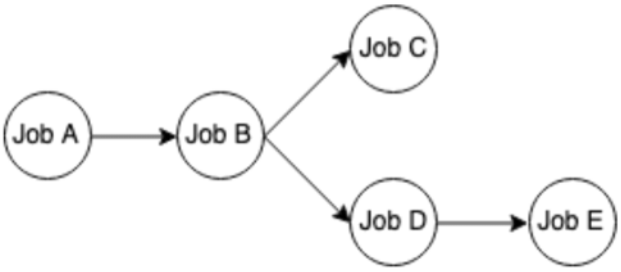
How to schedule?

- Manually
- `cron` scheduling tool
- What about dependencies?

DAGs

Directed Acyclic Graph

- Set of nodes
- Directed edges
- No cycles

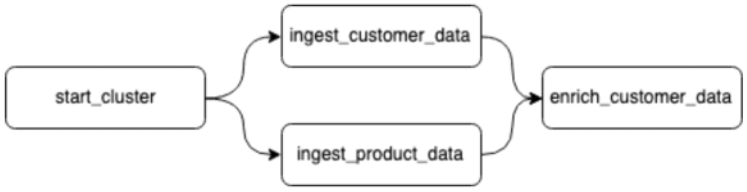


The tools for the job

- Linux's `cron`
- Spotify's Luigi
- Apache Airflow

- Created at Airbnb
- DAGs
- Python

Airflow: an example DAG

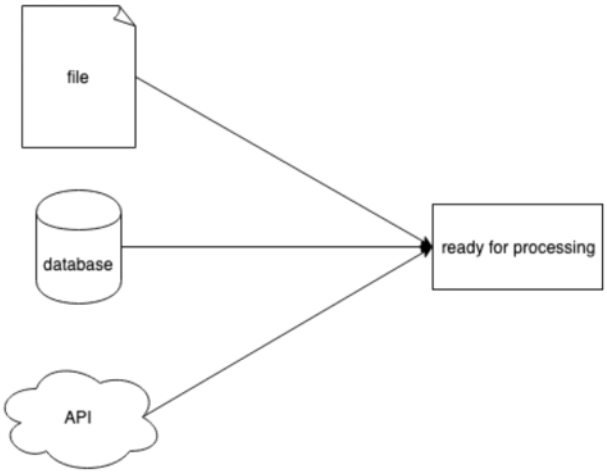


Airflow: an example in code

```
# Create the DAG object
dag = DAG(dag_id="example_dag", ..., schedule_interval="0 * * * *")

# Define operations
start_cluster = StartClusterOperator(task_id="start_cluster", dag=dag)
ingest_customer_data = SparkJobOperator(task_id="ingest_customer_data", dag=dag)
ingest_product_data = SparkJobOperator(task_id="ingest_product_data", dag=dag)
enrich_customer_data = PythonOperator(task_id="enrich_customer_data", ..., dag = dag)

# Set up dependency flow
start_cluster.set_downstream(ingest_customer_data)
ingest_customer_data.set_downstream(enrich_customer_data)
ingest_product_data.set_downstream(enrich_customer_data)
```

Extract, Transform and Load (ETL)

Extracting data: what does it mean?



Extract from text files

Unstructured

- Plain text
- E.g. chapter from a book

```
Call me Ishmael. Some years ago—never
mind how long precisely—having little or
no money in my purse, and nothing particular
to interest me on shore, I thought ....
```

Flat files

- Row = record
- Column = attribute
- E.g. `.tsv` or `.csv`

```
Year,Make,Model,Price
1997,Ford,E350,3000.00
1999,Chevy,"Venture Extended Edition",4900.00
1999,Chevy,"Venture Extended Edition",5000.00
1996,Jeep,Grand Cherokee,4799.00
```

JSON

- JavaScript Object Notation
- Semi-structured
- Atomic
  - `number`
  - `string`
  - `boolean`
  - `null`
- Composite
  - `array`
  - `object`

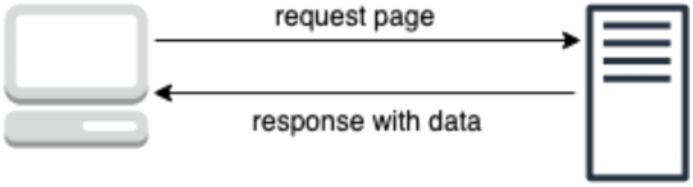
```
{
  "an_object": {
    "nested": [
      "one",
      "two",
      "three",
      {
        "key": "four"
      }
    ]
  }
}
```

```
import json
result = json.loads('{"key_1": "value_1",
                      "key_2":"value_2"}')

print(result["key_1"])
```

```
value_1
```

Data on the Web

Requests



Example

1. *Browse to Google*
2. *Request to Google Server*

Data on the Web through APIs

- Send data in JSON format
- API: application programming interface
- Examples
  - Twitter API

```
{ "statuses": [{ "created_at": "Mon May 06 20:01:29 +0000 2019", "text": "this is a tweet"}] }
```

- Hackernews API

```
import requests

response = requests.get("https://hacker-news.firebaseio.com/v0/item/16222426.json")
print(response.json())
```

```
{'by': 'neis', 'descendants': 0, 'id': 16222426, 'score': 17, 'time': 1516800333, 'title': .... }
```

Data in databases

Applications databases

- Transactions
- Inserts or changes
- OLTP
- Row-oriented

Analytical databases

- OLAP
- Column-oriented

Extraction from databases

Connection string/URI

```
postgresql://[user[:password]@][host][:port]
```

Use in Python

```
import sqlalchemy
connection_uri = "postgresql://repl:password@localhost:5432/pagila"
db_engine = sqlalchemy.create_engine(connection_uri)

import pandas as pd
pd.read_sql("SELECT * FROM customer", db_engine)
```

Kind of transformations

customer_id	email	state	created_at
1	jane.doe@theweb.com	New York	2019-01-01 07:00:00

- Selection of attribute (e.g. 'email')
- Translation of code values (e.g. 'New York' -> 'NY')
- Data validation (e.g. date input in 'created_at')
- Splitting columns into multiple columns
- Joining from multiple sources

An example: split (Pandas)

customer_id	email	username	domain
1	jane.doe@theweb.com	jane.doe	theweb.com

```
customer_df # Pandas DataFrame with customer data

# Split email column into 2 columns on the '@' symbol
split_email = customer_df.email.str.split("@", expand=True)
# At this point, split_email will have 2 columns, a first
# one with everything before @, and a second one with
# everything after @

# Create 2 new columns using the resulting DataFrame.
customer_df = customer_df.assign(
  username=split_email[0],
  domain=split_email[1],
)
```

Transforming in PySpark

Extract data into PySpark

```
import pyspark.sql

spark = pyspark.sql.SparkSession.builder.getOrCreate()

spark.read.jdbc("jdbc:postgresql://localhost:5432/pagila",
                "customer",
                properties={"user":"repl","password":"password"})
```

An example: join

A new ratings table

customer_id	film_id	rating
1	2	1
2	1	5
2	2	3
...	...	...

The customer table

customer_id	first_name	last_name	...
1	Jane	Doe	...
2	Joe	Doe	...
...	...	...	...

`customer_id` **overlaps with ratings table**

An example: join (PySpark)

```
customer_df # PySpark DataFrame with customer data
ratings_df # PySpark DataFrame with ratings data

# Groupby ratings
ratings_per_customer = ratings_df.groupBy("customer_id").mean("rating")

# Join on customer ID
customer_df.join(
  ratings_per_customer,
  customer_df.customer_id==ratings_per_customer.customer_id
)
```

Analytics or applications databases

Analytics

- Aggregate queries
- Online analytical processing (OLAP)

Applications

- Lots of transactions
- Online transaction processing (OLTP)

Column- and row-oriented

Analytics

- Column-oriented

name	diameter (cm)	weight (g)
apple	10	100
grape	2	10

- Queries about subset of columns
- Parallelization

Applications

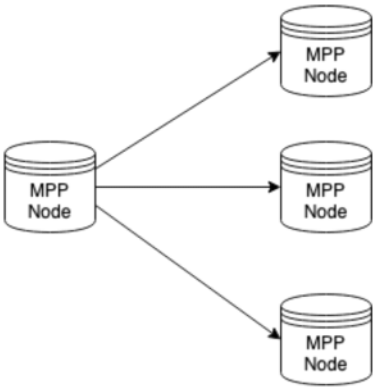
- Row-oriented

name	diameter (cm)	weight (g)
apple	10	100
grape	2	10

- Stored per record
- Added per transaction
- E.g. adding customer is fast

MPP Databases

Massively Parallel Processing Databases



- Amazon Redshift
- Azure SQL Data Warehouse
- Google BigQuery

An example: Redshift

Load from file to columnar storage format

```
# Pandas .to_parquet() method
df.to_parquet("./s3://path/to/bucket/customer.parquet")
# PySpark .write.parquet() method
df.write.parquet("./s3://path/to/bucket/customer.parquet")
```

```
COPY customer
FROM 's3://path/to/bucket/customer.parquet'
FORMAT as parquet
...
```

Load to PostgreSQL

`pandas.to_sql()`

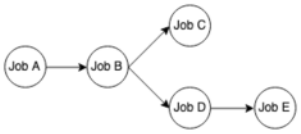
```
# Transformation on data
recommendations = transform_find_recommendatins(ratings_df)

# Load into PostgreSQL database
recommendations.to_sql("recommendations",
                       db_engine,
                       schema="store",
                       if_exists="replace")
```

The ETL function

```
def extract_table_to_df(tablename, db_engine):
  return pd.read_sql("SELECT * FROM {}".format(tablename), db_engine)

def split_columns_transform(df, column, pat, suffixes):
  # Converts column into str and splits it on pat...

def load_df_into_dwh(film_df, tablename, schema, db_engine):
  return pd.to_sql(tablename, db_engine, schema=schema, if_exists="replace")

db_engines = { ... } # Needs to be configured
def etl():
  # Extract
  film_df = extract_table_to_df("film", db_engines["store"])
  # Transform
  film_df = split_columns_transform(film_df, "rental_rate", ".", ["_dollar", "_cents"])
  # Load
  load_df_into_dwh(film_df, "film", "store", db_engines["dwh"])
```

Airflow refresher

- Workflow scheduler
- Python
- DAGs



- Tasks defined in operators (e.g. `BashOperator`)

Scheduling with DAGs in Airflow

```
from airflow.models import DAG

dag = DAG(dag_id="sample",
          ...,
          schedule_interval="0 0 * * *")
```

```
# cron
# .------------------------ minute           (0 - 59)
# | .---------------------- hour             (0 - 23)
# | | .-------------------- day of the month (1 - 31)
# | | | .------------------ month            (1 - 12)
# | | | | .---------------- day of the week  (0 - 6)
# * * * * * <command>

# Example
0 * * * * # Every hour at the 0th minute
```

There are multiple ways of defining the interval, but the most common one is using a cron expression.

That is a string which represents a set of times.

It's a string containing 5 characters, separated by a space.

The leftmost character describes minutes, then hours, day of the month, month, and lastly, day of the week.

cf. https://crontab.guru

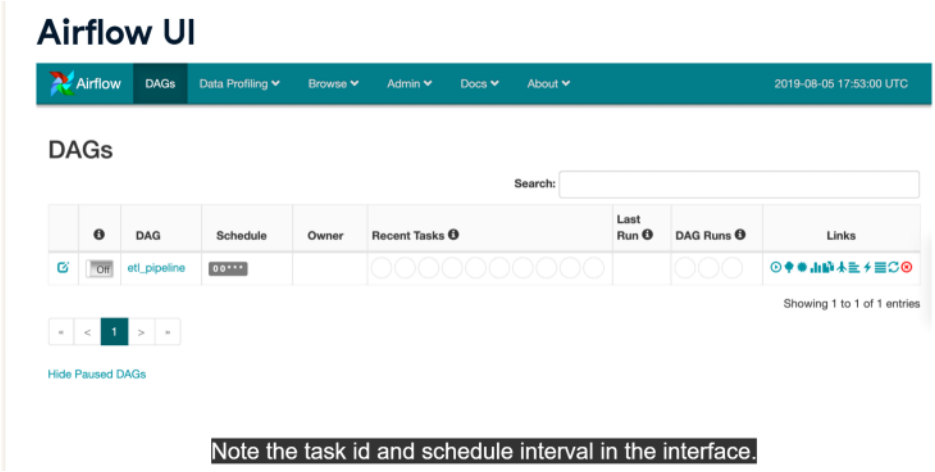
The DAG definition file

```
from airflow.models import DAG
from airflow.operators.python_operator import PythonOperator

dag = DAG(dag_id="etl_pipeline",
          schedule_interval="0 0 * * *")

etl_task = PythonOperator(task_id="etl_task",
                          python_callable=etl,
                          dag=dag)

etl_task.set_upstream(wait_for_this_task)
```

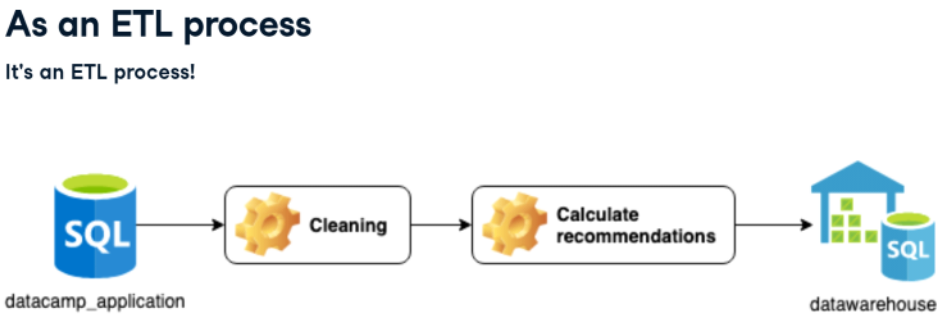
The DAG definition file

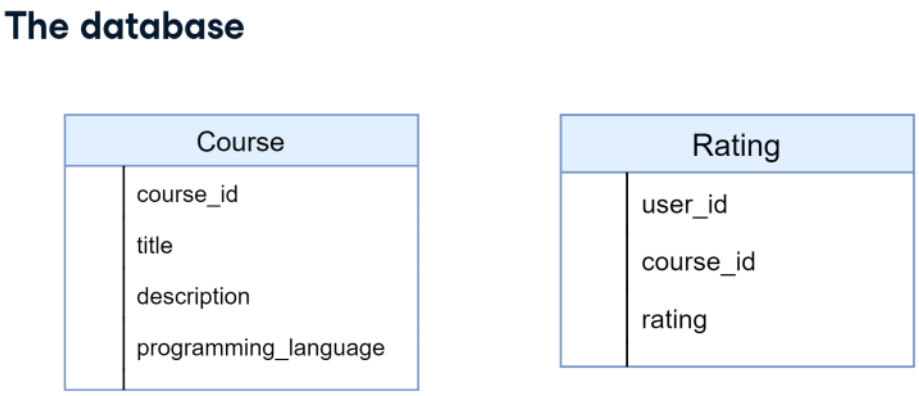
```
from airflow.models import DAG
from airflow.operators.python_operator import PythonOperator

...

etl_task.set_upstream(wait_for_this_task)
```

Saved as `etl_dag.py` in `~/airflow/dags/`

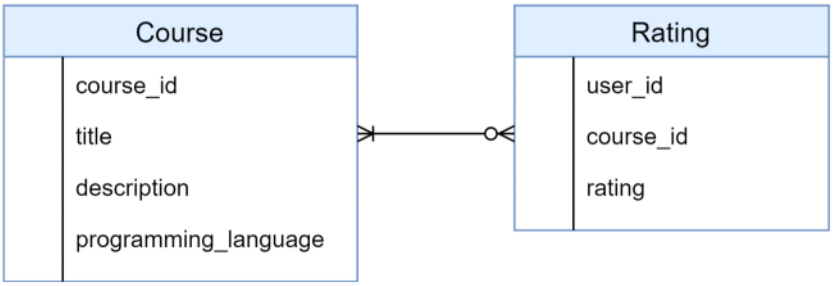


Case Study: DataCamp





The database relationship

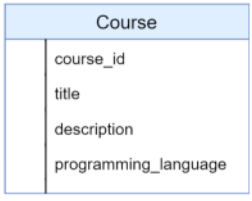


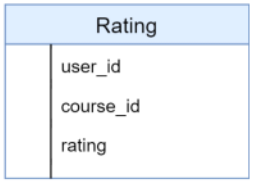
The recommendations table

user_id	course_id	rating
1	1	4.8
1	74	4.78
1	21	4.5
2	32	4.9

The estimated rating of a course the user hasn't taken yet.

Common sense transformation





Recommendations

user_id	course_id	rating
1	1	4.8
1	74	4.78
1	21	4.5
2	32	4.9

Use the right programming language

Rating

user_id	course_id	programming_language	rating
1	1	r	4.8
1	74	sql	4.78
1	21	sql	4.5
1	32	python	4.9

Recommend SQL course for user with id 1

Our recommendation transformation

- Use technology that user has rated most
- Don't recommend courses that user already rated
- Recommend three highest rated courses from remaining combinations

Rating

user_id	course_id	programming_language	rating
1	12	sql	4.78
1	52	sql	4.5
1	32	r	4.9

Recommend three highest rated SQL courses which are not 12 and 52.

What you've done so far

- Extract using `extract_course_data()` and `extract_rating_data()`
- Clean up using NA using `transform_fill_programming_language()`
- Average course ratings per course: `transform_avg_rating()`
- Get eligible user and course id pairs: `transform_courses_to_recommend()`
- Calculate the recommendations: `transform_recommendations()`

Loading to Postgres

- Use the calculations in data products
- Update daily
- Example use case: sending out e-mails with recommendations

The loading phase

```
recommendations.to_sql(
    "recommendations",
    db_engine,
    if_exists="append",
)
```

```
def etl(db_engines):
    # Extract the data
    courses = extract_course_data(db_engines)
    rating = extract_rating_data(db_engines)
    # Clean up courses data
    courses = transform_fill_programming_language(courses)
    # Get the average course ratings
    avg_course_rating = transform_avg_rating(rating)
    # Get eligible user and course id pairs
    courses_to_recommend = transform_courses_to_recommend(
        rating,
        courses,
    )
    # Calculate the recommendations
    recommendations = transform_recommendations(
        avg_course_rating,
        courses_to_recommend,
    )
    # Load the recommendations into the database
    load_to_dwh(recommendations, db_engine))
```

Creating the DAG

```
from airflow.models import DAG
from airflow.operators.python_operator import PythonOperator

dag = DAG(dag_id="recommendations",
          scheduled_interval="0 0 * * *")
task_recommendations = PythonOperator(
    task_id="recommendations_task",
    python_callable=etl,
)
```