# Introduction to Data Engineering

Introduction to Data Engineering

Black Raven (James Ng)
26 Mar 2021 · 31 min read

⏱ 4 hours ▷ 15 Videos ‹› 57 Exercises 👥 52,104 Participants 🗄 4,100 XP

This is a memo to share what I have learnt in Introduction to Data Engineering, capturing the learning objectives as well as my personal notes. The course is taught by Vincent Vankrunkelsven from DataCamp, and it includes 4 chapters:
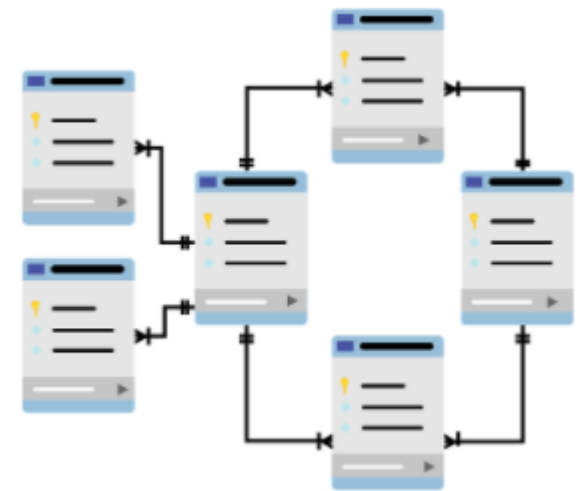
Chapter 1. Introduction to Data Engineering
Chapter 2. Data engineering toolbox
Chapter 3. Extract, Transform and Load (ETL)
Chapter 4. Case Study: Data Engineering at DataCamp

Have you heard people talk about data engineers and wonder what it is they do? Do you know what data engineers do but you're not sure how to become one yourself? This course is the perfect introduction. It touches upon all things you need to know to streamline your data processing.

This introductory course will give you enough context to start exploring the world of data engineering. It's perfect for people who work at a company with several data sources and don't have a clear idea of how to use all those data sources in a scalable way. Be the first one to introduce these techniques to your company and become the company star employee.

# Chapter 1. Introduction to Data Engineering

In this first chapter, you will be exposed to the world of data engineering! Explore the differences between a data engineer and a data scientist, get an overview of the various tools data engineers use and expand your understanding of how cloud technology plays a role in data engineering.

## What is data engineering?

Common real world situations
- Data is scattered
- Not available in an orderly fashion
- Not optimized for analysis
- Legacy code is causing corrupt data

Aim of data engineer:
- Gather data from different sources
- Optimize database scheme so it becomes faster to query
- Remove corrupted data
- Processing large amounts of data
- Use of clusters of machines



Data engineer
= an engineer that develops, constructs, tests, and maintains architectures such as databases and large-scale processing systems
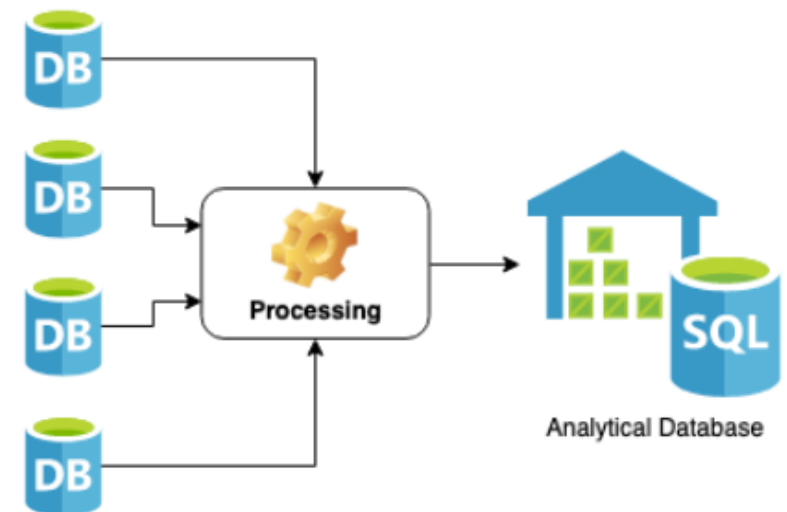
A data engineer's responsibilities
- Ingest data from different sources
- Optimize databases for analysis
- Remove corrupted data
- Develop, construct, test and maintain data architectures

Data engineers deliver:
- the correct data
- in the right form
- to the right people
- as efficiently as possible

Big data growth
- Sensors and devices
- Social media
- Enterprise data
- VoIP (voice communication, multimedia sessions)

The 5 Vs

- Volume – how much?
- Variety – what kind?
- Velocity – how frequent?
- Veracity – how accurate?
- Value – how useful?



Annual Size of the Global Datasphere — 175 ZB
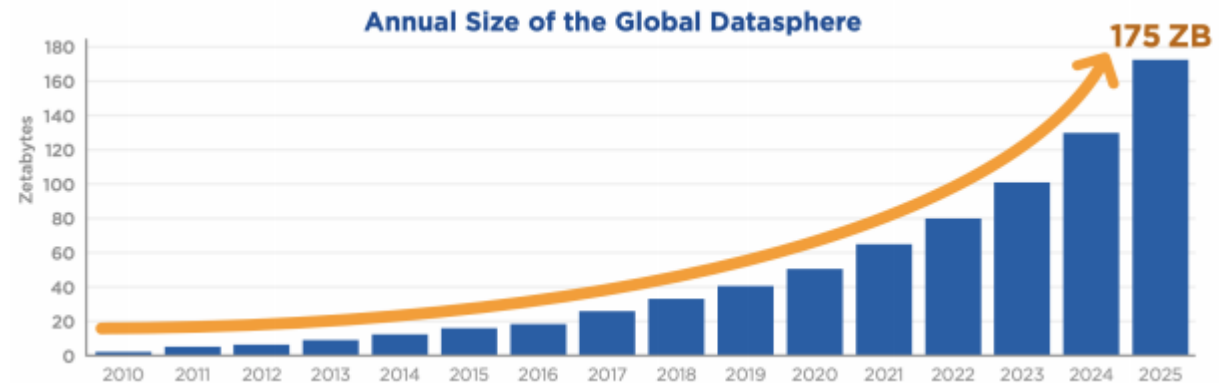
**Data Engineer**

- 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

Data Engineer has a deep understanding of cloud technology, whereas Data Scientist has a deep understanding of the business itself.

# Tasks of the data engineer

The video presented several tasks of the data engineer. You saw that there are some differences between the tasks of data scientists and the tasks of data engineers.

Below are three essential tasks that need to happen in a data-driven company. Can you find the one that best fits the job of a data engineer?

○ Apply a statistical model to a large dataset to find outliers.

◉ Set up scheduled ingestion of data from the application databases to an analytical database.

○ Come up with a database schema for an application.

This is very clearly the task of a data engineer.

# Data engineer or data scientist?

You now know that there's a difference between the job of the data engineer and the data scientist. Remember the definition of a data engineer:

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

Now let's see if you can separate tasks for the data engineer from the data scientist's tasks.

| Data Engineer | | Data Scientist | |
|---|---|---|---|
| Set up processes to bring together data | ☰ | Mining data for patterns | ☰ |
| Develop scalable data architecture | ☰ | Predictive models using machine learning | ☰ |
| Streamline data acquisition | ☰ | Statistical modeling | ☰ |
| Clean corrupt data | ☰ | Monitor business processes | ☰ |
| Cloud technology | ☰ | Clean statistical outliers in data | ☰ |

You classified all cards correctly!

# Data engineering problems

For this exercise, imagine you work in a medium-scale company that hosts an online market for pet toys. As the company is growing, there are unmistakably some technical growing pains.

As the first data engineer, you observe some problems and have to decide where you're best suited to be of help.

◉ Data scientists are querying the online store databases directly and slowing down the functioning of the application since it's using the same database.

◯ Harmful product recommendations are affecting the sales numbers of the online store.

◯ The online store is slow because the application's database server doesn't have enough memory.

# Tools of the data engineer

Database = computer system that holds large amounts of data
Applications rely on databases to provide certain functionality. For example, in an online store, a database holds product data like prices or amount in stock.
Other databases hold data specifically for analyses.

Data engineers use tools that can help them quickly process data. Processing data might be necessary to **clean** or **aggregate** data or to **join** it together from different sources. Typically, huge amounts of data have to be processed. That is where parallel processing comes into play. Instead of processing the data on one computer, data engineers use clusters of machines to process the data.
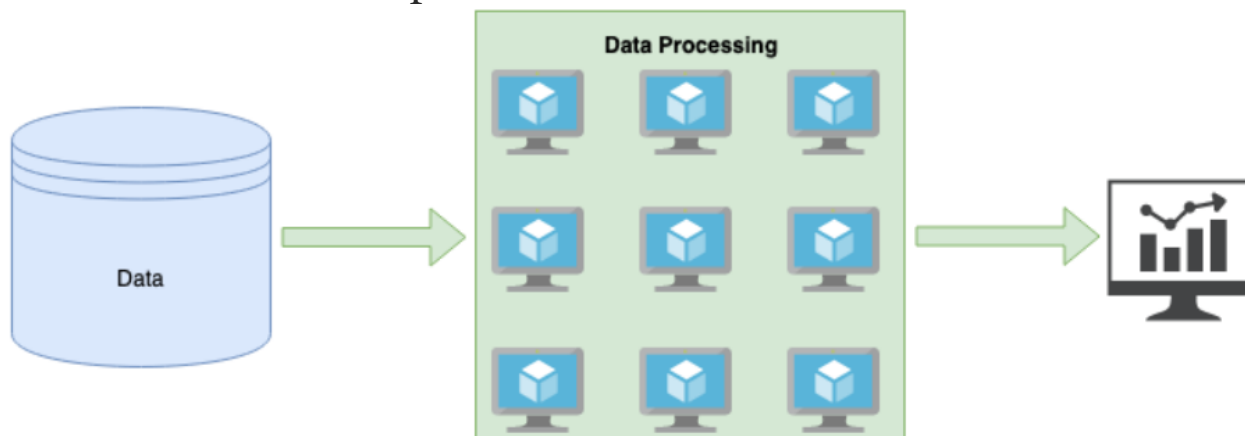
Often, these tools make an abstraction of the underlying architecture and have a simple API.
A good data engineer understands these abstractions and knows their limitations.
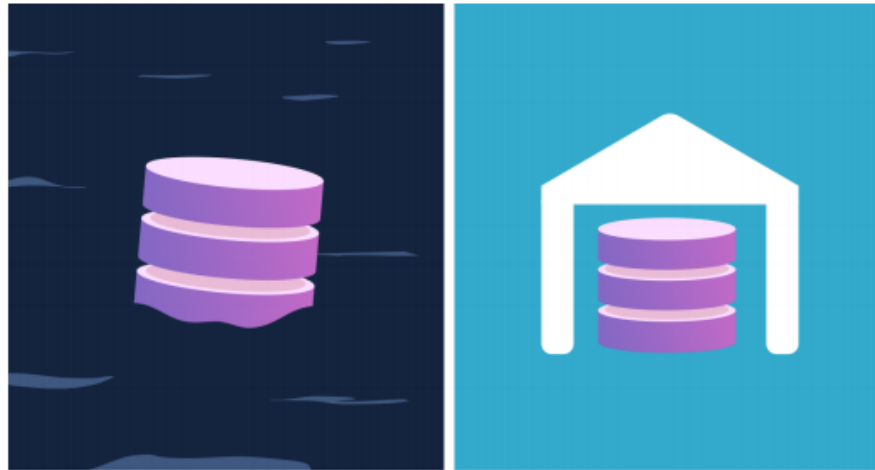
| Data Lakes | Data Warehouses |
| --- | --- |
| Stores all the raw data | Specific data for specific use |
| Can be petabytes (1 million GBs) | Relative smaller in size than Data Lakes |
| Stores all data structures | Stores mainly structured data |
| Cost-effective | More costly to update |
| Difficult to analyse<br>Requires an up-to-date data catalogue | Optimised for data analytics |
| Used by data scientists | Used by data analysts and business analysts |
| Big data, real-time analytics | Ad-noc, read-only queries |

**Data Catalog** for data lakes
- What is the source of this data?
- Where is this data used?
- Who is the owner of the data?
- How often is this data updated?
- Good practice in terms of data governance
- Ensures reproducibility

No catalogue ➔ data swamp

Good practice for any data storage solution
- Reliability
- Autonomy
- Scalability
- Speed

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

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

print(outliers.count())
```



**Scheduling tools** help to make sure data moves from one place to another at the correct time, with a specific interval. Data engineers make sure these processing jobs run in a timely fashion and that they run in the right order .
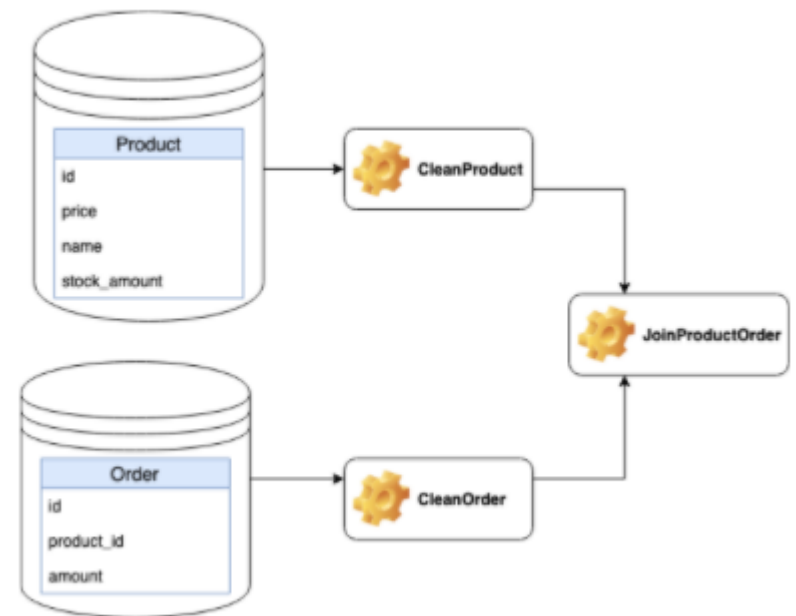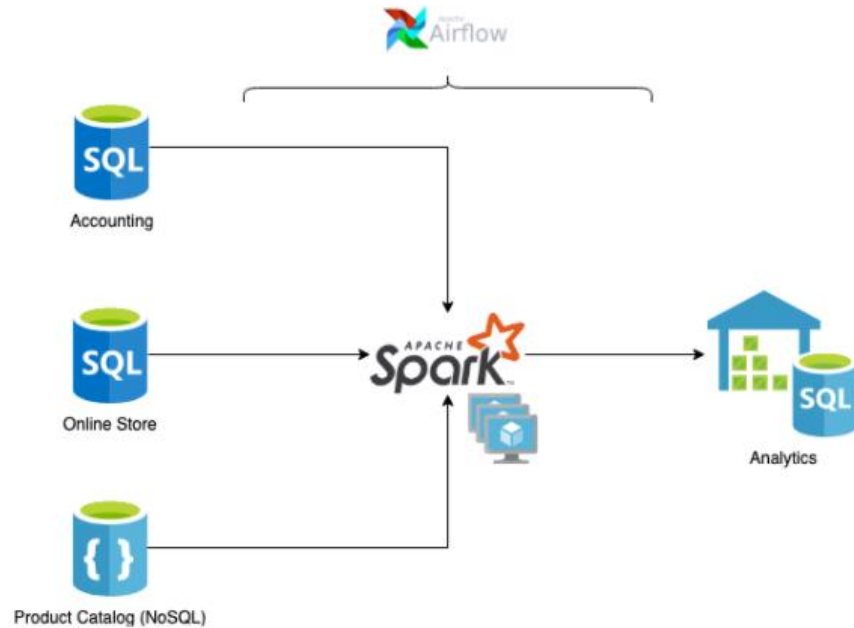
## Databases



MySQL

PostgreSQL

## Scheduling



Apache Airflow

## Processing



Apache Spark

HIVE

OOZIE

Bash tool: cron

To sum everything up, you can think of the data engineering pipeline through this diagram. It extracts all data through connections with several databases, transforms it using a cluster computing framework like Spark, and loads it into an analytical database. Also, everything is scheduled to run in a specific order through a

scheduling framework like Airflow. A small side note here is that the sources can be external APIs or other file formats too.



# Kinds of databases

In the video, you saw that databases are an essential tool for data engineers. The data engineer's workflow often begins and ends with databases.

However, databases are not always the first step. Sometimes, data has to be pulled in from external APIs or raw files. Can you identify the database in the schematics?
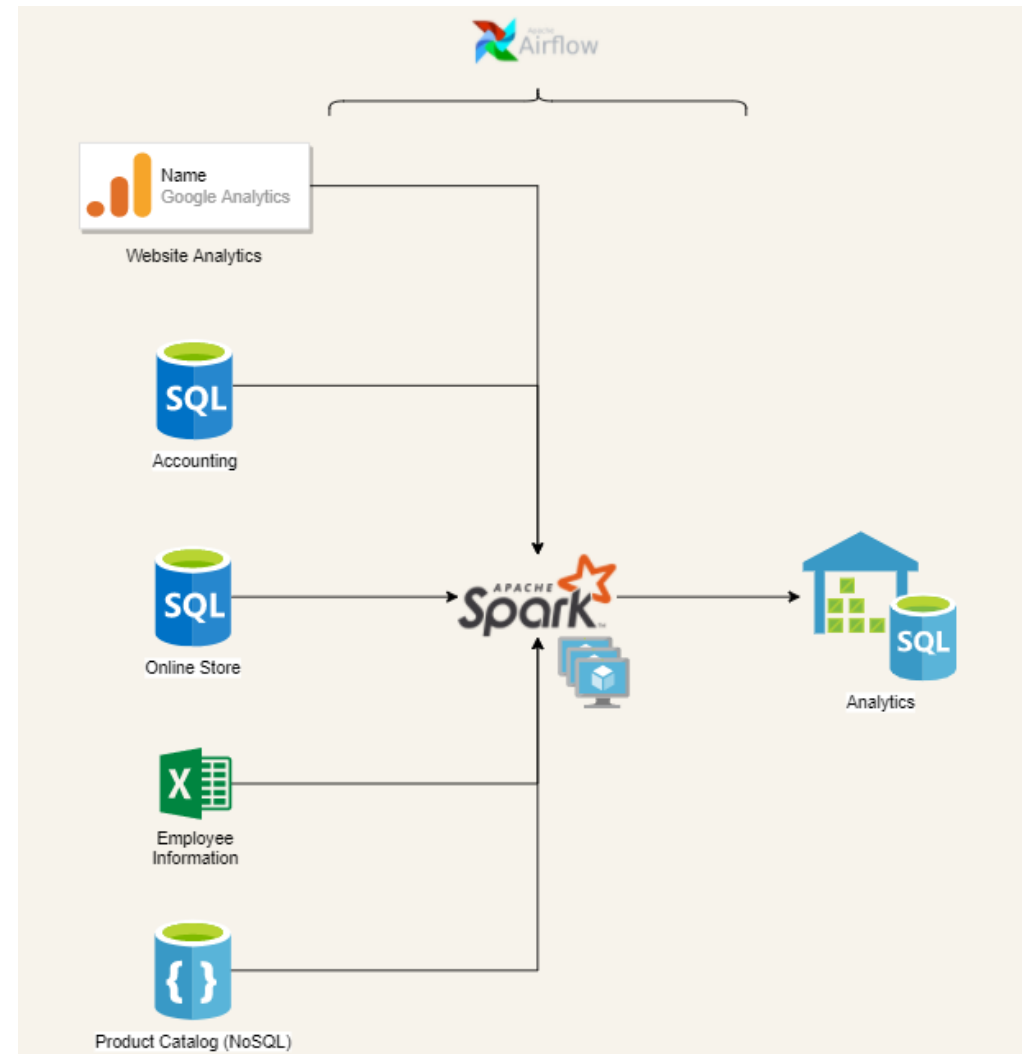
○ All database nodes are on the left.

○ All nodes on the left and the analytics node on the right are databases.

◉ Accounting, Online Store, Product Catalog, and Analytics are databases.

The analytics database is also a SQL database. All but the product catalog are SQL databases.

# Processing tasks

Data engineers often have to join, clean, or organize data before loading it into a destination analytics database. This is done in the data processing, or data transformation step.

In the diagram, this is the intermediate step. Can you select the **most correct** statement about data processing?

○ Data processing is often done on a single, very powerful machine.

○ Data processing is distributed over clusters of virtual machines.

○ Data processing is often very complicated because you have to manually distribute workload over several computers.

Answer: Data processing is distributed over clusters of virtual machines.
This makes it very easy to scale. If things are slowing down, assign some more virtual machines to the job.

# Scheduling tools

The last piece of the puzzle is the scheduler. In this example, Apache Airflow is used. You could see the scheduler as the glue of a data engineering system, holding each small piece together and organizing how they work together.

Do you know which one is NOT a responsibility of the scheduler?

○ Make sure jobs run in a specific order and all dependencies are resolved correctly.

○ Make sure the jobs run at midnight UTC each day.

○ Scale up the number of nodes when there's lots of data to be processed.

Answer: Scale up the number of nodes when there's lots of data to be processed.
The scheduler does not scale up processing power. That's the job of the processing tool.

# Cloud providers

Data engineers are heavy users of the cloud, as data processing often runs on clusters of machines.

In the past, companies that relied on data processing owned their own data center. You can imagine racks of servers, ready to be used. The electrical bill and maintenance were also at the company's cost. Moreover, companies needed to be able to provide enough processing power for peak moments. That also meant that at quieter times, much of the processing power remained unused. It's this **waste of resources** that made cloud computing so appealing. In the cloud, you use the resources you need, at the time you need them. You can see that once these cloud services came to be, many companies moved to the cloud as a way of cost optimization. Apart from the costs of maintaining data centers, another reason for using cloud computing is **database reliability**.

Three big players in the cloud provider market:
1. **Amazon Web Services** or AWS. Think about the last few websites you visited. Chances are AWS hosts at least a few of them. Back in 2017, AWS had an outage, it reportedly 'broke' the internet. That's how big AWS is. AWS took up 32% of the market share in 2018.
2. **Microsoft Azure** is the second big player and took 17% of the market.
3. **Google Cloud** held 10% of the market in 2018.

These big players provide three types of services: Storage, Computation, and Databases.

**32% market share in 2018**

- Storage

**17% market share in 2018**

- Computation

**10% market share in 2018**

- Databases.

**Storage**
Storage services allow you to upload files of all types to the cloud. In an online store for example, you could upload your product images to a storage service. Storage services are typically very cheap since they don't provide much functionality other than storing the files reliably.
AWS hosts **S3** as a storage service. Azure has **Blob Storage**, and Google has **Cloud Storage**.

**Computation**
Computation services allow you to perform computations on the cloud, example host web servers. These services are usually flexible, and you can start or stop virtual machines as needed.
AWS has **EC2** as a computation service, Azure has **Virtual Machines**, and Google has **Compute Engine**.

**Databases**
Cloud providers host databases.
For SQL databases, AWS has **RDS**. Azure has **SQL Database**, and Google has **Cloud SQL**.

| | File storage | Computation | Databases |
|---|---|---|---|
| **aws** 32.4% | AWS S3 | AWS EC2 | AWS RDS |
| **Microsoft Azure** 17.6% | Azure Blob Storage | Azure Virtual Machines | Azure SQL Database |
| **Google Cloud** 6% | Google Cloud Storage | Google Compute Engine | Google Cloud SQL |

# Why cloud computing?

In the video you saw the benefits of using cloud computing as opposed to self-hosting data centers. Can you select the most **correct** statement about cloud computing?

○ Cloud computing is always cheaper.

◉ The cloud can provide you with the resources you need, when you need them.

○ On premise machines give me full control over the situation when things break.

This property of cloud computing is also called cloud elasticity.

# Big players in cloud computing

You've seen a few examples of cloud service providers. You learned about the big three, who together took up more than 50% of the market share in 2018. They took 32%, 17%, and 10% of the market share.

Can you order the big three correctly?

Amazon Web Services (AWS)

Microsoft Azure

Google Cloud Platform
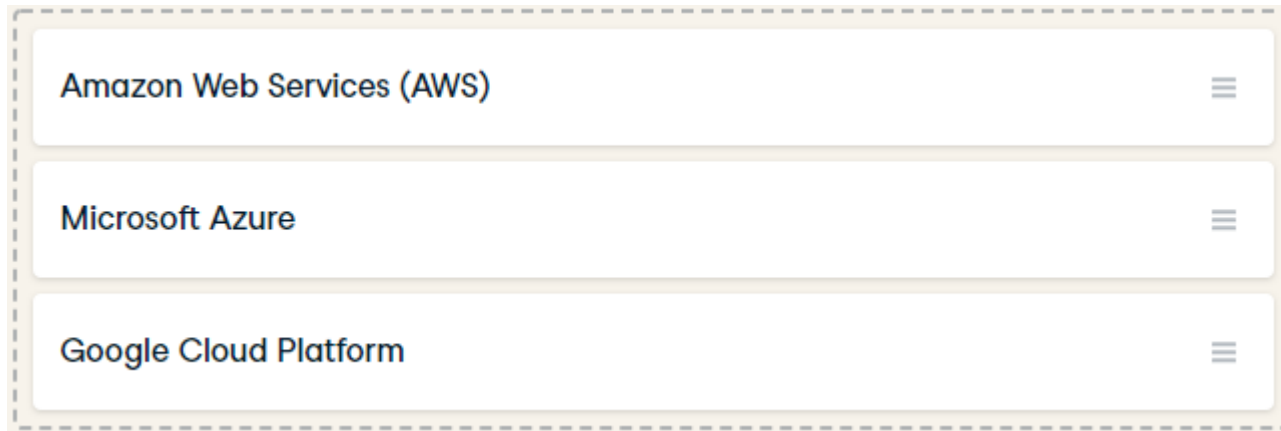
You'll see these names pop up in the data engineering world from time to time, so keep an eye out for them!

# Cloud services

There are a bunch of cloud services that all have different use cases. It can be hard to keep track of all of them.

Here's an overview of the ones you've seen:

- **Storage**: use the cloud to store unstructured data, like files.
- **Compute**: use virtual machines in the cloud to do complex calculations.
- **Database**: use databases, typically SQL, in the cloud to hold structured data.

In addition, different cloud providers have different names for each of these services…

| Storage | | Compute | | Database | |
|---|---|---|---|---|---|
| Amazon S3 | ≡ | Amazon EC2 | ≡ | Amazon RDS | ≡ |
| Azure Blob Storage | ≡ | Azure Virtual Machines | ≡ | Azure SQL Database | ≡ |
| Google Cloud Storage | ≡ | Google Compute Engine | ≡ | Google Cloud SQL | ≡ |

All of these different services can be a bit overwhelming at first, but you'll learn the difference quickly and learn to appreciate the variations.

# Chapter 2. Data engineering toolbox

Now that you know the primary differences between a data engineer and a data scientist, get ready to explore the data engineer's toolbox! Learn in detail about different types of databases data engineers use, how parallel computing is a cornerstone of the data engineer's toolkit, and how to schedule data processing jobs using scheduling frameworks.

## Databases

What is a database?
- Holds data
- Organizes data
- Retrieve/Search data through Database Management System (DBMS)

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

**Databases**

**File storage**

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

- Less organized
- Simple, less added functionality

**Structured:** database schema

- Relational database

**Semi-structured**

- JSON

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

**Unstructured:** schemaless, more like files

- Videos, photos

**SQL**
- Tables
- Database schema
- Relational databases

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

There are several types of NoSQL databases and they are not all unstructured. Two highly used NoSQL database types are key-value stores like Redis or document databases like MongoDB. In key-value stores, the values are simple. Typical use cases are caching or distributed configuration. Values in a document database are structured or semi-structured objects, for example, a JSON object.

The database schema
A **schema** describes the structure and relations of a database.

```
-- 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")
);
```
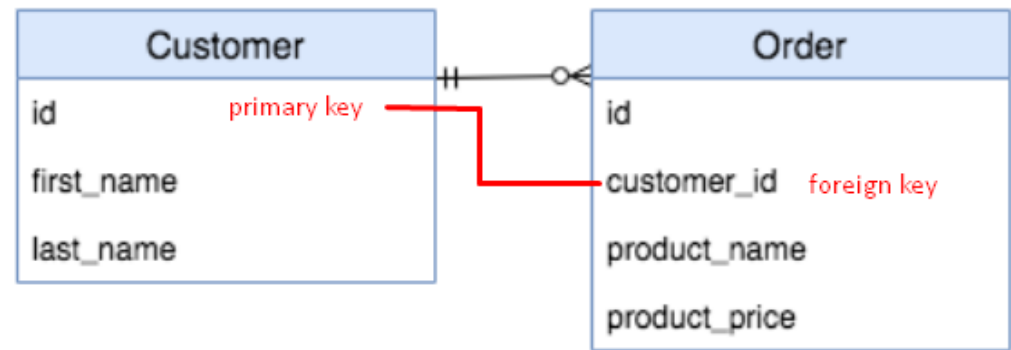


```
-- Join both tables on foreign key
SELECT * FROM "Customer"
INNER JOIN "Order"
ON "customer_id" = "Customer"."id";
```

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

The SQL statements on the left create the tables of the schema.

SQL: **Star schema**
In data warehousing, a schema you'll see often is the star schema. A lot of analytical databases like Redshift have optimizations for these kinds of schemas.
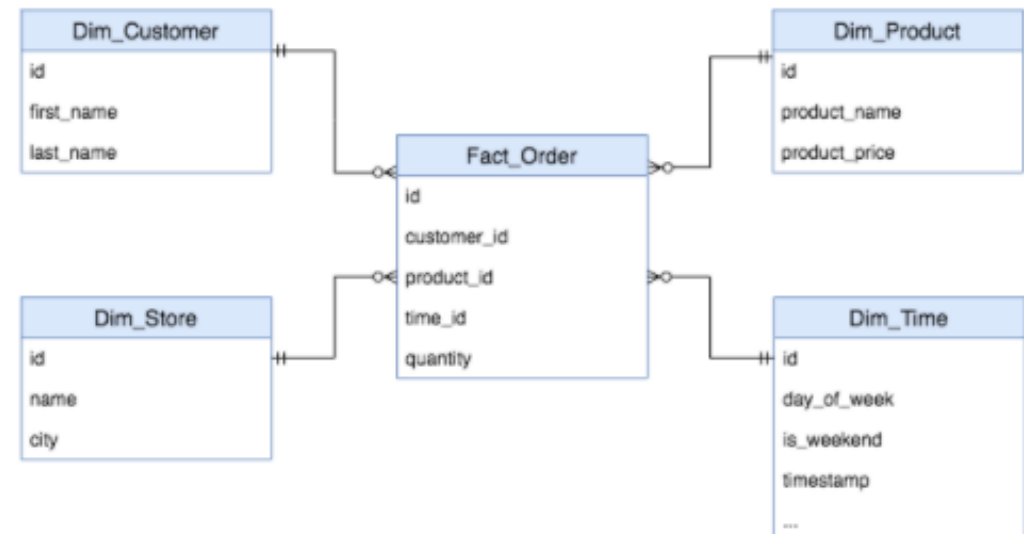
Wikipedia: https://en.wikipedia.org/wiki/Star_schema
According to Wikipedia, "the star schema consists of one or more fact tables referencing any number of dimension tables."
**Fact** tables contain records that represent things that happened in the world, like orders.
**Dimension** tables hold information on the world itself, like customer names or product prices.

## SQL vs NoSQL

In the video on Databases, you saw the difference between SQL and NoSQL. You saw that SQL uses database schemas to define relations and that NoSQL allows you to work with less structured data.

You might also remember some open-source examples of SQL and NoSQL databases and their use cases.

| SQL | | NoSQL | |
|---|---|---|---|
| Always has a database schema | ≡ | Caching layer in distributed web server | ≡ |
| Customer data in a store's database | ≡ | Can be schemaless | ≡ |
| PostgreSQL | ≡ | MongoDB | ≡ |
| MySQL | ≡ | | |

## The database schema

By now, you know that SQL databases always have a database schema. In the video on databases, you saw the following diagram:
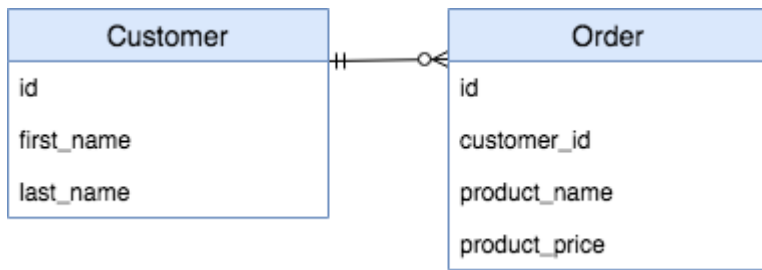
A PostgreSQL database is set up in your local environment, which contains this database schema. It's been filled with some example data. You can use `pandas` to query the database using the `read_sql()` function. You'll have to pass it a database engine, which has been defined for you and is called `db_engine`.

The `pandas` package imported as `pd` will store the query result into a DataFrame object, so you can use any DataFrame functionality on it after fetching the results from the database.

```python
# Complete the SELECT statement
data = pd.read_sql("""
SELECT first_name, last_name FROM "Customer"
ORDER BY last_name, first_name
""", db_engine)

# Show the first 3 rows of the DataFrame
print(data.head(3))

# Show the info of the DataFrame
print(data.info())
```

```
<script.py> output:
     first_name last_name
  0     Connagh    Bailey
  1       Brook     Bloom
  2         Ann    Dalton
  <class 'pandas.core.frame.DataFrame'>
  RangeIndex: 7 entries, 0 to 6
  Data columns (total 2 columns):
  first_name    7 non-null object
  last_name     7 non-null object
  dtypes: object(2)
  memory usage: 192.0+ bytes
  None
```

You now know how to query a SQL database from Python using pandas.

# Joining on relations

You've used the following diagram in the previous exercise:
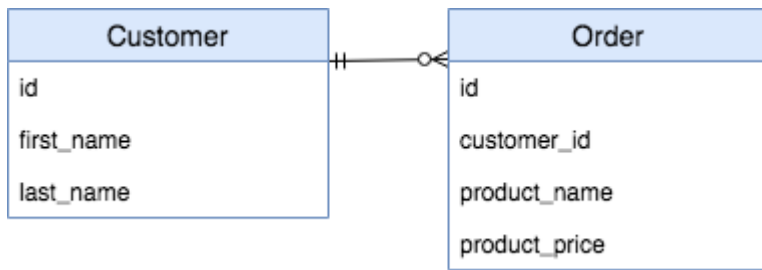
You've learned that you can use the `read_sql()` function from `pandas` to query the database. The real power of SQL is the ability to join information from multiple tables quickly. You do this by using the `JOIN` statement.

When joining two or more tables, `pandas` puts all the columns of the query result into a DataFrame.

```python
# Complete the SELECT statement
data = pd.read_sql("""
SELECT * FROM "Customer"
INNER JOIN "Order"
ON "Order"."customer_id"="Customer"."id"
""", db_engine)

# Show the id column of data
print(data.id)
```

```
<script.py> output:
        id  id
    0    1   1
    1    2   2
    2    1   3
    3    5   4
    4    3   5
```

In the IPython Shell, you can see that data.id outputs 2 columns. This is often a source of error, so a better strategy here would be to select specific columns or use the AS keyword in SQL to rename columns.

# Star schema diagram

Which of the following images is a star schema?

**Possible Answers**

○ A

● B

○ None

## Diagram (A.)

**Customer**
- id
- first_name
- last_name

**Order**
- id
- customer_id
- product_name
- product_price

**(A.)**

## Diagram (B.)

**Dim_Customer**
- id
- first_name
- last_name

**Dim_Product**
- id
- product_name
- product_price

**Fact_Order**
- id
- customer_id
- product_id
- time_id
- quantity

**Dim_Store**
- id
- name
- city

**Dim_Time**
- id
- day_of_week
- is_weekend
- timestamp
- ...

**(B.)**
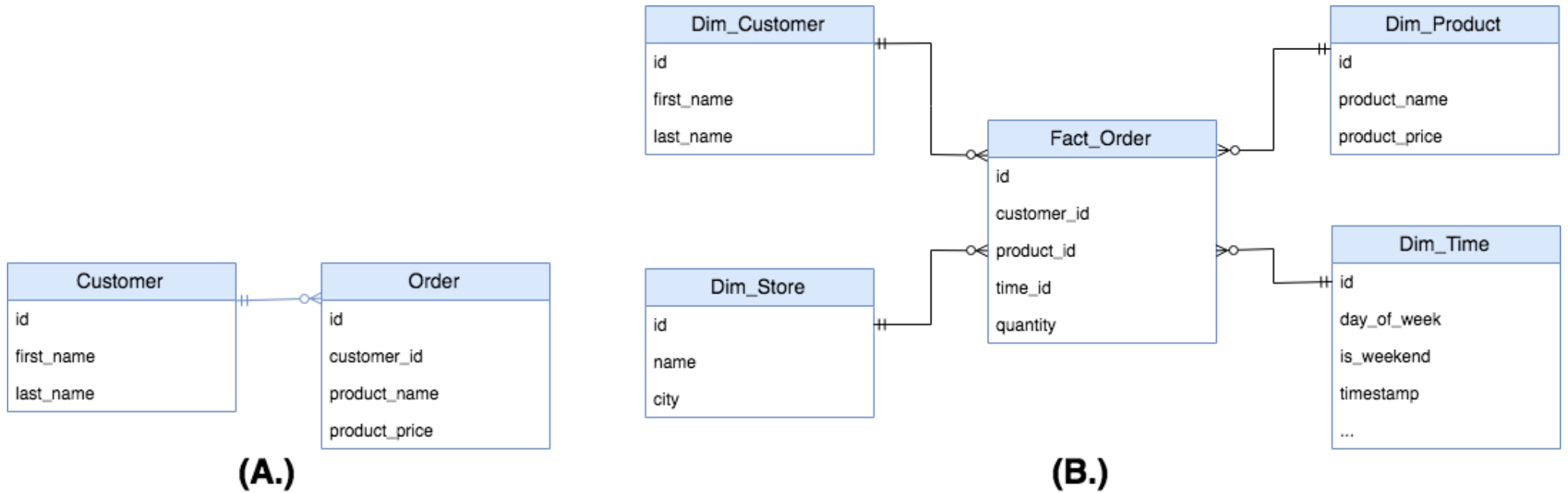
# What is parallel computing

Parallel computing forms the basis of almost all modern data processing tools:
- memory
- processing power

When big data processing tools perform a processing task, they split it up into several smaller subtasks.

The processing tools then distribute these subtasks over several computers. These are usually commodity computers, which means they are widely available and relatively inexpensive.
Individually, all of the computers would take a long time to process the complete task. However, since all the computers work in parallel on smaller subtasks, the task in its whole is done faster.
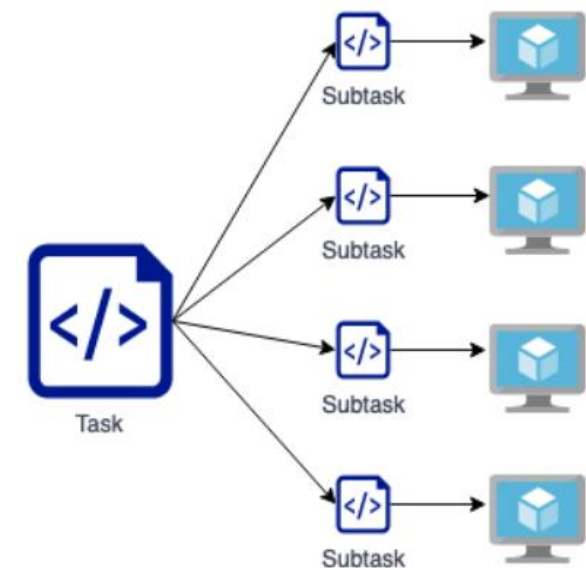
Benefit of parallel computing (having multiple processing units) for big data
- Having the extra processing power
- Instead of needing to load all of the data in one computer's memory, you can partition the data and load the subsets into memory of different computers. That means the memory footprint per computer is relatively small, and the data can fit in the memory closest to the processor, the RAM.

Risks of parallel computing
- **Communication overhead** - splitting a task into subtask and merging the results of the subtasks back into one final result requires some communication between processes. This can become a bottleneck if the processing requirements are not substantial, or if you have too little processing units.
- **Parallel slowdown** - due to the overhead, the speed does not increase linearly.

*Parallel slowdown:*
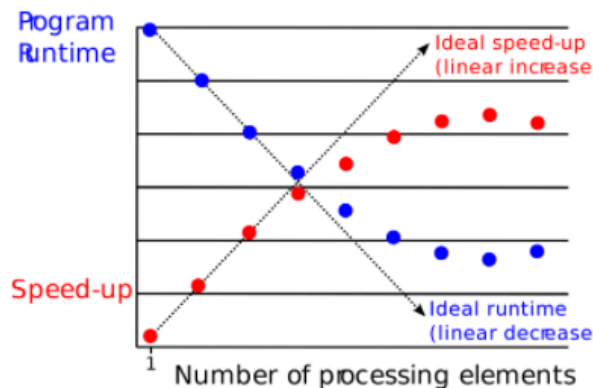


ID ... Year ... Age
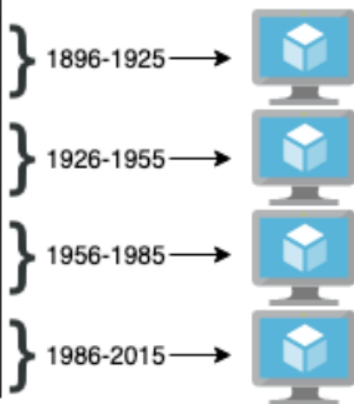
To split the task into smaller subtasks
In this example, the *average age* calculation for each group of years is as a subtask. You can achieve that through **'groupby.'** Then, you distribute all of these subtasks over the four processing units. This example illustrates roughly how the first distributed algorithms like Hadoop **MapReduce** work, the difference being the processing units are distributed over several machines.

`multiprocessing.Pool`

```python
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`

```python
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()
```

# Why parallel computing?

You've seen the benefits of parallel computing. However, you've also seen it's not the silver bullet to fix all problems related to computing.

Which of these statements is **not** correct?

- ● Parallel computing can be used to speed up any task.

- ○ Parallel computing can optimize the use of multiple processing units.

- ○ Parallel computing can optimize the use of memory between several machines.

You can't split every task successfully into subtasks. Additionally, some tasks might be too small to benefit from parallel computing due to the communication overhead.

# From task to subtasks

For this exercise, you will be using parallel computing to apply the function `take_mean_age()` that calculates the average athlete's age in a given year in the Olympics events dataset. The DataFrame `athlete_events` has been loaded for you and contains amongst others, two columns:

- `Year`: the year the Olympic event took place
- `Age`: the age of the Olympian

You will be using the `multiprocessor.Pool` API which allows you to distribute your workload over several processes. The function `parallel_apply()` is defined in the sample code. It takes in as input the function being applied, the grouping used, and the number of cores needed for the analysis. Note that the `@print_timing` decorator is used to time each operation.

```python
# Function to apply a function over multiple cores
@print_timing
def parallel_apply(apply_func, groups, nb_cores):
    with Pool(nb_cores) as p:
        results = p.map(apply_func, groups)
    return pd.concat(results)

# Parallel apply using 1 core
parallel_apply(take_mean_age, athlete_events.groupby('Year'), 1)

# Parallel apply using 2 cores
parallel_apply(take_mean_age, athlete_events.groupby('Year'), 2)

# Parallel apply using 4 cores
parallel_apply(take_mean_age, athlete_events.groupby('Year'), 4)
```

For educational purposes, we've used a little trick here to make sure the parallelized version runs faster. In reality, using parallel computing wouldn't make sense here since the computations are simple and the dataset is relatively small. Communication overhead costs the multiprocessing version its edge!

# Using a DataFrame

In the previous exercise, you saw how to split up a task and use the low-level python `multiprocessing.Pool` API to do calculations on several processing units.

It's essential to understand this on a lower level, but in reality, you'll never use this kind of APIs. A more convenient way to parallelize an apply over several groups is using the `dask` framework and its abstraction of the `pandas` DataFrame, for example.

The `pandas` DataFrame, `athlete_events`, is available in your workspace.

```python
import dask.dataframe as dd

# Set the number of pratitions
athlete_events_dask = dd.from_pandas(athlete_events, npartitions = 4)

# Calculate the mean Age per Year
print(athlete_events_dask.groupby('Year').Age.mean().compute())
```

```
<script.py> output:
    Year
    1896    23.580645       2002    25.916281
    1900    29.034031       2004    25.639515
    1904    26.698150       2006    25.959151
    1906    27.125253       2008    25.734118
    1908    26.970228       2010    26.124262
    1912    27.538620       2012    25.961378
    1920    29.290978       2014    25.987324
                            2016    26.207919
                            Name: Age, dtype: float64
```

Using the DataFrame abstraction makes parallel computing easier. You'll see in the next video that this abstraction is often used in the big data world, for example in Apache Spark.

# Parallel computation frameworks

Hadoop is a collection of open source projects, maintained by the Apache Software Foundation.

**HDFS**
HDFS is a distributed file system. It's similar to the file system you have on your computer, the only difference being the files reside on multiple different computers. HDFS has been essential in the big data world, and for parallel computing by extension. Nowadays, cloud-managed storage systems like Amazon S3 often replace HDFS.
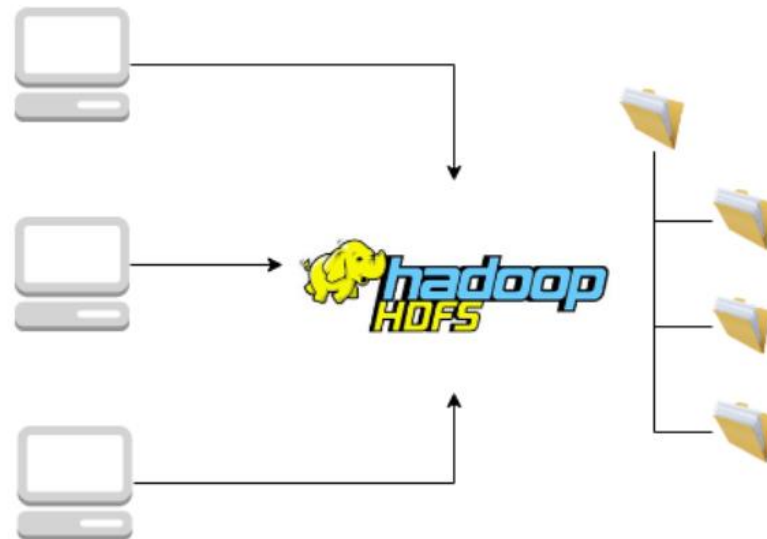
## MapReduce

MapReduce was one of the first popularized big-data processing paradigms. The program splits tasks into subtasks, distributing the workload and data between several processing units. For MapReduce, these processing units are several computers in the cluster. MapReduce had its flaws; one of it was that it was hard to write these MapReduce jobs. Many software programs popped up to address this problem, and one of them was Hive.
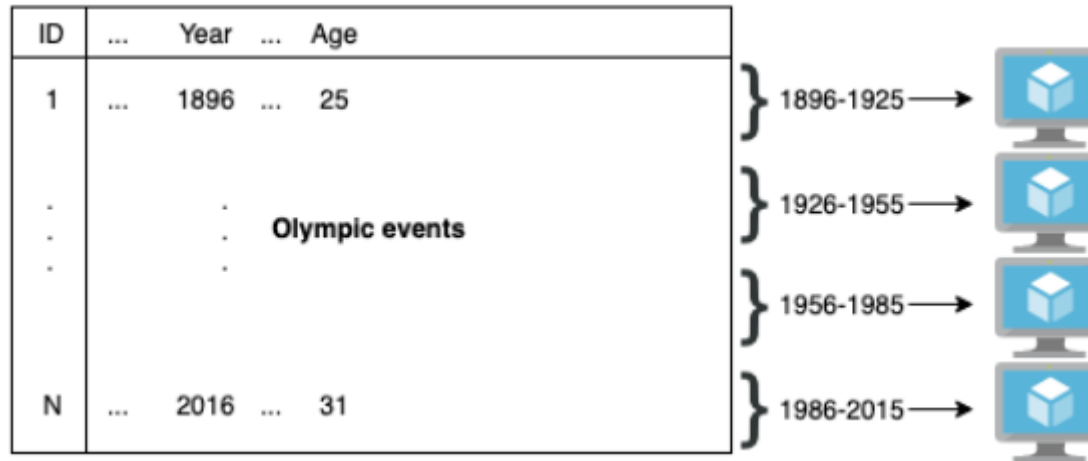


## Hive

Hive is a layer on top of the Hadoop ecosystem that makes data from several sources queryable in a structured way using Hive's SQL variant: Hive SQL. Although MapReduce was initially responsible for running the Hive jobs, it now integrates with several other data processing tools.

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



This Hive query selects the average age of the Olympians per Year they participated.
This query looks indistinguishable from a regular SQL query. However, behind the curtains, this query is transformed into a job that can operate on a cluster of computers.

**Spark** is another parallel computation framework that distributes data processing tasks between clusters of computers. While MapReduce-based systems tend to need expensive disk writes between jobs, Spark tries to keep as much processing as possible in memory. In that sense, Spark was also an answer to the limitations of MapReduce. The disk writes of MapReduce were especially limiting in interactive exploratory data analysis, where each step builds on top of a previous step.

Spark's architecture relies on **resilient distributed datasets**, or RDDs. This is a data structure that maintains data which is distributed between multiple nodes. Unlike DataFrames, RDDs don't have named columns. From a conceptual perspective, you can think of RDDs as *lists of tuples*. We can do two types of operations on these data structures: transformations, like map or filter, and actions, like count or first. Transformations result in transformed RDDs, while actions result in a single result.

**PySpark**
When working with Spark, people typically use a programming language interface like PySpark. PySpark is the Python interface to spark. There are interfaces to Spark in other languages, like R or Scala, as well. PySpark hosts a DataFrame abstraction, which means that you can do operations very similar to pandas DataFrames. PySpark and Spark take care of all the complex parallel computing operations.

```
# Load the dataset into athlete_events_spark first

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

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

# Spark, Hadoop and Hive

You've encountered quite a few open source projects in the previous video. There's Hadoop, Hive, and PySpark. It's easy to get confused between these projects.
They have a few things in common: they are all currently maintained by the Apache Software Foundation, and they've all been used for massive parallel processing. Can you spot the differences?

| Hadoop | PySpark | Hive |
|---|---|---|
| Collection of open source packages for Big Data ⊘ | Python interface for the Spark framework ⊘ | Initially used Hadoop MapReduce ⊘ |
| MapReduce is part of it ⊘ | Uses DataFrame abstraction ⊘ | Is built from the need to use structured queries for parallel processing ⊘ |
| HDFS is part of it ⊘ | | |

# A PySpark groupby

You've seen how to use the `dask` framework and its DataFrame abstraction to do some calculations. However, as you've seen in the video, in the big data world Spark is probably a more popular choice for data processing.

In this exercise, you'll use the PySpark package to handle a Spark DataFrame. The data is the same as in previous exercises: participants of Olympic events between 1896 and 2016.

The Spark Dataframe, `athlete_events_spark` is available in your workspace. The methods you're going to use in this exercise are:

- `.printSchema()`: helps print the schema of a Spark DataFrame.
- `.groupBy()`: grouping statement for an aggregation.
- `.mean()`: take the mean over each group.
- `.show()`: show the results.

```
# Print the type of athlete_events_spark
print(type(athlete_events_spark))

# Print the schema of athlete_events_spark
print(athlete_events_spark.printSchema())

# Group by the Year, and find the mean Age
print(athlete_events_spark.groupBy('Year').mean('Age'))

# The same, but now show the results
print(athlete_events_spark.groupBy('Year').mean('Age').show())
```

```
<script.py> output:
    <class 'pyspark.sql.dataframe.DataFrame'>
    root
     |-- ID: integer (nullable = true)
     |-- Name: string (nullable = true)
     |-- Sex: string (nullable = true)
     |-- Age: integer (nullable = true)
     |-- Height: string (nullable = true)
     |-- Weight: string (nullable = true)
     |-- Team: string (nullable = true)
     |-- NOC: string (nullable = true)
     |-- Games: string (nullable = true)
     |-- Year: integer (nullable = true)
     |-- Season: string (nullable = true)
     |-- City: string (nullable = true)
     |-- Sport: string (nullable = true)
     |-- Event: string (nullable = true)
     |-- Medal: string (nullable = true)

    None
```

```
DataFrame[Year: int, avg(Age): double]
+----+------------------+
|Year|          avg(Age)|
+----+------------------+
|1896|23.580645161290324|
|1924|28.373324544056253|
|2006|25.959151072569604|
|1908|26.970228384991845|
|1952|26.161546085232903|
|1956| 25.926673567977915|
|1988|24.079431552931485|
|1994|24.422102596580114|
|1968|24.248045555448314|
|2014|25.987323655694134|
|1904| 26.69814995131451|
|2004|25.639514989213716|
|1932| 32.58207957204948|
|1996|24.915045018878885|
|1998|25.163197335553704|
|1960|25.168848457954294|
```

However, we're not finished just yet. In the next exercise, you'll see more about how to set up PySpark locally.

# Running PySpark files

In this exercise, you're going to run a PySpark file using `spark-submit`. This tool can help you submit your application to a spark cluster.

For the sake of this exercise, you're going to work with a local Spark instance running on 4 threads. The file you need to submit is in `/home/repl/spark-script.py`. Feel free to read the file:

```
cat /home/repl/spark-script.py
```

You can use `spark-submit` as follows:

```
spark-submit \
  --master local[4] \
  /home/repl/spark-script.py
```

What does this output? *Note that it may take a few seconds to get your results.*

```
repl:~$ cat /home/repl/spark-script.py
from pyspark.sql import SparkSession


if __name__ == "__main__":
    spark = SparkSession.builder.getOrCreate()
    athlete_events_spark = (spark
        .read
        .csv("/home/repl/datasets/athlete_events.csv",
            header=True,
            inferSchema=True,
            escape='"'))

    athlete_events_spark = (athlete_events_spark
        .withColumn("Height",
                athlete_events_spark.Height.cast("integer")))

    print(athlete_events_spark
        .groupBy('Year')
        .mean('Height')
        .orderBy('Year')
        .show())
```

○ An error.

○ A DataFrame with average Olympian heights by year.

○ A DataFrame with Olympian ages.

```
repl:~$ spark-submit --master local[4] /home/repl/spark-script.py
Picked up _JAVA_OPTIONS: -Xmx512m
Picked up _JAVA_OPTIONS: -Xmx512m
21/03/28 03:36:20 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes w
here applicable
+----+------------------+
|Year|       avg(Height)|
+----+------------------+
|1896|  172.7391304347826|
|1900|176.63793103448276|
|1904|  175.7887323943662|
|1906|178.20622568093384|
|1908|177.54315789473685|
|1912|  177.4479889042996|
|1920|  175.7522816166884|
|1924|174.96303901437372|
|1928|  175.1620512820513|
|1932|174.22011541632315|
|1936|  175.7239932885906|
|1948|176.17279726261762|
|1952|174.13893967093236|
|1956|173.90096798212957|
|1960|173.14128595600675|
|1964|   173.448573701557|
|1968|  173.9458648072826|
|1972|174.56536284096757|
|1976|174.92052773737794|
|1980|175.52748832195473|
+----+------------------+
only showing top 20 rows

None
```

Answer: A DataFrame with average Olympian heights by year.

## Workflow scheduling frameworks

The task of the workflow scheduling framework is to orchestrate these jobs of pulling data from databases usng parallel computing frameworks like Spark.
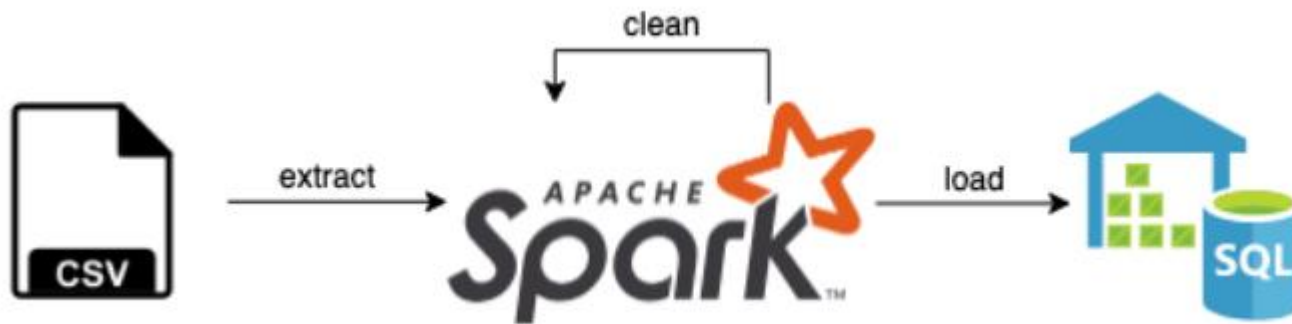
You can write a Spark job that pulls data from a CSV file, filters out some corrupt records, and loads the data into a SQL database ready for analysis. You could run jobs manually, or automatically using cron scheduling, the Linux tool. There are dependencies between the 3 jobs, ie, have to wait for 'extract' and 'clean' to complete before executing 'load'.

A great way to visualize these dependencies is through **Directed Acyclic Graphs**, or DAGs. A DAG is a set of nodes that are connected by directed edges. There are no cycles in the graph, which means that no path following the directed edges sees a specific node more than once.
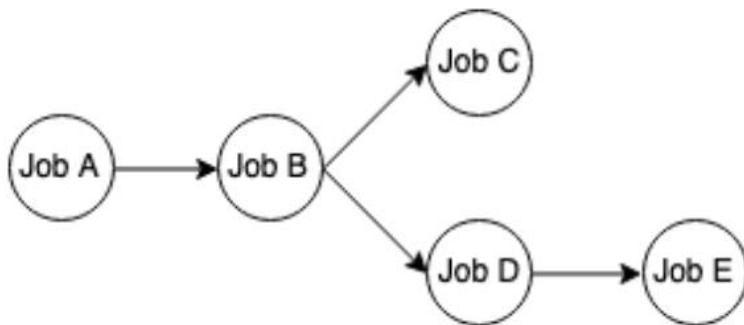


In this example, Job A needs to happen first, then Job B, which enables Job C and D and finally Job E. As you can see, it feels natural to represent this kind of workflow in a DAG. The jobs represented by the DAG can then run in a daily schedule, for example.

Tools for scheduling DAGs
- the Linux tool, cron
- full-fledged solution: Spotify's Luigi, which allows for the definition of DAGs for complex pipelines.
- **Apache Airflow**: growing out to be the de-facto workflow scheduling framework.

Apache Airflow example:
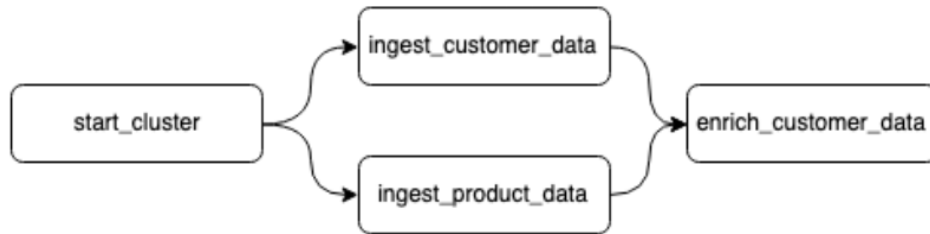


```python
# Create the DAG object
dag = DAG(dag_id="example_dag", ..., schedule_interval="0 * * * *")


# 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)
```

# Airflow, Luigi and cron

In the video, you saw several tools that can help you with the scheduling of your Spark jobs. You saw the limitations of cron and how it has led to the development of frameworks like Airflow and Luigi. There's a lot of useful features in Airflow, but can you select the feature from the list below which is also provided by cron?

○ You can program your workflow in Python.

○ You can use a directed acyclic graph as a model for dependency resolving.

◉ You have exact control over the time at which jobs run.          ← this core functionality is also offered by cron.

# Airflow DAGs

In Airflow, a pipeline is represented as a Directed Acyclic Graph or DAG. The nodes of the graph represent tasks that are executed. The directed connections between nodes represent dependencies between the tasks.

Representing a data pipeline as a DAG makes much sense, as some tasks need to finish before others can start. You could compare this to an assembly line in a car factory. The tasks build up, and each task can depend on previous tasks being finished. A fictional DAG could look something like this:
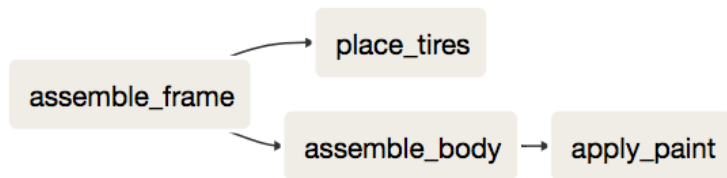


Assembling the frame happens first, then the body and tires and finally you paint. Let's reproduce the example above in code.

```python
# Create the DAG object
dag = DAG(dag_id="car_factory_simulation",
          default_args={"owner": "airflow","start_date": airflow.utils.dates.days_ago(2)},
          schedule_interval="0 * * * *")

# Task definitions
assemble_frame = BashOperator(task_id="assemble_frame", bash_command='echo "Assembling frame"', dag=dag)
place_tires = BashOperator(task_id="place_tires", bash_command='echo "Placing tires"', dag=dag)
assemble_body = BashOperator(task_id="assemble_body", bash_command='echo "Assembling body"', dag=dag)
apply_paint = BashOperator(task_id="apply_paint", bash_command='echo "Applying paint"', dag=dag)

# Complete the downstream flow
assemble_frame.set_downstream(place_tires)
assemble_frame.set_downstream(assemble_body)
assemble_body.set_downstream(apply_paint)
```

Of course, there are many other kinds of operators you can use in real life situations.

# Chapter 3. Extract, Transform and Load (ETL)

Having been exposed to the toolbox of data engineers, it's now time to jump into the bread and butter of a data engineer's workflow! With ETL, you will learn how to extract raw data from various sources, transform this raw data into actionable insights, and load it into relevant databases ready for consumption!

## Extract

means extracting data from persistent storage, which is not suited for data processing, into memory. Persistent storage could be a file on Amazon S3, for example, or a SQL database. It's the necessary stage before we can start transforming the data. The sources to extract from vary.
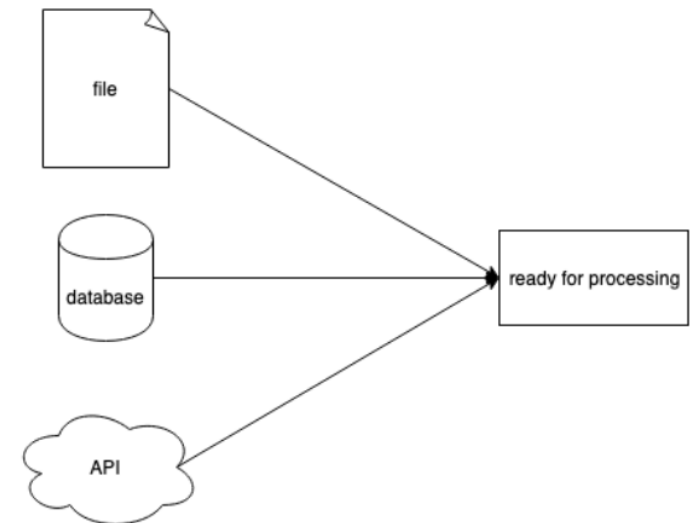
## Extract from text files

### Unstructured

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

### Flat files

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

```
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 ....
```

```
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
```

**Extract from JSON (JavaScript Object Notation)**

- JavaScript Object Notation

- Semi-structured

- Atomic
  - `number`
  - `string`
  - `boolean`
  - `null`

- Composite
  - `array`
  - `object`

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

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

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

```
value_1
```

## Data on the Web
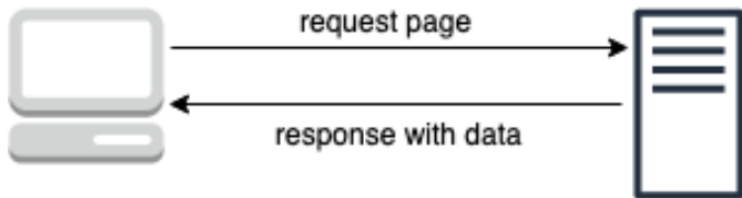


On the web, most communication happens to something called 'requests.' You can look at a request as a 'request for data.' A request gets a response. For example, if you browse Google in your web browser, your browser requests the content of the Google home page. Google servers respond with the data that makes up the page.

Some web servers serve data in a JSON data format. We call these servers **API**s or application programming interfaces. Example:
- Twitter - hosts an API that provides us with information on tweets in JSON format, a structured way of querying their data

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

- Hackernews API - can use the Python package, `requests` to request an API. We will use the `.get()` method and pass an URL. The resulting response object has a built-in helper method called `.json()` to parse the incoming JSON and transform it into a Python object.

```python
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': .... }

Extraction from existing **application databases**
Most applications, like web services, need a database to back them up and persist data. Databases that applications like web services use, are typically optimized for having lots of transactions. A transaction typically changes or inserts rows, or records, in the database.

For example, let's say we have a customer database. Each row, or record, represents data for one specific customer. A transaction could add a customer to the database, or change their address. These kinds of transactional databases are called OLTP, or **online transaction processing**. They are typically row-oriented, in which the system adds data per rows. In contrast, databases optimized for analysis are called OLAP, or **online analytical processing**. They are often column-oriented.

| Applications databases | Analytical databases |
|---|---|
| • Transactions | • OLAP |
| • Inserts or changes | • Column-oriented |
| • OLTP | |
| • Row-oriented | |

## Connection string/URI

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

## Use in Python

```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)
```

# Data sources

In the previous video you've learned about three ways of extracting data:

- Extract from text files, like `.txt` or `.csv`
- Extract from APIs of web services, like the Hacker News API
- Extract from a database, like a SQL application database for customer data

We also briefly touched upon row-oriented databases and OLTP. Can you select the statement about these topics which is **NOT true**?

○ OLTP means the system is optimized for transactions.

◉ APIs mostly use raw text to transfer data.

○ Row-oriented databases and OLTP go hand-in-hand.

APIs mostly use a more structured form of data, for example JSON.

# Fetch from an API

In the last video, you've seen that you can extract data from an API by sending a request to the API and parsing the response which was in JSON format. In this exercise, you'll be doing the same by using the `requests` library to send a request to the **Hacker News API**.

**Hacker News** is a social news aggregation website, specifically for articles related to computer science or the tech world in general. Each post on the website has a JSON representation, which you'll see in the response of the request in the exercise.

```python
import requests

# Fetch the Hackernews post
resp = requests.get("https://hacker-news.firebaseio.com/v0/item/16222426.json")

# Print the response parsed as JSON
print(resp.json())

# Assign the score of the test to post_score
post_score = resp.json()["score"]
print(post_score)
```
```
<script.py> output:
    {'by': 'neis', 'descendants': 0, 'id': 16222426, 'score': 17, 'time': 1516800333, 'title': 'Duolingo-Style Learning for Data Science: DataCamp for M
    17
```

You might have noticed that the response is not a real response. This is a result of DataCamp's sandboxing environment. However, you can interract with the real API using the exact same method.

# Read from a database

In this exercise, you're going to extract data that resides inside tables of a local PostgreSQL database. The data you'll be using is the **Pagila example database**. The database backs a fictional DVD store application, and educational resources often use it as an example database. You'll be creating and using a function that extracts a database table into a `pandas` DataFrame object. The tables you'll be extracting are:

- `film`: the films that are rented out in the DVD store.
- `customer`: the customers that rented films at the DVD store.

In order to connect to the database, you'll have to use a PostgreSQL connection URI, which looks something like this:

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

```python
# Function to extract table to a pandas DataFrame
def extract_table_to_pandas(tablename, db_engine):
    query = "SELECT * FROM {}".format(tablename)
    return pd.read_sql(query, db_engine)


# Connect to the database using the connection URI
connection_uri = "postgresql://repl:password@localhost:5432/pagila"
db_engine = sqlalchemy.create_engine(connection_uri)


# Extract the film table into a pandas DataFrame
extract_table_to_pandas("film", db_engine)


# Extract the customer table into a pandas DataFrame
extract_table_to_pandas("customer", db_engine)
```

The connection URI can also be a link to an external database. Keep in mind that in this case, pulling in full tables can result in loads of network traffic.

# Transform

| 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

## Example: Split email to username and domain

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

```python
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

```python
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**

```
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
)
```

Note how we set the matching keys of the two data frames when joining the data frames.

# Splitting the rental rate

In the video exercise, you saw how to use pandas to split the email address column of the `film` table in order to extract the users' domain names. Suppose you would want to have a better understanding of the rates users pay for movies, so you decided to divide the `rental_rate` column into dollars and cents.

In this exercise, you will use the same techniques used in the video exercises to do just that! The `film` table has been loaded into the `pandas` DataFrame `film_df`. Remember, the goal is to split up the `rental_rate` column into dollars and cents.

```
# Get the rental rate column as a string
rental_rate_str = film_df.rental_rate.astype("str")

# Split up and expand the column
rental_rate_expanded = rental_rate_str.str.split(".", expand=True)

# Assign the columns to film_df
film_df = film_df.assign(
    rental_rate_dollar=rental_rate_expanded[0],
    rental_rate_cents=rental_rate_expanded[1],
)
```

Pandas is a reasonable tool to transform your data with for small dataset sizes.

# Prepare for transformations

As mentioned in the video, before you can do transformations using PySpark, you need to get the data into the Spark framework. You saw how to do this using PySpark. Can you choose the **correct** code?

**(A)**

```
spark.read.jdbc("jdbc:postgresql://repl:password@localhost:5432/pagila",
                "customer")
```

**Possible Answers**

**(B)**

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

○ A

◉ B

○ C

**(C)**

```
spark.read.jdbc("jdbc:postgresql://repl:password@localhost:5432/pagila/customer")
```

The table name is the second argument and properties the third.

# Joining with ratings

In the video exercise, you saw how to use transformations in PySpark by joining the `film` and `ratings` tables to create a new column that stores the average rating per customer. In this exercise, you're going to create more synergies between the `film` and `ratings` tables by using the same techniques you learned in the video exercise to calculate the average rating for every film.

The PySpark DataFrame with films, `film_df` and the PySpark DataFrame with ratings, `rating_df`, are available in your workspace.

```
# Use groupBy and mean to aggregate the column
ratings_per_film_df = rating_df.groupBy('film_id').mean('rating')

# Join the tables using the film_id column
film_df_with_ratings = film_df.join(
```

```
    ratings_per_film_df,
    film_df.film_id==ratings_per_film_df.film_id
)

# Show the 5 first results
print(film_df_with_ratings.show(5))
```

```
<script.py> output:
    +-------+--------------+-----------------+------------+-----------+------------------+---------------+-----------+------+----------------+------+--
    |film_id|         title|      description|release_year|language_id|original_language_id|rental_duration|rental_rate|length|replacement_cost|rating|
    +-------+--------------+-----------------+------------+-----------+------------------+---------------+-----------+------+----------------+------+--
    |      1|ACADEMY DINOSAUR|A Epic Drama of a...|        2006|          1|              null|              6|       0.99|    86|           20.99|    PG|2(
    |      2|  ACE GOLDFINGER|A Astounding Epis...|        2006|          1|              null|              3|       4.99|    48|           12.99|     G|2(
    |      3|ADAPTATION HOLES|A Astounding Refl...|        2006|          1|              null|              7|       2.99|    50|           18.99| NC-17|2(
    |      4|AFFAIR PREJUDICE|A Fanciful Docume...|        2006|          1|              null|              5|       2.99|   117|           26.99|     G|2(
    |      5|     AFRICAN EGG|A Fast-Paced Docu...|        2006|          1|              null|              6|       2.99|   130|           22.99|     G|2(
    +-------+--------------+-----------------+------------+-----------+------------------+---------------+-----------+------+----------------+------+--
    only showing top 5 rows
```

You successfully joined data from two separate sources.

# Loading

| Analytics database | Application database |
|---|---|
| complex aggregate queries frequently run on analytical databases | application databases have lots of transactions per second |

- Aggregate queries
- Online analytical processing (OLAP)

- Lots of transactions
- Online transaction processing (OLTP)

| store data per column, analytical queries to be mostly about a small subset of columns in a table | store data per record, easy to add new rows in small transactions |
|---|---|

- Column-oriented

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

- Queries about subset of columns

- Parallelization

- 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



**Massively parallel processing** (MPP) databases are column-oriented databases optimized for analytics, that run in a distributed fashion. Specifically, this means that queries are not executed on a single compute node, but rather split into subtasks and distributed among several nodes. Famous managed examples of these are Amazon Redshift, Azure SQL Data Warehouse, or Google BigQuery.
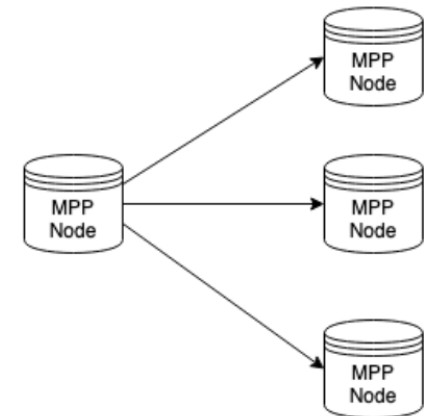
To load data into Amazon Redshift, an excellent way to do this would be to write files to S3, AWS's file storage service, and send a copy query to Redshift. Typically, MPP databases load data best from files that use a columnar storage format. CSV files would not be a good option, for example. We often use a file format called parquet for this purpose. There are helper functions to write this kind of files in several packages.

```python
# 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")
```

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

Load to PostgreSQL
To load the result of the transformation phase into a PostgreSQL database. For example, your data pipeline could extract from a rating table, transform it to find recommendations and load them into a PostgreSQL database, ready to be used by a recommendation service.

`pandas.to_sql()`

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


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

# OLAP or OLTP

You saw that there's a difference between OLAP and OLTP operations. A small recap:

- OLAP: Online analytical processing
- OLTP: Online transaction processing

It's essential to use the right database for the right job. There's a list of statements below. Find the most appropriate statement that is **true**.

○ Typically, analytical databases are column-oriented.

○ Massively parallel processing (MPP) databases are usually column-oriented.

○ Databases optimized for OLAP are usually not great at OLTP operations.

○ Analytical and application databases have different use cases and should be separated if possible.

○ None of the above.

◉ All of the above.

# Writing to a file

In the video, you saw that files are often loaded into a MPP database like Redshift in order to make it available for analysis.

The typical workflow is to write the data into columnar data files. These data files are then uploaded to a storage system and from there, they can be copied into the data warehouse. In case of Amazon Redshift, the storage system would be S3, for example.

The first step is to write a file to the right format. For this exercises you'll choose the **Apache Parquet** file format.

There's a PySpark DataFrame called `film_sdf` and a pandas DataFrame called `film_pdf` in your workspace.

```
# Write the pandas DataFrame to parquet
film_pdf.to_parquet("films_pdf.parquet")

# Write the PySpark DataFrame to parquet
film_sdf.write.parquet("films_sdf.parquet")
```

# Load into Postgres

In this exercise, you'll write out some data to a PostgreSQL data warehouse. That could be useful when you have a result of some transformations, and you want to use it in an application.

For example, the result of a transformation could have added a column with film recommendations, and you want to use them in your online store.

There's a `pandas` DataFrame called `film_pdf` in your workspace.

As a reminder, here's the structure of a connection URI for `sqlalchemy`:

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

```
# Finish the connection URI
connection_uri = "postgresql://repl:password@localhost:5432/dwh"
db_engine_dwh = sqlalchemy.create_engine(connection_uri)
```

```
# Transformation step, join with recommendations data
film_pdf_joined = film_pdf.join(recommendations)

# Finish the .to_sql() call to write to store.film
film_pdf_joined.to_sql("film", db_engine_dwh, schema="store", if_exists="replace")

# Run the query to fetch the data
pd.read_sql("SELECT film_id, recommended_film_ids FROM store.film", db_engine_dwh)
```

Loading insights into a PostgreSQL data warehouse isn't that hard! In the next lesson, you'll learn how to put it all together!

## Putting it all together

It's nice to have your ETL behavior encapsulated into a clean `etl()` function.

```python
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"])
```

Now that we have a python function that describes the full ETL, we need to make sure that this function runs at a specific time.

Airflow recap
Apache Airflow is a workflow scheduler written in Python. You can represent directed acyclic graphs in Python objects. DAGs lend themselves perfectly to manage workflows, as there can be a dependency relation between tasks in the DAG. An operator represents a unit of work in Airflow, and Airflow has many of them built-in. As we saw earlier, you can use a BashOperator to run a bash script, for example. There are plenty of other operators as well. Alternatively, you can write a custom operator.

- Workflow scheduler
- Python
- DAGs
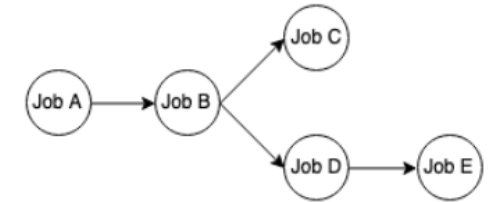


- Tasks defined in operators (e.g. `BashOperator` )

```python
from airflow.models import DAG

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

```python
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)
```

```
# 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
```
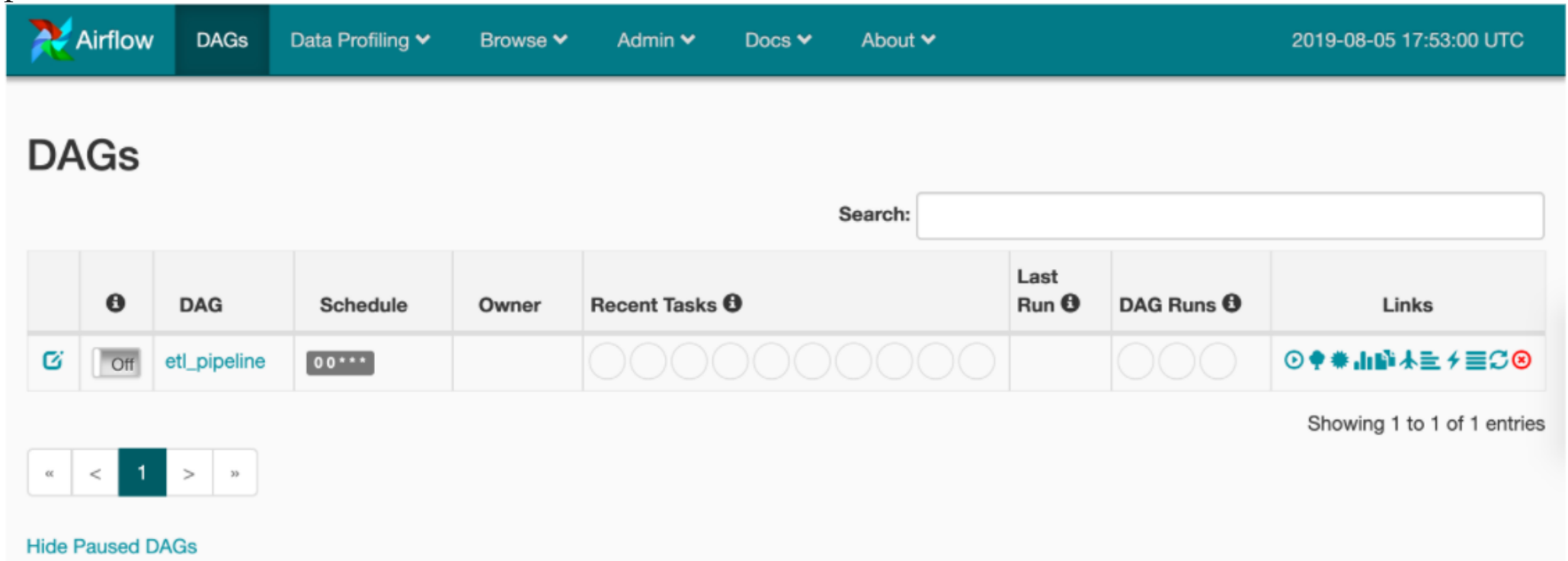
https://crontab.guru

Once you have this DAG definition and some tasks that relate to it, you can write it into a python file and place it in the DAG folder of Airflow. The service detects DAG and shows it in the interface.



# Defining a DAG

In the previous exercises you applied the three steps in the ETL process:

- **Extract**: Extract the `film` PostgreSQL table into `pandas`.
- **Transform**: Split the `rental_rate` column of the `film` DataFrame.
- **Load**: Load a the `film` DataFrame into a PostgreSQL data warehouse.

The functions `extract_film_to_pandas()`, `transform_rental_rate()` and `load_dataframe_to_film()` are defined in your workspace. In this exercise, you'll add an ETL task to an existing DAG. The DAG to extend and the task to wait for are defined in your workspace are defined as `dag` and `wait_for_table` respectively.

```python
# Define the ETL function
def etl():
    film_df = extract_film_to_pandas()
    film_df = transform_rental_rate(film_df)
    load_dataframe_to_film(film_df)

# Define the ETL task using PythonOperator
etl_task = PythonOperator(task_id='etl_film',
                          python_callable=etl,
                          dag=dag)

# Set the upstream to wait_for_table and sample run etl()
etl_task.set_upstream(wait_for_table)
etl()
```

```
<script.py> output:
    Extracting film table...
    Transforming the rental_rate column...
    Loading film DataFrame to table...
```

Be sure to experiment with a few queries once the data pipeline has run. For example:
`pd.read_sql('SELECT rating, AVG(rental_duration) FROM film GROUP BY rating ORDER BY AVG', db_engine)`

# Setting up Airflow

In this exercise, you'll learn how to add a DAG to Airflow. To the right, you have a terminal at your disposal. The workspace comes with Airflow pre-configured, but it's **easy to install on your own**.

You'll need to move the `dag.py` file containing the DAG you defined in the previous exercise to, the DAGs folder. Here are the steps to find it:

- The airflow home directory is defined in the `AIRFLOW_HOME` environment variable. Type `echo $AIRFLOW_HOME` to find out.
- In this directory, find the `airflow.cfg` file. Use `head` to read the file, and find the value of the `dags_folder`.

Now you can find the folder and move the `dag.py` file there: `mv ./dag.py <dags_folder>`.

Which files does the DAGs folder have after you moved the file?

```
repl:~$ echo $AIRFLOW_HOME
~/airflow
repl:~$ ls
airflow  chown  config  dag.py  datasets  repl:repl  start.sh  startup  workspace
```

```
repl:~$ cd airflow
repl:~/airflow$ ls
airflow.cfg  airflow.cfg.bak  airflow.db  airflow-webserver.pid  dags  logs  unittests.cfg
repl:~/airflow$ head airflow.cfg
[core]
# The home folder for airflow, default is ~/airflow
airflow_home = /home/repl/airflow

# The folder where your airflow pipelines live, most likely a
# subfolder in a code repository
# This path must be absolute
dags_folder = /home/repl/airflow/dags

# The folder where airflow should store its log files
repl:~/airflow$ cd
repl:~$ mv ./dag.py /home/repl/airflow/dags
repl:~$ cd airflow/dags
repl:~/airflow/dags$ ls
dag.py  dag_recommendations.py  __pycache__
repl:~/airflow/dags$ 
```

## Possible Answers

○ It has one DAG file: `dag.py` .

● It has two DAG files: `dag.py` and `dag_recommendations.py` .

○ It has three DAG files: `dag.py` , `you_wont_guess_this_dag.py` , and
`super_secret_dag.py` .

# Interpreting the DAG

Now that you've placed the DAG file in the correct place, it's time to check out the Airflow Web UI.

Can you find the scheduled interval of the `sample` DAG?

# DAGs



| | | DAG | Schedule | Owner | Recent Tasks ℹ | Last Run ℹ | DAG Runs ℹ | Links |
|---|---|---|---|---|---|---|---|---|
| ✎ | Off | recommendations | 0 0 * * * | | ○○○○○○○○○○○○○○ | | ○○○ | ⊙♠✳🏭🛫☰⚡☰↻⊗ |
| ✎ | Off | sample | 0 0 * * * | | ○○○○○○○○○○○○○○ | | ○○○ | ⊙♠✳🏭🛫☰⚡☰↻⊗ |

Showing 1 to 2 of 2 entries

« < 1 > »

Hide Paused DAGs

## Possible Answers

◉ Daily at midnight.

○ Hourly at 0 minutes and 0 seconds.

○ It runs once at midnight.

The cron string means it runs at 0:00, so at midnight.

# Chapter 4. Case Study: Data Engineering at DataCamp

Cap off all that you've learned in the previous three chapters by completing a real-world data engineering use case from DataCamp! You will perform and schedule an ETL process that transforms raw course rating data, into actionable course recommendations for DataCamp students!
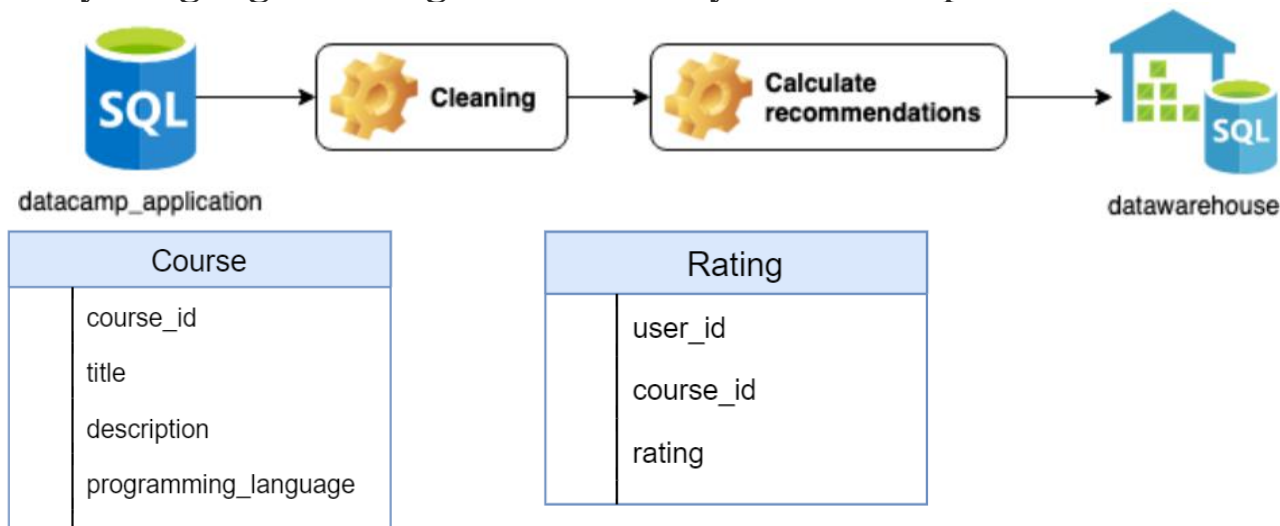
## Course ratings

We can aggregate these chapter ratings to get an estimate of how people rate specific courses. This kind of rating data lends itself to use in recommendation systems.

There are several ways to go about it, but in essence, we need to get this rating data, clean it where possible, and calculate the top-recommended courses for each user. We could re-calculate this daily, for example, and show the courses in the user's dashboard.

We need to extract rating data, transform it to get useful recommendations, and load it into an application database, ready to be used by several recommendation products.

The Data Scientist is responsible for the way recommendations are made, and the Data Engineer fits everything together to get to a stable system that updates recommendations on a schedule.



datacamp_application

| Course |
| --- |
| course_id |
| title |
| description |
| programming_language |

| Rating |
| --- |
| user_id |
| course_id |
| rating |

# Exploring the schema

Have a look at the diagram of the database schema of `datacamp_application`:
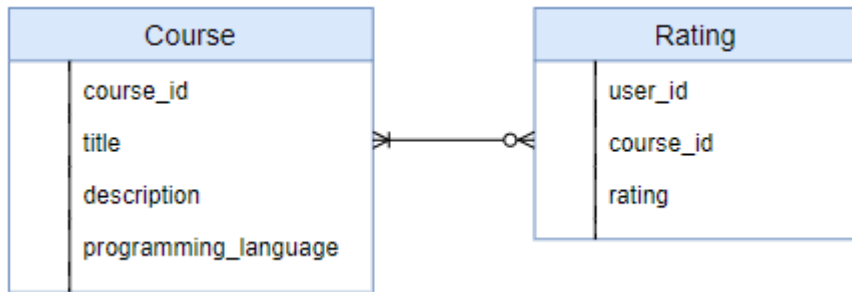


Which column forms the relationship between the two tables?

- ○ The `user_id` column.

- ○ There is no relationship.

- ◉ The `course_id` column.

- ○ The combination of `user_id` and `course_id` columns.

# Querying the table

Now that you have a grasp of what's happening in the `datacamp_application` database, let's go ahead and write up a query for that database.

The goal is to get a feeling for the data in this exercise. You'll get the rating data for three sample users and then use a predefined helper function, `print_user_comparison()`, to compare the sets of course ids these users rated.

```
# Complete the connection URI
connection_uri = "postgresql://repl:password@localhost:5432/datacamp_application"
db_engine = sqlalchemy.create_engine(connection_uri)
```

```
# Get user with id 4387
user1 = pd.read_sql("SELECT * FROM rating WHERE user_id=4387", db_engine)

# Get user with id 18163
user2 = pd.read_sql("SELECT * FROM rating WHERE user_id=18163", db_engine)

# Get user with id 8770
user3 = pd.read_sql("SELECT * FROM rating WHERE user_id=8770", db_engine)

# Use the helper function to compare the 3 users
print_user_comparison(user1, user2, user3)
```

```
<script.py> output:
    Course id overlap between users:
    ================================
    User 1 and User 2 overlap: {32, 96, 36, 6, 7, 44, 95}
    User 1 and User 3 overlap: set()
    User 2 and User 3 overlap: set()
```

As it seems, `user1` and `user2` show much more similar behavior than `user1` or `user2` and `user3`.

# Average rating per course

A great way to recommend courses is to recommend top-rated courses, as DataCamp students often like courses that are highly rated by their peers.

In this exercise, you'll complete a transformation function `transform_avg_rating()` that aggregates the rating data using the `pandas` DataFrame's `.groupby()` method. The goal is to get a DataFrame with two columns, a course id and its average rating:

| course_id | avg_rating |
|-----------|------------|
| 123       | 4.72       |
| 111       | 4.62       |
| ...       | ...        |

In this exercise, you'll complete this transformation function, and apply it on raw rating data extracted via the helper function `extract_rating_data()` which extracts course ratings from the `rating` table.

```python
# Complete the transformation function
def transform_avg_rating(rating_data):
    # Group by course_id and extract average rating per course
    avg_rating = rating_data.groupby('course_id').rating.mean()
    # Return sorted average ratings per course
    sort_rating = avg_rating.sort_values(ascending=False).reset_index()
    return sort_rating


# Extract the rating data into a DataFrame
rating_data = extract_rating_data(db_engines)


# Use transform_avg_rating on the extracted data and print results
avg_rating_data = transform_avg_rating(rating_data)
print(avg_rating_data)
```

```
<script.py> output:
     course_id    rating          89     100  4.282807
  0         46  4.800000          90       5  4.281250
  1         23  4.800000          91      98  4.269006
  2         96  4.692765          92       8  4.257143
  3         56  4.661765          93       2  4.253012
  4         24  4.653061          94      54  4.238095
  5         26  4.646259          95      92  4.222222
  6         61  4.629213          96      29  4.208333
  7         85  4.627119          97      17  4.147059
  8         87  4.626374          98      42  4.107570
  9         81  4.621339
                                  [99 rows x 2 columns]
```

The result of this transformation could now be loaded into another table and used by data products to recommend courses. It would be a very non-personal recommendation, though.

# From ratings to recommendations

It's time to make our recommendations. The goal is to end up with triplets (top three recommended courses) of data, where the first column is a user id, the second column is a course id, and the final column is a rating

prediction. By rating prediction we mean we'll estimate the rating the user would give the course, before they even took it.

| Course |
|---|
| course_id |
| title |
| description |
| programming_language |

**Recommendations**

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

We want to recommend highly rated courses. We want to recommend courses in the programming language that interests the user. We want only to recommend courses that haven't been rated yet by the user.

| Rating |
|---|
| user_id |
| course_id |
| rating |

Common-sense recommendation strategy:
- we'll recommend courses in the technologies for which the user has rated most courses.
- we'll not recommend courses that have already been rated for that user.
- we'll recommend the three courses that remain with the highest rating.

# Filter out corrupt data

One recurrent step you can expect in the transformation phase would be to clean up some incomplete data. In this exercise, you're going to look at course data, which has the following format:

| course_id | title | description | programming_language |
|---|---|---|---|
| 1 | Some Course | ... | r |

You're going to inspect this DataFrame and make sure there are no missing values by using the `pandas` DataFrame's `.isnull().sum()` methods. You will find that the `programming_language` column has some missing values.

As such, you will complete the `transform_fill_programming_language()` function by using the `.fillna()` method to fill missing values.

```python
course_data = extract_course_data(db_engines)

# Print out the number of missing values per column
print(course_data.isnull().sum())

# The transformation should fill in the missing values
def transform_fill_programming_language(course_data):
    imputed = course_data.fillna({"programming_language": "r"})
    return imputed

transformed = transform_fill_programming_language(course_data)

# Print out the number of missing values per column of transformed
print(transformed.isnull().sum())
```

```
<script.py> output:
    course_id                0
    title                    0
    description              0
    programming_language     3
    dtype: int64
    course_id                0
    title                    0
    description              0
    programming_language     0
    dtype: int64
```

You've managed to find out which columns had missing values and write up a transformation function to alleviate that problem.

# Using the recommender transformation

In the last few exercises, you calculated the average rating per course and cleaned up some course data. You will use this data to produce viable recommendations for DataCamp students.

As a reminder, here are the decision rules for producing recommendations:

- Use technology a student has rated the most.
- Exclude courses a student has already rated.
- Find the three top-rated courses from eligible courses.

In order to produce the final recommendations, you will use the average course ratings, and the list of eligible recommendations per user, stored in `avg_course_ratings` and `courses_to_recommend` respectively. You will do this by completing the `transform_recommendations()` function which merges both DataFrames and finds the top 3 highest rated courses to recommend per user.

```python
# Complete the transformation function
def transform_recommendations(avg_course_ratings, courses_to_recommend):
    # Merge both DataFrames
    merged = courses_to_recommend.merge(avg_course_ratings)
    # Sort values by rating and group by user_id
    grouped = merged.sort_values("rating", ascending = False).groupby('user_id')
    # Produce the top 3 values and sort by user_id
    recommendations = grouped.head(3).sort_values("user_id").reset_index()
    final_recommendations = recommendations[["user_id", "course_id","rating"]]
    # Return final recommendations
    return final_recommendations


# Use the function with the predefined DataFrame objects
recommendations = transform_recommendations(avg_course_ratings, courses_to_recommend)
```

In the next lesson, you're going to put it all together and schedule an ETL pipeline for developing recommendations.

# Scheduling daily jobs

What you have learnt 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()`

## The loading phase

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

## Scheduling daily jobs

```python
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

```python
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,
)
```

# The target table

In the previous exercises, you've calculated a DataFrame called `recommendations`. It contains pairs of `user_id`'s and `course_id`'s, with a rating that represents the average rating of this course. The assumption is the highest rated course, which is eligible for a user would be best to recommend.

It's time to put this table into a database so that it can be used by several products like a recommendation engine or an emailing system.

Since it's a `pandas.DataFrame` object, you can use the `.to_sql()` method. Of course, you'll have to connect to the database using the connection URI first. The `recommendations` table is available in your environment.

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


def load_to_dwh(recommendations):
    recommendations.to_sql("recommendations", db_engine, if_exists="replace")
```

We now have all the functions to build or whole etl() function!

# Defining the DAG

In the previous exercises, you've completed the extract, transform and load phases separately. Now all of this is put together in one neat `etl()` function that you can discover in the console.

The `etl()` function extracts raw course and ratings data from relevant databases, cleans corrupt data and fills in missing value, computes average rating per course and creates recommendations based on the decision rules for producing recommendations, and finally loads the recommendations into a database.

As you might remember from the video, `etl()` accepts a single argument: `db_engines`. You can pass this to the task using `op_kwargs` in the `PythonOperator`. You can pass it a dictionary that will be filled in as *kwargs* in the callable.

```python
# Define the DAG so it runs on a daily basis
dag = DAG(dag_id="recommendations",
          schedule_interval="0 0 * * *")

# Make sure `etl()` is called in the operator. Pass the correct kwargs.
task_recommendations = PythonOperator(
    task_id="recommendations_task",
    python_callable=etl,
    op_kwargs={"db_engines":db_engines},
)
```
you've defined the DAG! It's time to enable it.

# Enable the DAG

It's time to enable the DAG you just created, so it can start running on a daily schedule.
To the right you can find the Airflow interface. The DAG you created is called `recommendations`.
Can you find how to enable the DAG?



## Possible Answers

○ By switching the left-hand slide from `Off` to `On`.

○ It's already enabled!

○ By clicking the play icon on the right-hand side.

Answer: By switching the left-hand slide from `Off` to `On`.

# Querying the recommendations

In the previous exercises, you've learned how to calculate a table with course recommendations on a daily basis. Now that this `recommendations` table is in the data warehouse, you could also quickly join it with other tables in order to produce important features for DataCamp students such as customized marketing emails, intelligent recommendations for students and other features.

In this exercise, you will get a taste of how the newly created `recommendations` table could be utilized by creating a function `recommendations_for_user()` which automatically gets the top recommended courses based per user ID for a particular rating threshold.

```python
def recommendations_for_user(user_id, threshold=4.5):
    # Join with the courses table
    query = """
    SELECT title, rating FROM recommendations
        INNER JOIN courses ON courses.course_id = recommendations.course_id
        WHERE user_id=%(user_id)s AND rating>%(threshold)s
        ORDER BY rating DESC
    """
    # Add the threshold parameter
    predictions_df = pd.read_sql(query, db_engine, params = {"user_id": user_id,
                                                             "threshold": threshold})

    return predictions_df.title.values


# Try the function you created
print(recommendations_for_user(12, 4.65))
```

```
<script.py> output:
    ['Extreme Gradient Boosting with XGBoost']
```

This was the final exercise of this course. By using your skills as a data engineer, you're now able to build intelligent products like recommendation engines.

# Course completed!

Recap topics covered:
- Identify the tasks of a data engineer
- What kind of tools used by data engineers
- Cloud service providers introduction
- Databases
- Parallel computing & frameworks (Spark)
- Workflow scheduling with Airflow
- Extract: get data from several sources
- Transform: perform transformations using parallel computing frameworks
- Load: load data into target database
- Hands-on example on recommendations at DataCamp, represented as an ETL task
- Fetch data from multiple sources
- Transform it to form recommendations
- Load it into a target database that is ready to use by data products

## Next Steps
- More practice in ETL

Happy learning!