

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

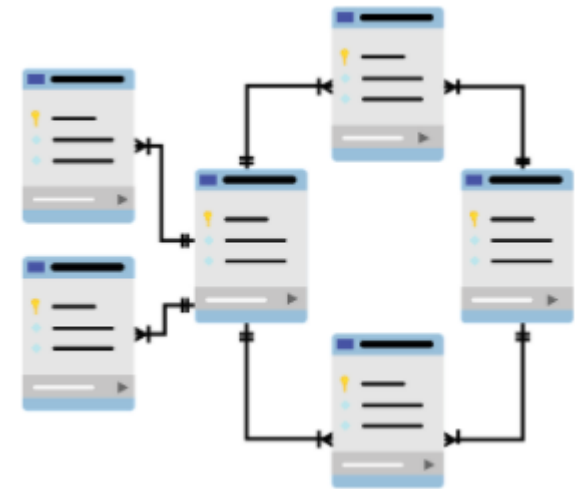
Chapter 2. Data engineering toolbox

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.

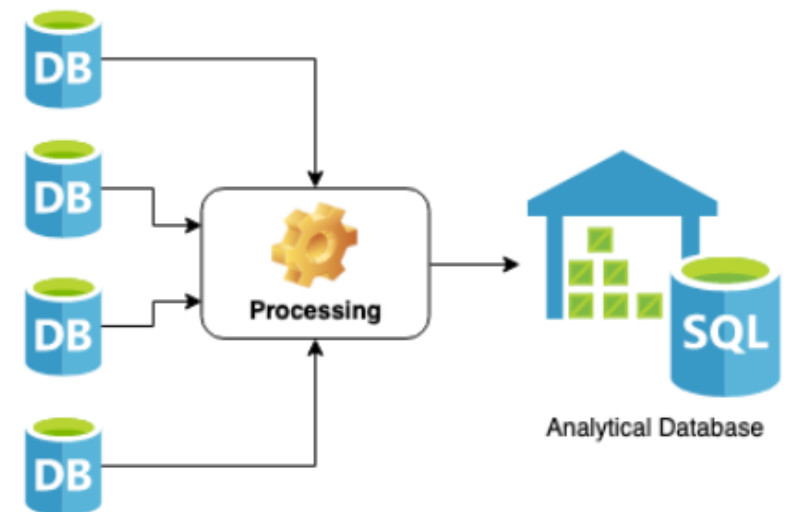
What is data engineering?

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

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.

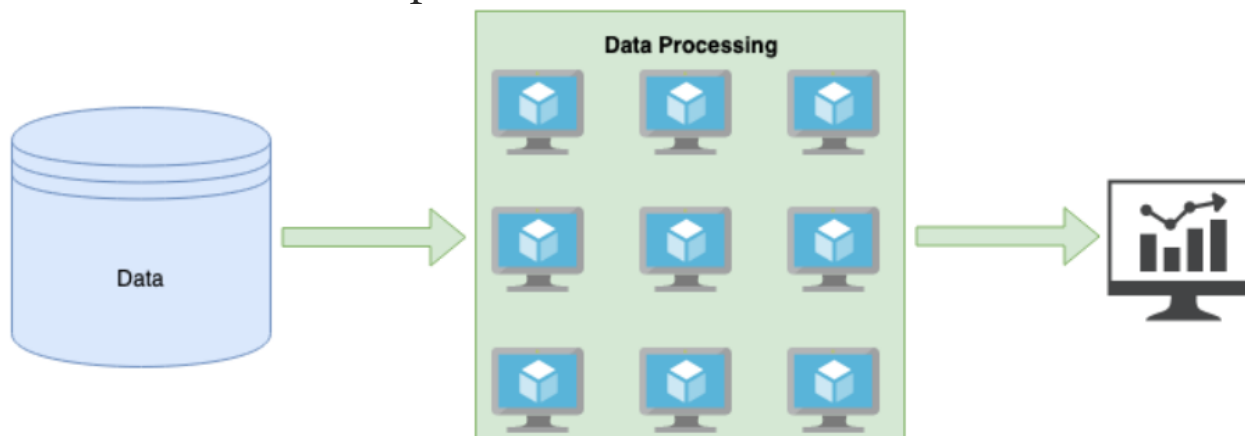
- ⦿ Data scientists are querying the online store databases directly and slowing down the functioning of the application since it's using the same database.
- Harmful product recommendations are affecting the sales numbers of the online store.
- The online store is slow because the application's database 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.

```
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



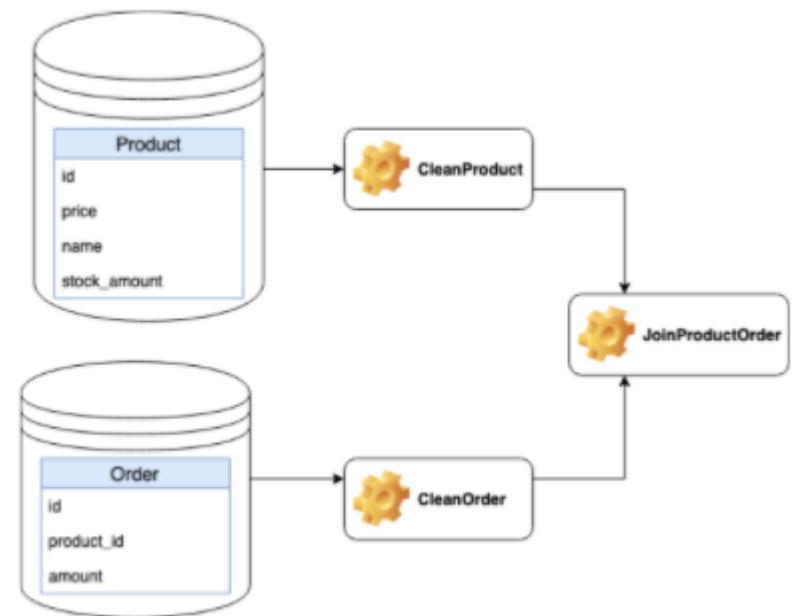
Scheduling



Processing

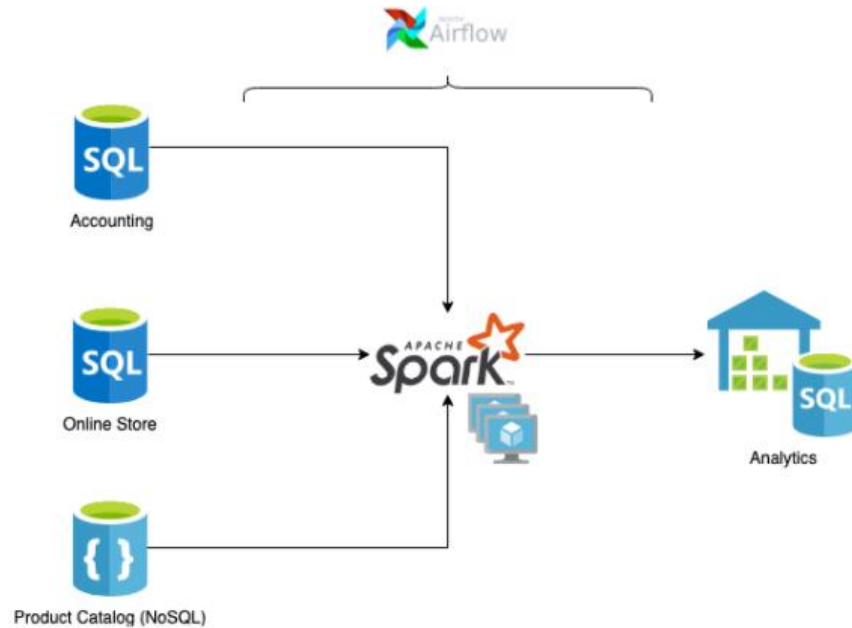


Bash tool: cron



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

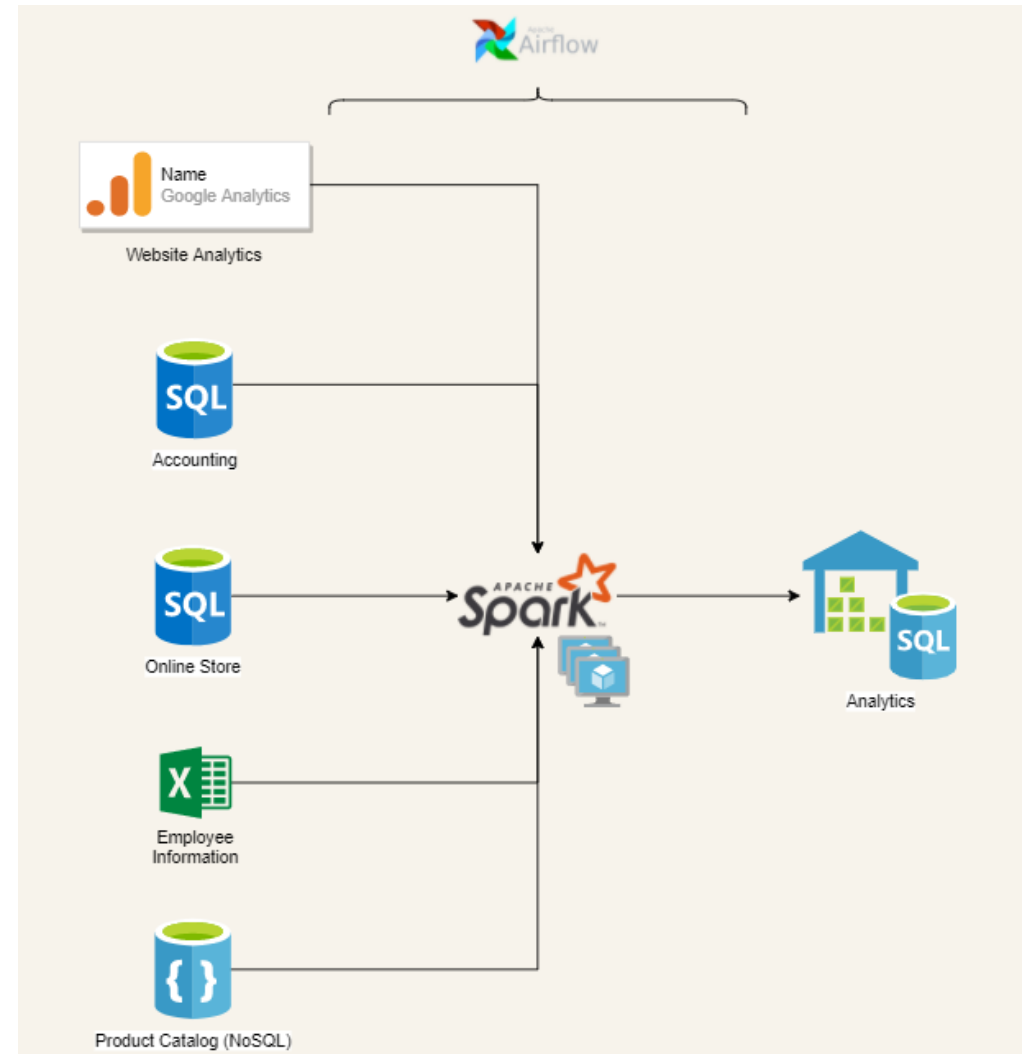
scheduling framework like Airflow. A small side note here is that the sources can be external APIs or other file formats too.



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



17% market share in 2018



10% market share in 2018

- Storage
- Computation
- 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**.

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?

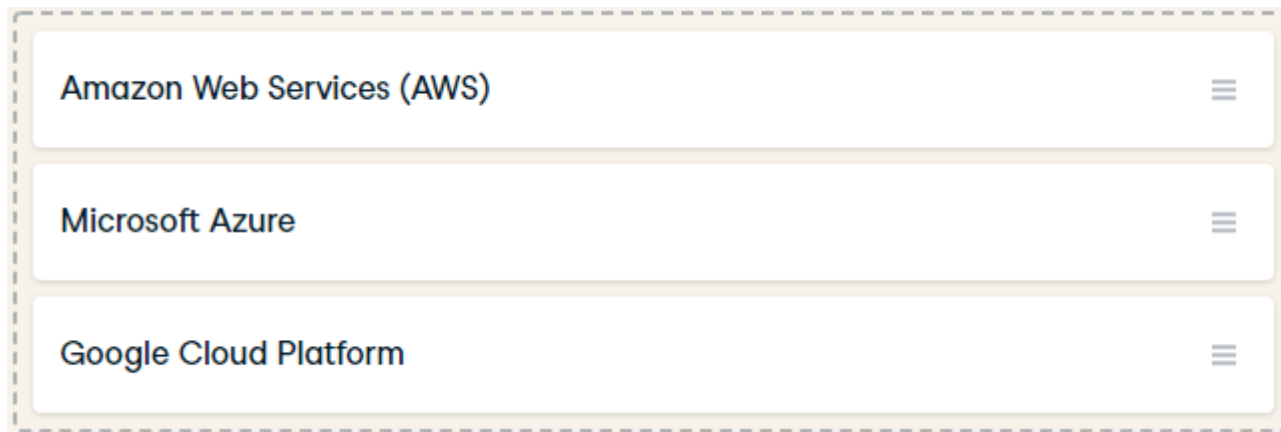
- ☐ Cloud computing is always cheaper.
- ☒ The cloud can provide you with the resources you need, when you need them.
- ☐ On premise machines give me full control over the situation when things break.

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



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

File storage



- 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)



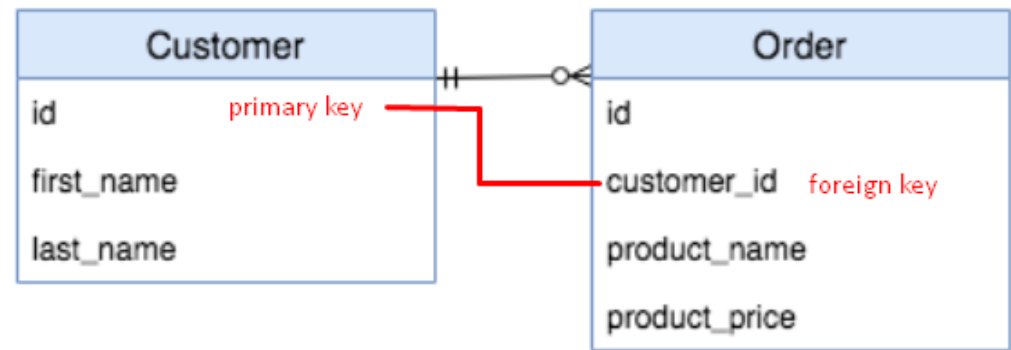
There are several types of NoSQL databases and they are not all unstructured. Two highly used NoSQL database types are key-value stores like Redis or document databases like MongoDB. In key-value stores, the values are simple. Typical use cases are caching or distributed configuration. Values in a document 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")
);
```



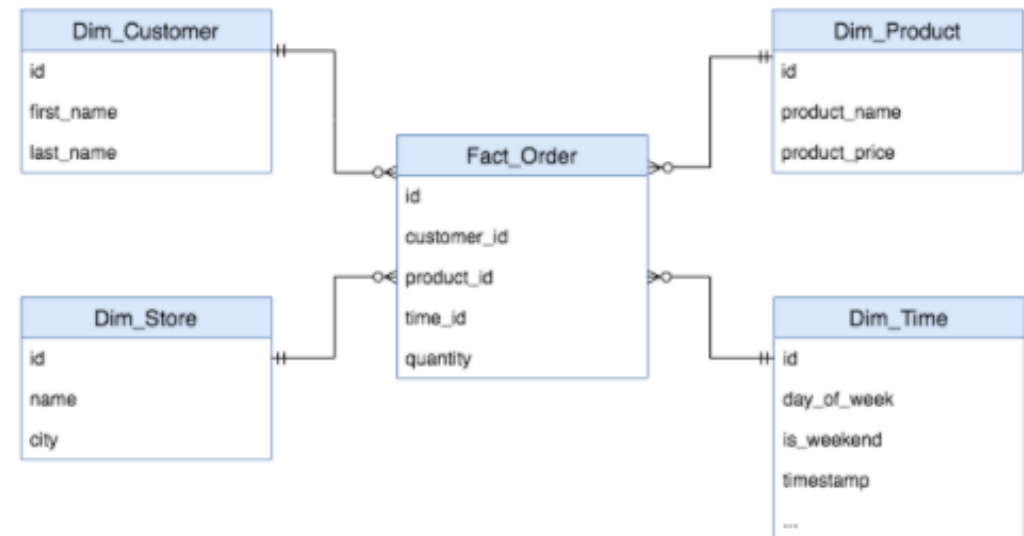
```
-- Join both tables on foreign key
SELECT * FROM "Customer"
INNER JOIN "Order"
ON "customer_id" = "Customer"."id";
```

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

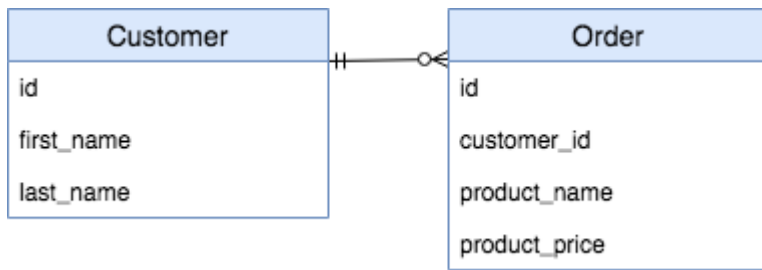
In the video on Databases, you saw the difference between SQL and NoSQL. You saw that SQL uses database schemas to define relations and that NoSQL allows you to work with less structured data.

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.

```
# 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))

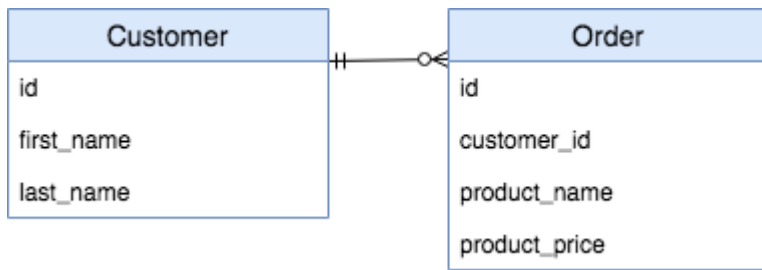
# Show the info of the DataFrame
print(data.info())
```

```
<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.

```
# 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

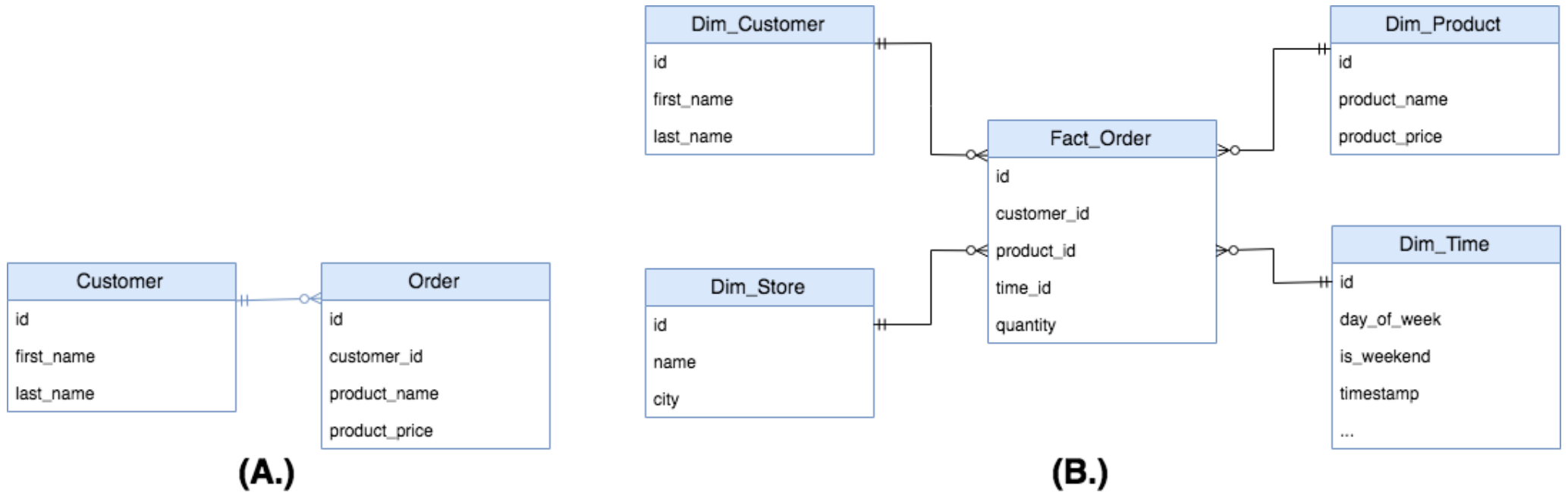
Which of the following images is a star schema?

Possible Answers

☐ A

☒ B

☐ None



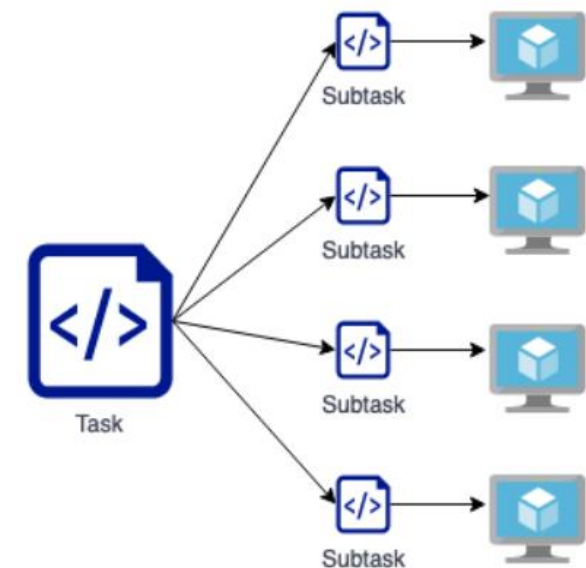
What is parallel computing

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.



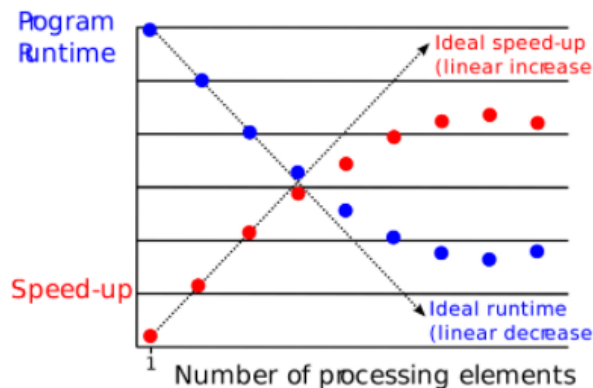
Benefit of parallel computing (having multiple processing units) for big data

- 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.

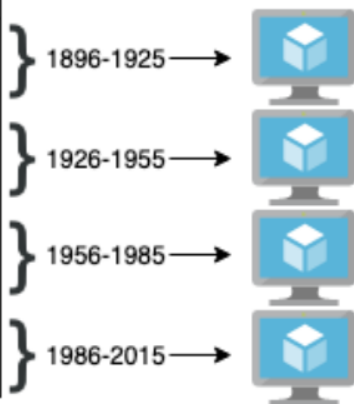
Risks of parallel computing

- **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
1	...	1896	...	25
.
.
.
N	...	2016	...	31



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

```
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()
```

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 `multiprocessing.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.

```
# 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.

```
import dask.dataframe as dd

# Set the number of partitions
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
1900	29.034031
1904	26.698150
1906	27.125253
1908	26.970228
1912	27.538620
1920	29.290978

2002 25.916281

2004 25.639515

2006 25.959151

2008 25.734118

2010 26.124262

2012 25.961378

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.

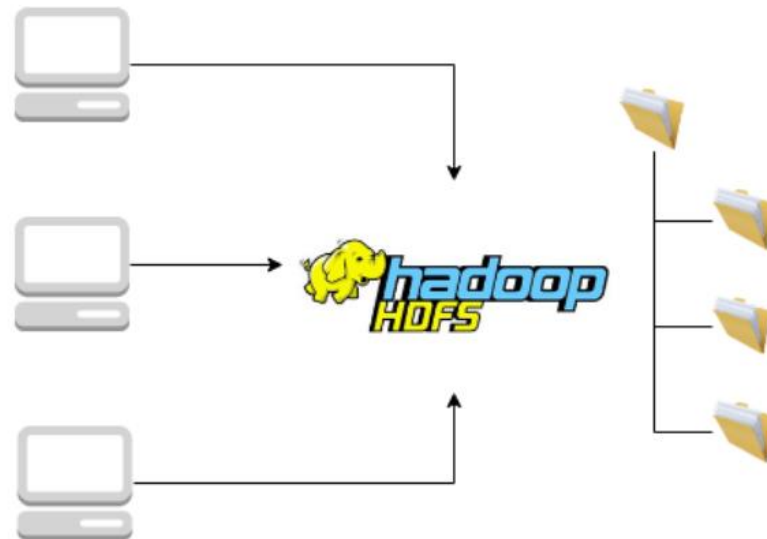
Parallel computation frameworks



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

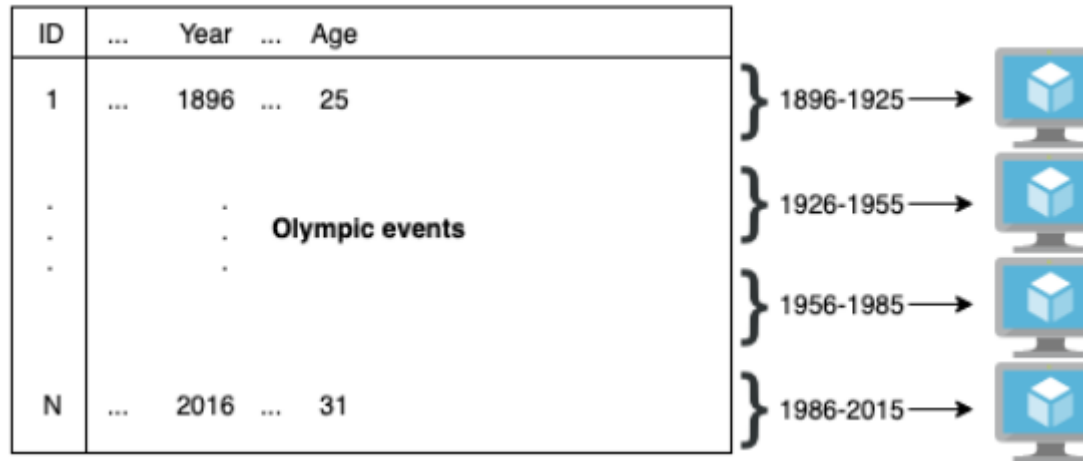
HDFS

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())
```

```
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 where 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

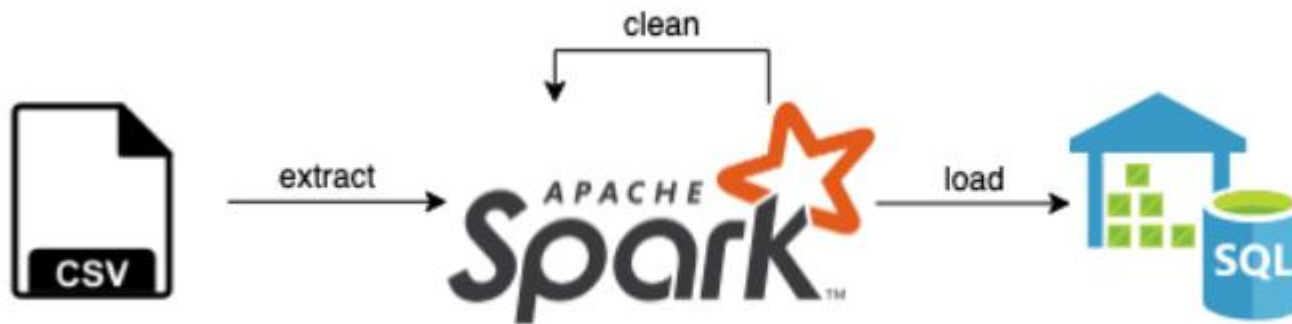
None

```

Answer: A DataFrame with average Olympian heights by year.

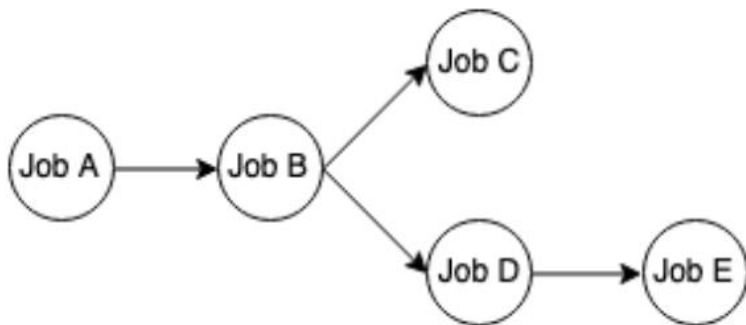
Workflow scheduling frameworks

The task of the workflow scheduling framework is to orchestrate these jobs of pulling data from databases using 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.



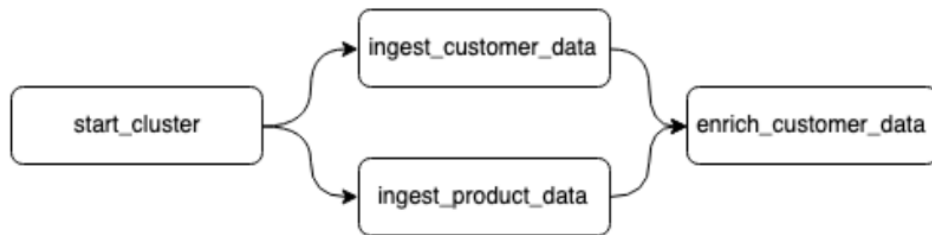
In this example, Job A needs to happen first, then Job B, which enables Job C and D and finally Job E. As you can see, it feels natural to represent this kind of workflow in a DAG. The jobs represented by the DAG can then run in a daily schedule, for example.

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:



```
# 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?

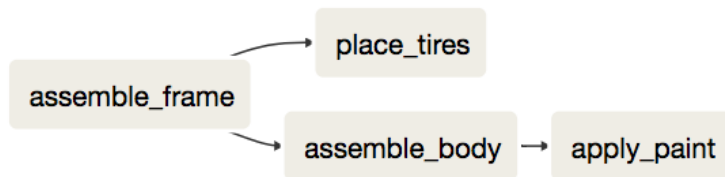
- ☐ You can program your workflow in Python.
- ☐ You can use a directed acyclic graph as a model for dependency resolving.
- ☒ You have exact control over the time at which jobs run.

← 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.

```
# 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

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 completed!

Recap topics covered:

- Basics

Next Steps

- Basics

Happy learning!

