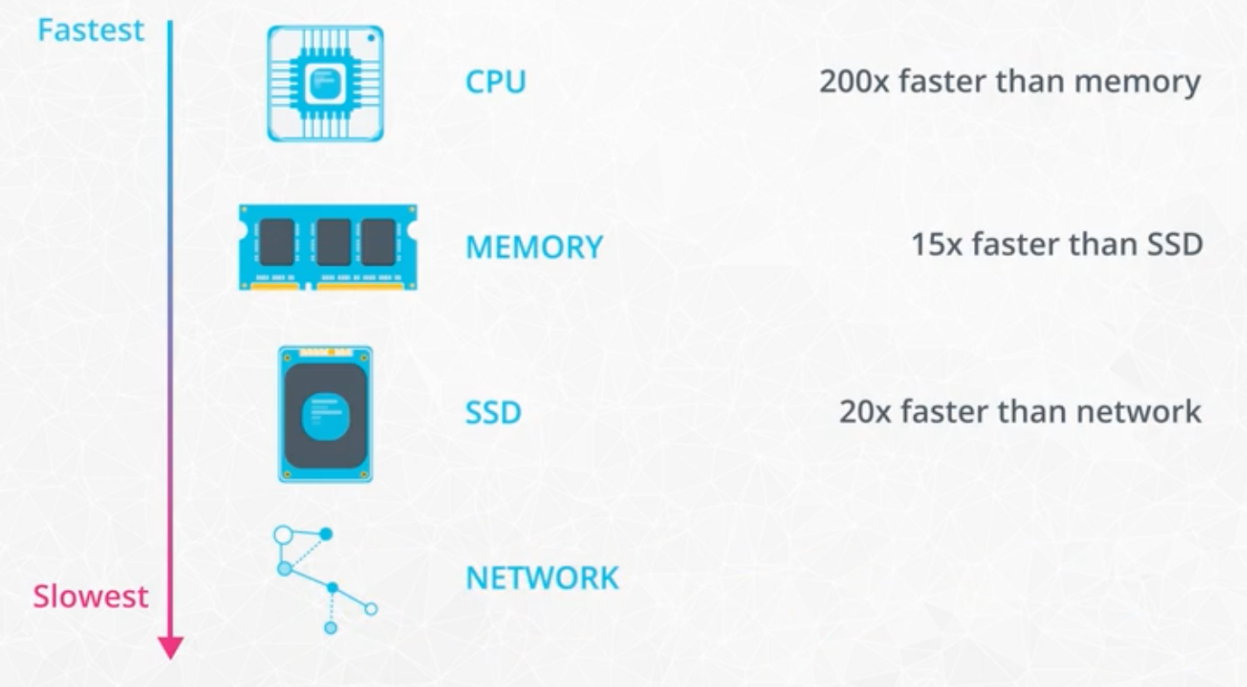
Data Lakes & Spark

Hardware concepts to be familiar with: CPU, RAM, Memory, Network speed



When we talk about distributed computing, we generally refer to a big computational job executing across a cluster of nodes. Each node is responsible for a set of operations on a subset of the data. At the end, we combine these partial results to get the final answer. But how do the nodes know which task to run and demote order? Are all nodes equal? Which machine are you interacting with when you run your code?

Most computational frameworks are organized into a master-worker hierarchy:

- The master node is responsible for orchestrating the tasks across the cluster
- Workers are performing the actual computations

There are four different modes to setup Spark:

- Local mode: In this case, everything happens on a single machine. So, while we use spark's APIs, we don't really do any distributed computing. The local mode can be useful to learn syntax and to prototype your project.

The other three modes are distributed and declare a cluster manager. The cluster manager is a separate process that monitors available resources and makes sure that all machines are responsive during the job. There are three different options of cluster managers:

- Spark's own Standalone Customer Manager
- In this course, you will set up and use your own distributed Spark cluster using Standalone mode.
- In Spark's Standalone mode there is a Driver Process. If you open a Spark shell, either Python or Scala, you are directly interacting with the driver program. It acts as the master and is responsible for scheduling tasks
- YARN from the Hadoop project
- Another open-source manager from UC Berkeley's AMPLab Coordinators.

▼ Data Wrangling with Spark

- Wrangling data with Spark
- Functional programming
- Read in and write out data
- Spark environment and Spark APIs
- RDD API

Functional Programming in Spark

One of the hardest parts of learning Spark is becoming familiar with the **functional style of programming**. Under the hood, Spark is written in a functional programming language called Scala.

- When you're programming with functional languages, you end up solving problems in a pretty different way than you would if you're using a general purpose language like Python.
- Although Spark is written in Scala, you can use it with other languages like Java, R, and even Python. In this course, you'll be developing applications with the Python programming interface or PySpark for short.

Even when you're using the PySpark API, you'll see the functional programming influence of Scala. For example, in the last lesson, you saw a MapReduce problem that counted up the number of times a song was played.

- This code went through each record and spit out a tuple with the name of the song, and the number one.
- The tuples were shuffled and reduced to a sum of the ones that came with each song name.

If you're used to counting with For Loops and found that logic a little strange, it's because this was a functional approach to summing up songs

In the procedural style that most Python programmers know, you'd use a counter variable to keep track of the play count for each song. Then you'd iterate through all the songs, and increment the counter by one if the song name matched.

Why Spark Uses Functional Programming

The core reason is that functional programming is perfect for distributed systems.

- Functional programming helps minimize mistakes that can cripple an entire distributed system.
- Functional programming gets its name from the functions you saw in your algebra class. These functions are more strict than your average Python function because in math a function can only give you one answer when you give it an input. On the other hand, Python allows you to make some flexible, albeit complex, functions that depend on the input and other parameters.
- When you evaluate a mathematical function, you would never change the inputs of that function, but this is exactly what can happen in Python.

Maps and Lambda Functions

one of the most common functions in Spark is Maps. Maps simply make a copy of the original input data, and transform that copy according to whatever function you put inside the map. You can think about them as directions for the data telling each input how to get to the output.

After some initialization to use Spark in our notebook, we

- Convert our log of songs which is just a normal Python list, and to a distributed dataset that Spark can use. This uses the special Spark context object, which is normally abbreviated to SC. The Spark context has a method parallelize that takes a Python object and distributes the object across the machines in your cluster, so Spark can use its functional features on the dataset.
- Once we have this small dataset accessible to Spark, we want to do something with it. One example is to simply convert the song title to a lowercase which can be a common pre-processing step to standardize your data.
- Next, we'll use the Spark function map to apply our converts song to lowercase function on each song in our dataset.

- You'll notice that all of these steps appear to run instantly but remember, the spark commands are using lazy evaluation, they haven't really converted the songs to lowercase yet. So far, Spark is still procrastinating to transform the songs to lowercase, since you might have several other processing steps like removing punctuation, Spark wants to wait until the last minute to see if they can streamline its work, and combine these into a single stage.
- If we want to force Spark to take some action on the data, we can use the `collect` Function which gathers the results from all of the machines in our cluster back to the machine running this notebook.
- You can use anonymous functions in Python, use this special keyword Lambda, and then write the input of the function followed by a colon, and the expected output. You'll see anonymous functions all over the place in Spark. They're completely optional, you could just define functions if you prefer, but there are best-practice, and small examples like these.

▼ [Exercise] **Maps & Lambda functions with Spark**

Maps

In Spark, maps take data as input and then transform that data with whatever function you put in the map. They are like directions for the data telling how each input should get to the output.

The first code cell creates a SparkContext object. With the SparkContext, you can input a dataset and parallelize the data across a cluster (since you are currently using Spark in local mode on a single machine, technically the dataset isn't distributed yet).

Run the code cell below to instantiate a SparkContext object and then read in the log_of_songs list into Spark.

```
# import findspark
# findspark.init('spark-2.3.2-bin-hadoop2.7')
#
# The findspark Python module makes it easier to install
# Spark in local mode on your computer. This is convenient
# for practicing Spark syntax locally.
# However, the workspaces already have Spark installed and you do not
# need to use the findspark module
#
###

import pyspark
sc = pyspark.SparkContext(appName="maps_and_lazy_evaluation_example")

log_of_songs = [
        "Despacito",
        "Nice for what",
        "No tears left to cry",
        "Despacito",
        "Havana",
        "In my feelings",
        "Nice for what",
        "despacito",
        "All the stars"
]

# parallelize the log_of_songs to use with Spark
distributed_song_log = sc.parallelize(log_of_songs)
```

This next code cell defines a function that converts a song title to lowercase. Then there is an example converting the word "Havana" to "havana".

```
def convert_song_to_lowercase(song):
    return song.lower()

convert_song_to_lowercase("Havana")
```

The following code cells demonstrate how to apply this function using a map step. The map step will go through each song in the list and apply the convert_song_to_lowercase() function.

```
distributed_song_log.map(convert_song_to_lowercase)
```

You'll notice that this code cell ran quite quickly. This is because of **lazy evaluation**. Spark does not actually execute the map step unless it needs to.

"RDD" in the output refers to resilient distributed dataset. RDDs are exactly what they say they are: fault-tolerant datasets distributed across a cluster. This is how Spark stores data.

To get Spark to actually run the map step, you need to use an "action". One available action is the collect method. The collect() method takes the results from all of the clusters and "collects" them into a single list on the master node.

```
distributed_song_log.map(convert_song_to_lowercase).collect()
```

Note as well that Spark is not changing the original data set: Spark is merely making a copy. You can see this by running collect() on the original dataset.

```
distributed_song_log.collect()
```

You do not always have to write a custom function for the map step. You can also use anonymous (lambda) functions as well as built-in Python functions like string.lower().

Anonymous functions are actually a Python feature for writing functional style programs.

```
distributed_song_log.map(lambda song: song.lower()).collect()

distributed_song_log.map(lambda x: x.lower()).collect()
```

▼ [Exercise] **Data Wrangling with Spark**

Data Wrangling with Spark

This is the code used in the previous screencast. Run each code cell to understand what the code does and how it works.

These first three cells import libraries, instantiate a SparkSession, and then read in the data set

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import desc
from pyspark.sql.functions import asc
from pyspark.sql.functions import sum as Fsum

import datetime

import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
```

```
spark = SparkSession \
    .builder \
    .appName("Wrangling Data") \
    .getOrCreate()

path = "data/sparkify_log_small.json"
user_log = spark.read.json(path)
```

```
#Data exploration, various cells that you can execute to understand your df
user_log.take(5)
```

```
user_log.printSchema()

user_log.describe().show()

user_log.describe("artist").show()
user_log.describe("sessionId").show()

user_log.select("page").dropDuplicates().sort("page").show()

user_log.select(["userId", "firstname", "page", "song"]).where(user_log.userId == "1046").collect()

#calculating stats by the hour
get_hour = udf(lambda x: datetime.datetime.fromtimestamp(x / 1000.0). hour)

user_log = user_log.withColumn("hour", get_hour(user_log.ts))
user_log.head()

songs_in_hour = user_log.filter(user_log.page == "NextSong").groupby(user_log.hour).count().orderBy(user_log.hour.cast("float"))
songs_in_hour.show()

songs_in_hour_pd = songs_in_hour.toPandas()
songs_in_hour_pd.hour = pd.to_numeric(songs_in_hour_pd.hour)

plt.scatter(songs_in_hour_pd["hour"], songs_in_hour_pd["count"])
plt.xlim(-1, 24);
plt.ylim(0, 1.2 * max(songs_in_hour_pd["count"]))
plt.xlabel("Hour")
plt.ylabel("Songs played");

#drop rows with missing values
user_log_valid = user_log.dropna(how = "any", subset = ["userId", "sessionId"])
user_log_valid.count()

user_log.select("userId").dropDuplicates().sort("userId").show()

user_log_valid = user_log_valid.filter(user_log_valid["userId"] != "")
user_log_valid.count()

#Users downgrade their account
user_log_valid.filter("page = 'Submit Downgrade'").show()
user_log.select(["userId", "firstname", "page", "level", "song"]).where(user_log.userId == "1138").collect()

flag_downgrade_event = udf(lambda x: 1 if x == "Submit Downgrade" else 0, IntegerType())

user_log_valid = user_log_valid.withColumn("downgraded", flag_downgrade_event("page"))
user_log_valid.head()

from pyspark.sql import Window
windowval = Window.partitionBy("userId").orderBy(desc("ts")).rangeBetween(Window.unboundedPreceding, 0)

user_log_valid = user_log_valid.withColumn("phase", Fsum("downgraded").over(windowval))

user_log_valid.select(["userId", "firstname", "ts", "page", "level", "phase"]).where(user_log.userId == "1138").sort("ts").collect(
```

▼ Spark in AWS

Apache Spark is a fast, in-memory data processing engine that is used to perform a wide range of data processing tasks, including batch processing, stream processing, machine learning, and interactive SQL queries. It is designed to be faster and more flexible than MapReduce, the original distributed computing paradigm used in Hadoop, and is often used as an alternative to MapReduce for ETL (extract, transform, load) tasks.

Spark has several key benefits:

1. Speed: Spark is designed to be much faster than MapReduce, as it can process data in-memory rather than reading from and writing to disk. This makes it ideal for handling large data sets that require fast processing speeds.
2. Flexibility: Spark allows for a wide range of data processing tasks, including batch processing, stream processing, machine learning, and interactive SQL queries. It also supports a variety of programming languages, including Java, Python, and Scala.
3. Ease of use: Spark has a simple API and a rich ecosystem of libraries and tools, which makes it easy to use and integrate with other systems. It also has strong support for interactive data exploration and visualization.

4. Scalability: Spark can scale up from a single server to thousands of machines, each offering local computation and storage. This makes it well-suited for handling very large data sets.

Overall, Spark is a powerful and flexible data processing tool that is widely used in a variety of industries, including finance, healthcare, retail, and government. It is often used

Hadoop

Apache Hadoop is an open-source software framework used for distributed storage and processing of large data sets on computer clusters. It is designed to scale up from a single server to thousands of machines, each offering local computation and storage.

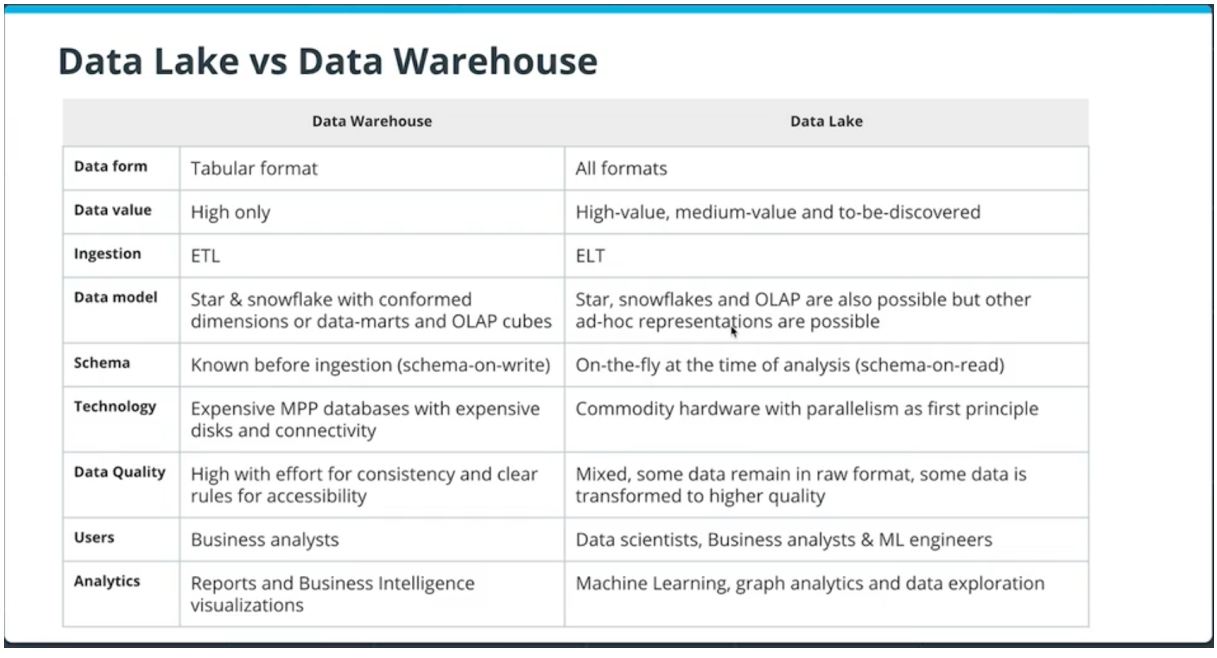
Hadoop is based on the MapReduce programming model, which allows for distributed processing of large data sets across a cluster of computers. It consists of two main components: the Hadoop Distributed File System (HDFS) for storing data, and the YARN resource management platform for scheduling tasks and allocating resources.

Hadoop is used by many organizations to process and analyze large data sets, such as web logs, social media data, and sensor data. It is particularly useful for handling data that is too large or too complex to be processed using traditional database systems.

Hadoop is one of the most widely used big data technologies and is used in many industries, including finance, healthcare, retail, and government. It is often used in conjunction with other big data tools, such as Apache Spark and Apache Flink, to build data pipelines and perform complex data analysis tasks.

Spark can be used in conjunction with Hadoop to perform a wide range of data processing tasks, including batch processing, stream processing, machine learning, and interactive SQL queries. It is often used as an alternative to MapReduce for ETL (extract, transform, load) tasks, as it is faster and more flexible than MapReduce.

Spark can be run on top of the Hadoop Distributed File System (HDFS) or on other storage systems, such as Amazon S3 or Azure Blob Storage. It can also be used in conjunction with other big data tools, such as Apache Flink and Apache Beam, to build data pipelines and perform complex data analysis tasks.



Data Lake vs Data Warehouse

	Data Warehouse	Data Lake
Data form	Tabular format	All formats
Data value	High only	High-value, medium-value and to-be-discovered
Ingestion	ETL	ELT
Data model	Star & snowflake with conformed dimensions or data-marts and OLAP cubes	Star, snowflakes and OLAP are also possible but other ad-hoc representations are possible
Schema	Known before ingestion (schema-on-write)	On-the-fly at the time of analysis (schema-on-read)
Technology	Expensive MPP databases with expensive disks and connectivity	Commodity hardware with parallelism as first principle
Data Quality	High with effort for consistency and clear rules for accessibility	Mixed, some data remain in raw format, some data is transformed to higher quality
Users	Business analysts	Data scientists, Business analysts & ML engineers
Analytics	Reports and Business Intelligence visualizations	Machine Learning, graph analytics and data exploration