Data Wrangling with Spark

In this lesson, you'll practice wrangling data with Spark. If you are familiar with both SQL and Python's pandas library, you'll notice quite a few similarities with the Spark SQL module and Spark Dataset API.

Lesson Overview

- 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.

If you want to use Spark effectively, you'll have to get comfortable with a few tools from functional programming.

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.

QUIZ QUESTION

In the video, David explains why Spark uses functional programming. He also implies that Python is not a functional programming language. How is it possible that Spark programs can be written in Python if Python is not a functional programming language? Mark all answers that are true.

- [ ] You cannot write Spark programs in Python.
- [x] The PySpark API is written with functional programming principles in mind.
- [ ] You can write Spark programs in Python using an API, but you need to be extra careful when writing Spark programs with the PySpark API because Python is an imperative language.
- [ ] Spark is written in Scala. So the PySpark API first translates your code to Scala, which is a functional programming language.

Yes, that's correct. The PySpark API allows you to write programs in Spark and ensures that your code uses functional programming practices. Underneath the hood, the Python code uses py4j to make calls to the Java Virtual Machine (JVM).

Procedural Programming Example

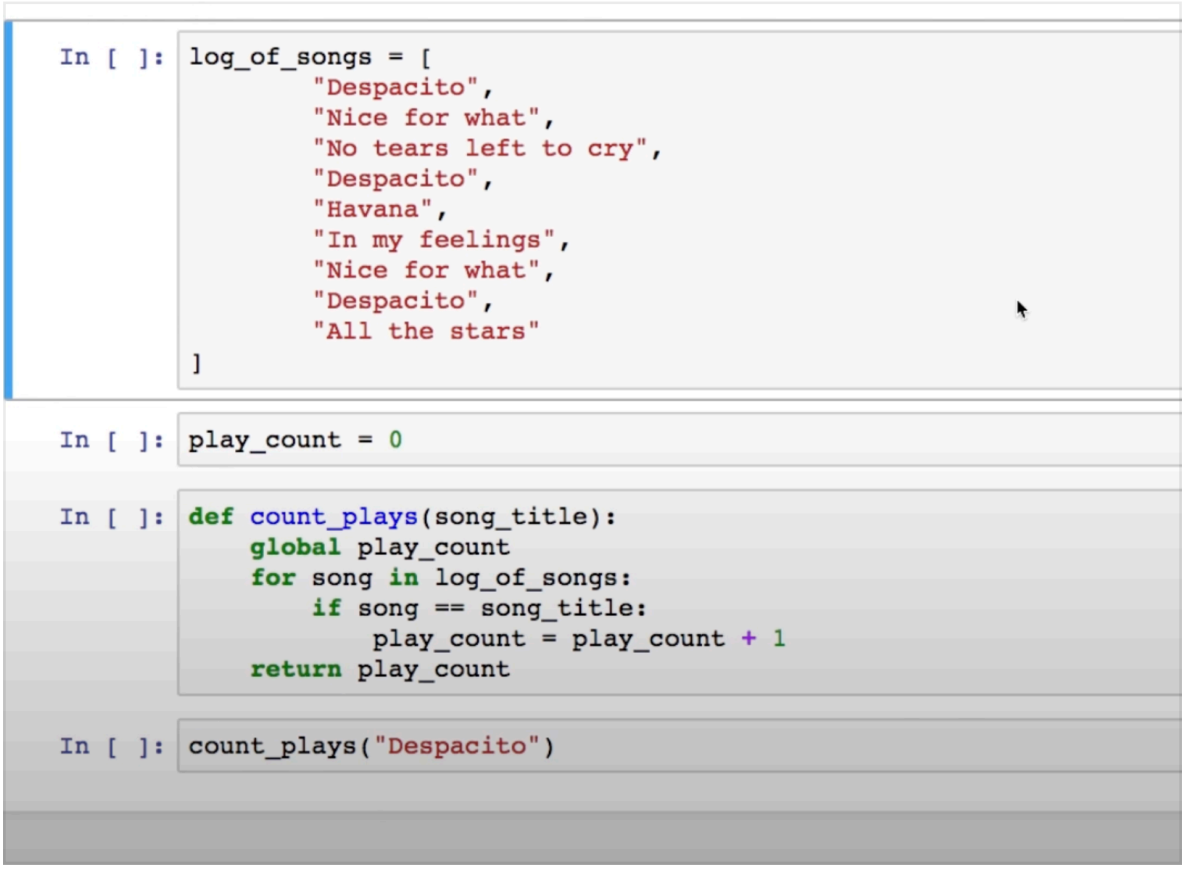
Here's a concrete example of a potential issue with procedural programming in Python:

This code counts up the number of times a specific song was played assuming the log of songs was converted to a Python list. We start by setting a global variable, play count, to keep track of the number of times the song was played, and we initialize it to zero.

We then use a for loop to iterate through all the songs in the list. If the song title is "Despacito," then we increase our play count by one.

- When we run this procedure in the final statement, the code that runs actually changes the variable play count. As a result, running the same code again with the exact same input of "Despacito," gives a different result each time.
- The problem in this example is easy to see, but when you have dozens of machines running code in parallel, and sometimes you need to

restart a calculation if one of the machines has a temporary issue.
- In Python, we casually refer to anything after a def as a function. But technically, these are often methods or procedures, not **pure** functions.



Procedural Programming Exercise

This notebook contains the code from the previous screencast. The code counts the number of times a song appears in the log_of_songs variable.

- The first time you run count_plays("Despacito"), you get the correct count.
- When you run the same code again count_plays("Despacito"), the results are no longer correct.This is because the global variable play_count stores the results outside of the count_plays function.

In Spark, if your data is split onto two different machines, machine A will run a function to count how many times 'Despacito' appears on machine A. Machine B will simultaneously run a function to count how many times 'Despacito' appears on machine B. After they finish counting individually, they'll combine their results together. You'll see how this works in the next parts of the lesson.

Imagine your program is like a bread factory, and your function is a specific machine in your factory that makes sourdough bread. But since your factory

needs to mass-produce bread at a large scale, you need to be a bit more careful.

One thing you'll need to avoid when designing your bread-maker is unintended side effects.

- After each loaf, your machine needs to leave the factory exactly the same as before it ran.
- If you don't, each machine could start interfering with the others.

For example, if running a single bread-maker made the entire factory one degree warmer, then running several of these machines all at once would start to heat up the room.
In distributed systems, your functions shouldn't have side effects on variables outside their scope, since this could interfere with other functions running on your cluster.

- Your bread making machine needs to get the ingredients without ruining them since other breadmakers will also need them.
- In distributed systems, you also need to be careful with how you design your functions. Whenever your functions run on some input data, it can alter it in the process. If your bread-making machine protects the input ingredients and doesn't cause any side effects, then you have a smooth and clean operation.

If you write functions that **preserve their inputs** and **avoid side effects**, these are called **pure functions**, and your spark code will work well at the scale of big data.

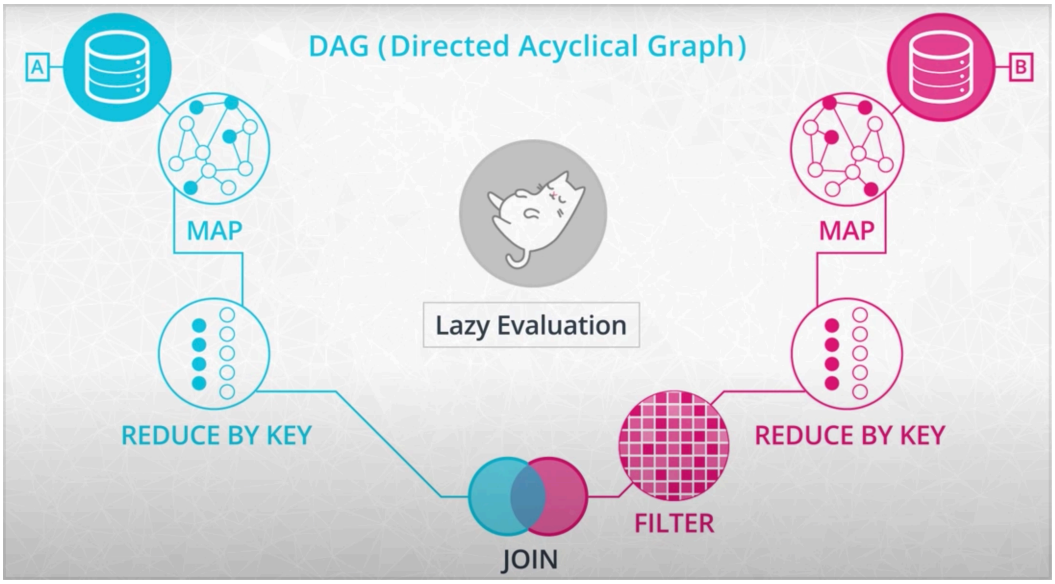
Just like bread companies make copies of the starter from their mother dough, every Spark function makes a copy of its input data and never changes the original parent data. Because Spark doesn't change or mutate the input data, it's known as **immutable**. This makes sense when you have a single function. But what happens when you have lots of functions in your program?

- In Spark, you do this by chaining together multiple functions that each accomplish a small chunk of the work.
- You'll often see a function that is composed of multiple sub-functions
- In order for this big function to be peer, each sub function also has to be peer.

Spark uses a functional programming concept called lazy evaluation. Before Spark does anything with the data in your program, it first built step-by-step directions of what functions and data it will need. These directions are like the recipe for your bread, and in Spark, this is called a **Directed Acyclic Graph (DAG)**. Once Spark builds the DAG from your code, it checks if it can procrastinate, waiting until the last possible moment to get the data.

When baking bread, you would look at the recipe before you start mixing ingredients together to see what you can grab and mix together in one big step.

In fact, you often mix all your dry ingredients, then mix all your wet ingredients, and then combine them together before baking.
In Spark, these multi-step combos are called **stages**.



Lazy Evaluation in the Spark Director Acyclic Graph (DAG)

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.
-

For more about the theory and origins of lambda functions, take a look at this blog post. Why are lambda functions called "lambda" functions? According to legend, the inventor of Lambda Calculus, Alonzo Church, originally used the wedge symbol

\wedge

$\wedge$ **as part of his notation. But the typsetter transcribing his manuscript used**

\lambda

λ instead. You can read more about it in the blog post.

Data Formats

The most common data formats you might come across are CSV, JSON, HTML, and XML.

- A CSV file or comma-separated values file, source tabular data in index format. Each line represents a record where fields are always in the same order defined usually by the first header. As the name suggests, the records are separated with a comma.
- JSON or JavaScript object notation storage records in attribute value pairs. The values may contain simple often numeric types or arrays.
- HTML or hypertext markup language, is a standard markup language for creating web pages and web applications. If you scrape data from the Internet, you will need to parse HTML.

Distributed Data Stores

When we have so much data that we need distributed computing, the data itself often needs to be stored in a distributed way as well.
Distributed file systems, storage services, and distributed databases store data in a fault-tolerant way. So if a machine breaks or becomes unavailable, we don't lose the information we have collected.

Hadoop has a Distributed File System, HDFS, to store data. HDFS splits files into 64 or 128 megabyte blocks and replicates these blocks across the cluster. This way, the data is stored in a fault tolerant way and can be accessed in digestible chunks.
If you don't want to maintain your own cluster, we can use many services such as the Amazon Simple Storage Service or S3. Companies using AWS or Amazon Web Services often use S3 to store the raw data they have collected.

QUIZ QUESTION

If you have big data, why does it need to be stored in a distributed way?

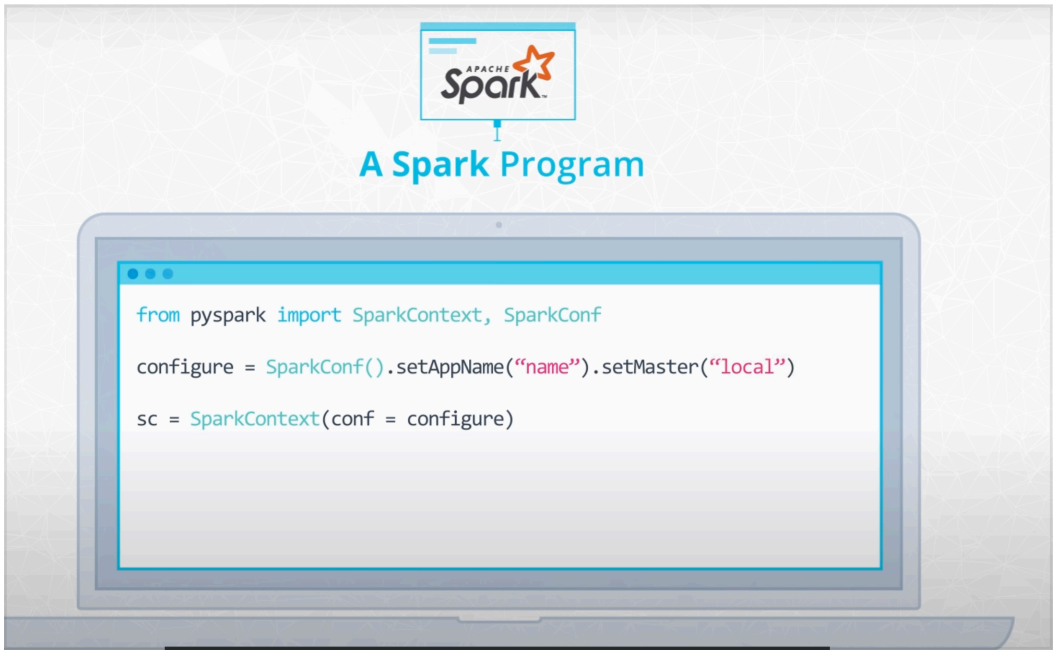
- ○ So you can have more memory for the data
- ○ Because you need to break up the data into chunks
- ✓ So you don't lose the data you have collected

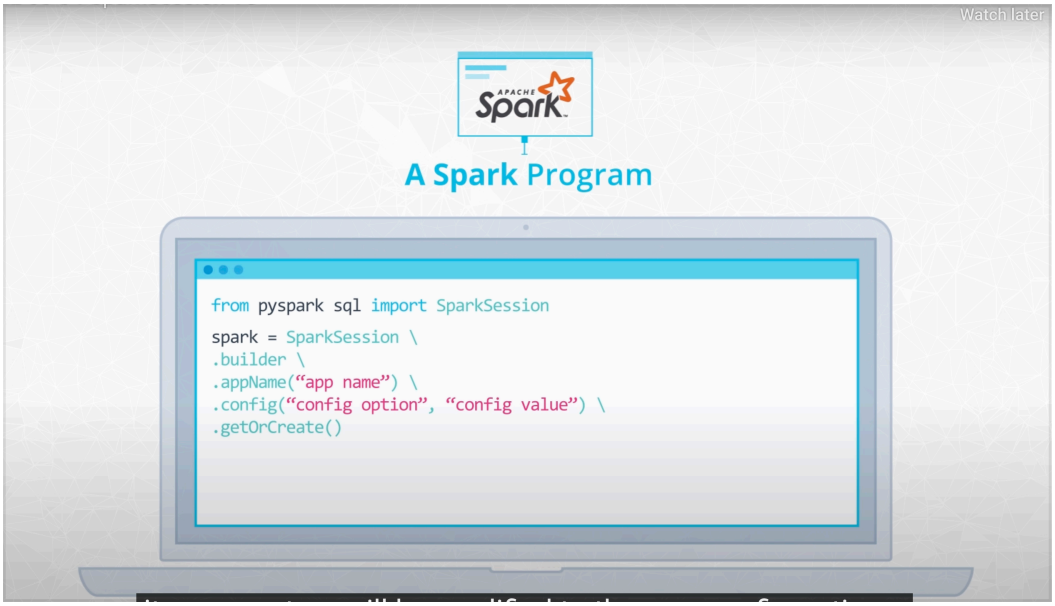
You can find more information about Amazon S3 and HDFS by following these links.

The SparkSession

- The first component of each Spark Program is the SparkContext. The SparkContext is the main entry point for Spark functionality and connects the cluster with the application.
- To create a SparkContext, we first need a SparkConf object to specify some information about the application such as its name and the master's nodes' IP address. If we run Spark in local mode, we can just put the string local as master.
- To read data frames, we need to use Spark SQL equivalent, the SparkSession.

- Similarity to the SparkConf, we can specify some parameters to create a SparkSession.
  - getOrCreate()for example, means that if you already have a SparkSession running, instead of creating a new one, the old one will be returned and its parameters will be modified to the new configurations.





Reading data frames

Reading and Writing into Spark Data Frames

In this video, we will go through an example of how to import and export data to and from Spark data frames using a dataset that describes log events coming from a music streaming service

- Create a Spark session with parameters.
- Load a JSON file into a Spark data frame called user_log.
- Print the schema with the printSchema method.

- Try the describe method to see what we can learn from our data.
- Use the take method to grab the first few records.
- Save it into a different format, for example, into a CSV file, with the write.save method
- Use the read.csv method

In this video, we went through an example of how to load and save data frames from and to HDFS. If a file is stored in S3, you can use the same methods. When specifying the file path we just need to make sure that we are pointing to the S3 bucket that stores our target file.

Tip

If Spark is used in a cluster mode all the worker nodes need to have access to the input data source. If you're trying to import a file saved only on the local disk of the driver node you'll receive an error message similar to this:

AnalysisException: u'Path does not exist: file:/home/ubuntu/test.csv;'
Loading the file should work if all the nodes have it saved under the same path.

Imperative VS Declarative Programming

We will cover two different ways to manipulate our data:
Imperative programming using DataFrames and Python

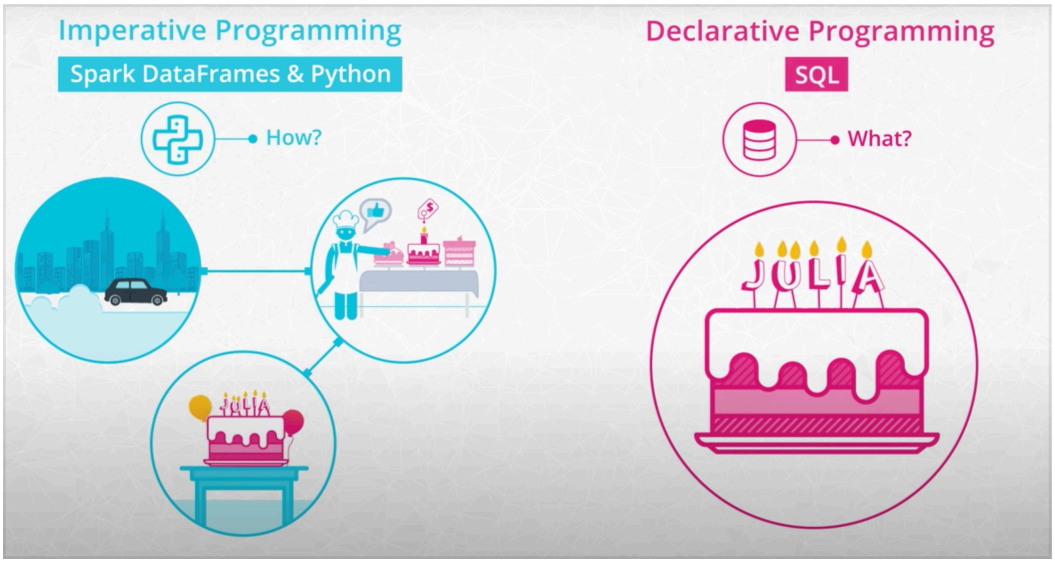
- Imperative programming is concerned with the How
- Let's get in the car, drive two miles down the road to my favorite bakery, go into the shop, select the cake from the counter, purchase the cake, and then drive home.

- Focus on the exact steps, how we get to the result
- Data transformations with DataFrames

Declarative programming using SQL

- Cares about the What
- Let's get the cake for Julia.
- concerned about the result we want to achieve
- abstraction layer of an imperative system

If you have used pandas DataFrames before, you are probably familiar with how to manipulate DataFrames programmatically. We can chain methods such as filter and group by one after another, transforming the DataFrame further and further. In the next few videos, we will dive into how to do data transformations with DataFrames and imperative programming.



Imperative vs Declarative programming paradigms

Data Wrangling with Data Frames

Using the same music streaming service log dataset, let's do some data wrangling using DataFrames

- Print the schema, so we can see the columns we have in the DataFrame
- Call describe on the whole frame to see the values of each column
- Check how many rows we have in the data frame with the count method
- Use dropDuplicates to see each kind once
- Filter out all the rows that have an empty string value as the user ID.
- Use a window function to compute cumulative sums

In this screencast, the SparkSession object wasn't shown instantiated explicitly. Remember to instantiate an object using code like this:

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

Functions

In the previous video, we've used a number of functions to manipulate our dataframe. Let's take a look at the different type of functions and their potential pitfalls.

General functions

We have used the following general functions that are quite similar to methods of pandas dataframes:

- select(): returns a new DataFrame with the selected columns

- filter(): filters rows using the given condition
- where(): is just an alias for filter()
- groupBy(): groups the DataFrame using the specified columns, so we can run aggregation on them
- sort(): returns a new DataFrame sorted by the specified column(s). By default the second parameter 'ascending' is True.
- dropDuplicates(): returns a new DataFrame with unique rows based on all or just a subset of columns
- withColumn(): returns a new DataFrame by adding a column or replacing the existing column that has the same name. The first parameter is the name of the new column, the second is an expression of how to compute it.

Aggregate functions
Spark SQL provides built-in methods for the most common aggregations such as count(), countDistinct(), avg(), max(), min(), etc. in the pyspark.sql.functions module. These methods are not the same as the built-in methods in the Python Standard Library, where we can find min() for example as well, hence you need to be careful not to use them interchangeably.
In many cases, there are multiple ways to express the same aggregations. For example, if we would like to compute one type of aggregate for one or more columns of the DataFrame we can just simply chain the aggregate method after a groupBy(). If we would like to use different functions on different columns, agg()comes in handy. For example agg({"salary": "avg", "age": "max"}) computes the average salary and maximum age.

User defined functions (UDF)
In Spark SQL we can define our own functions with the udf method from the pyspark.sql.functions module. The default type of the returned variable for UDFs is string. If we would like to return an other type we need to explicitly do so by using the different types from the pyspark.sql.types module.

Window functions
Window functions are a way of combining the values of ranges of rows in a DataFrame. When defining the window we can choose how to sort and group (with the partitionBy method) the rows and how wide of a window we'd like to use (described by rangeBetween or rowsBetween).
For further information see the Spark SQL, DataFrames and Datasets Guide and the Spark Python API Docs.

Spark SQL
Next, you'll learn to query data with a declarative approach. Spark comes with a SQL library that lets you query DataFrames using the same SQL syntax you'd use in a tool like MySQL or Postgres.
you'll be able to share your Spark code with a wider community, since many analysts and data scientists prefer using SQL.

Spark automatically optimizes your SQL code, to speed up the process of manipulating and retrieving data.
To follow the cake analogy, you simply tell your program to bake a cake, and Spark SQL will give you a great cake even if you don't detail the exact steps.

Spark SQL resources

Here are a few resources that you might find helpful when working with Spark SQL

- Spark SQL built-in functions
- Spark SQL guide

Resilient Distributed Datasets (RDDs)
RDDs are a low-level abstraction of the data. In the first version of Spark, you worked directly with RDDs. You can think of RDDs as long lists distributed across various machines. You can still use RDDs as part of your Spark code although data frames and SQL are easier. This course won't go into the details of RDD syntax, but you can find some further explanation of the difference between RDDs and DataFrames in Databricks' A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets blog post.
Here is a link to the Spark documentation's RDD programming guide.

Lesson Summary

We've covered quite a bit in this lesson!

- First, you've learned how to wrangle data with functional programming, and you saw this programming style helps us scale across distributed systems.
- Then you practice reading data into Spark, and writing data out.
- You also gained experience with high-level APIs, Spark DataFrames, and Spark SQL.

Now that you know how to wrangle data with Spark, go and find some datasets that you are interested in, and start exploring them with Spark. Once you play with a few of these datasets, you'll probably run into a few errors, or cases where Spark runs particularly slowly. Up to this point, we've focused on running Spark on smaller datasets, just to get you comfortable with the basic syntax. In the next lesson, we're going to talk about debugging your Spark code, and how to speed it up to work on big data.