**Lesson Overview**

In this lesson, we’ll go over the components of Apache Spark and focus on the ones that we will be using throughout the course. We’ll specifically look at

* Apache Spark Ecosystem
* Overview on Apache Spark’s building blocks (RDD/DataFrame/DataSet)
* Apache Spark Streaming and Structured Streaming
* Usage of Spark UI
* Concepts of Spark DAGs and Stages

**Slides for Each Lesson Available in Resources**

The slide deck for each lesson, seen in the videos, can be found in the Resources tab of the left sidebar of your classroom here.

**Glossary of Key Terms You Will Learn in this Lesson**

* **RDD** (Resilient Distributed Dataset) : The fundamental data structure of the Spark Core component. An immutable distributed collection of objects.
* **DataFrame** : A data structure of the Spark SQL component. A distributed collection of data organized into named columns.
* **Dataset** : A data structure of the Spark SQL component. A distributed collection of data organized into named columns **and also strongly typed**.
* **DAG (Directed Acyclic Graph)**: Each Spark job creates a DAG which consists of task stages to be performed on the clusters.
* **Logical plan** : A pipeline of operations that can be executed as one stage and does not require the data to be shuffled across the partitions — for example, map, filter, etc.
* **Physical plan** : The phase where the action is triggered, and the DAG Scheduler looks at lineage and comes up with the best execution plan with stages and tasks together, and executes the job into a set of tasks in parallel.
* **DAG Scheduler**: DAG scheduler converts a logical execution plan into physical plan.
* **Task** : A task is a unit of work that is sent to the executor.
* **Stage** : A collection of tasks.
* **State** : Intermediary and arbitrary information that needs to be maintained in streaming processing.
* **Lineage Graph**: A complete graph of all the parent RDDs of an RDD. RDD Lineage is built by applying transformations to the RDD.

## Spark Components

#### Core

Contains the basic functionality of Spark. Also home to the API that defines RDDs, which is Spark's main programming abstraction.

#### SQL

Package for working with structured data. It allows querying data via SQL as well as Apache Hive. It supports various sources of data, like Hive tables, Parquet, JSON, CSV, etc.

#### Streaming

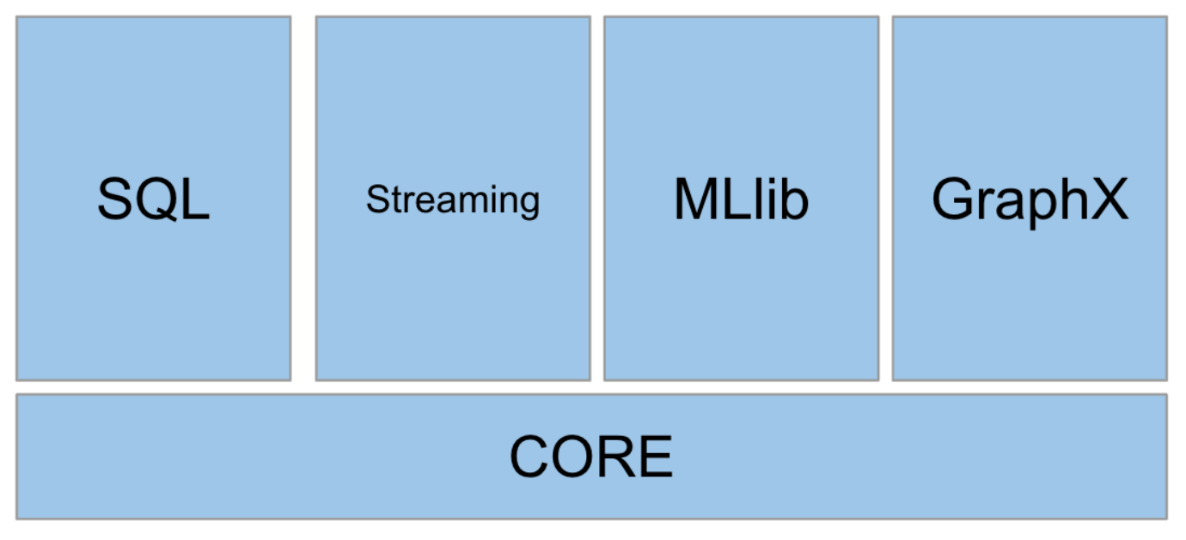
Enables processing of live streams of data. Spark Streaming provides an API for manipulating data streams that are similar to Spark Core's RDD API.

#### MLlib

Provides multiple types of machine learning algorithms, like classification, regression, clustering, etc. This component will not be a focus of this course.

#### GraphX

Library for manipulating graphs and performing graph-parallel computations. This library is where you can find PageRank and triangle counting algorithms. This component will not be a focus of this course.

[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/0bd04178-dc2f-4fed-8b1e-bd29079dc798)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/0bd04178-dc2f-4fed-8b1e-bd29079dc798)

**[Apache Spark Components](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/0bd04178-dc2f-4fed-8b1e-bd29079dc798)**

**RDD Key Points**

* RDD stands for Resilient Distributed Dataset:
  + Resilient because its fault-tolerance comes from maintaining RDD lineage, so even with loss during the operations, you can always go back to where the operation was lost.
  + Distributed because the data is distributed across many partitions and workers.
  + Dataset is a collection of partitioned data. RDD has characteristics like in-memory, immutability, lazily evaluated, cacheable, and typed (we don't see this much in Python, but you'd see this in Scala or Java).

**Code Used in SparkContext Example**

**from** pyspark **import** SparkConf, SparkContext

conf = SparkConf().setMaster("local[2]").setAppName("RDD Example")

sc = SparkContext(conf=conf)

# different way of setting **configurations**

#conf.setMaster('some url')

#conf.set('spark.executor.memory', '2g')

#conf.set('spark.executor.cores', '4')

#conf.set('spark.cores.max', '40')

#conf.set('spark.logConf', **True**)

# sparkContext.parallelize materializes data **into** RDD

# documentation: https:*//spark.apache.org/docs/2.1.1/programming-guide.html#parallelized-collections*

rdd = sc.parallelize([('Richard', 22), ('Alfred', 23), ('Loki',4), ('Albert', 12), ('Alfred', 9)])

rdd.**collect**() # [('Richard', 22), ('Alfred', 23), ('Loki', 4), ('Albert', 12), ('Alfred', 9)]

# create two different RDDs

left = sc.parallelize([("Richard", 1), ("Alfred", 4)])

right = sc.parallelize([("Richard", 2), ("Alfred", 5)])

joined\_rdd = left.**join**(right)

collected = joined\_rdd.**collect**()

collected #[('Alfred', (4, 5)), ('Richard', (1, 2))]

**Code Used in SparkSession Example**

# Notice we’re using pyspark.sql library here

**from** pyspark.sql **import** SparkSession

spark = SparkSession.builder \

.master("local") \

.appName("CSV file loader") \

.getOrCreate()

# couple ways of setting **configurations**

#spark.conf.set("spark.executor.memory", '8g')

#spark.conf.set('spark.executor.cores', '3')

#spark.conf.set('spark.cores.max', '3')

#spark.conf.set("spark.driver.memory", '8g')

file\_path = "./AB\_NYC\_2019.csv"

# Always load csv files with header=**True**

df = spark.**read**.csv(file\_path, header=**True**)

df.printSchema()

df.select('neighbourhood').distinct().show(10, **False**)

## Partitioning in Spark

By default in Spark, a partition is created for each block of the file in HDFS (128MB is the default setting for Hadoop) if you are using HDFS as your file system. If you read a file into an RDD from AWS S3 or some other source, Spark uses 1 partition per 32MB of data. There are a few ways to bypass this default upon creation of an RDD, or reshuffling the RDD to resize the number of partitions, by using rdd.repartition(<the partition number you want to repartition to>). For example, rdd.repartition(10) should change the number of partitions to 10.

In local mode, Spark uses as many partitions as there are cores, so this will depend on your machine. You can override this by adding a configuration parameter spark-submit --conf spark.default.parallelism=<some number>.

So hypothetically, if you have a file of 200 MB and if you were to load this into an RDD, how many partitions will this RDD have? If this file is on HDFS, this will produce 2 partitions (each of them being 128MB). If the file is on AWS S3 or some other file system, it will produce 7 partitions.

### Hash Partitioning

Hash partitioning in Spark is not different than the normal way of using a hash key in the data world to distribute data across partitions uniformly.

Usually this is defined by

partition = key.hashCode() % numPartitions

This mode of partitioning is used when you want to evenly distribute your data across partitions.

### Range Partitioning

Range partitioning is another well-known partitioning method in the data world. Range partitioning divides each partition in a continuous but non-overlapping way.

Let's pretend there is a table called employees, and it has the following schema:

**CREATE** **TABLE** employees (

employee\_id INT **NOT** NULL,

first\_name VARCHAR(30),

last\_name VARCHAR(30),

...

)

Range partitioning would come into play where you partition the employees table by employee\_id, like this:

PARTITION BY RANGE (employee\_id) (

PARTITION p0 VALUES LESS THAN (11),

PARTITION p0 VALUES LESS THAN (21),

PARTITION p0 VALUES LESS THAN (31),

...

)

In reality, you'd want to use range partition over a timestamp, but this example gives you a rough idea of what range partitioning means.

You can use the partitionByRange() function to partition your data into some kind of group. Range partitioning in Spark ensures that every range is contained in a single partition. This becomes useful when you want to reduce shuffles across machines, for example when you know for sure all your parent RDDs need to stay in the same partition.

**DataFrames and Datasets - Key Points**

You can think of DataFrames as tables in a relational database, or dataframes in Python’s pandas library. DataFrames provide memory management and optimized execution plans.

**DataFrames**

DataFrames appeared in Spark Release 1.3.0. We already know that both Datasets and DataFrames are an organized collection of data in columns, but the biggest difference is that DataFrames do not provide type safety. DataFrames are similar to the tables in a relational database. Unlike RDDs, DataFrames and Datasets are part of the spark.sql library, which means you can create a temporary view of these tables and apply SQL queries.

DataFrames allow users to process a large amount of structured data by providing Schema. The Schema is another feature that is very similar to a relational database, indicating types of data that should be stored in the column (String, Timestamp, Double, Long, etc... these are available in spark.sql.types library), and also whether the column can be nullable or not. The aspect that is different from relational databases is that DataFrames and Datasets have no notion of primary/foreign keys - you as a developer define these as you create your DataFrame or Dataset.

**Datasets**

A Dataset is a core building block in SparkSQL that is strongly typed, unlike DataFrames, You can think of Datasets as an extension of the DataFrame API with type-safety. The Dataset API has been available since the release of Spark 1.6. Although Datasets and DataFrames are part of the Spark SQL Component, RDDs, Datasets, and DataFrames still share common features which are: immutability, resilience, and the capability of distributed computing in-memory.

A Dataset provides the features of an RDD and a DataFrame:

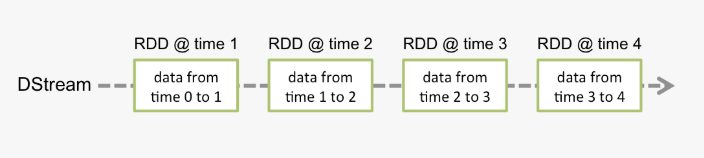
* The convenience of an RDD, as it is an extended library of a Spark DataFrame
* Performance optimization of a DataFrame using Catalyst
* Enforced type-safety
* Datasets are not available in Python, only in Java and Scala. So we won’t be spending much time with Datasets in this course, since we focus on Python.

## Intro to Spark Streaming and DStream

We’ve been primarily looking at batch ingestion but now we’ll start to look at streaming ingestion.

Spark DStream, Discretized Stream, is the basic abstraction and building block of Spark Streaming. DStream is a continuous stream of RDDs. It receives input from various sources like Kafka, Flume, Kinesis, or TCP sockets (we'll mostly be using sockets or Kafka). Another way of generating a Dstream is by operating transformation functions on top of existing DStream.

Another concept added in DStream is that now we're dealing with intervals (or windows).

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/704ece8c-a0cd-4c7f-bf47-b223997449dd)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/704ece8c-a0cd-4c7f-bf47-b223997449dd)**

**[Programming Model for DStream](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/704ece8c-a0cd-4c7f-bf47-b223997449dd)**

**Structured Streaming - Key Points**

Structured Streaming is a programming model, introduced in Spark 2.0, to provide support for building scalable and fault-tolerant applications using Spark SQL.

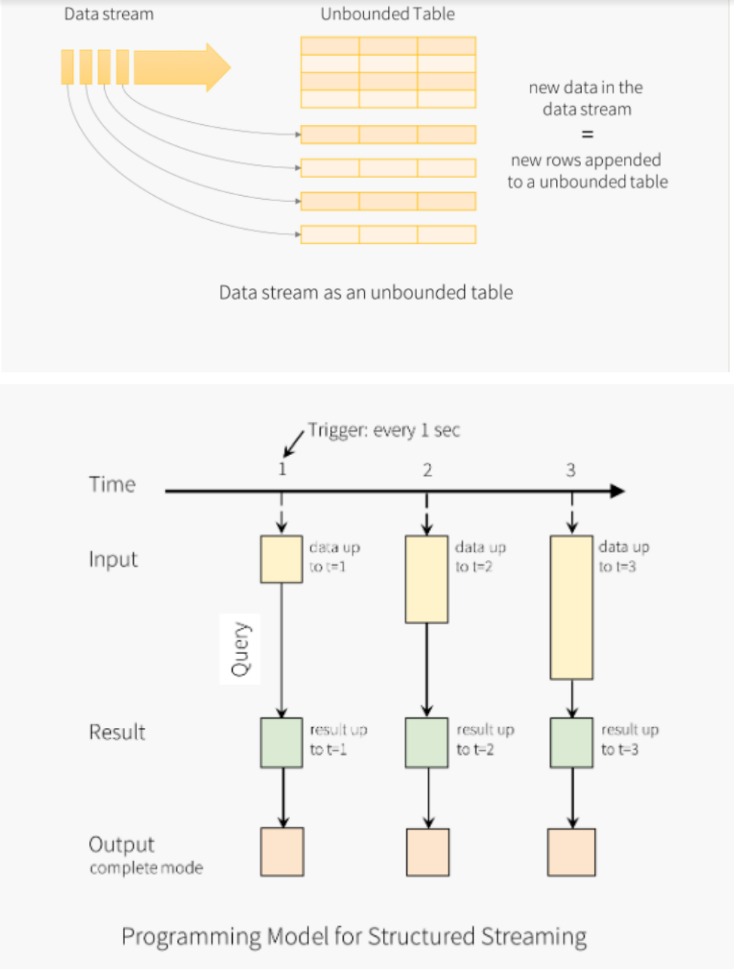
Internally, Structured Streaming is processed using a micro-batch. It processes data streams as a series of small batch jobs.

With Structured Streaming, users/developers don't have to worry about specific issues related to streaming applications, like fault-tolerance, data loss, state loss, or real-time processing of data. The application can now guarantee fault-tolerance using checkpointing.

The advantages of using Structured Streaming are:

* Continuous update of the final result
* Can be used in either Scala, Python, or Java
* Computations are optimized due to using the same Spark SQL component (Catalyst)

**Structured Streaming Architecture**

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/92cb2bec-04ca-427a-853b-c5d10142f4a1)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/92cb2bec-04ca-427a-853b-c5d10142f4a1)**

## Recap on Structured Streaming and State Management

Structured Streaming is a new streaming strategy developed from Discretized Stream. It added a few updates from Dstream, such as decoupling saving state to store to decouple the state management, and also checkpointing metadata. Because these two limitations are decoupled from the application, the developer is now able to exercise fault-tolerant end-to-end execution with ease.

## State Management

Remember that state management is a useful concept in the data streaming world. You can think of state as an intermediate information between micro-batches. There are two major types of state, the metadata of the micro-batch, and the accumulated data derived from processed data (so any data prior to the current micro-batch).

NEXT

## Intro to Spark UI/DAGs

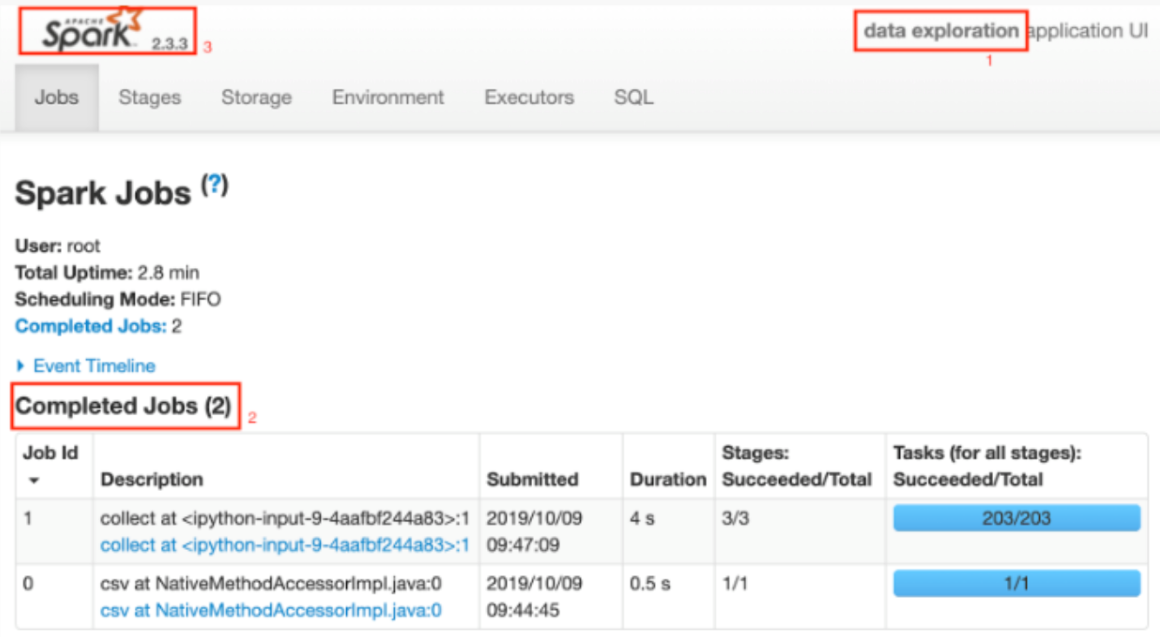
Spark UI is a web interface that gets created when you submit a Spark job. It's a convenient resource for the developer to monitor the status of the job execution. The developer can inspect jobs, stages, storages, environment, and executors in this page, as well as the visualized version of the DAGs (Directed Acyclic Graph) of the Spark job.

## Spark DAGs

At any level, when an action is called on the RDD, Spark generates a DAG. One different thing to note about DAGs is that, unlike Hadoop MapReduce, which creates a Map stage and a Reduce stage, DAGs in Spark can contain many stages.

The DAG scheduler divides operators into stages of tasks, and also puts operators together in the most optimized way.

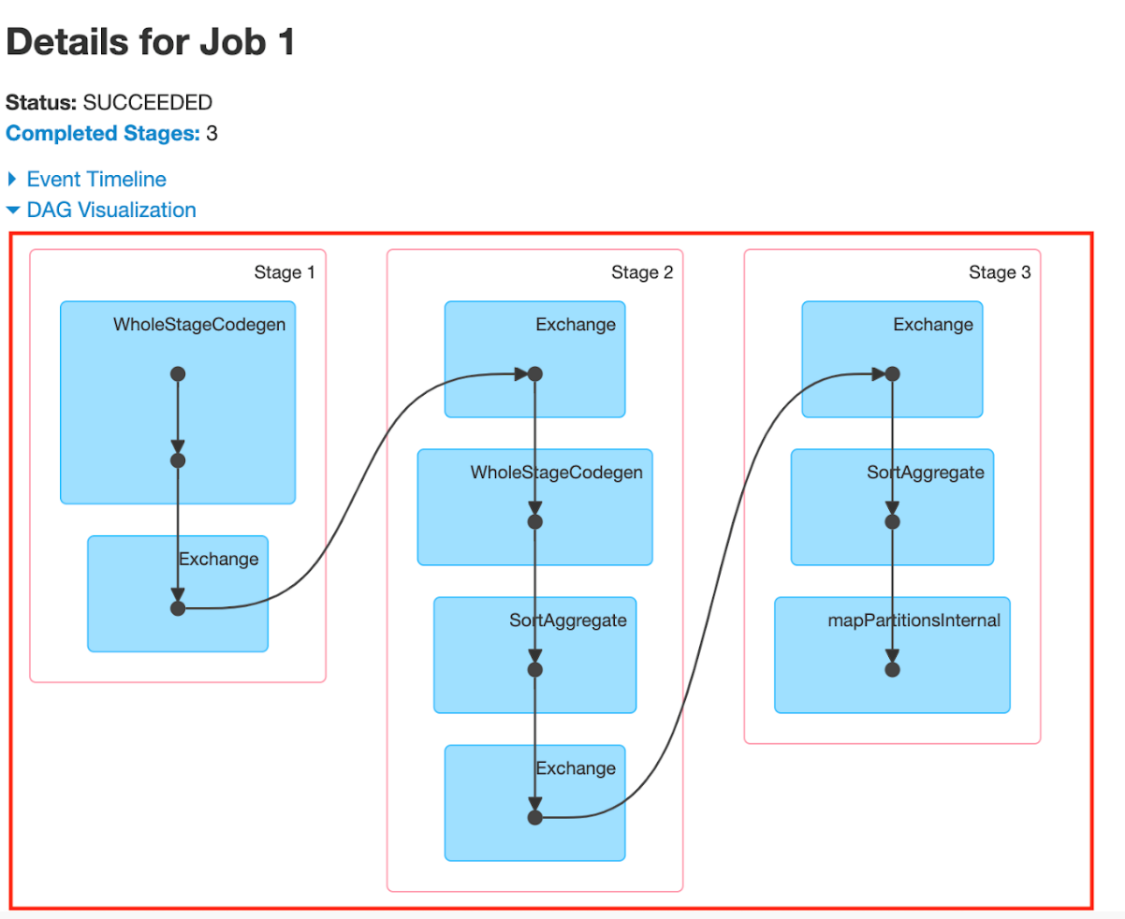
## Spark UI / DAGs Visuals with Explanations

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)**

**[Red 1 - This is where your job name will be from the configuration](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)**

**[Red 2 - Shows the list of action functions that were called](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)**

**[Red 3 - Spark version (other configurations and Spark version can be shown from Environment tab)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)**

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)**

**[Red box annotates DAG of the Job ID 1](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)**

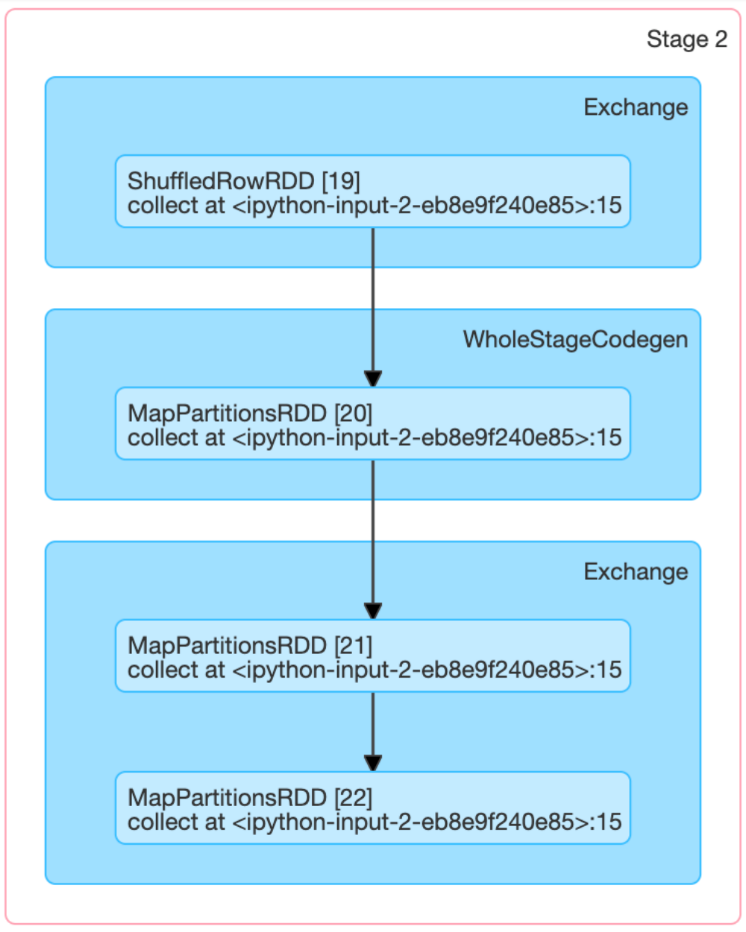
## Spark UI, DAGs - Key Points

DAG is not a new concept created by Spark, rather, if you ever have taken a course about graphs, you see the concept of DAGs fairly often. Spark applied this concept with lazy evaluation and actions (which we’ll be taking a look at in the next lesson) to visualize your work in the UI.

The Spark UI becomes very important when you want to debug at system level. It tells you how many stages you’ve created, the amount of resources you’re using, logs, workers, and a lot of other useful information, like lineage graph, duration of tasks, aggregated metrics, etc.

The lineage graph is the history of RDD transformations, and it’s the graph of all the parent RDDs of the current RDD.

## Lineage Graph Example

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)**

**[Lineage Graph Example](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)**

**[(WholeStageCodegen is a query optimization in Spark SQL that pipes multiple operators together into a single Java function.)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/70b8e08f-3a45-436f-b3b7-cf7f0d727618)**

A lineage graph shows you the history of how the final SparkRDD/DataFrame has been created. With the application of transformation functions, you’re building the lineage graph. A DAG shows you different stages of the Spark job. A compilation of DAGs could be the lineage graph, but a DAG contains more information than just stages - it tells you where the shuffles occurred, actions occurred, and with transformations, what kind of RDDs/DataFrames have been generated.

## Example of Spark UI, DAGs, stages and code

This is the code that was used to generate the image below.

**from** pyspark.sql **import** SparkSession

spark = SparkSession.builder \

.config('spark.ui.port', 3000) \

.master("local[2]") \

.appName("data exploration") \

.getOrCreate()

spark.conf.set('spark.executor.memory', '3g')

spark.conf.set('spark.executor.cores', '3g')

df = spark.**read**.csv('AB\_NYC\_2019.csv', header=**True**)

df1 = df.select('neighbourhood', 'price').distinct()

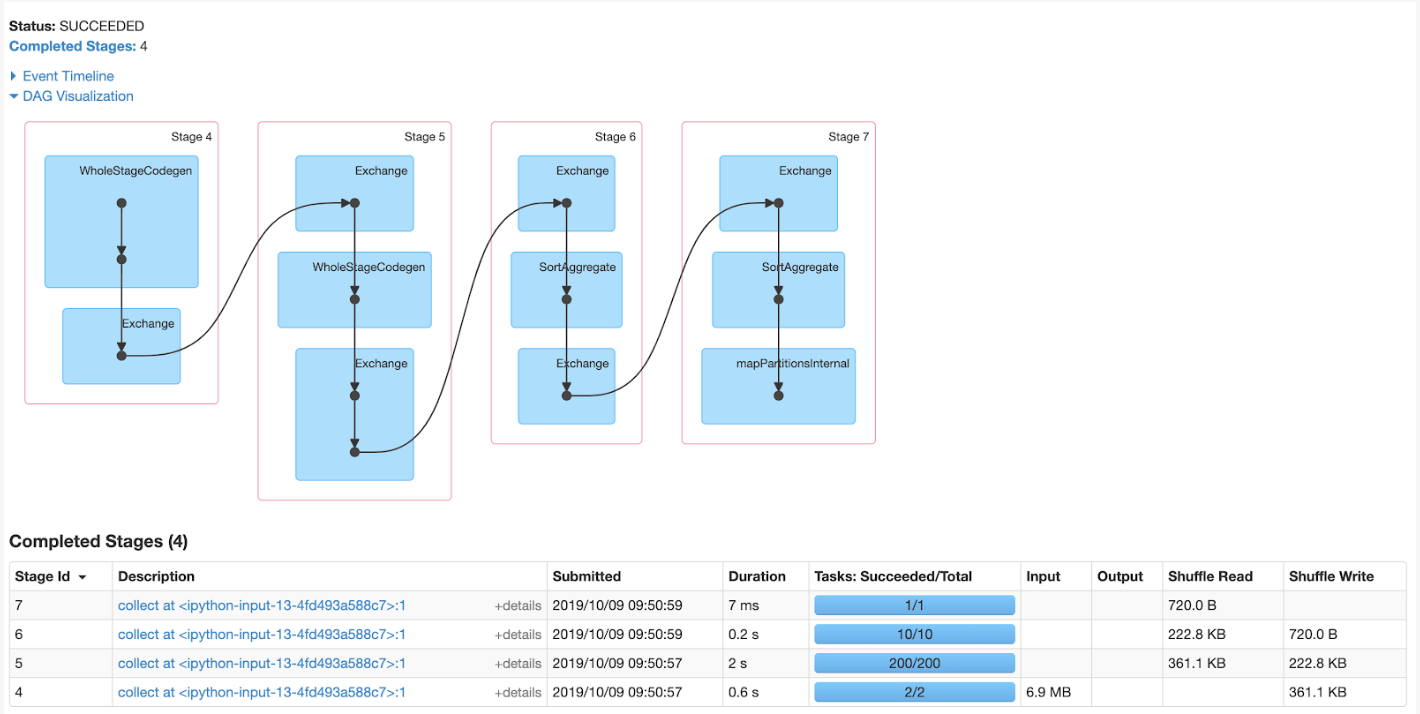
**import** pyspark.sql.functions

df1.rdd.getNumPartitions()

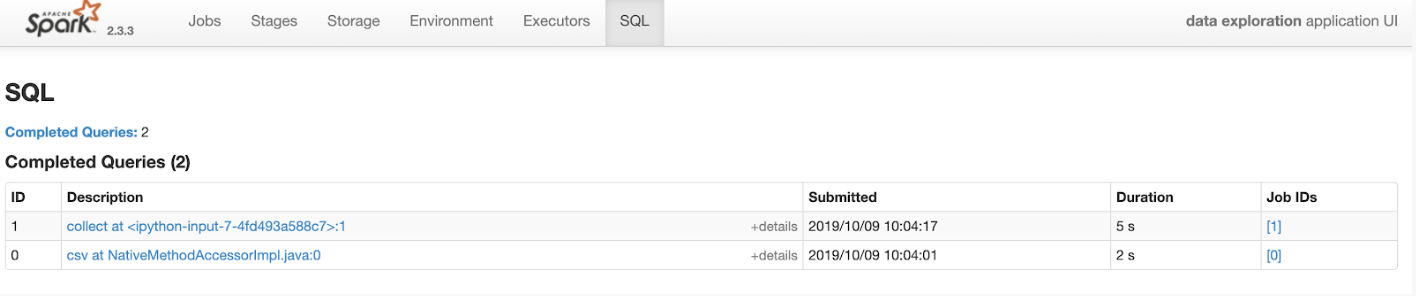
df1.repartition(10).agg({"price": "max"}).**collect**()

This will give you some nonsense data at the end, but we can take a closer look at how the tasks were split.

Since the code annotates local[2], it's using 2 partitions at the beginning. local[\*] means local[{Runtime.getRuntime.availableProcessors()}]. And then depending on the data, there are 200 tasks. Then there is repartition(10) which brought down the number of tasks to 10. Finally the last collect() has 1 task.

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/aaad3a67-3a04-4f48-bd45-dd215f5f0bd9)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/aaad3a67-3a04-4f48-bd45-dd215f5f0bd9)**

**[Spark UI Stages Tab](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/aaad3a67-3a04-4f48-bd45-dd215f5f0bd9)**

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/aaad3a67-3a04-4f48-bd45-dd215f5f0bd9)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/aaad3a67-3a04-4f48-bd45-dd215f5f0bd9)**

**[SQL Tab Seen When You Use SparkSession (If used with SparkContext, you will not see the SQL tab though, since SparkContext is the entry point for RDD, which is in the Core component.)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/aaad3a67-3a04-4f48-bd45-dd215f5f0bd9)**

**[[](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/aaad3a67-3a04-4f48-bd45-dd215f5f0bd9)](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/aaad3a67-3a04-4f48-bd45-dd215f5f0bd9)**

**[Detailed Logical Plan and Physical Plan of Spark Job](https://classroom.udacity.com/nanodegrees/nd029/parts/d3d2cbfa-dc05-44c3-8db8-84e1d931170d/modules/36ddbb88-7b71-4e78-983c-5a0890eb1ec2/lessons/952320e3-fbe2-44fa-ab23-2acc8ce14e68/concepts/aaad3a67-3a04-4f48-bd45-dd215f5f0bd9)**

# What is a Schema?

Generally, a schema is the description of the structure of your data. It tells you how your data is organized - you can say it’s the blueprint of your data. DataFrames and Datasets use this concept when you create DataFrame and Dataset during run time (implicit) or compile time (explicit).

StructField objects are in tuple (name, type of your data, and nullified represented in True/False), and you need to wrap these objects in StructType to build a schema.

StructType and StructField belong to the org.apache.spark.sql.types package so these need to be imported separately.

## Creating a DataFrame or Dataset using a Schema - Summary

Creating a Schema helps eliminate some errors that can arise while generating your DataFrame.

A Dataset is already type-safe but because it’s a feature not available in Python, we’ll use StructType to build schema for a DataFrame. In this case, a DataFrame’s schema can be represented by StructType and we can apply this schema through the createDataFrame function of SparkSession object.

**Lesson Summary**

In this lesson, we learned the fundamentals of Spark RDDs and DataFrames, and how we can leverage the Spark Web UI to efficiently monitor and debug Spark jobs. As a data engineer, you will always be monitoring through Spark Web UI to visualize if your code is optimized. Depending on your business needs, you will now be able to select which core building blocks to use (RDD vs DataFrame, or DataSet if you can use Scala or Java), and also run a simple SQL-like analysis on your data.

**Further Optional Reading**

* [**Spark UI**](https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html)
* [**Project Tungsten**](https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html) is a side project in Databricks to optimize Spark jobs for CPU and memory efficiency.
* [**Whole Stage CodeGen**](https://issues.apache.org/jira/browse/SPARK-12795)
* [**More on Whole Stage CodeGen**](http://www.vldb.org/pvldb/vol4/p539-neumann.pdf)
* NEXT