**Spark and what it does:**

Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters.

**Unified :**

Spark is designed to support a wide range of data analytics tasks, ranging from simple data loading and SQLqueries to machine learning and streaming computation, over the same computing engine and with a consistent set of APIs.

First, Spark provides consistent, composable APIs that you can use to build an application out of smaller pieces or out of existing libraries. It also makes it easy for you to write your own analytics libraries on top. However, composable APIs are not enough: Spark’s APIs are also designed to enable high performance by optimizing across the different libraries and functions composed together in a user program. For example, if you load data using a SQLquery and then evaluate a machine learning model over it using Spark’s MLlibrary, the engine can combine these steps into one scan over the data.

**Computing engine :**

Hadoop= Storage + Computation Engine 🡪 Closely /Integrated

This makes it difficult to run one of the systems without the other.

More important, this choice also makes it a challenge to write applications that access data stored anywhere else.

You can use Spark with a wide variety of persistent storage systems, including

->cloud storage systems ( Azure Storage and Amazon S3),

-> distributed file systems (Apache Hadoop, key-value stores such as Apache Cassandra, and message buses such as Apache Kafka)

The key motivation here is that most data already resides in a mix of storage systems. Data is expensive to move so Spark focuses on performing computations over the data, no matter where it resides.

**Libraries:**

Spark supports both standard libraries that ship with the engine as well as a wide array of external libraries published as third-party packages by the open source communities.

Spark includes libraries for SQLand structured data (Spark SQL), machine learning (MLlib), stream processing (Spark Streaming and the newer Structured Streaming), and graph analytics (GraphX). Beyond these libraries, there are hundreds of open source external libraries ranging from connectors for various storage systems to machine learning algorithms.

One index of external libraries is available at *spark-packages.org.*

Running spark:

There are two options we recommend for getting started with Spark:

1. Downloading and installing Apache Spark on your laptop,
2. Running a web-based version in Databricks Community Edition, a free cloud environment for learning Spark.

**Spark Architecture**



A cluster, or group, of computers, pools the resources of many machines together, giving us the ability to use all the cumulative resources as if they were a single computer.

Now, you need a framework to coordinate work across them. Spark does just that, managing and coordinating the execution of tasks on data across a cluster of computers. The cluster of machines that Spark will use to execute tasks is managed by a cluster manager like Spark’s standalone cluster manager, YARN, or Mesos.

**Spark Applications**

Spark Applications consist of **a driver process** and a **set of executor processes**.

The driver process runs your main() function, sits on a node in the cluster and is responsible for three things:

* maintaining information about the Spark Application
* responding to a user’s program or input
* analyzing, distributing, and scheduling work across the executors (discussed momentarily).

The executors are responsible for actually carrying out the work that the driver assigns them. This means that each executor is responsible for only two things: executing code assigned to it by the driver, and reporting the state of the computation on that executor back to the driver node

Cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of three core cluster managers:

* Spark’s standalone cluster manager
* YARN
* Mesos

**Spark Language APIs:**

-Scala

-Java

-Python

-R

**SparkSession**

You control your Spark Application through a driver process called the SparkSession.

The SparkSession instance is the way Spark executes user-defined manipulations across the cluster.

***One-to-one correspondence between a SparkSession and a Spark Application.***

**DataFrames**

A DataFrame is the most common Structured API and simply represents a table of data with rows and columns.

The list that defines the columns and the types within those columns is called the schema. You can think of a DataFrame as a spreadsheet with named columns

**Partitions**

To allow every executor to perform work in parallel, Spark breaks up the data into chunks called partitions. A partition is a collection of rows that sit on one physical machine in your cluster. A DataFrame’s partitions represent how the data is physically distributed across the cluster of machines during execution.

**Transformations**

In Spark, the core data structures are immutable, meaning they cannot be changed after they’re created.

To “change” a DataFrame, you need to instruct Spark how you would like to modify it to do what you want. These instructions are called transformations.

Spark will not act on transformations until we call an action (we discuss this shortly). Transformations are the core of how you express your business logic using Spark.

There are two types of transformations: those that specify *narrow dependencies*, and those that specify *wide dependencies*.

**Narrow Transformations:**

Transformations for which each input partition will contribute to only one output partition

**Wide Transformations**

A wide dependency (or wide transformation) style transformation will have input partitions contributing to many output partitions. You will often hear this referred to as a shuflle whereby Spark will exchange partitions across the cluster.

**Pipelining:**

With narrow transformations, Spark will automatically perform an operation called pipelining, meaning that if we specify multiple filters on DataFrames, they’ll all be performed in-memory. The same cannot be said for shuffles. When we perform a shuffle, Spark writes the results to disk

**Lazy Evaluation**

Lazy evaulation means that Spark will wait until the very last moment to execute the graph of computation instructions. In Spark, instead of modifying the data immediately when you express some operation, you build up a plan of transformations that you would like to apply to your source data. By waiting until the last minute to execute the code, Spark compiles this plan from your raw DataFrame transformations to a streamlined physical plan that will run as efficiently as possible across the cluster. This provides immense benefits because Spark can optimize the entire data flow from end to end.

***An example of this is something called predicate pushdown on DataFrames. If we build a large Spark job but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need. Spark will actually optimize this for us by pushing the filter down automatically.***

**Actions**

Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an action. An action instructs Spark to compute a result from a series of transformations. The simplest action is count, which gives us the total number of records in the DataFrame

There are three kinds of actions:

Actions to view data in the console

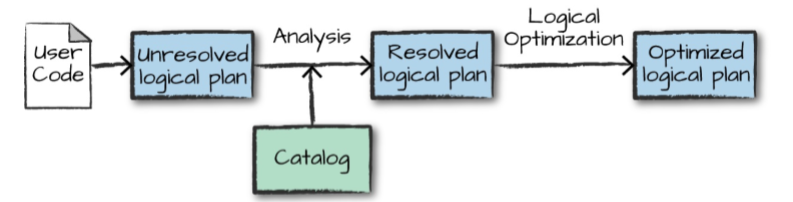
Actions to collect data to native objects in the respective language

Actions to write to output data sources

In specifying this action, we started a Spark job that runs our filter transformation (a narrow transformation), then an aggregation (a wide transformation) that performs the counts on a per partition basis, and then a collect, which brings our result to a native object in the respective language. You can see all of this by inspecting the Spark UI, a tool included in Spark with which you can monitor the Spark jobs running on a cluster. Spark UI You can monitor the progress of a job through the Spark web UI. The Spark UI is available on port 4040 of the driver node. If you are running in local mode, this will be http://localhost:4040. The Spark UI displays information on the state of your Spark jobs, its environment, and cluster state. It’s very useful, especially for tuning and debugging. Figure 2-6 shows an example UI for a Spark job where two stages containing nine tasks were executed.

**Overview of Structured API Execution**

This section will demonstrate how this code is actually executed across a cluster. This will help you understand (and potentially debug) the process of writing and executing code on clusters,



Here’s an overview of the steps:

1. Write DataFrame/Dataset/SQLCode.

2. If valid code, Spark converts this to a Logical Plan.

3. Spark transforms this Logical Plan to a Physical Plan, checking for optimizations along the way.

4. Spark then executes this Physical Plan (RDD manipulations) on the cluster.

To execute code, we must write code. This code is then submitted to Spark either through the console or via a submitted job. This code then passes through the Catalyst Optimizer, which decides how the code should be executed and lays out a plan for doing so before, finally, the code is run and the result is returned to the user

**Logical Planning**

The first phase of execution is meant to take user code and convert it into a logical plan.

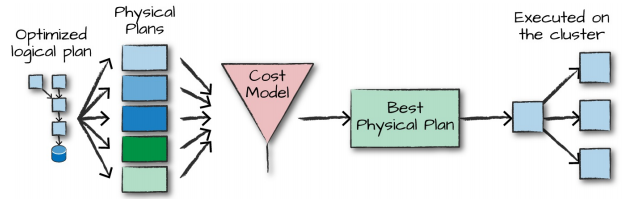
This logical plan only represents a set of abstract transformations that do not refer to executors or drivers, it’s purely to convert the user’s set of expressions into the most optimized version.

It does this by converting user code into an unresolved logical plan.

This plan is unresolved because although your code might be valid, the tables or columns that it refers to might or might not exist.

Spark uses the catalog, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer. The analyzer might reject the unresolved logical plan if the required table or column name does not exist in the catalog. If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections. Packages can extend the Catalyst to include their own rules for domainspecific optimizations. Physical Planning After successfully creating an optimized logical plan, Spark then begins the physical planning process.

**The physical plan,** often called a Spark plan, specifies how the logical plan will execute on the cluster by generating different



An example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table (how big the table is or how big its partitions are

This result is why you might have heard Spark referred to as a compiler—it takes queries in DataFrames, Datasets, and SQLand compiles them into RDD transformations for you.

**Execution**

Upon selecting a physical plan, Spark runs all of this code over RDDs, the lower-level programming interface of Spark (which we cover in Part III). Spark performs further optimizations at runtime, generating native Java bytecode that can remove entire tasks or stages during execution. Finally the result is returned to the user.

**Spark API Toolset**

Spark is composed of these primitives— the lower-level APIs and the Structured APIs—and then a series of standard libraries for additional functionality.

Rdds are low level APIs

Dataframes are High Level APIs

RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers *transformations* and *actions*.

[DataFrame](https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html) is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction

When to use RDDs:

* you want low-level transformation and actions and control on your dataset;
* your data is unstructured, such as media streams or streams of text;
* you don’t care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column; and
* you can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data.

**Spark’s libraries** support a variety of different tasks, from graph analysis and machine learning to streaming and integrations with a host of computing and storage systems

**Running Production Applications**

***spark-submit***, a built-in command-line tool.

* It lets you send your application code to a cluster and launch it to execute there.
* Upon submission, the application will run until it exits (completes the task) or encounters an error. You can do this with all of Spark’s support cluster managers including Standalone, Mesos, and YARN.

It offers several controls with which you can specify the resources your application needs as well as how it should be run and its command-line arguments.

**Dataframes and Datasets:**

DataFrames and Datasets are (distributed) table-like collections with well-defined rows and columns. Each column must have the same number of rows as all the other columns (although you can

use null to specify the absence of a value) and each column has type information that must be consistent for every row in the collection.

To Spark, DataFrames and Datasets represent immutable, lazily evaluated plans that specify what operations to apply to data residing at a location to generate some output. When we perform an action on a DataFrame, we instruct Spark to perform the actual transformations and return the result. These represent plans of how to manipulate rows and columns to compute the user’s desired result.

Difference between Dataframe and Dataset:

DataFrames are un-typed and datasets are typed.

Datasets are only available to Java Virtual Machine (JVM)–based languages (Scala and Java) and we specify types with case classes or Java beans. For the most part, you’re likely to work with DataFrames. To Spark (in Scala), DataFrames are simply Datasets of Type Row. The “Row” type is Spark’s internal representation of its optimized in-memory format for computation. This format makes for highly specialized and efficient computation because rather than using JVM types, which can cause high garbage-collection and object instantiation costs, Spark can operate on its own internal format without incurring any of those costs.

To Spark (in Python or R), there is no such thing as a Dataset: everything is a DataFrame and therefore we always operate on that optimized format.

**Datasets: Type Structured APIs**

Writing statically typed code in Java and Scala. The Dataset API is not available in Python and R, because those languages are dynamically typed.

The Dataset API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of typed objects, similar to a Java ArrayList or Scala Seq. The APIs available on Datasets are type-safe, meaning that you cannot accidentally view the objects in a Dataset as being of another class than the class you put in initially.

**When to use Dataset:**

**The Dataset API is type-safe**. Operations that are not valid for their types, say subtracting two string types, will fail at compilation time not at runtime. If correctness and bulletproof code is your highest priority, at the cost of some performance, this can be a great choice for you.

**Basic Structured API Operations:**

**Actions and Transformations are same for both datasets and dataframes**

**Schemas: A schema defines the column names and types of a DataFrame. We can either let a data source define the schema (called schema-on-read) or we can define it explicitly ourselves**

A schema is a StructType made up of a number of fields, StructFields, that have a name, type, a Boolean flag which specifies whether that column can contain missing or null values, and, finally, users can optionally specify associated metadata with that column. The metadata is a way of storing information about this column (Spark uses this in its machine learning library). Schemas can contain other StructTypes (Spark’s complex types).

**df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")**

**df.printSchema()**

**df.schema**

**df = spark.read.format("json").schema(myManualSchema).load("/data/flight-data/json/2015-summary.json")**

**Columns:**

from pyspark.sql.functions import col, column

col("someColumnName")

column("someColumnName")

df.col(“someColumnName”)

df.columns

**Row:**

Some functions:

df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json") df.createOrReplaceTempView("dfTable")

from pyspark.sql import Row

from pyspark.sql.types import StructField, StructType, StringType, LongType

myManualSchema = StructType([ StructField("some", StringType(), True), StructField("col", StringType(), True), StructField("names", LongType(), False) ])

myRow = Row("Hello", None, 1)

myDf = spark.createDataFrame([myRow], myManualSchema)

myDf.show()

**select and selectExpr :**

select and selectExpr allow you to do the DataFrame equivalent of SQL queries on a table of data:

# in Python df.select("DEST\_COUNTRY\_NAME").show(2)

-- in SQL

SELECT DEST\_COUNTRY\_NAME FROM dfTable LIMIT 2

You can select multiple columns by using the same style of query, just add more column name strings to your select method call:

# in Python df.select("DEST\_COUNTRY\_NAME", "ORIGIN\_COUNTRY\_NAME").show(2)

-- in SQL SELECT DEST\_COUNTRY\_NAME, ORIGIN\_COUNTRY\_NAME FROM dfTable LIMIT 2

df.select(expr("DEST\_COUNTRY\_NAME as destination").alias("DEST\_COUNTRY\_NAME"))\

.show(2)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*8

Spark has a shortcut for this:

# in Python df.selectExpr("DEST\_COUNTRY\_NAME as newColumnName", "DEST\_COUNTRY\_NAME").show(2)

**Converting to Spark Types (Literals)**

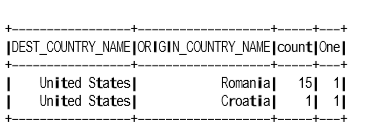
Sometimes, we need to pass explicit values into Spark that are just a value (rather than a new column). This might be a constant value or something we’ll need to compare to later on. The way we do this is through literals.

**# in Python from pyspark.sql.functions import lit**

**df.select(expr("\*"), lit(1).alias("One")).show(2)**

In SQL, literals are just the specific value:

-- in SQL SELECT \*, 1 as One FROM dfTable LIMIT 2



**Adding Columns**

# in Python df.withColumn("numberOne", lit(1)).show(2)

-- in SQL SELECT \*, 1 as numberOne FROM dfTable LIMIT 2

**Renaming Columns**

# in Python df.**withColumnRenamed**("DEST\_COUNTRY\_NAME", "dest").columns

**Removing Columns**

df.drop("ORIGIN\_COUNTRY\_NAME").columns

We can drop multiple columns by passing in multiple columns as arguments:

dfWithLongColName.drop("ORIGIN\_COUNTRY\_NAME", "DEST\_COUNTRY\_NAME")

**Changing a Column’s Type (cast)**

df.withColumn("count2", col("count").cast("long"))

**Filtering Rows**

To filter rows, we create an expression that evaluates to true or false. You then filter out the rows with an expression that is equal to false.

df.filter(col("count") < 2).show(2) df.where("count < 2").show(2)

**Getting Distinct Row**

df.select("ORIGIN\_COUNTRY\_NAME", "DEST\_COUNTRY\_NAME").distinct().count()

df.select("ORIGIN\_COUNTRY\_NAME").distinct().count()

**Concatenating and Appending Rows (Union)**

**Df1.union(df2)**

**\*schema should be same**

**Sorting Rows**

# in Python df.sort("count").show(5) df.orderBy("count", "DEST\_COUNTRY\_NAME").show(5) df.orderBy(col("count"), col("DEST\_COUNTRY\_NAME")).show(5)

To more explicitly specify sort direction, you need to use the asc and desc functions if operating on a column. These allow you to specify the order in which a given column should be sorted:

# in Python from pyspark.sql.functions import desc, asc df.orderBy(expr("count desc")).show(2) df.orderBy(col("count").desc(), col("DEST\_COUNTRY\_NAME").asc()).show(2)

**Limit**

**d.limit(5).show()**

**Repartition and Coalesce**

Repartition will incur a full shuffle of the data, regardless of whether one is necessary. This means that you should typically only repartition when the future number of partitions is greater than your current number of partitions or when you are looking to partition by a set of columns:

**df.rdd.getNumPartitions()**

**df.repartition(5)**

If you know that you’re going to be filtering by a certain column often, it can be worth repartitioning based on that column

**df.repartition(col("DEST\_COUNTRY\_NAME"))**

Coalesce, on the other hand, will not incur a full shuffle and will try to combine partitions. This operation will shuffle your data into five partitions based on the destination country name, and then coalesce them (without a full shuffle):

**df.repartition(5, col("DEST\_COUNTRY\_NAME")).coalesce(2)**

**Collecting Rows to the Driver**

**Spark maintains the state of the cluster in the driver. There are times when you’ll want to collect some of your data to the driver in order to manipulate it on your local machine.**

collectDF = df.limit(10) collectDF.take(5) # take works with an Integer count

collectDF.show() # this prints it out nicely

collectDF.show(5, False)

collectDF.collect()

There’s an additional way of collecting rows to the driver in order to iterate over the entire dataset. The method toLocalIterator collects partitions to the driver as an iterator. This method allows you to iterate over the entire dataset partition-by-partition in a serial manner:

**collectDF.toLocalIterator()**

**Note;**

Any collection of data to the driver can be a very expensive operation! If you have a large dataset and call collect, you can crash the driver. If you use toLocalIterator and have very large partitions, you can easily crash the driver node and lose the state of your application. This is also expensive because we can operate on a one-by-one basis, instead of running computation in parallel.