



**ILGlabs  
GNUGroup**

# What Is Apache Spark?



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

## Benefits

Spark supports multiple widely used programming languages (Python, Java, Scala, and R).

Includes libraries for diverse tasks ranging from SQL to streaming and machine learning, and

Runs anywhere from a laptop to a cluster of thousands of servers.

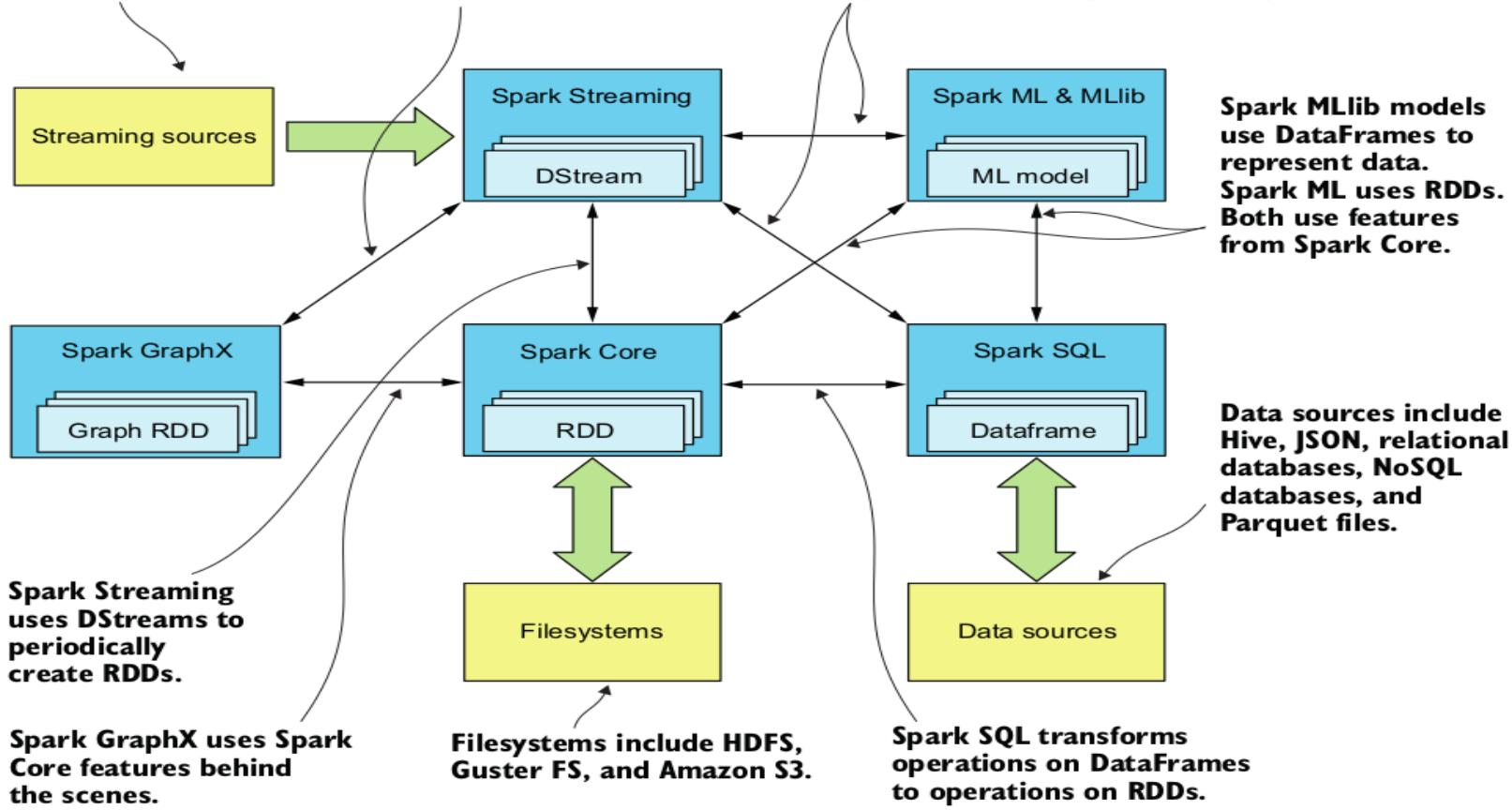
# Main Spark components and various runtime interactions and storage options



Streaming sources include Kafka, Flume, Twitter, HDFS, and ZeroMQ.

Spark Streaming can use GraphX features on the data it receives.

Spark Streaming can use machine-learning models and Spark SQL to analyze streaming data.



# Spark's toolkit



Structured  
Streaming

Advanced  
Analytics

Libraries &  
Ecosystem

Structured APIs

Datasets

DataFrames

SQL

Low-level APIs

RDDs

Distributed Variables

# Install Spark



## 1) Download Spark

<http://apachemirror.wuchna.com/spark/spark-2.4.4/spark-2.4.4-bin-hadoop2.7.tgz>

## 2) Installing Spark

Follow the steps given below for installing Spark.

### Extracting Spark tar

The following command for extracting the spark tar file.

```
$ tar xvf spark-1.3.1-bin-hadoop2.6.tgz
```

### Moving Spark software files

The following commands for moving the Spark software files to respective directory (/usr/local/spark).

```
$ su -
```

Password:

```
# cd /home/ilg/Downloads/  
# mv spark-1.3.1-bin-hadoop2.6 /usr/local/spark  
# exit
```



# Set the environment

## Setting up the environment for Spark

Add the following line to `~/.bashrc` file. It means adding the location, where the spark software file are located to the PATH variable.

```
export PATH=$PATH:/usr/local/spark/bin
```

Use the following command for sourcing the `~/.bashrc` file.

```
$ source ~/.bashrc
```

## Verify the Installation by launching

Write the following command for opening Spark shell.

```
$ spark-shell
```

OR

```
$ pyspark
```

Note: \$ is shell prompt.



# Spark's Architecture

# Framework of Spark, Cluster



*Spark is a framework* to coordinate a cluster, or group, of computers, for pooling the resources of many machines together, giving us the ability to use all the cumulative resources as if they were a single computer.

Cluster of machines that Spark will use to execute tasks is managed by a cluster manager like Spark's **stand alone** cluster manager, **YARN**, or **Mesos**.



# Spark Processing

Core architecture is master/slave

Driver node is the controller and the head in the setup.

This is where the status and state of everything in the cluster is being kept.

It's also from where you send your tasks for execution.

```
#####
```

When the job is being sent to the driver, a SparkContext is being initiated. This is the way your code communicates with Spark.

The driver program then looks at the job, slices it up into tasks, creates a plan (called a Directed Acyclic Graph [DAG] and figures out what resources it needs.

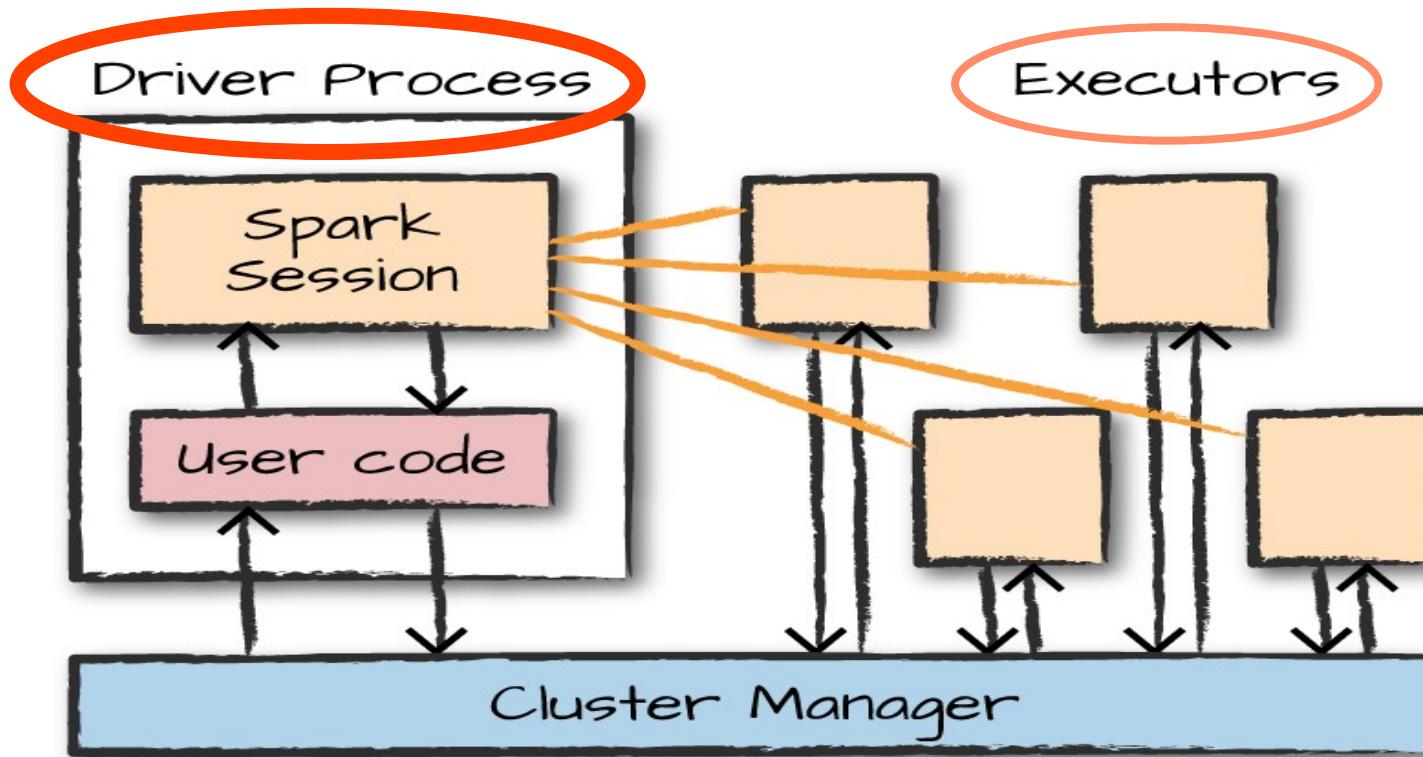
Driver then checks with the cluster manager for these resources. When possible, the cluster manager starts up the needed executor processes on the worker nodes. It tells the driver about them.

The tasks are then being sent from the driver directly to the executors. They do what they are asked for and deliver the results.

They communicate back to the driver so that it knows the status of the full job. Once all of the job is done, all the executors are freed up by the cluster manager.

They will then be available for the next job to be handled.

# Spark Applications



# Driver Process



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

- 1) Maintaining information about the Spark Application
- 2) Responding to a user's program or input
- 3) Analysing, distributing, and scheduling work across the executors

# Executors



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:

- 1) Executing code assigned to it by the driver.
- 2) Reporting the state of the computation on that executor back to the driver node.

# Cluster Manager



Cluster manager controls physical machines and allocates resources to Spark Applications.

This can be one of three core cluster managers:

- 1) Spark's standalone cluster manager,
- 2) YARN
- 3) Mesos.

# Spark's API



- Scala

Spark is primarily written in Scala, making it Spark's "default" language.

- Java

Even though Spark is written in Scala, Spark's authors have been careful to ensure that you can write Spark code in Java.

- Python

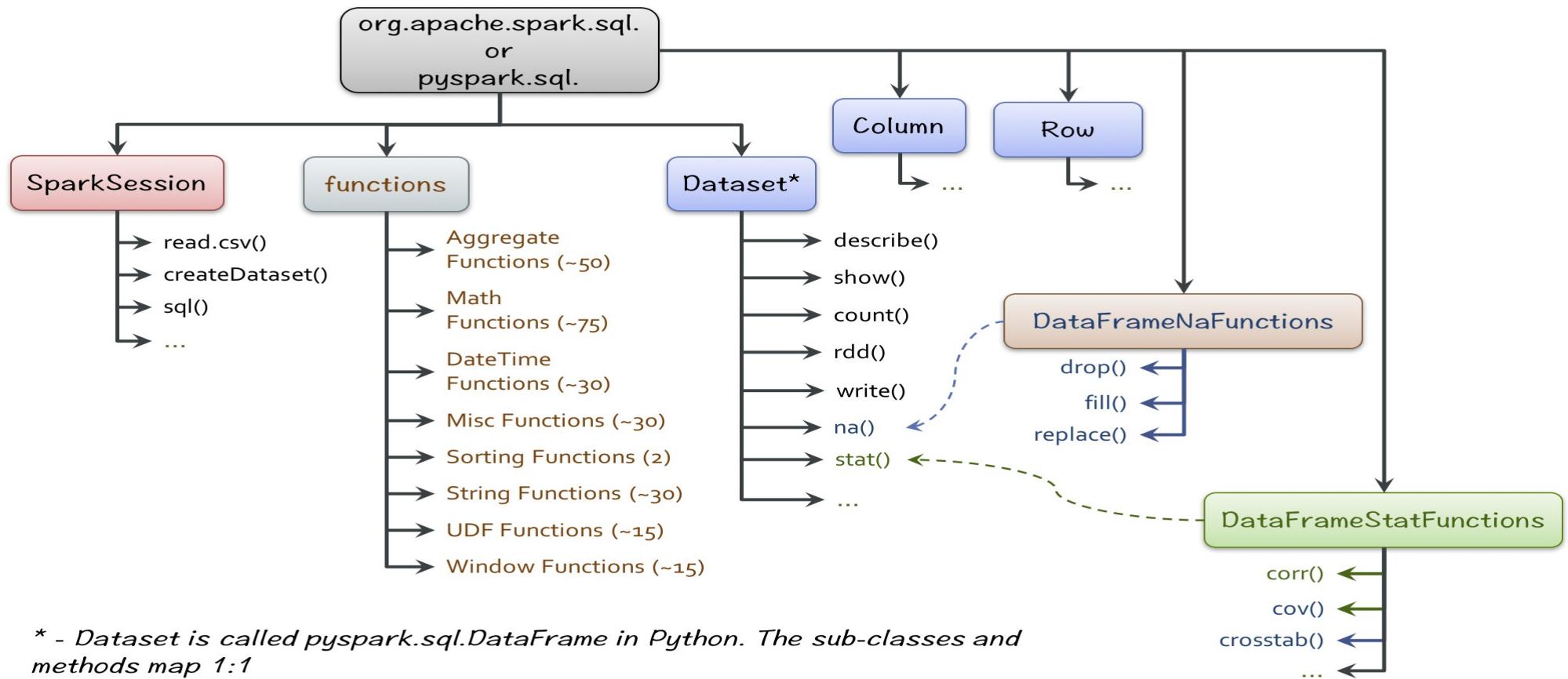
Python supports nearly all constructs that Scala supports.

- SQL

Spark supports a subset of the ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to take advantage of the big data powers of Spark.

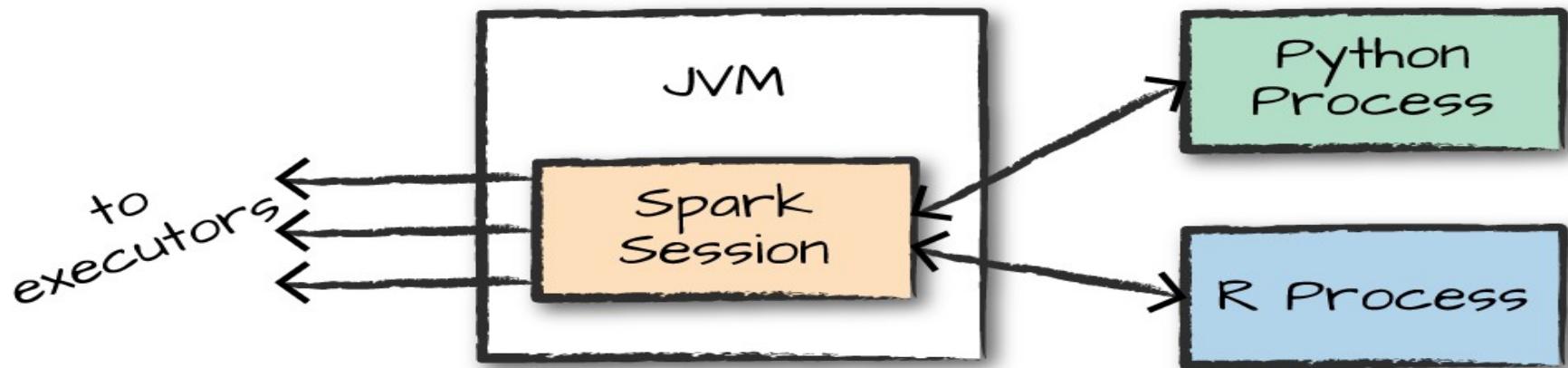
- R

Spark has two commonly used R libraries: one as a part of Spark core (SparkR) and another as an R community-driven package (sparklyr).





## The relationship between the SparkSession and Spark's Language API



# API – low level & High Level



Spark has two fundamental sets of APIs:

- 1) Low-level “unstructured” APIs
- 2) Higher-level structured APIs.

# pyspark



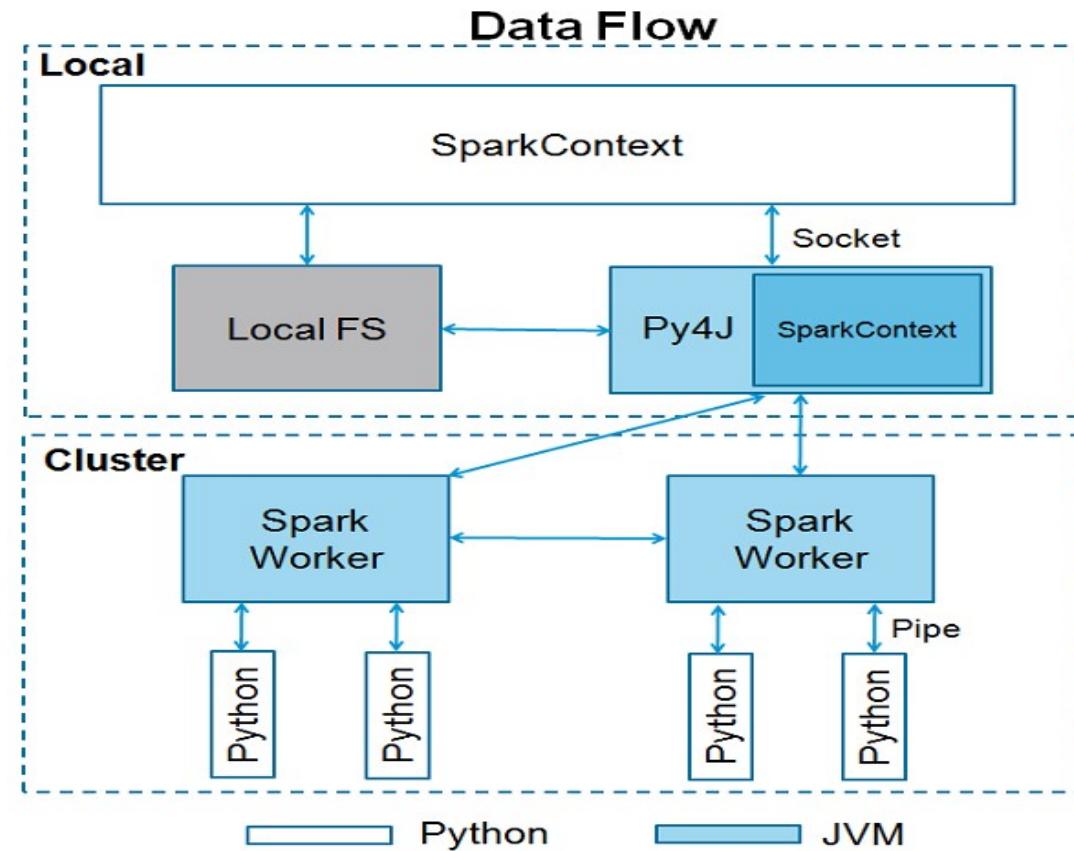
Apache Spark is written in Scala programming language.

To support Python with Spark, Apache Spark Community released a tool, PySpark.

Using PySpark, you can work with RDDs in Python programming language also. It is because of a library called Py4j that they are able to achieve this.

PySpark offers PySpark Shell which links the Python API to the spark core and initializes the Spark context.

*Majority of data scientists and analytics experts today use Python because of its rich library set. Integrating Python with Spark is a boon to them.*



# Starting Spark



To do this, we will start Spark's local mode.

## Scala

This means running **spark-shell** to access the Scala console to start an interactive session.

**\$ spark-shell**

## Pyspark

You can also start the Python console by using **pyspark**. This starts an interactive Spark Application.

**\$ pyspark**

## Spark submit

Process for submitting standalone applications to Spark called

**\$ spark-submit**

whereby you can submit a precompiled application to Spark.

# Spark Session



When you start Spark in this interactive mode, you implicitly create a **SparkSession** that manages the Spark Application.

*When we start it through a standalone application, we must create the SparkSession object yourself in your application code.*

# SparkSession





## E.g

Let's now perform the simple task of creating a range of numbers. This range of numbers is just like a named column in a spreadsheet:

```
In [2]: myRange = spark.range(5000).toDF('number')  
In [3]: myRange  
Out[3]: DataFrame[number: bigint]
```

We created a DataFrame with one column containing 5,000 rows with values from 0 to 4999.

This range of numbers represents a distributed collection.

When run on a cluster, each part of this range of numbers exists on a different executor. **This is a Spark DataFrame.**

# DataFrames



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**.*

We can think of a DataFrame as a spreadsheet with named columns.

**A Spark DataFrame can span thousands of computers.**

Spreadsheet on  
a single machine

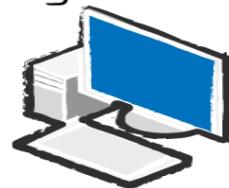


Table or Data Frame  
partitioned across servers  
in a data center



# 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*.

If we have **one partition**, Spark will have a parallelism of only one, even if you have thousands of executors.

If we have **many partitions** but only one executor, Spark will still have a parallelism of only one because there is only one computation resource.

# 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.

Let's perform a simple transformation to find all even numbers in our current DataFrame:

```
In [7]: divisionby2 = myRange.where("number % 2 = 0")
```

There's no output , as we specified the transformation, but no action.

# Transformations



**Transformations** are the core of how you express your business logic using Spark.

There are two types of transformations:

- 1) Those that specify **narrow** dependencies.
- 2) Those that specify **wide** dependencies.

*Transformations consisting of narrow dependencies are those for which each input partition will contribute to only one output partition.*

In the preceding code snippet, the **where** statement specifies a narrow dependency, where only one partition contributes to at most one output partition



# Narrow transformations

Narrow transformations  
1 to 1



# Wide Transformations

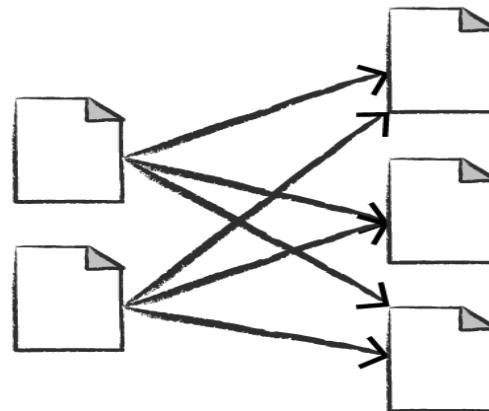


A wide dependency (or wide transformation) style transformation will have input partitions contributing to many output partitions.

We will often hear this referred to as a *shuffle* whereby Spark will exchange partitions across the cluster.

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.

Wide transformations  
(shuffles) 1 to N





# Lazy Evaluation

In Spark, *instead of modifying the data immediately when we express some operation, we build up a plan of transformations that we would like to apply to our source data.*

By waiting until the last minute to execute the code, Spark compiles this plan from our raw DataFrame transformations to a streamlined physical plan that will run as efficiently as possible across the cluster.

## Benefits of Lazy Evaluation:

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.

```
In [9]: divisionby2.count()
Out[9]: 2500
```

The output of the preceding code should be 2500. `count` is not the only action, there are many more.

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

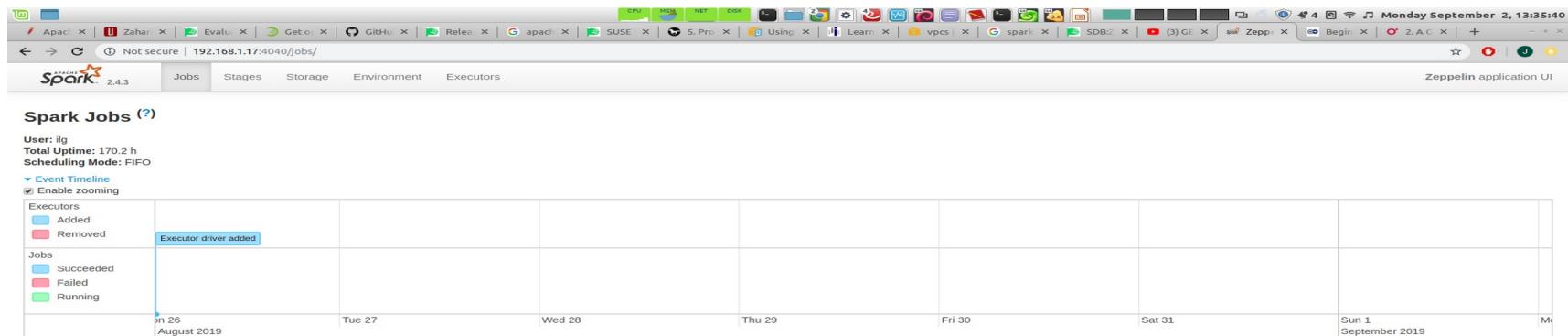


# Spark UI

We 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.



# An example



## Our data sample

```
ilg@Sparkle:~/Downloads/SparkData/data$ head flight-data/csv/2015-summary.csv
DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count
United States,Romania,15
United States,Croatia,1
United States,Ireland,344
Egypt,United States,15
United States,India,62
United States,Singapore,1
United States,Grenada,62
Costa Rica,United States,588
Senegal,United States,40
```

## An Example



Spark includes the ability to read and write from a large number of data sources.

To read this data, we will use a **DataFrameReader** that is associated with our SparkSession.

we will *specify the file format* as well as any options we want to specify.

we want to do something called ***schema inference***, which means that we want **Spark to take a best guess at what the schema of our DataFrame** should be.

We also want to specify that the first row is the header in the file.



# An Example

To get the schema information, Spark reads in a little bit of the data and then attempts to parse the types in those rows according to the types available in Spark.

We also have the option of strictly specifying a schema when you read in data

```
In [11]: flightData2015 = spark.read.option("inferSchema","true")\n....: .option("header","true")\n....: .csv("/home/ilg/Downloads/SparkData/data/flight-data/csv/2015-summary.csv")
```



Reading a CSV file into a DataFrame and converting it to a local array or list of rows

# An Example



If we perform the **take** action on the DataFrame

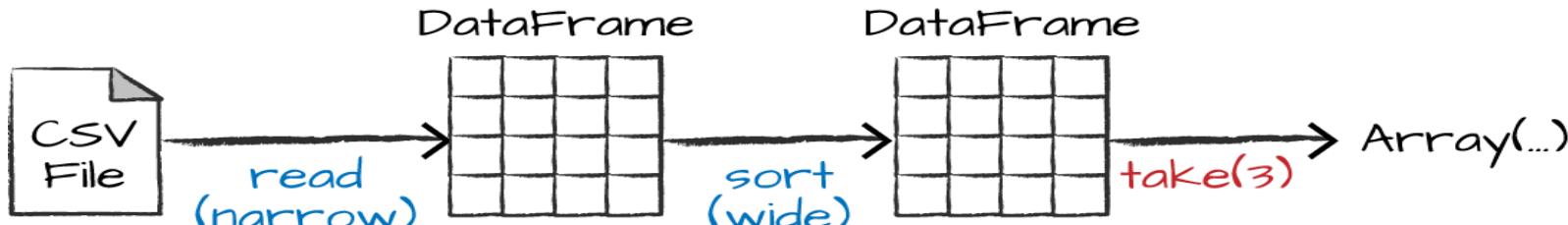
```
In [13]: flightData2015.take(10)
Out[13]:
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', count=344),
 Row(DEST_COUNTRY_NAME='Egypt', ORIGIN_COUNTRY_NAME='United States', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='India', count=62),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Singapore', count=1),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Grenada', count=62),
 Row(DEST_COUNTRY_NAME='Costa Rica', ORIGIN_COUNTRY_NAME='United States', count=588),
 Row(DEST_COUNTRY_NAME='Senegal', ORIGIN_COUNTRY_NAME='United States', count=40),
 Row(DEST_COUNTRY_NAME='Moldova', ORIGIN_COUNTRY_NAME='United States', count=1)]
```



# An Example

Now, let's sort our data according to the count column, which is an integer type.

Sort, is a transformation.



We can see that Spark is building up a plan for how it will execute this across the cluster by looking at the explain plan.

```
In [14]: flightData2015.sort("count").explain()
== Physical Plan ==
*(2) Sort [count#38 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#38 ASC NULLS FIRST, 200)
   +- *(1) FileScan csv [DEST_COUNTRY_NAME#36,ORIGIN_COUNTRY_NAME#37,count#38] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/ilg/Downloads/SparkData/data/flight-data/csv/2015-summary.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<DEST_COUNTRY_NAME:string,ORIGIN_COUNTRY_NAME:string,count:int>
```

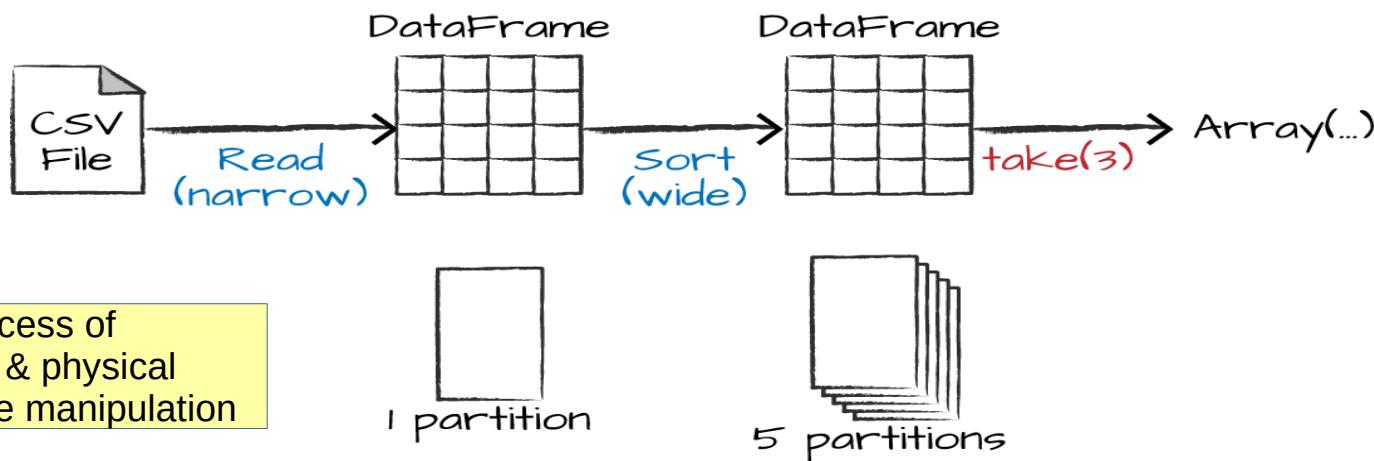


# An Example - Action

By default, when we perform a shuffle, Spark outputs 200 shuffle partitions.

Let's set this value to 5 to reduce the number of the output partitions from the shuffle:

```
In [15]: spark.conf.set("spark.sql.shuffle.partitions","5")
In [16]: flightData2015.sort("count").take(2)
Out[16]:
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Singapore', count=1),
 Row(DEST_COUNTRY_NAME='Moldova', ORIGIN_COUNTRY_NAME='United States', count=1)]
```



# DataFrames and SQL



Spark can run the same transformations, regardless of the language, in the exact same way.

We can **express your business logic in SQL or DataFrames** (either in R, Python, Scala, or Java) and Spark will compile that logic down to an underlying plan (that we can see in the explain plan) before actually executing your code.

With Spark SQL, **we can register any DataFrame as a table or view** (a temporary table) and query it using pure SQL.

There is no performance difference between writing SQL queries or writing DataFrame code, they both “compile” to the same underlying plan that we specify in DataFrame code.

# Dataframe into table or view



We can make any DataFrame into a table or view with one simple method call:

```
In [26]: flightData2015.createOrReplaceTempView("flightData2015")
```

To do so, we'll use the **spark.sql** function that conveniently returns a new DataFrame.

```
In [40]: flightData2015.createOrReplaceTempView("flight_Data_2015")

In [41]: waysql = spark.sql("""
...: select DEST_COUNTRY_NAME,count(1)
...: from flight_data_2015
...: group by DEST_COUNTRY_NAME
...:""")
```

```
In [42]: wayDataFrame = flightData2015\
...: .groupBy("DEST_COUNTRY_NAME")\
...: .count()

In [43]: waysql.explain()
== Physical Plan ==
*(2) HashAggregate(keys=[DEST_COUNTRY_NAME#36], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#36, 5)
  +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#36], functions=[partial_count(1)])
    +- *(1) FileScan csv [DEST_COUNTRY_NAME#36] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/ilg/Downloads/SparkData/data/flight-data/csv/2015-summary.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<DEST_COUNTRY_NAME:string>
```

```
In [44]: wayDataFrame.explain()
== Physical Plan ==
*(2) HashAggregate(keys=[DEST_COUNTRY_NAME#36], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#36, 5)
  +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#36], functions=[partial_count(1)])
    +- *(1) FileScan csv [DEST_COUNTRY_NAME#36] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/ilg/Downloads/SparkData/data/flight-data/csv/2015-summary.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<DEST_COUNTRY_NAME:string>
```

Same physical plan



# Interesting findings using spark



DataFrames (and SQL) in Spark already have a huge number of manipulations available.

We will use the **max** function, to establish the maximum number of flights to and from any given location.

This just scans each value in the relevant column in the DataFrame and checks whether it's greater than the previous values that have been seen.

This is a transformation, because we are effectively filtering down to one row.

```
In [45]: from pyspark.sql.functions import max
In [46]: flightData2015.select(max("count")).take(1)
Out[46]: [Row(max(count)=370002)]
```



## Let's find top 5 destinations

```
In [45]: from pyspark.sql.functions import max

In [46]: flightData2015.select(max("count")).take(1)
Out[46]: [Row(max(count)=370002)]

In [47]: SqlMax = spark.sql("""
....: select DEST_COUNTRY_NAME,sum(count) as destination_total
....: from flight_data_2015
....: group by DEST_COUNTRY_NAME
....: order by sum(count) desc
....: limit 5
....: """)

In [48]: SqlMax.show()
+-----+-----+
|DEST_COUNTRY_NAME|destination_total|
+-----+-----+
| United States | 411352 |
| Canada | 8399 |
| Mexico | 7140 |
| United Kingdom | 2025 |
| Japan | 1548 |
+-----+-----+
```



let's move to the DataFrame syntax that is semantically similar but slightly different in implementation and ordering.

We could write the data into any data source supported by Spark

e.g postgresql...

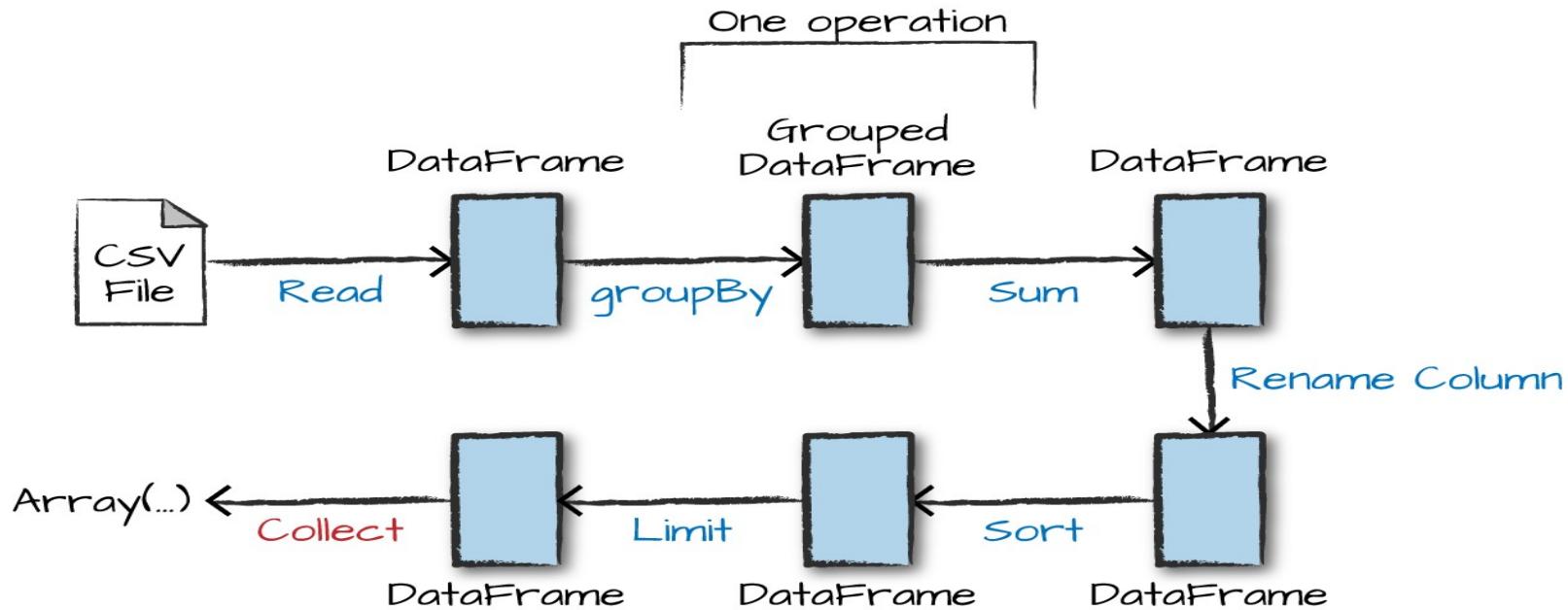
```
In [52]: from pyspark.sql.functions import desc

In [53]: flightData2015\
....: .groupBy("DEST_COUNTRY_NAME")\
....: .sum("count")\
....: .withColumnRenamed("sum(count)", "destination_total")\
....: .sort(desc("destination_total"))\
....: .limit(5)\
....: .show()
+-----+-----+
|DEST_COUNTRY_NAME|destination_total|
+-----+-----+
| United States|        411352|
|      Canada|         8399|
|     Mexico|          7140|
| United Kingdom|         2025|
|       Japan|          1548|
+-----+-----+
```



# How it got executed

This execution plan is a directed acyclic graph (DAG) of transformations, each resulting in a new immutable DataFrame, on which we call an action to generate a result.



# Explain



```
In [55]: flightData2015\  
....: .groupBy("DEST_COUNTRY_NAME")\  
....: .sum("count")\  
....: .withColumnRenamed("sum(count)", "destination_total")\  
....: .sort(desc("destination_total"))\  
....: .limit(5)\  
....: .explain()  
== Physical Plan ==  
TakeOrderedAndProject(limit=5, orderBy=[destination_total#199L DESC NULLS LAST], output=[DEST_COUNTRY_NAME#36,  
destination_total#199L])  
+- *(2) HashAggregate(keys=[DEST_COUNTRY_NAME#36], functions=[sum(cast(count#38 as bigint))])  
  +- Exchange hashpartitioning(DEST_COUNTRY_NAME#36, 5)  
    +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#36], functions=[partial_sum(cast(count#38 as bigint))])  
      +- *(1) FileScan csv [DEST_COUNTRY_NAME#36, count#38] Batched: false, Format: CSV, Location: InMemoryF  
ileIndex[file:/home/ilg/Downloads/SparkData/data/flight-data/csv/2015-summary.csv], PartitionFilters: [], Push  
edFilters: [], ReadSchema: struct<DEST_COUNTRY_NAME:string,count:int>
```

# Spark's Toolset



Structured  
Streaming

Advanced  
Analytics

Libraries &  
Ecosystem

Structured APIs

Datasets

DataFrames

SQL

Low-level APIs

RDDs

Distributed Variables

# Running Production Applications



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

spark-submit does one thing: 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.

spark-submit 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.

```
ilg@Sparkle:~$ spark-submit --master local /usr/local/spark/examples/src/main/python/pi.py 20
```

By changing the master argument of spark-submit, we can also submit the same application to a cluster running Spark's standalone cluster manager, Mesos or YARN.



Screenshot of the Apache Spark 2.4.3 application UI showing the Spark Jobs page.

The URL in the browser bar is circled in red: `192.168.1.17:4041/jobs/`.

**Spark Jobs (?)**

User: ig  
Total Uptime: 7 s  
Scheduling Mode: FIFO  
**Active Jobs: 1**

**Event Timeline**  
 Enable zooming

**Executors**  
Added (blue square)  
Removed (red square)

**Jobs**  
Succeeded (blue square)  
Failed (red square)  
Running (green square)

The timeline shows the following events:

- Executor driver added (08:15:44)
- reduce at /usr/local/spark/examples/src/main/python/pi.py:44 (Job 0) (08:15:46 - 08:15:47)

**Active Jobs (1)**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	reduce at /usr/local/spark/examples/src/main/python/pi.py:44 reduce at /usr/local/spark/examples/src/main/python/pi.py:44 (kill)	2019/09/02 08:15:46	3 s	0/1	<div style="width: 20%; background-color: blue;"></div> 2/20 (2 running)

At the bottom of the screen, there is a dock with various application icons.

# Datasets: Type-Safe Structured APIs



Dataset API is not available in Python and R, because those languages are dynamically typed.

# Structured Streaming



Structured Streaming is a high-level API for stream processing that became production-ready in Spark 2.2.

With Structured Streaming, we can take the same operations that we perform in batch mode using Spark's structured APIs and run them in a streaming fashion. *This can reduce latency and allow for incremental processing.*

*Structured Streaming is that it allows you to rapidly and quickly extract value out of streaming systems with virtually no code changes.*

## Example – Retail data



```
InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country  
536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,12/1/2010 8:26,2.55,17850,United Kingdom  
536365,71053,WHITE METAL LANTERN,6,12/1/2010 8:26,3.39,17850,United Kingdom  
536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,12/1/2010 8:26,2.75,17850,United Kingdom  
536365,84029G,KNITTED UNION FLAG HOT WATER BOTTLE,6,12/1/2010 8:26,3.39,17850,United Kingdom
```



let's first analyze the data as a static dataset and create a DataFrame to do so.  
We'll also create a schema from this static dataset

```
In [56]: staticDataFrame = spark.read.format("csv")\
.... .option("header","true")\
.... .option("inferSchema","true")\
.... .load("/home/ilg/Downloads/SparkData/data/retail-data/by-day/*.csv")  
In [57]: staticDataFrame.createOrReplaceTempView("retail_data")  
In [58]: staticSchema = staticDataFrame.schema
```

In this example we'll take a look at the *sale hours during which a given customer (identified by CustomerId) makes a large purchase.*

Consider e.g

Adding a total cost column and see on what days a customer spent the most.



# Window function

**Window function** will include all data from each day in the aggregation.

A window over the time-series column in our data.

This is a helpful tool for manipulating date and timestamps because we can specify our requirements in a more human form (via intervals), and Spark will group all of them together for us:

```
In [61]: from pyspark.sql.functions import window, column, desc, col
```

```
In [63]: staticDataFrame\
....: .selectExpr(
....: "CustomerId", "(UnitPrice * Quantity) as total_cost", "InvoiceDate")\
....: .groupBy(
....: col("CustomerId"), window(col("InvoiceDate"), "1 day"))\
....: .sum("total_cost")\
....: .show(5)
+-----+-----+-----+
|CustomerId|      window|  sum(total_cost)|
+-----+-----+-----+
| 14075.0|[2011-12-04 19:00...| 316.78000000000003|
| 18180.0|[2011-12-04 19:00...|       310.73|
| 15358.0|[2011-12-04 19:00...| 830.0600000000003|
| 15392.0|[2011-12-04 19:00...|304.4099999999997|
| 15290.0|[2011-12-04 19:00...| 263.0200000000004|
+-----+-----+-----+
only showing top 5 rows
```



# Streaming code

```
streamingDataFrame = spark.readStream\  
    .schema(staticSchema)\  
    .option("maxFilesPerTrigger", 1)\  
    .format("csv")  
    .option("header", "true")\  
    .load("/home/ilg/Downloads/SparkData/data/retail-data/by-day/*.csv")  
  
#Confirm, if it's streaming  
streamingDataFrame.isStreaming // returns true
```



We'll perform a summation in the process:

```
purchaseByCustomerPerHour = streamingDataFrame\  
    .selectExpr(  
        "CustomerId",  
        "(UnitPrice * Quantity) as total_cost",  
        "InvoiceDate")\  
    .groupBy(  
        col("CustomerId"), window(col("InvoiceDate"), "1 day"))\  
    .sum("total_cost")
```



we will need to call a streaming action to start the execution of this data flow.

The action we will use will output to an in-memory table that we will update after each trigger.

In this case, each trigger is based on an individual file (the read option that we set). *Spark will mutate the data in the in-memory table* such that we will always have the highest value as specified in our previous aggregation:

```
purchaseByCustomerPerHour.writeStream\  
    .format("memory")\  
    .queryName("customer_purchases")\  
    .outputMode("complete")\  
    .start()
```

```
purchaseByCustomerPerHour.writeStream  
    .format("console")  
    .queryName("customer_purchases_2")  
    .outputMode("complete")  
    .start()
```

Results on  
console

# Streaming code - query



When we start the stream, we can run queries against it to debug what our result will look like if we were to write this out to a production sink:

```
spark.sql("""  
    SELECT * FROM customer_purchases ORDER BY `sum(total_cost)` DESC """)\  
.show(5)
```

```
spark.sql(""" SELECT * FROM customer_purchases ORDER BY `sum(total_cost)` DESC """)  
.show(15)
```

**NOTE: We shouldn't use either of these streaming methods in production**

# Machine Learning and Advanced Analytics



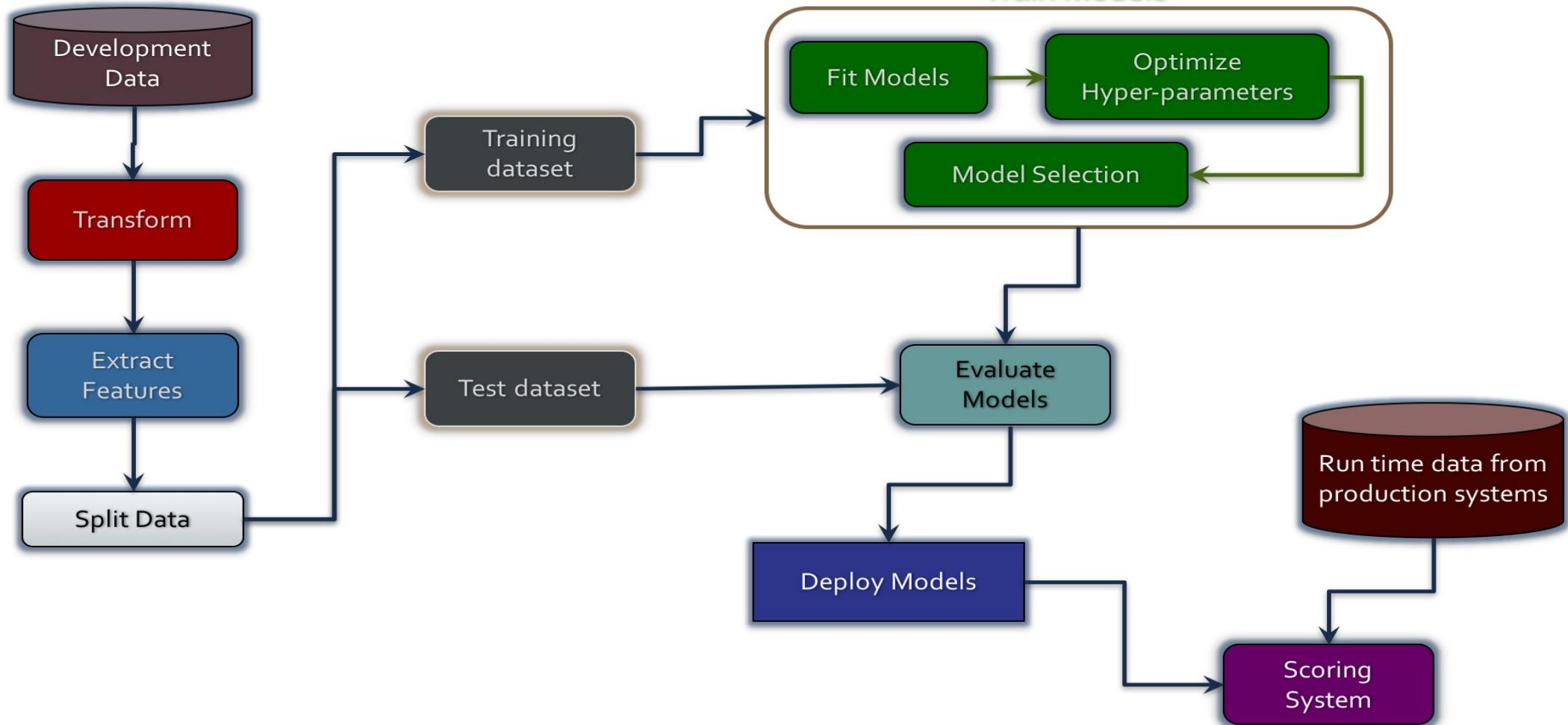
Spark has its ability to perform large-scale machine learning with a built-in library of machine learning algorithms called **Mllib**.

Mlib allows for **preprocessing**, **munging**, **training of models**, and **making predictions** at scale on data.

We can even use models trained in Mlib to make predictions in Structured Streaming.

Spark provides a sophisticated machine learning API for performing a variety of machine learning tasks, from classification to regression, and clustering to deep learning.

## Train Models



# K-means demo



k-means is a clustering algorithm in which “K” centers are randomly assigned within the data.

The points closest to that point are then “assigned” to a class and the center of the assigned points is computed. **This center point is called the centroid.**

We then label the points closest to that centroid, to the centroid’s class, and shift the centroid to the new center of that cluster of points.

We repeat this process for a finite set of iterations or until convergence (our center points stop changing).

Used business case:

Behavioral segmentation like purchase history.

Inventory categorization like grouping stock/inventory by sales.

Sorting sensor measurements like separate diff audio.

Detecting bots or anomalies like grouping valid and invalid activities.



# Data

```
In [2]: staticDataFrame = spark.read.format("csv")\
....: .option("header", "true")\
....: .option("inferSchema", "true")\
....: .load("/home/ilg/Downloads/SparkData/data/retail-data/by-day/*.csv")
....: staticDataFrame.createOrReplaceTempView("retail_data")
....: staticSchema = staticDataFrame.schema
....:

In [3]: staticDataFrame.printSchema()
root
|-- InvoiceNo: string (nullable = true)
|-- StockCode: string (nullable = true)
|-- Description: string (nullable = true)
|-- Quantity: integer (nullable = true)
|-- InvoiceDate: timestamp (nullable = true)
|-- UnitPrice: double (nullable = true)
|-- CustomerID: double (nullable = true)
|-- Country: string (nullable = true)
```



# Data presentation in what format ?

Machine learning algorithms in MLlib require that data is represented as **numerical values.**

Our current data is represented by a variety of different types, including timestamps, integers, and strings.

Therefore we need to transform this data into some numerical representation

```
In [4]: from pyspark.sql.functions import date_format,col  
  
In [5]: prepDF = staticDataFrame\  
....: .na.fill(0)\  
....: .withColumn("day_of_week",date_format(col("InvoiceDate"),"EEEE"))\  
....: .coalesce(5)  
  
In [6]: trainDF = prepDF\  
....: .where("InvoiceDate < '2011-07-01'")  
....: .cache()  
  
In [8]: testDF = prepDF\  
....: .where("InvoiceDate >= '2011-07-01'")
```

<https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html>



# Split data

We are also going to need to split the data into training and test sets.

In this instance, we are going to do this manually by the date on which a certain purchase occurred;

Note - we could also use MLlib's transformation APIs to create a training and test set via train validation splits or cross validation.

```
trainDF = preppedDataFrame\  
    .where("InvoiceDate < '2011-07-01'")  
  
testDF = preppedDataFrame\  
    .where("InvoiceDate >= '2011-07-01'")
```

Our Data is prepared now. let's split it into a training and test set.

Because this is a time-series set of data, ***we will split by an arbitrary date in the dataset.***

The data is split into roughly in half.

```
In [9]: trainDF.count()  
Out[9]: 245903  
  
In [10]: testDF.count()  
Out[10]: 296006
```

**Note- These transformations are DataFrame transformations**



# Sparks transformations

Spark has many transformations We will look at *StringIndexer*

```
In [34]: from pyspark.ml.feature import StringIndexer  
  
In [35]: indexer = StringIndexer()\n    ...:     .setInputCol("day_of_week")\n    ...:     .setOutputCol("day_of_week_index")
```

This will turn our days of weeks into corresponding numerical values.

In this e.g -

Spark might represent **Saturday as 6**, and **Monday as 1**. However, with this numbering scheme, we are implicitly stating that Saturday is greater than Monday (by pure numerical values). This is obviously incorrect.

**To fix this**, we therefore need to use a **OneHotEncoder** to encode each of these values as their own column. These Boolean flags state whether that day of week is the relevant day of the week

```
In [37]: encoder = OneHotEncoder()\n    ...:     .setInputCol("day_of_week_index")\n    ...:     .setOutputCol("day_of_week_encoded")
```



Each of these will result in a set of columns that we will “**assemble**” into a vector.

**Note:** All machine learning algorithms in Spark take as **input a Vector type**, which must be a set of numerical values:

```
In [38]: from pyspark.ml.feature import VectorAssembler  
  
In [39]: vectorAssembler = VectorAssembler()\n....:     .setInputCols(["UnitPrice", "Quantity", "day_of_week_encoded"])\n....:     .setOutputCol("features")
```

# pipeline



After the previous-

we'll set this up into a pipeline so that any future data we need to transform can go through the exact same process:

```
In [40]: from pyspark.ml import Pipeline  
  
In [41]: transformationPipeline = Pipeline()\\"  
.... .setStages([indexer, encoder, vectorAssembler])
```

In machine learning, it is common to run a sequence of algorithms to process and learn from data.

E.g. A simple text document processing workflow might include several stages:

- Split each document's text into words.
- Convert each document's words into a numerical feature vector.
- Learn a prediction model using the feature vectors and labels.

# Fitting our data



Preparing for training is a two-step process.

We need to fit our transformers to this dataset.

After we fit the training data.

we are ready to take that fitted pipeline and use it to transform all of our data in a consistent and repeatable way:

```
In [43]: fittedPipeline = transformationPipeline.fit(trainDF)  
In [44]: transformedTraining = fittedPipeline.transform(trainDF)
```

# Time to train the model



We now have a training set; it's time to train the model.

First we'll import the relevant model that we'd like to use and instantiate it:

```
In [32]: kmeans = KMeans().setK(20).setSeed(1)
```

```
In [34]: kmModel.computeCost(transformationTraining)
Out[34]: 84553739.96537484
```

```
In [35]: transformedTest = fittedPipeline.transform(testDF)
```

```
In [36]: kmModel.computeCost(transformedTest)
Out[36]: 517507094.7222117
```

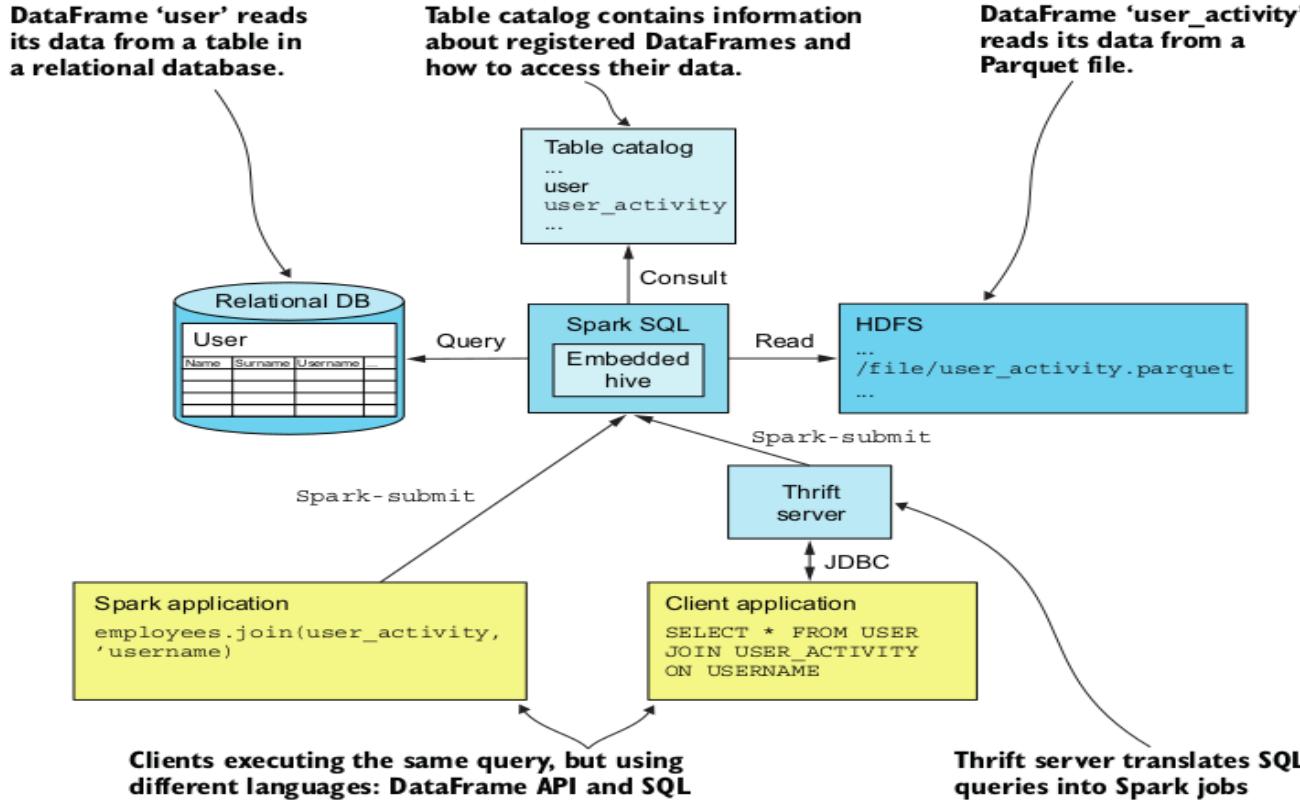


## Structured APIs—DataFrames, SQL, and Datasets

# DataFrames



DataFrames let's us work with structured data organized in rows and columns, where each column contains only values of a certain type.





# How Spark views DataFrame & DataSets

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.

# Schemas



A schema defines the column names and types of a DataFrame.

We can define schemas manually or read a schema from a data source (often called schema on read).

Schemas consist of types, meaning that you need a way of specifying what lies where.

# Structured Spark Types



Spark is effectively a programming language of its own.

Spark uses an engine called **Catalyst** that maintains its own type information through the planning and processing of work.

This provides wide variety of execution optimizations.

# DataFrames Versus Datasets



Dataframe's are “**untyped**”.

Spark maintains it's type internally.

Dataset's are “**typed**”

These are only available JVM based languages.

**For Spark in python or R – everything is DataFrames.**

**More optimized for performance, than natively available for python or R**

# Columns



Imagine Spark columns as columns in a table.

Represents

Simple type - integer or string

Complex type – array, map, null value.

# Rows



A row is record of data.

Each record in a Dataframe must be of type Row.

```
In [48]: spark.range(10).collect()
Out[48]:
[Row(id=0),
 Row(id=1),
 Row(id=2),
 Row(id=3),
 Row(id=4),
 Row(id=5),
 Row(id=6),
 Row(id=7),
 Row(id=8),
 Row(id=9)]
```

# Spark Types



Spark has a large number of internal type representations.

Using python types:

```
from pyspark.sql.types import *
b = ByteType()
```



# Python type reference

Data type	Value type in Python	API to access or create a data type
ByteType	int or long. Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Ensure that numbers are within the range of -128 to 127.	ByteType()
ShortType	int or long. Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Ensure that numbers are within the range of -32768 to 32767.	ShortType()
IntegerType	int or long. Note: Python has a lenient definition of "integer." Numbers that are too large will be rejected by Spark SQL if you use the IntegerType(). It's best practice to use LongType.	IntegerType()
LongType	long. Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Ensure that numbers are within the range of -9223372036854775808 to 9223372036854775807. Otherwise, convert data to decimal.Decimal and use DecimalType.	LongType()
FloatType	float. Note: Numbers will be converted to 4-byte single-precision floating-point numbers at runtime.	FloatType()
DoubleType	float	DoubleType()
DecimalType	decimal.Decimal	DecimalType()
StringType	string	StringType()
BinaryType	bytearray	BinaryType()
BooleanType	bool	BooleanType()
TimeStampType	datetime.datetime	TimeStampType()
DateType	datetime.date	DateType()

# Python type reference

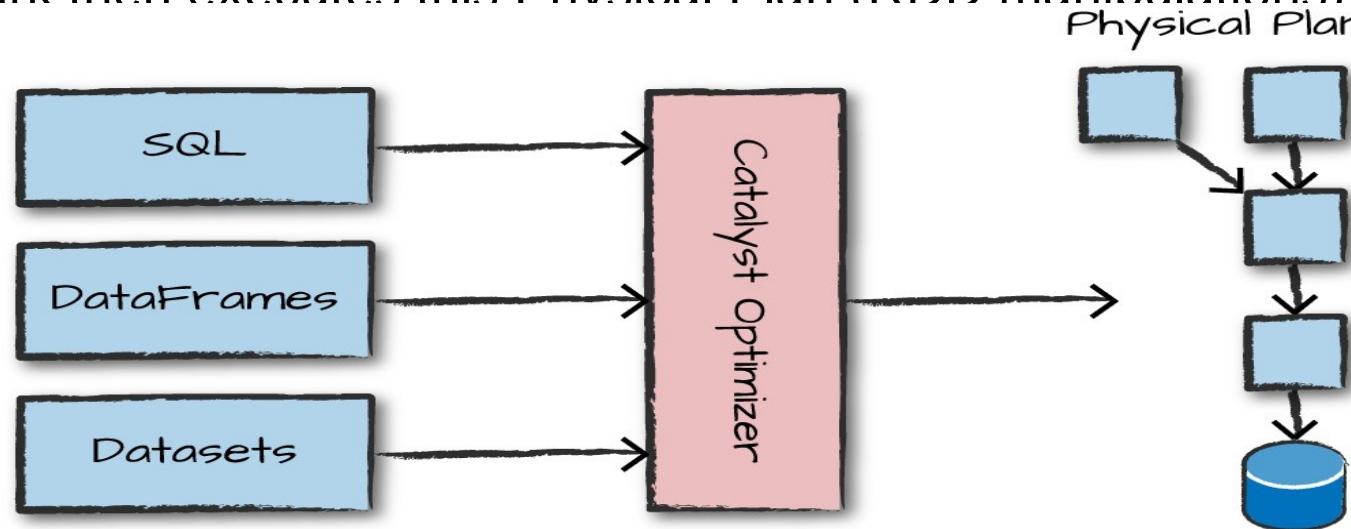


ArrayType	list, tuple, or array	ArrayType(elementType, [containsNull]). Note: The default value of containsNull is True.
MapType	dict	MapType(keyType, valueType, [valueContainsNull]). Note: The default value of valueContainsNull is True.
StructType	list or tuple	StructType(fields). Note: fields is a list of StructFields. Also, fields with the same name are not allowed.
StructField	The value type in Python of the data type of this field (for example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, [nullable]) Note: The default value of nullable is True.

# Structured API Execution



1. Write DataFrame/Dataset/SQL Code.
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.

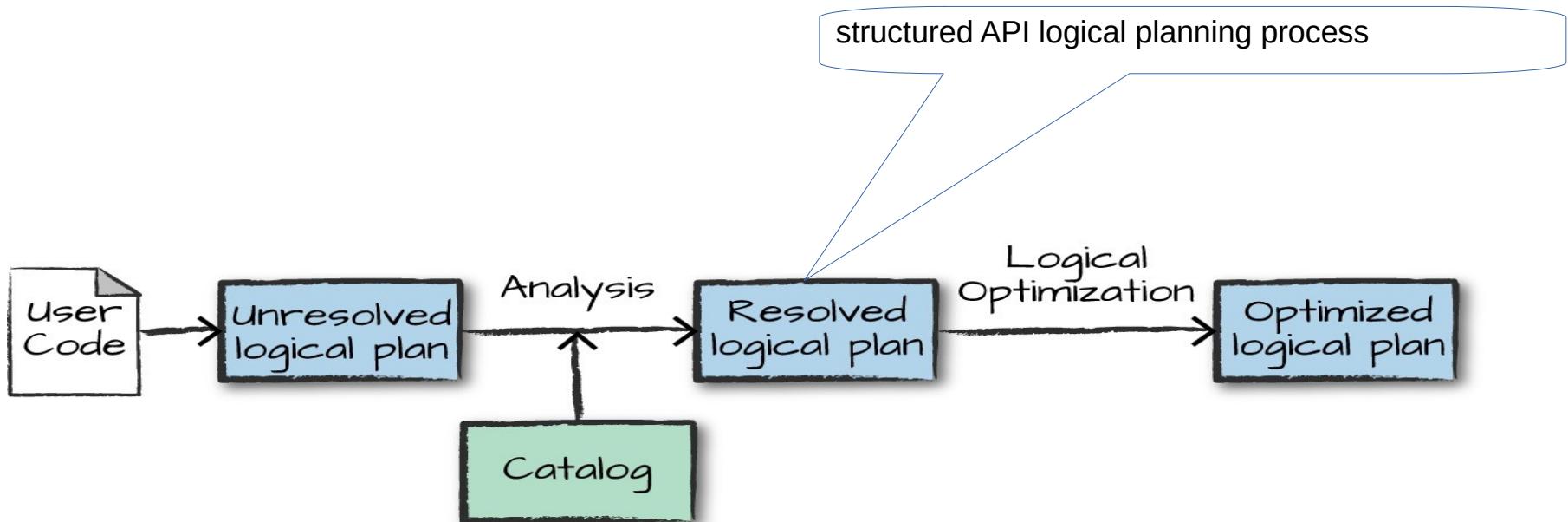


# Logical Planning



1<sup>st</sup> Phase

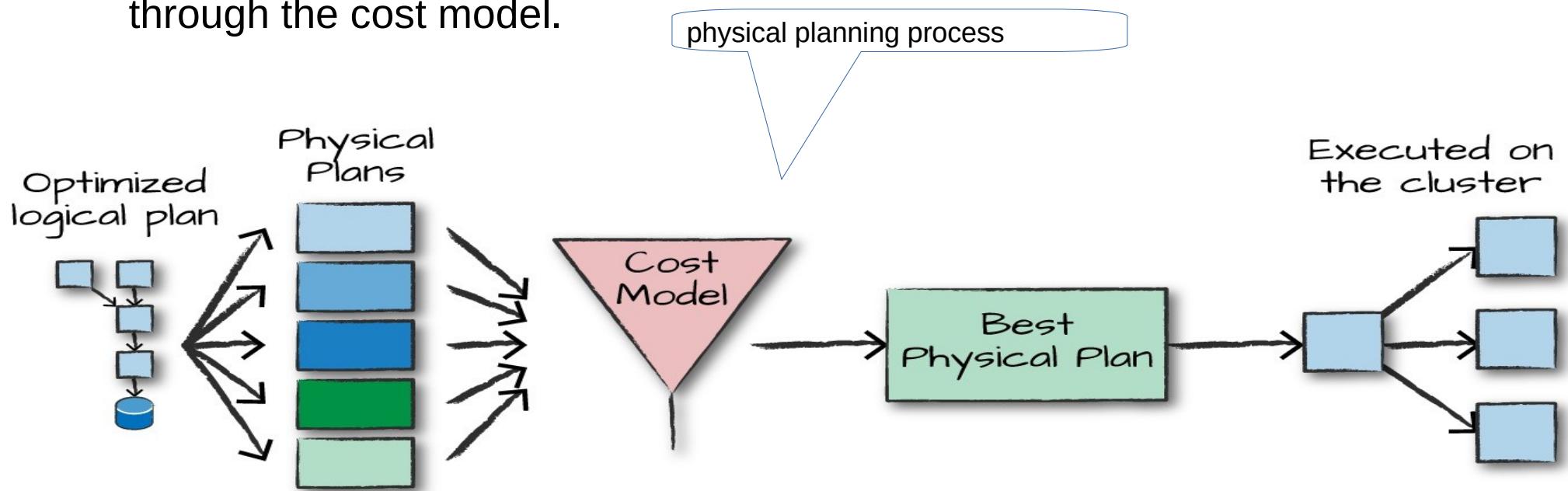
User code converted to Logical Plan





# Physical plan

Spark physical specifies how the logical plan will execute on the cluster by *generating different physical execution strategies* and comparing them through the cost model.



# Result of Physical plan



Physical planning results in a **series of RDDs and transformations**.

This result in Spark referred to as a compiler—it takes queries in `DataFrames`, `Datasets`, and `SQL` and compiles them into `RDD` transformations for us.

# Execution



After physical plan is selected, Spark runs all the code over RDD's.

Further optimizations are carried out during runtime, generates native java bytecode, there by delivering the results.



## Basic Structured Operations



Learning about:

Fundamental DataFrame Operations

# Create DataFrame



Let's create a DataFrame with which we can work:

```
In [52]: df = spark.read.format("json").load("/home/ilg/Documents/sparkdata/data/flight  
...: -data/json/2015-summary.json")  
  
In [53]: df.printSchema()  
root  
| -- DEST_COUNTRY_NAME: string (nullable = true)  
| -- ORIGIN_COUNTRY_NAME: string (nullable = true)  
| -- count: long (nullable = true)
```

# 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.

```
In [53]: df.printSchema()
root
| -- DEST_COUNTRY_NAME: string (nullable = true)
| -- ORIGIN_COUNTRY_NAME: string (nullable = true)
| -- count: long (nullable = true)
```

Deciding whether you need to define a schema prior to reading in your data depends on your use case.  
- Yes / No

# Spark determines schema



Reading the json file, what pyspark returns.

```
In [56]: spark.read.format("json").load("/home/ilg/Documents/sparkdata/data/flight-  
...: data/json/2015-summary.json").schema  
Out[56]: StructType(List(StructField(DEST_COUNTRY_NAME,StringType,true),StructField  
(ORIGIN_COUNTRY_NAME,StringType,true),StructField(count,LongType,true)))
```



# Enforce the schema

Let's see how to create and enforce a specific schema on a DataFrame.

*If the types in the data (at runtime) do not match the schema, Spark will throw an error.*

```
In [59]: from pyspark.sql.types import StructField, StructType, StringType, LongType
In [60]: ManualSchema = StructType([StructField("DEST_COUNTRY_NAME", StringType(), True), \
....: StructField("ORIGIN_COUNTRY_NAME", StringType(), True), \
....: StructField("count", LongType(), False, metadata={"Hello": "world"})])
In [61]: df = spark.read.format("json").schema(ManualSchema) \
....: .load("/home/ilg/Documents/sparkdata/data/flight-data/json/2015-summary.json")
```

# Columns and Expressions



To Spark, columns are logical constructions that simply represent a value computed on a per-record basis by means of an expression.

## Columns

Two simplest ways are by using the `col` or `column` functions.

To use either of these functions, you pass in a column name:

```
In [62]: from pyspark.sql.functions import col, column
In [63]: col("me")
Out[63]: Column<b'me'>
In [64]: column("you")
Out[64]: Column<b'you'>
```

# Explicit column references



If you need to refer to a specific DataFrame's column, we can use the **col** method on the specific DataFrame.

## Useful:

when you are performing a join and need to refer to a specific column in one DataFrame that might share a name with another column in the joined DataFrame.

```
df.col("count")
```

# Expressions



An expression is a set of transformations on one or more values in a record in a DataFrame.

Imagine it to be like a function that takes as input one or more column names, resolves them, and then potentially applies more expressions to create a single value for each record in the dataset.

**Note:** Importantly, this “single value” can actually be a complex type like a Map or Array.

An expression, created via the `expr` function, is just a DataFrame column reference.

**`expr("someCol")` is equivalent to `col("someCol")`.**

# Columns as expressions



Columns provide a subset of expression functionality.

If you use `col()` and want to perform *transformations on that column*, we must perform those on that **column reference**.

When using an expression, the `expr` function can actually parse transformations and column references from a string and can subsequently be passed into further transformations.

`expr("someCol - 5")`

is the same transformation as performing

`col("someCol") - 5,`

OR

`expr("someCol") - 5.`

**NOTE:**

1) Columns are just expressions.

2) Columns and transformations of those columns compile to the same logical expressions.

plan as parsed

E.g `((col("someCol") + 5) * 200) - 6) < col("otherCol")`



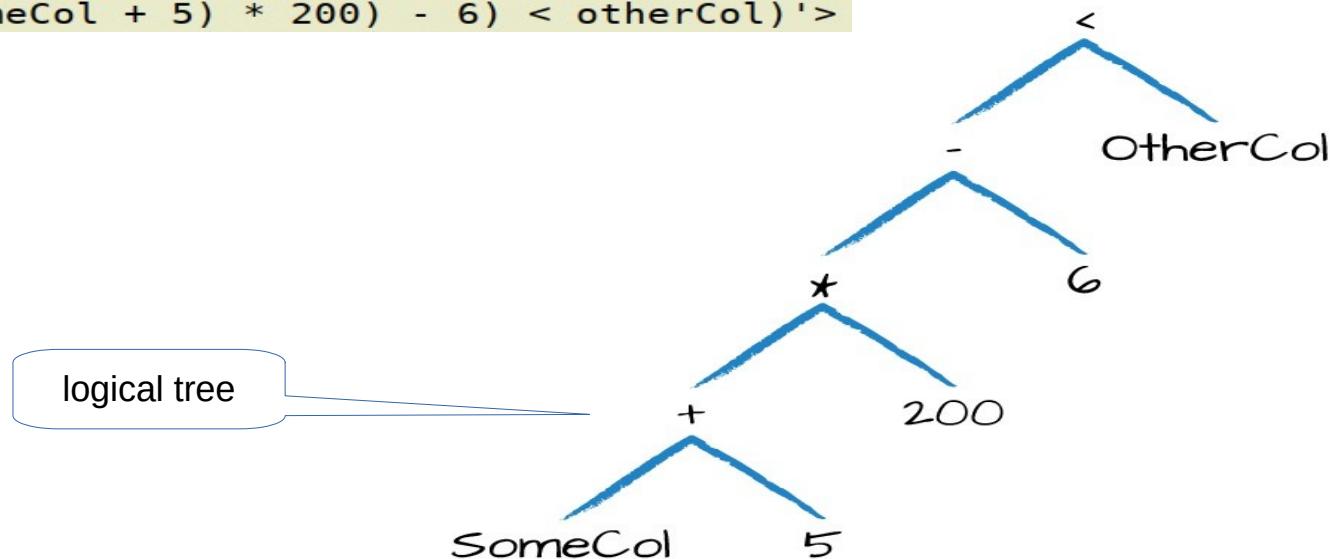
# A DAG

it's a directed acyclic graph

```
In [79]: from pyspark.sql.functions import expr
```

```
In [80]: expr("(((someCol + 5) * 200)-6) < otherCol")
```

```
Out[80]: Column'<b'('(((someCol + 5) * 200) - 6) < otherCol)'>
```



# Accessing a DataFrame's columns



if we want to programmatically access columns, we can use the columns property to see all columns on a DataFrame:

```
In [81]: spark.read.format("json").load("/home/ilg/Documents/sparkdata/data/flight-  
...: data/json/2015-summary.json").columns  
Out[81]: ['DEST_COUNTRY_NAME', 'ORIGIN_COUNTRY_NAME', 'count']
```

To see a DataFrame's columns, which we can do by using something like **printSchema**

```
In [82]: spark.read.format("json").load("/home/ilg/Documents/sparkdata/data/flight-  
...: data/json/2015-summary.json").printSchema  
Out[82]: <bound method DataFrame.printSchema of DataFrame[DEST_COUNTRY_NAME: string  
, ORIGIN_COUNTRY_NAME: string, count: bigint]>
```



# Records and Rows

In Spark, each row in a DataFrame is a single record.

Spark represents this record as an object of type Row.

Spark manipulates **Row objects** using column expressions in order to produce usable values.

Row objects internally represent arrays of bytes.

The byte array interface is never shown to users because we only use column expressions to manipulate them.

```
In [84]: df.first()
Out[84]: Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15)
```

# Creating Rows



We can create rows by manually instantiating a Row object with the values that belong in each column.

***Only DataFrames have schemas.***

Rows themselves ***do not*** have schemas.

*This means that if we create a Row manually, we must specify the values in the same order as the schema of the DataFrame to which they might be appended*

```
In [86]: from pyspark.sql import Row  
In [87]: mRow = Row("Hey",None,2,False)
```

# Accessing Row



```
In [86]: from pyspark.sql import Row  
  
In [87]: mRow = Row("Hey",None,2,False)  
  
In [88]: mRow[0]  
Out[88]: 'Hey'  
  
In [89]: mRow[1]  
  
In [90]: mRow[2]  
Out[90]: 2  
  
In [91]: mRow[3]  
Out[91]: False
```

# DataFrame Transformations



Manipulating DataFrames -

- *We can add rows or columns*
- *We can remove rows or columns*
- *We can transform a row into a column (or vice versa)*
- *We can change the order of rows based on the values in columns*

# Creating DataFrames



We can create DataFrames from raw data sources. Like...

```
In [98]: mdf = spark.read.format("json").load("/home/ilg/Documents/sparkdata/data/f  
.... light-data/json/2015-summary.json")  
  
In [99]: df.createOrReplaceTempView("dfTable")
```

# Creating DataFrames



We can also create DataFrames on the fly by taking a set of rows and converting them to a DataFrame.

```
In [5]: from pyspark.sql import Row  
  
In [6]: from pyspark.sql.types import StructField,StructType,StringType,LongType  
  
In [7]: myManualSchema = StructType([  
....:     StructField("some", StringType(), True),  
....:     StructField("col", StringType(), True),  
....:     StructField("names", LongType(), False)  
....: ])  
  
In [8]: myRow = Row("Hello", None, 1)  
  
In [9]: myDf = spark.createDataFrame([myRow], myManualSchema)  
  
In [10]: myDf.show()  
+---+---+---+  
| some| col|names|  
+---+---+---+  
|Hello|null|    1|  
+---+---+---+
```



# select and selectExpr

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

```
In [7]: from pyspark.sql.functions import expr, col, column  
  
In [8]: df.select(  
...:     expr("DEST_COUNTRY_NAME"),  
...:     col("DEST_COUNTRY_NAME"),  
...:     column("DEST_COUNTRY_NAME"))\n...: .show(2)  
+-----+-----+-----+  
|DEST_COUNTRY_NAME|DEST_COUNTRY_NAME|DEST_COUNTRY_NAME|  
+-----+-----+-----+  
| United States| United States| United States|  
| United States| United States| United States|  
+-----+-----+-----+  
only showing top 2 rows  
  
In [9]: df.select("DEST_COUNTRY_NAME").show(2)  
+-----+  
|DEST_COUNTRY_NAME|  
+-----+  
| United States|  
| United States|  
+-----+  
only showing top 2 rows
```



```
In [10]: df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)
+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|
+-----+-----+
|  United States|          Romania|
|  United States|          Croatia|
+-----+-----+
only showing top 2 rows
```

```
In [11]: df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME", "count").show(2)
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
|  United States|          Romania|    15|
|  United States|          Croatia|     1|
+-----+-----+-----+
only showing top 2 rows
```



```
In [15]: from pyspark.sql.functions import expr,col,column

In [16]: df.select(
....:     expr("DEST_COUNTRY_NAME"),
....:     col("DEST_COUNTRY_NAME"),
....:     column("DEST_COUNTRY_NAME"))\
....: .show(5)
+-----+-----+-----+
|DEST_COUNTRY_NAME|DEST_COUNTRY_NAME|DEST_COUNTRY_NAME|
+-----+-----+-----+
| United States | United States | United States |
| United States | United States | United States |
| United States | United States | United States |
|           Egypt |           Egypt |           Egypt |
| United States | United States | United States |
+-----+-----+-----+
only showing top 5 rows
```



```
In [31]: df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)
+-----+
| destination|
+-----+
|United States|
|United States|
+-----+
only showing top 2 rows
```

```
In [32]: df.select(expr("DEST_COUNTRY_NAME as destination").alias("DEST_COUNTRY_NAME"))\n    .show(2)
+-----+
|DEST_COUNTRY_NAME|
+-----+
| United States|
| United States|
+-----+
only showing top 2 rows
```

```
In [33]: df.selectExpr("DEST_COUNTRY_NAME as newColumnName", "DEST_COUNTRY_NAME").show(2)
+-----+-----+
|newColumnName|DEST_COUNTRY_NAME|
+-----+-----+
|United States| United States|
|United States| United States|
+-----+-----+
only showing top 2 rows
```



Here's a simple example that adds a new column **withinCountry** to our DataFrame that specifies *whether the destination and origin are the same*:

```
In [45]: df.selectExpr(  
....:     "*", # all original columns  
....:     "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry") \  
....:     .show(5)  
+-----+-----+-----+-----+  
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|  
+-----+-----+-----+-----+  
| United States|          Romania|    15|      false|  
| United States|          Croatia|     1|      false|  
| United States|          Ireland|   344|      false|  
|        Egypt|      United States|    15|      false|  
| United States|          India|    62|      false|  
+-----+-----+-----+-----+  
only showing top 5 rows
```



With **select expression**, we can also specify **aggregations** over the entire DataFrame by taking advantage of the functions that we have

```
In [47]: df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))").show(4)
+-----+-----+
| avg(count)|count(DISTINCT DEST_COUNTRY_NAME)|
+-----+-----+
| 1770.765625|          132|
+-----+-----+
```

# Converting to Spark Types (Literals)



Using 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.*

Literals are expressions. *Creates a Column of literal value.*

```
In [51]: df.select(expr("*"),lit(1).alias("One")).show(2)
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|One|
+-----+-----+-----+
|      United States|                  Romania|    15|   1|
|      United States|                  Croatia|     1|   1|
+-----+-----+-----+
only showing top 2 rows
```

This will come up when you might need to check whether a value is greater than some constant or other programmatically created variable.

# Adding Columns



Add columns with `withColumn`

let's add a column that just adds the number one as a column:

```
In [52]: df.withColumn("numberOne", lit(1)).show(3)
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|numberOne|
+-----+-----+-----+
| United States| Romania| 15| 1|
| United States| Croatia| 1| 1|
| United States| Ireland| 344| 1|
+-----+-----+-----+
only showing top 3 rows
```



Let's try another example, we'll set a Boolean flag for when the origin country is the same as the destination country:

```
In [53]: df.withColumn("withinCountry",expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME"))\n      .... .show(3)\n+-----+-----+-----+\n|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|\n+-----+-----+-----+\n| United States| Romania|    15|     false|\n| United States| Croatia|     1|     false|\n| United States| Ireland|  344|     false|\n+-----+-----+-----+\nonly showing top 3 rows
```

Notice that the **withColumn** function takes two arguments:

**column name & expression**

that will create the value for that given row in the DataFrame.

# Rename Columns



```
In [54]: df.withColumn("Destination", expr("DEST_COUNTRY_NAME")).columns  
Out[54]: ['DEST_COUNTRY_NAME', 'ORIGIN_COUNTRY_NAME', 'count', 'Destination']
```

```
In [55]: df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns  
Out[55]: ['dest', 'ORIGIN_COUNTRY_NAME', 'count']
```



# Reserved Characters and Keywords

One thing that we might come across is reserved characters like **spaces** or **dashes** in column names.

We handle these by using **backticks**

```
In [56]: dfWithLongColName = df.withColumn(  
....:     "This Long Column-Name",  
....:     expr("ORIGIN_COUNTRY_NAME"))  
  
In [57]: dfWithLongColName.selectExpr(  
....:     "`This Long Column-Name`",  
....:     "`This Long Column-Name` as `new col`")\n....: .show(2)  
+-----+-----+  
|This Long Column-Name|new col|  
+-----+-----+  
|                  Romania|Romania|  
|                  Croatia|Croatia|  
+-----+-----+  
only showing top 2 rows
```

We need to use backticks because we're referencing a column in an expression



We only need to escape expressions that use reserved characters or keywords.

```
dfWithLongColName.select(expr(`This Long Column-Name`)).columns
```

# Case Sensitivity



**By default Spark is case insensitive;**

But, we can make Spark case sensitive by setting the configuration:

```
-- in SQL  
set spark.sql.caseSensitive true
```



# Removing Columns

Let's take a look at how we can remove columns from DataFrames.

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

```
In [72]: df.drop("ORIGIN_COUNTRY_NAME").columns  
Out[72]: ['DEST_COUNTRY_NAME', 'count']
```



```
In [73]: dfWithLongColName.drop("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME")  
Out[73]: DataFrame[count: bigint, This Long Column-Name: string]
```



# Changing a Column's Type (cast)

We might need to convert from one type to another. if we have a set of **StringType** that should be **integers**.

We can convert columns from one type to another by casting the column from one type to another.

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



# Filtering Rows

```
In [97]: df.where("count < 2").show(2)
+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+
|      United States|                  Croatia|    1|
|      United States|                  Singapore|    1|
+-----+-----+
only showing top 2 rows
```

```
In [98]: df.where("count > 2").show(2)
+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+
|      United States|                  Romania|   15|
|      United States|                  Ireland| 344|
+-----+-----+
only showing top 2 rows
```



```
In [100]: df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") != "Croatia")\n      ....\n      .show(5)\n+-----+-----+-----+\n| DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|\n+-----+-----+-----+\n| United States|          Singapore|    1|\n|      Moldova|          United States|    1|\n|       Malta|          United States|    1|\n| United States|          Gibraltar|    1|\n| Saint Vincent and...|          United States|    1|\n+-----+-----+-----+\nonly showing top 5 rows
```

# Getting Unique Rows



A very common use case is to extract the unique or distinct values in a DataFrame.

Using the **distinct** method on a DataFrame, which allows us to deduplicate any rows that are in that DataFrame.

```
In [104]: df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").distinct().count()
Out[104]: 256

In [105]: df.select("ORIGIN_COUNTRY_NAME").distinct().count()
Out[105]: 125

In [106]: df.select("DEST_COUNTRY_NAME").distinct().count()
Out[106]: 132
```



# Random Samples

Sometimes, we might just want to sample some random records from your DataFrame.

We can do this by using the **sample** method on a DataFrame, which makes it possible for you to specify a fraction of rows to extract from a DataFrame and whether you'd like to sample with or without replacement.

Check results / count  
withReplacement = True ?

```
In [107]: seed = 5
In [108]: withReplacment = False
In [109]: fraction = 0.5
In [110]: df.sample(withReplacment,fraction,seed).count()
Out[110]: 126

In [111]: df.sample(withReplacment,fraction,seed).show()
+-----+-----+-----+
| DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
| United States|Romania|15
| Egypt|United States|15
| United States|India|62
| jasmit|United States|588
| Costa Rica|United States|40
| Senegal|United States|64
| Guyana|United States|30
| Bolivia|United States|6
| United States|Paraguay|4
| Algeria|United States|230
| Turks and Caicos ...|United States|1
| Saint Vincent and...|United States|1
| purva|United States|Russia|161
| United States|Pakistan|United States|12
| United States|Marshall Islands|United States|42
| Honduras|United States|362
| United States|Foxit Read|United States|Senegal|42
| Hong Kong|United States|332
| United States|Trinidad and Tobago|United States|211
| United States|United States|Cyprus|1
| Untitled|United States|Portugal|134
+-----+-----+-----+
only showing top 20 rows
```



# Random Splits

Randomly splits this DataFrame with the provided weights.

This is often used with machine learning algorithms to create training, validation, and test sets.

```
In [122]: mDf = df.randomSplit([0.25,0.75],seed)
In [123]: mDf[0].count()
Out[123]: 60
In [124]: mDf[1].count()
Out[124]: 196
In [125]: mDf[2].count()

In [126]: mDf[0].count() > mDf[1].count()
Out[126]: False
```

# Concatenating and Appending Rows (Union)



DataFrames are immutable.

This means users cannot append to DataFrames because that would be changing it.

To append to a DataFrame, you must *union* the original DataFrame along with the new DataFrame. This just concatenates the two DataFrames.

*To union two DataFrames, you must be sure that they have the same schema and number of columns; otherwise, the union will fail.*



```
from pyspark.sql import Row  
  
schema = df.schema  
  
newRows = [  
    Row("New Country", "Other Country", 5L),  
    Row("New Country 2", "Other Country 3", 1L)  
]  
parallelizedRows = spark.sparkContext.parallelize(newRows)  
newDF = spark.createDataFrame(parallelizedRows, schema)  
df.union(newDF)\br/>.where("count = 1")\br/>.where(col("ORIGIN_COUNTRY_NAME") != "United States")\br/>.show()
```

## Sorting Rows



When we sort the values in a DataFrame,  
we always want to sort with either the largest or smallest values at the top of  
a DataFrame.

There are two equivalent operations to do this **sort** and **orderBy** that work  
the exact same way.

They accept both column expressions and strings as well as multiple  
columns.

***The default is to sort in ascending order:***



# Sort

```
In [145]: df.sort("count").show(5)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
Malta	United States	1
Saint Vincent and...	United States	1
United States	Croatia	1
United States	Gibraltar	1
United States	Singapore	1

only showing top 5 rows

```
In [146]: df.orderBy("count","DEST_COUNTRY_NAME").show(5)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
Burkina Faso	United States	1
Cote d'Ivoire	United States	1
Cyprus	United States	1
Djibouti	United States	1
Indonesia	United States	1

only showing top 5 rows

```
In [147]: df.orderBy(col("count"),col("DEST_COUNTRY_NAME")).show(5)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
Burkina Faso	United States	1
Cote d'Ivoire	United States	1
Cyprus	United States	1
Djibouti	United States	1
Indonesia	United States	1

only showing top 5 rows



```
In [148]: from pyspark.sql.functions import desc,asc
```

```
In [149]: df.orderBy(expr("count desc")).show(2)
```

```
+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+
|      Moldova|        United States|    1|
|United States|            Croatia|    1|
+-----+-----+
only showing top 2 rows
```

```
In [150]: df.orderBy(col("count").desc(),col("DEST_COUNTRY_NAME").asc()).show(2)
```

```
+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME| count|
+-----+-----+
|   United States|        United States|370002|
|   United States|           Canada| 8483|
+-----+-----+
only showing top 2 rows
```



For optimization purposes, it's sometimes advisable to sort within each partition before another set of transformations.

We can use the **sortWithinPartitions** method to do this:

```
In [151]: spark.read.format("json").load("/home/ilg/Documents/sparkdata/data/flight-data/json/*-summ*.json")\
...: .sortWithinPartitions("count")
Out[151]: DataFrame[DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string, count: bigint]
```



# Limit

We might want to restrict what we extract from a DataFrame

E.g We might want just the top ten of some DataFrame.

We can do this by using the **limit** method:

```
In [152]: df.limit(5).show()
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
| United States | Romania | 15 |
| United States | Croatia | 1 |
| United States | Ireland | 344 |
| Egypt | United States | 15 |
| United States | India | 62 |
+-----+-----+-----+


In [153]: df.orderBy(expr("count desc")).limit(6).show()
...
+-----+-----+-----+
| DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
| Malta | United States | 1 |
| Saint Vincent and... | United States | 1 |
| United States | Croatia | 1 |
| United States | Gibraltar | 1 |
| United States | Singapore | 1 |
| Moldova | United States | 1 |
+-----+-----+-----+
```

# Repartition and Coalesce



We have important optimization opportunity to partition the data according to some frequently filtered columns, which control the physical layout of data across the cluster including the partitioning scheme and the number of partitions.

**Repartition will incur a full shuffle of the data**, regardless of whether one is necessary.

So when to repartition ?

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

# Repartition and Coalesce



```
In [154]: df.rdd.getNumPartitions()
Out[154]: 1

In [155]: df.repartition(5)
Out[155]: DataFrame[DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string, count: bigint]
```



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

```
In [156]: df.repartition(col("DEST_COUNTRY_NAME"))
Out[156]: DataFrame[DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string, count: bigint]
```

We can optionally specify the number of partitions we would like, too:

```
In [157]: df.repartition(5, col("DEST_COUNTRY_NAME"))
Out[157]: DataFrame[DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string, count: bigint]
```

# Coalesce



Coalesce 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)

```
In [158]: df.repartition(5,col("DEST_COUNTRY_NAME")).coalesce(2)
Out[158]: DataFrame[DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string, count: bigint]
```

# 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.

**collect** gets all data from the entire DataFrame, `take` selects the first N rows, and `show` prints out a number of rows nicely.



```
In [159]: collectDf = df.limit(10)

In [160]: collectDf.take(5)
Out[160]:
[Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'Romania', count=15),
 Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'Croatia', count=1),
 Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'Ireland', count=344),
 Row(DEST_COUNTRY_NAME=u'Egypt', ORIGIN_COUNTRY_NAME=u'United States', count=15),
 Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'India', count=62)]

In [161]: collectDf.show()
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
| United States| Romania| 15|
| United States| Croatia| 1|
| United States| Ireland| 344|
| Egypt| United States| 15|
| United States| India| 62|
| United States| Singapore| 1|
| United States| Grenada| 62|
| Costa Rica| United States| 588|
| Senegal| United States| 40|
| Moldova| United States| 1|
+-----+-----+-----+
```



```
In [163]: collectDf.show(5, False)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Romania	15
United States	Croatia	1
United States	Ireland	344
Egypt	United States	15
United States	India	62

only showing top 5 rows

```
In [164]: collectDf.collect()
```

```
Out[164]:
```

```
[Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'Romania', count=15),
 Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'Croatia', count=1),
 Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'Ireland', count=344),
 Row(DEST_COUNTRY_NAME=u'Egypt', ORIGIN_COUNTRY_NAME=u'United States', count=15),
 Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'India', count=62),
 Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'Singapore', count=1),
 Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'Grenada', count=62),
 Row(DEST_COUNTRY_NAME=u'Costa Rica', ORIGIN_COUNTRY_NAME=u'United States', count=588),
 Row(DEST_COUNTRY_NAME=u'Senegal', ORIGIN_COUNTRY_NAME=u'United States', count=40),
 Row(DEST_COUNTRY_NAME=u'Moldova', ORIGIN_COUNTRY_NAME=u'United States', count=1)]
```

# DANGEROUS



Any collection of data to the driver can be a very expensive operation!

If we have a large dataset and call collect, we can crash the driver.

If we use `toLocalIterator` and have very large partitions, we 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.



## Working with Different Types of Data



## Learning about:

Working with a variety of different kinds of data.

- Boolean
- Numbers
- Strings
- Dates and timestamps
- Handling null
- Complex types
- User-defined functions

# API References



DataFrameStatFunctions

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameStatFunctions>

DataFrameNaFunctions

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameNaFunctions>

Columns Methods

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column>

SQL and DataFrame Functions

[http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions\\$](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$)



# Our csv dataframe

let's read in the DataFrame that we'll be using for this analysis:

```
In [180]: csvdf = spark.read.format("csv")\
    ...: .option("header", "true")\
    ...: .option("inferSchema", "true")\
    ...: .load("/home/ilg/Documents/sparkdata/data/retail-data/by-day/2010-12-01.csv")

In [181]: csvdf.printSchema()
root
|-- InvoiceNo: string (nullable = true)
|-- StockCode: string (nullable = true)
|-- Description: string (nullable = true)
|-- Quantity: integer (nullable = true)
|-- InvoiceDate: timestamp (nullable = true)
|-- UnitPrice: double (nullable = true)
|-- CustomerID: double (nullable = true)
|-- Country: string (nullable = true)

In [182]: csvdf.createOrReplaceTempView("csvdftable")
```

# Sample Data - csv



```
ilgeD10Spark:~/Documents/sparkdata/data/retail-data/by-day$ head 2010-12-01.csv
InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country
536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,2010-12-01 08:26:00,2.55,17850.0,United Kingdom
536365,71053,WHITE METAL LANTERN,6,2010-12-01 08:26:00,3.39,17850.0,United Kingdom
536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,2010-12-01 08:26:00,2.75,17850.0,United Kingdom
536365,84029G,KNITTED UNION FLAG HOT WATER BOTTLE,6,2010-12-01 08:26:00,3.39,17850.0,United Kingdom
536365,84029E,RED WOOLLY HOTTIE WHITE HEART.,6,2010-12-01 08:26:00,3.39,17850.0,United Kingdom
536365,22752,SET 7 BABUSHKA NESTING BOXES,2,2010-12-01 08:26:00,7.65,17850.0,United Kingdom
536365,21730,GLASS STAR FROSTED T-LIGHT HOLDER,6,2010-12-01 08:26:00,4.25,17850.0,United Kingdom
536366,22633,HAND WARMER UNION JACK,6,2010-12-01 08:28:00,1.85,17850.0,United Kingdom
536366,22632,HAND WARMER RED POLKA DOT,6,2010-12-01 08:28:00,1.85,17850.0,United Kingdom
```

# Working with Booleans



Boolean statements consist of four elements: and, or, true, and false.

We use these structures to build logical statements that evaluate to either true or false.

These statements are often used as conditional requirements for when a row of data must either pass the test (evaluate to true) or else it will be filtered out.

# booleans



```
In [206]: from pyspark.sql.functions import col  
  
In [207]: csvdf.where(col("InvoiceNo") != 536365) \  
....: .select("InvoiceNo", "Description") \  
....: .show(3, True)
```

InvoiceNo	Description
536366	HAND WARMER UNION...
536366	HAND WARMER RED P...
536367	ASSORTED COLOUR B...

only showing top 3 rows

```
In [208]: csvdf.where(col("InvoiceNo") != 536365) \  
....: .select("InvoiceNo", "Description") \  
....: .show(3, False)
```

InvoiceNo	Description
536366	HAND WARMER UNION JACK
536366	HAND WARMER RED POLKA DOT
536367	ASSORTED COLOUR BIRD ORNAMENT

only showing top 3 rows

# Boolean with and,or...



## pyspark.sql.functions.instr(str, substr)

Locate the position of the first occurrence of substr column in the given string.

```
In [216]: from pyspark.sql.functions import instr
In [217]: priceFilter = col("UnitPrice") > 600
In [218]: descripFilter = instr(csvdf.Description, "POSTAGE") >= 1
In [219]: csvdf.where(csvdf.StockCode.isin("DOT")).where(priceFilter | descripFilter).show()
+-----+-----+-----+-----+-----+-----+
|InvoiceNo|StockCode| Description|Quantity|      InvoiceDate|UnitPrice|CustomerID|      Country|
+-----+-----+-----+-----+-----+-----+
| 536544|    DOT|DOTCOM POSTAGE|      1|2010-12-01 14:32:00|   569.77|      null|United Kingdom|
| 536592|    DOT|DOTCOM POSTAGE|      1|2010-12-01 17:06:00|   607.49|      null|United Kingdom|
+-----+-----+-----+-----+-----+-----+
```



## Boolean with and,or...

Boolean expressions are not just reserved to filters.

To filter a DataFrame, you can also just specify a Boolean column

```
In [236]: from pyspark.sql.functions import instr
In [238]: DOTCodeFilter = col("StockCode")=="DOT"
In [239]: priceFilter = col("UnitPrice") > 600
In [240]: descripFilter = instr(col("Description"),"POSTAGE") >= 1
In [241]: csvdf.withColumn("isExpensive",DOTCodeFilter & (priceFilter | descripFilter))\
....: .where("isExpensive")\
....: .select("unitPrice","isExpensive").show(5)
+-----+-----+
|unitPrice|isExpensive|
+-----+-----+
| 569.77 |      true |
| 607.49 |      true |
+-----+-----+
```

# Working with Numbers



When working with big data, the second most common task you will do after filtering things is counting things.

We simply need to express our computation, and that should be valid assuming that we're working with numerical data types.

For Instance:-

let's imagine that we found out that we mis-recorded the quantity in our retail dataset and the true quantity is equal to **(the current quantity \* the unit price)<sup>2</sup> + 5**.

This will introduce our first numerical function as well as the **pow** function that raises a column to the expressed power:



# pow

## pyspark.sql.functions.pow(col1, col2)

Returns the value of the first argument raised to the power of the second argument.

```
In [17]: from pyspark.sql.functions import expr,pow
In [18]: fabricatedQuantity = pow(col('Quantity') * col("UnitPrice"),2) + 5
In [19]: df.select(expr("CustomerId"),fabricatedQuantity.alias("realQuantity")).show(3)
+-----+-----+
|CustomerId|    realQuantity|
+-----+-----+
| 17850.0|239.0899999999997|
| 17850.0|      418.7156|
| 17850.0|        489.0|
+-----+-----+
only showing top 3 rows
```



We can add and subtract as necessary, as well.

```
In [36]: df.selectExpr(
    ...:     "CustomerId",
    ...:     "(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity").show(2)
+-----+-----+
|CustomerId|      realQuantity|
+-----+-----+
| 17850.0|239.08999999999997|
| 17850.0|        418.7156|
+-----+-----+
only showing top 2 rows
```



# Rounding

By default, the round function rounds up if you're exactly in between two numbers. You can round down by using the **bround**:

```
In [38]: from pyspark.sql.functions import lit, round, bround  
  
In [39]: df.select(round(lit("2.5")), bround(lit("2.5"))).show(2)  
+-----+-----+  
| round(2.5, 0) | bround(2.5, 0) |  
+-----+-----+  
|      3.0 |        2.0 |  
|      3.0 |        2.0 |  
+-----+-----+  
only showing top 2 rows
```

# Correlation of two columns



We can see the **Pearson correlation** coefficient for two columns to see if cheaper things are typically bought in greater quantities.

*In statistics, the Pearson correlation coefficient is a measure of the linear correlation between two variables X and Y.*

Pearson's correlation coefficient is a statistical measure of the strength of a linear relationship between paired data.

In a sample it is denoted by r and is by design constrained as follows

$$-1 \leq r \leq 1$$

Furthermore:

- 1) Positive values denote positive linear correlation;
- 2) Negative values denote negative linear correlation;
- 3) A value of 0 denotes no linear correlation;
- 4) The closer the value is to 1 or -1, the stronger the linear correlation.

# Correlation of two columns



```
In [47]: from pyspark.sql.functions import corr  
In [48]: df.stat.corr("Quantity", "UnitPrice")  
Out[48]: -0.04112314436835551  
  
In [49]: df.select(corr("Quantity", "UnitPrice")).show()  
+-----+  
|corr(Quantity, UnitPrice)|  
+-----+  
|-0.04112314436835551|  
+-----+
```

# Summary Stats



Compute summary statistics for a column or set of columns.

This will take all numeric columns and calculate the **count**, **mean**, **standard deviation**, **min**, and **max**.

In [50]: df.describe().show()

summary	InvoiceNo	StockCode	Description	Quantity	UnitPrice	CustomerID	Country
count	3108	3108	3098	3108	3108	1968	3108
mean	536516.684944841	27834.304044117645	null	8.627413127413128	4.151946589446603	15661.388719512195	null
stddev	72.89447869788873	17407.897548583845	null	26.371821677029203	15.638659854603892	1854.4496996893627	null
min	536365	10002	4 PURPLE FLOCK D...	-24	0.0	12431.0	Australia
max	C536548	POST	ZINC WILLIE WINKI...	600	607.49	18229.0	United Kingdom



```
In [51]: from pyspark.sql.functions import count,mean,stddev,min,max
```

```
In [90]: df.select(min("UnitPrice")).show()
+-----+
|min(UnitPrice)|
+-----+
|          0.0|
+-----+
```

```
In [91]: df.select(max("UnitPrice")).show()
+-----+
|max(UnitPrice)|
+-----+
|      607.49|
+-----+
```

```
In [92]: df.select(stddev("UnitPrice")).show()
+-----+
|stddev_samp(UnitPrice)|
+-----+
|    15.638659854603892|
+-----+
```



## StatFunction

These are DataFrame methods that you can use to calculate a variety of different things.

E.g

We can calculate either exact or approximate quantiles of your data using the **approxQuantile** method:

```
In [93]: colName = "UnitPrice"
In [94]: quantileProbs = [0.5]
In [95]: relErr = 0.05
In [96]: df.stat.approxQuantile(colName, quantileProbs, relErr)
Out[96]: [2.51]
```



# Crosstab & freqItems

We also can use this to see a **cross-tabulation** or **frequent** item pairs.

Note: Output could be very large.

## crosstab(col1, col2)

Computes a pair-wise frequency table of the given columns. Also known as a contingency table. The number of distinct values for each column should be less than 1e4. At most 1e6 non-zero pair frequencies will be returned.

The first column of each row will be the distinct values of col1 and the column names will be the distinct values of col2. The name of the first column will be \$col1\_\$col2. Pairs that have no occurrences will have zero as their counts.

```
In [110]: df.stat.freqItems(["StockCode", "Quantity"]).show()
```

StockCode_freqItems	Quantity_freqItems
[90214E, 20728, 2...	[200, 128, 23, 32...



# Crosstab

Not the full output has been shown



# Unique ID

We can also add a unique ID to each row by using the function **monotonically\_increasing\_id**. This function generates a unique value for each row, starting with 0.

```
In [113]: from pyspark.sql.functions import monotonically_increasing_id  
  
In [114]: df.select(monotonically_increasing_id()).show(2)  
+-----+  
|monotonically_increasing_id()|  
+-----+  
|          0|  
|          1|  
+-----+  
only showing top 2 rows
```

```
In [115]: df.select(monotonically_increasing_id()).show(10)  
+-----+  
|monotonically_increasing_id()|  
+-----+  
|          0|  
|          1|  
|          2|  
|          3|  
|          4|  
|          5|  
|          6|  
|          7|  
|          8|  
|          9|  
+-----+  
only showing top 10 rows
```



# Working with Strings



# initcap

## pyspark.sql.functions.initcap(col)

Translate the first letter of each word to upper case in the sentence.

```
In [120]: df.select(initcap(col("Description"))).show()
+-----+
|initcap(Description)|
+-----+
|White Hanging Hea...
| White Metal Lantern|
|Cream Cupid Heart...
|Knitted Union Fla...
|Red Woolly Hottie...
|Set 7 Babushka Ne...
|Glass Star Froste...
|Hand Warmer Union...
|Hand Warmer Red P...
|Assorted Colour B...
|Poppy's Playhouse...
|Poppy's Playhouse...
|Feltcraft Princes...
|Ivory Knitted Mug...
|Box Of 6 Assorted...
|Box Of Vintage Ji...
|Box Of Vintage Al...
|Home Building Blo...
|Love Building Blo...
|Recipe Box With M...
+-----+
only showing top 20 rows
```

```
In [6]: from pyspark.sql.functions import *
```

```
In [7]: spark.createDataFrame([('ab cd',)], ['a']).select(initcap("a").alias('v')).collect()
Out[7]: [Row(v=u'Ab Cd')]
```

# Upper and Lower case



We can cast strings in uppercase and lowercase

**pyspark.sql.functions.upper(col)**

Converts a string column to upper case.

**pyspark.sql.functions.lower(col)**

Converts a string column to lower case.

```
In [126]: df.select(col("Description"),lower(col("Description")),upper(lower(col("Description")))).show(5)
+-----+-----+-----+
| Description| lower	Description)|upper(lower>Description))|
+-----+-----+-----+
|WHITE HANGING HEA...|white hanging hea...|    WHITE HANGING HEA...|
| WHITE METAL LANTERN| white metal lantern|    WHITE METAL LANTERN|
| CREAM CUPID HEART...|cream cupid heart...|    CREAM CUPID HEART...|
| KNITTED UNION FLA...|knitted union fla...|    KNITTED UNION FLA...|
| RED WOOLLY HOTTIE...|red woolly hottie...|    RED WOOLLY HOTTIE...
+-----+-----+-----+
only showing top 5 rows
```



# Trimming

Adding or removing spaces around a string.

We can do this by using lpad, ltrim, rpad and rtrim, trim:

```
In [159]: from pyspark.sql.functions import lit,ltrim,rtrim,rpad,lpad,trim
In [160]: df.select(
...:     ...),
...:     ltrim(lit("    Hey    ")).alias("ltrim"),
...:     rtrim(lit("    Hey    ")).alias("rtrim"),
...:     trim(lit("    Hey    ")).alias("trim"),
...:     lpad(lit("Hello"),10, " ").alias("lp"),
...:     rpad(lit("Hey"),10, " ").alias("rp")).show(5)
+-----+-----+-----+-----+
| ltrim| rtrim| trim|   lp|   rp|
+-----+-----+-----+-----+
|Hey| Hey| Hey| Hello|Hey| |
+-----+-----+-----+-----+
only showing top 5 rows
```

**Note that** if lpad or rpad takes a number less than the length of the string, it will always remove values from the right side of the string.



# Regular Expressions

# regex\_replace



**pyspark.sql.functions regexp\_replace(str, pattern, replacement)**

Replace all substrings of the specified string value that match regexp with rep.

One of the most frequently performed tasks is searching for the existence of one string in another or replacing all mentions of a string with another value.

This is often done with a tool called regular expressions that exists in many programming languages.

Regular expressions give the user an ability to specify a set of rules to use to either extract values from a string or replace them with some other values.



```
In [165]: from pyspark.sql.functions import regexp_replace
In [166]: reg_string = "BLACK|WHITE|RED|GREEN|BLUE"
In [167]: df.select(
    ...:     regexp_replace(col("Description"), reg_string, "COLOR").alias("color_clean")
    ...:     .show(3, False)
+-----+-----+
|color_clean          |Description      |
+-----+-----+
|COLOR HANGING HEART T-LIGHT HOLDER|WHITE HANGING HEART T-LIGHT HOLDER|
|COLOR METAL LANTERN           |WHITE METAL LANTERN|
|CREAM CUPID HEARTS COAT HANGER|CREAM CUPID HEARTS COAT HANGER|
+-----+-----+
only showing top 3 rows
```

# Translate



**translate** function to replace given characters with other characters.

This is done *at the character level* and will replace all instances of a character with the indexed character in the replacement string:

```
In [176]: from pyspark.sql.functions import translate

In [177]: df.select(translate(col("Description"),"LEET","1337"),col("Description")).show(3,False)
+-----+-----+
|translate(Description, LEET, 1337)|Description
+-----+-----+
|WHI73 HANGING H3AR7 7-1IGH7 H01D3R|WHITE HANGING HEART T-LIGHT HOLDER
|WHI73 M37A1 1AN73RN               |WHITE METAL LANTERN
|CR3AM CUPID H3AR7S COA7 HANG3R   |CREAM CUPID HEARTS COAT HANGER
+-----+
only showing top 3 rows

In [178]: df.select(translate(col("Description"),"LEET","2019"),col("Description")).show(3,False)
+-----+-----+
|translate(Description, LEET, 2019)|Description
+-----+-----+
|WHI90 HANGING HOAR9 9-2IGH9 H02D0R|WHITE HANGING HEART T-LIGHT HOLDER
|WHI90 M09A2 2AN90RN               |WHITE METAL LANTERN
|CR0AM CUPID H0AR9S COA9 HANG0R   |CREAM CUPID HEARTS COAT HANGER
+-----+
only showing top 3 rows
```



# regexp\_extract

`pyspark.sql.functions.regexp_extract(str, pattern, idx)`

Extract a specific group matched by a Java regex, from the specified string column. If the regex did not match, or the specified group did not match, an empty string is returned.

```
In [227]: from pyspark.sql.functions import regexp_extract
```

```
In [228]: cvdf.select(
...:     regexp_extract(col("Description"), extract_str, 1).alias("color_clean"), col("Description")).show(4)
+-----+-----+
|color_clean|      Description|
+-----+-----+
|    WHITE|WHITE HANGING HEA...
|    WHITE|  WHITE METAL LANTERN
|        | CREAM CUPID HEART...
|        | KNITTED UNION FLA...
+-----+-----+
only showing top 4 rows
```

```
In [16]: from pyspark.sql.functions import regexp_extract
```

```
In [17]: hdf = spark.createDataFrame([('100-200',)], ['str'])
```

```
In [18]: hdf.select(regexp_extract('str', r'(\d+)-(\d+)', 1).alias('d')).collect()
Out[18]: [Row(d=u'100')]
```

```
In [19]: hdf.show()
```

```
+-----+
|      str|
+-----+
|100-200|
+-----+
```

# Checking existence - **contains**



'**contains**' method checks for existence of values.

return a Boolean declaring whether the value you specify is in the column's string.

```
In [229]: from pyspark.sql.functions import instr
In [230]: containsBlack = instr(col("Description"), "BLACK") >= 1
In [231]: containsWhite = instr(col("Description"), "WHITE") >= 1
In [232]: cvdf.withColumn("hasSimpleColor",containsBlack | containsWhite) \
    .... .where("hasSimpleColor") \
    .... .select("Description").show(3,False)
+-----+
|Description
+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|
|WHITE METAL LANTERN
|RED WOOLLY HOTTIE WHITE HEART.
+-----+
only showing top 3 rows
```

# Dynamic args



Spark's ability to accept a dynamic number of arguments.

When we convert a list of values into a set of arguments and pass them into a function, we use a language feature called **varargs**.

Using this feature, we can effectively unravel an array of arbitrary length and pass it as arguments to a function.

This, coupled with select makes it possible for us to create arbitrary numbers of columns dynamically



```
In [239]: from pyspark.sql.functions import expr,locate
In [240]: simpColors = ['black','white','red','green','blue']
In [241]: def colorLocator(column,color_string):
....:     return locate(color_string.upper(),column).cast("boolean").alias("is_" + c)
....:
In [242]: selectedColumns = [colorLocator(cvdf.Description,c) for c in simpColors]
In [243]: selectedColumns.append(expr("*"))
In [244]: cvdf.select(*selectedColumns).where(expr("is_white OR is_red"))\
....: .select("Description").show(3,False)
+-----+
|Description|
+-----+
|WHITE HANGING HEART T-LIGHT HOLDER|
|WHITE METAL LANTERN|
|RED WOOLLY HOTTIE WHITE HEART.|
+-----+
only showing top 3 rows
```



# Working with Dates and Timestamps

# Date & Time



Create a DF with current\_timestamp and current\_date

```
In [246]: from pyspark.sql.functions import current_timestamp, current_date
In [247]: datedf = spark.range(10) \
    ....: .withColumn("today", current_date()) \
    ....: .withColumn("now", current_timestamp())
In [248]: datedf.createOrReplaceTempView("dateTable")
In [249]: datedf.printSchema()
root
 |-- id: long (nullable = false)
 |-- today: date (nullable = false)
 |-- now: timestamp (nullable = false)
```

# Date,Time manipulation



Now that we have a simple DataFrame to work with.

let's add and subtract five days from today.

These functions take a column and then the number of days to either add or subtract as the arguments:

```
In [250]: from pyspark.sql.functions import date_add,date_sub
In [251]: datedf.select(date_sub(col("today"),5),date_add(col("today"),5)).show(1)
+-----+-----+
|date_sub(today, 5)|date_add(today, 5)|
+-----+-----+
|      2019-09-03|      2019-09-13|
+-----+-----+
only showing top 1 row
```

# Date & month difference



Let's take a look at the difference between two dates.

**datediff** function that will return the number of days in between two dates. Most often we just care about the days, and because the number of days varies from month to month, there also exists a function, **months\_between**, that gives you the number of months between two dates

```
In [254]: from pyspark.sql.functions import datediff,months_between,to_date  
  
In [255]: datedf.withColumn("week_ago",date_sub(col("today"),7))\  
....: .select(datediff(col("week_ago"),col("today"))).show(1)  
+-----+  
|datediff(week_ago, today)|  
+-----+  
|-7|  
+-----+  
only showing top 1 row
```



## to\_date, month\_between

```
In [256]: datedf.select(
    ...: to_date(lit("2016-01-01")).alias("start"),
    ...: to_date(lit("2017-05-25")).alias("end"))\
    ...: .select(months_between(col("start"),col("end")))).show(1)
+-----+
|months_between(start, end, true)|
+-----+
|-16.77419355|
+-----+
only showing top 1 row

In [257]: datedf.select(
    ...: to_date(lit("2019-01-01")).alias("end"),
    ...: to_date(lit("2017-05-25")).alias("start"))\
    ...: .select(months_between(col("end"),col("start")))).show(1)
+-----+
|months_between(end, start, true)|
+-----+
|19.22580645|
+-----+
only showing top 1 row
```

Spark will not throw an error if it cannot parse the date; rather, it will just return null.



## A problem – bug like

```
In [260]: datedf.select(to_date(lit("2016-20-12")),to_date(lit("2017-12-11"))).show(1)
+-----+-----+
| to_date('2016-20-12')|to_date('2017-12-11')|
+-----+-----+
|         null|          2017-12-11|
+-----+-----+
only showing top 1 row
```

Spark doesn't throw an error because it cannot know whether the days are mixed up or that specific row is incorrect.

# Fix



The first step is to remember that we need to specify our date format according to the Java **SimpleDateFormat** standard.

<https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html>

We will use two functions to fix this: **to\_date** and **to\_timestamp**.

```
In [263]: from pyspark.sql.functions import to_date  
  
In [264]: dateFormat = "yyyy-dd-MM"  
  
In [265]: corrDatedf = spark.range(1).select(  
....: to_date(lit("2017-12-11"),dateFormat).alias("date"),  
....: to_date(lit("2017-20-12"),dateFormat).alias("date2"))  
  
In [266]: corrDatedf.createOrReplaceTempView("dateTable2")
```

```
In [267]: corrDatedf.show()  
+-----+-----+  
|      date|    date2|  
+-----+-----+  
|2017-11-12|2017-12-20|  
+-----+-----+
```



```
In [269]: from pyspark.sql.functions import to_timestamp  
  
In [270]: corrDatedf.select(to_timestamp(col("date"),dateFormat)).show()  
+-----+  
|to_timestamp(`date`, 'yyyy-dd-MM')|  
+-----+  
| 2017-11-12 00:00:00|  
+-----+
```

After we have our date or timestamp in the correct format and type, comparing between them is actually easy.

We just need to be sure to either use a date/timestamp type or specify our string according to the right format of yyyy-MM-dd if we're comparing a date

```
In [272]: corrDatedf.filter(col("date2") > lit("2017-12-12")).show()  
+-----+-----+  
|      date|      date2|  
+-----+-----+  
|2017-11-12|2017-12-20|  
+-----+
```



## Working with Nulls in Data



## nulls

As a best practice, we should always use **nulls** to represent missing or empty data in your DataFrames.

Spark can optimize working with null values more than it can if you use empty strings or other values.

The primary way of interacting with null values, at DataFrame scale, is to use the **.na subpackage** on a DataFrame.

There are also several functions for performing operations and explicitly specifying how Spark should handle null values.

*There are two things you can do with null values: you can explicitly drop nulls or you can fill them with a value (globally or on a per-column basis).*

# Coalesce



Spark includes a function to allow you to select the first non-null value from a set of columns by using the coalesce function.

```
In [274]: cvdf.select(coalesce(col("Description"), col("CustomerId"))).show()
```

coalesce(Description, CustomerId)
WHITE HANGING HEA...
WHITE METAL LANTERN
CREAM CUPID HEART...
KNITTED UNION FLA...
RED WOOLLY HOTTIE...
SET 7 BABUSHKA NE...
GLASS STAR FROSTE...
HAND WARMER UNION...
HAND WARMER RED P...
ASSORTED COLOUR B...
POPPY'S PLAYHOUSE...
POPPY'S PLAYHOUSE...
FELTCRAFT PRINCES...

## ifnull, nullif, nvl, and nvl2



There are several other **SQL functions** that you can use to achieve similar things.

- 1) **ifnull** allows we to select the second value if the first is null, and defaults to the first.
- 2) we could use **nullif**, which returns null if the two values are equal or else returns the second if they are not.
- 3) **nvl** returns the second value if the first is null, but defaults to the first.
- 4) **nvl2** returns the second value if the first is not null; otherwise, it will return the last specified value



```
-- in SQL
SELECT
    ifnull(null, 'return_value'),
    nullif('value', 'value'),
    nvl(null, 'return_value'),
    nvl2('not_null', 'return_value', "else_value")
FROM dfTable LIMIT 1
```

	a	b	c	d
return_value	null	return_value	return_value	

# Drop



## **drop(how='any', thresh=None, subset=None)**

Returns a new DataFrame omitting rows with null values.

DataFrame.dropna() and DataFrameNaFunctions.drop() are aliases of each other.

### Parameters

- how – ‘**any**’ or ‘**all**’. If ‘any’, drop a row if it contains any nulls. If ‘all’, drop a row only if all its values are null.
- thresh – int, default None If specified, drop rows that have less than thresh non-null values. This overwrites the how parameter.
- subset – optional list of column names to consider.



# drop

**drop**, which removes rows that contain nulls.

The default is to drop any row in which any value is null:

```
In [20]: nuldf = spark.createDataFrame([(1,"B","Sieben"),(None,None,None),(None,"A","X1"),(None,"C",None)],("ID","TYPE","CODE"))
```

```
In [21]: nuldf.show()
```

```
+---+---+---+
| ID|TYPE| CODE|
+---+---+---+
| 1| B|Sieben|
| null| null| null|
| null| A| X1|
| null| C| null|
+---+---+---+
```

```
In [24]: nuldf.na.drop(how="any").show()
```

```
+---+---+---+
| ID|TYPE| CODE|
+---+---+---+
| 1| B|Sieben|
+---+---+---+
```



```
In [286]: cvdf.na.drop("all",subset=["StockCode","InvoiceNo"])
```

```
Out[286]: DataFrame[InvoiceNo: string, StockCode: string, Description: string, Quantity: int, InvoiceDate: timestamp, UnitPrice: double, CustomerID: double, Country: string]
```

```
In [35]: nuldf.na.drop(how='all').show()
```

```
+---+---+---+
| ID|TYPE| CODE|
+---+---+---+
| 1|  B|Sieben|
| null| A|    X1|
| null| C|  null|
+---+---+---+
```

```
In [36]: nuldf.na.drop(how='any').show()
```

```
+---+---+---+
| ID|TYPE| CODE|
+---+---+---+
| 1|  B|Sieben|
+---+---+---+
```

# fill



**fill(value, subset=None)**

Replace null values, alias for `na.fill()`.

`DataFrame.fillna()` and `DataFrameNaFunctions.fill()` are aliases of each other.

## Parameters

- **value** – int, long, float, string, bool or dict. Value to replace null values with. If the value is a dict, then subset is ignored and value must be a mapping from column name (string) to replacement value. The replacement value must be an int, long, float, boolean, or string.
- **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if value is a string, and subset contains a non-string column, then the non-string column is simply ignored.



Using the `fill` function, we can fill one or more columns with a set of values.

This can be done by specifying a map—that is a particular value and a set of columns.

To fill all null values in columns of type String, you might specify the following:

```
In [33]: nuludf = spark.createDataFrame([(10,80,'Alice'),(None,None,None),(20,30,'Kirk'),(None,None,None)],('ID','CODE','NAME'))
```

```
In [40]: nuludf.show()
```

ID	CODE	NAME
10	80	Alice
null	null	null
20	30	Kirk
null	null	null

```
In [43]: nuludf.na.fill("All is well").show()
```

ID	CODE	NAME
10	80	Alice
null	null	All is well
20	30	Kirk
null	null	All is well



```
In [54]: nuludf.na.fill({'ID':15,'CODE':40,'NAME':'Bones'}).show()
```

ID	CODE	NAME
10	80	Alice
15	40	Bones
20	30	Kirk
15	40	Bones

```
In [56]: nuludf.na.fill(10,subset=["ID","NAME","TYPE"]).show()
```

ID	CODE	NAME
10	80	Alice
10	null	null
20	30	Kirk
10	null	null



# replace

`replace(to_replace, value=<no value>, subset=None)`

Returns a new DataFrame replacing a value with another value.

*Replace all values in a certain column according to their current value.*

```
In [60]: nuludf.na.replace(20,25).show()
+---+---+---+
| ID|CODE| NAME|
+---+---+---+
| 10| 80|Alice|
| null|null| null|
| 25| 30| Kirk|
| null|null| null|
+---+---+---+
```

```
In [61]: nuludf.na.replace('Alice','Balice').show()
+---+---+---+
| ID|CODE| NAME|
+---+---+---+
| 10| 80|Balice|
| null|null| null|
| 20| 30| Kirk|
| null|null| null|
+---+---+---+
```

```
In [62]: nuludf.na.replace('Alice',None).show()
+---+---+---+
| ID|CODE|NAME|
+---+---+---+
| 10| 80|null|
| null|null| null|
| 20| 30|Kirk|
| null|null| null|
+---+---+---+
```



```
In [68]: nuludf.na.replace(['Alice','Kirk'],['Stevi','Spock'],'NAME').show()
```

ID	CODE	NAME
10	80	Stevi
null	null	null
20	30	Spock
null	null	null



## Working with Complex Types

# Complex types



Complex types can help you organize and structure your data in ways that make more sense for the problem that you are hoping to solve.

There are three kinds of complex types:

- 1) structs**
- 2) arrays**
- 3) maps.**

# Structs



```
In [305]: from pyspark.sql.functions import struct  
In [306]: complexdf = cvdf.select(struct("Description","InvoiceNo").alias("complex"))  
In [307]: complexdf.createOrReplaceTempView("complexDf")
```

# Structs



We can think of structs as DataFrames within DataFrames.

We can create a struct by wrapping a set of columns in parenthesis in a query

```
In [316]: cvdf.selectExpr("(Description,InvoiceNo) as complex","*").show()
```

StockCode	complex	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
[WHITE HANGING HE...	536365	85123A	WHITE HANGING HEA...		6	2010-12-01 08:26:00	2.55	17850.0	United Kingdom
[WHITE METAL LANT...	536365	71053	WHITE METAL LANTERN		6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
[CREAM CUPID HEAR...	536365	84406B	CREAM CUPID HEART...		8	2010-12-01 08:26:00	2.75	17850.0	United Kingdom
[KNITTED UNION FL...	536365	84029G	KNITTED UNION FLA...		6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
[RED WOOLLY HOTTIE...	536365	84029E	RED WOOLLY HOTTIE...		6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
[SET 7 BABUSHKA N...	536365	22752	SET 7 BABUSHKA NE...		2	2010-12-01 08:26:00	7.65	17850.0	United Kingdom
[GLASS STAR FROST...	536365	21730	GLASS STAR FROSTE...		6	2010-12-01 08:26:00	4.25	17850.0	United Kingdom
[HAND WARMER UNIO...	536366	22633	HAND WARMER UNION...		6	2010-12-01 08:28:00	1.85	17850.0	United Kingdom

```
In [318]: cvdf.selectExpr("struct(Description,InvoiceNo) as complex","*").show()
```

StockCode	complex	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
[WHITE HANGING HE...	536365	85123A	WHITE HANGING HEA...		6	2010-12-01 08:26:00	2.55	17850.0	United Kingdom
[WHITE METAL LANT...	536365	71053	WHITE METAL LANTERN		6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
[CREAM CUPID HEAR...	536365	84406B	CREAM CUPID HEART...		8	2010-12-01 08:26:00	2.75	17850.0	United Kingdom
[KNITTED UNION FL...	536365	84029G	KNITTED UNION FLA...		6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
[RED WOOLLY HOTTIE...	536365	84029E	RED WOOLLY HOTTIE...		6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
[SET 7 BABUSHKA N...	536365	22752	SET 7 BABUSHKA NE...		2	2010-12-01 08:26:00	7.65	17850.0	United Kingdom



# Array

To define arrays, let's work through a use case. With our current data, our objective is to take every single word in our Description column and convert that into a row in our DataFrame.

The first task is to turn our Description column into a complex type, an array.

## Split

We do this by using the split function and specify the delimiter:

```
In [330]: from pyspark.sql.functions import split  
  
In [331]: cvdf.select(split(col("Description"), " ")).show(2)  
+-----+  
|split	Description, )|  
+-----+  
| [WHITE, HANGING, ... |  
| [WHITE, METAL, LA... |  
+-----+  
only showing top 2 rows
```



# Array

Spark allows us to manipulate this complex type as another column.

```
In [333]: cvdf.select(split(col("Description"), " ")).alias("array_col"))\n...: .selectExpr("array_col[0]").show(2)\n+-----+\n|array_col[0]|\n+-----+\n|      WHITE|\n|      WHITE|\n+-----+\nonly showing top 2 rows
```

# Array Length



We can determine the array's length by querying for its size:

```
In [334]: from pyspark.sql.functions import size  
  
In [335]: cvdf.select(size(split(col("Description"), " "))).show(2)  
+-----+  
|size(split(Description, ))|  
+-----+  
|      5|  
|      3|  
+-----+  
only showing top 2 rows
```



## array\_contains

We can also see whether this array contains a value:

```
In [336]: from pyspark.sql.functions import array_contains
In [337]: cvdf.select(array_contains(split(col("Description"), " "), "WHITE")).show(2)
+-----+
|array_contains(split(Description, ), WHITE)|
+-----+
|          true|
|          true|
+-----+
only showing top 2 rows
```



# explode

The **explode** function takes a column that consists of arrays and creates one row (with the rest of the values duplicated) per value in the array.

```
In [338]: from pyspark.sql.functions import split,explode
In [339]: cvdf.withColumn("splitted",split(col("Description"), " "))\
....: .withColumn("exploded", explode(col("splitted"))))\
....: .select("Description","InvoiceNo","exploded").show(3)
+-----+-----+-----+
|      Description|InvoiceNo|exploded|
+-----+-----+-----+
|WHITE HANGING HEA...|    536365|    WHITE|
|WHITE HANGING HEA...|    536365| HANGING|
|WHITE HANGING HEA...|    536365|    HEART|
+-----+-----+-----+
only showing top 3 rows
```



# Maps

Maps are created by using the map function and key-value pairs of columns. We then can select them just like we might select from an array

```
In [340]: from pyspark.sql.functions import create_map  
  
In [341]: cvdf.select(create_map(col("Description"), col("InvoiceNo")).alias("complex_map"))\n...: .show(2)  
+-----+  
| complex_map |  
+-----+  
|[WHITE HANGING HE...|  
|[WHITE METAL LANT...|  
+-----+  
only showing top 2 rows
```

```
In [342]: cvdf.select(create_map(col("Description"), col("InvoiceNo")).alias("complex_map"))\n...: .show(2, False)  
+-----+  
|complex_map |  
+-----+  
|[WHITE HANGING HEART T-LIGHT HOLDER -> 536365]|  
|[WHITE METAL LANTERN -> 536365]|  
+-----+  
only showing top 2 rows
```



# Working with JSON

Spark has some unique support for working with JSON data.

We can operate directly on strings of JSON in Spark and parse from JSON or extract JSON objects.

```
In [89]: data = [{"1": {"f1": "value1", "f2": "value2"}}, {"2": {"f1": "value12"}}]  
In [90]: jsdf = spark.createDataFrame(data, ("key", "jstring"))  
In [91]: jsdf.select(jsdf.key, json_tuple(jsdf.jstring, 'f1', 'f2')).collect()  
Out[91]:  
[Row(key=u'1', c0=u'value1', c1=u'value2'),  
 Row(key=u'2', c0=u'value12', c1=None)]
```

# to\_json



```
In [95]: cvsdf.selectExpr("(InvoiceNo,Description) as myStruct")\n    .select(to_json(col("myStruct"))).show(20, False)\n+-----+\n|structstojson(myStruct)\n+-----+\n| {"InvoiceNo": "536365", "Description": "WHITE HANGING HEART T-LIGHT HOLDER"}\n| {"InvoiceNo": "536365", "Description": "WHITE METAL LANTERN"}\n| {"InvoiceNo": "536365", "Description": "CREAM CUPID HEARTS COAT HANGER"}\n| {"InvoiceNo": "536365", "Description": "KNITTED UNION FLAG HOT WATER BOTTLE"}\n| {"InvoiceNo": "536365", "Description": "RED WOOLLY HOTTIE WHITE HEART."}\n| {"InvoiceNo": "536365", "Description": "SET 7 BABUSHKA NESTING BOXES"}\n| {"InvoiceNo": "536365", "Description": "GLASS STAR FROSTED T-LIGHT HOLDER"}\n| {"InvoiceNo": "536366", "Description": "HAND WARMER UNION JACK"}\n| {"InvoiceNo": "536366", "Description": "HAND WARMER RED POLKA DOT"}\n| {"InvoiceNo": "536367", "Description": "ASSORTED COLOUR BIRD ORNAMENT"}\n| {"InvoiceNo": "536367", "Description": "POPPY'S PLAYHOUSE BEDROOM "}\n| {"InvoiceNo": "536367", "Description": "POPPY'S PLAYHOUSE KITCHEN"}\n| {"InvoiceNo": "536367", "Description": "FELTCRAFT PRINCESS CHARLOTTE DOLL"}
```



# User-Defined Functions



```
In [369]: udfDF = spark.range(5).toDF("num")

In [370]: def pow3(thric_value):
    ...:     return thric_value ** 3
    ...:

In [371]: pow3(2.0)
Out[371]: 8.0

In [372]: pow3(4.0)
Out[372]: 64.0
```



We need to register the functions with Spark so that we can use them on all of our worker machines.

Spark will serialize the function on the driver and transfer it over the network to all executor processes.

*This happens regardless of language.*

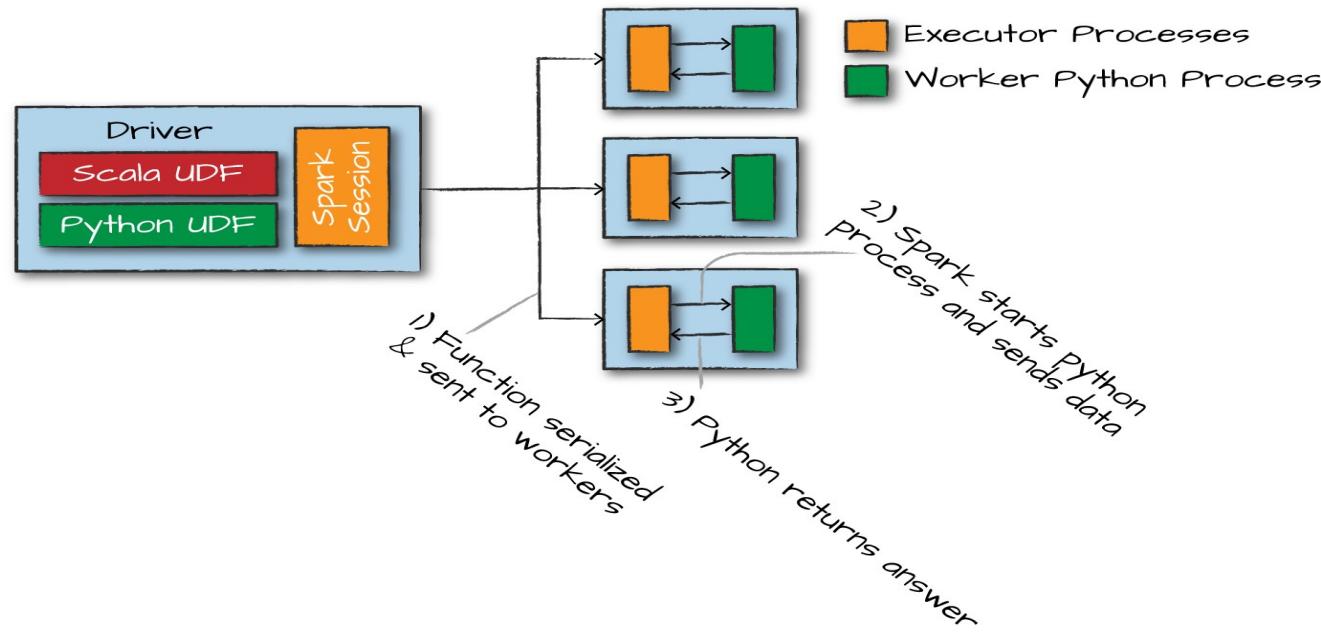
---

*When you use the function, there are essentially two different things that occur. If the function is written in Scala or Java, you can use it within the Java Virtual Machine (JVM).*

*This means that there will be little performance penalty aside from the fact that you can't take advantage of code generation capabilities that Spark has for built-in functions.*



**If the function is written in Python**, Spark starts a Python process on the worker, serializes all of the data to a format that Python can understand (remember, it was in the JVM earlier), executes the function row by row on that data in the Python process, and then finally returns the results of the row operations to the JVM and Spark.





first, we register it:

```
In [105]: udfDF = spark.range(5).toDF("num")

In [106]: def pow3(thric_value):
    return thric_value ** 3
    .....

In [107]: pow3(2.0)
Out[107]: 8.0

In [108]: from pyspark.sql.functions import udf
In [109]: power3udf = udf(pow3)

In [110]: from pyspark.sql.functions import col

In [111]: udfDF.select(power3udf(col("num"))).show(3)
+-----+
|pow3(num)|
+-----+
|        0|
|        1|
|       8|
+-----+
only showing top 3 rows
```



# Aggregations

# What is Aggregation....



Aggregating is the act of collecting something together.

In an aggregation, we specify a key or grouping and an aggregation function that specifies how we should transform one or more columns. This function must produce one result for each group, given multiple input values.

Spark's aggregation capabilities are sophisticated and mature, with a variety of different use cases and possibilities.

Overall :-

We use aggregations to summarize numerical data usually by means of some grouping. This might be a **summation**, a **product**, or simple **counting**. Also, with Spark you can aggregate any kind of value into an array, list, or map

# Preparing DataFrame with data



Let's begin by reading in our *data on purchases*, repartitioning the data to have far fewer partitions (because we know it's a small volume of data stored in a lot of small files), and caching the results for rapid access

```
In [377]: aggDF = spark.read.format("csv")\
....: .option("header","true")\
....: .option("inferSchema","true")\
....: .load("/home/ilg/Documents/sparkdata/data/retail-data/all//*.csv")\
....: .coalesce(5)

In [378]: aggDF.cache()
Out[378]: DataFrame[InvoiceNo: string, StockCode: string, Description: string, Quantity: int, InvoiceDate: string, UnitPrice: double, CustomerID: int, Country: string]

In [379]: aggDF.createOrReplaceTempView("aggDFTable")
```



```
In [380]: aggDF.show()
```

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
536365	85123A	WHITE HANGING HEA...	6	12/1/2010 8:26	2.55	17850	United Kingdom
536365	71053	WHITE METAL LANTERN	6	12/1/2010 8:26	3.39	17850	United Kingdom
536365	84406B	CREAM CUPID HEART...	8	12/1/2010 8:26	2.75	17850	United Kingdom
536365	84029G	KNITTED UNION FLA...	6	12/1/2010 8:26	3.39	17850	United Kingdom
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/2010 8:26	3.39	17850	United Kingdom
536365	22752	SET 7 BABUSHKA NE...	2	12/1/2010 8:26	7.65	17850	United Kingdom
536365	21730	GLASS STAR FROSTE...	6	12/1/2010 8:26	4.25	17850	United Kingdom
536366	22633	HAND WARMER UNION...	6	12/1/2010 8:28	1.85	17850	United Kingdom
536366	22632	HAND WARMER RED P...	6	12/1/2010 8:28	1.85	17850	United Kingdom

```
In [381]: aggDF.count()
```

```
Out[381]: 541909
```

## Count and cache



**count** is actually an action as opposed to a transformation, and so it returns immediately.

We can use count to get an idea of the total size of your dataset but another common pattern is to use it to *cache an entire DataFrame in memory*

```
In [378]: aggDF.cache()
Out[378]: DataFrame[InvoiceNo: string, StockCode: string, Description: string, Quantity: int, InvoiceDate: string, UnitPrice: double, CustomerID: int, Country: string]
```



# Aggregation Functions



# count

In this case, we can do one of two things:

Specify a specific column to count

OR

All the columns by using count(\*) or count(1) to represent that we want to count every row as the literal one.

```
In [382]: from pyspark.sql.functions import count
```

```
In [383]: aggDF.select(count("StockCode")).show()
```

```
+-----+  
| count(StockCode) |  
+-----+  
|      541909 |  
+-----+
```

```
In [384]: aggDF.select(count("*")).show()
```

```
+-----+  
| count(1) |  
+-----+  
|      541909 |  
+-----+
```



## KEEP IN MIND

**NOTE:** There are a number of **gotchas** when it comes to null values and counting.

An instance where performing a `count(*)`, Spark will count null values (including rows containing all nulls).

But, when counting an individual column, Spark will not count the null values.



## countDistinct

Sometimes, the total number is not relevant; rather, it's the number of unique groups that you want. To get this number, you can use the **countDistinct** function.

```
In [385]: from pyspark.sql.functions import countDistinct  
In [386]: aggDF.select(countDistinct("StockCode")).show()  
+-----+  
|count(DISTINCT StockCode)|  
+-----+  
|          4070|  
+-----+
```

## approx\_count\_distinct



**pyspark.sql.functions.approx\_count\_distinct(col, rsd=None)**

Aggregate function: returns a new Column for approximate distinct count of column col.

Parameters

**rsd** – maximum estimation error allowed (default = 0.05).

NOTE:For rsd < 0.01, it is more efficient to use countDistinct()



## approx\_count\_distinct

As we find ourselves working with large datasets and certain times exact distinct count is irrelevant.

There are times when an *approximation* to a certain degree of accuracy will work just fine, and for that, you can use the **approx\_count\_distinct** function

```
In [387]: from pyspark.sql.functions import approx_count_distinct

In [388]: aggDF.select(approx_count_distinct("StockCode", 0.1)).show()
+-----+
|approx_count_distinct(StockCode)|
+-----+
|          3364|
+-----+


In [389]: aggDF.select(approx_count_distinct("UnitPrice", 0.1)).show()
+-----+
|approx_count_distinct(UnitPrice)|
+-----+
|          1694|
+-----+
```



## first and last

We can get the first and last values from a DataFrame by using these two named functions.

This will be based on the rows in the DataFrame, not on the values in the DataFrame

```
In [391]: from pyspark.sql.functions import first,last  
In [392]: aggDF.select(first("StockCode"),last("StockCode")).show()  
+-----+-----+  
|first(StockCode, false)|last(StockCode, false)|  
+-----+-----+  
|          85123A|           22138|
```

# min and max



To extract the minimum and maximum values from a DataFrame, use the min and max functions.

## pyspark.sql.functions.min(col)

Aggregate function: returns the minimum value of the expression in a group.

```
In [393]: from pyspark.sql.functions import min,max  
  
In [394]: aggDF.select(min("Quantity"),max("Quantity")).show()  
+-----+-----+  
|min(Quantity)|max(Quantity)|  
+-----+-----+  
|-80995| 80995|
```

# sum



To add all the values in a row using the sum function

`pyspark.sql.functions.sum(col)`

- Aggregate function: returns the sum of all values in the expression.

```
In [395]: from pyspark.sql.functions import sum  
  
In [396]: aggDF.select(sum("Quantity")).show()  
+-----+  
|sum(Quantity)|  
+-----+  
|      5176450|  
+-----+
```



## sumDistinct

Summing a total, you also can sum a distinct set of values.

### pyspark.sql.functions.sumDistinct(col)

- Aggregate function: returns the sum of distinct values in the expression.

```
In [397]: from pyspark.sql.functions import sumDistinct  
  
In [398]: aggDF.select(sumDistinct("Quantity")).show()  
+-----+  
|sum(DISTINCT Quantity)|  
+-----+  
|          29310|  
+-----+
```

# avg



Spark provides an easier way to get that value via the **avg** or **mean** functions

## pyspark.sql.functions.avg(col)

- Aggregate function: returns the average of the values in a group.

```
In [399]: from pyspark.sql.functions import sum,count,avg,expr  
  
In [400]: aggDF.select(  
....: count("Quantity").alias("total_transactions"),  
....: sum("Quantity").alias("total_purchases"),  
....: avg("Quantity").alias("avg_purchases"),  
....: expr("mean(Quantity)").alias("mean_purchases"))\  
....: .selectExpr(  
....: "total_purchases/total_transactions",  
....: "avg_purchases",  
....: "mean_purchases").show()  
+-----+-----+-----+  
|(total_purchases / total_transactions)|  avg_purchases|  mean_purchases|  
+-----+-----+-----+  
|          9.55224954743324|9.55224954743324|9.55224954743324|  
+-----+-----+-----+
```



# Variance and Standard Deviation

Variance is the average of the squared differences from the mean, and

Standard deviation is the square root of the variance.

## **pyspark.sql.functions.stddev(col)**

- Aggregate function: alias for stddev\_samp.

## **pyspark.sql.functions.stddev\_pop(col)**

- Aggregate function: returns population standard deviation of the expression in a group.

NOTE: Spark has both the formula for the sample standard deviation as well as the formula for the population standard deviation. These are fundamentally different statistical formulae, and we need to differentiate between them.

NOTE: By default, Spark performs the formula for the sample standard deviation or variance if you use the variance or stddev functions.



```
In [401]: from pyspark.sql.functions import var_pop, stddev_pop
In [402]: from pyspark.sql.functions import var_samp, stddev_samp
In [403]: aggDF.select(var_pop("Quantity"),var_samp("Quantity"),
...: stddev_pop("Quantity"), stddev_samp("Quantity")).show()
+-----+-----+-----+-----+
|var_pop(Quantity)|var_samp(Quantity)|stddev_pop(Quantity)|stddev_samp(Quantity)|
+-----+-----+-----+-----+
|47559.30364660879| 47559.39140929848| 218.08095663447733| 218.08115785023355|
+-----+-----+-----+-----+
```

# skewness and kurtosis



Skewness and kurtosis are both measurements of extreme points in your data.

Skewness measures the **asymmetry** of the values in your data around the mean.

kurtosis is a measure of the **tail of data**.

These are both relevant specifically when modeling your data as a probability distribution of a random variable.

```
In [407]: from pyspark.sql.functions import skewness,kurtosis
In [408]: aggDF.select(skewness("Quantity"), kurtosis("Quantity")).show()
+-----+-----+
|skewness(Quantity)|kurtosis(Quantity)|
+-----+-----+
|-0.264075576105298|119768.05495534067|
+-----+-----+
```

# Covariance and Correlation



Functions comparing the interactions of the values in two difference columns together.

Two of these functions are cov and corr, for covariance and correlation.

Correlation measures the **Pearson correlation coefficient**, which is scaled between  $-1$  and  $+1$ .

The covariance is scaled according to the inputs in the data.

```
In [412]: from pyspark.sql.functions import corr,covar_pop,covar_samp
In [413]: aggDF.select(corr("InvoiceNo","Quantity"),covar_samp("InvoiceNo","Quantity"),
.... covar_pop("InvoiceNo","Quantity")).show()
+-----+-----+-----+
|corr(InvoiceNo, Quantity)|covar_samp(InvoiceNo, Quantity)|covar_pop(InvoiceNo, Quantity)|
+-----+-----+-----+
|      4.912186085636837E-4|          1052.7280543912716|        1052.7260778751674|
+-----+-----+-----+
```

# Aggregating to Complex Types



In Spark, we can perform aggregations not just of numerical values using formulas, we can also perform them on complex types.

Instance :

we can collect a list of values present in a given column or only the unique values by collecting to a set.

Later on results could be used to carry out some more programmatic access later on in the pipeline or pass the entire collection in a user-defined function (UDF)

```
In [416]: from pyspark.sql.functions import collect_list,collect_set
In [417]: aggDF.agg(collect_set("Country"),collect_list("Country")).show()
+-----+-----+
|collect_set(Country)|collect_list(Country)|
+-----+-----+
|[Portugal, Italy,...| [United Kingdom, ...|
+-----+-----+
```

# Grouping



To perform calculations based on groups in the data.

This is typically done on **categorical data** for which we group our data on one column and perform some calculations on the other columns that end up in that group.

Let's group by each unique invoice number and get the count of items on that invoice.

We do this grouping in two phases.

- 1) we specify the column(s) on which we would like to group, then
- 2) we specify the aggregation(s).

The first step returns a **RelationalGroupedDataset**, and the second step returns a **DataFrame**.



```
In [418]: aggDF.groupBy("InvoiceNo","CustomerId").count().show()
+-----+-----+-----+
|InvoiceNo|CustomerId|count|
+-----+-----+-----+
| 536846 |      14573 |    76 |
| 537026 |      12395 |    12 |
| 537883 |      14437 |     5 |
| Fax:538068 |      17978 |    12 |
| 538279 |      14952 |     7 |
| 538800 |      16458 |    10 |
| 538942 |      17346 |    12 |
| C539947 |      13854 |     1 |
| Doc:540096 |      13253 |    16 |
| 540530 |      14755 |    27 |
| 541225 |      14099 |    19 |
| 541978 |      13551 |     4 |
| 542093 |      17677 |    16 |
| 543188 |      12567 |    63 |
| 543590 |      17377 |    19 |
| C543757 |      13115 |     1 |
| C544318 |      12989 |     1 |
| 544578 |      12365 |     1 |
| 545165 |      16339 |    20 |
| 545289 |      14732 |    30 |
+-----+-----+-----+
only showing top 20 rows
```

# Grouping with Expressions



```
In [419]: from pyspark.sql.functions import count  
  
In [420]: aggDF.groupBy("InvoiceNo").agg(  
...:     count("Quantity").alias("quan"),  
...:     expr("count(Quantity)")).show()  
+-----+-----+  
|InvoiceNo|quan|count(Quantity)|  
+-----+-----+  
| 536596 |   6 |              6 |  
| 536938 |  14 |             14 |  
| 537252 |   1 |              1 |  
| Fox1 537691 |  20 |             20 |  
| 538041 |   1 |              1 |  
| 538184 |  26 |             26 |  
| 538517 |  53 |             53 |  
| Ur538879 |  19 |             19 |  
| Doc539275 |   6 |              6 |  
| 539630 |  12 |             12 |  
| 540499 |  24 |             24 |  
| 540540 |  22 |             22 |  
| C540850 |   1 |              1 |  
| 540976 |  48 |             48 |  
| 541432 |   4 |              4 |  
| 541518 | 101 |            101 |  
| 541783 |  35 |             35 |  
| 542026 |   9 |              9 |  
| 542375 |   6 |              6 |  
| C542604 |   8 |              8 |  
+-----+-----+  
only showing top 20 rows
```

Rather than passing that function as an expression into a select statement, **we specify it as within agg**. This makes it possible for you to pass-in arbitrary expressions that just need to have some aggregation specified.

we can even do things like alias a column after transforming it for later use in your data flow



# Grouping with Maps

Sometimes, it can be easier to specify our transformations as a series of Maps for which the key is the column, and the value is the aggregation function (as a string) that we would like to perform.

we can reuse multiple column names if you specify them inline.

```
In [424]: aggDF.groupBy("InvoiceNo").agg(expr("avg(Quantity)"),expr("stddev_pop(Quantity)"))\n...: .show()\n+-----+\n|InvoiceNo|      avg(Quantity)|stddev_pop(Quantity)|\n+-----+\n|  536596|        1.5| 1.1180339887498947|\n|  536938|33.142857142857146| 20.698023172885524|\n|  537252|        31.0|          0.0|\n|  537691|        8.15| 5.597097462078001|\n|  538041|        30.0|          0.0|\n|  538184|12.076923076923077| 8.142590198943392|\n|  538517|3.0377358490566038| 2.3946659604837897|\n|  538879|21.157894736842106|11.811070444356483|\n|  539275|        26.0| 12.806248474865697|\n|  539630|20.33333333333332|10.225241100118645|\n|  540499|        3.75| 2.6653642652865788|\n|  540540|2.1363636363636362| 1.0572457590557278|\n|C540850|        -1.0|          0.0|\n|  540976|10.52083333333334| 6.496760677872902|\n|  541432|        12.25| 10.825317547305483|\n|  541518|23.10891089108911|20.550782784878713|\n|  541783|11.314285714285715| 8.467657556242811|\n|  542026|7.6666666666666667| 4.853406592853679|\n|  542375|        8.0| 3.4641016151377544|\n|C542604|        -8.0| 15.173990905493518|\n+-----+\nonly showing top 20 rows
```



# Window Functions

What are Window Functions?

Built-in functions or UDFs, such as substr or round, take values from a single row as input, and they generate a single return value for every input row.

Aggregate functions, such as SUM or MAX, *operate on a group of rows* and calculate a *single return value for every group*.

With above, Specifically, there was no way to both **operate on a group of rows while still returning a single value for every input row**. This limitation makes it hard to conduct various data processing tasks like calculating a moving average, calculating a cumulative sum, or accessing the values of a row appearing before the current row.

So SOLUTION is window function.....

# Window functions



A window function calculates a return value for every input row of a table based on a group of rows, called the **Frame**.

Every input row can have a unique frame associated with it.

This characteristic of window functions makes them more powerful than other functions and allows users to express various data processing tasks that are hard to be expressed without window functions in a concise way.



# Using Window Functions

Spark SQL supports three kinds of window functions:

- 1) Ranking functions
- 2) Analytic functions
- 3) Aggregate functions.

For aggregate functions, users can use any existing aggregate function as a window function.

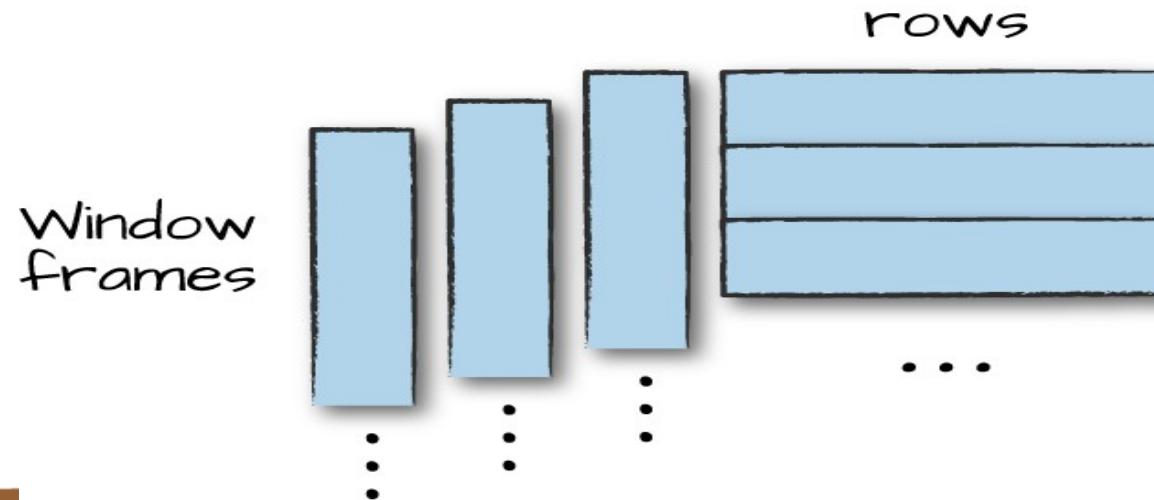
	SQL	DataFrame API
Ranking functions	rank	rank
	dense_rank	denseRank
	percent_rank	percentRank
	ntile	ntile
	row_number	rowNumber
Analytic functions	cume_dist	cumeDist
	first_value	firstValue
	last_value	lastValue
	lag	lag
	lead	lead



A **group-by** takes data, and every row can go only into one grouping.

A window function calculates a return value for every input row of a table based on a group of rows, called a frame. *Each row can fall into one or more frames.*

A common use case is to take a look at a rolling average of some value for which each row represents one day.





To demonstrate, we will add a date column that will convert our invoice date into a column that contains only date information

```
In [425]: from pyspark.sql.functions import col,to_date  
In [426]: dfwithDate = aggDF.withColumn("date",to_date(col("InvoiceDate"),"Mm/d/yyyy H:mm"))  
In [427]: dfwithDate.createOrReplaceTempView("dfwithDate")
```

```
In [428]: dfwithDate.show()
```

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	date
536365	85123A	WHITE HANGING HEA...	6	12/1/2010 8:26	2.55	17850	United Kingdom	2010-01-01
536365	71053	WHITE METAL LANTERN	6	12/1/2010 8:26	3.39	17850	United Kingdom	2010-01-01
536365	84406B	CREAM CUPID HEART...	8	12/1/2010 8:26	2.75	17850	United Kingdom	2010-01-01
536365	84029G	KNITTED UNION FLA...	6	12/1/2010 8:26	3.39	17850	United Kingdom	2010-01-01
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/2010 8:26	3.39	17850	United Kingdom	2010-01-01
536365	22752	SET 7 BABUSHKA NE...	2	12/1/2010 8:26	7.65	17850	United Kingdom	2010-01-01
536365	21730	GLASS STAR FROSTE...	6	12/1/2010 8:26	4.25	17850	United Kingdom	2010-01-01
536366	22633	HAND WARMER UNION...	6	12/1/2010 8:28	1.85	17850	United Kingdom	2010-01-01



The first step to a window function is to create a window specification.

The frame specification (the rowsBetween statement) states which rows will be included in the frame based on its reference to the current input row.

we look at all previous rows up to the current row:

```
In [434]: from pyspark.sql.window import Window
In [435]: from pyspark.sql.functions import desc
In [436]: windowSpec = Window\
    ....: .partitionBy("CustomerId","date")\
    ....: .orderBy(desc("Quantity"))\
    ....: .rowsBetween(Window.unboundedPreceding,Window.CurrentRow)
```

Types of boundaries (two positions and three offsets):

- UNBOUNDED PRECEDING - the first row of the partition
- UNBOUNDED FOLLOWING - the last row of the partition
- CURRENT ROW
- <value> PRECEDING
- <value> FOLLOWING

For Frames



Now we want to use an aggregation function to learn more about each specific customer.

An example

*Might be establishing the maximum purchase quantity over all time ?.*

To answer this, we use the aggregation functions used - by passing a column name or expression.

In addition, we indicate the window specification that defines to which frames of data this function will apply.

We notice that this returns a column (or expressions). We will used this in DataFrame select statement later.

```
In [437]: from pyspark.sql.functions import max  
In [438]: maxPurchaseQuantity = max(col("Quantity")).over(windowSpec)
```

Let's now create purchase quantity rank....



We will create the **purchase quantity rank**.

To do that we use the **dense\_rank** function to determine which date had the maximum purchase quantity for every customer.

*We use dense\_rank as opposed to rank to avoid gaps in the ranking sequence when there are tied values (or in our case, duplicate rows).*

*This also returns a column that we can use in select statements.*

```
In [440]: from pyspark.sql.functions import dense_rank, rank
In [441]: purchaseDenseRank = dense_rank().over(windowSpec)
In [442]: purchaseRank = rank().over(windowSpec)
```



Now we can perform a select to view the calculated window values.

```
from pyspark.sql.functions import col
In [481]: dfwithDate.where("CustomerId IS NOT NULL").orderBy("CustomerId") .select( col("CustomerId"),    col("date"),
...:     col("Quantity"),    purchaseRank.alias("quantityRank"),    purchaseDenseRank.alias("quantityDenseRank"),
...:     maxPurchaseQuantity.alias("maxPurchaseQuantity")).show()
+-----+-----+-----+-----+-----+
|CustomerId|    date|Quantity|quantityRank|quantityDenseRank|maxPurchaseQuantity|
+-----+-----+-----+-----+-----+
| 12346|2011-01-18|    74215|         1|             1|          74215|
| 12346|2011-01-18|   -74215|         2|             2|          74215|
| 12347|2010-12-07|       36|         1|             1|            36|
| 12347|2010-12-07|       30|         2|             2|            36|
| 12347|2010-12-07|       24|         3|             3|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       12|         4|             4|            36|
| 12347|2010-12-07|       6|        17|             5|            36|
| 12347|2010-12-07|       6|        17|             5|            36|
+-----+-----+-----+-----+-----+
only showing top 20 rows
```

# Rollup



A rollup is a **multidimensional aggregation** that performs a variety of group-by style calculations for us.

## `rollup(*cols)`

- Create a multi-dimensional rollup for the current DataFrame using the specified columns, so we can run aggregation on them.



# Rollup

Let's create a rollup that looks across time (with our new Date column) and space (with the Country column) and creates a new DataFrame that includes the grand total over all dates, the grand total for each date in the DataFrame, and the subtotal for each country on each date in the DataFrame

```
In [507]: rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))\n...: .selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")\n...: .orderBy("Date")\n\nIn [508]: rolledUpDF.show()\n+-----+-----+-----+\n|     Date|   Country|total_quantity|\n+-----+-----+-----+\n|    null|      null|      5176450|\n|2010-12-01|      null|       26814|\n|2010-12-01| Australia|        107|\n|2010-12-01|   France|        449|\n|2010-12-01|United Kingdom|      23949|\n|2010-12-01|      EIRE|        243|\n|2010-12-01|   Norway|      1852|\n|2010-12-01|Netherlands|         97|\n|2010-12-01|   Germany|        117|\n|2010-12-02|   Germany|      146|\n|2010-12-02|      EIRE|         4|\n|2010-12-02|      null|      21023|\n|2010-12-02|United Kingdom|      20873|\n|2010-12-03|   Poland|        140|\n|2010-12-03|   France|      239|\n|2010-12-03|   Germany|      170|\n|2010-12-03| Portugal|         65|\n|2010-12-03|   Spain|      400|\n|2010-12-03|Switzerland|       110|\n|2010-12-03|   Belgium|      528|\n+-----+-----+-----+\nonly showing top 20 rows
```



Now where you see the null values is where we will find the grand totals.

A null in both rollup columns specifies the grand total across both of those columns:

```
In [510]: rolledUpDF.where("Country is null").show()
+-----+-----+
| Date | Country | total_quantity |
+-----+-----+
| null | null | 5176450 |
| 2010-12-01 | null | 26814 |
| 2010-12-02 | null | 21023 |
| 2010-12-03 | null | 14830 |
| 2010-12-05 | null | 16395 |
| 2010-12-06 | null | 21419 |
| 2010-12-07 | null | 24995 |
| 2010-12-08 | null | 22741 |
| 2010-12-09 | null | 18431 |
| 2010-12-10 | null | 20297 |
| 2010-12-12 | null | 10565 |
| 2010-12-13 | null | 17623 |
| 2010-12-14 | null | 20098 |
| 2010-12-15 | null | 18229 |
| 2010-12-16 | null | 29632 |
| 2010-12-17 | null | 16069 |
| 2010-12-19 | null | 3795 |
| 2010-12-20 | null | 14965 |
| 2010-12-21 | null | 15467 |
| 2010-12-22 | null | 3192 |
+-----+-----+
only showing top 20 rows
```

```
In [509]: rolledUpDF.where("Country IS NULL").show()
```

# Cube



A cube takes the rollup to a level deeper.

Rather than treating elements hierarchically, **a cube does the same thing across all dimensions.** This means that it won't just go by date over the entire time period, but also the country.

- To pose this as a question again, can you make a table that includes the following?
  - The total across all dates and countries
  - The total for each date across all countries
  - The total for each country on each date
  - The total for each country across all dates

# Cube



## `cube(*cols)`

- Create a multi-dimensional cube for the current DataFrame using the specified columns, so we can run aggregation on them.



# Cube

```
In [511]: from pyspark.sql.functions import sum

In [512]: dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))\n...: .select("Date", "Country", "sum(Quantity)").orderBy("Date").show()
+-----+-----+
|Date|          Country|sum(Quantity)|
+-----+-----+
|null|          Japan|      25218|
|null|        Australia|     83653|
|null|       Portugal|     16180|
|null|        Germany|    117448|
|null|           RSA|      352|
|null|      Hong Kong|      4769|
|null|         Cyprus|      6317|
|null|  Unspecified|      3300|
|null|United Arab Emirates|      982|
|null|           null|    5176450|
|null|    Channel Islands|      9479|
|null|        Finland|     10666|
|null|        Denmark|     8188|
|null|         Spain|     26824|
|null|        Lebanon|      386|
|null|European Community|      497|
|null|        Singapore|     5234|
|null|         Norway|    19247|
|null|Czech Republic|      592|
|null|          USA|     1034|
+-----+
only showing top 20 rows
```

# Pivot



Pivots make it possible for you to convert a row into a column.

In our current data we have a Country column. *With a pivot, we can aggregate according to some function for each of those given countries and display them in an easy-to-query way.*

```
pivoted = dfWithDate.groupBy("date").pivot("Country").sum()
```

To Query

```
pivoted.where("date > '2011-12-05'").select("date" ,``USA_sum(Quantity)`").show()
```



# Joins

## Why Joins



Spark applications are going to bring together a large number of different datasets.

Therefore, joins are an essential part of nearly all Spark workloads.

Spark's ability to talk to different data means that you gain the ability to tap into a variety of data sources across your company.

# Join Expressions



A **join** brings together two sets of data, the left and the right, by comparing the value of one or more keys of the left and right and evaluating the result of a join expression that determines whether Spark should bring together the left set of data with the right set of data.

## equi-join

Compares whether the specified keys in your left and right datasets are equal. If they are equal, Spark will combine the left and right datasets.

The opposite is true for keys that do not match;

Spark discards the rows that do not have matching keys.



# Join Types

Inner joins (keep rows with keys that **exist** in the left and right datasets)

Outer joins (keep rows with keys in *either* the left or right datasets)

Left outer joins (keep rows with **keys** in the **left dataset**)

Right outer joins (keep rows with **keys** in the **right dataset**)

Left semi joins (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)

Left anti joins (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)

Natural joins (perform a join by **implicitly matching the columns** between the two datasets with the same names)

Cross (or Cartesian) joins (**match every row in the left dataset with every row in the right dataset**)

# Sample Data tables / sets



```
In [541]: person = spark.createDataFrame([
...:     (0, "Michael Corleone", 0, [100]),
...:     (1, "Sonny Corleone", 1, [500, 250, 100]),
...:     (2, "Fredo Corleone", 1, [250, 100]))]\n...:     .toDF("id", "name", "graduate_program", "spark_status")\n\nIn [542]: graduateProgram = spark.createDataFrame([
...:     (0, "Masters", "School of Information", "UC Berkeley"),
...:     (2, "Masters", "EECS", "UC Berkeley"),
...:     (1, "Ph.D.", "EECS", "UC Berkeley"))]\n...:     .toDF("id", "degree", "department", "school")\n\nIn [543]: sparkStatus = spark.createDataFrame([
...:     (500, "Vice President"),
...:     (250, "PMC Member"),
...:     (100, "Contributor"))]\n...:     .toDF("id", "status")\n\nIn [544]: person.createOrReplaceTempView("person")\n\nIn [545]: graduateProgram.createOrReplaceTempView("graduateProgram")\n\nIn [546]: sparkStatus.createOrReplaceTempView("sparkStatus")
```

Register  
the tables



# Inner Joins

Inner joins evaluate the keys in both of the DataFrames or tables and include (and join together) only the rows that evaluate to true.

*Keys that do not exist in both DataFrames will not show in the resulting DataFrame.*

*Inner joins are the default join, so we just need to specify our left DataFrame and join the right in the JOIN expression*

```
In [134]: joinExpression = person["graduate_program"] == graduateProgram['id']
```

```
In [136]: person.join(graduateProgram, joinExpression).show()
```

id	name	graduate_program	spark_status	id	degree	department	school
0	Michael Corleone	0	[100]	0	Masters	School of Informa...	UC Berkeley
1	Sonny Corleone	1	[500, 250, 100]	1	Ph.D.	EECS	UC Berkeley
2	Fredo Corleone	1	[250, 100]	1	Ph.D.	EECS	UC Berkeley



# Specify join type

We can also specify this explicitly by passing in a third parameter, the `joinType`:

```
In [137]: joinType = 'inner'

In [138]: person.join(graduateProgram, joinExpression, joinType).show()
+-----+-----+-----+-----+-----+-----+-----+
| id |      name|graduate_program| spark_status| id| degree|      department|      school|
+-----+-----+-----+-----+-----+-----+-----+
| 0 |Michael Corleone|          0|           [100]|  0|Masters|School of Informa...|UC Berkeley|
| 1 | Sonny Corleone|          1|[500, 250, 100]|  1| Ph.D.|                  EECS|UC Berkeley|
| 2 | Fredo Corleone|          1|[250, 100]|  1| Ph.D.|                  EECS|UC Berkeley|
+-----+-----+-----+-----+-----+-----+-----+
```



# Outer Joins

Outer joins evaluate the keys in both of the DataFrames or tables and includes (and joins together) the rows that evaluate to true or false.

If there is no equivalent row in either the left or right DataFrame, Spark will insert null.

```
In [139]: ojoinType = 'outer'

In [140]: person.join(graduateProgram, joinExpression, ojoinType).show()
+-----+-----+-----+-----+-----+-----+-----+
| id |      name|graduate_program| spark_status| id| degree|      department|      school|
+-----+-----+-----+-----+-----+-----+-----+
| 0 |Michael Corleone|          0|       [100]|  0|Masters|School of Informa...|UC Berkeley|
| 1 |Sonny Corleone|          1|[500, 250, 100]|  1| Ph.D.|                  EECS|UC Berkeley|
| 2 |Fredo Corleone|          1|[250, 100]|  1| Ph.D.|                  EECS|UC Berkeley|
| null|        null|      null|      null|  2|Masters|                  EECS|UC Berkeley|
+-----+-----+-----+-----+-----+-----+-----+
```



# Left Outer Joins

Left outer joins evaluate the keys in both of the DataFrames or tables and includes all rows from the left DataFrame as well as any rows in the right DataFrame that have a match in the left DataFrame.

If there is no equivalent row in the right DataFrame, Spark will insert null

```
In [145]: joinTy = 'left_outer'

In [146]: person.join(graduateProgram, joinExpression, joinTy).show()
+-----+-----+-----+-----+-----+-----+-----+
| id |      name|graduate_program| spark_status| id| degree|      department| school|
+-----+-----+-----+-----+-----+-----+-----+
| 0 |Michael Corleone|          0|     [100]|  0|Masters|School of Informa...|UC Berkeley|
| 1 | Sonny Corleone|          1|[500, 250, 100]|  1| Ph.D.|           EECS|UC Berkeley|
| 2 | Fredo Corleone|          1|[250, 100]|  1| Ph.D.|           EECS|UC Berkeley|
+-----+-----+-----+-----+-----+-----+-----+
```



## Right Outer Joins

Right outer joins evaluate the keys in both of the DataFrames or tables and includes all rows from the right DataFrame as well as any rows in the left DataFrame that have a match in the right DataFrame.

If there is no equivalent row in the left DataFrame, Spark will insert null

```
In [148]: joinTy = 'right_outer'

In [149]: person.join(graduateProgram, joinExpression, joinTy).show()
+-----+-----+-----+-----+-----+-----+-----+
| id | name | graduate_program | spark_status | id | degree | department | school |
+-----+-----+-----+-----+-----+-----+-----+
| 0 | Michael Corleone | 0 | [100] | 0 | Masters | School of Informa... | UC Berkeley |
| 1 | Sonny Corleone | 1 | [500, 250, 100] | 1 | Ph.D. | EECS | UC Berkeley |
| 2 | Fredo Corleone | 1 | [250, 100] | 1 | Ph.D. | EECS | UC Berkeley |
| null | null | null | null | 2 | Masters | EECS | UC Berkeley |
+-----+-----+-----+-----+-----+-----+-----+
```

# Left Semi Joins



Semi joins are a bit of a departure from the other joins. *They do not actually include any values from the right DataFrame.*

They only **compare values** to see if the **value exists in the second DataFrame**. If the value does exist, those rows will be kept in the result, even if there are duplicate keys in the left DataFrame.

*Think of left semi joins as filters on a DataFrame, as opposed to the function of a conventional join*

## left\_semi



```
In [150]: joinTy = 'left_semi'

In [151]: person.join(graduateProgram, joinExpression, joinTy).show()
+-----+-----+-----+
| id |      name|graduate_program| spark_status|
+-----+-----+-----+
| 0 |Michael Corleone|          0|      [100]|
| 1 |   Sonny Corleone|          1|[500, 250, 100]|
| 2 |   Fredo Corleone|          1|[250, 100]|
+-----+-----+-----+
```



```
In [154]: gradProgram2 = graduateProgram.union(spark.createDataFrame([(0,"Masters","Duplicated Row","Duplicated School")]))
```

```
In [155]: gradProgram2.createOrReplaceTempView("gradProgram2")
```

```
In [156]: gradProgram2.join(person,joinExpression,joinTy).show()
```

id	degree	department	school
0	Masters	School of Informa...	UC Berkeley
0	Masters	Duplicated Row	Duplicated School
1	Ph.D.	EECS	UC Berkeley



## Left Anti Joins

Left anti joins are the opposite of left semi joins.

Like left semi joins, they do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. they keep only the values that do not have a corresponding key in the second DataFrame.

Think of anti joins as a NOT IN SQL-style filter

```
In [160]: joinTy = 'left_anti'

In [161]: graduateProgram.join(person,joinExpression,joinTy).show()
+-----+
| id|degree|department|      school|
+---+---+-----+-----+
|  2|Masters|      EECS|UC Berkeley|
+---+---+-----+-----+
```

# Natural Joins



Natural joins make implicit guesses at the columns on which you would like to join. It finds matching columns and returns the results. Left, right, and outer natural joins are all supported.



## Cross (Cartesian) Joins

Cross-joins in simplest terms are inner joins that do not specify a predicate.

Cross joins will join every single row in the left DataFrame to every single row in the right DataFrame.

This will cause an **absolute explosion** in the number of rows contained in the resulting DataFrame.

In case

If you have 1,000 rows in each DataFrame, the cross-join of these will result in 1,000,000 ( $1,000 \times 1,000$ ) rows.

For this reason, you must very explicitly state that you want a cross-join by using the cross join keyword

# Cross join



They're dangerous!

```
In [162]: joinTy = 'cross'

In [163]: graduateProgram.join(person,joinExpression,joinTy).show()
+-----+-----+-----+-----+-----+-----+
| id| degree|      department|      school|  id|      name|graduate_program|    spark_status|
+-----+-----+-----+-----+-----+-----+
|  0|Masters|School of Informa...|UC Berkeley|  0|Michael Corleone|          0|      [100]|
|  1|  Ph.D.|                  EECS|UC Berkeley|  1|  Sonny Corleone|          1|[500, 250, 100]|
|  1|  Ph.D.|                  EECS|UC Berkeley|  2|  Fredo Corleone|          1|[250, 100]|
+-----+-----+-----+-----+-----+-----+
```



## Data Sources

# Data Sources



Spark has six “**core**” data sources and hundreds of external data sources written by the community.

The ability to read and write from all different kinds of data sources and for the community to create its own contributions is arguably one of Spark’s greatest strengths.

- **CSV**
- **JSON**
- **Parquet**
- **ORC**
- **JDBC/ODBC connections**
- **Plain-text files**

# Community data sources



Spark has numerous community-created data sources.

Cassandra

- HBase
- MongoDB
- AWS Redshift
- XML

And many, many others

# Structure of the Data Sources API



## Read API Structure

**core structure for reading data is as follows:**

`DataFrameReader.format(...).option("key", "value").schema(...).load()`

- **format** is optional because by default Spark will use the Parquet format.
- **option** allows you to set key-value configurations to parametrize how you will read data.
- **schema** is optional if the data source provides a schema or if you intend to use schema inference.

# Basics of Reading Data



Foundation for reading data in Spark is the **DataFrameReader**.

We access this through the **SparkSession** via the read attribute:

## **spark.read**

After we have a DataFrame reader, we specify several values:

- The format
- The schema
- The read mode
- A series of options

```
spark.read.format("csv")
    .option("mode", "FAILFAST")
    .option("inferSchema", "true")
    .option("path", "path/to/file(s)")
    .schema(someSchema)
    .load()
```



# READ MODES

Reading data from an external source naturally entails encountering malformed data, especially when working with only semi-structured data sources.

Read modes specify what will happen when Spark does come across malformed records.

## Spark's read modes

Read mode	Description
Default	permissive
dropMalformed	Drops the row that contains malformed records
failFast	Fails immediately upon encountering malformed records

# Write API Structure



Core structure for writing data is as follows:

```
DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy( ...  
.).save()
```

# Basics of Writing Data



We have the **DataFrameWriter**. Because we always need to write out some given data source, we access the **DataFrameWriter** on a **per-DataFrame** basis via the `write` attribute.

After we have a DataFrameWriter, we specify three values:

- 1) Format,
- 2) A series of options,
- 3) Save mode.

At a minimum, you must supply a path.

```
dataframe.write.format("csv")
    .option("mode", "OVERWRITE")
    .option("dateFormat", "yyyy-MM-dd")
    .option("path", "path/to/file(s)")
    .save()
```



# SAVE MODES

Save modes specify what will happen if Spark finds data at the specified location.

## Spark's save modes

Save mode	Description
append	Appends the output files to the list of files that already exist at that location
overwrite	Will completely overwrite any data that already exists there
errorIfExists	Throws an error and fails the write if data or files already exist at the specified location
ignore	If data or files exist at the location, do nothing with the current DataFrame

Default

# CSV Files



CSV stands for **comma-separated values**.

This is a common text file format in which each line represents a single record, and commas separate each field within a record. CSV files, while seeming well structured, are actually one of the **trickiest file formats** we will encounter because not many assumptions can be made in production scenarios about what they contain or how they are structured.

For this reason, the CSV reader has a *large number of options*.

These options give you the ability to work around issues like certain characters needing to be escaped—for example, commas inside of columns when the file is also comma-delimited or null values labeled in an unconventional way.

# CSV Options



Kindly refer for vast available options

<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions>



## Reading CSV Files

To read a CSV file, we must first create a **DataFrameReader** for that specific format. Here, we specify the format to be CSV:

**spark.read.format("csv")**

Like :-

**df = spark.read.csv('python/test\_support/sql/ages.csv')**



# Writing CSV Files

```
In [550]: csvfile = spark.read.format("csv")\
....: .option("header","true")\
....: .option("mode","FAILFAST")\
....: .option("inferSchema","true")\
....: .load("/home/ilg/Documents/sparkdata/data/retail-data/by-day/2011-07-11.csv")  
  
In [551]: csvfile.write.format("csv").mode("overwrite").option("sep","\t")\
....: .save("/tmp/ourTSVfile.tsv")
```

we can take our CSV file and write it out as a TSV file quite easily:

This actually reflects the number of partitions in our DataFrame at the time we write it out. If we were to repartition our data before then, we would end up with a different number of files.

```
ilg@D10Spark:~/Documents/sparkdata/data/retail-data/by-day$ less /tmp/ourTSVfile.tsv/  
part-00000-21846795-e27b-46f2-9bec-d86880cc6af6-c000.csv  
.part-00000-21846795-e27b-46f2-9bec-d86880cc6af6-c000.csvcrc  
_SUCCESS  
_.SUCCESS.crc
```

# JSON Files



In Spark, when we refer to JSON files, we refer to line-delimited JSON files.

Refer the options;

<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions>

The **line-delimited** versus multiline trade-off is controlled by a single option: **multiLine**. When you set this option to true, you can read an entire file as one json object and Spark will go through the work of parsing that into a DataFrame.

***Line-delimited JSON is actually a much more stable format because it allows you to append to a file with a new record (rather than having to read in an entire file and then write it out), which is what we recommend that you use. Another key reason for the popularity of line-delimited JSON is because JSON objects have structure, and JavaScript (on which JSON is based) has at least basic types.***

# Read a json file



Reading a line-delimited JSON file varies only in the format and the options that we specify.

**spark.read.format("json")**

# Reading JSON Files



Let's look at an example of reading a JSON file and compare the options that we're seeing:

```
In [552]: spark.read.format("json").option("mode","FAILFAST")\
....: .option("inferSchema", "true")\
....: .load("/home/ilg/Documents/sparkdata/data/flight-data/json/2010-summary.json").show(5)
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
| United States| Romania| 1|
| United States| Ireland| 264|
| United States| India| 69|
| Egypt| United States| 24|
| Equatorial Guinea| United States| 1|
+-----+-----+-----+
only showing top 5 rows
```



# Writing JSON Files

```
In [553]: csvfile.write.format("json").mode("overwrite")\n      .... .save("/tmp/ourJSON-file.json")
```

```
ilg@D10Spark:~/Documents/sparkdata/data/flight-data/json$ less /tmp/ourJSON-file.json/\npart-00000-7dd1224e-06c5-4cb7-93d3-42dcef45c0ba-c000.json\n.part-00000-7dd1224e-06c5-4cb7-93d3-42dcef45c0ba-c000.json.crc\n  SUCCESS\n:_SUCCESS.crc
```

# Parquet Files



Parquet is an open source column-oriented data store that provides a variety of storage optimizations, especially for analytics workloads.

It provides **columnar compression**, which saves storage space and allows for *reading individual columns instead of entire files*.

It is a file format that works exceptionally well with **Apache Spark** and is in fact the **default file format**.

We recommend **writing data out to Parquet for long-term storage** because

- 1) Reading from a Parquet file will always be more efficient than JSON or CSV.
- 2) Advantage of Parquet is that it supports complex types.

This means that if your column is an array (which would fail with a CSV file, for example), map, or struct, we'll still be able to read and write that file without issue.

# Reading Parquet Files



Parquet has very few options because it enforces its own schema when storing data.

```
In [554]: spark.read.format("parquet")\
...: .load("/home/ilg/Documents/sparkdata/data/flight-data/parquet/2010-summary.parquet").show(5)
+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+
| United States| Romania|    1|
| United States| Ireland|  264|
| United States| India|   69|
|    Egypt| United States|   24|
|Equatorial Guinea| United States|    1|
+-----+-----+
only showing top 5 rows
```

# Parquet data source options



Read/Write	Key	Potential Values	Default	Description
Write	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or snappy	None	Declares what compression codec Spark should use to read or write the file.
Read	mergeSchema	true, false	Value of the configuration spark.sql.parquet.mergeSchema	You can incrementally add columns to newly written Parquet files in the same table/folder. Use this option to enable or disable this feature.



# Writing Parquet Files

We simply specify the location for the file.

```
ilg@D10Spark:~/Documents/sparkdata/data/flight-data/parquet/2010-summary.parquet$ less /tmp/our-PARquest-file.parquet/
part-00000-b17596ab-c59b-451f-a96c-0d74e921b395-c000.snappy.parquet
.part-00000-b17596ab-c59b-451f-a96c-0d74e921b395-c000.snappy.parquet.crc
SUCCESS
._SUCCESS.crc
```

# ORC Files



ORC Optimized Row Columnar is a self-describing, type-aware columnar file format designed for Hadoop workloads.

It is **optimized for large streaming reads**, but with integrated support for *finding required rows quickly*.

ORC actually has no options for reading in data because Spark understands the file format quite well.

An often-asked question is:

**What is the difference between ORC and Parquet?**

For the most part, they're quite similar; the fundamental difference is that Parquet is further optimized for use with Spark, whereas ORC is further optimized for Hive.

# Reading Orc Files



```
In [556]: spark.read.format("orc").load("/home/ilg/Documents/sparkdata/data/flight-data/orc/2010-summary.orc").show(5)
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
| United States | Romania | 1 |
| United States | Ireland | 264 |
| United States | India | 69 |
| Egypt | United States | 24 |
| Equatorial Guinea | United States | 1 |
+-----+-----+-----+
only showing top 5 rows
```



# Writing Orc Files

```
In [557]: csvfile.write.format("orc").mode("overwrite")\n      ...: .save("/tmp/our-ORC-file.orc")
```

```
ilg@D10Spark:~/Documents/sparkdata/data/flight-data/orc/2010-summary.orc$ less /tmp/our-ORC-file.orc/\npart-00000-46f8b14b-1df6-4c5e-a476-8bedcaa4e34e-c000.snappy.orc\n.part-00000-46f8b14b-1df6-4c5e-a476-8bedcaa4e34e-c000.snappy.orc.crc\n.SUCCESS\n._SUCCESS.crc
```



# SQL Databases

# First things...



To read and write from these databases, we need to do two things:

- 1) include the Java Database Connectivity (JDBC) driver for your particular database on the spark classpath.
- 2) provide the proper JAR for the driver itself.

For Instance,

To be able to read and write from PostgreSQL, we might run something like this:

```
./bin/spark-shell \
--driver-class-path postgresql-9.4.1207.jar \
--jars postgresql-9.4.1207.jar
```

<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions>

# Reading from SQL Databases



When it comes to reading a file, SQL databases are no different from the other data sources.

**For sqlite package**

```
pyspark --packages "org.xerial:sqlite-jdbc:3.16.1"
```



```
In [1]: driver = "org.sqlite.JDBC"
In [2]: path = "/home/ilg/Documents/sparkdata/data/flight-data/jdbc/my-sqlite.db"
In [3]: url = "jdbc:sqlite:" + path
In [4]: tablename = "flight_info"
In [5]: dbDataFrame = spark.read.format("jdbc").option("url", url) \
...: .option("dbtable", tablename).option("driver", driver).load()
In [6]: dbDataFrame.show()
+-----+-----+-----+
| DEST_COUNTRY_NAME | ORIGIN_COUNTRY_NAME | count |
+-----+-----+-----+
| United States | Romania | 1 |
| United States | Ireland | 264 |
| United States | India | 69 |
| Egypt | United States | 24 |
| Equatorial Guinea | United States | 1 |
| United States | Singapore | 25 |
| United States | Grenada | 54 |
| Costa Rica | United States | 477 |
| Senegal | United States | 29 |
| United States | Marshall Islands | 44 |
| Guyana | United States | 17 |
| United States | Sint Maarten | 53 |
| Malta | United States | 1 |
| Bolivia | United States | 46 |
| Anguilla | United States | 21 |
| Turks and Caicos ... | United States | 136 |
| United States | Afghanistan | 2 |
| Saint Vincent and... | United States | 1 |
| Italy | United States | 390 |
| United States | Russia | 156 |
+-----+-----+-----+
only showing top 20 rows
```

# Conn to mysql



PostgreSQL, require more configuration parameters.

Let's perform the same read that we just performed, except using mysql this time:

```
pyspark --packages mysql:mysql-connector-java:5.1.38
```

```
In [1]: spark.read.format("jdbc").options(  
....: url ="jdbc:mysql://192.168.1.16/publications",  
....: driver="com.mysql.jdbc.Driver",  
....: dbtable="classics",  
....: user="ilg",  
....: password="ilg007su"  
....: ).load().take(10)
```

Dut[2]:

```
[Row(Author=u'Tolstoy', Title=u'War and Peace', Year=u'1865', Type=u'fiction'),  
 Row(Author=u'Palkhivala', Title=u'Indian Constitution', Year=u'1950', Type=u'non-fiction')]
```



# Writing to mysql DB

```
ilg@u16Spark:~$ pyspark --packages "mysql:mysql-connector-java:5.1.41"

In [1]: matrix = [('Dr.Suman','Law of DV',2016,'non-fiction'),('Jai','Python Book',2019,'Non-fiction')]

In [2]: jmydf = spark.createDataFrame(matrix, ['Author','Title', 'Year','Type'])

In [3]: url = 'jdbc:mysql://localhost/publications?user=ilg&password=ilg007su'

In [4]: properties = {'driver': 'com.mysql.jdbc.Driver'}

In [5]: jmydf.selectExpr('Author', 'Title', 'Year', 'Type') \
    .write.jdbc(url, table='classics', mode='append', properties=properties)
```

```
mysql> select * from classics;
+-----+-----+-----+-----+
| Author | Title          | Year | Type   |
+-----+-----+-----+-----+
| Tolstoy | War and Peace | 1865 | fiction |
| Palkhivala | Indian Constitution | 1950 | non-fiction |
| Mark Twain | The adventures of Tom | 1845 | fiction |
| Dr.Suman | Law of DV | 2016 | non-fiction |
| Jai | Python Book | 2019 | Non-fiction |
| Dr.Suman | Law of DV | 2016 | non-fiction |
| Jai | Python Book | 2019 | Non-fiction |
+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

## Text Files



Spark also allows you to read in plain-text files.

Each line in the file becomes a record in the DataFrame. It is then up to you to transform it accordingly.



# Reading Text Files

With `textFile`, partitioned directory names are ignored.

To read and write text files according to partitions, we should use `text`, which respects partitioning on reading and writing

## `text(paths, wholertext=False, lineSep=None)`

- Loads text files and returns a DataFrame whose schema starts with a string column named “value”, and followed by partitioned columns if there are any.
- By default, each line in the text file is a new row in the resulting DataFrame.

## Parameters

- **paths** – string, or list of strings, for input path(s).
- **wholertext** – if true, read each file from input path(s) as a single row.
- **lineSep** – defines the line separator that should be used for parsing. If None is set, it covers all \r, \r\n and \n.



```
In [591]: spark.read.text("/home/ilg/Documents/sparkdata/data/flight-data/csv/2010-summary.csv")\
....: .selectExpr("split(value, ',') as rows").show()
+-----+
|      rows|
+-----+
|[DEST_COUNTRY_NAM...
|[United States, R...
|[United States, I...
|[United States, I...
|[Egypt, United St...
|[Equatorial Guine...
|[United States, S...
|[United States, G...
|[Costa Rica, Unit...
|[Senegal, United ...
|[United States, M...
|[Guyana, United S...
|[United States, S...
|[Malta, United St...
|[Bolivia, United ...
|[Anguilla, United...
|[Turks and Caicos...
|[United States, A...
|[Saint Vincent an...
|[Italy, United St...
+-----+
only showing top 20 rows
```



# Writing Text Files

When you write a text file, we need to be sure to have only one string column; otherwise, the write will fail:

```
In [594]: csvfile.select("Description").write.text("/tmp/our-Simple_text.txt")
```

*If we perform some partitioning when performing our write (later slides), we can write more columns. However, those columns will manifest as directories in the folder to which you're writing out to, instead of columns on every single file.*

```
In [600]: csvfile.limit(10).select("Description","UnitPrice").write.partitionBy("UnitPrice")\n...: .text("/tmp/file-part.csv")
```

```
ilg@D10Spark:~/Documents/sparkdata/data/flight-data/csv$ ls -l /tmp/file-part.csv/\n total 0\n -rw-r--r-- 1 ilg ilg 0 Sep 10 03:01 _SUCCESS\n drwxr-xr-x 2 ilg ilg 139 Sep 10 03:01 'UnitPrice=1.25'\n drwxr-xr-x 2 ilg ilg 139 Sep 10 03:01 'UnitPrice=12.75'\n drwxr-xr-x 2 ilg ilg 139 Sep 10 03:01 'UnitPrice=1.65'\n drwxr-xr-x 2 ilg ilg 139 Sep 10 03:01 'UnitPrice=2.95'
```



## Advanced I/O Concepts

# Bucketing & Partitioning



We control specific data layout by controlling two things: bucketing and partitioning.

## **Splittable File Types and Compression**

Certain file formats are fundamentally “splittable.” This can improve speed because it makes it possible for Spark to avoid reading an entire file, and access only the parts of the file necessary to satisfy your query.

If you’re using something like Hadoop Distributed File System (HDFS), splitting a file can provide further optimization if that file spans multiple blocks.

*We recommend Parquet with gzip compression.*



## Reading Data in Parallel

Multiple executors cannot read from the same file at the same time necessarily, but they can read different files at the same time.

SO

when you read from a folder with multiple files in it, each one of those files will become a partition in your DataFrame and be read in by available executors in parallel (with the remaining queueing up behind the others)



## Writing Data in Parallel

The **number of files** or data **written** is **dependent on the number of partitions** the DataFrame has at the time you write out the data.

By default, **one file is written per partition of the data.**

*This means that although we specify a “file,” it’s actually a number of files within a folder, with the name of the specified file, with one file per each partition that is written.*

```
csvFile.repartition(5).write.format("csv").save("/tmp/multiple.csv")
```

will end up with five files inside of that folder.



## PARTITIONING

**Partitioning is a tool that allows you to control what data is stored** (and where) as you write it.

When you write a file to a partitioned directory (or table), we basically encode a column as a folder.

What this allows you to do is **skip lots of data when you go to read it in later**, allowing you to read in only the data relevant to your problem instead of having to scan the complete dataset. These are supported for all file-based data sources

```
# in Python
csvFile.limit(10).write.mode("overwrite").partitionBy("DEST_COUNTRY_NAME")\
    .save("/tmp/partitioned-files.parquet")
```



## BUCKETING

Bucketing is another file organization approach with which you can control the data that is specifically written to each file. This can **help avoid shuffles** later when you go to read the data because *data with the same bucket ID will all be grouped together into one physical partition.*

This means that the **data is prepartitioned** according to *how you expect to use that data later on*, meaning you can **avoid expensive shuffles** when joining or aggregating.

### NOTE:

Rather than partitioning on a specific column (which might write out a ton of directories), *it's probably worthwhile to explore bucketing the data instead. This will create a certain number of files and organize our data into those “buckets”*



## Writing Complex Types

Spark has a variety of different internal types. Although Spark can work with all of these types, not every single type works well with every data file format.

For instance, *CSV files do not support complex types, whereas Parquet and ORC do.*



## Managing File Size

Spark 2.2 introduced a new method for controlling file sizes in a more automatic way.

Previously we have seen, *the number of output files is a derivative of the number of partitions we had at write time* (and the partitioning columns we selected).

A tool in order to **limit output file sizes** so that you can target an optimum file size. we can use the **maxRecordsPerFile** option and specify a number of your choosing. This allows you to better control file sizes by controlling the number of records that are written to each file.

```
df.write.option("maxRecordsPerFile", 5000),
```

Spark will ensure that files will contain at most 5,000 records.



# Spark SQL

# Spark SQL



In Spark SQL we can run SQL queries against views or tables organized into databases.

We also can use system functions or define user functions and analyze query plans in order to optimize their workloads. *This integrates directly into the DataFrame and Dataset API.*

# Big Data and SQL: Apache Hive



Before Spark's rise, Hive was the de facto big data SQL access layer. Originally developed at Facebook, Hive became an incredibly popular tool across industry for performing SQL operations on big data. In many ways it helped propel Hadoop into different industries because analysts could run SQL queries. Although Spark began as a general processing engine with Resilient Distributed Datasets (RDDs), a large cohort of users now use Spark SQL.

# Big Data and SQL: Spark SQL



With the release of Spark 2.0, its authors created a superset of Hive's support, writing a native SQL parser that supports both ANSI-SQL as well as HiveQL queries.

## USE CASE

<https://engineering.fb.com/core-data/apache-spark-scale-a-60-tb-production-use-case/>

## NOTE

Spark SQL is intended to operate as an online analytic processing (OLAP) database, not an online transaction processing (OLTP) database.

This means that it is not intended to perform extremely low-latency queries.



## Spark's Relationship to Hive

Spark SQL has a great relationship with Hive because it can connect to Hive metastores.

The Hive metastore is the way in which Hive maintains table information for use across sessions. With Spark SQL, we can connect to your Hive metastore and access table metadata to reduce file listing when accessing information.

*This is popular for users who are **migrating** from a legacy Hadoop environment and beginning to run all their workloads using Spark.*



## How to Run Spark SQL Queries

Spark provides several interfaces to execute SQL queries.

### Spark SQL CLI

Spark SQL CLI is a convenient tool with which you can make basic Spark SQL queries in local mode from the command line.

```
./bin/spark-sql
```

We configure Hive by placing your **hive-site.xml, core-site.xml, and hdfs-site.xml files in conf/**.

For a complete list of all available options, you can run

```
./bin/spark-sql --help.
```



## Spark's Programmatic SQL Interface

We can also execute SQL in an ad hoc manner via any of Spark's language APIs.

We can do this via the method **sql** on the **SparkSession** object.

This returns a DataFrame.

```
In [602]: spark.sql("select 1 + 1").show()
```

(1 + 1)
2

The command `spark.sql("SELECT 1 + 1")` returns a DataFrame that we can then evaluate programmatically.

# SQL, Dataframes Interopere



We can completely interopere between SQL and DataFrames, as you see fit.

For instance,

we can create a DataFrame, manipulate it with SQL, and then manipulate it again as a DataFrame.

*It's a powerful abstraction that you will likely find yourself using quite a bit*

```
In [607]: spark.read.json("/home/ilg/Documents/sparkdata/data/flight-data/json/2015-summary.json")\
....: .createOrReplaceTempView("sql_view")

In [608]: spark.sql("""
....: select DEST_COUNTRY_NAME,sum(count)
....: from sql_view group by DEST_COUNTRY_NAME
....: """)\
....: .where("DEST_COUNTRY_NAME like 'S%'").where(`sum(count)` > 10")\
....: .count()

Out[608]: 12
```

# Catalog



Catalog is an abstraction for the *storage of metadata about the data stored in your tables as well as other helpful things like databases, tables, functions, and views*.

The catalog is available in the `org.apache.spark.sql.catalog`.

Catalog package and contains a number of helpful functions for doing things like listing tables, databases, and functions.

# Tables



To do anything useful with Spark SQL, we first need to define tables.

**Tables are logically equivalent to a DataFrame** in that they are a structure of data against which we run commands.

We can **join tables**, **filter them**, **aggregate them**, and perform different manipulations that we saw.

Difference between tables and DataFrames is ?

We define DataFrames in the scope of a programming language, whereas we define tables within a database



## Spark-Managed Tables

Concept of Managed and Unmanaged tables.

Tables store two important pieces of information.

- 1) Data within the tables
- 2) Data about the tables; that is, the metadata.

We can have Spark manage the metadata for a set of files as well as for the data.

*When you define a table from files on disk, we are defining an unmanaged table.*

*When you use **saveAsTable** on a **DataFrame**, we are creating a managed table for which Spark will track of all of the relevant information.*

## Creating Tables



We can create tables from a variety of sources.

Spark has the capability of reusing the entire Data Source API within SQL.

This means that you do not need to define a table and then load data into it;  
Spark lets you create one on the fly.

We can even specify all sorts of sophisticated options when you read in a file.

# Datasets



Datasets are a strictly Java Virtual Machine (JVM) language feature that work only with Scala and Java. Using Datasets, we can define the object that each row in your Dataset will consist of.

In Scala, this will be a case class object that essentially defines a schema that you can use, and in Java, you will define a Java Bean.



## Low-Level APIs

# Resilient Distributed Datasets (RDDs)

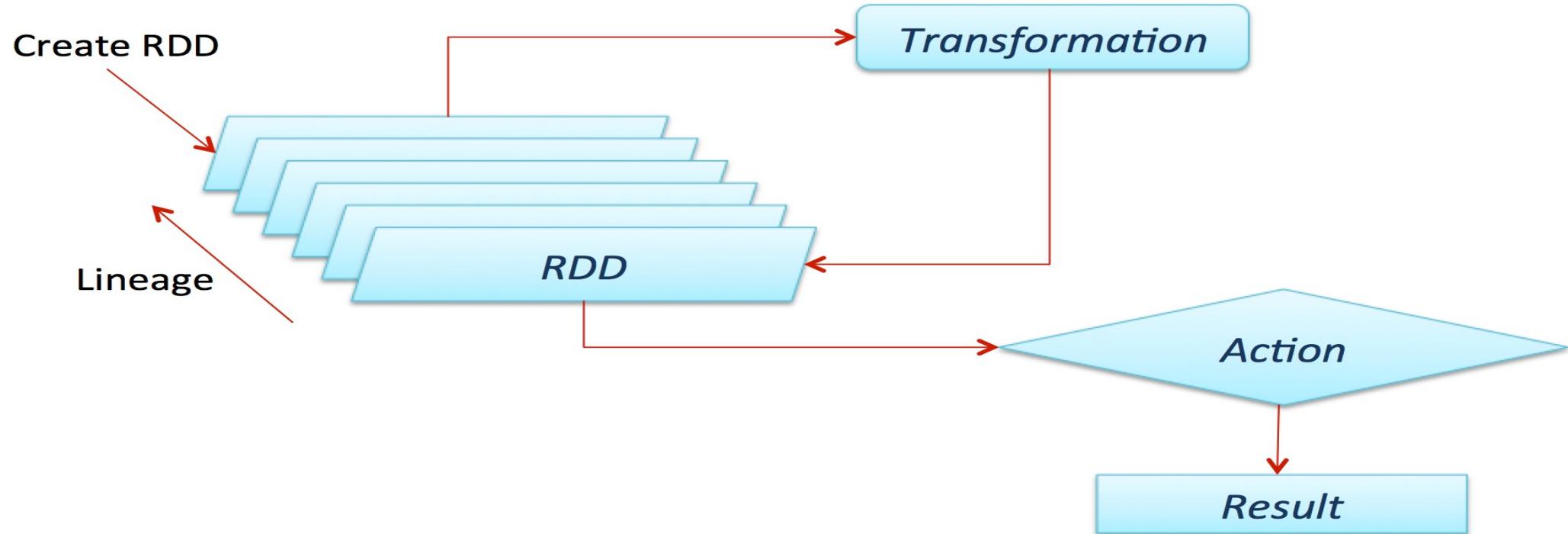


Structured APIs are the heavily favoured in all scenarios.

But, there are times when higher-level manipulation will not meet the business or engineering problem you are trying to solve.

For those cases:

We might need to use **Spark's lower-level APIs**, specifically the **Resilient Distributed Dataset (RDD)**, the **SparkContext**, and *distributed shared variables like accumulators and broadcast variables.*



# What Are the Low-Level APIs?



There are two sets of low-level APIs:

- 1) There is one for manipulating distributed data (RDDs).
- 2) For distributing and manipulating distributed shared variables (broadcast variables and accumulators).

# When to Use the Low-Level APIs?



Three situations:

- 1) We need some functionality that you cannot find in the higher-level APIs;

For Instance:

- *if you need very tight control over physical data placement across the cluster.*

- 2) We need to maintain some legacy codebase written using RDDs.

- 3) We need to do some custom shared variable manipulation.

# How to Use the Low-Level APIs?



A **SparkContext** is the entry point for low-level API functionality.

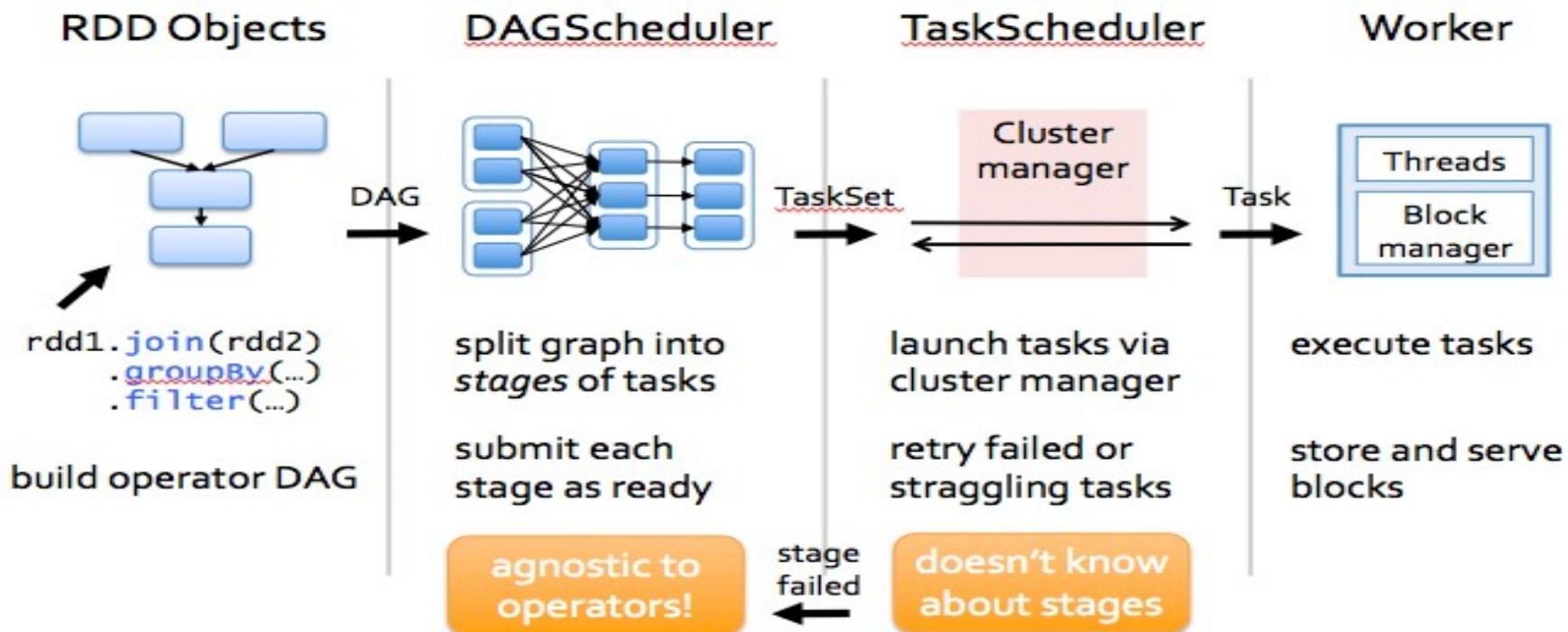
We access it through the **SparkSession**, which is the tool you use to perform computation across a Spark cluster.

We can access a SparkContext via the following call:

**spark.sparkContext**

```
In [612]: spark.sparkContext  
Out[612]: <SparkContext master=local[*] appName=PySparkShell>
```

# About RDDs





Virtually all Spark code you run, whether DataFrames or Datasets, compiles down to an RDD.

*An RDD represents an immutable, partitioned collection of records that can be operated on in parallel.*



RDDs give you complete control because every record in an RDD is a just a Java or Python object.

We can store anything you want in these objects, in any format you want.  
This gives you great power in data manipulation.



# Creating RDDs

## Interoperating Between DataFrames, Datasets, and RDDs

One of the easiest ways to get RDDs is from an existing DataFrame or Dataset.

Converting these to an RDD is simple: just use the **rdd method** on any of these data types.

Because Python doesn't have Datasets—it has only DataFrames—you will get an RDD of type Row:

```
In [613]: spark.range(10).rdd
Out[613]: MapPartitionsRDD[1566] at javaToPython at NativeMethodAccessorImpl.java:0

In [614]: spark.range(10).toDF("id").rdd.map(lambda row:row[0])
Out[614]: PythonRDD[1572] at RDD at PythonRDD.scala:53
```



To operate on this data, you will need to convert this Row object to the correct data type or extract values out of it. This is now an RDD of **type Row**



## toDF

To create dataset or dataframe from and RDD.

Use **toDF** method on the RDD.

```
In [6]: spark.range(10).rdd.toDF()  
Out[6]: DataFrame[id: bigint]
```

This command creates an RDD of type Row.

This row is the internal Catalyst format that Spark uses to represent data in the Structured APIs.

*This functionality makes it possible for you to jump between the Structured and low-level APIs as it suits your use case.*

# From a Local Collection



To create an RDD from a collection, we will need to use the **parallelize** method on a **SparkContext** (within a SparkSession).

This turns a single node collection into a parallel collection.

When creating this parallel collection, we can also explicitly state the number of partitions into which we would like to distribute this array.

```
In [23]: myColl = "I am learning spark. It's fun to work. I am using it to solve big data problem of businesses"\n.split(" ")  
In [24]: words = spark.sparkContext.parallelize(myColl,2)  
In [25]: words.setName('myWords')  
Out[25]: myWords ParallelCollectionRDD[25] at parallelize at PythonRDD.scala:195  
In [26]: words.name()  
Out[26]: u'myWords'
```



The **sc.parallelize()** method is the SparkContext's parallelize method to create a parallelized collection.

This allows Spark to distribute the data across multiple nodes, instead of depending on a single node to process the data

```
In [7]: myRDD = sc.parallelize([('Mike Ross',24),('Harvey',28),('Rachel',22),('Jessica',35)])  
In [8]: myRDD.take(4)  
Out[8]: [('Mike Ross', 24), ('Harvey', 28), ('Rachel', 22), ('Jessica', 35)]  
In [9]: myRDD.take(5)  
Out[9]: [('Mike Ross', 24), ('Harvey', 28), ('Rachel', 22), ('Jessica', 35)]
```



# From Data Sources

We can also read data as RDDs using sparkContext.

read a text file line by line

This creates an RDD for which each record in the RDD represents a line in that text file or files.

```
In [14]: spark.sparkContext.textFile("/home/ilg/txt/NewtonTOEinstein.txt")
Out[14]: /home/ilg/txt/NewtonTOEinstein.txt MapPartitionsRDD[6] at textFile at NativeMethodAccessorImpl.java:0

In [15]: spark.sparkContext.wholeTextFiles("/home/ilg/txt/NewtonTOEinstein.txt")
Out[15]: org.apache.spark.api.java.JavaPairRDD@6d334930 }
```

```
In [16]: spark.sparkContext.wholeTextFiles("/home/ilg/txt/AdventuresOfTom.txt")
Out[16]: org.apache.spark.api.java.JavaPairRDD@17568247 }
```

We can read in data for which *each text file should become a single record*.

In this RDD, the name of the file is the first object and the value of the text file is the second string object.

# Manipulating RDDs



## Transformations

We specify transformations on one RDD to create another.

In doing so, we define an RDD as a dependency to another along with some manipulation of the data contained in that RDD.

## Distinct

A distinct method call on an RDD removes duplicates from the RDD:

```
In [27]: words.distinct().count()  
Out[27]: 15
```



## Filter

Filtering is equivalent to creating a SQL-like where clause. You can look through our records in the RDD and see which ones match some predicate function.

we filter the RDD to keep only the words that begin with the letter “s”

```
In [30]: def startsWiths(indivi):
    return indivi.startswith("s")
....:

In [31]: words.filter(lambda word: startsWiths(word)).collect()
Out[31]: ["spark.It's", 'solve']
```



## Map

We specify a function that returns the value that you want, given the correct input. We then apply that, record by record.

we'll map the current word to the word, its starting letter, and whether the word begins with "s."

```
In [32]: word2 = words.map(lambda word: (word,word[0], word.startswith("s")))

In [33]: word2.filter(lambda record: record[2]).take(5)
Out[33]: [("spark.It's", 's', True), ('solve', 's', True)]
```



## FLATMAP

### **flatMap(f, preservesPartitioning=False)**

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

We might want to take our set of words and flatMap it into a set of characters.

```
In [40]: myColl = "I am learning spark.It's fun to work. I am using it to solve big data problem of businesses"\n.split(" ")
```

```
In [41]: words.flatMap(lambda word:list(word)).take(10)\nOut[41]: ['I', 'a', 'm', 'l', 'e', 'a', 'r', 'n', 'i', 'n']
```



## Sort

To sort an RDD you must use the `sortBy` method, and just like any other RDD operation, you do this by specifying a function to extract a value from the objects in your RDDs and then sort based on that.

```
In [42]: words.sortBy(lambda word: len(word) * -1).take(2)
Out[42]: ["spark.It's", 'businesses']
```

```
In [43]: tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
In [44]: sc.parallelize(tmp).sortBy(lambda x: x[0]).collect()
Out[44]: [('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
```



## Random Splits

We can also randomly split an RDD into an Array of RDDs by using the `randomSplit` method, which accepts an Array of weights and a random seed.

### `randomSplit(weights, seed=None)[source]`

Randomly splits this RDD with the provided weights.

#### Parameters

- `weights` – weights for splits, will be normalized if they don't sum to 1
- `seed` – random seed

```
In [62]: fiftyfiftysplit = words.randomSplit([0.5,0.5])

In [63]: fiftyfiftysplit[0]
Out[63]: PythonRDD[59] at RDD at PythonRDD.scala:53

In [64]: fiftyfiftysplit[1]
Out[64]: PythonRDD[60] at RDD at PythonRDD.scala:53

In [65]: fiftyfiftysplit1 = words.randomSplit([1,3])

In [66]: fiftyfiftysplit1[0]
Out[66]: PythonRDD[61] at RDD at PythonRDD.scala:53

In [67]: fiftyfiftysplit1[1]
Out[67]: PythonRDD[62] at RDD at PythonRDD.scala:53
```



```
In [69]: rdd = sc.parallelize(range(500), 1)
In [70]: rdd1, rdd2 = rdd.randomSplit([2, 3], 17)
In [71]: len(rdd1.collect() + rdd2.collect())
Out[71]: 500
In [72]: 150 < rdd1.count() < 250
Out[72]: True
In [73]: 250 < rdd2.count() < 350
Out[73]: True
```



## **Actions**

we specify actions to kick off our specified transformations.

Actions either collect data to the driver or write to an external data source.



## Reduce

### reduce(f)

Reduces the elements of this RDD using the specified commutative and associative binary operator. Currently reduces partitions locally.

We can use the *reduce method to specify a function to “reduce” an RDD of any kind of value to one value.*

For instance,

Given a *set of numbers, we can reduce this to its sum* by specifying a function that takes as input two values and reduces them into one.

```
In [74]: spark.sparkContext.parallelize(range(1,21)).reduce(lambda x,y: x + y)
Out[74]: 210
```

```
In [76]: from operator import add
```

```
In [77]: sc.parallelize([1,2,3,4,5]).reduce(add)
Out[77]: 15
```



```
myColl = "I am learning spark.It's fun to work. I am using it to solve big data problem of businesses"\n.split(" ")  
words = spark.sparkContext.parallelize(myColl,2)
```

```
In [80]: def wordLengthReducer(leftword,rightword):\n    if len(leftword) > len(rightword):\n        return leftword\n    else:\n        return rightword\n    ....:
```

```
In [81]: words.reduce(wordLengthReducer)\nOut[81]: 'businesses'
```



## Count

### count()

Return the number of elements in this RDD.

```
In [83]: sc.parallelize([2,3,4]).count()  
Out[83]: 3
```

```
In [84]: sc.parallelize([2,3,4,5,6]).count()  
Out[84]: 5
```



## COUNTAPPROX

This is an approximation of the count method , but it must execute within a timeout (and can return incomplete results if it exceeds the timeout).

`countApprox(timeout, confidence=0.95)`

Approximate version of count() that returns a potentially incomplete result within a timeout, even if not all tasks have finished.

```
In [85]: rdd = sc.parallelize(range(1000), 10)
In [86]: rdd.countApprox(1000, 1.0)
Out[86]: 1000

In [87]: rdd.countApprox(1000, 1.0)
KeyboardInterrupt

In [87]: rdd = sc.parallelize(range(100000), 10)
In [88]: rdd.countApprox(100000, 1.0)
Out[88]: 100000
```

***confidence is the probability*** that the error bounds of the result will contain the true value.

Confidence must be in the range [0,1], or an exception will be thrown



## COUNTAPPROXDISTINCT

**countApproxDistinct(relativeSD=0.05)**

Return approximate number of **distinct** elements in the RDD.

*The algorithm used is based on streamlib's implementation of "HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm"*

### Parameters

**relativeSD** – Relative accuracy.

Smaller values create counters that require more space.

It must be greater than 0.000017

```
In [101]: words.countApproxDistinct(0.05)
Out[101]: 15L
```



```
In [93]: n = sc.parallelize(range(1000)).map(str).countApproxDistinct()  
In [94]: 00 < n < 1100  
Out[94]: True  
  
In [95]: n = sc.parallelize([i % 20 for i in range(1000)]).countApproxDistinct()  
In [96]: 6 < n < 24  
Out[96]: True
```



## COUNTBYVALUE

### countByValue()

Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.

```
In [102]: words.countByValue()
Out[102]: defaultdict(<type 'int'>, {'work.': 1, 'I': 2, 'big': 1, 'am': 2, 'it': 1, 'to': 2, 'solve': 1, 'learning': 1, 'of': 1, 'fun': 1, 'using': 1, 'problem': 1, "spark.It's": 1, 'data': 1, 'businesses': 1})
```

```
In [103]: sorted(sc.parallelize([1, 2, 1, 2, 2], 2).countByValue().items())
Out[103]: [(1, 2), (2, 3)]
```



## First

The first method returns the first value in the dataset.

### first()

- Return the first element in this RDD.

```
In [105]: sc.parallelize([2, 3, 4]).first()  
Out[105]: 2
```

```
In [104]: words.first()  
Out[104]: 'I'
```



## max and min

max and min return the maximum values and minimum values, respectively:

key – A function used to generate key for comparing

```
In [109]: rdd = sc.parallelize([2.0, 5.0, 43.0, 10.0])
In [110]: rdd.min()
Out[110]: 2.0
In [111]: rdd.min(key=str)
Out[111]: 10.0
```

```
In [112]: rdd.max()
Out[112]: 43.0
In [113]: rdd.max(key=str)
Out[113]: 5.0
```



## **take**

**take** and its derivative methods take a number of values from your RDD.

This works by first scanning one partition and then using the results from that partition to estimate the number of additional partitions needed to satisfy the limit.

**takeOrdered, takeSample, and top.**

Are some variations

**takeOrdered(num, key=None)**

Get the N elements from an RDD ordered in ascending order or as specified by the optional key function.

**takeSample(withReplacement, num, seed=None)**

Return a fixed-size sampled subset of this RDD.

**top(num, key=None)[source]**

Get the top N elements from an RDD.



```
In [119]: sc.parallelize([10, 4, 2, 12, 3]).top(1)
Out[119]: [12]
```

```
In [120]: sc.parallelize([10, 4, 2, 12, 3]).top(2)
Out[120]: [12, 10]
```

```
In [121]: sc.parallelize([10, 4, 2, 12, 3]).top(3, key=str)
Out[121]: [4, 3, 2]
```

```
In [122]: rdd = sc.parallelize(range(0, 10))
```

```
In [123]: len(rdd.takeSample(True, 20, 1))
Out[123]: 20
```

```
In [124]: len(rdd.takeSample(False, 5, 2))
Out[124]: 5
```

```
In [125]: len(rdd.takeSample(False, 15, 3))
Out[125]: 10
```

```
In [126]: sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7]).takeOrdered(6)
Out[126]: [1, 2, 3, 4, 5, 6]
```

```
In [127]: sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7], 2).takeOrdered(6, key=lambda x: -x)
Out[127]: [10, 9, 7, 6, 5, 4]
```

# Saving Files



## saveAsTextFile

To save to a text file, we just specify a path and optionally a compression codec

```
In [128]: words.saveAsTextFile("file:/tmp/wbook")
```

```
ilg@ul6Spark:~$ ls -l /tmp/wbook/
total 8
-rw-r--r-- 1 ilg ilg 43 Sep 11 10:27 part-00000
-rw-r--r-- 1 ilg ilg 49 Sep 11 10:27 part-00001
-rw-r--r-- 1 ilg ilg  0 Sep 11 10:27 _SUCCESS
```



# Checkpointing

**Checkpointing** is the act of saving an RDD to disk so that future references to this RDD point to those intermediate partitions on disk rather than recomputing the RDD from its original source.

This is similar to caching except that it's not stored in memory, only disk. This can be helpful when performing iterative computation

```
In [132]: spark.sparkContext.setCheckpointDir("/tmp/checkpointing")
In [133]: words.checkpoint()
```

```
ilg@u16Spark:~$ ls -l /tmp/checkpointing/
total 4
drwxrwxr-x 2 ilg_ilg 4096 Sep 11 10:46 f036c703-3f7d-4698-b7b0-8f73ce861dfa
```

Now, when we reference this RDD, it will derive from the checkpoint instead of the source data. This can be a helpful optimization



# Pipe RDDs to System Commands

With pipe, we can return an RDD created by piping elements to a forked external process.

*Return an RDD created by piping elements to a forked external process*

*The resulting RDD is computed by executing the given process once per partition.*

All elements of each input partition are written to a process's stdin as lines of input separated by a newline.

The resulting partition consists of the process's stdout output, with each line of stdout resulting in one element of the output partition.

A process is invoked even for empty partitions.

```
In [134]: words.pipe("wc -l").collect()  
Out[134]: [u'9', u'9']
```

```
In [135]: sc.parallelize(['1', '2', '', '3']).pipe('cat').collect()  
Out[135]: [u'1', u'2', u'', u'3']
```



## mapPartitions

Returns a new RDD by applying a function to each partition of the wrapped RDD, where tasks are launched together in a barrier stage.

creates the value “1” for every partition in our data, and the sum of the following expression will count the number of partitions we have

```
In [136]: words.mapPartitions(lambda part: [1]).sum()  
Out[136]: 2
```

# glom



**glom** function takes every partition in your dataset and converts them to arrays.

This can be useful if you're going to collect the data to the driver and want to have an array for each partition.

Beware if it's large conversion, can crash driver.

```
In [137]: spark.sparkContext.parallelize(["Hello", "World"], 2).glom().collect()  
Out[137]: [['Hello'], ['World']]
```

# Advanced RDDs



DATA

```
In [11]: mywrdCol = "Spark,It's fun to learn. I will be solving big data problems.Business need to use it".split(" ")  
In [12]: words = spark.sparkContext.parallelize(mywrdCol,2)  
In [13]: words.count()  
Out[13]: 15
```

## Key-Value Basics (Key-Value RDDs)



There are many methods on RDDs that require you to put your data in a key–value format.

Whenever you see **ByKey** in a method name, it means that we can perform this only on a **PairRDD** type.

The easiest way is to just map over your current RDD to a basic key–value structure. This means having two values in each record of your RDD:

```
In [14]: words.map(lambda word: (word.lower(),1))
Out[14]: PythonRDD[7] at RDD at PythonRDD.scala:53
```



## KeyBy

We can also use the **keyBy** function to achieve the same result by specifying a function that creates the key from your current value.

In this case, we are keying by the first letter in the word.

Spark then keeps the record as the value for the keyed RDD:

```
In [9]: keyword = words.keyBy(lambda word: word.lower()[0])
```

```
In [10]: keyword.collect()
Out[10]:
[('i', 'I'),
 ('a', 'am'),
 ('l', 'learning'),
 ('s', "spark.It's"),
 ('f', 'fun'),
 ('t', 'to'),
 ('w', 'work.'),
 ('i', 'I'),
 ('a', 'am'),
 ('u', 'using'),
 ('i', 'it'),
 ('t', 'to'),
 ('s', 'solve'),
 ('b', 'big'),
 ('d', 'data'),
 ('p', 'problem'),
 ('o', 'of'),
 ('b', 'businesses')]
```



# Mapping over Values

## mapValues(f)

Pass each value in the key-value pair RDD through a **map function** without changing the keys; this also retains the original RDD's partitioning.

```
In [12]: keyword.mapValues(lambda word: word.upper()).collect()
Out[12]:
[('i', 'I'),
 ('a', 'AM'),
 ('l', 'LEARNING'),
 ('s', "SPARK. IT'S"),
 ('f', 'FUN'),
 ('t', 'TO'),
 ('w', 'WORK.'),
 ('i', 'I'),
 ('a', 'AM'),
 ('u', 'USING'),
 ('i', 'IT'),
 ('t', 'TO'),
 ('s', 'SOLVE'),
 ('b', 'BIG'),
 ('d', 'DATA'),
 ('p', 'PROBLEM'),
 ('o', 'OF'),
 ('b', 'BUSINESSES')]
```



```
In [17]: x = sc.parallelize([('a', ["apple", "banana", "lemon"]), ("b", ["grapes"]),(c,['guava'])])  
In [18]: def f(x):  
    return len(x)  
....:  
In [19]: x.mapValues(f).collect()  
Out[19]: [('a', 3), ('b', 1), ('c', 1)]  
In [20]: x = sc.parallelize([('a', ["apple", "banana", "lemon"]), ("b", ["grapes"]),(c,['guava','kiwi'])])  
In [21]: def f(x):  
    return len(x)  
....:  
In [22]: x.mapValues(f).collect()  
Out[22]: [('a', 3), ('b', 1), ('c', 2)]
```

# Extracting Keys and Values



When we are in the key–value pair format, we can also extract the specific keys or values by using the following methods

```
In [23]: keyword.keys().collect()  
Out[23]:  
['i',  
'a',  
'l',  
's',  
'f',  
't',  
't',  
'w',  
'i',  
'a',  
'u',  
'i',  
't',  
's',  
'b',  
'd',  
'p',  
'o',  
'b']
```

```
In [25]: keyword.values().collect()  
Out[25]:  
['I',  
'am',  
'learning',  
"spark.It's",  
'fun',  
'to',  
'work.',  
'I',  
'am',  
'using',  
'it',  
'to',  
'solve',  
'big',  
'data',  
'problem',  
'of',  
'businesses']
```



## lookup

look up the result for a particular key.

lookup(key)

Return the list of values in the RDD for key key.

This operation is done efficiently if the RDD has a known partitioner by only searching the partition that the key maps to.

```
In [26]: keyword.lookup('S')
Out[26]: []

In [27]: keyword.lookup('s')
Out[27]: ["spark.It's", 'solve']

In [28]: keyword.lookup('i')
Out[28]: ['I', 'I', 'it']

In [29]: keyword.lookup('I')
Out[29]: []
```

```
In [38]: l = range(1000)
In [39]: rdd = sc.parallelize(zip(l,l),10)
In [40]: sorted.lookup(412)
Out[40]: [412]

In [41]: sorted = rdd.sortByKey()
In [42]: sorted.lookup(412)
Out[42]: [412]

In [43]: sorted.lookup(99)
Out[43]: [99]
```



# sampleByKey

## sampleByKey(withReplacement, fractions, seed=None)

Return a subset of this RDD sampled by key (via stratified sampling).

Create a sample of this RDD using variable sampling rates for different keys as specified by fractions, a key to sampling rate map.

```
In [144]: import random
In [145]: distinctChars = words.flatMap(lambda word: list(word.lower())).distinct()\
....:     .collect()
In [146]: sampleMap = dict(map(lambda c: (c, random.random()), distinctChars))
In [147]: words.map(lambda word: (word.lower()[0], word))\
....:     .sampleByKey(True, sampleMap, 6).collect()
Out[147]:
[('a', 'am'),
 ('a', 'am'),
 ('l', 'learning'),
 ('t', 'to'),
 ('w', 'work.'),
 ('i', 'I'),
 ('o', 'of')]
```



# Aggregations

We can perform aggregations on plain RDDs or on PairRDDs, depending on the method that you are using.

```
In [148]: chars = words.flatMap(lambda word: word.lower())
In [149]: KVcharacters = chars.map(lambda letter: (letter, 1))
In [150]: def maxFunc(left, right):
....:     return max(left, right)
....: def addFunc(left, right):
....:     return left + right
....: nums = sc.parallelize(range(1,31), 5)
....:
In [151]: KVcharacters.countByKey()
Out[151]: defaultdict(<type 'int'>, {'a': 6, 'b': 3, 'e': 5, ' ': 1, 'g': 3, 'f': 2,
: 6, 'n': 5, 'p': 2, 's': 8, 'r': 4, 'u': 3, 't': 5, 'w': 1, 'v': 1, '.': 2, 'd': 1})
```

## countByKey

We can count the number of elements for each key, collecting the results to a local Map.



## groupBy(f, numPartitions=None, partitionFunc=<function portable\_hash>)

Return an RDD of grouped items.

```
In [1]: rdd = sc.parallelize([1, 1, 2, 3, 5, 8])
In [2]: result = rdd.groupBy(lambda x: x % 2).collect()
In [3]: sorted([(x, sorted(y)) for (x, y) in result])
Out[3]: [(0, [2, 8]), (1, [1, 1, 3, 5])]
```



**reduceByKey(func, numPartitions=None, partitionFunc=<function portable\_hash>)**

*Merge the values for each key using an associative and commutative reduce function.*

This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a “combiner” in MapReduce.

Output will be partitioned with numPartitions partitions, or the default parallelism level if numPartitions is not specified.

```
In [6]: from operator import add  
In [7]: rdd = sc.parallelize([('a', 1), ('b', 1), ('a', 1)])  
In [8]: sorted(rdd.reduceByKey(add).collect())  
Out[8]: [('a', 2), ('b', 1)]
```



# Controlling Partitions

With RDDs, we have control over how data is *exactly physically distributed across the cluster.*

## coalesce

coalesce effectively collapses partitions on the same worker in **order to avoid a shuffle of the data** when repartitioning.

For instance,

Our words RDD is currently two partitions, we can collapse that to one partition by using coalesce without bringing about a shuffle of the data:

```
In [9]: myColl = "I am learning Spark.It is fun filled.I am going to solve bigdata business problems with spark"\n...: .split(" ")  
In [10]: words = spark.sparkContext.parallelize(myColl,2)  
In [11]: words.count()  
Out[11]: 16  
In [12]: words.coalesce(1).getNumPartitions()  
Out[12]: 1
```



## Repartition

The repartition operation allows you to repartition your data up or down but performs a shuffle across nodes in the process.

*Increasing the number of partitions can increase the level of parallelism when operating in map- and filter-type operations:*

```
In [13]: words.repartition(10)
Out[13]: MapPartitionsRDD[19] at coalesce at NativeMethodAccessorImpl.java:0
```

Internally, this uses a shuffle to redistribute data.

If you are **decreasing** the number of partitions in this RDD, consider using coalesce, which can avoid performing a shuffle.

```
In [14]: rdd = sc.parallelize([1,2,3,4,5,6,7],4)
In [15]: sorted(rdd.glom().collect())
Out[15]: [[1], [2, 3], [4, 5], [6, 7]]
```

Return an RDD created by coalescing all elements within each partition into a list.

```
In [17]: len(rdd.repartition(2).glom().collect())
Out[17]: 2
In [18]: len(rdd.repartition(10).glom().collect())
Out[18]: 10
```



## repartitionAndSortWithinPartitions

**repartitionAndSortWithinPartitions(numPartitions=None, partitionFunc=<function portable\_hash>, ascending=True, keyfunc=<function RDD.<lambda>>)**

Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys.

```
In [19]: rdd = sc.parallelize([(0, 5), (3, 8), (2, 6), (0, 8), (3, 8), (1, 3)])  
In [20]: rdd2 = rdd.repartitionAndSortWithinPartitions(2, lambda x: x % 2, True)  
In [21]: rdd2.glom().collect()  
Out[21]: [[(0, 5), (0, 8), (2, 6)], [(1, 3), (3, 8), (3, 8)]]
```



## Life Cycle of a Spark Application (Inside Spark)

# SparkSession



The first step of any Spark Application is creating a `SparkSession`.

In many interactive modes, this is done for you, but in an application, you must do it manually.

## Creating a `SparkSession` in Python

```
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.master("local").appName("Word Count")\  
    .config("spark.some.config.option", "some-value")\  
    .getOrCreate()
```



# SPARKCONTEXT

A SparkContext object within the SparkSession represents the [connection to the Spark cluster.](#)

*This class is how you communicate with some of Spark's lower-level APIs, such as RDDs.*

Main entry point for Spark functionality.

A SparkContext represents the connection to a Spark cluster, and can be used to create RDD and broadcast variables on that cluster.

```
from pyspark import SparkContext
```

```
sc = SparkContext("local", "Simple App")
```



```
In [22]: df1 = spark.range(2,10000000,2)
In [23]: df2 = spark.range(2,10000000,4)
In [24]: step1 = df1.repartition(5)
In [25]: step12 = df2.repartition(6)
In [26]: step2 = step1.selectExpr("id * 5 as id")
In [27]: step3 = step2.join(step12, ["id"])
In [28]: step4 = step3.selectExpr("sum(id)")
In [29]: step4.collect()
Out[29]: [Row(sum(id)=250000000000)]
```

We are going to do a three-step job: using a simple DataFrame, we'll repartition it, perform a value-by-value manipulation, and then aggregate some values and collect the final result.

When you run this code, we can see that your action triggers one complete Spark job. Let's take a look at the explain plan to ground our understanding of the physical execution plan.

We can access this information on the SQL tab (after we actually run a query) in the Spark UI, as well:



```
In [30]: step4.explain()
t== Physical Plan ==
* (7) HashAggregate(keys=[], functions=[sum(id#6L)])
+- Exchange SinglePartition
  +- *(6) HashAggregate(keys=[], functions=[partial_sum(id#6L)])
    +- *(6) Project [id#6L]
      +- *(6) SortMergeJoin [id#6L], [id#2L], Inner
        :- *(3) Sort [id#6L ASC NULLS FIRST], false, 0
          :  +- Exchange hashpartitioning(id#6L, 200)
          :    +- *(2) Project [(id#0L * 5) AS id#6L]
          :      +- Exchange RoundRobinPartitioning(5)
          :        +- *(1) Range (2, 10000000, step=2, splits=1)
        +- *(5) Sort [id#2L ASC NULLS FIRST], false, 0
          +- Exchange hashpartitioning(id#2L, 200)
            +- Exchange RoundRobinPartitioning(6)
              +- *(4) Range (2, 10000000, step=4, splits=1)
```

# A Spark Job



In general, there should be one Spark job for one action.

Actions always return results.

Each job breaks down into a series of stages, the number of which depends on how many shuffle operations need to take place.



# Developing Spark Applications

# Writing Python Applications



Spark Applications are the combination of two things:

- 1) A Spark cluster and
- 2) Our code.

In this case, the cluster will be local mode and the application will be one that is pre-defined.



Writing PySpark Applications is writing normal Python applications or packages.

To facilitate code reuse, it is common to package multiple Python files into egg or ZIP files of Spark code.

In Python.

Specify a certain script as an executable script that builds the SparkSession.

This is the one that we will pass as the main argument to spark-submit.



# WRITING & RUNNING THE APPLICATION

```
from __future__ import print_function

if __name__ == '__main__':
    from pyspark.sql import SparkSession
    spark = SparkSession.builder \
        .master("local") \
        .appName("Word Count") \
        .config("spark.some.config.option", "some-value") \
        .getOrCreate()

    print(spark.range(5000).where("id > 500").selectExpr("sum(id)").collect())
```

```
ilg@D10Spark:~/spark_code$ spark-submit --master local AppInSP-0.py
```

```
ilg@D10Spark:~/spark_code$ spark-submit --master \
> local /usr/local/spark/examples/src/main/python/pi.py \
> >> /tmp/output.txt
```

Refer “

<https://spark.apache.org/docs/latest/submitting-applications.html>

# Job Scheduling Within an Application



By default, Spark's scheduler runs jobs in FIFO fashion.

If the jobs at the head of the queue don't need to use the entire cluster, later jobs can begin to run right away.

But if the jobs at the head of the queue are large, later jobs might be delayed significantly.

Configure fair sharing between jobs.

Under fair sharing, Spark assigns tasks between jobs in a round-robin fashion so that all jobs get a roughly equal share of cluster resources.

To enable the fair scheduler, *set the `spark.scheduler.mode` property to FAIR when configuring a `SparkContext`.*



# Monitoring and Debugging

# Spark Mon



## Monitoring Landscape

Components we can monitor

### Spark Applications and Jobs

Debugging or just understanding better how your application executes against the cluster is the Spark UI and the Spark logs.



## JVM

- Spark runs the executors in individual Java Virtual Machines (JVMs).
- Next level of detail would be to *monitor the individual virtual machines* (VMs) to better understand how your code is running.

JVM utilities such as **jstack** for providing *stack traces*,

- **jmap** for creating *heap-dumps*,
- **jstat** for reporting *time-series statistics*,
- **jconsole** for *visually exploring* various
- JVM properties are useful for those comfortable with JVM internals.

We can also use a tool like **jvisualvm** to help *profile Spark jobs*.



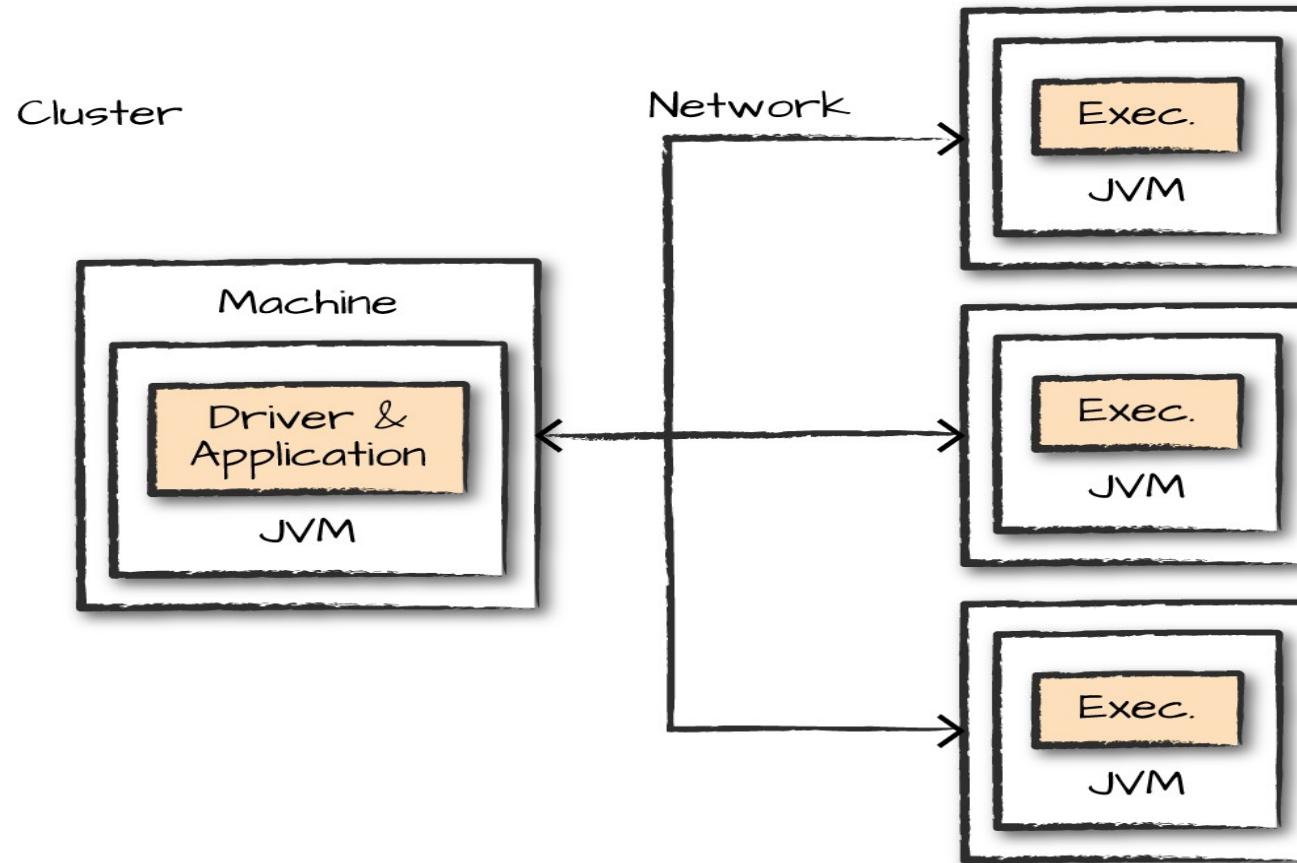
## OS/Machine

JVMs run on a host operating system (OS) and it's important to monitor the state of those machines to ensure that they are healthy.

This includes monitoring things like **CPU, network, and I/O**. These are often reported in cluster-level monitoring solutions

There are more specific tools that you can use, including **dstat, iostat, and iotop**.

# Components of a Spark application that you can monitor



# Spark UI



Let's walk through an example of how you can drill down into a given query.

Open a new Spark shell, run the following code, and we will trace its execution through the Spark UI:

```
In [31]: spark.read\  
....: .option("header", "true")\  
....: .csv("/home/ilg/Documents/sparkdata/data/retail-data/all/online-retail-dataset.csv")\  
....: .repartition(2)\  
....: .selectExpr("instr	Description, 'GLASS') >= 1 as is_glass")\  
....: .groupBy("is_glass")\  
....: .count()\  
....: .collect()
```



<http://192.168.1.26:4040/SQL/>



Not secure | 192.168.1.26:4040/SQL/

Apache Spark 2.4.4 Jobs Stages Storage Environment Executors SQL PySparkShell application UI

## SQL

Completed Queries: 3

Completed Queries (3)

ID	Description	Submitted	Duration	Job IDs
2	collect at <ipython-input-31-7ee3878dda31>:1	+details 2019/09/11 15:15:07	8 s	[9]
1	csv at NativeMethodAccessorImpl.java:0	+details 2019/09/11 15:15:06	0.6 s	[8]
0	collect at <ipython-input-29-c4d29897d8ca>:1	+details 2019/09/11 14:01:56	23 s	[7]



# Streaming

# Stream Processing Fundamentals



## What Is Stream Processing?

Stream processing is the act of continuously incorporating new data to compute a result.

In stream processing, the *input data is unbounded* and has *no predetermined beginning or end*.

It forms a series of events that arrive at the stream processing system (e.g., credit card transactions, clicks on a website, or sensor readings from Internet of Things [IoT] devices).

User applications can then compute various queries over this stream of events (e.g., tracking a running count of each type of event or aggregating them into hourly windows).

The application will output multiple versions of the result as it runs, or perhaps keep it up to date in an external “sink” system such as a key-value store.

# Stream Processing Use Cases



NOTIFICATIONS AND ALERTING

REAL-TIME REPORTING

INCREMENTAL ETL

UPDATE DATA TO SERVE IN REAL TIME

REAL-TIME DECISION MAKING

ONLINE MACHINE LEARNING

# Challenges of Stream Processing



Processing out-of-order data based on application timestamps (also called event time)

- Maintaining large amounts of state
- Supporting high-data throughput
- Processing each event exactly once despite machine failures
- Handling load imbalance and stragglers
- Responding to events at low latency
- Joining with external data in other storage systems
- Determining how to update output sinks as new events arrive
- Writing data transactionally to output systems
- Updating your application's business logic at runtime



# Structured Streaming Basics

# Core Concepts



## Transformations and Actions

### Input Sources

Structured Streaming supports several input sources for reading in a streaming fashion.

- Apache Kafka 0.10
- Files on a distributed file system like HDFS or S3 (Spark will continuously read new files in a directory)
- A socket source for testing

# Sinks



**sinks** specify the destination for the result set of that stream.

Sinks and the execution engine are also responsible for reliably tracking the exact progress of data processing.

- Apache Kafka 0.10
- Almost any file format
- A foreach sink for running arbitrary computation on the output records
- A console sink for testing
- A memory sink for debugging

# Output Modes



We also need to define how we want Spark to write data to that sink.

For instance,

Do we only want to append new information?

Do we want to update rows as we receive more information about them over time (e.g., updating the click count for a given web page)?

Do we want to completely overwrite the result set every single time (i.e. always write a file with the complete click counts for all pages)?

## Supported Modes

- Append (only add new records to the output sink)
- Update (update changed records in place)
- Complete (rewrite the full output)



## Triggers

triggers define when data is output—that is, when Structured Streaming should check for new input data and update its result.

By default, *Structured Streaming will look for new input records as soon as it has finished processing the last group of input data*, giving the lowest latency possible for new results.

***Spark also supports triggers based on processing time (only look for new data at a fixed interval).***



## Event-Time Processing

Structured Streaming also has support for event-time processing i.e. *processing data based on timestamps included in the record that may arrive out of order.*

### EVENT-TIME DATA

Event-time means time fields that are embedded in your data.

This means that rather than *processing data according to the time it reaches your system, you process it according to the time that it was generated*, even if records arrive out of order at the streaming application due to slow uploads or network delays.



## WATERMARKS

Watermarks are a feature of streaming systems that allow you to specify how late they expect to see data in event time.

For example,

In an application that processes logs from mobile devices, one might expect logs to be up to 30 minutes late due to upload delays.

Note:

Systems that support event time, including Structured Streaming, usually allow setting watermarks to limit how long they need to remember old data. Watermarks can also be used to control when to output a result for a particular event time window



## Structured Streaming in Action



## Data set for working

We're going to be working with the **Heterogeneity Human Activity Recognition Dataset**.

The data consists of smartphone and smartwatch sensor readings from a variety of devices—specifically, the accelerometer and gyroscope, sampled at the highest possible frequency supported by the devices.

Readings from these sensors were recorded while users performed activities like biking, sitting, standing, walking, and so on.

There are several different smartphones and smartwatches used, and nine total users.



Let's read in the **static version** of the dataset as a DataFrame:

```
In [34]: staticData = spark.read.json("/home/ilg/Documents/sparkdata/data/activity-data/")

In [35]: dataSchema = staticData.schema

In [36]: staticData.printSchema()
root
|-- Arrival_Time: long (nullable = true)
|-- Creation_Time: long (nullable = true)
|-- Device: string (nullable = true)
|-- Index: long (nullable = true)
|-- Model: string (nullable = true)
|-- User: string (nullable = true)
|-- gt: string (nullable = true)
|-- x: double (nullable = true)
|-- y: double (nullable = true)
|-- z: double (nullable = true)
```

```
ilg@D10Spark:~/Documents/sparkdata/data/activity-data$ head part-00071-tid-730451297822678341-1dda7027-207
21-4d73-a0e2-7fb6a91e1d1f-0-c000.json
{"Arrival_Time":1424686735154,"Creation_Time":1424686733156098398,"Device":"nexus4_1","Index":31,"Model":"nexus4","User":"g","gt":"stand","x":-7.324219E-4,"y":0.0015716553,"z":-0.016677856}
{"Arrival_Time":1424686735357,"Creation_Time":1424688581406464967,"Device":"nexus4_2","Index":78,"Model":"nexus4","User":"g","gt":"stand","x":6.866455E-4,"y":0.033355713,"z":0.015182495}
{"Arrival_Time":1424686735557,"Creation_Time":1424688581612733277,"Device":"nexus4_2","Index":119,"Model":"nexus4","User":"g","gt":"stand","x":6.866455E-4,"y":-0.010437012,"z":2.288818E-4}
 {"Arrival_Time":1424686735760,"Creation_Time":1424686733760499033,"Device":"nexus4_1","Index":151,"Model":"nexus4","User":"g","gt":"stand","x":0.003540039,"y":0.009048462,"z":-0.017745972}
 {"Arrival_Time":1424686735963,"Creation_Time":1424688582015748414,"Device":"nexus4_2","Index":199,"Model":"nexus4","User":"g","gt":"stand","x":-3.814697E-4,"y":-0.027526855,"z":-0.022201538}
```



Next, let's **create a streaming** version of the same Dataset, which will read each input file in the dataset one by one as if it was a stream.

```
In [37]: streamer = spark.readStream.schema(dataSchema) \
.... .option("maxFilesPerTrigger", 1) \
.... .json("/home/ilg/Documents/sparkdata/data/activity-data")
```

We can now specify transformations on our streaming DataFrame before finally calling an action to start the stream.

*we will group and count data by the gt column, which is the activity being performed by the user at that point in time.*

```
In [38]: activityCounts = streamer.groupBy("gt").count()
```

Because this code is being written in local mode on a small machine, we are going to set the shuffle partitions to a small value to avoid creating too many shuffle partitions:

```
In [39]: spark.conf.set("spark.sql.shuffle.partitions", 5)
```



Now that we set up our transformation, we need only to specify our action to start the query.

we will specify an output destination, or *output sink* for our *result of this query*.

we are going to write to a memory sink which keeps an in-memory table of the results.

we use the *complete output mode*.

This mode rewrites all of the keys along with their counts after every trigger.

```
In [40]: activityQuery = activityCounts.writeStream.queryName("activity_counts")\
....: .format("memory").outputMode("complete")\
....: .start()
```

We are now writing out our stream!



When we run the preceding code, we also want to include the following line:

```
In [41]: activityQuery.awaitTermination()
```

After this code is executed, the streaming computation will have started in the background.

The query object is a handle to that active streaming query, and we must specify that we would like to wait for the termination of the query using **activityQuery.awaitTermination()** to prevent the driver process from exiting while the query is active.

*It must be included in your production applications; otherwise, your stream won't be able to run.*

# Live streams



Spark lists this stream, and other active ones, under the active streams in our SparkSession. We can see a list of those streams by running

```
In [45]: spark.streams.active
Out[45]: [<pyspark.sql.streaming.StreamingQuery at 0x7f8d31fb21d0>]
```

```
In [46]: from time import sleep
In [47]: for x in range(5):
....:     spark.sql("select * from activity_counts").show()
....:     sleep(1)
....:
```

OUTPUT

gt	count
sit	984714
stand	910783
stairsdown	749059
walk	1060402
stairsup	836598
null	835725
bike	863710

# Transformations on Streams



## Selections and Filtering

All select and filter transformations are supported in Structured Streaming, as are all DataFrame functions and individual column manipulations.

```
In [50]: from pyspark.sql.functions import expr

In [51]: simpleTransform = streamer.withColumn("stairs",expr("gt like '%stairs%'"))\
....: .where("stairs")\
....: .where("gt is not null")\
....: .select("gt","model","arrival_time","creation_time")\
....: .writeStream\
....: .queryName("simple_transform")\
....: .format("memory")\
....: .outputMode("append")\
....: .start()
```

# Aggregations



Structured Streaming has excellent support for aggregations. We can specify arbitrary aggregations.

```
In [52]: deviceModelStats = streamer.cube("gt", "model").avg() \
....: .drop("avg(Arrival_time)") \
....: .drop("avg(creation_Time)") \
....: .drop("avg(Index)") \
....: .writeStream.queryName("device_counts").format("memory") \
....: .outputMode("complete") \
....: .start()
```



```
In [76]: spark.sql("SELECT * FROM device_counts").show(100)
```

gt	model	avg(x)	avg(y)	avg(z)
sit	null	-5.49433244039591E-4	2.791446281700069...	-2.33994461689890...
stand	null	-3.11082189691724...	3.218461665975315...	2.141300040636475E-4
sit	nexus4	-5.49433244039591E-4	2.791446281700069...	-2.33994461689890...
stand	nexus4	-3.11082189691724...	3.218461665975315...	2.141300040636475E-4
null	null	-0.00847688860109...	-7.30455258739177...	0.003090601491419...
null	null	4.796918779024539E-4	-0.00601540958963...	-0.01013356489164804
walk	null	-0.00390116006094368	0.001052508689953...	-6.95435553042992...
null	nexus4	-0.00847688860109...	-7.30455258739177...	0.003090601491419...
null	nexus4	4.796918779024539E-4	-0.00601540958963...	-0.01013356489164804
bike	null	0.02268875955086682	-0.00877912156368...	-0.08251001663412386
stairsup	null	-0.02479965287771635	-0.00800392344379...	-0.10034088415060402
stairsdown	null	0.02161390866916528	-0.03249018824752614	0.12035922691504054
bike	nexus4	0.02268875955086682	-0.00877912156368...	-0.08251001663412386
walk	nexus4	-0.00390116006094368	0.001052508689953...	-6.95435553042992...
stairsdown	nexus4	0.02161390866916528	-0.03249018824752614	0.12035922691504054
stairsup	nexus4	-0.02479965287771635	-0.00800392344379...	-0.10034088415060402



# Joins

Structured Streaming supports joining streaming DataFrames to static DataFrames.

Spark 2.4 has added the ability to join multiple streams together.

We can do multiple column joins and supplement streaming data with that from static data sources.

```
In [152]: historicalAgg = staticD.groupBy("gt","model").avg()

In [153]: deviceModelStats = streamer.drop("Arrival_Time","Creating_Time","Index")\
    ...: .cube("gt","model").avg()\
    ...: .join(historicalAgg,[ "gt", "model" ])\ 
    ...: .writeStream.queryName("device_counts").format("memory")\
    ...: .outputMode("complete")\
    ...: .start()
```



```
In [155]: spark.sql("select * from device_countes").show(20, False)
```



```
In [155]: spark.sql("select * from device_countes").show(20, False)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|gt    |model |avg(Creation_Time) |avg(x)   |avg(y)      |avg(z)      |avg(Arrival_Time) |avg(Creation_Time)|avg(Index)   |avg(x)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|bike  |nexus4|1.42475212887321677E18|0.02292265563822366 |-0.007740414574840931|-0.08221121509759116 |1.4247511343399863E12|1.42475212736958874E18|326459.6867328154 |0.02268875955086
6845  |-0.008779121563686761|-0.08251001663412343 |
|null  |nexus4|1.42474991923179085E18|-0.008880155571234151 |-9.875229400002118E-4|0.0032457577022213593 |1.4247490028763398E12|1.42474991948212864E18|219276.9663669269 |-0.0084768886010
96473 |-7.304552587391897E-4|0.00309060149141993441 |
|stairsdown|nexus4|1.42474550392778445E18|0.021483202810971803 |-0.03254710383877547|0.1199408224961491 |1.4247445914128572E12|1.42474550363563802E18|230452.44623187225|0.02161390866916
542   |-0.03249018824752616|0.12035922691504071 |
```



# Event-Time and Stateful Processing

## Event Time



At a higher level, in stream-processing systems there are effectively two relevant times for each event:

- 1) Time at which it actually occurred (event time)
- 2) Time that it was processed or reached the stream-processing system (processing time)



## Event time

Event time is the time that is embedded in the data itself. The time that an event actually occurs.

This helps us in comparing events.

*Challenge here is out of order data.stream processing system must be able to handle out-of-order or late data.*

## Processing time

Processing time is the time at which the stream-processing system actually receives data.

# Stateful Processing



Structured Streaming maintains and updates the information for us.

We simply specify the logic. When performing a stateful operation, Spark stores the intermediate information in a state store.

Spark's current state store implementation is an in-memory state store that is made fault tolerant by storing intermediate state to the checkpoint directory.

## **When stateful processing necessary ?.**

Stateful processing is only necessary when you need to use or update intermediate information (state) over longer periods of time (in either a microbatch or a record-at-a-time approach).



# Event-Time Basics

```
In [99]: spark.conf.set("spark.sql.shuffle.partitions",5)

In [100]: staticD = spark.read.json("/home/ilg/Documents/sparkdata/data/activity-data/")

In [101]: streamingD = spark\
....: .readStream\
....: .schema(staticD.schema)\
....: .option("maxFilesPerTrigger",10)\
....: .json("/home/ilg/Documents/sparkdata/data/activity-data/")
```

```
In [103]: streamingD.printSchema()
root
|-- Arrival_Time: long (nullable = true)
|-- Creation_Time: long (nullable = true)
|-- Device: string (nullable = true)
|-- Index: long (nullable = true)
|-- Model: string (nullable = true)
|-- User: string (nullable = true)
|-- gt: string (nullable = true)
|-- x: double (nullable = true)
|-- y: double (nullable = true)
|-- z: double (nullable = true)
```

In this dataset, there are two time-based columns.

The **Creation\_Time** column defines when an event was created, whereas the **Arrival\_Time** defines when an event hit our servers somewhere upstream.

# Windows on Event Time



The first step in event-time analysis is to convert the timestamp column into the proper Spark SQL timestamp type.

Our current column is unixtime nanoseconds (represented as a long), therefore we're going to have to do a little manipulation to get it into the proper format:

```
In [104]: withEventTimeT = streamingD.selectExpr(  
    ...: "*",  
    ...: "cast(cast(Creation_Time as double)/1000000000 as timestamp) as event_time")
```

We're now prepared to do arbitrary operations on event time!

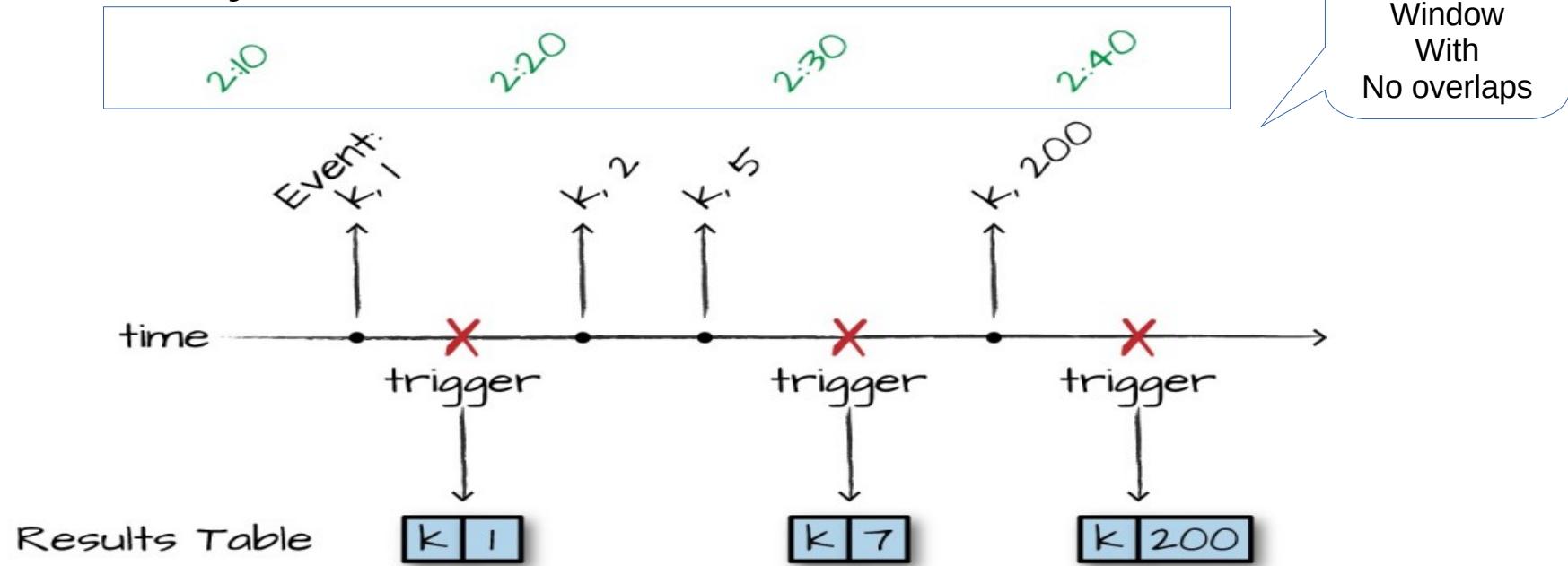
```
In [105]: withEventTimeT.printSchema()  
root  
| -- Arrival_Time: long (nullable = true)  
| -- Creation_Time: long (nullable = true)  
| -- Device: string (nullable = true)  
| -- Index: long (nullable = true)  
| -- Model: string (nullable = true)  
| -- User: string (nullable = true)  
| -- gt: string (nullable = true)  
| -- x: double (nullable = true)  
| -- y: double (nullable = true)  
| -- z: double (nullable = true)  
| -- event_time: timestamp (nullable = true)
```



# Tumbling Windows

The simplest operation is simply to count the number of occurrences of an event in a given window.

Diagram - Depicts the process when performing a simple summation based on the input data and a key.





Now we're writing out to the in-memory sink for debugging, so we can query it with SQL after we have the stream running

```
In [108]: from pyspark.sql.functions import window,col  
  
In [109]: withEventTimeT.groupBy(window(col("event_time"),"10 minutes")).count()\  
...: .writeStream\  
...: .queryName("pyevents_per_window")\  
...: .format("memory")\  
...: .outputMode("complete")\  
...: .start()  
Out[109]: <pyspark.sql.streaming.StreamingQuery at 0x7f8d31636240>
```

```
In [111]: spark.sql("select * from pyevents_per_window").printSchema()  
root  
|-- window: struct (nullable = false)  
|   |-- start: timestamp (nullable = true)  
|   |-- end: timestamp (nullable = true)  
|-- count: long (nullable = false)
```

```
In [123]: spark.sql("select * from pyevents_per_window").show(10)  
+-----+-----+  
|      window| count|  
+-----+-----+  
|[2015-02-24 06:50...|150773|  
|[2015-02-24 08:00...|133323|  
|[2015-02-23 07:30...|100853|  
|[2015-02-23 05:20...| 99178|  
|[2015-02-24 07:30...|125679|  
|[2015-02-24 08:10...|105494|  
|[2015-02-23 05:30...|100443|  
|[2015-02-23 05:40...| 88681|  
|[2015-02-23 08:20...|106075|  
|[2015-02-21 19:40...|     35|  
+-----+-----+  
only showing top 10 rows
```



# Sliding windows

Another approach is that we can decouple the window from the starting time of the window

```
In [134]: withEventTimeT.groupBy(window(col("event_time"), "10 minutes", "5 minutes")).count()\
...: .writeStream\
...: .queryName("events_per_window")\
...: .format("memory")\
...: .outputMode("complete")\
...: .start()
Out[134]: <pyspark.sql.streaming.StreamingQuery at 0x7f8d315cea20>
```

```
In [140]: spark.sql("select * from events_per_window").show(20, False)
+-----+-----+
|window|count|
+-----+-----+
|[2015-02-23 09:15:00, 2015-02-23 09:25:00]|107668|
|[2015-02-24 06:50:00, 2015-02-24 07:00:00]|150773|
|[2015-02-24 08:00:00, 2015-02-24 08:10:00]|133323|
|[2015-02-21 19:35:00, 2015-02-21 19:45:00]|35|
|[2015-02-23 07:30:00, 2015-02-23 07:40:00]|100853|
|[2015-02-23 05:20:00, 2015-02-23 05:30:00]|99178|
|[2015-02-23 08:25:00, 2015-02-23 08:35:00]|91684|
|[2015-02-24 09:25:00, 2015-02-24 09:35:00]|203945|
```

# Handling Late Data with Watermarks



In preceding examples, We never specified how late we expect to see data.

This means that Spark is going to need to store that intermediate data forever because we never specified a watermark, or a time at which we don't expect to see any more data.

This applies to all stateful processing that operates on event time.

*We must specify this watermark in order to age-out data in the stream, so that we don't overwhelm the system over a long period of time.*

What is a watermark ?

Watermark is an amount of time following a given event or set of events after which we do not expect to see any more data from that time.





Now, Structured Streaming will wait until 30 minutes after the final timestamp of this 10-minute rolling window before it finalizes the result of that window.

We can query our table and see the intermediate results because we're using complete mode—they'll be updated over time.

In append mode, this information won't be output until the window closes.

```
In [141]: from pyspark.sql.functions import window,col  
  
In [142]: withEventTimeT\  
....: .withWatermark("event_time","30 minute")\  
....: .groupBy(window(col("event_time"),"10 minutes","5 minutes"))\  
....: .count()\  
....: .writeStream\  
....: .queryName("pyevents_per_windows")\  
....: .format("memory")\  
....: .outputMode("complete")\  
....: .start()  
Out[142]: <pyspark.sql.streaming.StreamingQuery at 0x7f8d315de5c0>
```

```
In [143]: spark.sql("select * from pyevents_per_windows").show(20, False)  
+-----+-----+  
|window|count |  
+-----+-----+  
|[2015-02-23 09:15:00, 2015-02-23 09:25:00]|107668|  
|[2015-02-24 06:50:00, 2015-02-24 07:00:00]|150773|  
|[2015-02-24 08:00:00, 2015-02-24 08:10:00]|133323|  
|[2015-02-21 19:35:00, 2015-02-21 19:45:00]|35|  
|[2015-02-23 07:30:00, 2015-02-23 07:40:00]|100853|  
|[2015-02-23 05:20:00, 2015-02-23 05:30:00]|99178|  
|[2015-02-23 08:25:00, 2015-02-23 08:35:00]|91684|
```

## Benefit of Watermark



In Spark, specifying a watermark allows it to free those objects from memory, allowing your stream to continue running for a long time.

# Dropping Duplicates in a Stream



**Deduplication** is an important tool in many applications, especially when messages might be delivered multiple times by upstream systems.

A perfect example of this are **Internet of Things (IoT) applications** that have upstream producers generating messages in unstable network environments, and the same message might end up being sent multiple times.

Our downstream applications and aggregations should be able to assume that there is only one of each message.



Let's begin the de-duplication process.

The goal here will be to de-duplicate the number of events per user by removing duplicate events.

```
from pyspark.sql.functions import expr
```

```
In [150]: withEventTime\  
....: .withWatermark("event_time", "5 seconds")\  
....: .dropDuplicates(["User", "event_time"]).groupBy("User")\  
....: .count()\  
....: .writeStream\  
....: .queryName("pydeduplicated")\  
....: .format("memory")\  
....: .outputMode("complete")\  
....: .start()  
Out[150]: <pyspark.sql.streaming.StreamingQuery at 0x7f8d31f2b710>
```



```
In [151]: spark.sql("select * from pydeduplicated").show(20, False)
+---+---+
|User|count|
+---+---+
|a   |80855|
|b   |91241|
|c   |77155|
|g   |91672|
|h   |77328|
|e   |96875|
|f   |92055|
|d   |81245|
|i   |92554|
+---+---+
```



# Structured Streaming in Production

# Fault Tolerance and Checkpointing



The most important operational concern for a streaming application is failure recovery.

Faults are inevitable:

we're going to lose a machine in the cluster,

A schema will change by accident without a proper migration, or

we may even intentionally restart the cluster or application.

In any of these cases, *Structured Streaming* allows us to **recover** an application by just restarting it.

To do this, you must configure the application to use

- checkpointing
- write-ahead logs,

both of which are handled automatically by the engine.



## Recovery – how ?

- We must configure a query to write to a checkpoint location on a reliable file system (e.g., HDFS, S3, or any compatible filesystem).
- Structured Streaming will then *periodically save all relevant progress information as well as the current intermediate state values to the checkpoint location.*
- In a failure scenario, we simply need to restart your application, making sure to point to the same checkpoint location, and it will automatically recover its state and start processing data where it left off.
- We do not have to manually manage this state on behalf of the application —Structured Streaming does it for us.



To use **checkpointing**, specify your checkpoint location before starting your application through the **checkpointLocation** option on **writeStream**.

```
In [3]: staticJ = spark.read.json("/home/ilg/Documents/sparkdata/data/activity-data/")
```

```
In [5]: streamJ = spark\  
....: .readStream\  
....: .schema(staticJ.schema)\  
....: .option("maxFilesPerTrigger", 10)\  
....: .json("/home/ilg/Documents/sparkdata/data/activity-data/")\  
....: .groupBy("gt")\  
....: .count()
```

```
In [6]: queryJ = streamJ\  
....: .writeStream\  
....: .outputMode("complete")\  
....: .option("checkpointLocation", "/tmp/python/spck/")\  
....: .queryName("test_python_stream")\  
....: .format("memory")\  
....: .start()
```



# Query Status

The query status is the most basic monitoring API, so it's a good starting point. It aims to answer the question,

“What processing is my stream performing right now?”

This information is reported in the status field of the query object returned by `startStream`.

```
In [7]: queryJ.status
Out[7]:
{'message': 'Getting offsets from FileStreamSource[file:/home/ilg/Documents/sparkdata/data/activity-data']
, 'isDataAvailable': False,
 'isTriggerActive': True}
```

To get the status of a given query, simply running the command `query.status` will return the current status of the stream.



## Query recentprogress

- Equally important is an ability to view the query's progress. The progress API allows us to answer questions like :-

“At what rate am I processing tuples?” or “How fast are tuples arriving from the source?”

By running **query.recentProgress**, we will get access to more time-based information like the processing rate and batch durations.

- The streaming query progress also includes information about the input sources and output sinks behind our stream.

```
In [10]: queryJ.recentProgress
Out[10]:
[{'id': 'f2ca2eae-a6bf-406e-a432-3e5ae77db1c9',
 'runId': '691f1944-945a-4622-9f36-ae4d63deab89',
 'name': 'test_python_stream',
 'timestamp': '2019-09-12T16:16:55.771Z',
 'batchId': 0,
 'numInputRows': 780127,
 'processedRowsPerSecond': 44277.59804756229,
 'durationMs': {'addBatch': 16731,
 'getBatch': 241,
 'getOffset': 209,
 'queryPlanning': 216,
```



# Advanced Analytics and Machine Learning



Advanced analytics tools in Spark, including:

Preprocessing your data (cleaning data and feature engineering)

- Supervised learning
- Recommendation learning
- Unsupervised engines
- Graph analytics
- Deep learning

# Advanced Analytics primer



Supervised learning, including **classification** and **regression**, where the goal is to predict a label for each data point based on various features.

Recommendation engines to suggest products to **users based on behaviour**

Unsupervised learning, including **clustering**, **anomaly** detection, and **topic modeling**, where the goal is to discover structure in the data.

Graph analytics tasks such as **searching for patterns** in a social network

# Supervised Learning



Using **historical data that already has labels** (often called the dependent variables), train a model to ***predict the values*** of those labels based on various features of the data points.

An example would be to predict a person's income (the dependent variable) based on age (a feature).

Training process usually proceeds through an iterative optimization algorithm such as gradient descent.

The training algorithm starts with a basic model and gradually improves it by adjusting various internal parameters (coefficients) during each training iteration. *The result of this process is a trained model that you can use to make predictions on new data.*

# Classification



Classification is the *act of training an algorithm to predict a dependent variable* that is **categorical** (belonging to a discrete, finite set of values).

The most common case is **binary classification**, where our resulting model will make a prediction that a given item belongs to one of two groups.

The canonical example is classifying email spam.

Multiclassification:

When we classify items into **more than just two categories**, we call this multiclass classification.

# Classification use cases



## Predicting disease

A doctor or hospital might have a historical dataset of behavioral and physiological attributes of a set of patients.

They could use this dataset to train a model on this historical data (and evaluate its success and ethical implications before applying it) and then leverage it to predict whether or not a patient has heart disease or not.

This is an example of binary classification (healthy heart, unhealthy heart)

OR

multiclass classification (healthly heart, or one of several different diseases).

# Classification use cases



## Classifying images

There are a number of applications from companies like Apple, Google, or Facebook that can predict who is in a given photo by running a classification model that has been trained on historical images of people in your past photos.

Another common use case is to classify images or label the objects in images.

# Classification use cases



## Predicting customer churn

A more business-oriented use case might be predicting customer churn—that is, which customers are likely to stop using a service.

We can do this by training a binary classifier on past customers that have churned (and not churned) and using it to try and predict whether or not current customers will churn.

# Classification use cases



## Buy or won't buy

Companies often want to predict whether visitors of their website will purchase a given product.

They might use information about users' browsing pattern or attributes such as location in order to drive this prediction.

# Regression



In regression, we try to ***predict a continuous variable*** (a real number).

In a way, rather than predicting a category, we want to predict a value on a number line.

*We will train on historical data to make predictions about data we have never seen.*

# Regression use cases



## Predicting sales

A store may want to predict total product sales on given data using historical sales data. There are a number of potential input variables, but a simple example might be using last week's sales data to predict the next day's data.

## Predicting height

Based on the heights of two individuals, we might want to predict the heights of their potential children.

## Predicting the number of viewers of a show

A media company like Netflix might try to predict how many of their subscribers will watch a particular show.

# Recommendation



By studying people's *explicit preferences (through ratings) or implicit ones (through observed behavior)* for various products or items, an algorithm can make recommendations on what a user may like by drawing similarities between the users or items.

By looking at these similarities, the algorithm makes recommendations to users based on what similar users liked, or what other products resemble the ones the user already purchased.

# Recommendation use cases



## Movie recommendations

Netflix uses Spark, although not necessarily its built-in libraries, to make large-scale movie recommendations to its users.

It does this by studying what movies users watch and do not watch in the Netflix application.

In addition, Netflix likely takes into consideration how similar a given user's ratings are to other users'.

# Recommendation use cases



## Product recommendations

Amazon uses product recommendations as one of its main tools to increase sales.

For instance,

based on the items in our shopping cart, Amazon may recommend other items that were added to similar shopping carts in the past. Likewise, on every product page, Amazon shows similar products purchased by other users.

# Unsupervised Learning



Unsupervised learning is the act of trying to ***find patterns or discover the underlying structure*** in a given set of data.

## Anomaly detection

Given some standard event type often occurring over time, we might want to report when a nonstandard type of event occurs.

For example,

A security officer might want to receive notifications when a strange object (think vehicle, skater, or bicyclist) is observed on a pathway.

# Unsupervised Learning



## User segmentation

Given a set of **user behaviors**, we might want to better understand what attributes certain users share with other users.

For instance,

A gaming company might cluster users based on properties like the number of hours played in a given game.

The algorithm might reveal that casual players have very different behaviour than hardcore gamers, and possibly allow the company to offer different recommendations or rewards to each player.

# Unsupervised Learning



## Topic modeling

Given a set of documents, we might analyze the different words contained therein to see if there is some underlying relation between them.

For example,

Given a number of web pages on data analytics, a topic modeling algorithm can cluster them into pages about machine learning, SQL, streaming, and so on based on groups of words that are more common in one topic than in others.

# Graph Analytics



Graph analytics is the study of structures in which we specify vertices (which are objects) and edges (which represent the relationships between those objects).

For example,

The vertices might represent people and products, and edges might represent a purchase.

By looking at the properties of vertices and edges, we can better understand the connections between them and the overall structure of the graph.

*Since graphs are all about relationships, anything that specifies a relationship is a great use case for graph analytics.*

# Graph Analytics use cases



## Fraud prediction

Capital One uses Spark's graph analytics capabilities to better understand fraud networks.

By using historical fraudulent information (like phone numbers, addresses, or names) they discover fraudulent credit requests or transactions.

For instance,

- any user accounts within two hops of a fraudulent phone number might be considered suspicious.

# Graph Analytics use cases



## Anomaly detection

By looking at how networks of individuals connect with one another, outliers and anomalies can be flagged for manual analysis.

For instance,

- if typically in our data each vertex has ten edges associated with it and a given vertex only has one edge, that might be worth investigating as something strange.

# Graph Analytics use cases



## Classification

Given some facts about certain vertices in a network, we can classify other vertices according to their connection to the original node.

For instance,

- if a certain individual is labeled as an influencer in a social network, we could classify other individuals with similar network structures as influencers.

# Graph Analytics use cases



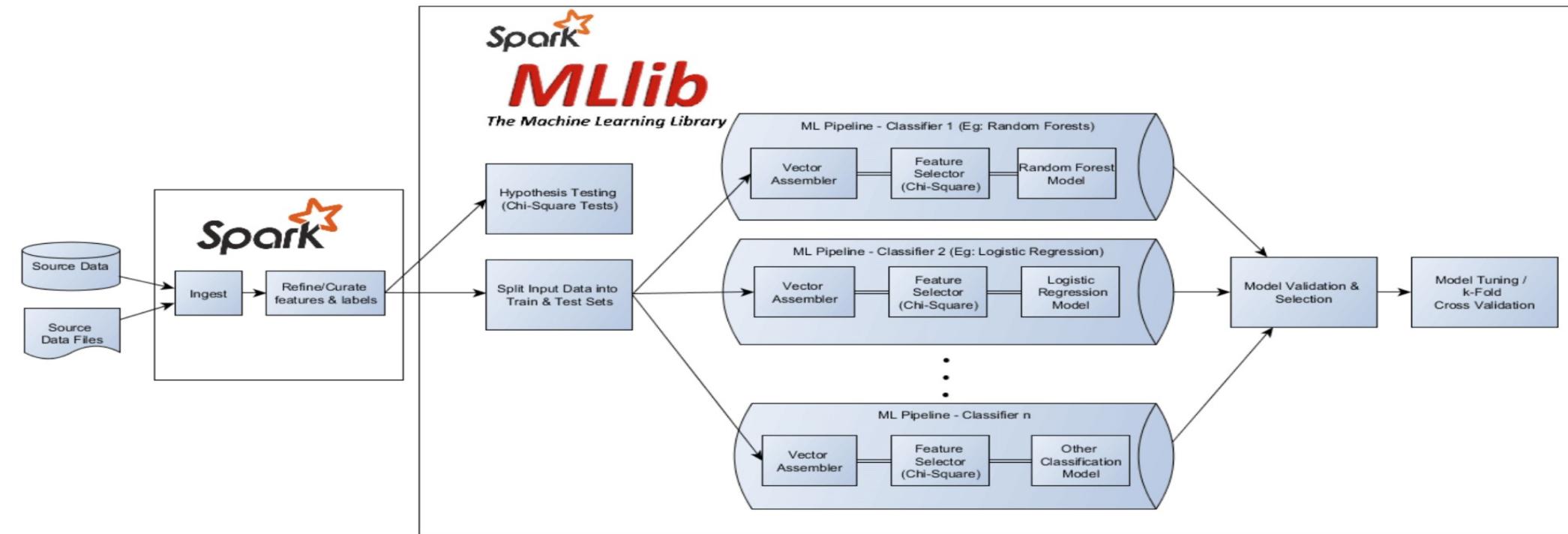
## Recommendation

Google's original web recommendation algorithm, PageRank, is a graph algorithm that analyzes website relationships in order to rank the importance of web pages.

For example,

- a web page that has a lot of links to it is ranked as more important than one with no links to it.

# machine learning workflow





## Process involves :-

1. Gathering and collecting the relevant data for your task.
2. Cleaning and inspecting the data to better understand it.
3. Performing feature engineering to allow the algorithm to leverage the data in a suitable form (e.g., converting the data to numerical vectors).
4. Using a portion of this data as a training set to train one or more algorithms to generate some candidate models.
5. Evaluating and comparing models against your success criteria by objectively measuring results on a subset of the same data that was not used for training. This allows you to better understand how your model may perform in the wild.
6. Leveraging the insights from the above process and/or using the model to make predictions, detect anomalies, or solve more general business challenges.

# Data collection



Naturally it's hard to create a training set without first collecting data. This means at least gathering the datasets you'll want to leverage to train your algorithm.

*Spark is an excellent tool for this because of its ability to speak to a variety of data sources and work with data big and small.*

# Data cleaning



After you've gathered the proper data, you're going to need to clean and inspect it.

This is typically done as part of a process called exploratory data analysis, or EDA.

EDA generally means using interactive queries and visualization methods in order to better understand distributions, correlations, and other details in your data.

During this process you may notice you need to remove some values that may have been **misrecorded** upstream or that other values may be missing.

The multitude of Spark functions in the structured APIs will provide a simple way to clean and report on your data.



## Feature engineering

Now that you collected and cleaned your dataset, it's time to convert it to a form suitable for machine learning algorithms, which generally means numerical features. Proper feature engineering can often make or break a machine learning application, so this is one task you'll want to do carefully. The process of feature engineering includes a variety of tasks, such as *normalizing data, adding variables to represent the interactions of other variables, manipulating categorical variables, and converting them to the proper format* to be input into our machine learning model.



## Feature engineering

In MLlib, Spark's machine learning library, all *variables will usually have to be input as **vectors of doubles*** (regardless of what they actually represent).

Spark provides the essentials we'll need to manipulate your data using a variety of machine learning statistical techniques.

# Optional or Case based



The following few steps

- 1) Training models,**
- 2) Model tuning,**
- 3) Evaluation**

are not relevant to all use cases.

This is a general workflow that may vary significantly based on the end objective you would like to achieve.

## Training models



At this point in the process we have a dataset of historical information (e.g., spam or not spam emails) and a task we would like to complete (e.g., classifying spam emails).

Next, we will want to **train a model to predict the correct output**, given some input.

During the training process, the parameters inside of the model will change according to how well the model performed on the input data.

For instance,

To classify spam emails, our algorithm will likely find that certain words are better predictors of spam than others and therefore weight the parameters associated with those words higher.

# Training models



In the end, the trained model will find that certain words should have more influence (because of their consistent association with spam emails) than others.

The **output of the training process is what we call a model.**

Models can then be used to gain insights or to make future predictions.

To make predictions, *we will give the model an input and it will produce an output based on a mathematical manipulation of these inputs.*

Using the classification example, *given the properties of an email, it will predict whether that email is spam or not by comparing to the historical spam and not spam emails that it was trained on.*



We want to leverage our model to produce insights.  
Thus, we must answer the question:  
how do we know our model is any good at what it's supposed to do?  
That's where **model tuning and evaluation** come in.

# Model tuning and evaluation



**We should split your data into multiple portions and use only one for training.** This is an essential step in the machine learning process because when you build an advanced analytics model you want that model to generalize to data it has not seen before.

Splitting our dataset into multiple portions allows us to objectively test the effectiveness of the trained model against a set of data that it has never seen before. *The objective is to see if your model understands something fundamental about this data process* or whether or not it just noticed the things particular to only the training set (sometimes called overfitting). That's why it is called a test set.

# Model tuning and evaluation



In the process of training models, we also might take another, separate subset of data and treat that as another type of test set, called a validation set, in order to try out different **hyperparameters** (*parameters that affect the training process*) and compare different variations of the same model without overfitting to the test set.

In our classification example :

we have three sets of data:

A training set for training models,

A validation set for testing different variations of the models that we're training, and lastly,

A test set we will use for the final evaluation of our different model variations to see which one performed the best.

## Leveraging the model and/or insights



After running the model through the training process and ending up with a well-performing model, you are now ready to use it!

# Spark's Advanced Analytics Toolkit



The primary package is **MLlib**, which provides an interface for building machine learning pipelines.

## What Is MLlib?

MLlib is Apache Spark's scalable machine learning library.

MLlib is Spark's scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives.



When and why should you use MLlib (versus scikit-learn, TensorFlow, any other package) ?

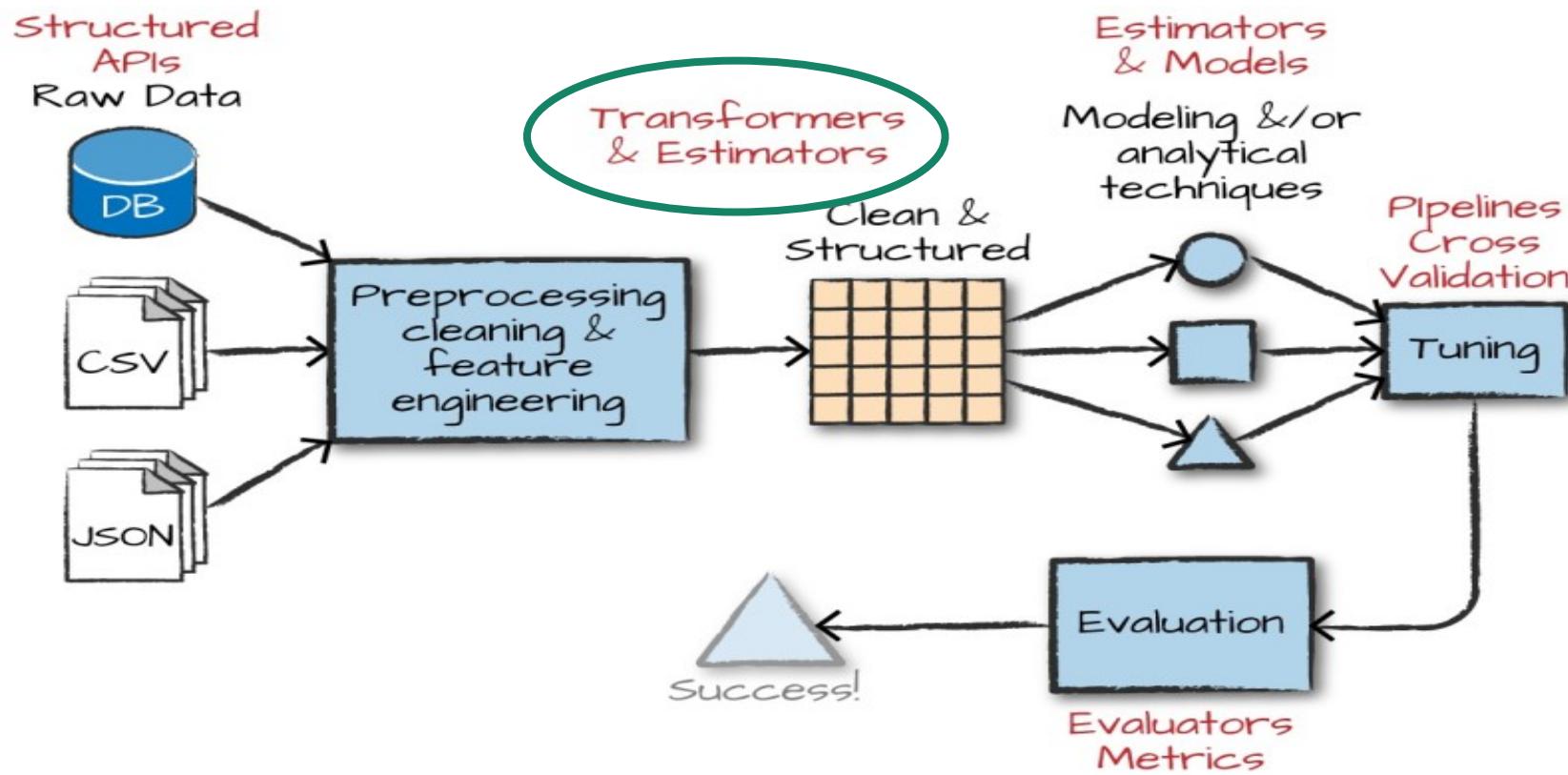
Simple, Scalability.

- 1) We want to *leverage Spark for preprocessing and feature generation* to reduce the amount of time it might take to produce training and test sets from a large amount of data.
- 2) When your input *data or model size become too difficult* or inconvenient to put on one machine, use Spark to do the heavy lifting. Spark makes distributed machine learning very simple.

# High-Level MLlib Concepts



The machine learning workflow, in Spark





# Transformers

Transformers are functions that convert raw data in some way.

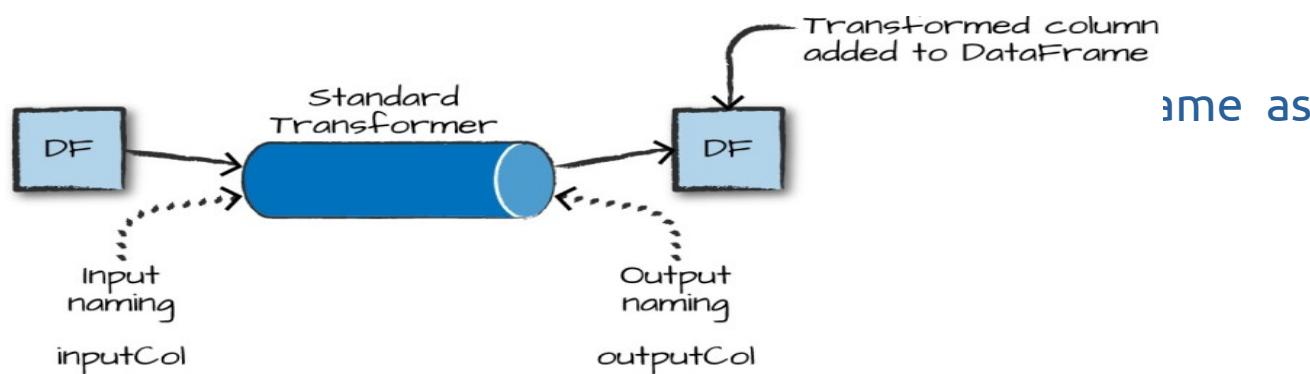
This might be to create a new interaction variable (from two other variables), normalize a column, or simply change an Integer into a Double type to be input into a model.

---

An example of a transformer is one that *converts string categorical variables into numerical values that can be used in MLlib*.

- Transformers are primarily used in preprocessing and feature engineering.

- Transformers output



# Estimators



Two kinds :-

- 1) Estimators can be a kind of transformer that is initialized with data.

For instance,

To normalize numerical data we'll need to initialize our transformation with some information about the current values in the column we would like to normalize. This requires two passes over our data—the initial pass generates the initialization values.

- 2) To actually apply the generated function over the data.

In the Spark's nomenclature, algorithms that allow users to train a model from data are also referred to as estimators

# Evaluator



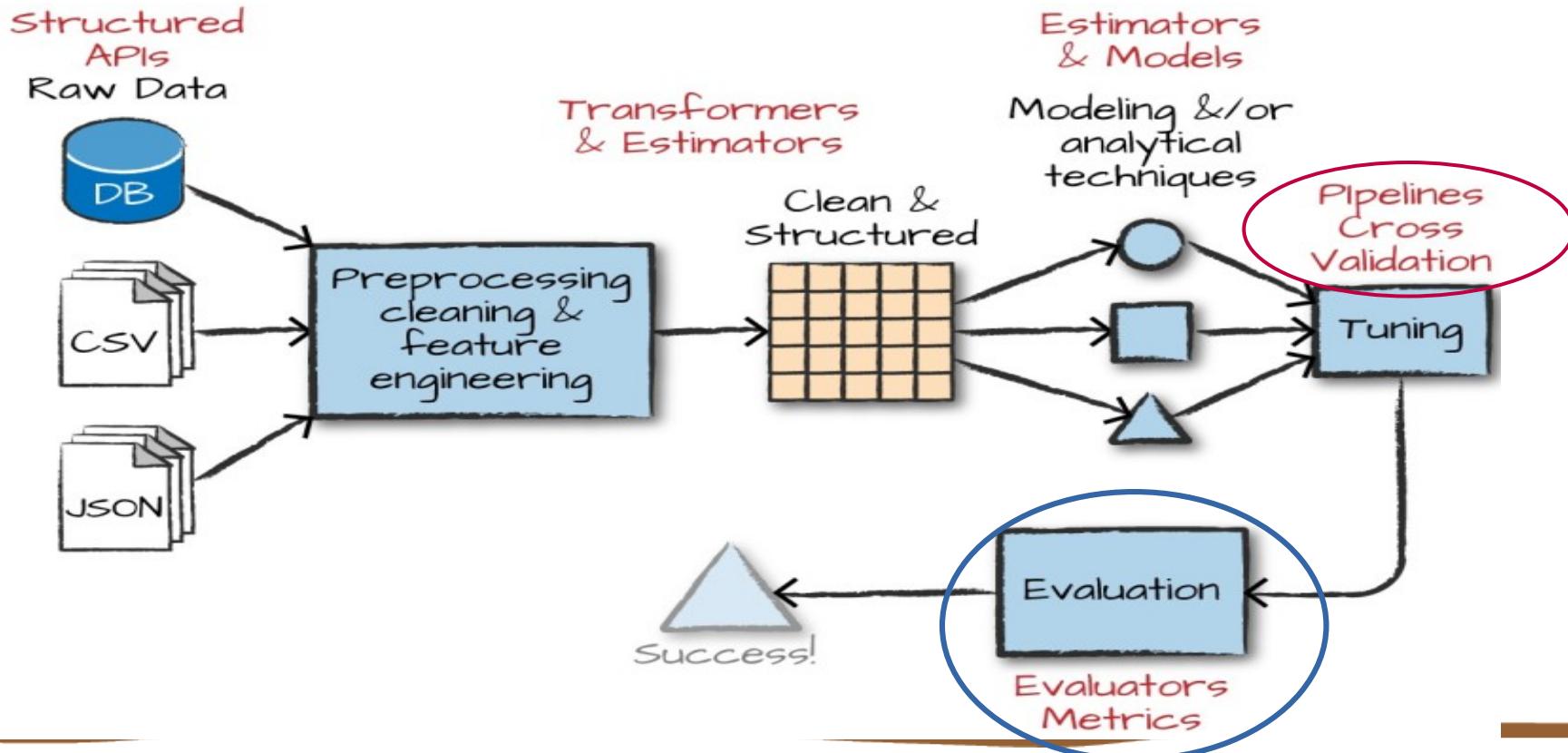
An evaluator allows us to see how a given model performs according to criteria we specify like a **receiver operating characteristic (ROC) curve**.

After we use an evaluator to select the best model from the ones we tested, we can then use that model to make predictions.

# Pipeline



From a high level we can specify each of the **transformations**, **estimations**, and **evaluations** one by one, as steps in a pipeline.





# MLlib in Action

let's create a simple pipeline to demonstrate each of the components. We'll use a small synthetic dataset that will help illustrate our point.

Let's read the data in and see a sample – it's health data.(Categorical)

Health state

True Health

```
In [11]: jdf = spark.read.json("/home/ilg/Documents/sparkdata/data/simple-ml")
In [12]: jdf.orderBy("value2").show()
```

color	lab	value1	value2
green	good	1	14.386294994851129
green	bad	16	14.386294994851129
blue	bad	8	14.386294994851129
blue	bad	8	14.386294994851129
blue	bad	12	14.386294994851129
green	bad	16	14.386294994851129
green	good	12	14.386294994851129
red	good	35	14.386294994851129
red	good	35	14.386294994851129
red	bad	2	14.386294994851129
red	bad	16	14.386294994851129
red	bad	16	14.386294994851129
blue	bad	8	14.386294994851129
green	good	1	14.386294994851129
green	good	12	14.386294994851129
blue	bad	8	14.386294994851129
red	good	35	14.386294994851129
blue	bad	12	14.386294994851129
red	bad	16	14.386294994851129
green	good	12	14.386294994851129

only showing top 20 rows

numerical measures of activity  
within an application e.g  
minutes spent on site and purchases



Suppose that we want to train a classification model where we hope to predict a binary variable—the label—from the other values.

# Feature Engineering with Transformers



When we use MLlib, all inputs to machine learning algorithms in Spark must consist of type **Double (for labels)** and **Vector[Double]** (for features).

The current dataset does not meet that requirement and therefore we need to transform it to the proper format.

How?

To achieve this in our example, we are going to specify an **RFormula**.

This is a declarative language for specifying machine learning transformations and is simple to use.

# Rformula operators



- ~ Separate target and terms
- + Concat terms; "+ 0" means removing the intercept (this means that the y- intercept of the line that we will fit will be 0)
- Remove a term; "- 1" means removing the intercept (this means that the y- intercept of the line that we will fit will be 0—yes, this does the same thing as "+ 0")
- :
- .
- : Interaction (multiplication for numeric values, or binarized categorical values)
- .
- All columns except the target/dependent variable



In order to specify transformations with this syntax,  
we need to import the relevant class.

```
In [13]: from pyspark.ml.feature import RFormula  
In [14]: jsupervised = RFormula(formula="lab ~ . + color:value1 + color:value2")
```

## Rformula, fit, trained version



The next step is to fit the RFormula transformer to the data to let it discover the possible values of each column.

Rformula will automatically handle categorical variables for us, it needs to determine which columns are categorical and which are not, as well as what the distinct values of the categorical columns are.

We use **fit** method , Once we call fit, it returns a “trained” version of our transformer we can then use to actually transform our data.



```
In [13]: from pyspark.ml.feature import RFormula
```

```
In [14]: jsupervised = RFormula(formula="lab ~ . + color:value1 + color:value2")
```

```
In [15]: jfittedRF = jsupervised.fit(jdf)
```

```
In [16]: jprepDF = jfittedRF.transform(jdf)
```

column features that has  
our previously raw data.

```
In [17]: jprepDF.show()
```

color	lab	value1	value2	features	label
green	good	1	14.386294994851129	(10,[1,2,3,5,8],[...])	1.0
blue	bad	8	14.386294994851129	(10,[2,3,6,9],[8,...])	0.0
blue	bad	12	14.386294994851129	(10,[2,3,6,9],[12,...])	0.0
green	good	15	38.97187133755819	(10,[1,2,3,5,8],[...])	1.0
green	good	12	14.386294994851129	(10,[1,2,3,5,8],[...])	1.0
green	bad	16	14.386294994851129	(10,[1,2,3,5,8],[...])	0.0

We (pre)processed the data and added some features along the way. Now it is time to actually train a model (or a set of models) on this dataset.

*In order to do this, we first need to prepare a test set for evaluation.*



Let's create a simple test set based off a random split of the data now.

```
In [19]: jtrain,jtest = jprepDF.randomSplit([0.7,0.3])
```

# Estimators



As we have transformed the data into correct format and valuable features, it's time to actually fit our model.

In this case we will use a ***classification algorithm called logistic regression.***

To create our classifier we instantiate an instance of LogisticRegression, using the default configuration or hyperparameters.

We then set the label columns and the feature columns.

## Note:

The column names we are setting—label and features—are actually the default labels for all estimators in Spark MLlib



let's inspect the parameters

it shows an explanation of all of the parameters for Spark's implementation of logistic regression.

```
In [20]: from pyspark.ml.classification import LogisticRegression  
  
In [21]: jlr = LogisticRegression(labelCol="label", featuresCol="features")  
  
In [22]: print(jlr.explainParams())  
aggregationDepth: suggested depth for treeAggregate (>= 2). (default: 2)  
elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the penalty is an L2 pen-  
alty. For alpha = 1, it is an L1 penalty. (default: 0.0)  
family: The name of family which is a description of the label distribution to be used in the model. Suppo-  
rted options: auto, binomial, multinomial (default: auto)  
featuresCol: features column name. (default: features, current: features)  
fitIntercept: whether to fit an intercept term. (default: True)
```

Note:

The **explainParams** method exists on all algorithms available in MLlib.



Upon instantiating an untrained algorithm, it becomes time to fit it to data. In this case, this returns a LogisticRegressionModel:

```
In [28]: jfittedLR = jlr.fit(jtrain)
```

```
In [29]: jfittedLR.transform(jtrain).select("label","prediction").show()
```

label	prediction
0.0	0.0
0.0	0.0
0.0	0.0
0.0	0.0
0.0	0.0
0.0	0.0
0.0	0.0



Our next step would be to manually evaluate this model and calculate performance metrics like the true positive rate, false negative rate, and so on.

Spark helps you avoid manually trying different models and evaluation criteria by allowing you to specify your workload as a ***declarative pipeline*** of work that includes all your transformations as well as tuning your **hyperparameters**.

## What are Hyperparameters ?

Hyperparameters are configuration parameters that affect the training process, such as model architecture and regularization. They are set prior to starting training.

For instance,

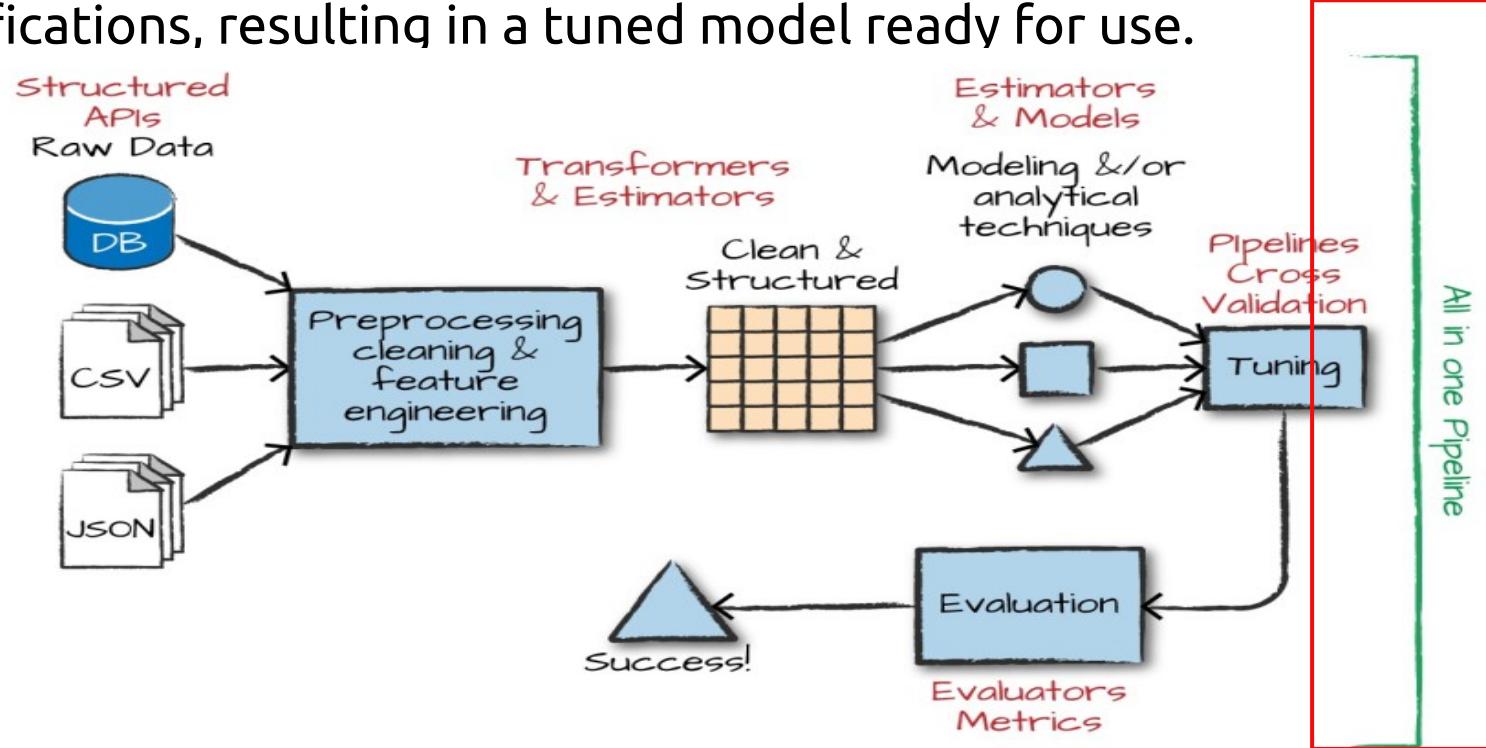
logistic regression has a hyperparameter that *determines how much regularization should be performed on our data through the training phase* (regularization is a technique that pushes models against overfitting data).

# Pipelining Our Workflow



Its tedious to keep track of dataframes, so spark pipelines.

A pipeline allows you to set up a dataflow of the relevant transformations that ends with an estimator that is automatically tuned according to your specifications, resulting in a tuned model ready for use.





**Note:**

it is essential that instances of transformers or models are not reused across different pipelines.

Always create a new instance of a model before creating another pipeline.



In order to make sure we don't overfit, we are going to create a holdout test set and tune our hyperparameters based on a validation set

Note:

we create this validation set based on the original dataset, not the preparedDF used in the previous pages

```
In [30]: jtrain,jtest = jdf.randomSplit([0.7,0.3])
```



Now that you have a holdout set,  
let's create the base stages in our pipeline.

A stage simply represents a transformer or an estimator.

In our case, we will have two estimators.

The RFomula will first analyze our data to understand the types of input features and then transform them to create new features.

Subsequently, the LogisticRegression object is the algorithm that we will train to produce a model



## Pipelining

```
In [31]: jrForm = RFormula()  
In [32]: jlr = LogisticRegression().setLabelCol("label").setFeaturesCol("features")  
In [33]: from pyspark.ml import Pipeline  
In [34]: jstages = [jrForm,jlr]  
In [35]: jpipeline = Pipeline().setStages(jstages)
```

# Training and Evaluation



Now that we arranged the logical pipeline.

We will train several variations of the model by specifying different combinations of hyperparameters that we would like Spark to test.

We will then select the best model using an Evaluator that compares their predictions on our validation data.

We can test different hyperparameters in the entire pipeline, even in the RFormula that we use to manipulate the raw data.



In our current parameter grid, there are three hyperparameters that will diverge from the defaults:

- Two different versions of the Rformula
- Three different options for the ElasticNet parameter
- Two different options for the regularization parameter

This gives us a total of 12 different combinations of these parameters, which means we will be training 12 different versions of logistic regression.

```
In [36]: from pyspark.ml.tuning import ParamGridBuilder  
  
In [37]: jparams = ParamGridBuilder()\\" data-bbox="137 636 807 915">  
.... .addGrid(jrForm.formula,[  
.... "lab ~ . + color:value1",  
.... "lab ~ . + color:value1 + color:value2"])\\" data-bbox="137 636 807 915">  
.... .addGrid(jlr.elasticNetParam,[0.0,0.5,1.0])\\" data-bbox="137 636 807 915">  
.... .addGrid(jlr.regParam,[0.1,2.0])\\" data-bbox="137 636 807 915">  
.... .build()
```



Now that the grid is built, it's time to specify our evaluation process.

The evaluator allows us to automatically and objectively compare multiple models to the same evaluation metric.

We will use the **BinaryClassificationEvaluator**, which has a number of potential evaluation metrics

In this case we will use **areaUnderROC**, which is the total area under the receiver operating characteristic, a common measure of classification performance

```
In [38]: from pyspark.ml.evaluation import BinaryClassificationEvaluator  
  
In [39]: jevaluator = BinaryClassificationEvaluator()\n....: .setMetricName("areaUnderROC")\n....: .setRawPredictionCol("prediction")\n....: .setLabelCol("label")
```



Now that we have a pipeline that specifies how our data should be transformed.

We will perform model selection to try out different hyperparameters in our logistic regression model and measure success by comparing their performance using the areaUnderROC metric.

# Best Practice



it is a best practice in machine learning to fit hyperparameters on a validation set(instead of our test set) to prevent overfitting.

For this reason, we cannot use our holdout test set (that we created before) to tune these parameters.

Spark provides two options for performing hyperparameter tuning automatically.

We can use

**TrainValidationSplit**, which will simply perform an arbitrary random split of our data into two different groups, or

**CrossValidator**, which performs K-fold cross-validation by splitting the dataset into k non-overlapping, randomly partitioned folds



Let's run the entire pipeline we constructed.

To review, running this pipeline will test out every version of the model against the validation set.

Note the type of `tvsFitted` is `TrainValidationSplitModel`.

Any time we fit a given model, it outputs a "model" type

```
In [42]: from pyspark.ml.tuning import TrainValidationSplit
```

```
In [43]: jtvs = TrainValidationSplit()\
....: .setTrainRatio(0.75) \
....: .setEstimatorParamMaps(jparams) \
....: .setEstimator(jpipeline) \
....: .setEvaluator(jevalutor)
```

```
In [47]: jtvsFitted = jtvs.fit(jtrain)
```

```
In [48]: jevalutor.evaluate(jtvsFitted.transform(jtest))  
Out[48]: 0.9318181818181819
```

And of course evaluate how it performs on the test set!



# Preprocessing and Feature Engineering

# Formatting Models According to Your Use Case



To preprocess data for Spark's different advanced analytics tools, we must consider your end objective

- 1) In the case of most **classification** and **regression** algorithms, we want to get your data into a column of type Double to represent the label and a column of type Vector (either dense or sparse) to represent the features.
- 2) In the case of **recommendation**, we want to get your data into a column of users, a column of items (say movies or books), and a column of ratings.
- 3) In the case of **unsupervised learning**, a column of type Vector (either dense or sparse) is needed to represent the features.
- 4) In the case of graph analytics, you will want a DataFrame of vertices and a DataFrame of edges.



# Sample DataSets

Before we proceed, we're going to read in several different sample datasets, each is different type and we will manipulate.

```
In [59]: jsales = spark.read.format("csv")\
...: .option("header","true")\
...: .option("inferSchema","true")\
...: .load("/home/ilg/Documents/sparkdata/data/retail-data/by-day/*.csv")\
...: .coalesce(5)\
...: .where("Description IS NOT NULL")
```

```
In [60]: jfakeIntDF = spark.read.parquet("/home/ilg/Documents/sparkdata/data/simple-ml-integers/")  
In [61]: jsimpleDF = spark.read.json("/home/ilg/Documents/sparkdata/data/simple-ml/")  
In [62]: jscaleDF = spark.read.parquet("/home/ilg/Documents/sparkdata/data/simple-ml-scaling/")
```



Let's also check out the first several rows of data in order to better understand what's in the dataset.

```
jsales.cache()
```

```
jsales.show()
```

In [63]:	jsales.cache()
Out[63]:	DataFrame[InvoiceNo: string, StockCode: string, Description: string, Quantity: int, InvoiceDate: timestamp, UnitPrice: double, CustomerID: double, Country: string]
In [64]:	jsales.show()
<pre>+-----+-----+-----+-----+-----+-----+-----+  InvoiceNo StockCode       Description Quantity InvoiceDate UnitPrice CustomerID Country  +-----+-----+-----+-----+-----+-----+-----+   580538  23084  RABBIT NIGHT LIGHT      48 2011-12-05 08:38:00     1.79  14075.0 United Kingdom    580538  23077  DOUGHNUT LIP GLOSS      20 2011-12-05 08:38:00     1.25  14075.0 United Kingdom    580538  22906  12 MESSAGE CARDS ...      24 2011-12-05 08:38:00     1.65  14075.0 United Kingdom    580538  21914  BLUE HARMONICA IN...      24 2011-12-05 08:38:00     1.25  14075.0 United Kingdom    580538  22467  GUMBALL COAT RACK       6 2011-12-05 08:38:00     2.55  14075.0 United Kingdom    580538  21544  SKULLS WATER TRA...      48 2011-12-05 08:38:00     0.85  14075.0 United Kingdom    580538  23126  FELTCRAFT GIRL AM...       8 2011-12-05 08:38:00     4.95  14075.0 United Kingdom </pre>	

NOTE:

It is important to note that we filtered out null values here.

MLlib does not always play nicely with null values at this point in time. This is a frequent cause for problems and errors and a great first step when you are debugging. Improvements are also made with every Spark release to improve algorithm handling of null values.

# Transformers



Transformers are functions that convert raw data in some way.

Double to be input into a model.

Transformers are primarily used in preprocessing or feature generation.

# Estimators for Preprocessing

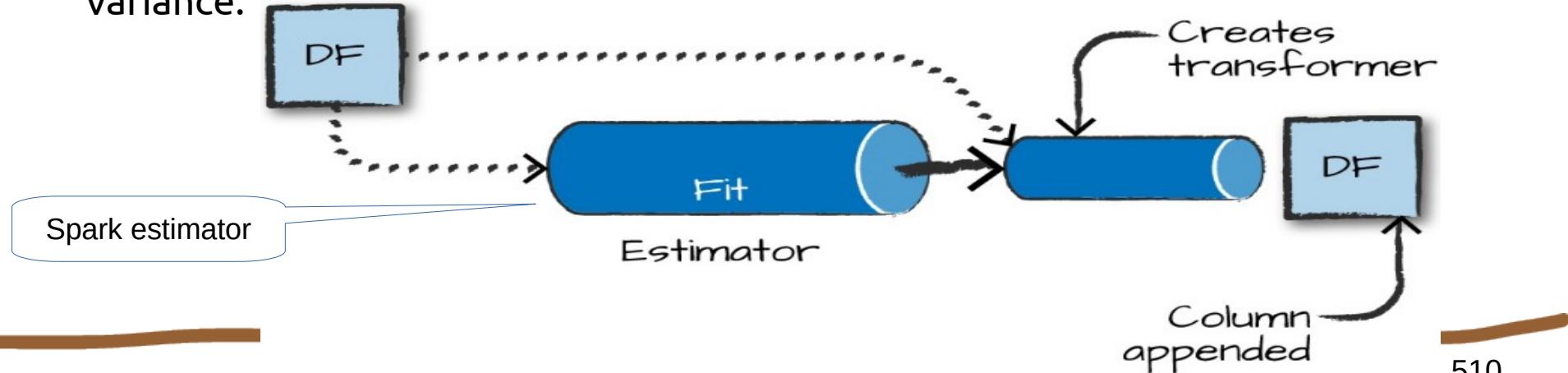


Another tool for preprocessing.

An estimator is necessary when a transformation we would like to perform must be initialized with data or information about the input column (often derived by doing a pass over the input column itself).

For example,

if you wanted to scale the values in our column to have mean zero and unit variance, you would need to perform a pass over the entire data in order to calculate the values you would use to normalize the data to mean zero and unit variance.



# Transformer Properties



All transformers require you to specify, at a minimum, the **inputCol** and the **outputCol**, which represent the column name of the input and output.

You set these with **setInputCol** and **setOutputCol**.

Estimators require you to **fit** the transformer to your particular dataset and then call `transform` on the resulting object.

# High-Level Transformers



## Rformula

allow us to concisely specify a number of transformations in one.

These operate at a “high level”, and allow us to avoid doing data manipulations or transformations one by one

# RFormula



RFormula is the easiest transformer to use when you have “conventionally” formatted data. With this transformer, values can be either **numerical** or **categorical** and we do not need to extract values from strings or manipulate them in any way. The RFormula will automatically handle categorical inputs (specified as strings) by performing something called **one-hot encoding**.

---

Note:

one-hot encoding converts a set of values into a set of binary columns specifying whether or not the data point has each particular value.

Note:

With the RFormula, numeric columns will be cast to Double but will not be one-hot encoded. If the label column is of type String, it will be first transformed to Double with **StringIndexer**.



## Rformula e.g

In this case, we want to use all available variables (**the .**) and then specify an interaction between **value1** and **color** and **value2** and **color** as additional features to generate

```
from pyspark.ml.feature import RFormula  
  
supervised = RFormula(formula="lab ~ . + color:value1 + color:value2")  
supervised.fit(simpleDF).transform(simpleDF).show()
```

color	lab	value1	value2	features	label
green	good	1	14.386294994851129	(10,[1,2,3,5,8],[...])	1.0
blue	bad	8	14.386294994851129	(10,[2,3,6,9],[8,...])	0.0
...					
red	bad	1	38.97187133755819	(10,[0,2,3,4,7],[...])	0.0
red	bad	2	14.386294994851129	(10,[0,2,3,4,7],[...])	0.0



# SQL Transformers

A SQLTransformer allows you to leverage Spark's vast library of SQL-related manipulations just as you would a MLlib transformation.

```
In [65]: from pyspark.ml.feature import SQLTransformer  
  
In [66]: jbasicTransformation = SQLTransformer()\\"  
....: .setStatement("""  
....: select sum(Quantity),count(*),CustomerID from __THIS__  
....: group by CustomerID  
....: """)  
  
In [67]: jbasicTransformation.transform(jsales).show()
```

```
In [67]: jbasicTransformation.transform(jsales).show()  
19/09/13 01:37:09 WARN ObjectStore: Failed to get data  
+-----+-----+-----+  
|sum(Quantity)|count(1)|CustomerID|  
+-----+-----+-----+  
|          119|       62|    14452.0|  
|          440|      143|    16916.0|  
|          630|       72|    17633.0|  
|           34|        6|    14768.0|  
|         1542|       30|    13094.0|  
|          854|      117|    17884.0|
```



# VectorAssembler

The **VectorAssembler** is a tool we'll use in nearly every single pipeline we generate.

It helps **concatenate** all your features into one big vector you can then pass into an estimator.

It's used typically *in the last step of a machine learning pipeline* and takes as input a number of columns of Boolean, Double, or Vector.

This is particularly helpful if you're going to perform a number of manipulations using a variety of transformers and need to gather all of those results together.

```
In [68]: from pyspark.ml.feature import VectorAssembler  
  
In [69]: jva = VectorAssembler().setInputCols(["int1","int2","int3"])  
  
In [70]: jva.transform(jfakeIntDF).show()  
+---+---+---+-----+  
|int1|int2|int3|VectorAssembler_04c64892a175__output|  
+---+---+---+-----+  
| 1 | 2 | 3 | [1.0,2.0,3.0] |  
| 7 | 8 | 9 | [7.0,8.0,9.0] |  
| 4 | 5 | 6 | [4.0,5.0,6.0] |  
+---+---+---+-----+
```

# Working with Continuous Features



Continuous features are just **values** on the *number line, from positive infinity to negative infinity.*

There are two common transformers for continuous features.

- 1) We can convert continuous features into categorical features via a process called bucketing, or
- 2) we can scale and normalize your features according to several different requirements.

```
In [71]: jconfDF = spark.range(20).selectExpr("cast(id as double)")
```

NOTE:

These transformers will only work on Double types, so make sure you've turned any other numerical values to Double

# Working with Continuous Features



```
In [73]: jconfDF.show()
```

id
0.0
1.0
2.0
3.0
4.0
5.0
6.0

# Bucketing



The most straightforward approach to bucketing or binning is using the **Bucketizer**.

This will split a given continuous feature into the buckets of your designation.

*To specify the bucket, set its borders.*

When specifying your bucket points, the values you pass into splits must satisfy three requirements:

- The minimum value in your splits array must be less than the minimum value in your DataFrame.
- The maximum value in your splits array must be greater than the maximum value in your DataFrame.
- We need to specify at a minimum three values in the splits array, which creates two buckets.



# Bucket ranges

In Python we specify this in the following way:

**float("inf"), float("-inf")**

```
In [78]: from pyspark.ml.feature import Bucketizer  
  
In [79]: jbucketBorders = [-1.0,5.0,10.0,250.0,600.0]  
  
In [80]: jbucketer = Bucketizer().setSplits(jbucketBorders).setInputCol("id")  
  
In [81]: jbucketer.transform(jconfDF).show()  
+-----+  
| id|Bucketizer_eb6275200d6b__output|  
+-----+  
| 0.0| 0.0|  
| 1.0| 0.0|  
| 2.0| 0.0|  
| 3.0| 0.0|  
| 4.0| 0.0|  
| 5.0| 1.0|  
| 6.0| 1.0|  
| 7.0| 1.0|
```



# StandardScaler

**StandardScaler** standardizes a set of features to have **zero mean** and a **standard deviation of 1**.

```
In [95]: from pyspark.ml.feature import StandardScaler  
  
In [96]: jsScaler = StandardScaler().setInputCol("features")  
  
In [97]: jsScaler.fit(jscaleDF).transform(jscaleDF).show()  
+-----+-----+  
| id | features|StandardScaler_b80b12d4e199__output|  
+-----+-----+  
| 0 | [1.0,0.1,-1.0] | [1.19522860933439...]|  
| 1 | [2.0,1.1,1.0] | [2.39045721866878...]|  
| 0 | [1.0,0.1,-1.0] | [1.19522860933439...]|  
| 1 | [2.0,1.1,1.0] | [2.39045721866878...]|  
| 1 | [3.0,10.1,3.0] | [3.58568582800318...]|  
+-----+-----+
```



# MinMaxScaler

**MinMaxScaler** will scale the values in a vector (component wise) to the proportional values on a scale from a given min value to a max value.

If you specify the minimum value to be 0 and the maximum value to be 1, then all the values will fall in between 0 and 1

```
In [108]: jfittedminMax.transform(jscaleDF).show()
+-----+
| id |      features |MinMaxScaler_07af21c93cf6__output |
+-----+
| 0 | [1.0,0.1,-1.0] | [5.0,5.0,5.0] |
| 1 | [2.0,1.1,1.0] | [7.5,5.5,7.5] |
| 0 | [1.0,0.1,-1.0] | [5.0,5.0,5.0] |
| 1 | [2.0,1.1,1.0] | [7.5,5.5,7.5] |
| 1 | [3.0,10.1,3.0] | [10.0,10.0,10.0] |
+-----+
```



# MaxAbsScaler

The max absolute scaler (MaxAbsScaler) scales the data by dividing each value by the maximum absolute value in this feature.

All values therefore end up between -1 and 1.

This transformer does not shift or center the data at all in the process

```
In [109]: from pyspark.ml.feature import MaxAbsScaler
In [110]: jmaScaler = MaxAbsScaler().setInputCol("features")
In [111]: jfittedmaScaler = jmaScaler.fit(jscaleDF)
In [112]: jfittedmaScaler.transform(jscaleDF).show()
+-----+-----+
| id |      features |MaxAbsScaler_638e838da6e6__output |
+-----+-----+
|  0 | [1.0,0.1,-1.0] |          [0.333333333333333... |
|  1 | [2.0,1.1,1.0] |          [0.666666666666666... |
|  0 | [1.0,0.1,-1.0] |          [0.333333333333333... |
|  1 | [2.0,1.1,1.0] |          [0.666666666666666... |
|  1 | [3.0,10.1,3.0] |          [1.0,1.0,1.0] |
+-----+-----+
```

# ElementwiseProduct



**ElementwiseProduct** allows us to scale each value in a vector by an arbitrary value.

For example,

Given the vector below and the row “1, 0.1, -1” the output will be “10, 1.5, -20.”

Naturally the dimensions of the scaling vector must match the dimensions of the vector inside the relevant column



# ElementwiseProduct

```
In [115]: from pyspark.ml.feature import ElementwiseProduct
In [116]: from pyspark.ml.linalg import Vectors
In [117]:
In [117]: jscalUpVec = Vectors.dense(10.0,15.0,20.0)
In [118]: jscalingUp = ElementwiseProduct()\
....: .setScalingVec(jscalUpVec)\n....: .setInputCol("features")
In [119]: jscalingUp.transform(jscaleDF).show()
+-----+-----+
| id |      features|ElementwiseProduct_02b7e6020e0a__output|
+-----+-----+
| 0 | [1.0,0.1,-1.0] | [10.0,1.5,-20.0] |
| 1 | [2.0,1.1,1.0] | [20.0,16.5,20.0] |
| 0 | [1.0,0.1,-1.0] | [10.0,1.5,-20.0] |
| 1 | [2.0,1.1,1.0] | [20.0,16.5,20.0] |
| 1 | [3.0,10.1,3.0] | [30.0,151.5,60.0] |
+-----+-----+
```

# Normalizer



**normalizer** allows us to scale multidimensional vectors using one of several power norms, set through the parameter “p”.

For example,

We can use the Manhattan norm (or Manhattan distance) with p = 1, Euclidean norm with p = 2, and so on.

The Manhattan distance is a measure of distance where we can only travel from point to point along the straight lines of an axis (like the streets)

```
In [120]: from pyspark.ml.feature import Normalizer  
  
In [121]: jmanhattanDistance = Normalizer().setP(1).setInputCol("features")  
  
In [122]: jmanhattanDistance.transform(jscaleDF).show()  
+-----+-----+  
| id | features|Normalizer_0a63b46cc19a__output|  
+-----+-----+  
| 0 | [1.0,0.1,-1.0] | [0.47619047619047...]|  
| 1 | [2.0,1.1,1.0] | [0.48780487804878...]|  
| 0 | [1.0,0.1,-1.0] | [0.47619047619047...]|  
| 1 | [2.0,1.1,1.0] | [0.48780487804878...]|  
| 1 | [3.0,10.1,3.0] | [0.18633540372670...]|  
+-----+-----+
```



# Working with Categorical Features



Common task for categorical features is indexing.

Indexing converts a **categorical variable** in a column to a **numerical one** that you can plug into machine learning algorithms.

## StringIndexer

```
In [123]: from pyspark.ml.feature import StringIndexer
In [124]: jlbIndxr = StringIndexer().setInputCol("lab").setOutputCol("labelInd")
In [125]: jidxRes = jlbIndxr.fit(jsimpleDF).transform(jsimpleDF)
In [126]: jidxRes.show()
+---+---+-----+-----+
|color| lab|value1|           value2|labelInd|
+---+---+-----+-----+
|green|good|      1|14.386294994851129|     1.0|
| blue| bad|      8|14.386294994851129|     0.0|
| blue| bad|     12|14.386294994851129|     0.0|
|green|good|     15| 38.97187133755819|     1.0|
|green|good|     12|14.386294994851129|     1.0|
|green| bad|     16|14.386294994851129|     0.0|
```



We can also apply **StringIndexer** to columns that are not strings, in which case, they will be converted to strings before being indexed.

```
In [127]: from pyspark.ml.feature import StringIndexer  
  
In [128]: jvalIndexer = StringIndexer().setInputCol("value1").setOutputCol("valueInd")  
  
In [129]: jvalIndexer.fit(jsimpleDF).transform(jsimpleDF).show()  
+-----+-----+-----+-----+  
|color| lab|value1|      value2|valueInd|  
+-----+-----+-----+-----+  
|green| good|    1|14.386294994851129|    2.0|  
| blue|  bad|    8|14.386294994851129|    4.0|  
| blue|  bad|   12|14.386294994851129|    0.0|  
|green| good|   15| 38.97187133755819|    5.0|  
|green| good|   12|14.386294994851129|    0.0|  
|green|  bad|   16|14.386294994851129|    1.0|  
| red| good|   35|14.386294994851129|    6.0|  
| red|  bad|    1| 38.97187133755819|    2.0|
```

Keep in mind that the **StringIndexer** is an estimator that must be fit on the input data.



# Converting Indexed Values Back to Text

## IndexToString

When inspecting your machine learning results, you're likely going to want to map back to the original values.

Since MLlib classification models make predictions using the indexed values, this conversion is useful for converting model predictions (indices) back to the original categories. We can do this with `IndexToString`.

We notice that we do not have to input our value to the `String` key; Spark's MLlib maintains this metadata for you.

```
In [130]: from pyspark.ml.feature import IndexToString  
  
In [131]: jlabelReverse = IndexToString().setInputCol("labelInd")  
  
In [132]: jlabelReverse.transform(jidxRes).show()  
+-----+-----+-----+-----+-----+-----+  
|color| lab|value1|      value2|labelInd|IndexToString_5cd9500ba9bd__output|  
+-----+-----+-----+-----+-----+-----+  
|green| good|    1|14.386294994851129|    1.0|          good|  
| blue|  bad|    8|14.386294994851129|    0.0|          bad|  
| blue|  bad|   12|14.386294994851129|    0.0|          bad|  
|green| good|   15| 38.97187133755819|    1.0|          good|  
|green| good|   12|14.386294994851129|    1.0|          good|  
|green|  bad|   16|14.386294994851129|    0.0|          bad|  
| red|good|   35|14.386294994851129|    1.0|          good|
```



## Indexing in Vectors

**VectorIndexer** is a helpful tool for working with categorical variables that are already found inside of vectors in your dataset.

This tool will automatically find categorical features inside of your input vectors and convert them to categorical features with zero-based category indices.

For example,

In the following DataFrame, the first column in our Vector is a categorical variable with two different categories while the rest of the variables are continuous.

By setting maxCategories to 2 in our VectorIndexer, we are instructing Spark to take any column in our vector with two or less distinct values and convert it to a categorical variable.



```
In [133]: from pyspark.ml.feature import VectorIndexer
In [134]: from pyspark.ml.linalg import Vectors
In [135]: jidxIn = spark.createDataFrame([
    ...:     (Vectors.dense(1,2,3),1),
    ...:     (Vectors.dense(2,5,6),2),
    ...:     (Vectors.dense(1,8,9),3)
    ...: ]).toDF("features","label")
In [136]: jindxr = VectorIndexer()\
    ...: .setInputCol("features")\
    ...: .setOutputCol("idxed")\
    ...: .setMaxCategories(2)
In [137]: jindxr.fit(jidxIn).transform(jidxIn).show()
+-----+-----+
|   features|label|      idxed|
+-----+-----+
|[1.0,2.0,3.0]|    1|[0.0,2.0,3.0]|
|[2.0,5.0,6.0]|    2|[1.0,5.0,6.0]|
|[1.0,8.0,9.0]|    3|[0.0,8.0,9.0]|
+-----+-----+
```



# One-Hot Encoding

One-hot encoding is an extremely common data transformation performed after indexing categorical variables.

This is because indexing does not always represent our categorical variables in the correct way for downstream models to process.

For instance,

When we index our “color” column, you will notice that some colors have a higher value (or index number) than others (in our case, blue is 1 and green is 2)

This is incorrect because it gives the mathematical appearance that the input to the machine learning algorithm seems to specify that **green > blue**, which makes no sense in the case of the current categories.



To avoid this, we use **OneHotEncoder**, which will convert each distinct value to a Boolean flag (1 or 0) as a component in a vector.

When we encode the color value, then we can see these are no longer ordered, making them easier for downstream models

```
In [138]: from pyspark.ml.feature import OneHotEncoder, StringIndexer  
  
In [139]: jlblIndxr = StringIndexer().setInputCol("color").setOutputCol("colorInd")  
  
In [140]: jcolorLab = jlblIndxr.fit(jsimpleDF).transform(jsimpleDF.select("color"))  
  
In [141]: johe = OneHotEncoder().setInputCol("colorInd")
```

```
In [143]: johe.transform(jcolorLab).show()  
+-----+  
|color|colorInd|OneHotEncoder_33eca37b8fc0__output|  
+-----+  
|green|    1.0|          (2,[1],[1.0])|  
|blue |    2.0|          (2,[],[])|  
|blue |    2.0|          (2,[],[])|  
|green|    1.0|          (2,[1],[1.0])|  
|green|    1.0|          (2,[1],[1.0])|  
|green|    1.0|          (2,[1],[1.0])|  
|red  |    0.0|          (2,[0],[1.0])|  
|red  |    0.0|          (2,[0],[1.0])|  
|red  |    0.0|          (2,[0],[1.0])|  
|red  |    0.0|          (2,[0],[1.0])|
```

# Text Data Transformers



## Tokenizing Text

Tokenization is the process of converting free-form text into a list of “tokens” or individual words.

The easiest way to do this is by using the **Tokenizer** class.

This transformer will take a string of words, separated by whitespace, and convert them into an array of words.

```
In [144]: from pyspark.ml.feature import Tokenizer  
  
In [145]: jtkn = Tokenizer().setInputCol("Description").setOutputCol("DesOut")  
  
In [146]: jotkenized = jtkn.transform(jsales.select("Description"))  
  
In [147]: jotkenized.show(20, False)  
+-----+-----+  
|Description|DesOut|  
+-----+-----+  
|RABBIT NIGHT LIGHT|[rabbit, night, light]  
|DOUGHNUT LIP GLOSS|[doughnut, lip, gloss]  
|12 MESSAGE CARDS WITH ENVELOPES|[12, message, cards, with, envelopes]  
|BLUE HARMONICA IN BOX|[blue, harmonica, in, box]  
|GUMBALL COAT RACK|[gumball, coat, rack]  
|SKULLS WATER TRANSFER TATTOOS|[skulls, , water, transfer, tattoos]  
|FELTCRAFT GIRL AMELIE KIT|[feltcraft, girl, amelie, kit]  
|CAMOUFLAGE LED TORCH|[camouflage, led, torch]  
|WHITE SKULL HOT WATER BOTTLE|[white, skull, hot, water, bottle]  
|ENGLISH ROSE HOT WATER BOTTLE|[english, rose, hot, water, bottle]  
|HOT WATER BOTTLE KEEP CALM|[hot, water, bottle, keep, calm]  
|SCOTTIE DOG HOT WATER BOTTLE|[scottie, dog, hot, water, bottle]
```



# Regex based tokens

The format of the regular expression should conform to the Java Regular Expression (RegEx) syntax

```
In [154]: from pyspark.ml.feature import RegexTokenizer

In [155]: rt = RegexTokenizer()\
...: .setInputCol("Description")\
...: .setOutputCol("DescOut")\
...: .setPattern(" ")\
...: .setToLowercase(True)

In [156]: rt.transform(jsales.select("Description")).show(20, False)
+-----+-----+
|Description|DescOut
+-----+-----+
|RABBIT NIGHT LIGHT|[rabbit, night, light]
|DOUGHNUT LIP GLOSS|[doughnut, lip, gloss]
|12 MESSAGE CARDS WITH ENVELOPES|[12, message, cards, with, envelopes]
|BLUE HARMONICA IN BOX|[blue, harmonica, in, box]
|GUMBALL COAT RACK|[gumball, coat, rack]
|SKULLS WATER TRANSFER TATTOOS|[skulls, water, transfer, tattoos]
|FELTCRAFT GIRL AMELIE KIT|[feltcraft, girl, amelie, kit]
|CAMOUFLAGE LED TORCH|[camouflage, led, torch]
|WHITE SKULL HOT WATER BOTTLE|[white, skull, hot, water, bottle]
```

# Removing Common Words



A common task after **tokenization** is to **filter stop words**, common words that are not relevant in many kinds of analysis and should thus be removed.

Frequently occurring stop words in English include “**the**,” “**and**,” and “**but**.”

Spark contains a list of default stop words you can see by calling the following method, which can be made case insensitive if necessary.

Spark support many other languages too.



Notice how the word of is removed in the output column

```
In [161]: from pyspark.ml.feature import StopWordsRemover  
In [162]: jstops = StopWordsRemover()\n....: .setStopWords(jenglishStopWords)\n....: .setInputCol("DesOut")  
In [163]: jstops.transform(jotkenized).show()  
+-----+-----+-----+  
| Description | DesOut | StopWordsRemover_31ee4a75b7b8__output |  
+-----+-----+-----+  
| RABBIT NIGHT LIGHT | [rabbit, night, l...] | [rabbit, night, l...] |  
| DOUGHNUT LIP GLOSS | [doughnut, lip, g...] | [doughnut, lip, g...] |  
| 12 MESSAGE CARDS ... | [12, message, car...] | [12, message, car...] |  
| BLUE HARMONICA IN... | [blue, harmonica,...] | [blue, harmonica,...] |  
| GUMBALL COAT RACK | [gumball, coat, r...] | [gumball, coat, r...] |
```

# Creating Word Combinations



It is often of interest to look at combinations of words, usually by looking at colocated words.

Word combinations are technically referred to as n-grams—that is, sequences of words of length n.

An n-gram of length 1 is called a unigrams;

those of length 2 are called bigrams,

those of length 3 are called trigrams.

For Instance:

Bigrams of “**Spark Processing made simple**” are :-

“**Spark Processing**”, “**Processing made**”, “**made simple**”



With n-grams, we can look at sequences of words that commonly co-occur and use them as inputs to a machine learning algorithm.

```
# in Python
from pyspark.ml.feature import NGram
unigram = NGram().setInputCol("DescOut").setN(1)
bigram = NGram().setInputCol("DescOut").setN(2)
unigram.transform(tokenized.select("DescOut")).show(False)
bigram.transform(tokenized.select("DescOut")).show(False)
```

```
+-----+-----+
DescOut           |ngram_104c4da6a01b__output   ...
+-----+-----+
|[rabbit, night, light] |[rabbit, night, light]   ...
|[doughnut, lip, gloss] |[doughnut, lip, gloss]   ...
...
|[airline, bag, vintage, world, champion] |[airline, bag, vintage, world, cha...
|[airline, bag, vintage, jet, set, brown] |[airline, bag, vintage, jet, set, ...
+-----+-----+
```

# Converting Words into Numerical Representations



it's time to start counting instances of words and word combinations for use in our models.

The simplest way is just to include binary counts of a word in a given document (in our case, a row).

Essentially, we're measuring whether or not each row contains a given word.

*This is a simple way to normalize for document sizes and occurrence counts and get numerical features that allow us to classify documents based on content.*



A CountVectorizer operates on our tokenized data and does two things:

1. During the fit process, it finds the set of words in all the documents and then counts the occurrences of those words in those documents.
2. It then counts the occurrences of a given word in each row of the DataFrame column during the transformation process and outputs a vector with the terms that occur in that row.

Conceptually this transformer treats every row as a document and every word as a term and the total collection of all terms as the vocabulary.



```
In [195]: from pyspark.ml.feature import CountVectorizer  
  
In [196]: jcv = CountVectorizer()\  
....: .setInputCol("DesOut")\  
....: .setOutputCol("countVec")\  
....: .setVocabSize(500) \  
....: .setMinTF(1) \  
....: .setMinDF(2)  
  
In [197]: jfittedCV = jcv.fit(jotkenized)  
  
In [198]: jfittedCV.transform(jotkenized).show(False)
```

```
+-----+  
| DescOut          | countVec  
+-----+  
|[rabbit, night, light] |(500,[150,185,212],[1.0,1.0,1.0])  
|[doughnut, lip, gloss]|(500,[462,463,492],[1.0,1.0,1.0])  
...  
|[airline, bag, vintage, world,...|(500,[2,6,328],[1.0,1.0,1.0])  
|[airline, bag, vintage, jet, s...|(500,[0,2,6,328,405],[1.0,1.0,1.0,1.0,1.0])  
+-----+
```

# Term frequency–inverse document frequency



TF-IDF measures how often a word occurs in each document, weighted according to how many documents that word occurs in.

The result is that words that occur in a few documents are given more weight than words that occur in many documents.

In practice,

a word like “the” would be weighted very low because of its prevalence while a more specialized word like “streaming” would occur in fewer documents and thus would be weighted higher.



```
In [199]: jtfidfin = jotkenized\
....: .where("array_contains(DesOut,'red')")\
....: .select("DesOut")\
....: .limit(10)

In [200]: jtfidfin.show(10, False)
+-----+
| DesOut
+-----+
|[gingham, heart, , doorstop, red]
|[red, floral, feltcraft, shoulder, bag]
|[alarm, clock, bakelike, red]
|[pin, cushion, babushka, red]
|[red, retrospot, mini, cases]
|[red, kitchen, scales]
|[gingham, heart, , doorstop, red]
|[large, red, babushka, notebook]
|[red, retrospot, oven, glove]
|[red, retrospot, plate]
+-----+
```



# Putting the output through TF-IDF

Now let's input that into TF-IDF.

To do this,

we're going to hash each word and convert it to a numerical representation, and then weigh each word in the vocabulary according to the inverse document frequency.

```
In [212]: from pyspark.ml.feature import HashingTF, IDF  
  
In [213]: jtf = HashingTF()\n        .setInputCol("DesOut")\n        .setOutputCol("TFOut")\n        .setNumFeatures(10000)  
  
In [214]: jidf = IDF()\n        .setInputCol("TFOut")\n        .setOutputCol("IDFout")\n        .setMinDocFreq(2)  
  
In [215]: jidf.fit(jtf.transform(jtfidfin)).transform(jtf.transform(jtfidfin)).show(10, False)
```



```
In [215]: jidf.fit(jtf.transform(jtfidfin)).transform(jtf.transform(jtfidfin)).show(10, False)
```

```
+-----+-----+-----+
|DesOut          |TFOut           |IDFout
+-----+-----+-----+
|[gingham, heart, , doorstop, red]  |(10000,[3372,4291,4370,6594,9160],[1.0,1.0,1.0,1.0,1.0])|(10000,[3372,4291,4370,6594,9160],[1.2992829841302609,0.0,1.2992829841302609,1.2992829841302609,1.2992829841302609])
|[red, floral, feltcraft, shoulder, bag]|(10000,[155,1152,4291,5981,6756],[1.0,1.0,1.0,1.0,1.0])|(10000,[155,1152,4291,5981,6756],[0.0,0.0,0.0,0.0,0.0])
|[alarm, clock, bakelike, red]        |(10000,[4291,4852,4995,9668],[1.0,1.0,1.0,1.0])      |(10000,[4291,4852,4995,9668],[0.0,0.0,0.0,0.0])
|[pin, cushion, babushka, red]       |(10000,[4291,5111,5673,7153],[1.0,1.0,1.0,1.0])      |(10000,[4291,5111,5673,7153],[0.0,0.0,0.0,1.2992829841302609])
```

# Observations



Observe that a certain value is assigned to “red” and that this value appears in every document.

Also note that this term is weighted extremely low because it appears in every document.

The output format is a sparse Vector we can subsequently input into a machine learning model in a form like this:

(10000,[2591,4291,4456],[1.0116009116784799,0.0,0.0])

This vector is represented using three different values:

- 1) total vocabulary size
- 2) hash of every word appearing in the document
- 3) weighting of each of those terms.

# Word2Vec



**Word2Vec** is a deep learning–based tool for computing a vector representation of a set of words.

The goal is to have similar words close to one another in this vector space, so we can then make generalizations about the words themselves.

This model is easy to train and use, and has been shown to be useful in a number of natural language processing applications, including entity recognition, disambiguation, parsing, tagging, and machine translation.

**Word2Vec** is notable for capturing relationships between words based on their semantics

Word2Vec works best with continuous, free-form text in the form of tokens.



```
In [216]: from pyspark.ml.feature import Word2Vec  
  
In [217]: jdocumentDF = spark.createDataFrame([  
    ...:     ("Hi I heard about Spark".split(" ")),  
    ...:     ("I wish Java could use case classes".split(" ")),  
    ...:     ("Logistic regression models are neat and clean".split(" ")),  
    ...:     ("I am learning Spark with Pyspark".split(" ")),  
    ...: ],["text"])  
  
In [218]: jword2Vec = Word2Vec(vectorSize=3,minCount=0,inputCol="text",outputCol="result")  
  
In [219]: jmodel = jword2Vec.fit(jdocumentDF)  
  
In [220]: result = jmodel.transform(jdocumentDF)  
  
In [221]: for row in result.collect():  
    ...:     text,vector = row  
    ...:     print("Text:[%s]==>\nVector:%s\n" % (" ".join(text),str(vector)))
```

```
Text:[Hi, I, heard, about, Spark]==>  
Vector:[-0.014583978056907655,-0.05927877128124237,-0.07684874124825002]  
  
Text:[I, wish, Java, could, use, case, classes]==>  
Vector:[0.008082617839266146,0.01930048157061849,0.013548465445637703]  
  
Text:[Logistic, regression, models, are, neat, and, clean]==>  
Vector:[0.043606125616601536,0.04660506519888128,-0.04343815447230424]  
  
Text:[I, am, learning, Spark, with, Pyspark]==>  
Vector:[0.034617058001458645,-0.011196521421273548,-0.07944053023432691]
```

# Feature Manipulation



## PCA

**Principal Components Analysis (PCA)** is a mathematical technique for finding the most important aspects of our data (the principal components).

It changes the feature representation of our data by creating a new set of features ("aspects").

Each new feature is a combination of the original features.

The power of PCA is that it can create a smaller set of more meaningful features to be input into your model, at the potential cost of interpretability.



```
In [225]: from pyspark.ml.feature import PCA  
In [226]: jPCA = PCA().setInputCol("features").setK(2)  
In [227]: jPCA.fit(jscaleDF).transform(jscaleDF).show(20, False)
```

+-----+ <th>  id   features   PCA_ef627502df5e__output  </th>	id   features   PCA_ef627502df5e__output
0   [1.0,0.1,-1.0]   [0.07137194992484153, -0.45266548881478463]	
1   [2.0,1.1,1.0]   [-1.6804946984073725, 1.2593401322219144]	
0   [1.0,0.1,-1.0]   [0.07137194992484153, -0.45266548881478463]	
1   [2.0,1.1,1.0]   [-1.6804946984073725, 1.2593401322219144]	
1   [3.0,10.1,3.0]   [-10.872398139848944, 0.030962697060149758]	



# Interaction

In some cases, we might have domain knowledge about specific variables in your dataset.

For example,

We might know that a certain interaction between the two variables is an important variable to include in a downstream estimator.

The feature transformer `Interaction` allows us to create an interaction between two variables manually.

It just multiplies the two features together—something that a typical linear model would not do for every possible pair of features in your data.

This transformer is currently only available directly in Scala but can be called from any language using the `RFormula`.

# Polynomial Expansion



Polynomial expansion is used to generate interaction variables of all the input columns. With polynomial expansion, we specify to what degree we would like to see various interactions.

For example,

for a degree-2 polynomial, Spark takes every value in our feature vector, multiplies it by every other value in the feature vector, and then stores the results as features.

For instance,

if we have two input features, we'll get four output features

*if we use a second-degree polynomial ( $2 \times 2$ ). If we have three input features, we'll get nine output features ( $3 \times 3$ ).*

*If we use a third-degree polynomial, we'll get 27 output features ( $3 \times 3 \times 3$ ) and so on.*

This transformation is useful when you want to see interactions between particular features but aren't necessarily sure about which interactions to consider.

# Polynomial Expansion



```
In [228]: from pyspark.ml.feature import PolynomialExpansion  
  
In [229]: jpe = PolynomialExpansion().setInputCol("features").setDegree(2)  
  
In [230]: jpe.transform(jscaleDF).show()  
+-----+-----+  
| id | features|PolynomialExpansion_f77add56ef5f__output|  
+-----+-----+  
| 0 | [1.0,0.1,-1.0] | [1.0,1.0,0.1,0.1,...] |  
| 1 | [2.0,1.1,1.0] | [2.0,4.0,1.1,2.2,...] |  
| 0 | [1.0,0.1,-1.0] | [1.0,1.0,0.1,0.1,...] |  
| 1 | [2.0,1.1,1.0] | [2.0,4.0,1.1,2.2,...] |  
| 1 | [3.0,10.1,3.0] | [3.0,9.0,10.1,30....] |  
+-----+-----+
```

# Feature Selection



Feature selection tries to identify relevant features for use in model construction.

It reduces the size of the feature space, which can improve both speed and statistical learning behaviour

ChiSqSelector implements Chi-Squared feature selection.

It operates on labeled data with categorical features.

ChiSqSelector uses the Chi-Squared test of independence to decide which features to choose.

It supports five selection methods: -



- **numTopFeatures** chooses a fixed number of top features according to a chi-squared test. This is akin to yielding the features with the most predictive power.
- **percentile** is similar to numTopFeatures but chooses a fraction of all features instead of a fixed number.
- **fpr** chooses all features whose p-values are below a threshold, thus controlling the false positive rate of selection.
- **fdr** uses the Benjamini-Hochberg procedure to choose all features whose false discovery rate is below a threshold.
- **fwe** chooses all features whose p-values are below a threshold. The threshold is scaled by 1/numFeatures, thus controlling the family-wise error rate of selection.

**By default, the selection method is numTopFeatures**, with the default number of top features set to 50. *The user can choose a selection method using setSelectorType.*



```
In [249]: from pyspark.ml.feature import ChiSqSelector,Tokenizer
In [250]: jptkn = Tokenizer().setInputCol("Description").setOutputCol("DesOut")
In [251]: jptokenized = jptkn\
....: .transform(jsales.select("Description","CustomerId"))\
....: .where("CustomerId is NOT NULL")
In [252]: jpprechi = jfittedCV.transform(jptokenized)\
....: .where("CustomerId IS NOT NULL")
In [253]: jpchisq = ChiSqSelector()\
....: .setFeaturesCol("countVec")\
....: .setLabelCol("CustomerId")\
....: .setNumTopFeatures(2)
In [254]: jpchisq.fit(jpprechi).transform(jpprechi)\
....: .drop("customerId","Description","DesOut").show()
```

# Classification



Classification is the task of predicting a label, category, class, or discrete variable given some input features.

The key difference from other ML tasks, such as regression, is that the output label has a finite set of possible values

# Use Cases



## Predicting credit risk

- A financing company might look at a number of variables before offering a loan to a company or individual. Whether or not to offer the loan is a binary classification problem.

## News classification

- An algorithm might be trained to predict the topic of a news article (sports, politics, business, etc.).

## Classifying human activity

- By collecting data from sensors such as a phone accelerometer or smart watch, you can predict the person's activity. The output will be one of a finite set of classes (e.g., walking, sleeping, standing, or running).

# Types of Classification



## Binary Classification

The simplest example of classification is binary classification, where there are only two labels you can predict.

An example is fraud analytics, where a given transaction can be classified as fraudulent or not; or email spam, where a given email can be classified as spam or not spam.

# Types of Classification



## Multiclass Classification

Beyond binary classification lies multiclass classification, where one label is chosen from more than two distinct possible labels.

A typical example is Facebook predicting the people in a given photo or a meteorologist predicting the weather (rainy, sunny, cloudy, etc.).

Note how there is always a finite set of classes to predict; it's never unbounded. This is also called multinomial classification.

# Types of Classification



## Multilabel Classification

Finally, there is multilabel classification, where a given input can produce multiple labels.

For example,

we might want to predict a book's genre based on the text of the book itself. While this could be multiclass, it's probably better suited for multilabel because a book may fall into multiple genres.

# Classification Models in MLlib



Spark has several models available for performing binary and multiclass classification out of the box.

- Logistic regression
- Decision trees
- Random forests
- Gradient-boosted trees



# Model Scalability

Model	Features count	Training examples	Output classes
Logistic regression	1 to 10 million	No limit	Features x Classes < 10 million
Decision trees	1,000s	No limit	Features x Classes < 10,000s
Random forest	10,000s	No limit	Features x Classes < 100,000s
Gradient-boosted trees	1,000s	No limit	Features x Classes < 10,000s

# Sample Data



Let's start looking at the classification models by loading in some data:

```
In [256]: cjbInput = spark.read.format("parquet").load("/home/ilg/Documents/sparkdata/data/binary-classifi  
....: cation/")\  
....: .selectExpr("features","cast(label as double) as label")
```



# Logistic Regression



# Logistic Regression

It is a linear method that combines each of the individual inputs (or features) with specific weights (these weights are generated during the training process) that are then combined to get a probability of belonging to a particular class.

These weights are helpful because they are good representations of feature importance;

if you have a **large weight**, we can assume that variations in that feature **have a significant effect on the outcome** (assuming you performed normalization).

A **smaller weight** means the feature is less likely to be important.

# Model Hyperparameters



Model hyperparameters are configurations that determine the basic structure of the model itself.

## **family**

Can be

- 1) multinomial (two or more distinct labels; multiclass classification) or
- 2) binary (only two distinct labels; binary classification)



# Model Hyperparameters

## ElasticNetParam

A floating-point value from 0 to 1.

This parameter specifies the mix of L1 and L2 regularization according to elastic net regularization (which is a linear combination of the two).

Our choice of L1 or L2 depends a lot on your particular use case but the intuition is as follows:

**L1 regularization (a value of 1)** will create sparsity in the model because certain feature weights will become zero (that are of little consequence to the output). For this reason, it can be used as a simple feature-selection method.

**L2 regularization (a value of 0)** does not create sparsity because the corresponding weights for particular features will only be driven toward zero, but will never completely reach zero.

# Model Hyperparameters



## `fitIntercept`

Can be true or false.

This hyperparameter determines whether or not to fit the intercept or the arbitrary number that is added to the linear combination of inputs and weights of the model.

Typically you will want to fit the intercept if we haven't normalized our training data.

# Model Hyperparameters



## **regParam**

A value  $\geq 0$ . that determines how much weight to give to the regularization term in the objective function.

Choosing a value here is again going to be a function of noise and dimensionality in our dataset.

In a pipeline, try a wide range of values (e.g., 0, 0.01, 0.1, 1).

## **standardization**

Can be true or false, whether or not to standardize the inputs before passing them into the model.



# Training Parameters

Training parameters are used to specify how we perform our training.

## **maxIter**

Total number of iterations over the data before stopping. Changing this parameter probably won't change your results a ton, so it shouldn't be the first parameter you look to adjust.

The default is 100.

## **tol**

This value specifies a threshold by which changes in parameters show that we optimized our weights enough, and can stop iterating. It lets the algorithm stop before maxIter iterations.

The default value is 1.0E-6.

This also shouldn't be the first parameter you look to tune.



## **weightCol**

The name of a weight column used to weigh certain rows more than others.

This can be a useful tool if you have some other measure of how important a particular training example is and have a weight associated with it.

For example,

- We might have 10,000 examples where we know that some labels are more accurate than others. we can weigh the labels we know are correct more than the ones you don't.

# Prediction Parameters



These parameters help determine how the model should actually be making predictions at prediction time, but do not affect training.

Prediction parameters for logistic regression :-

## **threshold**

A Double in the range of 0 to 1.

This parameter is the probability threshold for when a given class should be predicted.

We can tune this parameter according to your requirements to balance between false positives and false negatives.

For instance,

if a mistaken prediction would be costly—you might want to make its prediction threshold very high.



## thresholds

This parameter lets you specify an array of threshold values for each class when using multiclass classification.

It works similarly to the single threshold parameter



# E.g

# see all parameters

```
In [257]: from pyspark.ml.classification import LogisticRegression  
In [258]: jplr = LogisticRegression()  
In [259]: print(jplr.explainParams())
```

An example using LogisticRegression model.

```
In [261]: from pyspark.ml.classification import LogisticRegression  
In [262]: jplr = LogisticRegression()  
In [263]: jplrModel = jplr.fit(cjbInput)
```



Once the model is trained you can get information about the model by taking a look at the coefficients and the intercept.

The **coefficients correspond to the individual feature weights** (each feature weight is multiplied by each respective feature to compute the prediction) while the *intercept is the value of the italics-intercept* (if we chose to fit one when specifying the model).

Seeing the coefficients can be helpful for inspecting the model that you built and comparing how features affect the prediction

```
In [264]: print(jplrModel.coefficients)
[6.848741326854929, 0.3535658901019745, 14.814900276915923]

In [265]: print(jplrModel.intercept)
-10.22569586448109
```



# Model Summary

Logistic regression provides a model summary that gives you information about the final, trained model.

```
In [266]: jsummary = jplrModel.summary
```

```
In [267]: print(jsummary.areaUnderROC)
1.0
```

```
In [268]: jsummary.roc.show()
```

FPR	TPR
0.0	0.0
0.0	0.3333333333333333
0.0	1.0
1.0	1.0
1.0	1.0

```
In [269]: jsummary.pr.show()
```

recall	precision
0.0	1.0
0.3333333333333333	1.0
1.0	1.0
1.0	0.6



The speed at which the model descends to the final result is shown in the objective history.

We can access this through the objective history on the model summary.

```
In [270]: jsummary.objectiveHistory  
Out[270]:  
[0.6730116670092563,  
 0.5042829330409727,  
 0.36356862066874396,  
 0.1252407018038337,  
 0.08532556611276214,  
 0.035504876415730455,  
 0.018196494508571255,  
 0.008817369922959136,  
 0.004413673785392143,
```

# Decision Trees



Decision trees are one of the more friendly and interpretable models for performing classification because they're similar to simple decision models that humans use quite often.

For example,

if you have to predict whether or not someone will eat ice cream when offered, a good feature might be whether or not that individual likes ice cream.

In pseudocode,

```
if person.likes("ice_cream"), they will eat ice cream; otherwise, they won't eat ice cream.
```

Note:

It can overfit data extremely quickly.

# Model Hyperparameters – decision trees



## **maxDepth**

Since we're training a tree, it can be helpful to specify a max depth in order to avoid overfitting to the dataset (in the extreme, every row ends up as its own leaf node).

The default is 5.

## **maxBins**

In decision trees, continuous features are converted into categorical features and maxBins determines how many bins should be created from continuous features. More bins gives a higher level of granularity.

The value must be greater than or equal to 2 and greater than or equal to the number of categories in any categorical feature in your dataset.

The default is 32.



## impurity

To build up a “tree” you need to configure when the model should branch.

Impurity represents the metric (information gain) to determine whether or not the model should split at a particular leaf node.

This parameter can be set to either be “entropy” or “gini” (default), two commonly used impurity metrics.



## **minInfoGain**

This parameter determines the minimum information gain that can be used for a split. A higher value can prevent overfitting. This is largely something that needs to be determined from testing out different variations of the decision tree model.

- The default is zero.

## **minInstancePerNode**

This parameter determines the minimum number of training instances that need to end in a particular node. Think of this as another manner of controlling max depth. We can prevent overfitting by limiting depth or we can prevent it by specifying that at minimum a certain number of training values need to end up in a particular leaf node. If it's not met we would “prune” the tree until that requirement is met. A higher value can prevent overfitting.

- The default is 1, but this can be any value greater than 1.



# Training Parameters

## **checkpointInterval**

Checkpointing is a way to save the model's work over the course of training so that if nodes in the cluster crash for some reason, we don't lose your work.

A value of 10 means the model will get checkpointed every 10 iterations.

Set this to -1 to turn off checkpointing.

This parameter needs to be set together with a checkpointDir (a directory to checkpoint to) and with useNodeIdCache=true.

# Prediction Parameters



There is only one prediction parameter for decision trees: **thresholds**.

```
In [271]: from pyspark.ml.classification import DecisionTreeClassifier
In [272]: jdt = DecisionTreeClassifier()
In [273]: print(jdt.explainParams())
cacheNodeIds: If false, the algorithm will pass trees to executors to match instances with nodes. If true,
the algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. Users
can set how often should the cache be checkpointed or disable it by setting checkpointInterval. (default:
False)
```

```
In [274]: jdtModel = jdt.fit(cjbInput)
19/09/13 08:53:26 WARN DecisionTreeMetadata: DecisionTree reducing maxBins from 32 to 5 (= number of train
ing instances)
```



# Random Forest and Gradient-Boosted Trees

# Random Forest and Gradient-Boosted Trees



These methods are extensions of the decision tree.

Rather than training one tree on all of the data, we train multiple trees on varying subsets of the data.

The intuition behind doing this is that various decision trees will become “experts” in that particular domain while others become experts in others.

By combining these various experts, we then get a

**“wisdom of the crowds” effect**, where the group’s performance exceeds any individual.

In addition, these methods can help prevent overfitting.

# Model Hyperparameters



Random forests and gradient-boosted trees provide all of the same model hyperparameters supported by decision trees.

## **Random forest only**

### **numTrees**

The total number of trees to train.

### **featureSubsetStrategy**

This parameter determines how many features should be considered for splits. This can be a variety of different values including “auto”, “all”, “sqrt”, “log2”, or a number “n.” When your input is “n” the model will use  $n * \text{number of features}$  during training.

When n is in the range (1, number of features), the model will use n features during training.

*There's no one-size-fits-all solution here, so it's worth experimenting with different values in your pipeline.*



## **Gradient-boosted trees (GBT) only**

### **lossType**

This is the loss function for gradient-boosted trees to minimize during training. Currently, only logistic loss is supported.

### **maxIter**

Total number of iterations over the data before stopping. Changing this probably won't change your results a ton, so it shouldn't be the first parameter you look to adjust.

The default is 100.

### **stepSize**

This is the **learning rate for the algorithm**. A larger step size means that larger jumps are made between training iterations. This can help in the optimization process and is something that should be tested in training.

The default is 0.1 and this can be any value from 0 to 1.

# Training Parameters



There is only one training parameter for these models,  
**checkpointInterval**.

# Prediction Parameters



These models have the same prediction parameters as decision trees.

```
In [288]: from pyspark.ml.classification import RandomForestClassifier  
In [289]: jrfClassifier = RandomForestClassifier()  
In [290]: print(jrfClassifier.explainParams())  
In [294]: jptrainedModel = jrfClassifier.fit(cjbInput)
```

```
In [290]: from pyspark.ml.classification import GBTClassifier  
In [291]: jgbtClassifier = GBTClassifier()  
In [292]: print(jgbtClassifier.explainParams())  
In [293]: jtrainedModel = jgbtClassifier.fit(cjbInput)
```



# Naive Bayes

# Naive Bayes



Naive Bayes classifiers are a collection of classifiers based on Bayes' theorem.

The core assumption behind the models is that all features in your data are independent of one another. Naturally, strict independence is a bit naive, but even if this is violated, useful models can still be produced.

Naive Bayes classifiers are commonly used in *text or document classification*.

There are two different model types:

- 1) Multivariate Bernoulli model, where indicator variables represent the existence of a term in a document.
- 2) Multinomial model, where the total counts of terms are used.

One important note when it comes to Naive Bayes is that all input features must be non-negative.

# Model Hyperparameters



## **modelType**

Either “bernoulli” or “multinomial.”

## **weightCol**

Allows weighing different data points differently. Refer back to “Training Parameters” for the explanation of this hyperparameter.



# Training Parameters

## **smoothing**

This determines the amount of regularization that should take place using additive smoothing.

This helps smooth out categorical data and avoid overfitting on the training data by changing the expected probability for certain classes.

The default value is 1.

# Prediction Parameters



**Naive Bayes** shares the same prediction parameter, **thresholds**, as all of our other models.

```
In [295]: from pyspark.ml.classification import NaiveBayes
In [296]: jnb = NaiveBayes()
In [297]: jtrainedModel = jnb.fit(cjbInput.where("label != 0"))
In [298]: print(jnb.explainParams)
<bound method Params.explainParams of NaiveBayes_98a47a3891a1>
In [299]: print(jnb.explainParams())
featuresCol: features column name. (default: features)
labelCol: label column name. (default: label)
```



# Evaluators for Classification and Automating Model Tuning

# Evaluators



**evaluators** allow us to specify the metric of success for our model.

when we use it in a pipeline, we can automate a grid search of our various parameters of the models and transformers—trying all combinations of the parameters to see which ones perform the best.

Evaluators are most useful in this pipeline and parameter grid context.

Two evaluators

## 1) **BinaryClassificationEvaluator**

- This supports optimizing for two different metrics “areaUnderROC” and “areaUnderPR.”

## 2) **MulticlassClassificationEvaluator**,

- which supports optimizing for “f1”, “weightedPrecision”, “weightedRecall”, and “accuracy”

# Detailed Evaluation Metrics



There are three different classification metrics we can use:

- Binary classification metrics
- Multiclass classification metrics
- Multilabel classification metrics

```
from pyspark.mllib.evaluation import BinaryClassificationMetrics
out = model.transform(bInput) \
    .select("prediction", "label") \
    .rdd.map(lambda x: (float(x[0]), float(x[1])))
metrics = BinaryClassificationMetrics(out)
```

Once we've done that, we can see typical classification success metrics on this metric's object

```
print metrics.areaUnderPR
print metrics.areaUnderROC
print "Receiver Operating Characteristic"
metrics.roc.toDF().show()
```



# Regression



Regression is a logical extension of classification.

Rather than just predicting a single value from a set of values, *regression is the act of predicting a real number (or continuous variable) from a set of features (represented as numbers)*.

# Use Cases



## Predicting movie viewership

- Given information about a movie and the movie-going public, such as how many people have watched the trailer or shared it on social media, you might want to predict how many people are likely to watch the movie when it comes out.

## Predicting company revenue

- Given a current growth trajectory, the market, and seasonality, you might want to predict how much revenue a company will gain in the future.

## Predicting crop yield

- Given information about the particular area in which a crop is grown, as well as the current weather throughout the year, you might want to predict the total crop yield for a particular plot of land

# Regression Models in MLlib



- Linear regression
- Generalized linear regression
- Isotonic regression
- Decision trees
- Random forest
- Gradient-boosted trees
- Survival regression



# Model Scalability

Model	Number features	Training examples
Linear regression	1 to 10 million	No limit
Generalized linear regression	4,096	No limit
Isotonic regression	N/A	Millions
Decision trees	1,000s	No limit
Random forest	10,000s	No limit
Gradient-boosted trees	1,000s	No limit
Survival regression	1 to 10 million	No limit

# Sample Date



Let's read in some sample data that we will use throughout

```
jdf = spark.read.load("/home/ilg/Documents/sparkdata/data/regression/")
```

# Linear Regression



Linear regression assumes that a *linear combination of your input features (the sum of each feature multiplied by a weight) results along with an amount of Gaussian error in the output.*

This linear assumption (along with Gaussian error) does not always hold true, but it does make for a simple, interpretable model that's hard to overfit.

Spark implements ElasticNet regularization for this, allowing you to mix L1 and L2 regularization.

# Model Hyperparameters



Linear regression has the same model hyperparameters as logistic regression.



# Training Parameters

Linear regression also shares all of the same training parameters from logistic regression.

```
In [310]: jdf = spark.read.load("/home/ilg/Documents/sparkdata/data/regression/")
In [311]: from pyspark.ml.regression import LinearRegression
In [312]: jlr2 = LinearRegression().setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)
In [313]: jlrModel2 = jlr2.fit(jdf)
In [314]: print(jlr2.explainParams())
aggregationDepth: suggested depth for treeAggregate (>= 2). (default: 2)
elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the penalty is an L2 pen-
alty. For alpha = 1, it is an L1 penalty. (default: 0.0, current: 0.8)
epsilon: The shape parameter to control the amount of robustness. Must be > 1.0. Only valid when loss is h
uber (default: 1.35)
featuresCol: features column name. (default: features)
```



# Training Summary

The summary method returns a summary object with several fields

```
In [315]: jsummary2 = jlrModel2.summary  
  
In [316]: jsummary2.residuals.show()  
+-----+  
|      residuals|  
+-----+  
| 0.12805046585610147|  
| -0.14468269261572053|  
| -0.41903832622420595|  
| -0.41903832622420595|  
|  0.8547088792080308|  
+-----+
```

```
In [317]: print(jsummary2.totalIterations)  
6  
  
In [318]: print(jsummary2.objectiveHistory)  
[0.5000000000000001, 0.4315295810362787, 0.3132335933881022, 0.31225692666554117, 0.309150608198303, 0.30915058933480255]  
  
In [319]: print(jsummary2.rootMeanSquaredError)  
0.47308424392175996  
  
In [320]: print(jsummary2.r2)  
0.7202391226912209
```



# Generalized Linear Regression



The standard linear regression that we saw in the previous slides is actually a part of a family of algorithms called **generalized linear regression**.

Spark has two implementations of this algorithm.

1) optimized for working with very large sets of features – simple linear regression.

Family	Response type	Supported links
Gaussian	Continuous	Identity*, Log, Inverse
Binomial	Binary	Logit*, Probit, CLLogLog
Poisson	Count	Log*, Identity, Sqrt
Gamma	Continuous	Inverse*, Identity, Log
Tweedie	Zero-inflated continuous	Power link function

algorithms, but doesn't scale.

Regression families, response types, and link functions

# Model Hyperparameters



## family

- A description of the error distribution to be used in the model. Supported options are Poisson, binomial, gamma, Gaussian, and tweedie.

## link

- The name of link function which provides the relationship between the linear predictor and the mean of the distribution function. Supported options are cloglog, probit, logit, inverse, sqrt, identity, and log (default: identity).

# Model Hyperparameters



## **solver**

- The solver algorithm to be used for optimization. The only currently supported solver is `irls` (iteratively reweighted least squares).

## **variancePower**

- The power in the variance function of the Tweedie distribution, which characterizes the relationship between the variance and mean of the distribution. Only applicable to the Tweedie family. Supported values are 0 and [1, Infinity).

The default is 0.

## **linkPower**

- The index in the power link function for the Tweedie family.

## Training Parameters



The training parameters are the same that you will find for logistic regression

# Prediction Parameters



This model adds one prediction parameter:

## **linkPredictionCol**

- A column name that will hold the output of our link function for each prediction.



## An example of using **GeneralizedLinearRegression**:

```
In [321]: from pyspark.ml.regression import GeneralizedLinearRegression
In [322]: jgblr = GeneralizedLinearRegression()\
....: .setFamily("gaussian")\
....: .setLink("identity")\
....: .setMaxIter(10)\
....: .setRegParam(0.3) \
....: .setLinkPredictionCol("linkOut")
In [323]: jgblrModel = jgblr.fit(jdf)
In [324]: print(jgblr.explainParams())
family: The name of family which is a description of the error distribution to be used in the model. Suppo
rted options: gaussian (default), binomial, poisson, gamma and tweedie. (default: gaussian, current: gauss
ian)
featuresCol: features column name. (default: features)
```



# Training Summary

## R squared

- The coefficient of determination; a measure of fit.

## The residuals

```
In [331]: print(jglr2summary.residuals)
<bound method GeneralizedLinearRegressionSummary.residuals of Coefficients:
    Feature Estimate Std. Error T Value P Value
(Intercept)  0.0867    1.2210   0.0710  0.9549
features_0   0.3661    0.7686   0.4764  0.7170
features_1   0.0466    0.1380   0.3377  0.7927
features_2   0.1831    0.3843   0.4764  0.7170

(Dispersion parameter for gaussian family taken to be 0.8466)
  Null deviance: 4.0000 on 1 degrees of freedom
Residual deviance: 0.8466 on 1 degrees of freedom
AIC: 15.3094>
```

```
In [340]: jglr2summary.coefficientStandardErrors
Out[340]:
[0.7685662635498633,
 0.13798813723773948,
 0.3842831317749312,
 1.2210120959942767]
```



# Decision Trees



Decision trees as applied to regression work fairly similarly to decision trees applied to classification.

The main **difference** is that *decision trees for regression output a single number per leaf node instead of a label* (as we saw with classification).

Decision trees -

Rather than trying to train coefficients to model a function, decision tree regression simply creates a tree to predict the numerical outputs.

This is of significant consequence because unlike generalized linear regression, we can predict nonlinear functions in the input data.

# Model Hyperparameters



The model hyperparameters that apply decision trees for regression are the same as those for classification except for a slight change to the impurity parameter.

## **impurity**

The impurity parameter represents the metric (information gain) for whether or not the model should split at a particular leaf node with a particular value or keep it as is.

The only metric currently supported for regression trees is “variance.”



# Training Parameters

classification and regression trees also share the same training parameters.

```
In [342]: from pyspark.ml.regression import DecisionTreeRegressor
In [343]: jdtr = DecisionTreeRegressor()
In [344]: jdtrModel = jdtr.fit(jdf)
19/09/13 10:27:43 WARN DecisionTreeMetadata: DecisionTree reducing maxBins from 32 to 5 (= number of training instances)
In [345]: print(jdtr.explainParams())
cacheNodeIds: If false, the algorithm will pass trees to executors to match instances with nodes. If true, the algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. Users can set how often should the cache be checkpointed or disable it by setting checkpointInterval. (default: False)
```



# Random Forests and Gradient-Boosted Trees

# Random Forests and Gradient-Boosted Trees



The random forest and gradient-boosted tree models can be applied to both classification and regression.

These both follow the same basic concept as the decision tree, except rather than training one tree, many trees are trained to perform a regression.

In the random forest model, many de-correlated trees are trained and then averaged.

With gradient-boosted trees, each tree makes a weighted prediction (such that some trees have more predictive power for some classes over others).

Random forest and gradient-boosted tree regression have the same model hyperparameters and training parameters as the corresponding classification models, except for the purity measure

## Model Hyperparameters



These models share many of the same parameters as we saw in the previous slides as well as for regression decision trees.

As for a single regression tree, however, the only impurity metric currently supported is variance.

# Training Parameters



These models support the same **checkpointInterval** parameter as classification trees



```
In [346]: from pyspark.ml.regression import RandomForestRegressor
In [347]: from pyspark.ml.regression import GBTRRegressor
In [348]: jrf3 = RandomForestRegressor()
In [349]: jrf3Model = jrf3.fit(jdf)
19/09/13 10:34:47 WARN DecisionTreeMetadata: DecisionTree reducing maxBins from 32 to 5 (= number of training instances)
In [350]: jgbt3 = GBTRRegressor()
In [351]: jgbt3Model = jgbt3.fit(jdf)
19/09/13 10:35:42 WARN DecisionTreeMetadata: DecisionTree reducing maxBins from 32 to 5 (= number of training instances)
In [352]: print(jrf3.explainParams())
In [353]: print(jgbt3.fit(jdf))
[...]
GBTRegressionModel (uid=GBTRegressor_55e55a0c0c3e) with 20 trees
```



## Advanced Methods

# Survival Regression (Accelerated Failure Time)



In spark.ml, we implement the Accelerated failure time (AFT) model which is a parametric survival regression model for censored data.

It describes a model for the **log of survival time**, so it's often called a log-linear model for survival analysis.

Different from a Proportional hazards model designed for the same purpose, the AFT model is easier to parallelize because each instance contributes to the objective function independently.

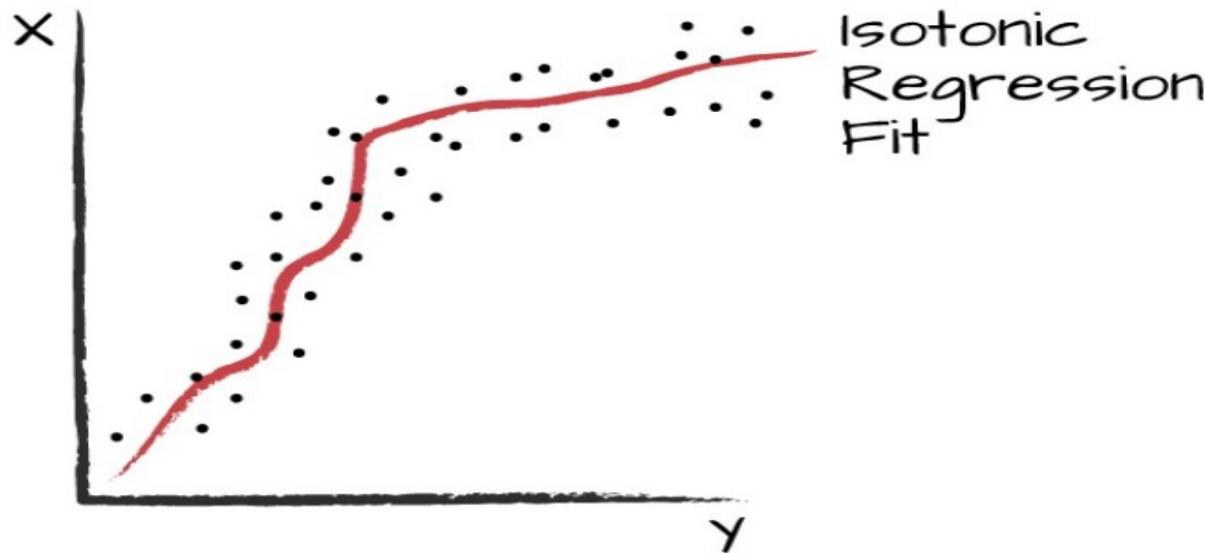
```
In [354]: from pyspark.ml.regression import AFTSurvivalRegression
In [355]: from pyspark.ml.linalg import Vectors
In [356]: training = spark.createDataFrame([
....:     (1.218, 1.0, Vectors.dense(1.560, -0.605)),
....:     (2.949, 0.0, Vectors.dense(0.346, 2.158)),
....:     (3.627, 0.0, Vectors.dense(1.380, 0.231)),
....:     (0.273, 1.0, Vectors.dense(0.520, 1.151)),
....:     (4.199, 0.0, Vectors.dense(0.795, -0.226))), ["label", "censor", "features"])
In [357]: quantileProbabilities = [0.3, 0.6]
In [358]: aft = AFTSurvivalRegression(quantileProbabilities=quantileProbabilities, quantilesCol="quantiles")
In [359]: model = aft.fit(training)
In [360]: print("Coefficients: " + str(model.coefficients))
Coefficients: [-0.4963111466650707, 0.19844437699933098]
In [361]: print("Intercept: " + str(model.intercept))
Intercept: 2.63809461510401
In [362]: print("Scale: " + str(model.scale))
Scale: 1.5472345574364692
In [363]: model.transform(training).show(truncate=False)
+-----+-----+-----+-----+
|label|censor|features      |prediction      |quantiles
+-----+-----+-----+-----+
|1.218|1.0   |[1.56,-0.605] |5.718979487635007 |[1.1603238947151664,4.99545601027477] |
|2.949|0.0   |[0.346,2.158] |18.07652118149533 |[3.667545845471739,15.789611866277625] |
|3.627|0.0   |[1.38,0.231]  |7.381861804239096 |[1.4977061305190829,6.44796261233896] |
|0.273|1.0   |[0.52,1.151]  |13.577612501425284 |[2.7547621481506854,11.8598722240697] |
|4.199|0.0   |[0.795,-0.226]|9.013097744073898 |[1.8286676321297826,7.87282650587843] |
+-----+-----+-----+-----+
```



## Isotonic Regression

isotonic regression specifies a piecewise linear function that is always monotonically increasing. It cannot decrease.

This means that if your data is going up and to the right in a given plot, this is an appropriate model. If it varies over the course of input values, then this is not appropriate.





```
In [376]: from pyspark.ml.regression import IsotonicRegression

In [377]: dataset = spark.read.format("libsvm")\
...: .load("/home/ilg/Documents/sparkdata/mllib/sample_isotonic_regression_libsvm_data.txt")
19/09/13 13:39:34 WARN LibSVMFileFormat: 'numFeatures' option not specified, determining the number of features by going though the input. If you know the number in advance, please specify it via 'numFeatures' option to avoid the extra scan.

In [378]: model = IsotonicRegression().fit(dataset)

In [379]: print("Boundaries in increasing order: %s\n" % str(model.boundaries))
Boundaries in increasing order: [0.01,0.17,0.18,0.27,0.28,0.29,0.3,0.31,0.34,0.35,0.36,0.41,0.42,0.71,0.72,0.74,0.75,0.76,0.77,0.78,0.79,0.8,0.81,0.82,0.83,0.84,0.85,0.86,0.87,0.88,0.89,1.0]

In [380]: print("Predictions associated with the boundaries: %s\n" % str(model.predictions))
Predictions associated with the boundaries: [0.15715271294117644,0.15715271294117644,0.189138196,0.189138196,0.20040796,0.29576747,0.43396226,0.5081591025000001,0.5081591025000001,0.54156043,0.5504844466666667,0.5504844466666667,0.563929967,0.563929967,0.5660377366666667,0.5660377366666667,0.56603774,0.57929628,0.64762876,0.66241713,0.67210607,0.67210607,0.674655785,0.674655785,0.73890872,0.73992861,0.84242733,0.89673636,0.89673636,0.90719021,0.9272055075,0.9272055075]
```



```
In [382]: model.transform(dataset).show()
```

label	features	prediction
0.24579296	(1, [0], [0.01])	0.15715271294117644
0.28505864	(1, [0], [0.02])	0.15715271294117644
0.31208567	(1, [0], [0.03])	0.15715271294117644
0.35900051	(1, [0], [0.04])	0.15715271294117644
0.35747068	(1, [0], [0.05])	0.15715271294117644
0.16675166	(1, [0], [0.06])	0.15715271294117644
0.17491076	(1, [0], [0.07])	0.15715271294117644
0.0418154	(1, [0], [0.08])	0.15715271294117644
0.04793473	(1, [0], [0.09])	0.15715271294117644
0.03926568	(1, [0], [0.1])	0.15715271294117644
0.12952575	(1, [0], [0.11])	0.15715271294117644
0.0	(1, [0], [0.12])	0.15715271294117644
0.01376849	(1, [0], [0.13])	0.15715271294117644
0.13105558	(1, [0], [0.14])	0.15715271294117644
0.08873024	(1, [0], [0.15])	0.15715271294117644
0.12595614	(1, [0], [0.16])	0.15715271294117644
0.15247323	(1, [0], [0.17])	0.15715271294117644
0.25956145	(1, [0], [0.18])	0.189138196
0.20040796	(1, [0], [0.19])	0.189138196
0.19581846	(1, [0], [0.2])	0.189138196

only showing top 20 rows



# Evaluators and Automating Model Tuning

# Evaluator



We can specify an evaluator, pick a metric to optimize for, and then train our pipeline to perform that parameter tuning on our part.

The evaluator for regression is called the **RegressionEvaluator** and allows us to optimize for a number of common regression success Metrics.

**RegressionEvaluator** expects two columns,

- 1) A column representing the prediction and another
- 2) Representing the True label.

The supported metrics to optimize for are the

root mean squared error (“**rmse**”),

mean squared error (“**mse**”),

r 2 metric (“**r2**”),

mean absolute error (“**mae**”).



To use **RegressionEvaluator**, we build up our pipeline, specify the parameters we would like to test, and then run it.

Spark will automatically select the model that performs best and return this to us

```
In [383]: from pyspark.ml.evaluation import RegressionEvaluator
In [384]: from pyspark.ml.regression import GeneralizedLinearRegression
In [385]: from pyspark.ml import Pipeline
In [386]: from pyspark.ml.tuning import CrossValidator,ParamGridBuilder
In [387]: glr = GeneralizedLinearRegression().setFamily("gaussian").setLink("identity")
In [388]: pipeline = Pipeline().setStages([glr])
In [389]: params = ParamGridBuilder().addGrid(glr.regParam,[0,0.5,1]).build()
In [390]: evaluator = RegressionEvaluator()\
...: .setMetricName("rmse")\
...: .setPredictionCol("prediction")\
...: .setLabelCol("label")
In [391]: cv = CrossValidator()\
...: .setEstimator(pipeline)\n...: .setEvaluator(evaluator)\n...: .setEstimatorParamMaps(params)\n...: .setNumFolds(2)
In [392]: model = cv.fit(jdf)
```



# Metrics

Evaluators allow us to evaluate and fit a model according to one specific metric, but we can also access a number of regression metrics via the RegressionMetrics object.

let's see how we can inspect the results of the previously trained model

```
In [393]: from pyspark.mllib.evaluation import RegressionMetrics
In [394]: out = model.transform(jdf) \
...: .select("prediction","label").rdd.map(lambda x:(float(x[0]),float(x[1])))
In [395]: metrics = RegressionMetrics(out)
In [397]: print('MSE: ' + str(metrics.meanSquaredError))
MSE: 0.15705521472392633
In [398]: print('RMSE: ' + str(metrics.rootMeanSquaredError))
RMSE: 0.3963019236944559
In [399]: print("R-squared: " + str(metrics.r2))
R-squared: 0.803680981595092
In [400]: print("MAE: " + str(metrics.meanAbsoluteError))
MAE: 0.3141104294478528
In [401]: print("Explained variance: " + str(metrics.explainedVariance))
Explained variance: 0.6429447852760737
```



# Recommendation

# Recommendation



By studying people's explicit preferences (through ratings) or implicit preferences (through observed behaviour), we can make recommendations on what one user may like by drawing similarities between the user and other users, or between the products they liked and other products.

# Use Cases



## Movie recommendations

Amazon, Netflix, and HBO all want to provide relevant film and TV content to their users. Netflix utilizes Spark, to make large scale movie recommendations to their users.

## Course recommendations

A school might want to recommend courses to students by studying what courses similar students have liked or taken. Past enrollment data makes for a very easy to collect training dataset for this task.

# Alternating Least Squares (ALS)



This algorithm leverages a technique called **collaborative filtering**, which makes recommendations based only on which items users interacted with in the past.

# Collaborative Filtering with Alternating Least Squares



ALS finds a  $-$ -dimensional feature vector for each user and item such that the dot product of each user's feature vector with each item's feature vector approximates the user's rating for that item.

Therefore this only requires an **input dataset of existing ratings** between user-item pairs, with three columns: a user ID column, an item ID column (e.g., a movie), and a rating column.

# Model Hyperparameters



## rank

- The rank term determines the dimension of the feature vectors learned for users and items. This should normally be tuned through experimentation. The core trade-off is that by specifying too high a rank, the algorithm may overfit the training data; but by specifying a low rank, then it may not make the best possible predictions.

The default value is 10.

## alpha

- When training on implicit feedback (behavioural observations), the alpha sets a baseline confidence for preference.
- This has a default of 1.0 and should be driven through experimentation.

# Model Hyperparameters



## **regParam**

**Controls regularization to prevent overfitting.** You should test out different values for the regularization parameter to find the optimal value for your problem.

The default is 0.1.

## **implicitPrefs**

This Boolean value specifies whether you are training on implicit (true) or explicit (false) (refer back to the preceding discussion for an explanation of the difference between explicit and implicit). This value should be set based on the data that you're using as input to the model. If the data is based off passive endorsement of a product (say, via a click or page visit), then you should use implicit preferences.

In contrast, if the data is an explicit rating (e.g., the user gave this restaurant 4/5 stars), you should use explicit preferences.

Explicit preferences are the default.

# Model Hyperparameters



## **nonnegative**

- If set to true, this parameter configures the model to place non-negative constraints on the least-squares problem it solves and only return non-negative feature vectors. This can improve performance in some applications. The default value is false.



# Training Parameters

## **numUserBlocks**

- This determines how many blocks to split the users into. The default is 10.

## **numItemBlocks**

- This determines how many blocks to split the items into. The default is 10.

## **maxIter**

- Total number of iterations over the data before stopping. Changing this probably won't change your results a ton, so this shouldn't be the first parameter you adjust.
- The default is 10.
- An example of when you might want to increase this is that after inspecting your objective history and noticing that it doesn't flatline after a certain number of training iterations.



# Training Parameters

## **checkpointInterval**

Checkpointing allows you to save model state during training to more quickly recover from node failures. You can set a checkpoint directory using **SparkContext.setCheckpointDir**.

## **seed**

Specifying a random seed can help you replicate your results.

# Prediction Parameters



Prediction parameters determine how a trained model should actually make predictions.

In our case, there's one parameter: the cold start strategy (set through **coldStartStrategy**). This setting determines what the model should predict for users or items that did not appear in the training set.

The cold start challenge commonly arises when you're serving a model in production, and new users and/or items have no ratings history, and therefore the model has no recommendation to make.

# E.g



```
In [406]: from pyspark.ml.recommendation import ALS
In [407]: from pyspark.sql import Row
In [408]: ratings = spark.read.text("/home/ilg/Documents/sparkdata/data/sample_movielens_ratings.txt")\
...: .rdd.toDF()\
...: .selectExpr("split(value ,':') as col")\
...: .selectExpr(
...: "cast(col[0] as int) as userId",
...: "cast(col[1] as int) as movieId",
...: "cast(col[2] as float) as rating",
...: "cast(col[3] as long) as timestamp")
In [409]: training,test = ratings.randomSplit([0.8,0.2])
In [411]: als = ALS()\
...: .setMaxIter(5)\
...: .setRegParam(0.01)\
...: .setUserCol("userId")\
...: .setItemCol("movieId")\
...: .setRatingCol("rating")
In [412]: print(als.explainParams())
alpha: alpha for implicit preference (default: 1.0)
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations. Note: this setting will be ignored if the checkpoint directory is not set in the SparkContext. (default: 10)
In [413]: alsModel = als.fit(training)
In [414]: predictions = alsModel.transform(test)
```



We can now output the top recommendations for each user or movie. The model's recommendForAllUsers method returns a DataFrame of a userId, an array of recommendations, as well as a rating for each of those movies.

```
In [415]: alsModel.recommendForAllUsers(10)\n... .selectExpr("userId","explode(recommendations)").show()
```

userId	col
28	[46, 5.267052]
28	[92, 4.8686347]
28	[81, 4.6571302]
28	[12, 4.4722905]
28	[49, 4.1764855]
28	[76, 4.0431314]
28	[2, 3.8680873]
28	[29, 3.6960588]
28	[23, 3.3141887]
28	[0, 3.0353324]
26	[92, 8.075099]
26	[83, 7.3489685]
26	[37, 6.7769547]
26	[93, 6.291627]
26	[19, 6.28405]
26	[12, 5.25953]
26	[7, 5.1548767]
26	[87, 5.1083903]
26	[74, 4.939903]
26	[24, 4.93427]

only showing top 20 rows



```
In [417]: alsModel.recommendForAllItems(10)\n...: .selectExpr("movieId","explode(recommendations)").show()\n+-----+-----+\n|movieId|      col|\n+-----+-----+\n|      31| [21, 4.702184]|\n|      31| [10, 3.8390143]|\n|      31| [12, 3.5174427]|\n|      31| [14, 3.0250764]|\n|      31| [23, 2.851439]|\n|      31| [7, 2.7176976]|\n|      31| [8, 2.6995623]|\n|      31| [15, 2.6019657]|\n|      31| [6, 2.4296222]|\n|      31| [28, 2.3890374]|\n|      85| [16, 4.692398]|\n|      85| [7, 4.023702]|\n|      85| [10, 3.3503313]|\n|      85| [6, 3.197785]|\n|      85| [21, 2.8491447]|\n|      85| [19, 2.807176]|\n|      85| [1, 2.350979]|\n|      85| [18, 2.2987618]|\n|      85| [8, 2.20776]|\n|      85| [0, 1.9538894]|\n+-----+-----+\nonly showing top 20 rows
```



# Evaluators for Recommendation

```
In [418]: from pyspark.ml.evaluation import RegressionEvaluator  
  
In [419]: evaluator = RegressionEvaluator()\\"  
....: .setMetricName("rmse")\"  
....: .setLabelCol("rating")\"  
....: .setPredictionCol("prediction")  
  
In [420]: rmse = evaluator.evaluate(predictions)  
  
In [421]: print("Root-mean-square error = %f" % rmse)  
Root-mean-square error = 1.903952
```

# Metrics



## Regression Metrics

We can recycle the regression metrics for recommendation. This is because we can simply see how close each prediction is to the actual rating for that user and item

```
In [422]: from pyspark.mllib.evaluation import RegressionMetrics  
  
In [423]: regComparison = predictions.select("rating","prediction")\  
....: .rdd.map(lambda x:(x(0),x(1)))  
  
In [424]: metrics = RegressionMetrics(regComparison)
```



## Ranking Metrics

A RankingMetric allows us to compare our recommendations with an actual set of ratings (or preferences) expressed by a given user.

```
In [428]: from pyspark.mllib.evaluation import RankingMetrics, RegressionMetrics
In [429]: from pyspark.sql.functions import col, expr
In [430]: perUserActual = predictions\
....: .where("rating > 2.5")\
....: .groupBy("userId")\
....: .agg(expr("collect_set(movieId) as movies"))
```

At this point, we have a collection of users, along with a truth set of previously ranked movies for each user.

Now we will get our top 10 recommendations from our algorithm on a per-user basis.

*We will then see if the top 10 recommendations show up in our truth set. If we have a well-trained model, it will correctly recommend the movies a user already liked. If it doesn't, it may not have learned enough about each particular user to successfully reflect their preferences*



Now we have two DataFrames, one of predictions and another the top-ranked items for a particular user.

We can pass them into the RankingMetrics object. This object accepts an RDD of these combinations

```
In [431]: perUserPredictions = predictions\  
....: .orderBy(col("userId"),expr("prediction DESC"))\  
....: .groupBy("userId")\  
....: .agg(expr("collect_list(movieId) as movies"))
```

```
In [432]: perUserActualvPred = perUserActual.join(perUserPredictions, ["userId"]).rdd\  
....: .map(lambda row: (row[1], row[2][:15]))  
In [433]: ranks = RankingMetrics(perUserActualvPred)
```



Now we can see the metrics from that ranking.

For instance,

we can see how precise our algorithm is with the mean average precision. We can also get the precision at certain ranking points,

for instance,

To see where the majority of the positive recommendations fall

```
In [434]: ranks.meanAveragePrecision  
Out[434]: 0.3174765234765235
```

```
In [435]: ranks.precisionAt(5)  
Out[435]: 0.6159999999999999
```

# Assignment – Build, train, test the model



## Frequent Pattern Mining

MLlib provides for creating recommendations is **frequent pattern mining**. Frequent pattern mining, sometimes referred to as **market basket analysis**, looks at raw data and finds association rules.

For instance, given a large number of transactions it might identify that users who buy hot dogs almost always purchase hot dog buns.

This technique can be applied in the recommendation context, especially when people are filling shopping carts (either on or offline).

Spark implements the **FP-growth** algorithm for frequent pattern mining.

Refer

<https://spark.apache.org/docs/latest/ml-frequent-pattern-mining.html#fp-growth>



# Unsupervised Learning

# Use Cases



## Finding anomalies in data

- If the majority of values in a dataset cluster into a larger group with several small groups on the outside, those groups might warrant further investigation.

## Topic modeling

- By looking at large bodies of text, it is possible to find topics that exist across those different documents.

# Model Scalability



Model	Statistical recommendation	Computation limits	Training examples
$k$ -means	50 to 100 maximum	Features x clusters < 10 million	No limit
Bisecting $k$ -means	50 to 100 maximum	Features x clusters < 10 million	No limit
GMM	50 to 100 maximum	Features x clusters < 10 million	No limit
LDA	An interpretable number	1,000s of topics	No limit

# E.g



Let's get started by loading some example numerical data:

```
In [436]: from pyspark.ml.feature import VectorAssembler  
  
In [437]: va = VectorAssembler()\n        .setInputCols(["Quantity", "UnitPrice"])\n        .setOutputCol("features")  
  
In [438]: sales = va.transform(spark.read.format("csv")\n        .option("header", "true")\n        .option("inferSchema", "true")\n        .load("/home/ilg/Documents/sparkdata/data/retail-data/by-day/*.csv")\n        .limit(5)\n        .coalesce(1)\n        .where("Description is NOT NULL"))  
  
In [439]: sales.cache()  
Out[439]: DataFrame[InvoiceNo: string, StockCode: string, Description: string,  
timestamp, UnitPrice: double, CustomerID: double, Country: string, features:
```

## k-means



In this algorithm, a user-specified number of clusters ( $k$ ) are randomly assigned to different points in the dataset.

The unassigned points are then “assigned” to a cluster based on their proximity (measured in Euclidean distance) to the previously assigned point.

Once this assignment happens, the center of this cluster (called the centroid) is computed, and the process repeats.

All points are assigned to a particular centroid, and a new centroid is computed. We repeat this process for a finite number of iterations or until convergence (i.e., when our centroid locations stop changing).

# Model Hyperparameters



These are configurations that we specify to determine the basic structure of the model:

- This is the number of clusters that you would like to end up with.



# Training Parameters

## **initMode**

The initialization mode is the algorithm that determines the starting locations of the centroids. The supported options are random and -means|| (the default).

The latter is a parallelized variant of the -means|| method.

## **initSteps**

- The number of steps for -mean|| initialization mode. Must be greater than 0. (The default value is 2.)



## **maxIter**

- Total number of iterations over the data before stopping. Changing this probably won't change your results a ton, so don't make this the first parameter you look to adjust.
- The default is 20.

## **tol**

- Specifies a threshold by which changes in centroids show that we optimized our model enough, and can stop iterating early, before maxIter iterations.
- The default value is 0.0001.



## E.g

This algorithm is generally robust to these parameters, and the main trade-off is that running more initialization steps and iterations may lead to a better clustering at the expense of longer training time

```
In [442]: from pyspark.ml.clustering import KMeans
In [443]: km = KMeans().setK(5)
In [444]: print(km.explainParams())
distanceMeasure: the distance measure. Supported options: 'euclidean' and 'cosine'. (default: euclidean)
featuresCol: features column name. (default: features)
initMode: The initialization algorithm. This can be either "random" to choose random points as initial cluster centers, or "k-means||" to use a parallel variant of k-means++ (default: k-means||)
initSteps: The number of steps for k-means|| initialization mode. Must be > 0. (default: 2)
k: The number of clusters to create. Must be > 1. (default: 2, current: 5)
maxIter: max number of iterations (>= 0). (default: 20)
predictionCol: prediction column name. (default: prediction)
seed: random seed. (default: 7969353092125344463)
tol: the convergence tolerance for iterative algorithms (>= 0). (default: 0.0001)

In [445]: kmModel = km.fit(sales)
19/09/13 15:23:28 WARN KMeans: The input data is not directly cached, which may hurt performance if its parent RDDs are also uncached.
```

# k-means Metrics Summary



K-means includes a summary class that we can use to evaluate our model.

The -means summary includes information about the clusters created, as well as their relative sizes.

We can also compute the within set sum of squared errors, which can help measure how close our values are from each cluster centroid, using **computeCost**.

```
In [446]: summary = kmModel.summary
In [447]: print(summary.clusterSizes)
[2, 1, 1, 1, 0]
In [448]: kmModel.computeCost(sales)
Out[448]: 0.0799999999992724
In [449]: centers = kmModel.clusterCenters()
In [450]: print("Cluster Centers: ")
Cluster Centers:
In [451]: for center in centers:
...:     print(center)
...:
[24.    1.45]
[48.    1.79]
[6.     2.55]
[20.   1.25]
```



## Bisecting k-means

Bisecting -means is a variant of -means. The core difference is that instead of clustering points by starting “bottom-up” and assigning a bunch of different groups in the data, this is a top-down clustering method.

This means that it will start by creating a single group and then splitting that group into smaller groups in order to end up with the number of clusters specified by the user.

This is usually a faster method than -means and will yield different results.

# Model Hyperparameters



These are configurations that we specify to determine the basic structure of the model:

- This is the number of clusters that you would like to end up with.



# Training Parameters

## **minDivisibleClusterSize**

- The minimum number of points (if greater than or equal to 1.0) or the minimum proportion of points (if less than 1.0) of a divisible cluster. The default is 1.0, meaning that there must be at least one point in each cluster.

## **maxIter**

- Total number of iterations over the data before stopping. Changing this probably won't change your results a ton, so don't make this the first parameter you look to adjust.

The default is 20.



# E.g

```
In [452]: from pyspark.ml.clustering import BisectingKMeans
```

```
In [453]: bkm = BisectingKMeans().setK(5).setMaxIter(5)
```

```
In [454]: bkmModel = bkm.fit(sales)
```

```
19/09/13 15:31:33 WARN BisectingKMeans: The input RDD 5239 is not directly cached, which may hurt performance if its parent RDDs are also not cached.
```



## Bisecting k-means Summary

Bisecting -means includes a summary class that we can use to evaluate our model, that is largely the same as the -means summary.

This includes information about the clusters created, as well as their relative sizes

```
In [455]: summary = bkmModel.summary  
  
In [456]: print(summary.clusterSizes)  
[1, 1, 1, 1, 1]  
  
In [457]: kmModel.computeCost(sales)  
Out[457]: 0.0799999999992724  
  
In [458]: centers = kmModel.clusterCenters()  
  
In [459]: print("Cluster Centers: ")  
Cluster Centers:  
  
In [460]: for center in centers:  
...:     print(center)  
...:  
[24.    1.45]  
[48.    1.79]  
[6.     2.55]  
[20.   1.25]
```

# Gaussian Mixture Models



A simplified way of thinking about Gaussian mixture models is that they're like a soft version of -means. -means creates very rigid clusters—each point is only within one cluster.

GMMs allow for a more nuanced cluster associated with probabilities, instead of rigid boundaries.

# Model Hyperparameters



These are configurations that we specify to determine the basic structure of the model:

- This is the number of clusters that you would like to end up with.



# Training Parameters

## **maxIter**

Total number of iterations over the data before stopping. Changing this probably won't change your results a ton, so don't make this the first parameter you look to adjust.

The default is 100  
**tol**

This value simply helps us specify a threshold by which changes in parameters show that we optimized our weights enough.

A smaller value can lead to higher accuracy at the cost of performing more iterations (although never more than maxIter).

The default value is 0.01.



# E.g

```
In [461]: from pyspark.ml.clustering import GaussianMixture
In [462]: gmm = GaussianMixture().setK(5)
In [463]: model = gmm.fit(sales)
In [464]: print(gmm.explainParams())
featuresCol: features column name. (default: features)
k: Number of independent Gaussians in the mixture model. Must be > 1
maxIter: max number of iterations (>= 0). (default: 100)
```

# Gaussian Mixture Model Summary



This includes information about the clusters created, like the weights, the means, and the covariance of the Gaussian mixture, which can help us learn more about the underlying structure inside of our data

```
In [466]: summary = model.summary
In [467]: print(model.weights)
[0.200000000000012, 0.1000000000000117, 0.1000000000000117, 0.599999999999993, 3.5032730501583805e-15]
In [468]: model.gaussiansDF.show()
+-----+-----+
|      mean |      cov |
+-----+-----+
|[47.999999999998...|4.092726157978153...|
|[6.0000000000019...|3.289812866569225...|
|[6.0000000000019...|3.289812866569225...|
|[22.666666666666...|3.555555555555547...|
|[22.6666728586391...|3.555547299577572...|
+-----+-----+
```



```
In [469]: summary.cluster.show()
```

```
+-----+  
|prediction|  
+-----+  
|      0|  
|      3|  
|      3|  
|      3|  
|      1|  
+-----+
```

```
In [470]: summary.clusterSizes
```

```
Out[470]: [1, 1, 0, 3, 0]
```

```
In [471]: summary.probability.show()
```

```
+-----+  
|      probability|  
+-----+  
|[1.0,3.2878094560...|  
|[1.94626254955070...|  
|[1.94626254955067...|  
|[1.94626254955069...|  
|[7.65964830754763...|  
+-----+
```

# Latent Dirichlet Allocation



**Latent Dirichlet Allocation (LDA)** is a hierarchical clustering model typically used to perform topic modelling on text documents.

LDA tries to extract high-level topics from a series of documents and keywords associated with those topics.

It then interprets each document as having a variable number of contributions from multiple input topics

# Model Hyperparameters



These are configurations that we specify to determine the basic structure of the model:

- The total number of topics to infer from the data. The default is 10 and must be a positive number

# Model Hyperparameters



## **docConcentration**

Concentration parameter (commonly named “alpha”) for the prior placed on documents’ distributions over topics (“theta”). This is the parameter to a Dirichlet distribution, where larger values mean more smoothing (more regularization).

If not set by the user, then `docConcentration` is set automatically. If set to singleton vector `[alpha]`, then `alpha` is replicated to a vector of length `k` in fitting. Otherwise, the `docConcentration` vector must be length `.`

## **topicConcentration**

The concentration parameter (commonly named “beta” or “eta”) for the prior placed on a topic’s distributions over terms. This is the parameter to a symmetric Dirichlet distribution. If not set by the user, then `topicConcentration` is set automatically.



# Training Parameters

## **maxIter**

- Total number of iterations over the data before stopping. Changing this probably won't change your results a ton, so don't make this the first parameter you look to adjust.
- The default is 20.

## **optimizer**

- This determines whether to use EM or online training optimization to determine the LDA model.
- The default is online.



## **LearningDecay**

- **Learning** rate, set as an exponential decay rate. This should be between (0.5, 1.0] to guarantee asymptotic convergence.
- The default is 0.51 and only applies to the online optimizer.

## **learningOffset**

- A (positive) learning parameter that downweights early iterations. Larger values make early iterations count less.
- The default is 1,024.0 and only applies to the online optimizer.



- **optimizeDocConcentration**
    - Indicates whether the **docConcentration** (Dirichlet parameter for document-topic distribution) will be optimized during training. The default is true but only applies to the online optimizer.
  - **subsamplingRate**
    - The fraction of the corpus to be sampled and used in each iteration of mini-batch gradient descent, in range  $(0, 1]$ . The default is 0.5 and only applies to the online optimizer.
  - **seed**
    - This model also supports specifying a random seed for reproducibility.
- checkpointInterval**
- This is the same checkpoint feature that we saw

# Prediction Parameters



## **topicDistributionCol**

- The column that will hold the output of the topic mixture distribution for each document.



## E.g

```
In [472]: from pyspark.ml.feature import Tokenizer, CountVectorizer
In [473]: tkn = Tokenizer().setInputCol("Description").setOutputCol("DescOut")
In [474]: tokenized = tkn.transform(sales.drop("features"))
In [475]: cv = CountVectorizer()\
.... .setInputCol("DescOut")\
.... .setOutputCol("features")\
.... .setVocabSize(500)\ 
.... .setMinTF(0)\ 
.... .setMinDF(0)\ 
.... .setBinary(True)
In [476]: cvFitted = cv.fit(tokenized)
In [477]: prepped = cvFitted.transform(tokenized)
In [478]: from pyspark.ml.clustering import LDA
In [479]: lda = LDA().setK(10).setMaxIter(5)
In [480]: print(lda.explainParams())
checkpointInterval: set checkpoint interval (>= 1) or disable checkpoint (-1).
In [481]: model = lda.fit(prepped)
```



After we train the model, you will see some of the top topics

```
In [482]: model.describeTopics(3).show()
```

topic	termIndices	termWeights
0	[11, 1, 3]	[0.06543235153816...]
1	[2, 9, 14]	[0.06134516450423...]
2	[13, 16, 2]	[0.06623482455173...]
3	[3, 16, 2]	[0.06241956820943...]
4	[0, 8, 13]	[0.07144965181761...]
5	[0, 3, 5]	[0.07143817527325...]
6	[5, 1, 12]	[0.06857395247118...]
7	[11, 6, 8]	[0.06906358498944...]
8	[13, 11, 15]	[0.06830205776515...]
9	[8, 9, 14]	[0.06842627585816...]

```
In [483]: cvFitted.vocabulary
```

```
Out[483]:
```

```
['box',
 'lip',
 'harmonica',
 'rabbit',
 'with',
 'doughnut',
 'night',
 'in',
 'blue',
 'light',
 'message',
 'coat',
 'rack',
 '12',
 'gloss',
 'gumball',
 'envelopes',
 'cards']
```



# Graph Analytics

# Graphs

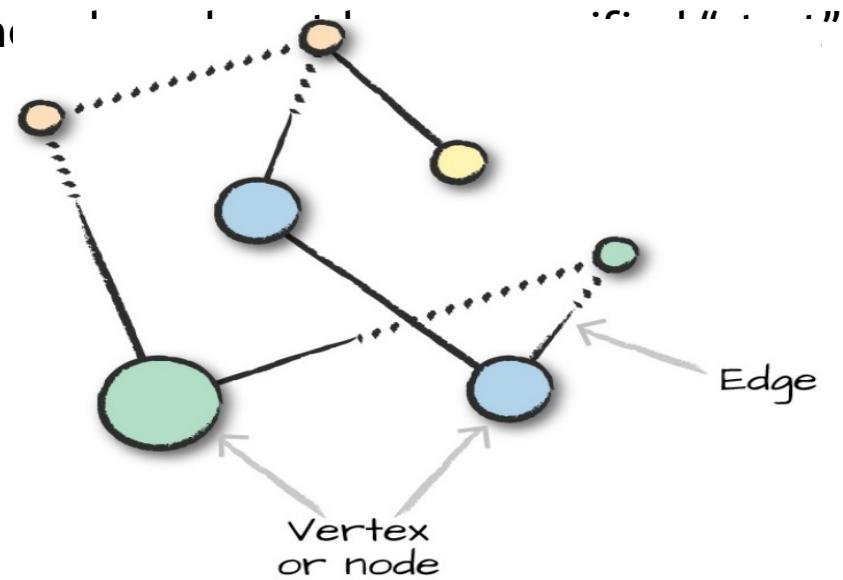


Graphs are data structures composed of nodes, or vertices, which are arbitrary objects, and edges that define relationships between these nodes.

Graph analytics is the process of analyzing these relationships.

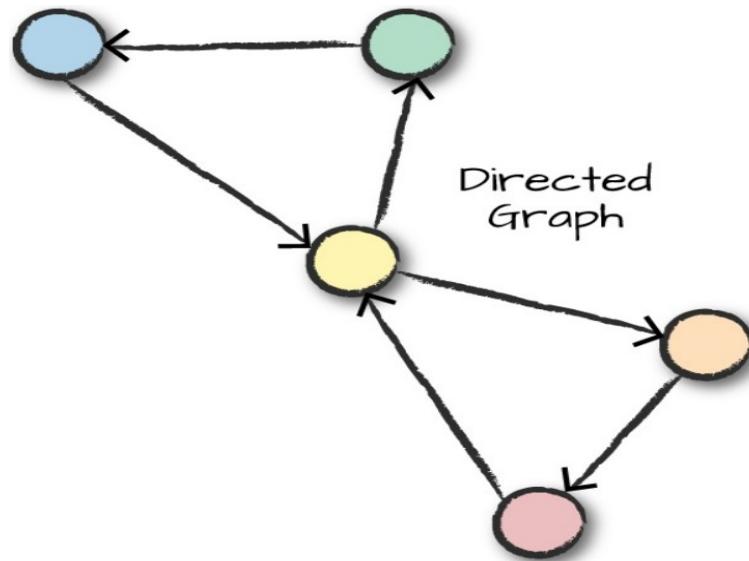
An example graph might be your friend group.

This particular graph is undirected, in that there is no “begin” and “end” vertex.





There are also directed graphs that specify a start and end.  
A directed graph where the edges are directional.





# Graphs & use cases

Graphs are a natural way of describing relationships and many different problem sets, and Spark provides several ways of working in this analytics paradigm.

## **use cases**

detecting credit card fraud, motif finding, determining importance of papers in bibliographic networks (i.e., which papers are most referenced), and ranking web pages, as Google famously used the PageRank algorithm to do.



The goal of these slides is to show / explain us how to use GraphFrames to perform graph analytics on Spark.

To get set up, you're going to need to point to the proper package. To do this from the command line, you'll run:

```
$ spark-shell --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11
```

```
$ pyspark --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11
```

```
$ pyspark --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11
```

```
In [1]: bikeStations = spark.read.option("header","true")\
....: .csv("/home/ilg/Documents/sparkdata/data/bike-data/201508_station_data.csv")  
  
In [2]: tripData = spark.read.option("header","true")\
....: .csv("/home/ilg/Documents/sparkdata/data/bike-data/201508_trip_data.csv")
```



## Building a Graph

The first step is to build the graph. To do this we need to define the vertices and edges, which are DataFrames with some specifically named columns.

In our case, we're creating a directed graph.

This graph will point from the source to the location. In the context of this bike trip data, this will point from a trip's starting location to a trip's ending location.

To define the graph, we use the naming conventions for columns presented in the GraphFrames library.

In the vertices table we define our identifier as `id` (in our case this is of string type), and in the edges table we label each edge's source vertex ID as `src` and the destination ID as `dst`



```
In [3]: stationVertices = bikeStations.withColumnRenamed("name", "id").distinct()  
In [4]: tripEdges = tripData\  
....: .withColumnRenamed("Start Station", "src")\  
....: .withColumnRenamed("End Station", "dst")
```



We can now build a GraphFrame object, which represents our graph, from the vertex and edge DataFrames we have so far.

We will also leverage caching because we'll be accessing this data frequently in later queries

```
In [5]: from graphframes import GraphFrame  
  
In [6]: stationGraph = GraphFrame(stationVertices, tripEdges)  
  
In [7]: stationGraph.cache()  
Out[7]: GraphFrame(v:[id: string, station_id: string ... 5 more fields])
```



Now we can see the basic statistics about graph (and query our original DataFrame to ensure that we see the expected results)

```
In [8]: print("Total Number of Stations: " + str(stationGraph.vertices.count()))
Total Number of Stations: 70

In [9]: print("Total Number of Trips in Graph: " + str(stationGraph.edges.count()))
Total Number of Trips in Graph: 354152

In [10]: print("Total Number of Trips in Original Data: " + str(tripData.count()))
Total Number of Trips in Original Data: 354152
```



# Querying the Graph

The most basic way of interacting with the graph is simply querying it, performing things like counting trips and filtering by given destinations.

GraphFrames provides simple access to both vertices and edges as DataFrames.

Note that our graph retained all the additional columns in the data in addition to IDs, sources, and destinations, so we can also query those if needed.

```
In [11]: from pyspark.sql.functions import desc
In [12]: stationGraph.edges.groupBy("src", "dst").count().orderBy(desc("count")).show(10)
+-----+-----+-----+
|      src|          dst|count|
+-----+-----+-----+
|San Francisco Cal...| Townsend at 7th| 3748|
|Harry Bridges Pla...|Embarcadero at Sa...| 3145|
| 2nd at Townsend| Harry Bridges Pla...| 2973|
| Townsend at 7th| San Francisco Cal...| 2734|
|Harry Bridges Pla...| 2nd at Townsend| 2640|
|Embarcadero at Fo...| San Francisco Cal...| 2439|
| Steuart at Market| 2nd at Townsend| 2356|
|Embarcadero at Sa...| Steuart at Market| 2330|
| Townsend at 7th| San Francisco Cal...| 2192|
|Temporary Transba...| San Francisco Cal...| 2184|
+-----+-----+-----+
only showing top 10 rows
```



We can also filter by any valid DataFrame expression. In this instance, I want to look at one specific station and the count of trips in and out of that station.

```
In [13]: stationGraph.edges\
....: .where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")\
....: .groupBy("src", "dst").count()\
....: .orderBy(desc("count"))\
....: .show(10)
+-----+-----+-----+
|       src|          dst|count|
+-----+-----+-----+
| San Francisco Cal...| Townsend at 7th| 3748|
| Townsend at 7th| San Francisco Cal...| 2734|
| Townsend at 7th| San Francisco Cal...| 2192|
| Townsend at 7th| Civic Center BART...| 1844|
| Civic Center BART...| Townsend at 7th| 1765|
| San Francisco Cal...| Townsend at 7th| 1198|
| Temporary Transba...| Townsend at 7th| 834|
| Townsend at 7th| Harry Bridges Pla...| 827|
| Steuart at Market| Townsend at 7th| 746|
| Townsend at 7th| Temporary Transba...| 740|
+-----+-----+-----+
only showing top 10 rows
```



# Subgraphs

Subgraphs are just smaller graphs within the larger one. We saw in the last section how we can query a given set of edges and vertices. We can use this query ability to create subgraphs

```
In [14]: townAnd7thEdges = stationGraph.edges\  
.... .where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")  
  
In [15]: subgraph = GraphFrame(stationGraph.vertices, townAnd7thEdges)
```



We can then apply the following algorithms to either the original graph or the subgraph.

## Motif Finding

Motifs are a way of expressing structural patterns in a graph. When we specify a motif, we are querying for patterns in the data instead of actual data. In GraphFrames, we specify our query in a domain-specific language similar to Neo4J's Cypher language.

This language lets us specify combinations of vertices and edges and assign them names.

For example,

- if we want to specify that a given vertex a connects to another vertex b through an edge ab, we would specify (a)-[ab]->(b).

The names inside parentheses or brackets do not signify values but instead what the columns for matching vertices and edges should be named in the resulting DataFrame.



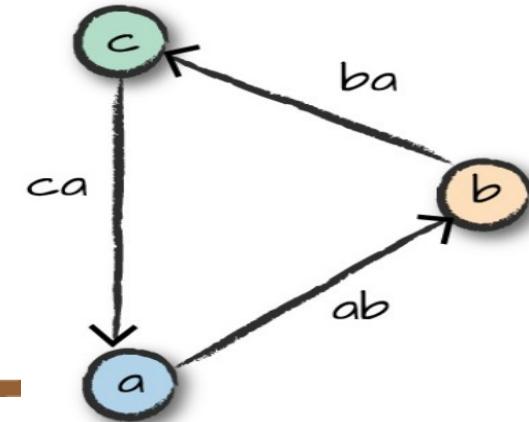
Let's perform a query on our bike data. i.e

let's find all the rides that form a "triangle" pattern between three stations.

We express this with the following motif, using the find method to query our GraphFrame for that pattern.

(a) signifies the starting station, and [ab] represents an edge from (a) to our next station (b). We repeat this for stations (b) to (c) and then from (c)

```
In [16]: motifs = stationGraph.find("(a)-[ab]->(b); (b)-[bc]->(c); (c)-[ca]->(a)")
```





The DataFrame we get from running this query contains nested fields for vertices a, b, and c, as well as the respective edges. We can now query this as we would a DataFrame.

For example,

given a certain bike, what is the shortest trip the bike has taken from station a, to station b, to station c, and back to station a ?

The following logic will parse our timestamps, into Spark timestamps and then we'll do comparisons to make sure that it's the same bike, traveling from station to station, and that the start times for each trip are correct



```
In [23]: motifs.selectExpr("*",
...: "to_timestamp(ab.`Start Date`, 'MM/dd/yyyy HH:mm') as abStart",
...: "to_timestamp(bc.`Start Date`, 'MM/dd/yyyy HH:mm') as bcStart",
...: "to_timestamp(ca.`Start Date`, 'MM/dd/yyyy HH:mm') as caStart")\
...: .where("ca.`Bike #` = bc.`Bike #`").where("ab.`Bike #` = bc.`Bike #`")\
...: .where("a.id != b.id").where("b.id != c.id")\
...: .where("abStart < bcStart").where("bcStart < caStart")\
...: .orderBy(expr("cast(caStart as long) - cast(abStart as long)"))\
...: .selectExpr("a.id", "b.id", "c.id", "ab.`Start Date`", "ca.`End Date`")\
...: .limit(1).show(1, False)
```

We see the fastest trip is approximately 20 minutes.

# Graph Algorithms



A graph is just a logical representation of data.

Graph theory provides numerous algorithms for analyzing data in this format, and GraphFrames allows us to leverage many algorithms out of the box.

# PageRank



One of the most prolific graph algorithms is PageRank. Larry Page, cofounder of Google, created PageRank as a research project for how to rank web pages.

PageRank generalizes quite well outside of the web domain. We can apply this right to our own data and get a sense for important bike stations (specifically, those that receive a lot of bike traffic). In this example, important bike stations will be

```
In [23]: from pyspark.sql.functions import desc
```

```
In [24]: ranks = stationGraph.pageRank(resetProbability=0.15, maxIter=10)
```

```
In [25]: ranks.vertices.orderBy(desc("pagerank")).select("id", "pagerank").show(10)
+-----+-----+
|       id|    pagerank|
+-----+-----+
|San Jose Diridon ...| 4.051504835989922|
|San Francisco Cal...| 3.3511832964279518|
|Mountain View Cal...| 2.51439077101569|
|Redwood City Calt...| 2.326308771371272|
|San Francisco Cal...| 2.2311442913697364|
|Harry Bridges Pla...| 1.8251120118883524|
| 2nd at Townsend| 1.5821217785041168|
|Santa Clara at Al...| 1.5730074084908332|
| Townsend at 7th| 1.568456580534273|
|Embarcadero at Sa...| 1.5414242087749768|
+-----+-----+
only showing top 10 rows
```

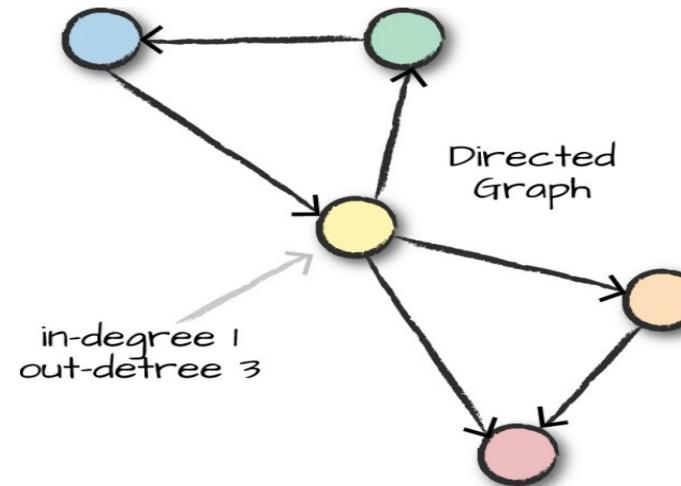


# In-Degree and Out-Degree Metrics

Our graph is a directed graph. This is due to the bike trips being directional, starting in one location and ending in another.

One common task is to count the number of trips into or out of a given station.

To measure trips in and out of stations, we will use a metric called in-degree and out-degree





This is particularly applicable in the context of social networking because certain users may have many more inbound connections (i.e., followers) than outbound connections (i.e., people they follow).

Using the following query, you can find interesting people in the social network who might have more influence than others.

GraphFrames provides a simple way to query our graph for this information.

```
In [26]: inDeg = stationGraph.inDegrees  
  
In [27]: inDeg.orderBy(desc("inDegree")).show(5, False)  
+-----+-----+  
| id | inDegree |  
+-----+-----+  
| San Francisco Caltrain (Townsend at 4th) | 34810 |  
| San Francisco Caltrain 2 (330 Townsend) | 22523 |  
| Harry Bridges Plaza (Ferry Building) | 17810 |  
| 2nd at Townsend | 15463 |  
| Townsend at 7th | 15422 |  
+-----+-----+  
only showing top 5 rows
```



We can query the out degrees in the same fashion:

```
In [28]: outDeg = stationGraph.outDegrees
```

```
In [29]: outDeg.orderBy(desc("outDegree")).show(5, False)
```

+-----+ <th> id</th> <th>+-----+<th> outDegree </th><th>+-----+</th></th>	id	+-----+ <th> outDegree </th> <th>+-----+</th>	outDegree	+-----+
San Francisco Caltrain (Townsend at 4th)		26304		
San Francisco Caltrain 2 (330 Townsend)		21758		
Harry Bridges Plaza (Ferry Building)		17255		
Temporary Transbay Terminal (Howard at Beale)		14436		
Embarcadero at Sansome		14158		

+-----+  
only showing top 5 rows



The ratio of these two values is an interesting metric to look at.

A higher ratio value will tell us where a large number of trips end (but rarely begin), while a lower value tells us where trips often begin (but infrequently end).

Those queries result in the following data:

```
In [30]: degreeRatio = inDeg.join(outDeg, "id")\
.... .selectExpr("id", "double(inDegree)/double(outDegree) as degreeRatio")

In [31]: degreeRatio.orderBy(desc("degreeRatio")).show(10, False)
+-----+-----+
|id                |degreeRatio   |
+-----+-----+
|Redwood City Medical Center |1.533333333333334|
|San Mateo County Center |1.4724409448818898|
|SJSU 4th at San Carlos |1.3621052631578947|
|San Francisco Caltrain (Townsend at 4th)|1.3233728710462287|
|Washington at Kearny |1.3086466165413533|
|Paseo de San Antonio |1.2535046728971964|
|California Ave Caltrain Station |1.24|
|Franklin at Maple |1.2345679012345678|
|Embarcadero at Vallejo |1.2201707365495336|
|Market at Sansome |1.2173913043478262|
+-----+-----+
only showing top 10 rows
```



```
In [32]: degreeRatio.orderBy("degreeRatio").show(10, False)
+-----+-----+
|id                |degreeRatio   |
+-----+-----+
|Grant Avenue at Columbus Avenue|0.5180520570948782|
|2nd at Folsom          |0.5909488686085761|
|Powell at Post (Union Square)|0.6434241245136186|
|Mezes Park            |0.6839622641509434|
|Evelyn Park and Ride    |0.7413087934560327|
|Beale at Market        |0.75726761574351|
|Golden Gate at Polk      |0.7822270981897971|
|Ryland Park             |0.7857142857142857|
|San Francisco City Hall|0.7928849902534113|
|Palo Alto Caltrain Station|0.8064516129032258|
+-----+-----+
only showing top 10 rows
```



## Breadth-First Search

Breadth-first search will search our graph for how to connect two sets of nodes, based on the edges in the graph.

In our context, we might want to do this to find the shortest paths to different stations, but the algorithm also works for sets of nodes specified through a SQL expression.

We can specify the maximum of edges to follow with the `maxPathLength`, and we can also specify an `edgeFilter` to filter out edges that do not meet a requirement, like trips during nonbusiness hours.

We'll choose two fairly close stations so that this does not run too long.

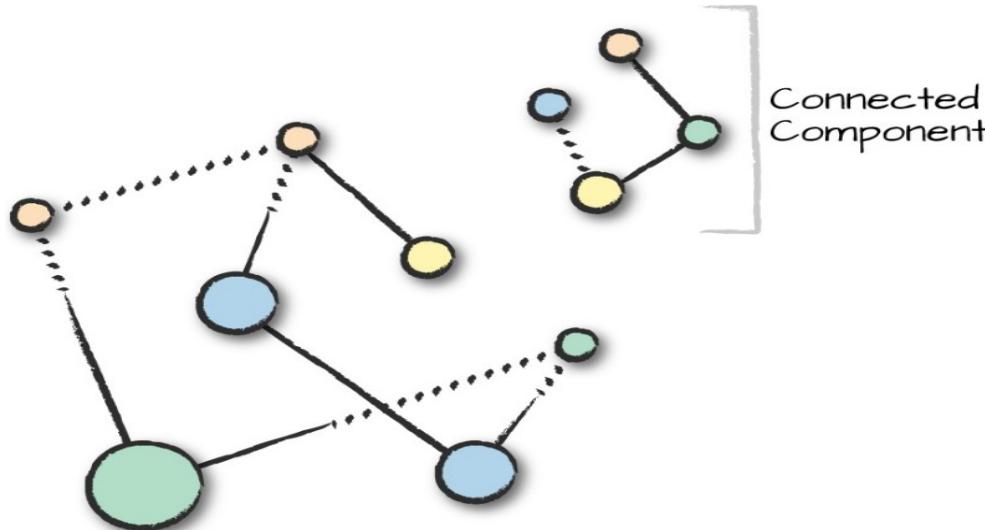


```
In [33]: stationGraph.bfs(fromExpr="id = 'Townsend at 7th'",  
...: toExpr="id = 'Spear at Folsom'", maxPathLength=2).show(10)  
+-----+-----+-----+  
|      from |          e0 |      to |  
+-----+-----+-----+  
|[65, Townsend at ...|[913371, 663, 8/3...|[49, Spear at Fol...  
|[65, Townsend at ...|[913265, 658, 8/3...|[49, Spear at Fol...  
|[65, Townsend at ...|[911919, 722, 8/3...|[49, Spear at Fol...  
|[65, Townsend at ...|[910777, 704, 8/2...|[49, Spear at Fol...  
|[65, Townsend at ...|[908994, 1115, 8/...|[49, Spear at Fol...  
|[65, Townsend at ...|[906912, 892, 8/2...|[49, Spear at Fol...  
|[65, Townsend at ...|[905201, 980, 8/2...|[49, Spear at Fol...  
|[65, Townsend at ...|[904010, 969, 8/2...|[49, Spear at Fol...  
|[65, Townsend at ...|[903375, 850, 8/2...|[49, Spear at Fol...  
|[65, Townsend at ...|[899944, 910, 8/2...|[49, Spear at Fol...  
+-----+-----+-----+  
only showing top 10 rows
```

# Connected Components



A connected component defines an (undirected) subgraph that has connections to itself but does not connect to the greater graph,

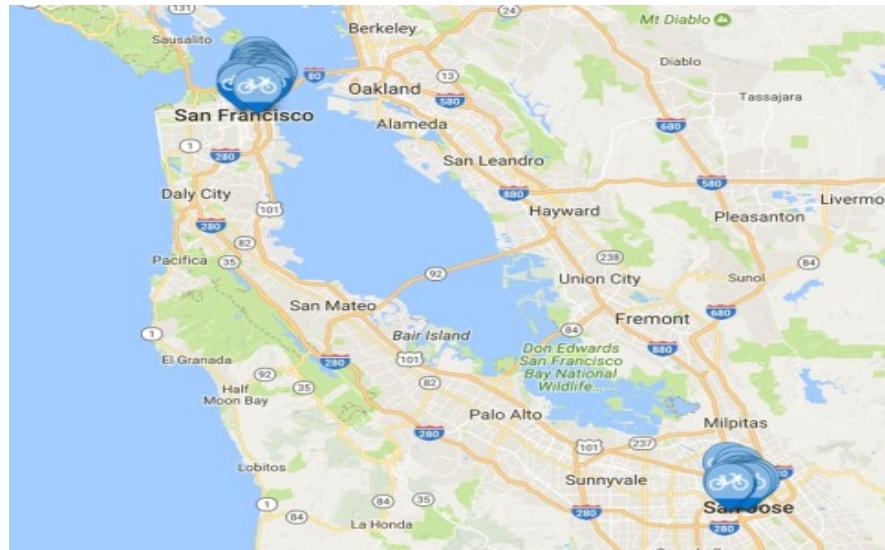




The connected components algorithm does not directly relate to our current problem because they assume an undirected graph.

However, we can still run the algorithm, which just assumes that there are no directionality associated with our edges.

In fact, if we look at the bike share map, we assume that we would get two distinct connected components





```
spark.sparkContext.setCheckpointDir("/tmp/checkpoints")
minGraph = GraphFrame(stationVertices, tripEdges.sample(False, 0.1))
cc = minGraph.connectedComponents()
```

From this query we get two connected components but not necessarily the ones we might expect.

Our sample may not have all of the correct data or information so we'd probably need more compute resources to investigate further

```
In [37]: cc.where("component != 0").show()
+-----+-----+-----+-----+-----+-----+-----+
|station_id|      id|      lat|      long|dockcount|      landmark|installation| component|
+-----+-----+-----+-----+-----+-----+-----+
|        47|Post at Kearney|37.788975|-122.403452|       19|San Francisco| 8/19/2013|317827579904|
|        46|Washington at Kea...|37.795425|-122.404767|       15|San Francisco| 8/19/2013| 17179869184|
+-----+-----+-----+-----+-----+-----+-----+
```



# Strongly Connected Components

GraphFrames includes another related algorithm that relates to directed graphs: strongly connected components, which takes directionality into account.

A strongly connected component is a subgraph that has paths between all pairs of vertices inside it

```
In [38]: scc = minGraph.stronglyConnectedComponents(maxIter=3)

In [39]: scc.groupBy("component").count().show()
+-----+-----+
| component | count |
+-----+-----+
| 8589934592 |    35 |
|          0 |    33 |
| 17179869184 |     1 |
| 317827579904 |     1 |
+-----+-----+
```



# Deep Learning

# Spark and Deep learning



Deep learning is one of the most exciting areas of development around Spark due to its ability to solve several previously difficult machine learning problems, especially those involving unstructured data such as images, audio, and text.

# What Is Deep Learning?

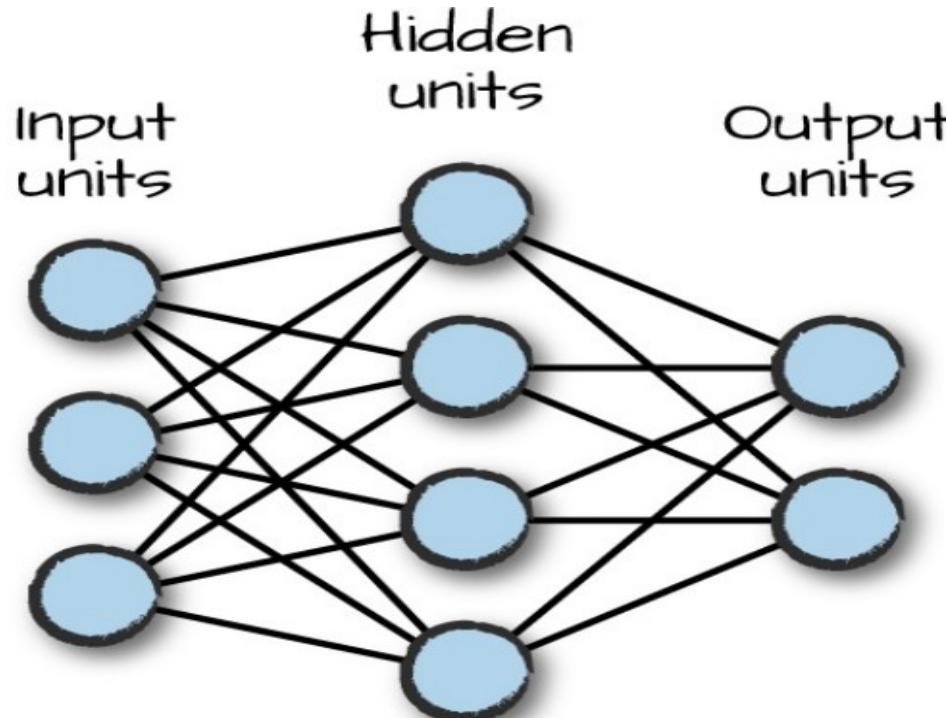


To define deep learning, we must first define neural networks.

- A neural network is a graph of nodes with weights and activation functions. These nodes are organized into layers that are stacked on top of one another.
- Each layer is connected, either partially or completely, to the previous layer in the network.
- By stacking layers one after the other, these simple functions can learn to recognize more and more complex signals in the input: simple lines with one layer, circles and squares with the next layer, complex textures in another, and finally the full object or output you hope to identify.
- The goal is to train the network to associate certain inputs with certain outputs by tuning the weights associated with each connection and the values of each node in the network.



# A neural network



# Ways of Using Deep Learning in Spark



## Inference

The simplest way to use deep learning is to take a pretrained model and apply it to large datasets in parallel using Spark.

For example,

We could use an image classification model, trained using a standard dataset like ImageNet, and apply it to your own image collection to identify pandas, flowers, or cars.



## Featurization and transfer learning

The next level of complexity is to use an existing model as a featurizer instead of taking its final output. Many deep learning models learn useful feature representations in their lower layers as they get trained for an end-to-end task.

For example,

A classifier trained on the ImageNet dataset will also learn low-level features present in all natural images, such as edges and textures.

We can then use these features to learn models for a new problem not covered by the original dataset.

This method is called **transfer learning**, and generally involves the last few layers of a pretrained model and retraining them with the data of interest.



## Model training

- Spark can also be used to train a new deep learning model from scratch. There are two common methods here.
  - 1) We can use a Spark cluster to parallelize the training of a single model over multiple servers, communicating updates between them. Alternatively,
  - 2) Some libraries let the user train multiple instances of similar models in parallel to try various model architectures and hyperparameters, accelerating the model search and tuning process.

In both cases, Spark's deep learning libraries make it simple to pass data from RDDs and DataFrames to deep learning algorithms.

# Deep Learning Libraries



## MLlib Neural Network Support

Spark's MLlib currently has native support for a single deep learning algorithm:

The **ml.classification.MultilayerPerceptronClassifier** class's multilayer perceptron classifier.

This class is limited to training relatively shallow networks containing fully connected layers with the sigmoid activation function and an output layer with a softmax activation function.

This class is most useful for training the last few layers of a classification model when using transfer learning on top of an existing deep learning-based featurizer.



## TensorFrames

TensorFrames is an inference and transfer learning-oriented library that makes it easy to pass data between Spark DataFrames and TensorFlow.

It supports Python and Scala interfaces and focuses on providing a simple but optimized interface to pass data from TensorFlow to Spark and back.



## BigDL

BigDL is a distributed deep learning framework for Apache Spark primarily developed by Intel.

It aims to support distributed training of large models as well as fast applications of these models using inference. One key advantage of BigDL over the other libraries described here is that it is primarily optimized to use CPUs instead of GPUs, making it efficient to run on an existing, CPU-based cluster (e.g., an Apache Hadoop deployment).

BigDL provides high-level APIs to build neural networks from scratch and automatically distributes all operations by default.

It can also train models described with the Keras DL library.



## TensorFlowOnSpark

TensorFlowOnSpark is a widely used library that can train TensorFlow models in a parallel fashion on Spark clusters.

TensorFlow includes some foundations to do distributed training, but it still needs to rely on a cluster manager for managing the hardware and data communications. It does not come with a cluster manager or a distributed I/O layer out of the box. TensorFlowOnSpark launches TensorFlow's existing distributed mode inside a Spark job, and automatically feeds data from Spark RDDs or DataFrames into the TensorFlow job.



## DeepLearning4J

DeepLearning4j is an open-source, distributed deep learning project in Java and Scala that provides both single-node and distributed training options.

One of its advantages over Python-based deep learning frameworks is that it was primarily designed for the JVM, making it more convenient for groups that do not wish to add Python to their development process.

It includes a wide variety of training algorithms and support for CPUs as well as GPUs.



## Deep Learning Pipelines

Deep Learning Pipelines is an open source package from Databricks that integrates deep learning functionality into Spark's ML Pipelines API.

The package existing deep learning frameworks (TensorFlow and Keras at the time of writing), but focuses on two goals:

- 1) Incorporating these frameworks into standard Spark APIs (such as ML Pipelines and Spark SQL) to make them very easy to use.
- 2) Distributing all computation by default

# Deep learning libraries



Library	Underlying DL framework	Use cases
BigDL	BigDL	Distributed training, inference, ML Pipeline integration
DeepLearning4J	DeepLearning4J	Inference, transfer learning, distributed training
Deep Learning Pipelines	TensorFlow, Keras	Inference, transfer learning, multi-model training, ML Pipeline and Spark SQL integration
MLlib Perceptron	Spark	Distributed training, ML Pipeline integration
TensorFlowOnSpark	TensorFlow	Distributed training, ML Pipeline integration
TensorFrames	TensorFlow	Inference, transfer learning, DataFrame integration



## A Simple Example with Deep Learning Pipelines



# Setup

```
$SPARK_HOME/bin/pyspark --packages databricks:tensorframes:0.6.0-s_2.11
```

The following packages may need to be installed.

```
apt install python-pip3  
apt install python-pip  
apt install python3-pip  
pip3 install keras  
pip3 install h5py  
pip3 install sparkdl  
pip3 install sparkdl==0.2.2 ( In case version specific )  
pip3 install Pillow  
pip3 install tensorflow  
pip3 install pandas  
pip3 install kafka-python  
pip3 install tensorflowonspark  
pip3 install jieba
```

# Images and DataFrames



```
In [1]: from sparkdl import readImages
Using TensorFlow backend.

In [2]: img_dir = '/home/ilg/Documents/sparkdata/data/deep-learning-images/'

In [3]: image_df = readImages(img_dir)

In [4]: image_df.printSchema()
root
|-- filePath: string (nullable = false)
|-- image: struct (nullable = true)
|   |-- mode: string (nullable = false)
|   |-- height: integer (nullable = false)
|   |-- width: integer (nullable = false)
|   |-- nChannels: integer (nullable = false)
|   |-- data: binary (nullable = false)
```



# Alternative

```
In [23]: from pyspark.ml.image import ImageSchema  
  
In [24]: img_dir = '/home/ilg/data/sample'  
  
In [25]: image_df = ImageSchema.readImages(img_dir)  
  
In [26]: image_df.printSchema()  
root  
| -- image: struct (nullable = true)  
| | -- origin: string (nullable = true)  
| | -- height: integer (nullable = false)  
| | -- width: integer (nullable = false)  
| | -- nChannels: integer (nullable = false)  
| | -- mode: integer (nullable = false)  
| | -- data: binary (nullable = false)
```



# Transfer Learning

Now that we have some data

we can get started with some simple transfer learning.

Note: This means leveraging a model that someone else created and modifying it to better suit our own purposes.

First Step :

We will load the data for each type of flower and create a training and test set:

```
In [27]: from pyspark.sql.functions import lit  
  
In [28]: tulips_df = ImageSchema.readImages(img_dir + "/tulips").withColumn("label", lit(1))  
  
In [29]: daisy_df = ImageSchema.readImages(img_dir + "/daisy").withColumn("label", lit(0))  
  
In [30]: tulips_train, tulips_test = tulips_df.randomSplit([0.6, 0.4])  
  
In [31]: daisy_train, daisy_test = daisy_df.randomSplit([0.6, 0.4])  
  
In [32]: train_df = tulips_train.unionAll(daisy_train)  
  
In [33]: test_df = tulips_test.unionAll(daisy_test)
```



## Second Step:

we will leverage a transformer called the **DeepImageFeaturizer**.

This will allow us to leverage a **pretrained** model called Inception, a powerful neural network successfully used to identify patterns in images.

The version we are using is pretrained to work well with images of various common objects and animals.

This is one of the standard pretrained models that ship with the **Keras** library. *However, this particular neural network **is not trained** to recognize daisies and roses.*

*So we're going to use transfer learning in order to make it into something useful for our own purposes: distinguishing different flower types.*



Note:

It's very resource intensive so may take a while to complete

```
In [34]: from pyspark.ml.classification import LogisticRegression
In [35]: from pyspark.ml import Pipeline
In [36]: from sparkdl import DeepImageFeaturizer
In [37]: featurizer = DeepImageFeaturizer(inputCol="image",outputCol="features",modelName="InceptionV3")
In [38]: lr = LogisticRegression(maxIter=1, regParam=0.05,elasticNetParam=0.3,labelCol="label")
In [39]: p = Pipeline(stages=[featurizer, lr])
In [40]: p_model = p.fit(train_df)
[Stage 8:> (0 + 1) / 2]■
```



Once we've trained the model, we can use the classification

```
In [41]: from pyspark.ml.evaluation import MulticlassClassificationEvaluator  
In [42]: tested_df = p_model.transform(test_df)  
In [43]: evaluator = MulticlassClassificationEvaluator(metricName="accuracy")  
In [44]: print("Test set accuracy = " + str(evaluator.evaluate(tested_df.select("prediction", "label"))))  
Test set accuracy = 1.0
```



# Pyspark avro

# Module



The spark-avro module is external and not included in spark-submit or spark-shell by default.

As with any Spark applications,

spark-submit is used to launch your application.

spark-avro\_2.12 and its dependencies can be directly added to spark-submit using --packages, such as, or pyspark....

```
ilg@Sparkle:~$ pyspark --packages org.apache.spark:spark-avro_2.11:2.4.0
```



## **to\_avro() and from\_avro() - Both functions are currently only available in Scala and Java.**

The Avro package provides function **to\_avro** to encode a column as binary in Avro format, and **from\_avro()** to decode Avro binary data into a column.

Both functions transform one column to another column, and the input/output SQL data type can be complex type or primitive type.

Using Avro record as columns are useful when reading from or writing to a streaming source like Kafka. Each Kafka key-value record will be augmented with some metadata, such as the ingestion timestamp into Kafka, the offset in Kafka, etc.

- If the “value” field that contains your data is in Avro, you could use **from\_avro()** to extract your data, enrich it, clean it, and then push it downstream to Kafka again or write it out to a file.
- **to\_avro()** can be used to turn structs into Avro records.

*This method is particularly useful when you would like to re-encode multiple columns into a single one when writing data out to Kafka.*

# pyspark avro example



```
ilg@Sparkle:~$ pyspark --packages org.apache.spark:spark-avro_2.11:2.4.0
```

## Custom from\_ & to\_



```
In [4]: from pyspark.sql.functions import col, struct
```

```
In [5]: avro_type_struct = """
...: {
...:     "type": "record",
...:     "name": "struct",
...:     "fields": [
...:         {"name": "col1", "type": "long"},
...:         {"name": "col2", "type": "string"}
...:     ]
...: }"""
```

```
In [6]: df = spark.range(10).select(struct(
...:     col("id"),
...:     col("id").cast("string").alias("id2")
...: ).alias("struct"))
```

```
In [7]: avro_struct_df = df.select(to_avro(col("struct")).alias("avro"))
```

```
In [8]: avro_struct_df.show(3)
```

```
+-----+
|      avro|
+-----+
|[00 02 30]|
|[02 02 31]|
|[04 02 32]|
+-----+
only showing top 3 rows
```

```
In [9]: avro_struct_df.select(from_avro("avro", avro_type_struct)).show(3)
```

```
+-----+
|from_avro(avro, struct<col1:bigint,col2:string>)|
+-----+
|          [0, 0]|
|          [1, 1]|
|          [2, 2]|
+-----+
only showing top 3 rows
```



## Example from docs

```
In [10]: avdf = spark.read.format("avro").load("/tmp/users.avro")
```

```
In [11]: avdf.select("name", "favorite_color").write.format("avro").save("/tmp/namesAndFavColors.avro")
```

```
ilg@Sparkle:/usr/local/spark/examples/src/main/resources$ ls -lt /tmp/
total 376
drwxr-xr-x 2 ilg ilg 4096 Sep 15 22:31 namesAndFavColors.avro
```

```
ilg@Sparkle:/usr/local/spark/examples/src/main/resources$ ls -lt /tmp/namesAndFavColors.avro/
total 4
-rw-r--r-- 1 ilg ilg 0 Sep 15 22:31 _SUCCESS
-rw-r--r-- 1 ilg ilg 241 Sep 15 22:31 part-00000-3481b043-57e0-475a-b13c-a43ab419e1c6-c000.avro
```



# Develop **XGBoost** Model in Python with scikit-learn

# XGBoost



XGBoost is an implementation of gradient boosted decision trees designed for speed and performance that is dominative competitive machine learning.

# Learning outcome



- Install XGBoost for use with Python.
- Problem definition and download dataset.
- Load and prepare data.
- Train XGBoost model.
- Make predictions and evaluate model.
- Tie it all together and run the example.



## Install XGBoost for Use in Python

XGBoost can be installed easily using pip.

```
$ sudo pip install xgboost
```

## Problem Description: Predict Onset of Diabetes



We are going to use the Pima Indians onset of diabetes dataset.

- This dataset is comprised of 8 input variables that describe medical details of patients and one output variable to indicate whether the patient will have an onset of diabetes within 5 years.
- We can learn more about this dataset on the UCI Machine Learning Repository website.
- This is a good dataset for a first XGBoost model because all of the input variables are numeric and the problem is a simple binary classification problem. It is not necessarily a good problem for the XGBoost algorithm because it is a relatively small dataset and an easy problem to model.

<https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv>

# Load and Prepare Data



We will load the data from file and prepare it for use for training and evaluating an XGBoost model.

We will start off by importing the classes and functions we intend to use in this example.

In jupyter, type the following:

```
from numpy import loadtxt  
from xgboost import XGBClassifier  
from sklearn.model_selection import train_test_split  
from sklearn.metrics import accuracy_score
```

## Loading the csv



Next, we can load the CSV file as a NumPy array using the NumPy function `loadtxt()`.

```
# load data  
  
dataset = loadtxt('pima-indians-diabetes.data', delimiter=',')
```



We must separate the columns (attributes or features) of the dataset into input patterns (X) and output patterns (Y).

We can do this by specifying the column indices in the NumPy array format.

```
# split data into X and Y  
X = dataset[:,0:8]  
Y = dataset[:,8]
```



# Training & Test data

Finally, we must split the X and Y data into a training and test dataset.

The training set will be used to prepare the XGBoost model and the test set will be used to make new predictions, from which we can evaluate the performance of the model.

For this we will use the `train_test_split()` function from the scikit-learn library.

We also specify a seed for the random number generator so that we always get the same split of data each time this example is executed.

```
# split data into train and test sets
seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=test_size,
random_state=seed)
```

- We are now ready to train our model.



## Train the XGBoost Model

XGBoost provides a wrapper class to allow models to be treated like classifiers or regressors in the scikit-learn framework.

This means we can use the full scikit-learn library with XGBoost models.

The XGBoost model for classification is called XGBClassifier.

We can create and fit it to our training dataset. Models are fit using the scikit-learn API and the `model.fit()` function.

Parameters for training the model can be passed to the model in the constructor. Here, we use the sensible defaults.

```
# fit model no training data  
model = XGBClassifier()  
model.fit(X_train, y_train)
```

We can see the parameters used in a trained model by printing the model, for example:

```
print(model)
```

We are now ready to use the trained model to make predictions.

## Make Predictions with XGBoost Model



We can make predictions using the fit model on the test dataset.

To make predictions we use the scikit-learn function **model.predict()**.

By default, the predictions made by XGBoost are probabilities. Because this is a binary classification problem, each prediction is the probability of the input pattern belonging to the first class. We can easily convert them to binary class values by rounding them to 0 or 1.

```
# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]
```



Now that we have used the fit model to make predictions on new data, we can evaluate the performance of the predictions by comparing them to the expected values.

For this we will use the built in `accuracy_score()` function in scikit-learn.

```
# evaluate predictions  
accuracy = accuracy_score(y_test, predictions)  
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

# Full Code



```
# First XGBoost model for Pima Indians dataset
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# load data
dataset = loadtxt('pima-indians-diabetes.data', delimiter=',')
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
# split data into train and test sets
seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=test_size,
random_state=seed)
# fit model no training data
model = XGBClassifier()
model.fit(X_train, y_train)
# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]
# evaluate predictions
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```



Running this example produces the following output.

**Accuracy: 77.95%**

This is a good accuracy score on this problem, which we would expect, given the capabilities of the model and the modest complexity of the problem.

[https://xgboost.readthedocs.io/en/latest/python/python\\_api.html#module-xgboost.sklearn](https://xgboost.readthedocs.io/en/latest/python/python_api.html#module-xgboost.sklearn)

<https://xgboost.readthedocs.io/en/latest//parameter.html>