Introduction to PySpark

What is PySpark?

PySpark is a Spark library written in Python to run Python applications using Apache Spark capabilities, using PySpark we can run applications parallelly on the distributed cluster (multiple nodes).

In other words, PySpark is a Python API for Apache Spark. Apache Spark is an analytical processing engine for large scale powerful distributed data processing and machine learning applications.

As discussed PySpark is the Python API for Apache Spark, an open source, distributed computing framework and set of libraries for real-time, large-scale data processing. If you're already familiar with Python and libraries such as Pandas, then PySpark is a good language to learn to create more scalable analyses and pipelines.

Spark basically written in Scala and later on due to its industry adaptation it's API PySpark released for Python using Py4J. Py4J is a Java library that is integrated within PySpark and allows python to dynamically interface with JVM objects, hence to run PySpark you also need Java to be installed along with Python, and Apache Spark.

Additionally, For the development, you can use Anaconda distribution (widely used in the Machine Learning community) which comes with a lot of useful tools like Spyder IDE, Jupyter notebook to run PySpark applications.

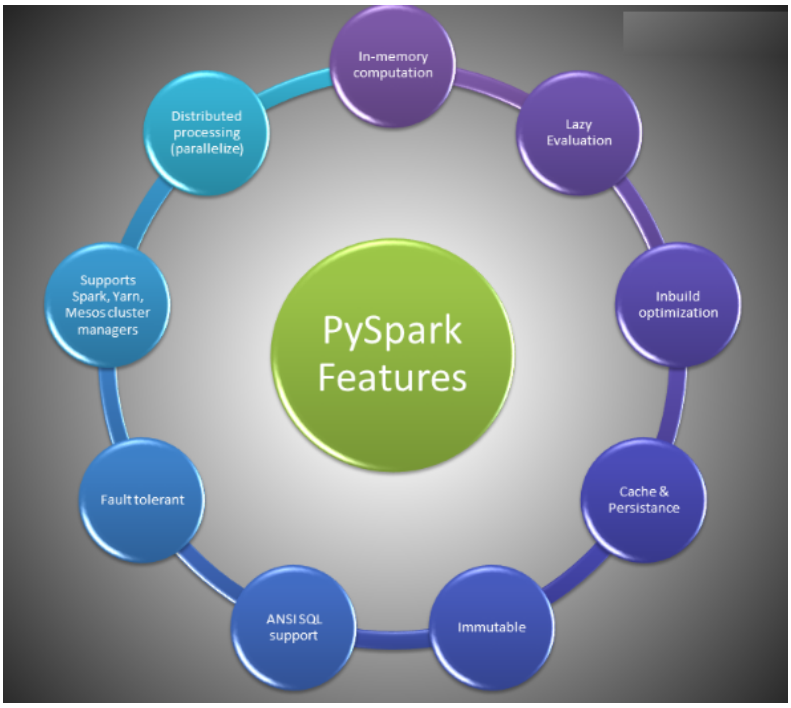
In real-time, PySpark has used a lot in the machine learning & Data scientists community; thanks to vast python machine learning libraries. Spark runs operations on billions and trillions of data on distributed clusters 100 times faster than the traditional python applications.

Who uses PySpark?

PySpark is very well used in Data Science and Machine Learning community as there are many widely used data science libraries written in Python including NumPy, TensorFlow. Also used due to its efficient processing of large datasets. PySpark has been used by many organizations like Walmart, Trivago, Sanofi, Runtastic, and many more.

Features

Following are the main features of PySpark -



- **In-memory computation -** PySpark loads the data from disk and process in memory and keeps the data in memory, this is the main difference between PySpark and Mapreduce (I/O intensive). In between the transformations, we can also cache/persists the RDD in memory to reuse the previous computations.
- **Distributed processing using parallelize -** When you create RDD (Resilient Distributed Dataset) from a data, It by default partitions the elements in a RDD. By default it partitions to the number of cores available.
- **Fault-tolerant -** PySpark operates on fault-tolerant data stores on HDFS, S3 e.t.c hence any RDD operation fails, it automatically reloads the data from other partitions. Also, When PySpark applications running on a cluster, PySpark task failures are automatically recovered for a certain number of times (as per the configuration) and finish the application seamlessly.
- **Immutable -** PySpark RDD's are immutable in nature meaning, once RDDs are created you cannot modify. When we apply transformations on RDD, PySpark creates a new RDD and maintains the RDD Lineage.
- **Lazy evaluation -** PySpark does not evaluate the RDD transformations as they appear/encountered by Driver instead it keeps the all transformations as it encounters(DAG) and evaluates the all transformation when it sees the first RDD action.
- **Can be used with many cluster managers (Spark, Yarn (Yet Another Resource Negotiator), Mesos etc)**
- **Cache & persistence**
- **Inbuild-optimization when using DataFrames**
- **Supports ANSI SQL**

PySpark RDD (Resilient Distributed Dataset) Limitations

PySpark RDDs are not much suitable for applications that make updates to the state store such as storage systems for a web application. For these applications, it is more efficient to use systems that perform traditional update logging and data checkpointing, such as databases. The goal of RDD is to provide an efficient programming model for batch analytics and leave these asynchronous applications.

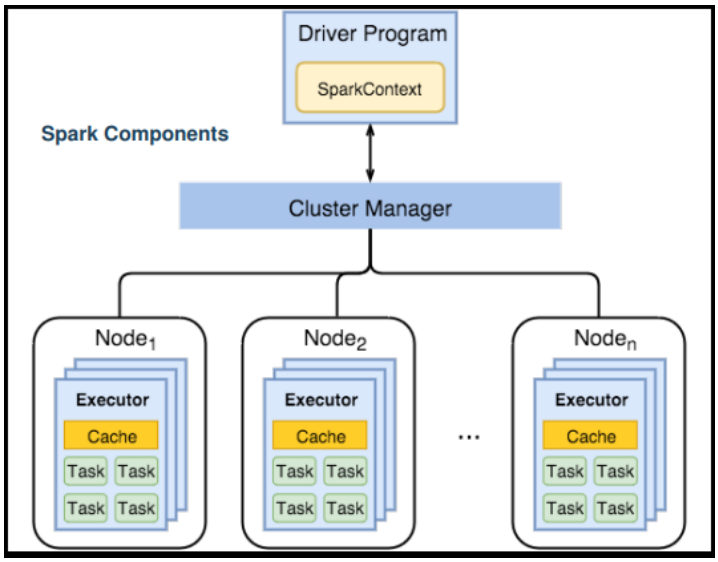
Advantages of PySpark

- PySpark is a general-purpose, in-memory, distributed processing engine that allows you to process data efficiently in a distributed fashion.
- Applications running on PySpark are 100x faster than traditional systems.
- You will get great benefits using PySpark for data ingestion pipelines.
- Using PySpark we can process data from Hadoop HDFS, AWS S3, and many file systems.
- PySpark also is used to process real-time data using Streaming and Kafka.
- Using PySpark streaming you can also stream files from the file system and also stream from the socket.
- PySpark natively has machine learning and graph libraries.

Core Concepts of Apache Spark

Most of the following content comes from [Kirillov2016]. So the copyright belongs to Anton Kirillov. We will refer you to get more details from Apache Spark core concepts, architecture and internals. Before diving deep into how Apache Spark works, lets understand the jargon of Apache Spark

- **Job:** A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- **Stages:** Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- **Tasks:** Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- **DAG:** DAG stands for *Directed Acyclic Graph*, in the present context its a DAG of operators.
- **Executor:** The process responsible for executing a task.
- **Master:** The machine on which the Driver program runs
- **Slave:** The machine on which the Executor program runs



1. Spark Driver

- separate process to execute user applications
- creates SparkContext to schedule jobs execution and negotiate with cluster manager

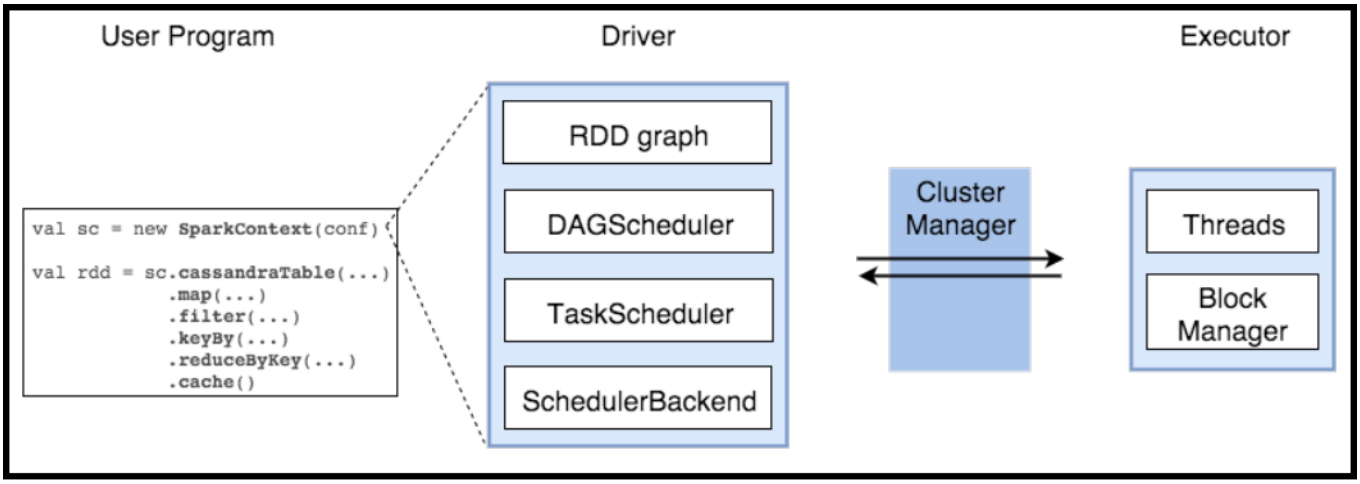
2. Executors

- run tasks scheduled by driver
- store computation results in memory, on disk or off-heap
- interact with storage systems

3. Cluster Manager

- Mesos
- YARN (Yet Another Resource Negotiator)
- Spark Standalone

Spark Driver contains more components responsible for translation of user code into actual jobs executed on cluster:



- **SparkContext -**
  – represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster

- **Accumulators -**
  – PySpark Accumulators are another type shared variable that are only “added” through an associative and commutative operation and are used to perform counters (Similar to Map-reduce counters) or sum operations. PySpark by default supports creating an accumulator of any numeric type and provides the capability to add custom accumulator types. Programmers can create named accumulators and unnamed accumulators

- **DAGScheduler -**
  – computes a DAG of stages for each job and submits them to TaskScheduler determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs

- **TaskScheduler -**
  – responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers

- **SchedulerBackend -**
  – backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN (Yet Another Resource Negotiator), Standalone, local)

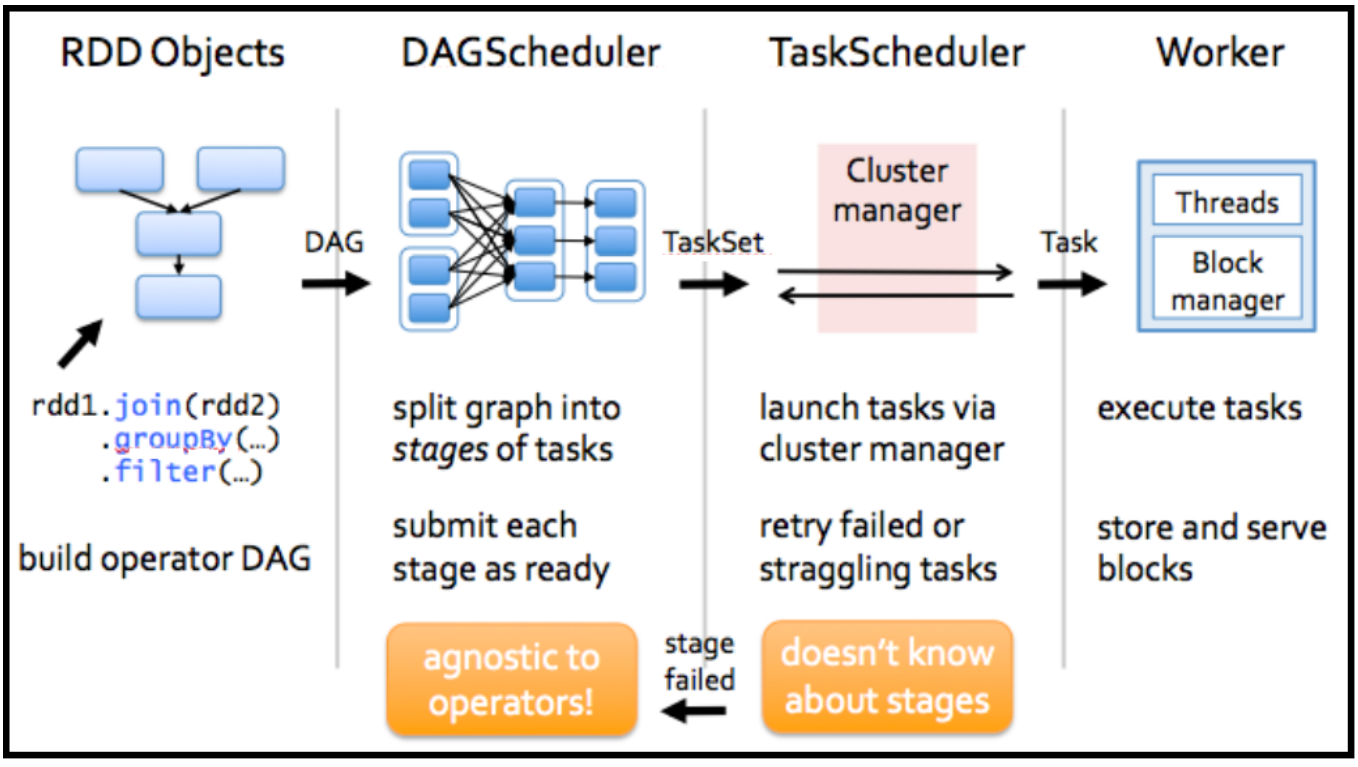
- **BlockManager -**
  – provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

How does Spark Architecture work?

Spark has a small code base and the system is divided in various layers. Each layer has some responsibilities. The layers are independent of each other.

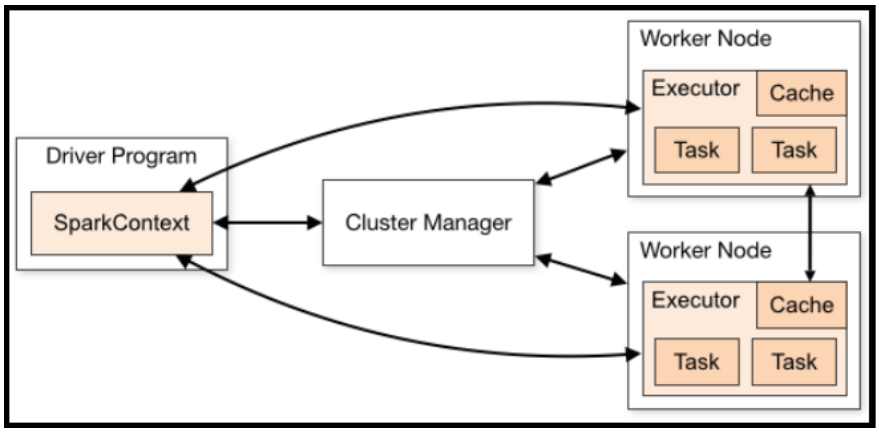
The first layer is the interpreter, Spark uses a Scala interpreter, with some modifications. As you enter your code in spark console (creating RDD’s and applying operators), Spark creates a operator graph. When the user runs an action (like collect), the Graph is submitted to a DAG Scheduler. The DAG scheduler divides operator graph into (map and reduce) stages. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together to optimize the graph. For e.g. Many map operators can be scheduled in a single stage.

This optimization is key to Sparks performance. The final result of a DAG scheduler is a set of stages. The stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager. (Spark Standalone/YARN/Mesos). The task scheduler doesn’t know about dependencies among stages.



PySpark Architecture

Apache Spark works in a master-slave architecture where the master is called “Driver” and slaves are called “Workers”. When you run a Spark application, Spark Driver creates a context that is an entry point to your application, and all operations (transformations and actions) are executed on worker nodes, and the resources are managed by Cluster Manager.



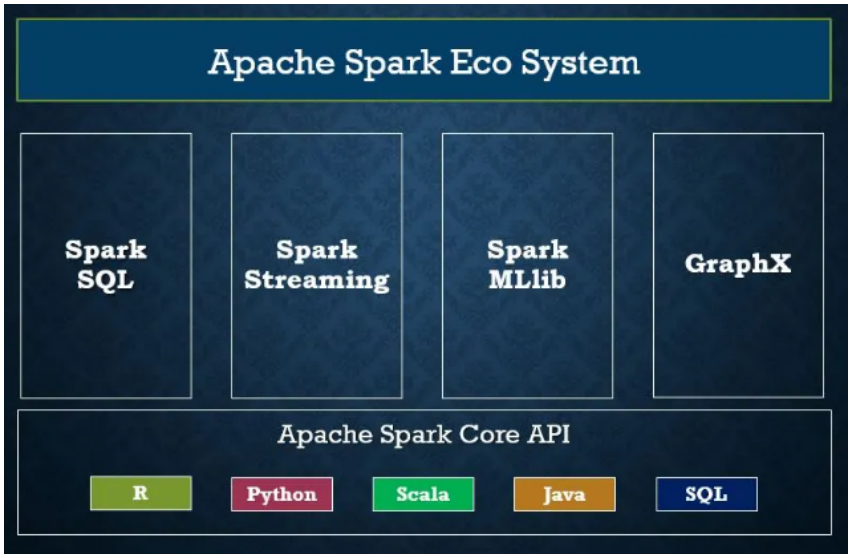
Cluster Manager Types

As of writing this Spark with Python (PySpark) tutorial, Spark supports below cluster managers:

- **Standalone –** a simple cluster manager included with Spark that makes it easy to set up a cluster.
- **Apache Mesos –** Mesons is a Cluster manager that can also run Hadoop MapReduce and PySpark applications.
- **Hadoop YARN (Yet Another Resource Negotiator) –** the resource manager in Hadoop 2. This is mostly used, cluster manager.
- **Kubernetes –** an open-source system for automating deployment, scaling, and management of containerized applications.
- **local –** which is not really a cluster manager but still I wanted to mention as we use “local” for master() in order to run Spark on your laptop/computer.

PySpark Modules & Packages

- **PySpark RDD** (Resilient Distributed Dataset)
- **PySpark DataFrame and SQL**
- **PySpark Streaming**
- **PySpark MLib**
- **PySpark GraphFrames**
- **PySpark Resource**, It is new in PySpark 3.0



Besides these, if you wanted to use third-party libraries, you can find them at https://spark-packages.org/ (https://spark-packages.org/) . This page is kind of a repository of all Spark third-party libraries.

PySpark RDD – Resilient Distributed Dataset

PySpark **RDD (Resilient Distributed Dataset)** is a fundamental data structure of PySpark that is fault-tolerant, immutable distributed collections of objects, which means once you create an RDD you cannot change it. Each dataset in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.

RDD Creation

In order to create an RDD, first, you need to create a SparkSession which is an entry point to the PySpark application. SparkSession can be created using a builder() or newSession() methods of the SparkSession.

Spark session internally creates a sparkContext variable of SparkContext. You can create multiple SparkSession objects but only one SparkContext per JVM. In case if you want to create another new SparkContext you should stop existing Sparkcontext (using stop()) before creating a new one.

master() – If you are running it on the cluster you need to use your master name as an argument to master(). usually, it would be either YARN (Yet Another Resource Negotiator) or mesos depends on your cluster setup.

local[x] - Use it when running in Standalone mode. x should be an integer value and should be greater than 0; this represents how many partitions it should create when using RDD, DataFrame, and Dataset. Ideally, x value should be the number of CPU cores you have.

appName() – Used to set your application name.

getOrCreate() – This returns a SparkSession object if already exists, and creates a new one if not exist.

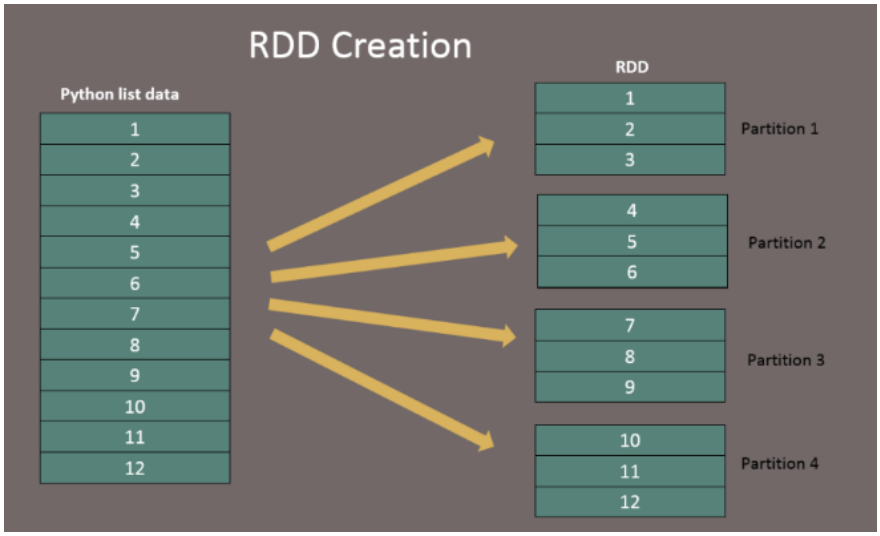
In [8]:
```
# Import SparkSession
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder \
      .master("local[1]") \
      .appName("PySparkExamples") \
      .getOrCreate()
```

Create RDD using sparkContext.parallelize()
By using parallelize() function of SparkContext (sparkContext.parallelize()) you can create an RDD. This function loads the existing collection from your driver program into parallelizing RDD. This is a basic method to create RDD and is used when you already have data in memory that is either loaded from a file or from a database. and it required all data to be present on the driver program prior to creating RDD.

For production applications, we mostly create RDD by using external storage systems like HDFS, S3, HBase e.t.c.

In [9]:
```
# Create RDD from parallelize
data = [1,2,3,4,5,6,7,8,9,10,11,12]
rdd=spark.sparkContext.parallelize(data)
```



Using textFile()
RDD can also be created from a text file using textFile() function of the SparkContext.

In [ ]:
```
# Create RDD from external Data source
rdd2 = spark.sparkContext.textFile("/path/test.txt")
```

Once you have an RDD, you can perform transformation and action operations. Any operation you perform on RDD runs in parallel.

RDD Operations

On PySpark RDD, you can perform two kinds of operations.

- **RDD transformations –** Transformations are lazy operations. When you run a transformation(for example update), instead of updating a current RDD, these operations return another RDD.
- **RDD actions –** operations that trigger computation and return RDD values to the driver.

RDD Transformations

Transformations on Spark RDD returns another RDD and transformations are lazy meaning they don't execute until you call an action on RDD. Some transformations on RDD's are flatMap(), map(), reduceByKey(), filter(), sortByKey() and return new RDD instead of updating the current.

RDD Actions

RDD Action operation returns the values from an RDD to a driver node. In other words, any RDD function that returns non RDD[T] is considered as an action.

Some actions on RDDs are count(), collect(), first(), max(), reduce() and more.

PySpark DataFrame

DataFrame definition by Databricks is -

DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as structured data files, tables in Hive, external databases, or existing RDDs.
– Databricks

PySpark DataFrame is mostly similar to Pandas DataFrame with the exception PySpark DataFrames are distributed in the cluster (meaning the data in DataFrame's are stored in different machines in a cluster) and any operations in PySpark executes in parallel on all machines whereas Panda Dataframe stores and operates on a single machine.

Due to parallel execution on all cores on multiple machines, PySpark runs operations faster then pandas. In other words, pandas DataFrames run operations on a single node whereas PySpark runs on multiple machines.

DataFrame creation

The simplest way to create a DataFrame is from a Python list of data. DataFrame can also be created from an RDD and by reading files from several sources.

using createDataFrame()
By using createDataFrame() function of the SparkSession you can create a DataFrame.

In [ ]:
```
dataset = [('James','','Smith','1991-04-01','M',3000),
           ('Michael','Rose','','2000-05-19','M',4000),
           ('Robert','','Williams','1978-09-05','M',4000),
           ('Maria','Anne','Jones','1967-12-01','F',4000),
           ('Jen','Mary','Brown','1980-02-17','F',-1)]

columns = ["firstname","middlename","lastname","dob","gender","salary"]
df = spark.createDataFrame(data = dataset, schema = columns)
```

Since DataFrame's are structure format which contains names and columns, we can get the schema of the DataFrame using df.printSchema()

df.show() shows the 20 elements from the DataFrame.

```
+---------+----------+--------+----------+------+------+
|firstname|middlename|lastname|dob       |gender|salary|
+---------+----------+--------+----------+------+------+
|James    |          |Smith   |1991-04-01|M     |3000  |
|Michael  |Rose      |        |2000-05-19|M     |4000  |
|Robert   |          |Williams|1978-09-05|M     |4000  |
|Maria    |Anne      |Jones   |1967-12-01|F     |4000  |
|Jen      |Mary      |Brown   |1980-02-17|F     |-1    |
+---------+----------+--------+----------+------+------+
```

DataFrame operations

Like RDD, DataFrame also has operations like Transformations and Actions.

DataFrame from external data sources

In real-time applications, DataFrames are created from external sources like files from the local system, HDFS, S3 Azure, HBase, MySQL table etc. Below is an example of how to read a CSV file from a local system.

In [ ]:
```
df = spark.read.csv("/tmp/resources/zipcodes.csv")
df.printSchema()
```

Supported file formats

DataFrame has a rich set of API which supports reading and writing several file formats

- **csv -** A comma-separated values file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format.
- **text -** A text file is a kind of computer file that is structured as a sequence of lines of electronic text. A text file exists stored as data within a computer file system.
- **avro -** Avro files include markers that can be used to split large data sets into subsets suitable for Apache MapReduce processing. Some data exchange services use a code generator to interpret the data definition and produce code to access the data. Avro doesn't require this step, making it ideal for scripting languages.
- **parquet -** Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk. Apache Parquet is designed to be a common interchange format for both batch and interactive workloads.
- **tsv -** A tab-separated values file is a simple text format for storing data in a tabular structure, e.g., a database table or spreadsheet data, and a way of exchanging information between databases. Each record in the table is one line of the text file.
- **xml -** An XML file is a file used to store data in the form of hierarchical elements. Data stored in XML files can be read by computer programs with the help of custom tags, which indicate the type of element.
- **json -** JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays. It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers.

and many more

PySpark SQL

PySpark SQL is one of the most used PySpark modules which is used for processing structured columnar data format. Once you have a DataFrame created, you can interact with the data by using SQL syntax.

In other words, Spark SQL brings native RAW SQL queries on Spark meaning you can run traditional ANSI SQL's on Spark Dataframe, in the later section of this PySpark SQL tutorial, you will learn in detail using SQL select, where, group by, join, union etc.

In order to use SQL, first, create a temporary table on DataFrame using createOrReplaceTempView() function. Once created, this table can be accessed throughout the SparkSession using sql() and it will be dropped along with your SparkContext termination.

Use sql() method of the SparkSession object to run the query and this method returns a new DataFrame.

In [ ]:
```
df.createOrReplaceTempView("PERSON_DATA")
df2 = spark.sql("SELECT * from PERSON_DATA")
df2.printSchema()
df2.show()
```

Let's see another pyspark example using group by.

In [ ]:
```
groupDF = spark.sql("SELECT gender, count(*) from PERSON_DATA group by gender")
groupDF.show()
```

This yields the below output

```
+------+--------+
|gender|count(1)|
+------+--------+
|     F|       2|
|     M|       3|
+------+--------+
```

PySpark Streaming

PySpark Streaming is a scalable, high-throughput, fault-tolerant streaming processing system that supports both batch and streaming workloads. It is used to process real-time data from sources like file system folder, TCP socket, S3, Kafka, Flume, Twitter, and Amazon Kinesis to name a few. The processed data can be pushed to databases, Kafka, live dashboards etc.

PySpark GraphFrames

PySpark GraphFrames are introduced in Spark 3.0 version to support Graphs on DataFrame's. Prior to 3.0, Spark has GraphX library which ideally runs on RDD and loses all Data Frame capabilities.

"GraphFrames is a package for Apache Spark which provides DataFrame-based Graphs. It provides high-level APIs in Scala, Java, and Python. It aims to provide both the functionality of GraphX and extended functionality taking advantage of Spark DataFrames. This extended functionality includes motif finding, DataFrame-based serialization, and highly expressive graph queries."
– GRAPHFRAMES.GITHUB.IO

Difference between GraphX and GraphFrame is that GraphX works on RDDs whereas GraphFrames works with DataFrames.

In [ ]:
```
```