**Apache Spark**

**Started in 2009 zaharia**

**What is Apache spark**

**Open source data processing engine to store and process data in real-time across clusters using simpler programming languages (Python, Scala, R)**

**Hadoop Vs Spark**

|  |  |
| --- | --- |
| **Hadoop** | **Spark** |
| **Mapreduce for processing the data** | **Uses in-memory and 100x faster** |
| **Performs batch processing (mapping / reducing) written to hdfs to and from** | **Batch and real time processing (retail giant transactions)** |
| **Takes more to execute, need to write own mapper, reducer** | **Few lines of code, dynamically written code.** |
| **Supports Kerberos auth’s which is difficult to manage** | **Supports auth via shared secret, also can run on yarn utilising the capability of kerberos** |

**Spark features:**

* **Fast processing 🡪 mainly attained by use of RDD(resilient distributed datasets).**
* **In-memory/caching 🡪 caching represents storing a copy of the data for faster accessing and processing and in-memory comes into picture when there is action performed.**
* **Flexible 🡪 supports multiple languages(Python, Scala, R).**
* **Fault tolerance 🡪 RDD are designed to overcome any data failure through execution plans(literally stores how to execute unless an action is specified).**
* **Better analytics 🡪 has rich set of SQL queries, ML Algorithms, complex analytics which are helpful for analytics.**

**Components of Spark:**

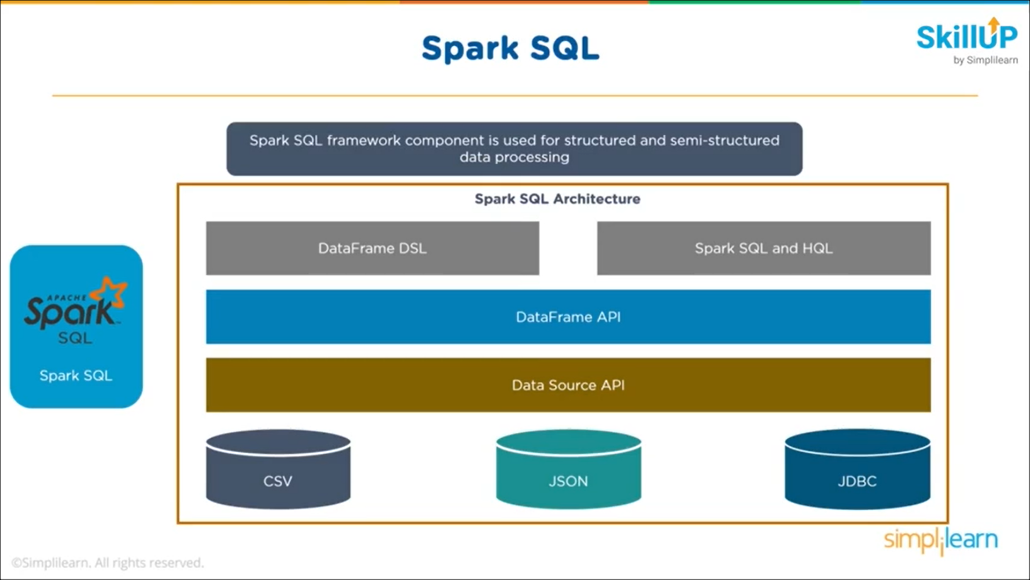
* **Spark-core 🡪 core engine 🡪 has RDD**
* **Spark SQL 🡪 has dataframes / datasets used to process the data**
* **Spark Streaming 🡪 process the incoming the data 🡪 transform/analyse**
* **MLlib 🡪 predictive analysis**
* **GraphX 🡪 data has network kind of flow (visualisation of data)**

**Spark Core:**

* **Base engine for large scale processing and distributed data processing**
* **Has RDD as they are Build Blocks**
* **Spark Core is responsible for**
  + **Memory management**
  + **Fault recovery**
  + **Scheduling, distributing and monitoring the jobs on cluster**
  + **Interacting with storage systems**
* **As it doesn’t have its own storage systems it relies on HDFS / NoSQL(HBase) / RDBMS.**
* **RDD (Resilient Distributed Datasets)**
  + **Immutable, fault tolerable**
  + **Has mainly 2 operations**
    - **Transformation**
    - **Action**
  + **When we instantiate a value an RDD is created which is distributed across the system**
    - **Val x = sc.textFile 🡪 only creates as an RDD but do not perform any function**
  + **Transformations (map, join, union) creates a new RDD when applied on a SparkContext (only creates execution logic)**
  + **Actions (show, count, first, reduce) 🡪 these triggers the operations on the transformations and yields the output by fallowing the DAG which has stored the execution plan from the starting.**

**Spark SQL:**

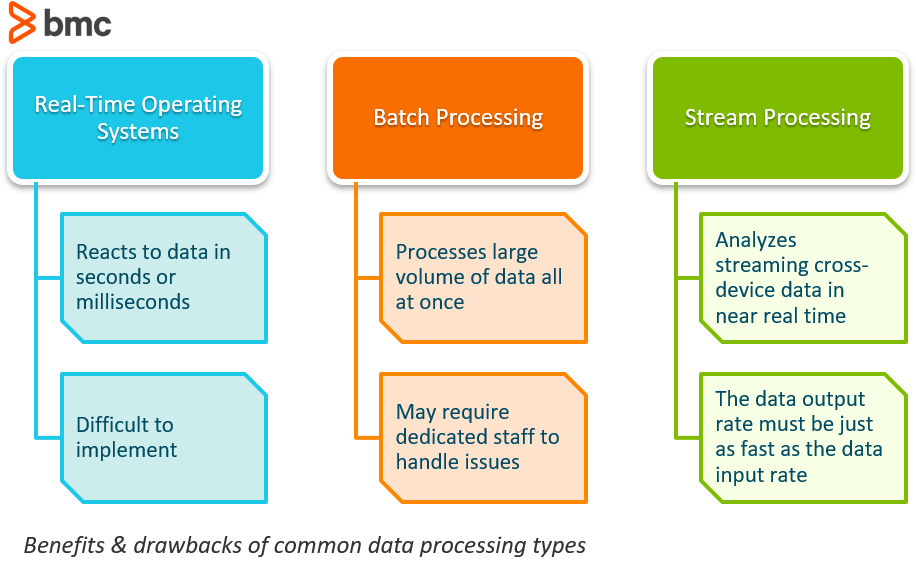
* **Framework component used for structured (RDMS, CSV’s, text file, json) and semi-structured data-processing.**

****

* **DataSource API 🡪 mainly used for reading the data in different formats**
* **DataFrame API 🡪 data which can be represent in rows / columns with column headings usually in tabular format.**
  + **Above API’s components are created using SparkContext**

**Spark Streaming:**

* **A lightweight API that allows to perform batch processing and real-time processing**

****

* **Provides secure, reliable and fast processing of live input data streams**

**MLlib:**

* **Low level ML Library that is simple to use, scalable and compatible with multiple programming languages**

**GraphX:**

* **Sparks own graph computation engine and data store**
* **Data having network connections x 🡪 y 🡪 z**

**Spark Architecture:**

* **Spark can run with different cluster managers yarn/ mesos**
* **Can work in standalone 🡪 without Hadoop**
* **Uses Master & Slave nodes**
* **Multiple executors which runs on slave nodes**
  + **Container 🡪 combination of RAM and CPU Cores**
* **For any spark application master node which has Driver program which internally has SparkContext interacts with cluster manager (resource manager in YARN) and distribute the tasks to multiple slave nodes (basically to node manager) .**
* **The resource manager requests the node manager to create a container for the execution 🡪 then AppMaster is created which is responsible for Spark application (or MapReduce) uses the containers to run the tasks.**
* **Within the container executer resides which takes up the tasks to complete**

**Applications of Spark:**

* **Banks (JP Morgan) 🡪 transactions**
* **E-Commerce (Alibaba) 🡪 retail transactions**
* **HealthCare (IQVIA) 🡪 to analyse patients data**
* **Entertainment (Netflix)**

[**https://lms.simplilearn.com/courses/4238/Apache-Spark-Beginners-Course/syllabus**](https://lms.simplilearn.com/courses/4238/Apache-Spark-Beginners-Course/syllabus)

* + - **Spark Streaming**

**Shared Variables: Accumulators / Broadcast variables**

* **Accumulators 🡪 variables that are added through associative and commutative operation**
* **Broadcast 🡪 allows the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks**

**Spark SQL:**

* **Sparks module for working with structured & semi-structured data**
* **Originated to overcome limitations of HIVE**
  + **Hive lags in performance as it uses MapReduce jobs for executing adhoc queries**
  + **No resume option if the job fails in middle**
* **It has 3 main Layers**
  + **Language API 🡪 compatible with multiple languages**
  + **SchemaRDD 🡪 as Spark works on schema/tables and internally creates the schema(or temporary tables)**
  + **Data Sources 🡪 supports multiple data sources(csv, json, text, parquet, hive tables)**
* **DataFrame API**
  + **It is a domain specific Language(DSL) for working with Structured and semi structured data (datasets with schema)**
  + **Dataset 🡪 distributed collection of data**
* **DataSource API**
  + **Supports working on various data sources through DataFrame interface**
  + **Can be accessed through SQL Context and Hive Context**

**Spark SQL Query optimiser:**

* **Leverages programming languages features to build extensible query optimizer**
* **Working**
  + **Analysing logical plan to resolve references**
  + **Logical plan optimisation**
  + **Physical planning**
  + **Code generation to compile parts of the query to java byte code**

**SQL Context:**

* **An entry point for utilising the functionalities of Spark SQL**
* **Older version (1.6)**
  + **Spark 🡪SparkSession is automatically created**
  + **Sc 🡪SparkContext**
* **Starting a spark shell**
  + **Spark-shell**
  + **Create SQL Context 🡪 SQLContext(sc)**
* **Create a spark session**
  + **Spark = SparkSession**

**.builder.appName(“APP”)**

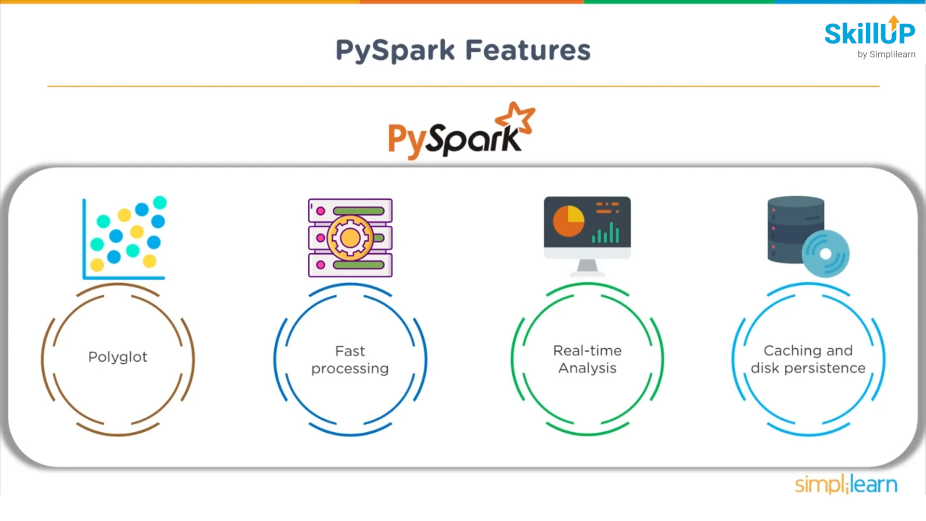
**.config(“spark.some.config.option”, “some-value”)**

**.getOrCreate()**

* **Pyspark 🡪 Python API to support Spark**

**Features**

* + **Polyglot 🡪 as it supports multiple languages**
  + **Fast processing**
  + **Real time processing**

****

**Pyspark Contents**

* **Sparkconf**
* **Sparkcontext**
* **Spark files 🡪 upload files using the methods 🡪 sc.addFile and get on worker node using SparkFiles.get() 🡪 to use import the SparkFiles module from pyspark**
* **RDD**
* **Storage level**
* **Data Frames**
* **Broadcast / Accumulators**

**Pyspark Conf 🡪 provides configurations to run spark application**

**Common attributes for spark**

* **Set(key, value) 🡪 to set a configuration property**
* **setAppName(value) 🡪 to set the master**
* **setMaster(value) 🡪 to set the master URL**
* **get(key, default=None) 🡪 to get a configuration value of a particular key**

**SparkContext 🡪 main entry point to the Spark Program**

**When to use what 🡪 sparkcontext, sparksession, hive context**

**RDD:**

**Distributed collection of immutable data**

**Note:**

**Not all the methods works in both RDD and spark dataframes**

**Storage Level:**

* **decides where to store the data of an RDD whether in memory or in the disk or both**

**------------------------------------storagelevel.py-------------------------------------**

**from pyspark import SparkContext**

**import pyspark**

**sc = SparkContext (**

**"local",**

**"storagelevel app"**

**)**

**rdd1 = sc.parallelize([1,2])**

**rdd1.persist( pyspark.StorageLevel.MEMORY\_AND\_DISK\_2 )**

**rdd1.getStorageLevel()**

**print(rdd1.getStorageLevel())**

**DataFrames:**

**Table of Data with rows / columns**

**Characteristics of RDD;**

* **Immutable**
* **Lazily evaluated**
* **Distribution**

**Ways to create Dataframe**

* **Created using different data formats**
* **Using existing RDD**
* **Programmatically specifying schema**

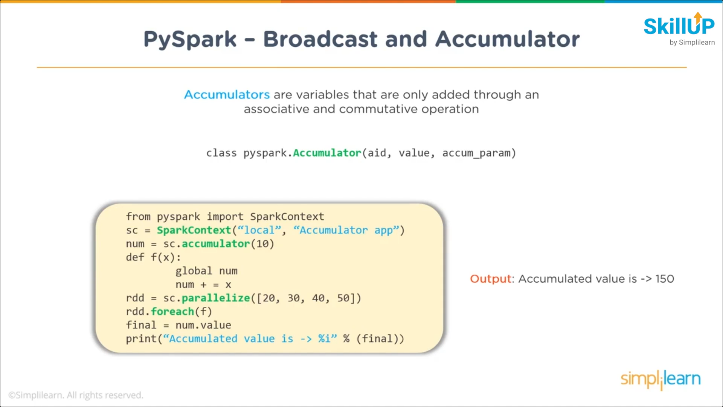
**Broadcast**

**Broadcast variables allows programmers to keep a read only value cached on each machine rather than shipping the whole copy and its tasks**

* **Create with SparkContext.broadcast(values)**

**Accumulator:**

**Variable that are added through assosiative and commutative operation**

****

**Subpackages in Pyspark:**

****

### What is Spark

[**Apache Spark**](https://spark.apache.org/)**is an open-source cluster-computing framework. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since. Spark it is a fast and general engine for large-scale data processing.**[**Databricks**](https://databricks.com/)**is a company founded by the creators of Apache Spark, that aims to help clients with cloud-based big data processing using Spark.**

**Traditional analysis tools like R and Python Pandas run on a single machine but data are growing faster than computation speed.  
The Opportunity: Cloud computing is a game-changer.  
It provides access to low-cost computing and storage.  
Distributing data over cluster of machines means lots of hard drives, lots of CPUs but also lots of memory !  
Storage is getting cheaper but stalling CPU speeds are the bottlenecks.  
A new Opportunity: Keep more data in-memory! In-memory can make a big difference, up to 100x faster. Spark is a new distributed execution engine that leverages the in-memory paradigm.**

**The challenge with cloud computing has always been programming the resources.  
Spark is developed in Scala and - besides Scala itself - supports other languages such as Java and Python. We are using for this example the Python programming interface to Spark (pySpark). pySpark provides an easy-to-use programming abstraction and parallel runtime: “Here’s an operation, run it on all of the data”**

#### **Spark Context**

**In Spark, communication occurs between a driver and executors. The driver has Spark jobs that it needs to run and these jobs are split into tasks that are submitted to the executors for completion. Executor programs run on cluster nodes or in local threads. The results from these tasks are delivered back to the driver. Where does code run?**

* **Locally, in the driver**
* **Distributed at the executors (Executors run in parallel and have much more memory)**
* **Both at the driver and the executors**

**Problems with cheap hardware are: failures, network speeds versus shared memory, much more latency, network slower than storage, uneven performance.  
How do we deal with machine failures? We launch another task.  
How do we deal with slow tasks? We launch another task.**

**When running Spark, you start a new Spark application by creating a**[**SparkContext**](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext)**.  
SparkContext tells Spark how and where to access a cluster.  
The program next creates a [SQLContext](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html" \l "pyspark.sql.SQLContext) object which is used to create and manage the DataFrames.  
When the SparkContext is created, it asks the master for some cores to use to do work. The master sets these cores aside just for you; they won't be used for other applications.  
When using Databricks, both a SparkContext and a SQLContext are created for you automatically. sc is your SparkContext, and sqlContext is your SQLContext**

**# Display the type of the Spark sqlContext**

**type(sqlContext)**

**Out[35]: pyspark.sql.context.HiveContext**

**Note that the type is HiveContext. This means we're working with a version of Spark that has**[**Hive**](https://hive.apache.org/)**support. Compiling Spark with Hive support is a good idea, even if you don't have a Hive metastore. As the**[**Spark Programming Guide**](http://spark.apache.org/docs/latest/sql-programming-guide.html#starting-point-sqlcontext)**states, a HiveContext "provides a superset of the functionality provided by the basic SQLContext. Additional features include the ability to write queries using the more complete HiveQL parser, access to Hive UDFs [user-defined functions], and the ability to read data from Hive tables."**

**Spark Program Lifecycle**

1. **Create DataFrames from external data or create DataFrame from a collection in driver program**
2. **Lazily transform them into new DataFrames**
3. **cache() some DataFrames for reuse (optional)**
4. **Perform actions to execute parallel computation and produce results**

**Most of Python code runs in driver, except for code passed to transformations. Transformations run at executors. Actions can run both at executors and driver**

## **Part 1: Creating a base DataFrame and performing operations**

DataFrames (DF) are the key concept, the primary abstraction in Spark. Similar to Python Pandas dataframe, they are immutable once constructed and enable operations on collection of elements in parallel.  
You construct DataFrames by parallelizing existing Python collections (lists), by transforming an existing Spark or pandas DFs or from files in HDFS or any other storage system.  
DataFrames support two types of operations: transformations and actions.  
Transformations are lazy (not computed immediately). A transformed DF is executed when action runs on it.  
A transformation to a DataFrame is for example select. Actions to a DataFrame are for example show and count.

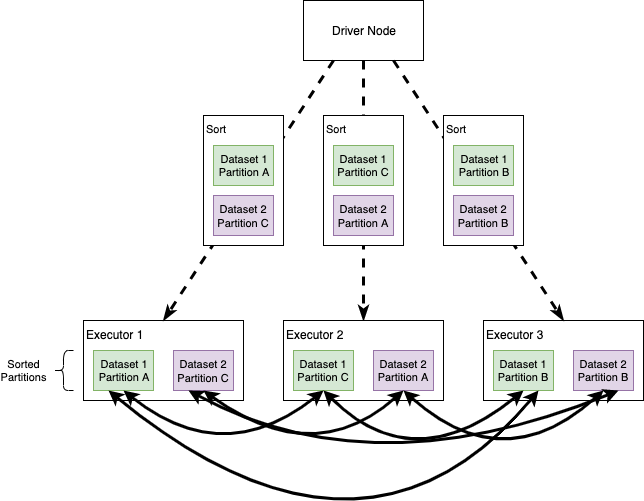
Joins:

<https://towardsdatascience.com/the-art-of-joining-in-spark-dcbd33d693c>

Spark will perform (or be forced by us to perform) joins in two different ways: either using **Sort Merge Joins** if we are joining two big tables, or **Broadcast Joins** if at least one of the datasets involved is small enough to be stored in the memory of the single all executors

**Sort Merge Join:**

When Spark translates an operation in the execution plan as a **Sort Merge Join** it enables an all-to-all communication strategy among the nodes: the Driver Node will orchestrate the Executors, each of which will hold a particular set of joining keys. Before running the actual operation, the partitions are first sorted (this operation is obviously heavy itself). As you can imagine this kind of strategy can be expensive: nodes need to use the network to share data; note that **Sort Merge Joins tend to minimize data movements in the cluster,** especially compared to Shuffle Hash Joins.



**Broadcast Joins**

Broadcast joins happen when Spark decides to send a copy of a table to all the executor nodes. The intuition here is that, if we broadcast one of the datasets, Spark no longer needs an all-to-all communication strategy and each Executor will be self-sufficient in joining the big dataset records in each node, with the small (broadcasted) table. We’ll see that this simple idea improves performance… usually.

