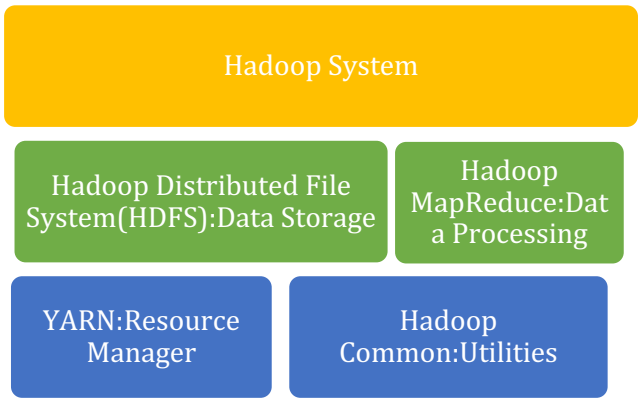
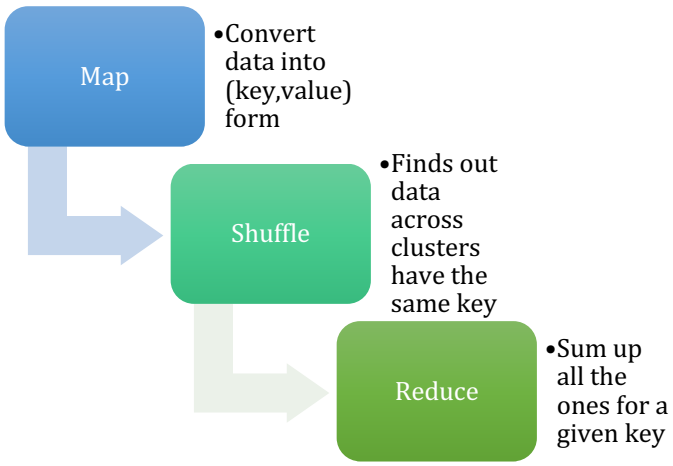
Spark Concepts1

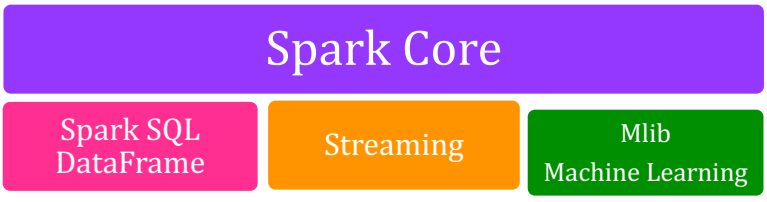
1. Distributed Computing & Parallel Computing:
   - Distributed Computing: each CPU has its own memory, each computer/machine is connected to the other machines across a network.
   - Parallel Computing: multiple CPUs share the same memory
2. Hadoop System:
   - Framework:



   - Streaming: Storm, Flink, Spark Streaming
3. MapReduce: programming technique for manipulating large data sets



4. Spark DAGs (Directed Acyclical Graph): lazy evaluation
   Stage1: dataset A->MAP->ReducebyKey---------->
   Stage2: dataset B->MAP->ReducebyKey->Filter->Join
5. Lambda function
6. PySpark Documentation



7. SparkContext: read and write data into Spark data frame
   - A spark program, the main entry point for spark functionality and connects the clusters with the application
   - From pyspark import SparkContext, SparkConf #specify the information about the application
     Configure=SparkConf().setAppName('name').setMaster('IP Address') #if we run spark in the local mode, we put 'local' in the bracket
     Sc=SparkContext(conf=configure)

- Read dataframes:

  From pyspark sql import SparkSession
  Spark=SparkSession \ #specify some parameters
  .builder \
  .appname ('app name') \
  .config ('config option', 'config value') \
  .getOrCreate() #create a new one or fix the old one

8. Data Wrangling with DataFrames
   - General Functions
     - Select (): returns a new DataFrame with the selected columns
     - Filter (): filters rows using the given condition
     - Where (): is just an alias for filter()
     - groupBy (): groups the DataFrame using the specified columns, so we can run aggregation on them
     - sort(): returns a new DataFrame sorted by the specified column(s). By default, the second parameter 'ascending' is True.
     - dropDuplicates(): returns a new DataFrame with unique rows based on all or just a subset of columns
     - withColumn(): returns a new DataFrame by adding a column or replacing the existing column that has the same name. The first parameter is the name of the new column, the second is an expression of how to compute it.
   - Aggregate Functions
     - count(), countDistinct(), avg(), max(), min(), etc. in spark.sql.functions
     - agg ({"salary": "avg", "age": "max"})← use different functions on different column
   - User Defined Functions (udf)
     - In spark sql we can use spark.sql.functions module to define our own functions
     - The default type of returns is string.
     - If we want to return different type, we use spark.sql.types module using different types.
   - Window Functions:
     - combine the values of ranges of rows in a DataFrame
     - choose how to sort and group (with the *partitionBy* method) the rows
     - how wide of a window we'd like to use (described by *rangeBetween* or *rowsBetween*)
9. Spark SQL
   - Build in Functions: https://spark.apache.org/docs/latest/api/sql/index.html
   - Guide: https://spark.apache.org/docs/latest/sql-getting-started.html
10. RDDs
    -
    - Transformations:

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.

sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
cogroup(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

- Actions:

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

count**()**	Return the number of elements in the dataset.
first**()**	Return the first element of the dataset (similar to take(1)).
take**(*n*)**	Return an array with the first *n* elements of the dataset.
takeSample**(*withReplacement*, *num*, [*seed*])**	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered**(*n*, *[ordering]*)**	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile**(*path*)**	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
~~saveAsSequenceFile**(*path*)** **(Java and Scala)**~~	~~Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).~~
~~saveAsObjectFile**(*path*)** **(Java and Scala)**~~	~~Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().~~
countByKey**()**	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach**(*func*)**	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.