1. **Pass Functions to PySpark**

Spark API require you to pass functions to driver program so that it will be executed on the distributed cluster. There are three ways to pass functions to Spark.

**--Lambda expressions**

Lambda Expressions in pyspark

Lambda Expressions in pyspark are simple functions that can be written as an expression.

For example, let us say you are trying to replace all the None values in each row in rdd\_source with empty strings, in this case you can use a list comprehension something like below.

rdd\_output = rdd\_source.map(lambda row: [r if r is not None else "" for r in row])

1. Local defs inside the function calling into Spark

For longer code, you can create local functions and call it with Spark RDD or any actions.

if \_\_name\_\_ == "\_\_main\_\_":

def myFunc(s):

words = s.split(" ")

return len(words)

sc = SparkContext(...)

sc.textFile("file.txt").map(myFunc)

**--Working with Key-Value Pairs:**

While most Spark operations work on RDDs containing any type of objects, a few special operations are only available on RDDs of key-value pairs. The most common ones are distributed “shuffle” operations, such as grouping or aggregating the elements by a key.

In Python, these operations work on RDDs containing built-in Python tuples such as (1, 2). Simply create such tuples and then call your desired operation.

For example, the following code uses the reduceByKey operation on key-value pairs to count how many times each line of text occurs in a file:

lines = sc.textFile("data.txt")

pairs = lines.map(**lambda** s: (s, 1))

counts = pairs.reduceByKey(**lambda** a, b: a + b)

We could also use counts.sortByKey(), for example, to sort the pairs alphabetically, and finally counts.collect() to bring them back to the driver program as a list of objects.

**--Creating Paired RDDs:**

Paired RDDs can be created by running a map() function that returns key/value pairs. The procedure to build key/value RDDs differs by language. In python, for making the functions on the keyed data to work, we need to return an RDD composed of tuples.

Creating a paired RDD using the first word as the key in Python:

pairs = lines.map(lambda x: (x.split(" ")[0], x))

1. **RDD Parallelize**

When we use parallelize() or textFile() or wholeTextFiles() methods of SparkContxt to initiate RDD, it automatically splits the data into partitions based on resource availability. When you run it on a laptop it would create partitions as the same number of cores available on your system.

getNumPartitions() – This a RDD function which returns a number of partitions our dataset split into.

print("initial partition count:"+str(rdd.getNumPartitions()))

Set parallelize manually – We can also set a number of partitions manually, all, we need is, to pass a number of partitions as the second parameter to these functions for example

sparkContext.parallelize([1,2,3,4,56,7,8,9,12,3], 10).

1. **Repartition and Coalesce**

Sometimes we may need to repartition the RDD, PySpark provides two ways to repartition; first using repartition() method which shuffles data from all nodes also called full shuffle and second coalesce() method which shuffle data from minimum nodes, for examples if you have data in 4 partitions and doing coalesce(2) moves data from just 2 nodes.

Both of the functions take the number of partitions to repartition rdd as shown below. Note that repartition() method is a very expensive operation as it shuffles data from all nodes in a cluster.

reparRdd = rdd.repartition(4)

print("re-partition count:"+str(reparRdd.getNumPartitions()))

#Outputs: "re-partition count:4

Note: repartition() or coalesce() methods also returns a new RDD.

1. **PySpark RDD Operations**

**RDD transformations – Transformations are lazy operations, instead of updating an RDD, these operations return another RDD.**

**RDD actions – operations that trigger computation and return RDD values.**

**RDD Transformations with example:**

Transformations on PySpark 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.

I will explain transformations using the word count example:

First, create an RDD by reading a text file.

rdd = spark.sparkContext.textFile("/tmp/test.txt")

**flatMap** – flatMap() transformation flattens the RDD after applying the function and returns a new RDD. On the below example, first, it splits each record by space in an RDD and finally flattens it. Resulting RDD consists of a single word on each record.

rdd2 = rdd.flatMap(lambda x: x.split(" "))

**map** – map() transformation is used to apply any complex operations like adding a column, updating a column e.t.c, the output of map transformations would always have the same number of records as input.

In our word count example, we are adding a new column with value 1 for each word, the result of the RDD is PairRDDFunctions which contains key-value pairs, word of type String as Key and 1 of type Int as value.

rdd3 = rdd2.map(lambda x: (x,1))

**reduceByKey** – reduceByKey() merges the values for each key with the function specified. In our example, it reduces the word string by applying the sum function on value. The result of our RDD contains unique words and their count.

rdd5 = rdd4.reduceByKey(lambda a,b: a+b)

**sortByKey** – sortByKey() transformation is used to sort RDD elements on key. In our example, first, we convert RDD[(String,Int]) to RDD[(Int, String]) using map transformation and apply sortByKey which ideally does sort on an integer value. And finally, foreach with println statements returns all words in RDD and their count as key-value pair

rdd6 = rdd5.map(lambda x: (x[1],x[0])).sortByKey()

#Print rdd6 result to console

print(rdd6.collect())

**filter** – filter() transformation is used to filter the records in an RDD. In our example we are filtering all words starts with “a”.

rdd4 = rdd3.filter(lambda x : 'an' in x[1])

print(rdd4.collect())

**RDD Actions with example:**

RDD Action operations return the values from an RDD to a driver program. In other words, any RDD function that returns non-RDD is considered as an action.

**count()** – Returns the number of records in an RDD

# Action - count

print("Count : "+str(rdd6.count()))

**first()** – Returns the first record.

# Action - first

firstRec = rdd6.first()

**take()** – Returns the record specified as an argument.

# Action - take

data3 = rdd6.take(3)

**saveAsTextFile()** – Using saveAsTestFile action, we can write the RDD to a text file.

rdd6.saveAsTextFile("/tmp/wordCount")

1. **Shuffle Operations:**

Shuffling is a mechanism PySpark uses to redistribute the data across different executors and even across machines. PySpark shuffling triggers when we perform certain transformation operations like gropByKey(), reduceByKey(), join() on RDDS

PySpark Shuffle is an expensive operation since it involves the following:

**Disk I/O**

Involves data serialization and deserialization

**Network I/O**

When creating an RDD, PySpark doesn’t necessarily store the data for all keys in a partition since at the time of creation there is no way we can set the key for data set.

Hence, when we run the reduceByKey() operation to aggregate the data on keys, PySpark does the following. Needs to first run tasks to collect all the data from all partitions. For example, when we perform reduceByKey() operation, PySpark does the following:

-PySpark first runs map tasks on all partitions which groups all values for a single key.

-The results of the map tasks are kept in memory.

-When results do not fit in memory, PySpark stores the data into a disk.

-PySpark shuffles the mapped data across partitions, some times it also stores the shuffled data into a disk for reuse when it needs to recalculate.

-Run the garbage collection

-Finally runs reduce tasks on each partition based on key.

-PySpark RDD triggers shuffle and repartition for several operations like repartition() and coalesce(), groupByKey(), reduceByKey(), cogroup() and join() but not countByKey() .

**Shuffle partition size & Performance**

Based on your dataset size, a number of cores and memory PySpark shuffling can benefit or harm your jobs. When you dealing with less amount of data, you should typically reduce the shuffle partitions otherwise you will end up with many partitioned files with less number of records in each partition. which results in running many tasks with lesser data to process.

On other hand, when you have too much of data and having less number of partitions results in fewer longer running tasks and some times you may also get out of memory error.

Getting the right size of the shuffle partition is always tricky and takes many runs with different values to achieve the optimized number. This is one of the key properties to look for when you have performance issues on PySpark jobs.

**PySpark RDD Persistence**:

PySpark Cache and Persist are optimization techniques to improve the performance of the RDD jobs that are iterative and interactive.

Though PySpark provides computation 100 x times faster than traditional Map Reduce jobs, If you have not designed the jobs to reuse the repeating computations you will see degrade in performance when you are dealing with billions or trillions of data. Hence, we need to look at the computations and use optimization techniques as one of the ways to improve performance.

**Using cache() and persist() methods, P**ySpark provides an optimization mechanism to store the intermediate computation of an RDD so they can be reused in subsequent actions.

When you persist or cache an RDD, each worker node stores it’s partitioned data in memory or disk and reuses them in other actions on that RDD. And Spark’s persisted data on nodes are fault-tolerant meaning if any partition is lost, it will automatically be recomputed using the original transformations that created it.

**Advantages of Persisting RDD**

**Cost efficient** – PySpark computations are very expensive hence reusing the computations are used to save cost.

**Time efficient** – Reusing the repeated computations saves lots of time.

**Execution time –** Saves execution time of the job which allows us to perform more jobs on the same cluster.

**RDD Cache**

PySpark RDD cache() method by default saves RDD computation to storage level `MEMORY\_ONLY` meaning it will store the data in the JVM heap as unserialized objects.

PySpark cache() method in RDD class internally calls persist() method which in turn uses sparkSession.sharedState.cacheManager.cacheQuery to cache the result set of RDD. Let’s look at an example.

cachedRdd = rdd.cache()

**RDD Persist**

PySpark persist() method is used to store the RDD to one of the storage levels MEMORY\_ONLY,MEMORY\_AND\_DISK, MEMORY\_ONLY\_SER, MEMORY\_AND\_DISK\_SER, DISK\_ONLY, MEMORY\_ONLY\_2,MEMORY\_AND\_DISK\_2 and more.

PySpark persist has two signature first signature doesn’t take any argument which by default saves it to MEMORY\_ONLY storage level and the second signature which takes StorageLevel as an argument to store it to different storage levels.

import pyspark

dfPersist = rdd.persist(pyspark.StorageLevel.MEMORY\_ONLY)

dfPersist.show(false)

RDD Unpersist

PySpark automatically monitors every persist() and cache() calls you make and it checks usage on each node and drops persisted data if not used or by using least-recently-used (LRU) algorithm. You can also manually remove using unpersist() method. unpersist() marks the RDD as non-persistent, and remove all blocks for it from memory and disk.

rddPersist2 = rddPersist.unpersist()

unpersist(Boolean) with boolean as argument blocks until all blocks are deleted.

**Persistence Storage Levels**

All different storage level PySpark supports are available at org.apache.spark.storage.StorageLevel class. Storage Level defines how and where to store the RDD.

MEMORY\_ONLY – This is the default behavior of the RDD cache() method and stores the RDD as deserialized objects to JVM memory. When there is no enough memory available it will not save to RDD of some partitions and these will be re-computed as and when required. This takes more storage but runs faster as it takes few CPU cycles to read from memory.

MEMORY\_ONLY\_SER – This is the same as MEMORY\_ONLY but the difference being it stores RDD as serialized objects to JVM memory. It takes lesser memory (space-efficient) then MEMORY\_ONLY as it saves objects as serialized and takes an additional few more CPU cycles in order to deserialize.

MEMORY\_ONLY\_2 – Same as MEMORY\_ONLY storage level but replicate each partition to two cluster nodes.

MEMORY\_ONLY\_SER\_2 – Same as MEMORY\_ONLY\_SER storage level but replicate each partition to two cluster nodes.

MEMORY\_AND\_DISK – In this Storage Level, The RDD will be stored in JVM memory as a deserialized objects. When required storage is greater than available memory, it stores some of the excess partitions in to disk and reads the data from disk when it required. It is slower as there is I/O involved.

MEMORY\_AND\_DISK\_SER – This is same as MEMORY\_AND\_DISK storage level difference being it serializes the RDD objects in memory and on disk when space not available.

MEMORY\_AND\_DISK\_2 – Same as MEMORY\_AND\_DISK storage level but replicate each partition to two cluster nodes.

MEMORY\_AND\_DISK\_SER\_2 – Same as MEMORY\_AND\_DISK\_SER storage level but replicate each partition to two cluster nodes.

DISK\_ONLY – In this storage level, RDD is stored only on disk and the CPU computation time is high as I/O involved.

DISK\_ONLY\_2 – Same as DISK\_ONLY storage level but replicate each partition to two cluster nodes.

1. **PySpark Shared Variables**

What are the different types of PySpark Shared variables and how they are used in PySpark transformations?

When PySpark executes transformation using map() or reduce() operations, It executes the transformations on a remote node by using the variables that are shipped with the tasks and these variables are not sent back to PySpark Driver hence there is no capability to reuse and sharing the variables across tasks. PySpark shared variables solve this problem using the below two techniques. PySpark provides two types of shared variables.

1. **Broadcast variables (read-only shared variable)**
2. **Accumulator variables (updatable shared variables)**

Broadcast read-only Variables

Broadcast variables are read-only shared variables that are cached and available on all nodes in a cluster in-order to access or use by the tasks. Instead of sending this data along with every task, PySpark distributes broadcast variables to the machine using efficient broadcast algorithms to reduce communication costs. One of the best use-case of PySpark RDD Broadcast is to use with lookup data for example zip code, state, country lookups e.t.c

When you run a PySpark RDD job that has the Broadcast variables defined and used, PySpark does the following.

PySpark breaks the job into stages that have distributed shuffling and actions are executed with in the stage.

Later Stages are also broken into tasks

PySpark broadcasts the common data (reusable) needed by tasks within each stage.

The broadcasted data is cache in serialized format and deserialized before executing each task.

The PySpark Broadcast is created using the broadcast(v) method of the SparkContext class. This method takes the argument v that you want to broadcast.

broadcastVar = sc.broadcast([0, 1, 2, 3])

broadcastVar.value

Note that broadcast variables are not sent to executors with sc.broadcast(variable) call instead, they will be sent to executors when they are first used.

**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 following accumulators

**-named accumulators**

**-unnamed accumulators**

When you create a named accumulator, you can see them on PySpark web UI under the “Accumulator” tab. On this tab, you will see two tables; the first table “accumulable” – consists of all named accumulator variables and their values. And on the second table “Tasks” – value for each accumulator modified by a task.

Whereas unnamed accumulators are not shows on PySpark web UI, For all practical purposes it is suggestable to use named accumulators.

Accumulator variables are created using SparkContext.longAccumulator(v)

accum = sc.longAccumulator("SumAccumulator")

sc.parallelize([1, 2, 3]).foreach(lambda x: accum.add(x))

PySpark by default provides accumulator methods for long, double and collection types. All these methods are present in SparkContext class and return LongAccumulator, DoubleAccumulator, and CollectionAccumulator respectively.