SPARK

Apache Spark is an open-source unified analytics engine for large-scale data processing. Spark provides an interface for programming clusters with implicit data parallelism and fault tolerance. 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.

PySpark is an interface for Apache Spark in Python. It not only allows you to write Spark applications using Python APIs, but also provides the PySpark shell for interactively analyzing your data in a distributed environment. PySpark supports most of Spark’s features such as Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) and Spark Core.

RDD:

A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel.

# SPARK CORE

map: Return a new RDD by applying a function to each element of this RDD.

rdd = sc.parallelize(["b", "a", "c"])

**>>>** sorted(rdd.map(**lambda** x: (x, 1)).collect())

[('a', 1), ('b', 1), ('c', 1)]

mapPartitions : Return a new RDD by applying a function to each partition of this RDD.

rdd = sc.parallelize([1, 2, 3, 4], 2)

**>>> def** f(iterator): **yield** sum(iterator)

**>>>** rdd.mapPartitions(f).collect()

[3, 7]

rdd.max : Find the maximum item in this RDD.

rdd = sc.parallelize([1.0, 5.0, 43.0, 10.0])

**>>>** rdd.max()

43.0

**>>>** rdd.max(key=str)

5.0

rdd.min: Find the minimum item in this RDD.

rdd = sc.parallelize([2.0, 5.0, 43.0, 10.0])

**>>>** rdd.min()

2.0

**>>>** rdd.min(key=str)

10.0

filter: Return a new RDD containing only the elements that satisfy a predicate.

rdd = sc.parallelize([1, 2, 3, 4, 5])

**>>>** rdd.filter(**lambda** x: x % 2 == 0).collect()

[2, 4]

first: Return the first element in this RDD.

sc.parallelize([2, 3, 4]).first()

2

**>>>** sc.parallelize([]).first()

Traceback (most recent call last):

...

ValueError: RDD is empty

flatMap: Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

rdd = sc.parallelize([2, 3, 4])

**>>>** sorted(rdd.flatMap(**lambda** x: range(1, x)).collect())

[1, 1, 1, 2, 2, 3]

**>>>** sorted(rdd.flatMap(**lambda** x: [(x, x), (x, x)]).collect())

[(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]

foreach: Applies a function to all elements of this RDD.

**def** f(x): print(x)

**>>>** sc.parallelize([1, 2, 3, 4, 5]).foreach(f)

getNumPartitions: Returns the number of partitions in RDD

rdd = sc.parallelize([1, 2, 3, 4], 2)

**>>>** rdd.getNumPartitions()

2

groupBy : Return an RDD of grouped items.

rdd = sc.parallelize([1, 1, 2, 3, 5, 8])

**>>>** result = rdd.groupBy(**lambda** x: x % 2).collect()

**>>>** sorted([(x, sorted(y)) **for** (x, y) **in** result])

[(0, [2, 8]), (1, [1, 1, 3, 5])]

gropByKey: Group the values for each key in the RDD into a single sequence. Hash-partitions the resulting RDD with numPartitions partitions.

If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will provide much better performance.

rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

**>>>** sorted(rdd.groupByKey().mapValues(len).collect())

[('a', 2), ('b', 1)]

**>>>** sorted(rdd.groupByKey().mapValues(list).collect())

[('a', [1, 1]), ('b', [1])]

intersection: Return the intersection of this RDD and another one. The output will not contain any duplicate elements, even if the input RDDs did.

rdd1 = sc.parallelize([1, 10, 2, 3, 4, 5])

**>>>** rdd2 = sc.parallelize([1, 6, 2, 3, 7, 8])

**>>>** rdd1.intersection(rdd2).collect()

[1, 2, 3]

isEmpty: Returns true if and only if the RDD contains no elements at all.

sc.parallelize([]).isEmpty()

True

**>>>** sc.parallelize([1]).isEmpty()

False

join: Return an RDD containing all pairs of elements with matching keys in self and other.

Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in self and (k, v2) is in other.

Performs a hash join across the cluster.

x = sc.parallelize([("a", 1), ("b", 4)])

**>>>** y = sc.parallelize([("a", 2), ("a", 3)])

**>>>** sorted(x.join(y).collect())

[('a', (1, 2)), ('a', (1, 3))]

distinct : Return a new RDD containing the distinct elements in this RDD.

sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())

[1, 2, 3]

sortByKey: Sorts this RDD, which is assumed to consist of (key, value) pairs.

tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]

**>>>** sc.parallelize(tmp).sortByKey().first()

('1', 3)

**>>>** sc.parallelize(tmp).sortByKey(**True**, 1).collect()

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

**>>>** sc.parallelize(tmp).sortByKey(**True**, 2).collect()

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

**>>>** tmp2 = [('Mary', 1), ('had', 2), ('a', 3), ('little', 4), ('lamb', 5)]

**>>>** tmp2.extend([('whose', 6), ('fleece', 7), ('was', 8), ('white', 9)])

**>>>** sc.parallelize(tmp2).sortByKey(**True**, 3, keyfunc=**lambda** k: k.lower()).collect()

[('a', 3), ('fleece', 7), ('had', 2), ('lamb', 5),...('white', 9), ('whose', 6)]

reduceByKey:

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. Default partitioner is hash-partition.

**from** **operator** **import** add

**>>>** rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

**>>>** sorted(rdd.reduceByKey(add).collect())

[('a', 2), ('b', 1)]

repartition:

Return a new RDD that has exactly numPartitions partitions.

Can increase or decrease the level of parallelism in this RDD. 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.

rdd = sc.parallelize([1,2,3,4,5,6,7], 4)

**>>>** sorted(rdd.glom().collect())

[[1], [2, 3], [4, 5], [6, 7]]

**>>>** len(rdd.repartition(2).glom().collect())

2

**>>>** len(rdd.repartition(10).glom().collect())

10

coalesce:

Return a new RDD that is reduced into *numPartitions* partitions.

sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()

[[1], [2, 3], [4, 5]]

**>>>** sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom().collect()

[[1, 2, 3, 4, 5]]

cache:

Persist this RDD with the default storage level (*MEMORY\_ONLY*).

persist:

ll different storage level PySpark supports are available at org.apache.spark.storage.StorageLevel class. The storage level specifies how and where to persist or cache a PySpark DataFrame.

MEMORY\_ONLY – This is the default behavior of the RDD cache() method and stores the RDD or DataFrame as deserialized objects to JVM memory. When there is not enough memory available it will not save DataFrame of some partitions and these will be re-computed as and when required. This takes more memory. but unlike RDD, this would be slower than MEMORY\_AND\_DISK level as it recomputes the unsaved partitions, and recomputing the in-memory columnar representation of the underlying table is expensive

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) than 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 – This is the default behavior of the DataFrame. In this Storage Level, The DataFrame will be stored in JVM memory as a deserialized object. When required storage is greater than available memory, it stores some of the excess partitions into a disk and reads the data from the disk when required. It is slower as there is I/O involved.

MEMORY\_AND\_DISK\_SER – This is the same as MEMORY\_AND\_DISK storage level difference being it serializes the DataFrame objects in memory and on disk when space is 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, DataFrame is stored only on disk and the CPU computation time is high as I/O is involved.

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

dfPersist = df.persist(StorageLevel.MEMORY\_ONLY)

# unpersist the DataFrame

dfPersist = dfPersist.unpersist()