

RDD Operations



cerenode.io

Types of RDD

- PairRDD
- DoubleRDD
- SequenceFileRDD
- CoGroupedRDD
- ShuffleRDD
- UnionRDD
- HadoopRDD



Aggregation on PairRDDs

- `groupByKey`
- `reduceByKey`
- `aggregateByKey`
- `combineByKey`



groupByKey

- **groupByKey** groups the values for each key in the RDD into a single sequence.
- **groupByKey** also allows controlling the partitioning of the resulting key-value pair RDD by passing a partitioner.
- By default, a **HashPartitioner** is used but a custom partitioner can be given as an argument.
- In a **groupByKey** call, all key-value pairs will be shuffled across the network to a reducer where the values are collected together
- **groupByKey** is an expensive operation due to all the data shuffling needed

reduceByKey

- **reduceByKey** tends to improve the performance by not sending all elements of the **PairRDD** using shuffles, rather using a local combiner to first do some basic aggregations locally and then send the resultant elements as in **groupByKey**.
- This greatly reduces the data transferred, as we don't need to send everything over



aggregateByKey

- `aggregateByKey` is quite similar to `reduceByKey`, except that `aggregateByKey` allows more flexibility and customization of how to aggregate within partitions and between partitions to allow much more sophisticated use cases
- `aggregateByKey` works by performing an aggregation within the partition operating on all elements of each partition and then applies another aggregation logic when combining the partitions themselves



aggregateByKey

The aggregateByKey function requires 3 parameters:

- An initial 'zero' value that will not effect the total values to be collected.
- A combining function accepting two paremeters. The second paramter is merged into the first parameter. This function combines/merges values within a partition.
- A merging function function accepting two parameters. In this case the paremters are merged into one. This step merges values across partitions.



combineByKey

- `combineByKey` is very similar to `aggregateByKey`
- In `aggregateByKey`, the first argument is simply a zero value but in `combineByKey`, we provide the initial function which takes the current value as a parameter.



Partitions

- A **partition** is a logical chunk of a large distributed data set.
- Spark manages data using partitions that helps parallelize distributed data processing with minimal network traffic for sending data between executors.
- You can request for the minimum number of partitions, using the second input parameter to many transformations.



Types of Partitioning

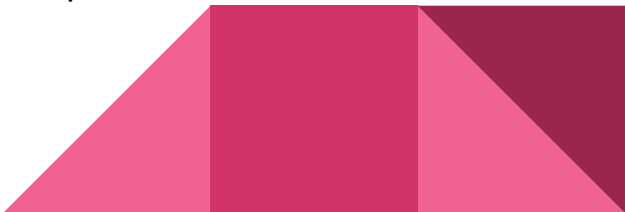
- **Hash Partitioning in Spark**

Hash Partitioning attempts to spread the data evenly across various partitions based on the key.

Object.hashCode method is used to determine the partition in Spark as $\text{partition} = \text{key.hashCode} () \% \text{numPartitions}$.

- **Range Partitioning in Spark**

Some Spark RDDs have keys that follow a particular ordering, for such RDDs, range partitioning is an efficient partitioning technique. In range partitioning method, tuples having keys within the same range will appear on the same machine. Keys in a range partitioner are partitioned based on the set of sorted range of keys and ordering of keys.

A decorative graphic in the bottom right corner consisting of several overlapping triangles in shades of pink and magenta.

coalesce

- The coalesce method reduces the number of partitions
- The coalesce algorithm changes the number of partitions by moving data from some partitions to existing partitions.
- Coalesce can create uneven partitions
- In case of pair RDD's coalesce does not group entries with similar keys on the same partition



repartition

- The repartition method can be used to either increase or decrease the number of partitions
- The repartition algorithm does a full shuffle of the data and creates equal sized partitions of data
- In case of pair RDD's repartition will try to put similar keys in single partitions so any subsequent join , groupByKey, reduceByKey like operations will be faster



Broadcast Variables

- Shared variable across all executors
- Created on driver
- Read-only on executors
- Supported operations
 - value
 - unpersist
 - destroy



Accumulators

- Accumulators are shared variables across executors typically used to add counters to your Spark program.
- Accumulators can only be updated by adding to the value.
- They can be used to implement counters (as in MapReduce) or sums.
- We can create custom Accumulators extending `AccumulatorV2`



RDD Persistence Mechanism

- Spark RDD persistence is an optimization technique which saves the result of RDD evaluation. Using this we save the intermediate result so that we can use it further if required. It reduces the computation overhead.
- We can make persisted RDD through **cache()** and **persist()** methods.
- Storage levels of Persisted RDDs
 - MEMORY_ONLY
 - MEMORY_AND_DISK
 - MEMORY_ONLY_SER
 - MEMORY_AND_DISK_SER
 - DISK_ONLY