|  |  |
| --- | --- |
| **groupByKey**([*numTasks*]) | 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 numTasks argument to set a different number of tasks |
| **reduceByKey**(*func*, [*numTasks*]) | 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*, [*numTasks*]) | 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.  Here sqqOp is combiner logic and combOp is reducer logic |

|  |
| --- |
| **groupByKey** - not intended for aggressive aggregations It does not use combiner only reducer is used for do the combining process)  val ordersByStatus = ordersMap.groupByKey().map(t => (t.\_1, t.\_2.sum)) |
| **reduceByKey** - reduceByKey uses combiner when both combiner logic and reducer logic are same and also input type and output type has to be same  val ordersByStatus = ordersMap.reduceByKey((acc, value) => acc + value)  like here input type is int and final sum is also int |
| **combineByKey** - combineByKey can be used when reduce logic and combine logic are different  #Both reduceByKey and combineByKey expects same type of input data and output data  val ordersByStatus = ordersMap.combineByKey(value => 1, (acc: Int, value: Int) => acc+value, (acc: Int, value: Int) => acc+value)  Here 2nd parameter is intermediate aggregation logic and 3rd parameter is final aggregation logic  Here we keep the same logic for both intermediate and final logic but in that case reduceByKey is best one.  https://www.edureka.co/blog/apache-spark-combinebykey-explained |
| **aggregateByKey** - #aggregateByKey can be used when reduce logic and combine logic is different  #Also type of input data and output data need not be same  val revenuePerDay = revenuePerDayPerOrderMap.aggregateByKey((0.0, 0))(  (acc, revenue) => (acc.\_1 + revenue, acc.\_2 + 1),  (total1, total2) => (total1.\_1 + total2.\_1, total1.\_2 + total2.\_2)  )  Here (0.0, 0) is initial value which is a tuple in this case  acc is accumulator and is of tuple here , and revenue is of float type  total1 is of tuple which is accumulator and total2 is also tuple which is output of combiner step  (acc.\_1 + revenue, acc.\_2 + 1)  <http://codingjunkie.net/spark-agr-by-key/> |