①

1.

[understand the Hash partitioning Algorithm of the dataset / keys]

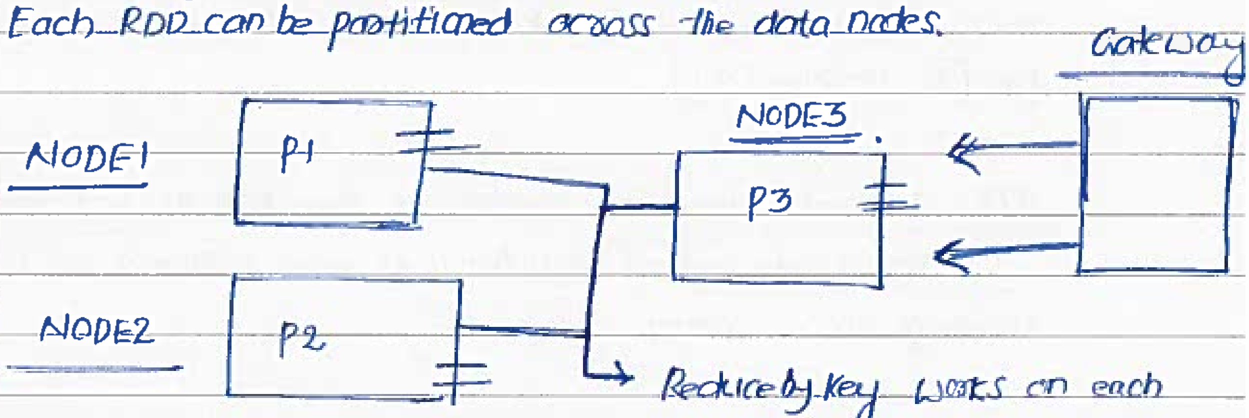
reduceByKey().

Reduce the dataset values based on the key.

1. ReduceByKey works only one dataset. (only one RDD)
2. Each RDD can be partitioned across the data nodes.

PARTITION-3

K6, (1,4)
K9, (1,7)
K6, (1,8)
K9, (1,7)



ReduceByKey works on each partition locally.

⇒ partition is just your data split across the data nodes.

K3, (1,4)
K4, (1,4)
K5 (1,6)
K3, (1,9)
K4, (2,7)

PARTITION-2

K1, (1,2)
K1 (1,7)
K2, (1,8)
K2, (2,9)
K1, (1,8)
K1, (2,6)

PARTITION-1

⇒ K1 is the key and associated value is tuple (1,2)

⇒ K1 ⇒ Key (1,2) ⇒ Tuple.

(1,2) ⇒ This entire value is tuple.

⇒ Same partition can contain multiple keys.

⇒ The dataset partitioned across the nodes in the hadoop cluster are represented in single RDD.

Y => ?

Syntax: reduceByKey((K1,(1,2)), (K1,(1,7)))

X => ?

1, 2, 3, 4, 5

+ + +

③

⇒ 3, 3, 4, 5

+ + +

⇓

6, 4, 5

+ +

10, 5

⇒ This is the 'iterative algoritham' implemented one by one

⇒ reduceByKey() needs atleast 2 inputs,

②

(x,y) represents the ~~same~~ different value from ~~different~~ same key.

↳ reduceByKey $(((K_1)\overset{x}{(1,2)}),((K_1)\overset{y}{(1,7)}))$

<u>Complete Syntax</u> :- ① $1+1 \Rightarrow 2$, $2+7 \Rightarrow 9$ $\Rightarrow (K_1,(2,9)) \Rightarrow [2,9]$

dataRDD.reduceByKey((x,y) => x._1 + y._1 , x._2 + y._2))

(*) <u>Explains</u>:- Take the first value of x and also take the first value of (y) and apply the Summation.

Similarly take the second value of (x) and second value of (y) and apply the Summation.

→ As it is iterative algoritham, it scans through all other keys in the dataset across the partition.

② $\Rightarrow ((K_1,(2,9)),(K_1,(2,6)))$

↳ $K_1 \Rightarrow (4,15)$

Locally on each partition.

Partition 1 ⇒ [(K1,(4,15)), (K2,(3,17)), (K7,(9,8))]

Partition 2 ⇒ [(K3,(3,19)), (K4,(3,11)), (K5, (1,6))]

Partition 3 ⇒ [(K6, (2,12)), K9, (2,14)]

groupByKey()

```
Val words = Array("one", "two", "two", hadoop, hadoop, hadoop)
Val generatepairRDD = sc.parallelize(words).map(words => (word,1))
```

→ Val words is my Array element, Parallelize distributes the data across the cluster.

→ Map function picks up each element in the list and generates the results based on the functions defined.

→ (one,1), (two,1), (two,1), (hadoop,1) (hadoop,1), (hadoop,1)

↳ For each element in the list, it will generate a tuple.

groupBy Key :- Whereever the same key is available, group it.

```
(one,1)
(two,1)        →    hadoop (1,1,1)
(two,1)             one (1)
(hadoop,1)          two (1,1)
(hadoop,1)
(hadoop,1)     ⇒
```

val wordCountWithReduce = generatepairRDD.reduceByKey(_+_).collect()

↳ perform the ~~groups~~ (+) operation on all the values in the group. (not across the group, in the same group)

↳ hadoop is one group, one is one group, two is one group

```
↳   hadoop (3)            (hadoop,3)
    one (1)        ⇒      (one, 1)
    two (2)               (two, 2)
```

10L. ④

GroupByKey Just groups the data and on top of it you apply functions

Second approach for groupByKey :-

```
Val WordWithGroup = generatePairRDD.groupByKey().
                    map(t => (t._1, t._2, Sum)).collect().
```

In here, (t) means tuple. The first 't' represents entire element

Explaination :- [hadoop,(1,1,1)]
[one,(1,1)]
[two,(1)]

t._1 is hadoop => t._2 is (1,1,1) => you are asking map function to sum it up => So => [hadoop,3], (one,2), (two,1)

④ Avoid groupByKey as much as possible.

1. reduceByKey() and groupByKey() gives same result.
2. reduceByKey() works better.
3. groupByKey() => first shuffles and then aggregates.
   ↳ This involves huge traffic, recurring performance issue.
4. Think twice and thrice when you think of using groupByKey() function.
   ↳ Refer to Databricks best practises documentation.

foldByKey() :-

foldByKey & fold()

fold() : Example.

Sc. paralelize (1 to 10) . fold(0) { (acc, element) => acc + element) }

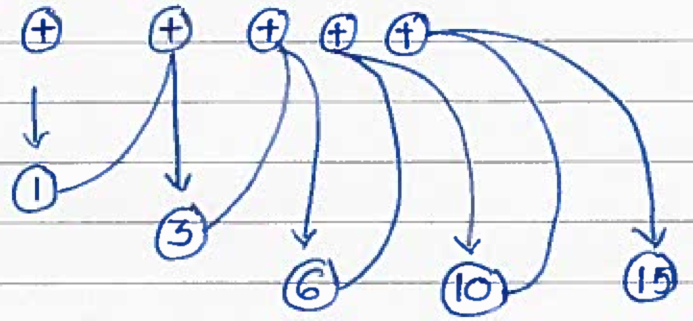
Example:- 1, 2, 3, 4, 5 => fold this ~~array~~ set with ⊕

For the fold operation to start the summation, it needs an initial value to start, which is initialized as zero.

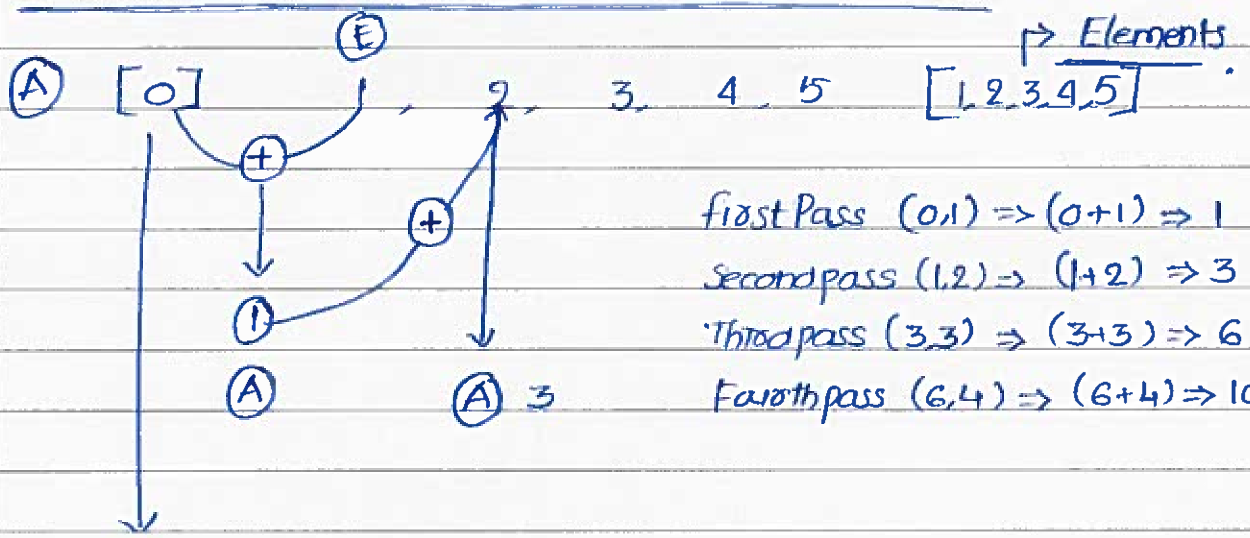
→ using zero, which wont impact the correct result.

↳ 1, 2, 3, 4, 5

↳ 0, 1, 2, 3, 4, 5



understanding Accumulator and element is important?



first Pass (0,1) => (0+1) => 1

Second pass (1,2) => (1+2) => 3

Third pass (3,3) => (3+3) => 6

Fourth pass (6,4) => (6+4) => 10

0 is the initial accumulator.

In tuples, first value will be key and Second is value

tuple (vs) Element.

Second approach of foldByKey.

↳ count elements in the list.

↳ sc.parallelize (1 to 10). fold(0) { (acc, element) => acc+1 }.

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

↳ This is element.

0+1 = 1
1+1 = 2
2+1 = 3
3+1 = 4

foldByKey()
↓
fold() function is applied on each key, individually.

K1,	[1 to 10]	→	55
K2,	[1 to 5]	→	15
K3,	[1 to 7]	→	28

'foldByKey' requires a dummy value, similar way as we have used 0 as accumulator

Perfect use case:- find the max score by department.

```
val maxdept = empRDD. foldByKey(("dummy", 0.0))
        ((acc, element) => if (acc._2 > element._2)
                              acc
                              else
                              element)
```

cs is the key and (Amit,1000) is the tuple,

```
val depEmployee = list (
                    (cs, (Amit, 1000)),
                    (cs, (Rahul, 1200)),
                    ( ECE, (Rakesh, 1500)),
                    ( ECE, (Ankit, 1200))
```

first pass => acc Element

CS → [$(\overset{-1}{\text{dummy}}, \overset{-2}{0.0})$, $(\overset{-1}{\text{Amit}}, \overset{-2}{1000})$]

↳ (Amit, 1000).

↳ This becomes accumulator

CS → [(Amit, 1000), (Rahul, 1200)]

↳ (Rahul, 1200).

acc Element

EC → [$(\overset{-1}{\text{dummy}}, \overset{-2}{0.0})$, $(\overset{-1}{\text{Rakesh}}, \overset{-2}{1500})$]

↳ (Rakesh, 1500)

EC → [(Rakesh, 1500), (Ankit, 1200)]

→ (Rakesh, 1500)

max Dept. [(CS, (Rahul, 1200)), (EC, (Rakesh, 1500))]

⑧

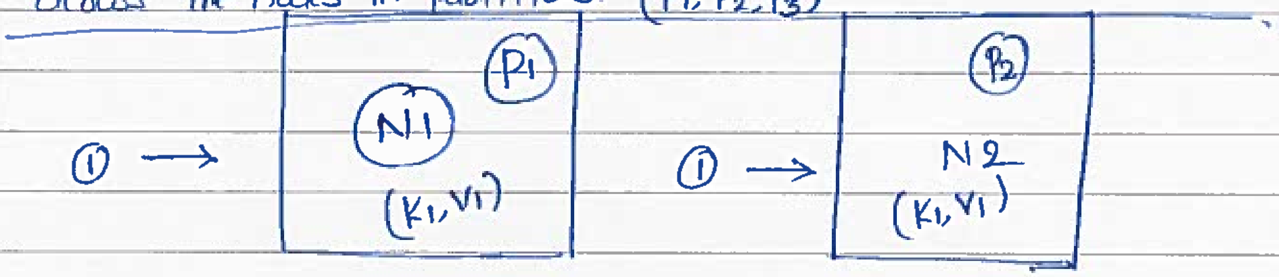
combine By Key( )

combine By key requires 4 parameters :-

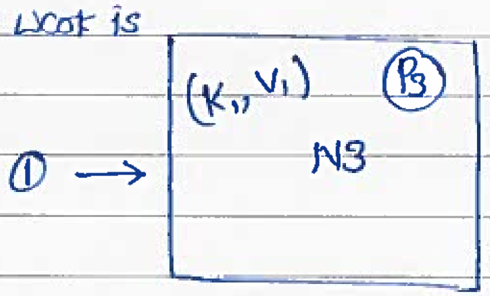
1. Create combiner.
2. Merge values.
3. Merge combiner.
4. partitioner.

Example: calculating the Average

→ calculate the Average for each key, we have an dataset RDD distributed across the nodes in partitions. (P1, P2, P3)



→ In the Spark cluster work is done partially



1. Create the combiner ⇒ (V) = (V,1), combiner will be created only when the key comes first time, in the same partition.

↳ Sequential iteration on each partition. partition wise its parellal only

(K2, 14)
(K3, 11)
(K3, 12)
(K3, 11)
(K1, 14)
(K2, 14)

(K1, 12)
(K2, 13)
(K1, 11)
(K1, 16)
(K1, 12)
(K2, 13)

P1

P1 ⇒ (K1, 12) → (12, 1)
(K2, 13) → (13, 1)
already appeared (K1, 11)

↳ here we apply the second rule, which is

(acc : (Int, Int), V) ⇒ acc.-1 + V , acc.-2 + 1)

↳ accumulator will be (12,1), 11 → value

(acc·-1+v, acc·-2+1)

⇒ acc·-1 is 12. and value is 11 ⇒ (12+11) ⇒ 23.
⇒ acc·-2+1 ⇒ acc·-2 is 1 ⇒ (1+1) ⇒ 2.
↳ (23,2)

3rd time ⇒ K1 → (23,2), 12
→ (23+12) = 35 ; 2+1 = 3
→ (35,3).

↳ This process continues for each partitions. ⇒ K1 → (12,1),14
⇒ → (26,2)

3 "Merge combiners":-

acc1 : (Int,Int), acc2 : (Int,Int)
⇒ (acc·-1+acc·-2+1 , acc·-2 + acc·-2·-2)
→ ACCM2;
(23,2) (26,2)
↳ ACCM1;
⇒ acc·-1+acc·-2-1 ⇒ 23+26 = 49 ⇒ (49,4)
⇒ acc·-2 + acc.2·2 ⇒ 2+2 = 4

⇒ [K1,(49,4)].