19 FEB 2024 – POLIONLINE

MAPREDUCE & HADOOP

```
/**
 * Mapper first job
 */
class MapperBigData1 extends Mapper<LongWritable, // Input key type
        Text, // Input value type
        Text, // Output key type
        NullWritable> { // Output value type

    protected void map(LongWritable key, // Input key type
            Text value, // Input value type
            Context context) throws IOException, InterruptedException {

        String[] fields = value.toString().split(",");
        String userId = fields[1];
        String productId = fields[2];
        String timestamp = fields[0].split("-")[0];

        // filter based on date of purchase
        if(timestamp.compareTo("2020/01/01") >= 0 &&
            timestamp.compareTo("2023/12/21") <= 0)
            context.write(new Text(userId + "_" + productId),
NullWritable.get());
    }
}


/**
 * Reducer first job
 */
class ReducerBigData1 extends Reducer<Text, // Input key type
        NullWritable, // Input value type
        Text, // Output key type
        IntWritable> { // Output value type

    @Override
    protected void reduce(Text key, // Input key type
            Iterable<NullWritable> values, // Input value type
            Context context) throws IOException, InterruptedException {

        // the reduce method guarantees that for each user we obtain the distinct
items
        // the user purchased
```

```
        String userProduct = key.toString();
        String userId = userProduct.split("_")[0];

        context.write(new Text(userId), new IntWritable(1));
    }

}


/**
 * Mapper second job
 */
class MapperBigData2 extends Mapper<
                    Text,  // Input key type
                    Text,         // Input value type
                    Text,         // Output key type
                    IntWritable> {        // Output value type

    protected void map(
            Text key,   // Input key type
            Text value,         // Input value type
            Context context) throws IOException, InterruptedException {

            int count = Integer.parseInt(value.toString());
            context.write(key, new IntWritable(count));
    }
}


/**
 *  Reducer second job
 */
class ReducerBigData2 extends Reducer<
                Text,           // Input key type
                IntWritable,    // Input value type
                Text,    // Output key type
                NullWritable> {  // Output value type

    @Override
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context) throws IOException, InterruptedException {

        int count = 0;
```

```
        for(IntWritable v : values) {
            count += v.get();
        }

        if(count >= 50)
            context.write(new Text(key), NullWritable.get());

    }
}
```

SPARK

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setAppName('Exam 12 Sept 2024')
sc = SparkContext(conf = conf)

purchasePath = "data/Purchases.txt"
# usersPath = "data/Users.txt"
cataloguePath = "data/Catalogue.txt"

outputPath1 = "outPart1/"
outputPath2 = "outPart2/"

# Define the rdds associated with Purchases and Catalogue
# SaleTimestamp,UserID,ItemID,SalePrice
purchaseRDD = sc.textFile(purchasePath)

# ItemID,Name,Category,StillInProduction
catalogueRDD = sc.textFile(cataloguePath)

########################################
# PART 1
########################################

purchases2223 = purchaseRDD.filter(lambda s: s.startswith('2022') or
s.startswith('2023')).cache()

def userCounts(line):
    fields = line.split(',')
    # userId = fields[1]
    if line.startswith('2022'):
        return (fields[1], (1, 0))
    else:
        return (fields[1], (0, 1))

# (userId, count)
userIdCounts = purchases2223.map(userCounts)\
    .reduceByKey(lambda p1, p2: (p1[0] + p2[0], p1[1] + p2[1]))

max2223 = userIdCounts.values().reduce(lambda a, b: (max(a[0], b[0]), max(a[1],
b[1])))

max22 = max2223[0]
max23 = max2223[1]
```

```
res1 = userIdCounts.filter(lambda p: (p[1][0] == max22 or p[1][1] == max23))\
    .keys()

res1.saveAsTextFile(outputPath1)

########################################
# PART 2 - v1
########################################

# considering the purchases in year 2022/2023 (purchases2223 RDD)
# we use a mapToPair with
# key = itemID
# value = userID
# and a distinct to obtain the distinct user-product purchases.
#
# Then, perform a map + reduceByKey to count for each itemID,
# the number of distinct users who bought that item
# key = itemID
# value = numberOfDistinctUsersPurchases


def ItemUser(line):
    fields = line.split(",")
    userId = fields[1]
    itemId = fields[2]

    return (itemId, userId)

itemDistinctUsersPurchases = purchases2223\
                             .map(ItemUser)\
                             .distinct()\
                             .map(lambda t: (t[0], 1))\
                             .reduceByKey(lambda v1, v2: v1 + v2)

# for each item, we retrieve the corresponding category
def ItemCategory(line):
    fields = line.split(",")
    itemId = fields[0]
    category = fields[2]

    return (itemId, category)

itemCategory = catalogueRDD.map(ItemCategory).cache()
```

```
# join itemCategory RDD with itemDistinctUsersPurchases
itemCategoryPurchases = itemCategory.join(itemDistinctUsersPurchases)\
                        .cache()

# compute for each category the maximum number of distinct users who purchased
the item
# first, we obtain the following RDD
# key = category
# value = number of distinct users who purchased an item
# and then we use a reduceByKey to compute the maximum value for each category
maxDistinctUsersPurchasesPerCategory = itemCategoryPurchases\
                                    .map(lambda t:  (t[1][0], t[1][1]))\
                                    .reduceByKey(lambda v1, v2: max(v1, v2))

# map itemCategoryPurchases to ( (category, numPurchases), itemid), join with
maxDistinctUsersPurchasesPerCategory
# (first map to ((category, maxPurchases), None))
# after join, format is
# key = (category,numPurchases)
# value = (itemId, None),
# then, use a map to obtain the format for the result
# key = category
# value = itemId

def CatItemId(t):
    category = t[0][0]
    itemId = t[1][0]

    return (category, itemId)

res2Partial = itemCategoryPurchases\
        .map(lambda t: ( (t[1][0], t[1][1]), t[0]))\
        .join(maxDistinctUsersPurchasesPerCategory.map(lambda tmax: (tmax,
None)))\
        .map(CatItemId)

# Alternative solution for this step
#
# map itemCategoryPurchases to (category, (itemid, numPurchases), join with
maxDistinctUsersPurchasesPerCategory
# and filter, keeping only the entries with numPurchases == maxPurchases
# after join, format is
# key = category
# value = (itemId, numPurchases), maxPurchasesPerCategory
# then, use a map to obtain the format for the result
```

```
# key = category
# value = itemId

#def CatItemId(t):
#    category = t[0]
#    itemId = t[1][0][0]
#
#    return (category, itemId)
#
#
#res2Partial = itemCategoryPurchases\
#        .map(lambda t: (t[1][0], (t[0], t[1][1])))\
#        .join(maxDistinctUsersPurchasesPerCategory)\
#        .filter(lambda t: t[1][0][1] == t[1][1])\
#        .map(CatItemId)


# from res2Partial we need to add the 0-case, i.e., categories with items which
were never purchased.
# Consider all distinct categories (itemCategory.values().distinct()) and
subtract those in res2Partial.
# At the end, map the selected categories to pairs
# key = category
# value = "NoPurchases"
unsoldCategories = itemCategory.values().distinct()\
        .subtract(res2Partial.keys())\
        .map(lambda cat: (cat, "NoPurchases"))

# update the result of the second part with a final Union
res2Final = res2Partial.union(unsoldCategories)

res2Final.saveAsTextFile(outputPath2)
```