Transformations based on one RDD

Transformation	Purpose	Example	Result
examples based on: {1,2,3,3}			
`JavaRDD<T> filter(Function<T,Boolean> f)`	Return an RDD consisting only of the elements of the "input"" RDD that pass the condition passed to filter().The "input" RDD and the new RDD have the same data type.	`x != 1`	{2,3,3}
`JavaRDD<R> map(Function<T, R>)`	Apply a function toeach element in the RDD and return an RDD of the result. The applied function return one element for each element of the "input" RDD.The "input" RDD and the new RDD can have a different data type.	`x -> x+1` (i.e., for each input element x, the element with value x+1 is included in the new RDD)	{2,3,4,4}
`JavaRDD<R> flatMap( Function<T,Iterator<R>> f )`	Apply a function to each element in the RDD and return an RDD of the result. The applied function return a set of elements (from 0 to many) for each element of the "input" RDD.The "input" RDD and the new RDD can have a different data type.	`x ->x.to(3)` (i.e., for each input element x, the set of elements with values from x to 3 are returned)	{1,2,3,2,3,3,3}
`JavaRDD<T> distinct()`	Remove duplicates		{1,2,3}
`JavaRDD<T> sample( boolean withReplacement, double fraction )`	Sample the content of the "input" RDD, with or without replacement and return the selected sample.The "input" RDD and the new RDD have the same data type.		Nondeterministic
examples based on {("k1", 2), ("k3", 4), ("k3", 6)}			
`JavaPairRDD<K,V> reduceByKey( Function2<V,V,V> f )`	Return a PairRDD<K,V> containing one pair for each key of the "input" PairRDD. The value of each pair of the new PairRDD is obtained by combining the values of the input PairRDD with the same key.The "input" PairRDD and the new PairRDD have the same data type.	`(x, y) -> x + y`	{("k1", 2), {"k3", 10}
`JavaPairRDD<K,V> foldByKey( V zeroValue, Function2<V,V,V> f )`	Similar to the reduceByKey() transformation. However, foldByKey() is characterized also by a zero value	`0, (x, y) -> x + y`	{("k1", 2), {"k3", 10}
`JavaPairRDD<K,V> combineByKey( Function<C,V> createCombiner, Function2<V,C,V> mergeValue, Function2<V,V,V> mergeCombiners )`	Return a PairRDD<K,U> containing one pair for each key of the "input" PairRDD. The value of each pair of the new PairRDD is obtained by combining the values of the input PairRDD with the same key.The "input" PairRDD and the new PairRDD can be different.	average value per key	{("k1", 2), ("k3", 5)}
`JavaPairRDD<K,Iterable<V>> groupByKey()`	Return a PairRDD<K,Iterable containing one pair for each key of the "input" PairRDD. The value of each pair of the new PairRDD is a "list" containing the values of the input PairRDD with the same key.		{("k1", [2]), ("k3", [4, 6])}
`JavaPairRDD<K,V> mapValues( Function<W,V> f )`	Apply a function over each pair of a PairRDD and return a new PairRDD. The applied function returns one pair for each pair of the "input" PairRDD. The function is applied only to the value without changing the key.The "input" PairRDD and the new PairRDD can have adifferent data type.	`x -> x+1`	{("k1", 3), ("k3", 5), ("k3", 7)}
`JavaPairRDD<K,V> flatMapValues( Function<W,Iterable<V>> f )`	Apply a function over each pair in the input PairRDD and return a new RDD of the result. The applied function returns a set of pairs (from 0 to many) for each pair of the "input" RDD. The function is applied only to the value without changing the key. The "input" RDD and the new RDD can have adifferent data type.	`x -> x.to(5)` (i.e., for each input pair (k,v),the set of pairs (k,u) with values of u from v to 5 are returned and included in the new PairRDD)	{("k1", 2), ("k1", 3), ("k1", 4),("k1", 5), ("k3", 4), ("k3", 5)}
`JavaRDD<K> keys()`	Return an RDD containing the keys of the input PairRDD.	`inputPairRDD.keys()`	{"k1", "k3", "k3"}
`JavaRDD<V> values()`	Return an RDD containing the values of the input PairRDD.	`inputPairRDD.values()`	{2, 4, 6}
`JavaPairRDD<K,V> sortByKey()`	Return a PairRDD<K,V> containing the pairs of the input PairRDD sorted by key.		{("k1", 2), ("k3", 4), ("k3", 6)}

Transformations based on two RDDs

Transformation	Purpose	Example	Result
examples based on: {1,2,2,3,3} and {3,4,5}			
`JavaRDD<T> union(JavaRDD<T> other)`	Return a new RDD containingthe union of the elements of the "input"" RDD and the elements of the one passedas parameter to union().Duplicate values are notremoved.All the RDDs have the same data type.	`inputRDD1.union(inputRDD2)`	{1, 2, 2, 3, 3, 3, 4, 5}
`JavaRDD<T> intersection(JavaRDD<T> other)`	Return a new RDD containingthe intersection of the elements of the "input"" RDD and the elements of the one passed as parameter to intersection().All the RDDs have the same data type.	`inputRDD1.intersection( inputRDD2 )`	{3}
`JavaRDD<T> subtract( JavaRDD<T> other )`	Return a new RDD the elements appearing only in the "input"" RDD and not in the one passed as parameter to subtract().All the RDDs have the same data type.	`inputRDD1.subtract( inputRDD2 )`	{1, 2, 2}
`JavaRDD<T> cartesian( JavaRDD<T> other )`	Return a new RDD containingthe cartesian product of the elements of the "input"" RDD and the elements of the one passed as parameter to cartesian().All the RDDs have the same data type.	`inputRDD1.cartesian( inputRDD2 )`	{(1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (2,3), (2,4), (2,5), (3,3), (3,4), (3,5), (3,3), (3,4), (3,5)}
examples based on {("k1", 2), ("k3", 4), ("k3", 6)} and {("k3", 9)}			
`JavaPairRDD<K,V> subtractByKey( JavaPairRDD<K,W> other )`	Return a new PairRDD where the pairs associated with a key appearing only in the "input"" PairRDD and not in the one passed as parameter.The values are not considered to take the decision.	`inputRDD1.subtract( inputRDD2 )`	{("K1",2)}
`JavaPairRDD<K,Tuple2<V,W>> join( JavaPairRDD<K,W> other )`	Return a new PairRDD corresponding to join of the two PairRDDs. The join is based in the value of the key.	`inputRDD1.join(inputRDD2)`	{("k3", (4, 9)), ("k3", (6, 9))}
`JavaPairRDD< K,Tuple2<Iterable<V>,Iterable<W>>> cogroup( JavaPairRDD<K,W> other )`	For each key k in one of the two PairRDDs, return a pair (k, tuple), where tuple containsthe list of values of the first PairRDD with key k in the first element of the tuple and the list of values of thesecond PairRDD with key k in the second element of the tuple.	`inputRDD1.cogroup( inputRDD2 )`	{("k1", ([2], [])), ("k3", ([4, 6], [9]))}

Actions

Action	Purpose	Example	Result
examples based on {1,2,3,3}			
`List<T> collect()`	Return a (Java) List containing all the elements of the RDD on which it is applied.The objects of the RDD and objects of the returned list are objects of the same class.	`inputRDD.collect()`	{1,2,3,3}
`long count()`	Return the number of elements in the RDD on which it is applied.	`inputRDD.count()`	4
`Map<T,Long> countByValue()`	Return a Map object containing the information about the number of times each element occurs in the RDD.	`inputRDD.countByValue()`	{(1,1), (2,1), (3,2)}
`List<T> take(int n)`	Return a (Java) List containing the first num elements of the RDD.The objects of the RDD and objects of the returned list are objects of the same class.	`inputRDD.take(2)`	{1,2}
`T first()`	Return the first element of the RDD.	`inputRDD.first()`	1
`List<T> top(int n)`	Return a (Java) List containing the top num elements of the RDD based on the default sort order/comparator of the objects.The objects of the RDD and objects of the returned list are objects of the same class.	`inputRDD.top(2)`	{3,3}

`List<T> takeSample(`
`  boolean withReplacement,`
`  int num,`
`  long seed`
`)`
	Return a (Java) List containing a random sample of size n of the RDD.The objects of the RDD and objects of the returned list are objects of the same class.	`inputRDD.takeSample(`
		`  false, 1, 1`
		`)`
			Nondeterministic

`T reduce(`
`  Function2<T,T,T> f`
`)`
	Return a single Java object obtained by combining the values of the objects of the RDD by using a user provide “function”. The provided “function” must be associative and commutativeThe object returned by the method and the objects of the RDD belong to the same class.	`inputRDD.reduce(`
		`  (x, y) → x + y`
		`)`
			9

`T fold(`
`  T zeroValue,`
`  Function2<T,T,T> f`
`)`
	Same as reduce but with the provided zero value.	`inputRDD.fold(`
		`  0, (x, y) → x + y`
		`)`
			9

`U aggregate(`
`  U zeroValue,`
`  Function2<U,T,U> seqOp,`
`  Function2<U,U,U> combOp`
`)`
	Similar to reduce() but used to return a different type.	Compute a pair of integers where the first one is the sum of the values of the RDD and the second the number of elements	(9,4)

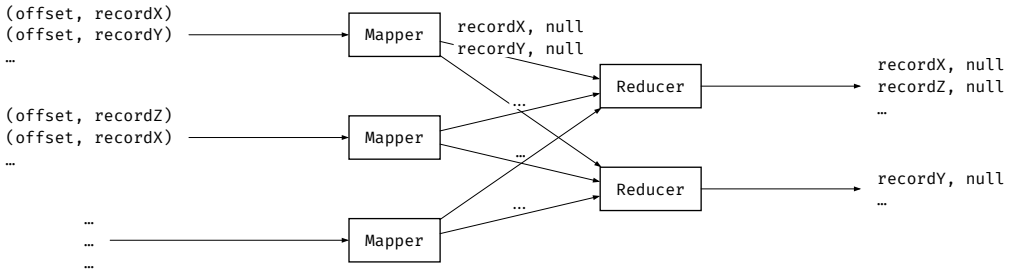
examples based on {(“k1”, 2), (“k3”, 4), (“k3”, 6)}			
`Map<K,V> countByKey()`	Return a local Java java.util.Map containing the number of elements in the input PairRDD for each key of the input PairRDD.	`inputRDD.countByKey()`	{(“k1”, 1), (“k3”, 2)}
`Map<K,V> collectAsMap()`	Return a local Java java.util.Map containing the pairs of the input PairRDD	`inputRDD.collectAsMap()`	{(“k1”, 2), (“k3”, 6)} Or{(“k1”, 2), (“k3”, 4)} Depending on the order of the pairs in the PairRDD
`List<V> lookup(K key)`	Return a local Java java.util.List containing all the values associated with the key specified as parameter	`inputRDD.lookup(“k3”)`	{4, 6}
examples based on {1.5, 3.5,2.0} as a JavaDoubleRDD			
`Double sum()`	Return the sum of the elements of the RDD.	`inputRDD.sum()`	7.0
`Double mean()`	Return the mean of the elements of the RDD.	`inputRDD.mean()`	2.3333
`Double variance()`	Return the variance of the elements of the RDD.	`inputRDD.variance()`	0.7223
`Double stdev()`	Return the standard deviation of the elements of the RDD.	`inputRDD.stdev()`	0.8498
`Double max()`	Return the maximum of the elements of the RDD.	`inputRDD.max()`	3.5
`Double min()`	Return the minimum of the elements of the RDD.	`inputRDD.min()`	1.5

Persistance and Cache: Storage Levels

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on (local) disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

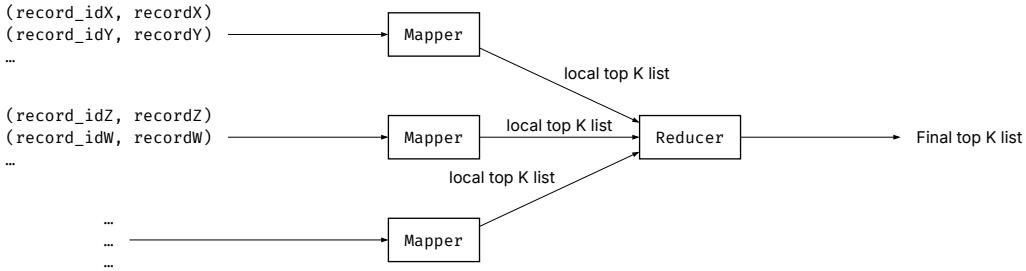
Design Patterns for MapReduce applications

Distinct



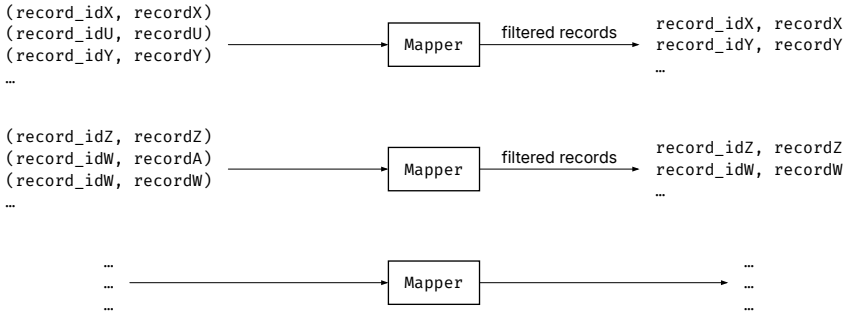
Top K

- k is small enough to fit in memory
- initialization perfomed in the setup method of the Mapper
- the map function updates the current in-mapper top k list
- the cleanup method emits the (key, value) pairs associated with the local top k records
  - key is a NullWritable
  - value is the value of the record
- only one reducer is instanciated
- all pairs have the same key, the reduce method is called only once



Filtering

- map-only job



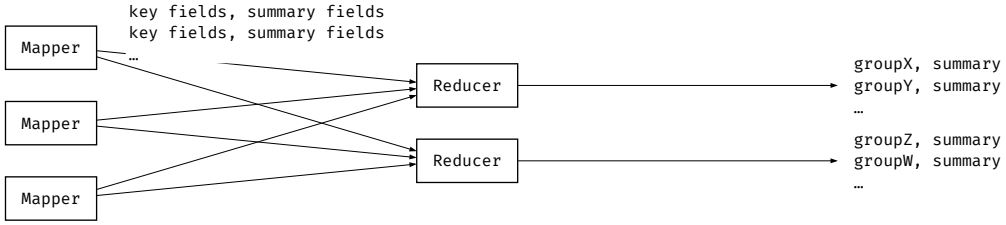
Numerical Summarization

Group records by key and compute a numerical aggregate function (e.g., sum, average, max, min, etc.) for each group

Mapper emits (key, value) pairs
- key is associated with the fields used to define groups
- value is associated with the fields used to compute the aggregate statistic

Reducer receives a set of numerical values for each `GROUP BY` key and computes the final statistic for each group

Combiners can be used if the statistic is commutative and associative to speed up the computation

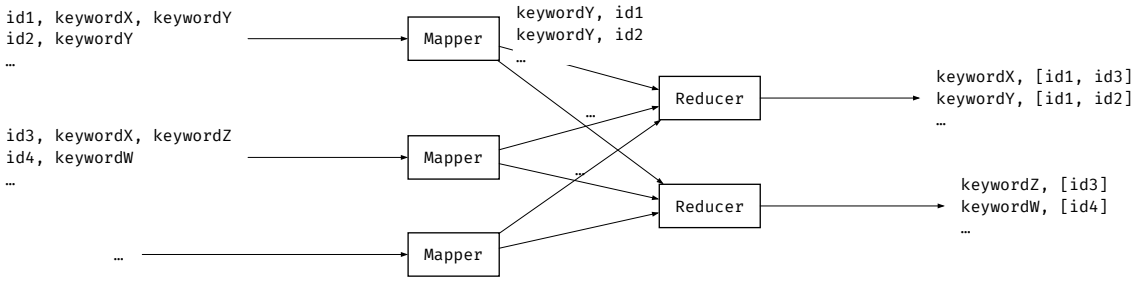


Inverted Index

Used to improve search efficiency

Mapper emits (key, value) pairs where
- key is the set of fields to index (a keyword)
- value is a unique identifier of the objects associated with the keyword

Reducer receives a set of unique identifiers for each keyword and concatenates them



Reduce-side Natural Join

Mappers are 2, one for each table, they emit (key, value) pairs where
- key is the join attribute(s)
- value is the concatenation of the name of the table of the current record and the content of the current record

Reducers iterate over the values associated with each key and compute the “local natural join” for the current key
- generate a copy for each pair of values such that one record is a record of the first table and the other is the record of the other table

For instance the (key, [list of values]) pair (UID1,[“User:Paolo,Garza”,"Likes:horror","Likes:adventure"]) will generate the following pairs
- (UID1, “Paolo,Garza,horror”)
- (UID1, “Paolo,Garza,adventure”)

Map-side Natural Join

Used when one of the two tables is small enough to fit in memory

Map-only job

Mapper receives one input (key, value) pair for each record of the large table and joins it with the “small” table
- the distributed cache approeaach is used to provide a copy of the small table to each mapper
- each mapper perforoms the “local natural join” between the current record it is processing and the records of the small table
- the content of the small table (file) is loaded in the main memory of each mapper during the execution of its setup method

Example Hadoop Mapper

```
class MapperBigData extends Mapper<LongWritable, Text, String, Integer> {

  private final Map<String, Integer> osToNumPatches;

  @Override
  protected void setup(Context context) {
    osToNumPatches = new HashMap<>();
  }

  protected void map(LongWritable key, Text value, Context context) {
    String[] fields = value.toString().split(","); // for blank you can use "\\s+"

    if (fields[0].matches("S*")) // matches accepts regex
      return;

    String[] dateS = fields[1].split("/");
    Integer date = Integer.parseInt("" + dateS[0] + dateS[1] + dateS[2]);

    if (date < 20210704 || date > 20220703)
      return;

    osToNumPatches.merge(fields[0], 1, Integer::sum);
  }

  @Override
  protected void cleanup(Context context) {
    osToNumPatches.forEach((k, v) -> context.write(k, v));
  }
}
```

Example Hadoop Reducer

```
class ReducerBigData extends Reducer<String, Integer, String, NullWritable> {

  private final Map<String, Integer> osToNumPatches = new HashMap<>();

  protected void reduce(String key, Iterable<Integer> values, Context context) {
    values.forEach(v -> osToNumPatches.merge(key, v, Integer::sum));

    int max = Collections.max(osToNumPatches.values());

    List<String> maxOs = new ArrayList<>();
    osToNumPatches.forEach((k, v) -> {
      if (v == max)
        maxOs.add(k);
    });

    Collections.sort(maxOs); // to sort in reverse use Collections.sort(
                             // maxOs, Collections.reverseOrder());

    context.write(maxOs.get(0), NullWritable.get());
  }
}
```

Example Spark Driver

```
public class SparkDriver {

    public static void main(String[] args) {
        SparkConf conf = new SparkConf().setAppName("Exam20220202 Spark");
        JavaSparkContext sc = new JavaSparkContext(conf);

        JavaRDD<String> houseRDD = sc.textFile(args[0]);
        JavaRDD<String> consumptionRDD = sc.textFile(args[1]);

        // Part 1 filter only the readings associated with year 2022
        JavaRDD<String> consumption2022 = consumptionRDD
                .filter(s -> {
                    String[] fields = s.split(",");
                    String date = fields[1];
                    return date.startsWith("2022");
                });

        // compute the total amount of energy consumed in year 2022 for each house
        // key = houseID
        // value = kWh consumed in year 2022
        JavaPairRDD<String, Double> totalCons2022 = consumption2022
                .mapToPair(s -> {
                    String[] fields = s.split(",");
                    String hid = fields[0];
                    Double consumption = Double.parseDouble(fields[2]);
                    return new Tuple2<>(hid, consumption);
                })
                .reduceByKey((v1, v2) -> v1 + v2);

        // compute the avg power consumption per day
        // key = houseID
        // value = avg kWh consumed per day in year 2022
        // and filter only those with high avg consumption
        JavaPairRDD<String, Double> highAvgDailyCons = totalCons2022
                .mapValues(v -> v / 365)
                .filter(i -> i._2() > 30);

        // compute the pairRDD house -> country
        // key = houseID
        // value = country
        JavaPairRDD<String, String> houseCountry = houseRDD.mapToPair(s -> {
            String[] fields = s.split(",");
            String hid = fields[0];
            String country = fields[2];
            return new Tuple2<>(hid, country);
        });

        // keep an RDD containing countries with at least one house with high avg power
        // consumption
        JavaRDD<String> countriesWithHighAvgPwrConsHouses = houseCountry
                .join(highAvgDailyCons)
                .map(it -> it._2()._1());

        // compute an RDD with all the countries
        // and subtract the countries with at least one house with high avg power
        // consumption
        JavaRDD<String> res1 = houseCountry
                .map(v -> v._2())
                .distinct()
                .subtract(countriesWithHighAvgPwrConsHouses);
        res1.saveAsTextFile(args[2]);

        // Part 2 keep only the houses for which the total power consumption over 2021 is >
        // threshold kWh
        JavaPairRDD<String, Double> highTotalPowerCons2021 = consumptionRDD
                .filter(s -> {
                    String[] fields = s.split(",");
                    String date = fields[1];
                    return date.startsWith("2021");
                })
                .mapToPair(s -> {
                    String[] fields = s.split(",");
                    String hid = fields[0];
                    Double consumption = Double.parseDouble(fields[2]);
                    return new Tuple2<>(hid, consumption);
                })
                .reduceByKey((v1, v2) -> v1 + v2)
                .filter(v -> v._2() > 10000);

        // compute an RDD with
        // key = houseID
        // value = (country, city)
        JavaPairRDD<String, Tuple2<String, String>> citiesRDD = houseRDD
                .mapToPair(s -> {
                    String[] fields = s.split(",");
                    String hid = fields[0];
                    String city = fields[1];
                    String country = fields[2];

                    return new Tuple2<>(hid, new Tuple2<>(country, city));
                });

        // join the two RDDs and count for each city the number of houses with high
        // annual power consumption
        // and filter only those cities with value > 500
        // key = (country, city)
        // value = #houses with high power consumption
        JavaPairRDD<Tuple2<String, String>, Integer> highPwrConsHousesPerCity = highTotalPowerCons2021
                .join(citiesRDD)
                .mapToPair(it -> new Tuple2<>(it._2()._2(), 1))
                .reduceByKey((v1, v2) -> v1 + v2)
                .filter(it -> it._2() > 500);

        // count for each country the number of cities with at least 500 houses with
        // high annual power consumption
        // key = country
        // value = number of cities with at least 500 houses with high power consumption
        JavaPairRDD<String, Integer> highPwrConsCitiesPerCountry = highPwrConsHousesPerCity
                .mapToPair(it -> new Tuple2<>(it._1()._1(), 1))
                .reduceByKey((v1, v2) -> v1 + v2);

        JavaPairRDD<String, Integer> countries = houseCountry
                .mapToPair(it -> new Tuple2<>(it._2(), 0))
                .distinct();

        // add with a rightOuterJoin the countries with count == 0
        JavaPairRDD<String, Integer> res2 = highPwrConsCitiesPerCountry
                .rightOuterJoin(countries)
                .mapValues(it -> it._1().orElse(0));

        res2.saveAsTextFile(args[3]);
        sc.close();
    }
}
```

Example Spark Driver with SQL

```
public class SparkDriver {
    public static void main(String[] args) {
        // Create a Spark Session object and set the name of the application
        SparkSession ss = SparkSession.builder().appName("Exam20220202 Spark")
                .master("local").getOrCreate();

        // Define the dataframes associated with the input files
        Dataset<Row> houseDF = ss.read().format("csv")
                .option("header", false).option("inferSchema", true)
                .load(args[0])
                .withColumnRenamed("_c0", "HouseID")
                .withColumnRenamed("_c1", "City")
                .withColumnRenamed("_c2", "Country")
                .withColumnRenamed("_c3", "SizeSQM");

        // Register the temporary table houses
        houseDF.createOrReplaceTempView("houses");

        Dataset<Row> consumptionDF = ss.read().format("csv")
                .option("header", false).option("inferSchema", true)
                .load(args[1])
                .withColumnRenamed("_c0", "HouseID")
                .withColumnRenamed("_c1", "Date")
                .withColumnRenamed("_c2", "kWh");

        // Register the temporary table consumption
        consumptionDF.createOrReplaceTempView("consumption");

        // Part 1 Register a function that given a date in the format "" returns the
        // year
        ss.udf().register("YEAR",
                (String date) -> Integer.parseInt(date.split("/")[0]),
                DataTypes.IntegerType);

        // Consider only the readings associated with the year 2022,
        // compute the average daily consumption for each house in the year 2022,
        // and select the houses with a daily consumption >30
        Dataset<Row> housesHighConsumptionDF = ss.sql("SELECT HouseID "
                + "FROM consumption "
                + "WHERE YEAR(Date)=2022 "
                + "GROUP BY HouseID "
                + "HAVING SUM(kWh)/365>30");

        // Register the temporary table housesHighCons
        housesHighConsumptionDF.createOrReplaceTempView("housesHighCons");

        Dataset<Row> res1DF = ss.sql("SELECT DISTINCT Country "
                + "FROM houses "
                + "WHERE Country NOT IN ("
                + " SELECT Country "
                + " FROM housesHighCons, houses "
                + " WHERE housesHighCons.HouseID=houses.HouseID)");

        res1DF.write()
                .format("csv")
                .option("header", false)
                .save(args[2]);

        // Part 2 keep only the houses for which the total power consumption over 2021
        // is > threshold kWh
        Dataset<Row> highTotalPowerCons2021DF = ss.sql("SELECT HouseID "
                + "FROM consumption "
                + "WHERE YEAR(Date)=2021 "
                + "GROUP BY HouseID "
                + "HAVING SUM(kWh)>10000");

        // Register the temporary table housesHighCons2021
        highTotalPowerCons2021DF.createOrReplaceTempView("housesHighCons2021");

        // Select for each country the cities with at least 500 houses with high annual
        // power consumption
        Dataset<Row> coutryCityManyHighConsHousesDF = ss.sql("SELECT Country "
                + "FROM houses, housesHighCons2021 "
                + "WHERE houses.HouseID=housesHighCons2021.HouseID "
                + "GROUP BY City, Country "
                + "HAVING COUNT(*)>500");

        // Register the temporary table citiesWithManyHighConsHouses
        coutryCityManyHighConsHousesDF.createOrReplaceTempView("citiesWithManyHighConsHouses");

        // Compute for each country the number of cities with at least 500 houses with
        // high annual power consumption
        // We must consider also the countries without cities with at least 500 houses
        // with high annual power consumption
        Dataset<Row> highPwrConsCitiesPerCountryDF = ss.sql("SELECT Country,
SUM(CityWithManyHighConsHouses) "
                + "FROM ( (SELECT Country, 1 as CityWithManyHighConsHouses FROM
citiesWithManyHighConsHouses) "
                + "        UNION ALL " // Note SQL command "UNION ALL" and not the SQL command "UNION"
                + "        (SELECT Country, 0 as CityWithManyHighConsHouses FROM Houses) "
                + "   ) "
                + "GROUP BY Country");

        highPwrConsCitiesPerCountryDF.write()
                .format("csv")
                .option("header", false)
                .save(args[3]);

        ss.stop(); // Close the Spark session
    }
}
```