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)} (create from regular RDDs with maptToPair() or flatMapToPair())			
`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

Conversions

Conversion	Purpose	Example	Result
starting from a regular RDD			

`JavaPairRDD<K,V>`
`mapToPair(`
`  PairFunction<T,K,V> f`
`)`
	Return a PairRDD<K,V> containing the pairs of the input RDD. The function f is applied to each element of the input RDD and returns a pair (k,v) for each element.	`x -> new Tuple2<>(x, x+1)`	{(1,2), (2,3), (3,4), (3,4)}

`JavaPairRDD<K,V>`
`flatMapToPair(`
`  PairFlatMapFunction<T,K,V> f`
`)`
	Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.	`line → {`
		`  List<Tuple2<String,Integer>>`
		`  pairs = new ArrayList<>();`
		`  line.split(" ").forEach(word → pairs.add(`
		`  new Tuple2<>(word, 1)`
		`  ));`
		`  // creates a JavaPairRDD of (word, 1) pairs`
		`  // from a JavaRDD of lines`
		`  return pairs.iterator();`
		`}`	

`JavaDoubleRDD`
`mapToDouble(`
`  DoubleFunction<T> f`
`)`
	Return a DoubleRDD containing the result of applying the function f to each element of the input RDD.	`x -> x+1`	{2.0, 3.0, 4.0, 4.0}

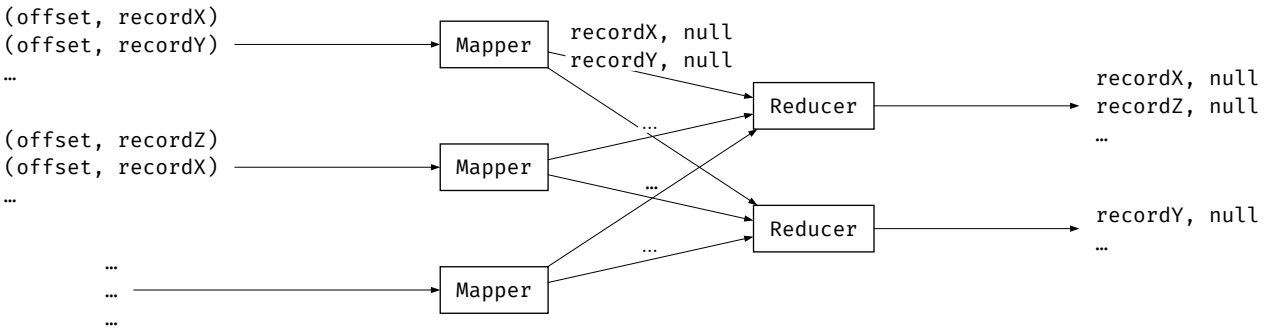
`JavaDoubleRDD`
`flatMapToDouble(`
`  DoubleFlatMapFunction<T> f`
`)`
	Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.	`sentence → {`
		`  ArrayList<Double> lengths=new ArrayList<Double>();`
		`  sentence.split(" ").forEach(word → lengths.add(`
		`  new Double(word.length())`
		`  ));`
		`  // creates a JavaDoubleRDD of word lengths`
		`  // from a JavaRDD of lines`
		`  return lengths.iterator();`
		`}`	

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.

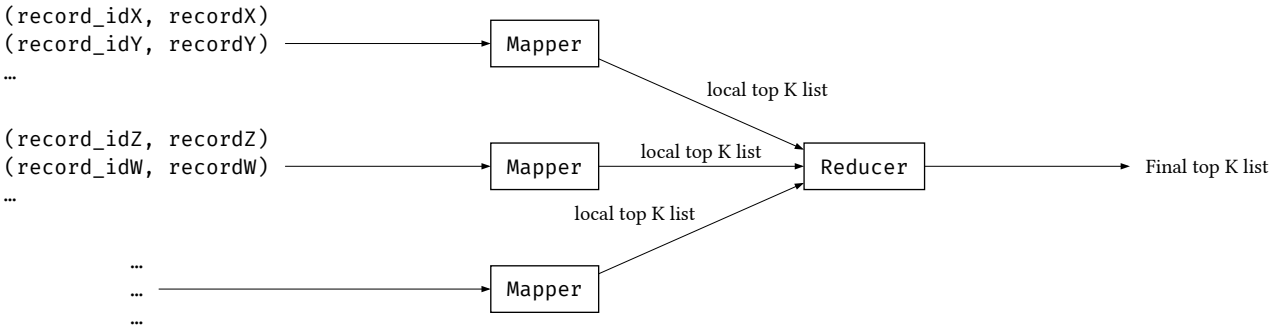
Design Patterns for MapReduce applications

Distinct



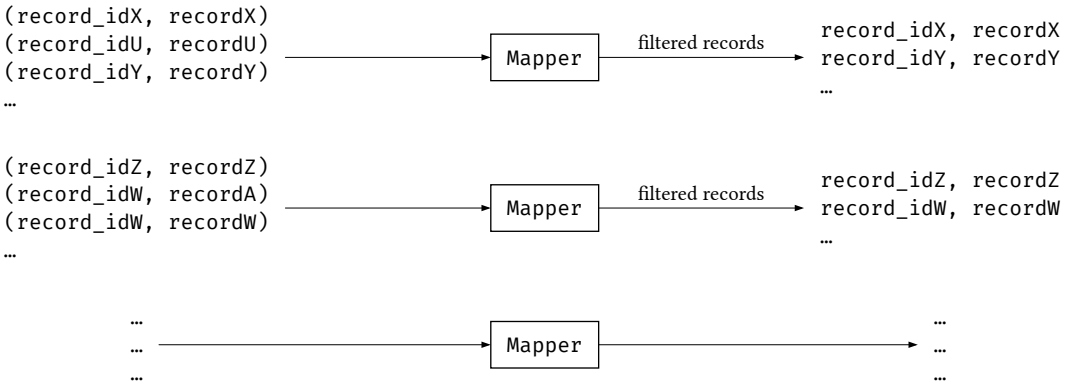
Top K

- 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



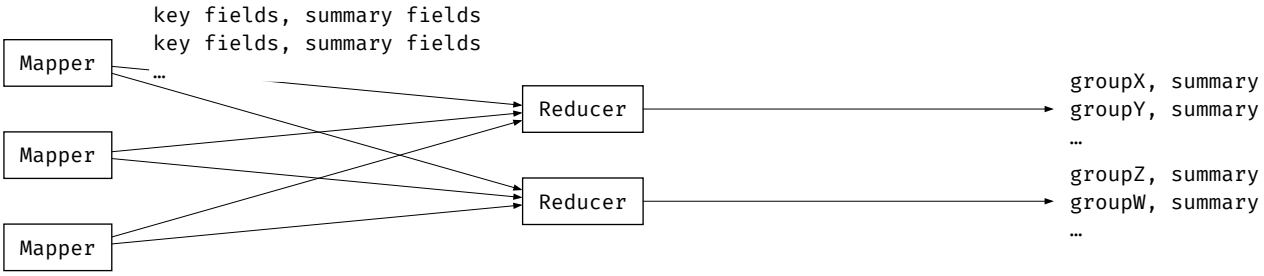
Numerical Summarization

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

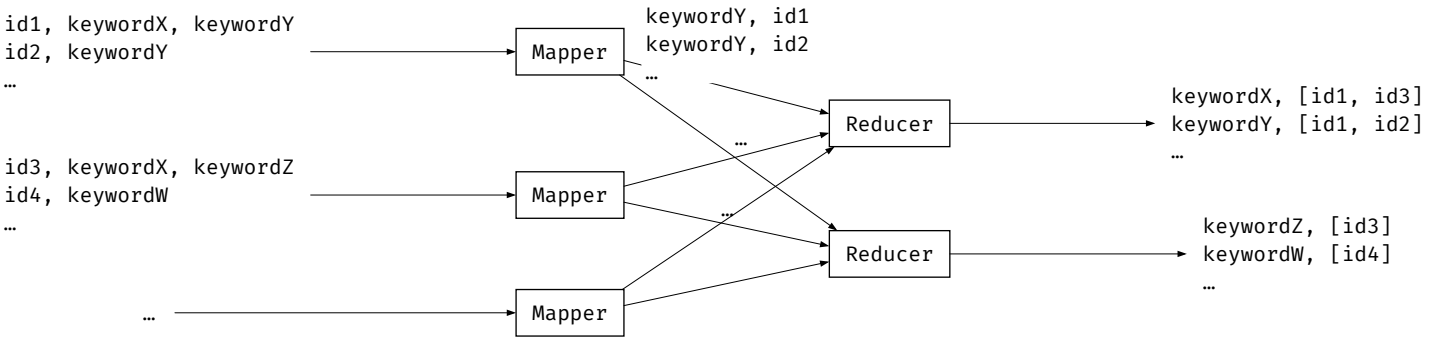


Inverted Index

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*")
    ) return; // matches accepts regex

    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();
  }
}
```

Another Example

```
public class SparkDriver {

  public static void main(String[] args) {
    // The following two lines are used to switch off some verbose log messages
    Logger.getLogger("org").setLevel(Level.OFF);
    Logger.getLogger("akka").setLevel(Level.OFF);

    // String companiesPath; Useless for this program
    String dataCenterPath, dailyPwrConsPath;
    String outputPath1, outputPath2;

    // companiesPath = args[0]; Useless for this program
    dataCenterPath = args[1];
    dailyPwrConsPath = args[2];
    outputPath1 = args[3];
    outputPath2 = args[4];

    // Create a configuration object and set the name of the application
    SparkConf conf = new SparkConf()
      .setAppName("Spark - Exam20220906")
      .setMaster("local");

    // Use the following command to create the SparkConf object if you want to run
    // your application inside Eclipse.
    // Remember to remove .setMaster("local") before running your application on
the
    // cluster
    // SparkConf conf=new SparkConf().setAppName("Spark Lab
#5").setMaster("local");
```

```
    // Create a Spark Context object
    JavaSparkContext sc = new JavaSparkContext(conf);

    //JavaRDD<String> companiesRDD = sc.textFile(companiesPath); Useless for this
program
    JavaRDD<String> dataCenterRDD = sc.textFile(dataCenterPath).cache();
    JavaRDD<String> pwrConsRDD = sc.textFile(dailyPwrConsPath);

    // Part 1
    // Count the total number of data centers worl-wide and compute the threshold
    int threshold = (int) (dataCenterRDD.count() * 0.9);

    // Filter only the dates associated with high power consumption
    // and prepare a pairRDD with
    // key = date
    // value = +1
    // to count the number of data centers with high power consumption for each
day.
    JavaPairRDD<String, Integer> highPwrConsDCPerDay = pwrConsRDD
      .filter(line -> {
        String[] fields = line.split(",");
        double pwrCons = Double.parseDouble(fields[2]);
        return pwrCons >= 1000;
      })
      .mapToPair(line -> {
        String[] fields = line.split(",");
        String date = fields[1];
        return new Tuple2<>(date, 1);
      })
      .reduceByKey((v1, v2) -> v1 + v2);

    JavaRDD<String> res1 = highPwrConsDCPerDay
      .filter(t -> t._2() >= threshold)
      .keys();

    res1.saveAsTextFile(outputPath1);

    // Part 2

    // Consider the power consumptions and keep only the entries related to year
2021
    // and obtain the following pairRDD
    // key = codDC
    // value = kWh
    // and use a reduceByKey to sum the power consumption for the entire year for
    // each data center
    JavaPairRDD<String, Double> yearlyPwrCons = pwrConsRDD
      .filter(line -> {
        String[] fields = line.split(",");
        return fields[1].startsWith("2021");
      })
      .mapToPair(line -> {
        String[] fields = line.split(",");
        String codDC = fields[0];
        double pwrCons = Double.parseDouble(fields[2]);
        return new Tuple2<>(codDC, pwrCons);
      })
      .reduceByKey((v1, v2) -> v1 + v2);

    // for each data center, keep the continent information
    // key = codDC
    // value = continent
    JavaPairRDD<String, String> dcAndContinent = dataCenterRDD.mapToPair(line -> {
        String[] fields = line.split(",");
        String codDC = fields[0];
        String continent = fields[4];
        return new Tuple2<>(codDC, continent);
      }
    );

    // Join yearlyPwrCons with dcAndContinent and
    // returns pairs
    // key = continent
    // value = (+1, kWhPerDataCenter2021)
    JavaPairRDD<String, Tuple2<Integer, Double>> continentOnePwr = yearlyPwrCons
      .join(dcAndContinent)
      .mapToPair(t ->
        new Tuple2<String, Tuple2<Integer, Double>>(
          t._2()._2(),
          new Tuple2<Integer, Double>(1, t._2()._1())
        )
      );

    // Sum the value parts to compute for each continent
    // the number of data centers and the total power consumption in the year
2021.
    // Finally, compute the avg power consumption
    // key = continent
    // value = (the number of data centers, avg power consumption in the year
2021)
    JavaPairRDD<String, Tuple2<Integer, Double>> numDCandAvgPwrCons =
continentOnePwr
      .reduceByKey((t1, t2) ->
        new Tuple2<Integer, Double>(t1._1() + t2._1(), t1._2() + t2._2())
      )
      .mapValues(t -> new Tuple2<Integer, Double>(t._1(), t._2() / t._1()))
      .cache();

    // compute the maximum number of data centers and the maximum avg consumption
in
    // the 2021 among the continents
    Tuple2<Integer, Double> maxDCAndConsumptionPerContinentThresholds =
numDCandAvgPwrCons
      .values()
      .reduce((t1, t2) ->
        new Tuple2<Integer, Double>(
          t1._1() > t2._1() ? t1._1() : t2._1(),
          t1._2() > t2._2() ? t1._2() : t2._2()
        )
      );

    // filter only those continents for which both constraints are satisfied
    JavaRDD<String> res2 = numDCandAvgPwrCons
      .filter(t ->
        (
          t
            ._2()
            ._1()
            .compareTo(maxDCAndConsumptionPerContinentThresholds._1()) ==
          0 &&
          t
            ._2()
            ._2()
            .compareTo(maxDCAndConsumptionPerContinentThresholds._2()) ==
          0
        )
      )
      .keys();

    res2.saveAsTextFile(outputPath2);
    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
  }
}
```

Example Spark Driver Dataset API

```
public class SparkDriver {

  public static void main(String[] args) {
    SparkSession spark = SparkSession
      .builder()
      .appName("MeetingStatistics")
      .getOrCreate();
    JavaSparkContext sc = new JavaSparkContext(spark.sparkContext());

    // Load the data
    Dataset<Row> users = spark.read().option("header", "true").csv(args[0]);
    Dataset<Row> meetings = spark.read().option("header", "true").csv(args[1]);
    Dataset<Row> invitations = spark
      .read()
      .option("header", "true")
      .csv(args[2]);

    // Filter for users with a Business pricing plan who organized at least one
    // meeting
    Dataset<Row> businessUsers = users.filter(
      col("PricingPlan").equalTo("Business")
    );
    Dataset<Row> businessMeetings = meetings.join(
      businessUsers,
      meetings.col("OrganizerUID").equalTo(businessUsers.col("UID"))
    );

    // Group by user ID and calculate statistics
    Dataset<Row> result1 = businessMeetings
      .groupBy("UID")
      .agg(
        avg("Duration").as("AverageDuration"),
        max("Duration").as("MaxDuration"),
        min("Duration").as("MinDuration")
      );

    // Store the result to HDFS
    result1.write().csv(args[3]);

    // Calculate the distribution of the number of invitations per organized
    // meeting
    Dataset<Row> invitationCounts = invitations.groupBy("MID").count();
    Dataset<Row> meetingsWithCounts = businessMeetings.join(
      invitationCounts,
      businessMeetings.col("MID").equalTo(invitationCounts.col("MID"))
    );

    // Classify meetings by size
    Dataset<Row> result2 = meetingsWithCounts.withColumn(
      "MeetingSize",
      when(col("count").gt(20), "large")
        .when(col("count").between(5, 20), "medium")
        .otherwise("small")
    );

    // Count the number of each size of meeting per user
    result2 = result2.groupBy("UID", "MeetingSize").count();

    // Store result to HDFS
    result2.write().csv(args[4]);

    sc.close();
  }
}
```

Another Example

```
public class HouseWaterConsumption {

  public static void main(String[] args) {
    SparkSession spark = SparkSession
      .builder()
      .appName("HouseWaterConsumption")
      .getOrCreate();

    // Read the input files
    Dataset<Row> houses = spark
      .read()
      .format("csv")
      .option("header", "false")
      .load(args[0]);
    houses =
      houses.withColumnRenamed("_c0", "HID").withColumnRenamed("_c1", "City");

    Dataset<Row> consumption = spark
      .read()
      .format("csv")
      .option("header", "false")
      .load(args[1]);
    consumption =
      consumption
        .withColumnRenamed("_c0", "HID")
        .withColumnRenamed("_c1", "Date")
        .withColumnRenamed("_c2", "M3");

    // Calculate the water consumption per trimester for each house for the years
2021 and 2022
    consumption =
      consumption
        .withColumn("Year", year(to_date(col("Date"), "yyyy/MM")))
        .withColumn("Trimester", quarter(to_date(col("Date"), "yyyy/MM")))
        .groupBy("HID", "Year", "Trimester")
        .agg(sum("M3").alias("M3"));

    WindowSpec windowSpec = Window
      .partitionBy("HID", "Trimester")
      .orderBy("Year");
    consumption =
      consumption
        .withColumn("PrevM3", lag("M3", 1).over(windowSpec))
        .withColumn(
          "Increased",
          when(col("M3").gt(col("PrevM3")), 1).otherwise(0)
        );

    // Count the number of trimesters with increased consumption in 2022
    Dataset<Row> increasedConsumption = consumption
      .filter("Year = 2022")
      .groupBy("HID")
      .agg(sum("Increased").alias("CountIncreased"));

    // Filter the houses that have an increased consumption in at least three
trimesters in 2022
    Dataset<Row> selectedHouses = increasedConsumption.filter(
      "CountIncreased >= 3"
    );

    // Join with the houses DataFrame and save the result to the first HDFS output
folder
    Dataset<Row> result = houses
      .join(selectedHouses, "HID")
      .select("HID", "City");
    result.write().format("csv").save(args[2]);

    // Calculate the annual water consumption for each house
    Dataset<Row> annualConsumption = consumption
      .groupBy("HID", "Year")
      .agg(sum("M3").alias("AnnualM3"));

    WindowSpec windowSpec2 = Window.partitionBy("HID").orderBy("Year");
    annualConsumption =
      annualConsumption
        .withColumn("PrevAnnualM3", lag("AnnualM3", 1).over(windowSpec2))
        .withColumn(
          "Decreased",
          when(col("AnnualM3").lt(col("PrevAnnualM3")), 1).otherwise(0)
        );

    // Count the number of houses with at least one annual consumption decrease
for each city
    Dataset<Row> decreasedConsumption = annualConsumption
      .filter("Decreased = 1")
      .groupBy("HID")
      .agg(count("Decreased").alias("CountDecreased"));
    Dataset<Row> housesWithDecreasedConsumption = houses.join(
      decreasedConsumption,
      "HID"
    );
    Dataset<Row> countDecreased = housesWithDecreasedConsumption
      .groupBy("City")
      .agg(count("HID").alias("CountHouses"));

    // Filter the cities with at most 2 houses with at least one annual
consumption decrease
    Dataset<Row> selectedCities = countDecreased.filter("CountHouses <= 2");

    // Save the result to the second HDFS output folder
    selectedCities.select("City").write().format("csv").save(args[3]);

    spark.stop();
  }
}
```

Another Example

```
public class SparkDriver {

  public static void main(String[] args) {
    ...

    // Number of observations for the Most Relevant NEOs from 2023. Considering
    // only the observations starting from 2023, the application aims to
    // calculate the number of observations associated with the Most Relevant NEOs
    // The Most Relevant NEOs are the NEOs that (i) have not already fallen and
    // (ii) are characterized by a dimension exceeding the average dimension
    // considering all the registered NEOs in the “NEObjects.txt” file. Calculate
    // the number of observations starting from 2023 for each Most Relevant NEO,
    // sort the result by this number in descending order, and store the result
    // in the first HDFS output folder. Consider only the Most Relevant NEOs that
    // have been observed at least one time starting from 2023 in this first part
    // of the application. The first HDFS output folder must contain information
    // in the format "NEOID,Number of observations starting from 2023" for
    // the Most Relevant NEOs (one line for each Most Relevant NEO).

    Dataset<Row> avgDimension = neos.agg(avg("Dimension").as("AvgDim"));

    Dataset<Row> mostRelNEOs = neos
      .join(avgDimension, neos.col("Dimension").gt(avgDimension.col("AvgDim")))
      .filter(neos.col("alreadyFallen").equalTo(false))
      .select("NEOID")
      .cache();

    Dataset<Row> observationsFrom2023 = observations
      .filter(year(observations.col("ObsDateTime")).geq(2023))
      .select("NEOID", "ObservatoryID")
      .cache();

    Dataset<Row> res1 = observationsFrom2023
      .join(
        mostRelNEOs,
        observationsFrom2023.col("NEOID").equalTo(mostRelNEOs.col("NEOID"))
      )
      .groupBy("NEOID")
      .agg(count("*").as("ObservationCount"))
      .orderBy(desc("ObservationCount"));

    res1.write().format("csv").option("header", false).save(outputPath1);

    // The most relevant NEOs observed by a few observatories starting from 2023.
    // The second part of this application considers only the Most Relevant NEOs
    // observed by less than 10 unique observatories starting from the year 2023.
    // For each Most Relevant NEO of that subset, store in the second HDFS output
    // folder its identifier (NEOID) and the identifiers (ObservatoryIDs) of the
    // unique observatories that observed it starting from the year 2023
    // (one pair (NEOID, ObservatoryID) per output line). If a NEOD For each of
    // the selected Most Relevant NEOs never observed starting from 2023, store
    // the pair (NEOID, "NONE") in the output folder. The output format of each
    // output line is “NEOID, ObservatoryID”. Report the string "NONE" instead
    // of the ObservatoryID for each of the selected Most Relevant NEOs
    // never observed starting from 2023.
    Dataset<Row> res2 = mostRelNEOs
      .join(
        observationsFrom2023,
        mostRelNEOs.col("NEOID").equalTo(observationsFrom2023.col("NEOID")),
        "left_outer"
      )
      .groupBy("NEOID")
      .agg(countDistinct("ObservatoryID").as("count"))
      .filter(col("count").lt(10))
      .select(
        col("NEOID"),
        // coalesce returns the first column that is not null, or null if all
        // inputs are null. lit creates a Column of literal value.
        coalesce(col("ObservatoryID"), lit("NONE")).as("ObservatoryID")
      );

    res2.write().format("csv").option("header", false).save(outputPath2);

    ss.stop();
  }
}
```

Spark Example Left and Left-Anti Join

```
public class SparkDriver {

  public static void main(String[] args) {
    ...

    // Countries without houses that are characterized by a high average daily
    // consumption in the year 2022. The first part of this application
    // considers only the year 2022 and selects the countries that are never
    // associated with houses with a high average daily consumption in the year
    // 2022. A house is considered a house with a high average daily consumption
    // in the year 2022 if the average daily consumption of that house in the
    // year 2022 is greater than 30 kWh. Store the selected countries in the
    // first HDFS output folder (one country per line).

    Dataset<Row> housesWithHighAvgConsumption2022 = consumption
      .filter(year(consumption.col("date")).equalTo(2022))
      .groupBy("id")
      .agg(avg("consumption").as("avg_consumption"))
      .filter("avg_consumption > 30")
      .select("id");

    Dataset<Row> res1 = houses.join(
      housesWithHighAvgConsumption2022,
      houses.col("id").equalTo(housesWithHighAvgConsumption2022.col("id")),
      "left_anti"
    );

    res1.select("country").distinct().write().csv(args[2]);

    // The number of cities with many houses with high power consumption in the
    // year 2021 for each country. This second part of the application considers
    // only the consumption in the year 2021 and computes for each country the
    // number of cities each one with at least 500 houses with high annual
    // consumption in the year 2021. Specifically, a house is classified as a
    // “house with a high annual consumption in the year 2021” if its annual
    // consumption in the year 2021 is greater than 10000 kWh. Store the result
    // in the second output folder (one country per output line). The output
    // format is “country,number of cities each one with at least 500 houses
    // with a high annual consumption in the year 2021 for that country”. Save
    // also the information for the countries with no cities with at least 500
    // houses with a high annual consumption in the year 2021. In those cases,
    // the output line will be “country,0”.

    Dataset<Row> housesWithHighCumConsumption2021 = consumption
      .filter(year(consumption.col("date")).equalTo(2021))
      .groupBy("id")
      .agg(sum("consumption").as("tot_consumption"))
      .filter("tot_consumption > 10000");

    Dataset<Row> citiesWithMoreThan500Houses = houses
      .join(housesWithHighCumConsumption2021, "id")
      .groupBy("city")
      .agg(count("*").as("num_houses"))
      .filter("num_houses ⩾ 500");

    Dataset<Row> countriesWithNumCities = houses
      .join(citiesWithMoreThan500Houses, "city")
      .groupBy("country")
      .agg(countDistinct("city").as("num_cities"));

    Dataset<Row> res2 = houses
      .join(countriesWithNumCities, "country", "left")
      .select(
        houses.col("country"),
        coalesce(col("num_cities"), lit(0).cast("integer")).as("num_cities")
      );

    res2.distinct().write().csv(args[3]);

    ss.stop();
  }
}
```

Another Example

```
public class SparkDriver {

  public static void main(String[] args) {
  ...

    customers =
      customers.withColumn(
        "DateOfBirth",
        to_date(customers.col("DateOfBirth"), "yyyy/MM/dd")
      );
    itemsCatalog =
      itemsCatalog.withColumn(
        "FirstTimeInCatalog",
        to_timestamp(itemsCatalog.col("FirstTimeInCatalog"), "yyyy/MM/dd-
HH:mm:ss")
      );
    purchases =
      purchases.withColumn(
        "SaleTimestamp",
        to_timestamp(purchases.col("SaleTimestamp"), "yyyy/MM/dd-HH:mm:ss")
      );

    // Items purchased at least 10K times in 2020 and at least 10K times in
    // 2021. This first part of the application considers all the items included
    // in the catalog and selects the subset of items bought at least 10000
    // times in the year 2020 and at least 10000 times in the year 2021. Only
    // the subset of items that satisfy both conditions are selected. The
    // identifiers of the selected items are stored in the first HDFS output
    // folder (one ItemID per line).

    Dataset<Row> itemsPurchased10KTimes2020 = purchases
      .filter(year(purchases.col("SaleTimestamp")).equalTo(2020))
      .groupBy("ItemID")
      .agg(count("*").as("count"))
      .filter("count ⩾ 10000");

    Dataset<Row> itemsPurchased10KTimes2021 = purchases
      .filter(year(purchases.col("SaleTimestamp")).equalTo(2021))
      .groupBy("ItemID")
      .agg(count("*").as("count"))
      .filter("count ⩾ 10000");

    Dataset<Row> res1 = itemsPurchased10KTimes2020
      .join(itemsPurchased10KTimes2021, "ItemID")
      .select("ItemID");

    res1.write().csv(args[3]);

    // Items included in the catalog before the year 2020 with at least two
    // months in the year 2020 each one with less than 10 distinct customers.
    // This second part of the application considers only the items that were
    // included in the catalog before the year 2020
    // (i.e., the items with FirstTimeInCatalog<’2020/01/01’). Considering only
    // those items, an item is selected if it is characterized by at least two
    // months in the year 2020 such that each of those months has less than 10
    // distinct customers who purchased that item. The identifiers and the
    // categories of the selected items are stored in the second output folder
    // (one pair (ItemID, Category) per output line).
    //
    // NOTE: The months with less than 10 distinct customers can be either
    // consecutive or not consecutive.

    Dataset<Row> itemsIncludedInCatalogBefore2020 = itemsCatalog
      .filter(year(itemsCatalog.col("FirstTimeInCatalog")).lt(2020))
      .select("ItemID");

    Dataset<Row> itemsWithLessThan10CustomersEachMonth = purchases
      .join(itemsIncludedInCatalogBefore2020, "ItemID")
      .withColumn("Year", year(purchases.col("SaleTimestamp")))
      .withColumn("Month", month(purchases.col("SaleTimestamp")))
```

```
      .groupBy("Year", "Month", "ItemID")
      .agg(count_distinct(purchases.col("Username")).as("NumCustomers"))
      .filter("NumCustomers < 10");

    Dataset<Row> itemsWithLessThan10CustomersEachMonthInMoreThan2Months =
itemsWithLessThan10CustomersEachMonth
      .groupBy("Year", "ItemID")
      .agg(count_distinct(col("Month")).as("NumMonths"))
      .filter("NumMonths >= 2")
      .select("ItemID");

    Dataset<Row> res2 = itemsWithLessThan10CustomersEachMonthInMoreThan2Months
      .join(itemsCatalog, "ItemID")
      .select("ItemID", "Category");

    res2.write().csv(args[4]);

    ss.stop();
  }
}
```

Spark Example WindowSpec

```
public class SparkDriver {

  public static void main(String[] args) {
    ...

    // Movies that have been watched frequently but only in one year in the last
    // five years. Considering only the lines of WatchedMovies.txt related to
    // the last five years (i.e., the lines with StartTimestamp in the range
    // September 17, 2015 - September 16, 2020), the application selects the
    // movies that (i) have been watched only in one of those 5 years and (ii)
    // at least 1000 times in that year. For each of the selected movies, the
    // application stores in the first HDFS output folder its identifier (MID)
    // and the single year in which it has been watched at least 1000 times
    // (one pair (MID, year) per line).
    //
    // NOTE: The value of StartTimestamp is used to decide in which year a user
    // watched a specific movie. Do not consider the value of EndTimestamp.

    Dataset<Row> watchedInLast5Years = watchedMovies.where(
      col("StartTimestamp")
        .between(
          lit("2015-09-18 00:00:00").cast(DataTypes.TimestampType),
          lit("2020-09-16 23:59:59").cast(DataTypes.TimestampType)
        )
    );

    Dataset<Row> res1 = watchedInLast5Years
      .withColumn("Year", year(col("StartTimestamp")))
      .groupBy("MID")
      .agg(
        count_distinct(col("Year")).as("DifferentYearsCount"),
        count("*").as("TimesWatched"),
        first("Year").as("Year")
      )
      .filter("DifferentYearsCount == 1")
      .filter("TimesWatched >= 1000")
      .select("MID", "Year");

    res1.write().mode("overwrite").csv(args[3]);

    // Most popular movie in at least two years. Considering all the lines of
    // WatchedMovies.txt (i.e., all years), the application selects the movies
    // that have been the most popular movie in at least two years. The annual
    // popularity of a movie in a specific year is given by the number of
    // distinct users who watched that movie in that specific year. A movie is
    // the most popular movie in a specific year if it is associated with the
    // highest annual popularity of that year. The application stores in the
    // second HDFS output folder the identifiers (MIDs) of the selected movies
    // (one MID per line).
    //
    // NOTE: The value of StartTimestamp is used to decide in which year a user
    // watched a specific movie. Do not consider the value of EndTimestamp.

    Dataset<Row> moviesWithNumDistinctUsersPerYear = watchedMovies
      .withColumn("Year", year(col("StartTimestamp")))
      .groupBy("MID", "Year")
      .agg(count_distinct(col("Username")).as("NumDistinctUsers"));

    // use window when you need to partition without discarding the other
    // columns of the record
    WindowSpec windowSpec = Window
      .partitionBy("Year")
      .orderBy(col("NumDistinctUsers").desc());

    Dataset<Row> mostPopularMovies = moviesWithNumDistinctUsersPerYear
      // can also use row_number() instead of rank() to only select the first
      .withColumn("Rank", rank().over(windowSpec))
      .filter(col("Rank").equalTo(1));

    Dataset<Row> res2 = mostPopularMovies
      .groupBy("MID")
      .agg(count(col("Year")).as("NumYears"))
      .filter("NumYears >= 2")
      .select("MID");

    res2.write().mode("overwrite").csv(args[4]);

    ss.stop();
  }
}
```

Lots of Fun with WindowSpec

```
public class SparkDriver {

  public static void main(String[] args) {
    BasicConfigurator.configure();

    // The following two lines are used to switch off some verbose log messages
    Logger.getLogger("org").setLevel(Level.OFF);
    Logger.getLogger("akka").setLevel(Level.OFF);

    SparkSession ss = SparkSession
      .builder()
      .appName("exam_2020-06-16 spark")
      .master("local[12]")
      .config("spark.executor.memory", "3G")
      .config("spark.driver.memory", "12G")
      .getOrCreate();

    Dataset<Row> books = ss
      .read()
      .option("header", true)
      .option("inferSchema", true)
      .csv(args[0]);

    Dataset<Row> purchases = ss
      .read()
      .option("header", true)
      .option("inferSchema", true)
      .csv(args[1]);

    purchases = purchases.withColumn("Date", to_date(col("Date"), "yyyyMMdd"));

    books.printSchema();
    purchases.printSchema();

    // Maximum number of daily purchases per book in year 2018. Consider only
    // the purchases of years 2018 and only the books with at least one purchase
    // in year 2018. The application computes the maximum number of daily
    // purchases for each book. The application stores in the first HDFS output
    // folder for each book its identifier and its maximum number of daily
    // purchases (one pair (BID, maximum number of daily purchases) per line).

    Dataset<Row> purchases2018 = purchases
      .filter(year(col("date")).equalTo(2018))
      .cache();

    WindowSpec windowSpec = Window.partitionBy("Date", "BID");

    Dataset<Row> res1 = purchases2018
      .withColumn("NumOfDailyPurchases", count("*").over(windowSpec))
      .groupBy("BID")
      .agg(max("NumOfDailyPurchases"));

    res1.write().mode("overwrite").csv(args[2]);

    // Windows of three consecutive days with many purchases. Consider only the
    // purchases of years 2018. The application must select, for each book, all
    // the windows of three consecutive dates such that each date of the window
    // is characterized by a number of purchases that is greater than 10% of the
    // purchases of the considered book in year 2018. Specifically, given a book
    // and a window of three consecutive dates, that window of three consecutive
    // dates is selected for that book if and only if in each of those three
    // dates the number of purchases of that book is greater than 0.1*total
    // number of purchases of that book in year 2018. The application stores
    // the result in the second HDFS output folder. Specifically, each of the
    // selected combinations (book, window of three consecutive dates) is stored
    // in one output line and the used format is the following: (BID of the
    // selected book, first date of the selected window of three consecutive
    // dates).
    //
    // NOTE: that you can have overlapped windows of three consecutive dates
    // among the selected windows.

    // partition by date, order by date and select the range -1 to 1
    // (groups of 3 days at a time)
    WindowSpec windowSpec2 = Window
      .partitionBy("BID", "Date")
      .orderBy(col("Date"))
      .rowsBetween(-1, 1); // rangeBetween() can only be used
                           // with int on the order by

    WindowSpec windowSpec3 = Window.partitionBy(col("BID"), year(col("Date")));

    Dataset<Row> res2 = purchases2018
      .withColumn("Sum3Days", sum("Price").over(windowSpec2))
      .withColumn("SumYear", sum("Price").over(windowSpec3).as("SumYear"))
      .withColumn("FirstDay", first("Date").over(windowSpec2))
      .filter(col("Sum3Days").gt(col("SumYear").multiply(0.1)))
      .select("BID", "FirstDay");

    res2.write().mode("overwrite").csv(args[3]);

    ss.stop();
  }
}
```

Hadoop Example First with Max

```
// The first year with the maximum number of purchases. The application
// considers all purchases and selects the year with the maximum number of
// purchases. If there is more than one year associated with maximum number of
// purchases, the first one in the temporal order is selected. Store the
// selected year and the associated number of purchases in the output HDFS
// folder (the pair (year, number of purchases in that year).

class MapperBigData1
  extends Mapper<LongWritable, Text, IntWritable, IntWritable> {

  private IntWritable year = new IntWritable();
  private static IntWritable one = new IntWritable(1);

  protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    String[] fields = value.toString().split(",");
    String[] date = fields[0].split("/");

    year.set(Integer.parseInt(date[0]));
    context.write(year, one);
  }
}

public class ReducerBigData1
  extends Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {

  private IntWritable year = new IntWritable();
  private IntWritable count = new IntWritable();

  private int maxCount;
  private int minYear;

  @Override
  protected void setup(
    Reducer<IntWritable, IntWritable, IntWritable, IntWritable>.Context context
  ) throws IOException, InterruptedException {
    maxCount = -1;
    minYear = Integer.MAX_VALUE;
  }

  @Override
  protected void reduce(
    IntWritable key,
    Iterable<IntWritable> values,
    Reducer<IntWritable, IntWritable, IntWritable, IntWritable>.Context context
  ) throws IOException, InterruptedException {
    int accumulator = 0;
    for (IntWritable value : values) {
      accumulator += value.get();
    }

    if (accumulator > maxCount) {
      maxCount = accumulator;
      minYear = key.get();
    } else if (accumulator == maxCount && minYear > key.get()) {
      minYear = key.get();
    }
  }

  @Override
  protected void cleanup(
    Reducer<IntWritable, IntWritable, IntWritable, IntWritable>.Context context
  ) throws IOException, InterruptedException {
    year.set(minYear);
    count.set(maxCount);
    context.write(year, count);
  }
}

class MapperBigData2 extends Mapper<Text, Text, NullWritable, Text> {

  private Text year_count = new Text();

  protected void map(Text key, Text value, Context context)
    throws IOException, InterruptedException {
    year_count.set("" + key + "," + value);
    context.write(NullWritable.get(), year_count);
  }
}

public class ReducerBigData2
  extends Reducer<NullWritable, Text, IntWritable, IntWritable> {

  @Override
  protected void reduce(
    NullWritable key,
    Iterable<Text> values,
    Reducer<NullWritable, Text, IntWritable, IntWritable>.Context context
  ) throws IOException, InterruptedException {
    int maxCount = -1;
    int minYear = Integer.MAX_VALUE;

    for (Text value : values) {
      String[] split = value.toString().split(",");

      int year = Integer.parseInt(split[0]);
      int count = Integer.parseInt(split[1]);

      if (count > maxCount) {
        maxCount = count;
        minYear = year;
      } else if (count == maxCount && minYear > year) {
        minYear = year;
      }
    }

    context.write(new IntWritable(minYear), new IntWritable(maxCount));
  }
}
```

Hadoop Example Inverted Index

```
// Movies watched by one single user in year 2019. The application considers
// only the visualizations related to year 2019 (i.e., the lines of
// WatchedMovies with StartTimestamp in the range January 1, 2019 – December 31,
// 2019) and selects the movies that have been watched by one single user in
// 2019. Store the identifiers (MIDs) of the selected movies in the output HDFS
// folder (one MID per line).
//
// NOTE: If a movie has been watched many times in 2019 but always by the same
// user, that movie satisfies the constraint and must be selected.

class MapperBigData extends Mapper<LongWritable, Text, Text, Text> {

  private Text movie = new Text();
  private Text username = new Text();

  protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    String[] fields = value.toString().split(",");
    String[] date = fields[2].split("/");

    int year = Integer.parseInt(date[0]);

    if (year != 2019) return;

    movie.set(fields[1]);
    username.set(fields[0]);

    context.write(movie, username); // inverted index
  }
}

public class ReducerBigData extends Reducer<Text, Text, Text, NullWritable> {

  @Override
  protected void reduce(
    Text key,
    Iterable<Text> values,
    Reducer<Text, Text, Text, NullWritable>.Context context
  ) throws IOException, InterruptedException {
    String first_user = values.iterator().next().toString();

    for (Text value : values) {
      if (!value.toString().equals(first_user)) return;
    }

    context.write(key, NullWritable.get());
  }
}
```

Hadoop example combine two values

```
// Customers who bought the same book at least two times in year 2018. The
// application considers only the purchases of year 2018 and selects the
// identifiers of the customers who bought the same book at least two times in
// year 2018. For each of the selected customers store in the output HDFS folder
// one line for each of the books he/she bought at least two times in year 2018.
// Each output line has the format Customerid\tBID.

class MapperBigData extends Mapper<LongWritable, Text, Text, NullWritable> {

  private Text customer_book = new Text();

  protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    String[] fields = value.toString().split(",");
    String date = fields[2];

    if (!date.matches("^2018.*")) return; // equivalent to .startsWith("2018")

    customer_book.set("" + fields[0] + "," + fields[1]);

    context.write(customer_book, NullWritable.get());
  }
}

public class ReducerBigData extends Reducer<Text, NullWritable, Text, Text> {

  private Text customer = new Text();
  private Text book = new Text();

  @Override
  protected void reduce(
    Text key,
    Iterable<NullWritable> values,
    Reducer<Text, NullWritable, Text, Text>.Context context
  ) throws IOException, InterruptedException {
    int count = 0;

    while (values.iterator().hasNext()) {
      count++;
      values.iterator().next();
    }

    if (count < 2) return;

    String[] customer_book = key.toString().split(",");
    customer.set(customer_book[0]);
    book.set(customer_book[1]);

    context.write(customer, book);
  }
}
```