Master of Science and Technology - Data Science for Business

MAP543 - Database Management

CCF: Fast and Scalable Connected Component Computation in MapReduce

Arsène TRIPARD
Paul CUSSAC
Augustin DE LA BROSSE

Contact:
arsene.tripart@polytechnique.edu
paul.cussac@polytechnique.edu
augustin.de-la-brosse@polytechnique.edu

April 2022

1. Description of the adopted solution

As a reminder, our problem was to implement fast and scalable connected component computation in MapReduce as explained by Hakan Kardes, Siddharth Agrawal, Xin Wang, and Ang Su in their research paper “CCF: Fast and Scalable Connected Component Computation in MapReduce”

The adopted solution to solve this problem was coded in python with RDDs and is composed of 6 fonctions :

- *iterate_mapper* duplicates tuples by switching positions of the elements to get all combinations possible by mapping each pair of the input graph with a duplicate with switched positions.
- *iterate_sort_group_by_key* sorts and groups by the key.
- *one_record_iterate_reducer* is a function used in the *iterate_reducer* function (see below). For each pair (key, list of values), it outputs new pairs (value, minimum value among the list of values) under the condition that the key is greater than the minimum value among the list of values.
- *iterate_reducer* applies the previous function on all pairs of the input graph. And, thanks to an accumulator, counts the number of duplicated emits.
- *dedup* removes all duplicates from the output graph.
- *ccf_iterate* runs all the functions as long as new pairs are created during one iteration

This algorithm is based on the first version of CCF_iterate which pseudo code is given in Figure 2 of the paper.

2. Designed algorithms plus related global comments/description and comments to main fragments of code

```
def iterate_mapper(input_rdd, collect=False):
    """Takes RDD as input. Outputs an RDD.
    Duplicates tuples by switching position of the elements to get all
combinations possible.
    """
    # Maps each pair of the input rdd with a duplicate with switched
positions
    duplicated_rdd = input_rdd.map(lambda x: (x, (x[1], x[0])))

    # Flattens the result
    mapped_rdd = duplicated_rdd.flatMap(lambda x: x)

    if not collect:
        return mapped_rdd
    else:
```

```python
        return mapped_rdd.collect()

def iterate_sort_group_by_key(input_rdd, collect=False):
    """Takes RDD as input. Outputs an RDD.
    Sorts and groups by key.
    """
    # Groups by key
    grouped_rdd = input_rdd.groupByKey()

    # Sort by key
    sorted_rdd = grouped_rdd.sortBy(lambda x: x[0])

    if not collect:
        return sorted_rdd
    else:
        return sorted_rdd.collect()


def one_record_iterate_reducer(record, accum):
    """Takes as input a record (ie. a pair with a key and a list of
values) of input RDD. Outputs a record (ie. a pair with a key and a
value).
    For a record, does CCF-iterate job as in figure 2.
    """
    # Initializes the output with an empty list, the key with the first
element of the record, the values with the second element of the record
(a list of values that were grouped by key), the list of values with an
empty list and the minimum with the initialized key
    output = []
    key = record[0]
    values = record[1]
    valueList = []
    mini = key

    # This loop assigns to the minimum the smallest value of the second
element of the pair (ie. the list of values) if any of these values is
smaller than the key.
    # It also appends to the list of values initialized at the beginning
with each value of the second element of the pair
    for value in values:
        if value < mini:
            mini = value
        valueList.append(value)

    # Emits a pair (key, minimum value) if the key is greater than the
minimum value.
```

```
    # Also, we use a loop to incremente the accumulator by one each time
a value of the list of values is different from the minimum and to emit
a pair (value, minimum value)
    if mini < key:
        output.append((key, mini))
        for value in valueList:
            if mini != value:
                accum += 1
                output.append((value, mini))

    if output:
        return output

def iterate_reducer(input_rdd, accum, collect=False):
    """Takes RDD as input. Outputs an RDD.
    For each record in input RDD, calls one_record_iterate_reducer.
    """
    # CCF-iterate as in figure 2 by using our function
one_record_iterate_reducer
    reduced_rdd = input_rdd.map(lambda x: one_record_iterate_reducer(x,
accum))

    # Filters records equal to None
    filtered_rdd = reduced_rdd.filter(lambda x: x is not None)

    # Flattens the result
    output_rdd = filtered_rdd.flatMap(lambda x: x)

    if not collect:
        return output_rdd
    else:
        return output_rdd.collect()

def dedup(input_rdd, collect=False):
    """Takes RDD as input. Outputs same RDD without duplicates.
    """
    output_rdd = input_rdd.map(lambda x: ((x[0], x[1]),
None)).groupByKey().map(lambda x: x[0]).sortBy(lambda x: x[1]) # removes
all duplicates by sorting the pairs

    if not collect:
        return output_rdd
    else:
        return output_rdd.collect()
```

```python
def ccf_iterate(input_rdd, collect=False):
    """Takes RDD as input. Outputs RDD with connected components when
there is no more change.
    """
    first_iteration = True
    accum = sc.accumulator(0)
    count_iter = 0
    # Runs all the previous functions as long as new pairs are created
during one iteration
    while accum.value > 0 or first_iteration:
        first_iteration = False
        count_iter += 1
        start_time = time.time()

        # Re initialize the accumulator
        accum = sc.accumulator(0)

        # Maps input RDD thanks to the function iterate_mapper
        mapped_rdd = iterate_mapper(input_rdd, collect=False)

        # Sorts and groups by key thanks to the function
iterate_sort_group_by_key
        sorted_grouped_rdd = iterate_sort_group_by_key(mapped_rdd,
collect=False)

        # Performs reduce task thanks to the function iterate_reducer
        reduced_rdd = iterate_reducer(sorted_grouped_rdd, accum,
collect=False)

        # Removes duplicates thanks to the function dedup
        dedup_rdd = dedup(reduced_rdd, collect=False)

        if accum.value != 0:
            # Set input for new iteration
            input_rdd = dedup_rdd

        # Concludes when the graph stopped evoluating
        print(f"Finished iteration {count_iter} in
{time.time()-start_time:.2f} seconds")
        print(f"Accumulator is at {accum.value}")

    print(f"Job done in {count_iter} iterations")

    if not collect:
        return dedup_rdd
    else:
        return dedup_rdd.collect()
```

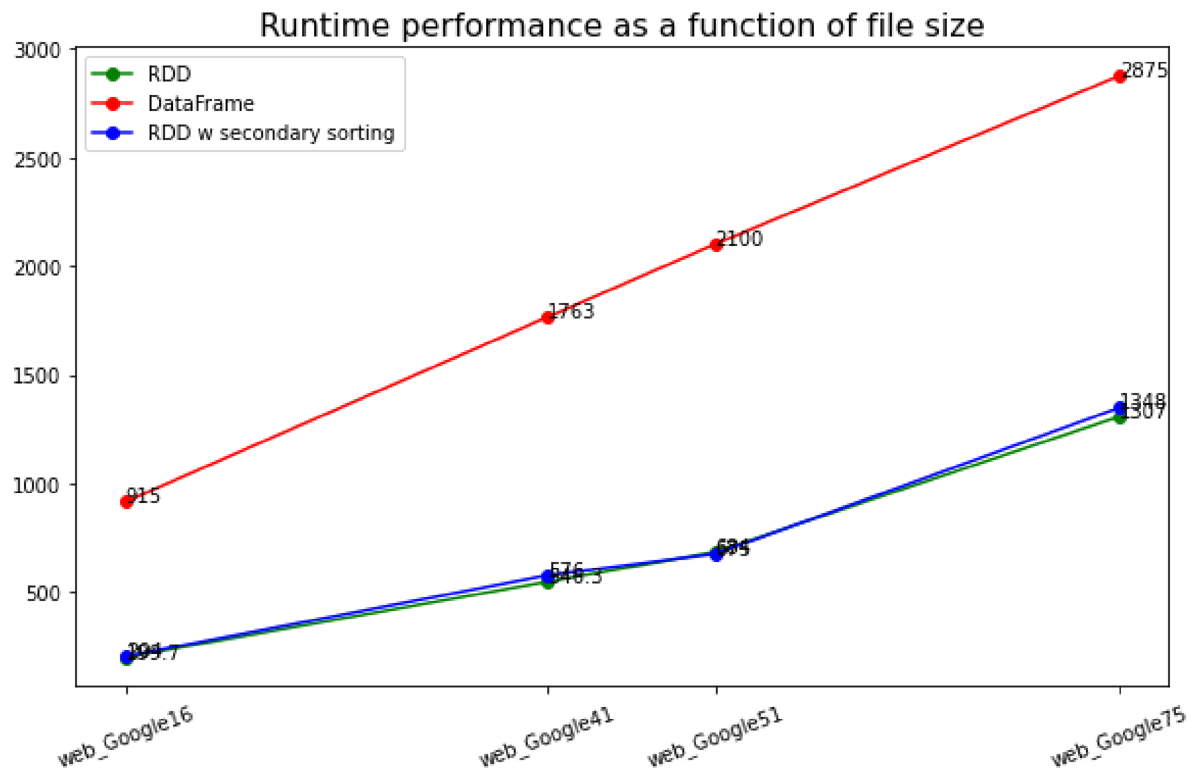
3. Experimental analysis, concerning in particular scalability

Unfortunately, our GCP accounts were not working properly as we showed you at a Lab session. So there was no way to conduct an experimental analysis on >1GB graph datasets, nor to reach a high parallelism level.

All analysis presented below have been been run locally on Jupyter Pyspark, and on Databricks with the free cluster they provide (15.25 GB | 2 cores).

We used the Google Web graph data, as proposed in the research paper, and we duplicated it in 4 files of increasing sizes: 16MB, 41MB, 51MB and 75MB. We compared the runtime of three versions of the algorithm: dataframe based, rdd based and rdd based with secondary sorting. According to the research paper, secondary sorting becomes very much worth it when the size of the connected components exceeds around 50K nodes. In the graph dataset we used for experimentation, the largest connected component contains over 250K nodes. So, we expect to see a performance improvement with the secondary sort approach.

Without getting into too much detail, here are the results of the first experimental analysis conducted:



4. Weak and strong points of the algorithms

+ **Runtime performance grows almost linearly with the size of the graph dataset.** This is very good news because it suggests that our approach is pretty scalable. It’s a shame our GCP credentials were out of order because we would have liked to see its performance on bigger graph datasets.
- **Secondary sort does not seem effective.** Two explanations are possible: either the secondary sort becomes useful with a much bigger dataset size. Or, the way we encoded it is incorrect. Looking back at it, we are using the simplest way of ordering values before going into the reduce phase. Maybe some repartitioning could make the values sorting faster.
- **The Spark DataFrame approach is pretty disappointing**. Our explanation is that Spark DataFrame are not suited for the reduce part of the algorithm. In the reduce part, we have to dig inside the values list, and loop over it before emitting. This is not an SQL-like type of operation. Similarly with Pandas, using the map function iterated over the rows is pretty time consuming.

5. Appendix

For each approach, we provide here our utils.py file which contains the functions later used in our notebook execution script.

DataFrame approach (python)

```
import time

from pyspark.sql.types import *
from pyspark.sql.functions import col, collect_list, udf, explode, size

def iterate_mapper(inputDF):
    """Takes Spark DataFrame as input. Outputs a Spark DF.
    Reverses the order of nodes in a second DF, and then unions original
DF and new one.
    """

    inv_inputDF = inputDF.select(["node2", "node1"])

    outputDF = inputDF.union(inv_inputDF)

    return outputDF

def iterate_sort_group_by_key(inputDF):
    """Takes Spark DataFrame as input. Outputs a Spark DF.
    Sorts and groups by node 1.
    """

    outputDF = inputDF\
        .orderBy("node1")\
        .groupBy("node1")\
        .agg(collect_list("node2").alias("values"))

    return outputDF

def one_record_reduce(key, values):
    """Takes as input a key and values extracted from Spark DataFrame.
Outputs a record.
    For a record, does CCF-iterate job as in figure 2.
    """

    output = []

    valueList = []
    mini = key
    for value in values:
        if value < mini:
            mini = value
        valueList.append(value)
    if mini < key:
        output.append((key, mini))
        for value in valueList:
            if mini != value:
                output.append((value, mini))

    return output

# Converts a custom function into Spark user-defined function
one_record_reduceUDF = udf(one_record_reduce,
ArrayType(ArrayType(IntegerType())))

def update_accumulator(inputDF, accum):
    """Takes Spark DataFrame as input. Outputs a Spark DF.
    Increases a Spark accumulator object as shown in Reduce pseudo-code.
    """
    # Compute number of elements in Arrays strictly bigger than 1
    arrayLengths = inputDF\
        .withColumn("values", size("values"))

    toAdd = arrayLengths\
        .where(arrayLengths.values > 2)\
        .count()

    accum.value += toAdd

    return accum


def iterate_reduce(inputDF, accum):
    """Takes Spark DataFrame as input. Outputs a Spark DF.
    For each record of input DataFrame, calls one_record_reduce.
    """
    # Implement reduce algo by row
    outputDF = inputDF\
        .withColumn("values", one_record_reduceUDF(col("node1"),
col("values")))

    # Update accumulator
    accum = update_accumulator(outputDF, accum)

    # Expand ArrayType column into rows
```

```python
    outputDF = outputDF\
    .drop("node1")\
    .withColumn("values", explode("values"))


    # Remove duplicates and orderBy
    outputDF = outputDF\
        .select(outputDF.values[0].alias("node1"),
outputDF.values[1].alias("node2"))\
        .dropDuplicates()\
        .orderBy(["node2", "node1"])

    return outputDF, accum

def ccf_iterate(sc, graphDF):
    """Takes RDD as input. Outputs RDD with connected components.
    """
    firstIter = True
    countIter = 0
    accum = sc.accumulator(1)
    initialDF = graphDF
    cumElapsedTime = 0

    while accum.value > 0 or firstIter:
        startTime = time.time()
        if not firstIter:
            initialDF = reducedDF

        print(f"Started iteration {countIter+1}")
        mappedDF = iterate_mapper(initialDF)
        groupedSortedDF = iterate_sort_group_by_key(mappedDF)
        accum = sc.accumulator(0)
        reducedDF, accum = iterate_reduce(groupedSortedDF, accum)

        firstIter = False
        countIter += 1
        elapsedTime = time.time() - startTime
        cumElapsedTime += elapsedTime

        print(f"Finished iteration {countIter} in {elapsedTime:.2f}")
        print(f"At this point, accumulator stands at {accum.value}")
        print("="*50)

    print(f"""Job done in {countIter} iterations and
{cumElapsedTime:.2f} seconds, {cumElapsedTime//60:.0f} min
{cumElapsedTime%60:.0f} seconds """)
```

RDD approach (python)

```python
import time

def iterate_mapper(input_rdd, collect=False):
    """Takes RDD as input. Outputs an RDD.
    Duplicates tuples by switching position of the elements, to get all
combinations possible.
    """
    # Duplicates the output by switching position
    duplicated_rdd = input_rdd.map(lambda x: (x, (x[1], x[0])))

    # Flattens the result
    mapped_rdd = duplicated_rdd.flatMap(lambda x: x)

    if not collect:
        return mapped_rdd
    else:
        return mapped_rdd.collect()

def iterate_sort_group_by_key(input_rdd, collect=False):
    """Takes RDD as input. Outputs an RDD.
    Sorts and groups by key.
    """
    # Groups by key
    grouped_rdd = input_rdd.groupByKey()

    # Groups by key
    sorted_rdd = grouped_rdd.sortBy(lambda x: x[0])

    if not collect:
        return sorted_rdd
    else:
        return sorted_rdd.collect()


def one_record_iterate_reducer(record, accum):
    """Takes as input a record of input RDD. Outputs a record.
    For a record, does CCF-iterate job as in figure 2.
    """
    output = []

    key = record[0]
    values = record[1]
    valueList = []
    mini = key
    for value in values:
        if value < mini:
            mini = value
        valueList.append(value)
    if mini < key:
        output.append((key, mini))
        for value in valueList:
            if mini != value:
                accum += 1
                output.append((value, mini))

    if output:
        return output

def iterate_reducer(input_rdd, accum, collect=False):
    """Takes RDD as input. Outputs an RDD.
    For each record in input RDD, calls one_record_iterate_reducer.
    """
    # CCF-iterate as in figure 2
    reduced_rdd = input_rdd.map(lambda x: one_record_iterate_reducer(x,
accum))

    # Filters records equal to None
    filtered_rdd = reduced_rdd.filter(lambda x: x is not None)

    # Flattens
    output_rdd = filtered_rdd.flatMap(lambda x: x)

    if not collect:
        return output_rdd
    else:
        return output_rdd.collect()

def dedup(input_rdd, collect=False):
    """Takes RDD as input. Outputs same RDD without duplicates.
    """
    output_rdd = input_rdd.map(lambda x: ((x[0], x[1]),
None)).groupByKey().map(lambda x: x[0]).sortBy(lambda x: x[1])

    if not collect:
        return output_rdd
    else:
        return output_rdd.collect()


def ccf_iterate(sc, input_rdd, collect=False):
    """Takes RDD as input. Outputs RDD with connected components.
    """
    first_iteration = True
    accum = sc.accumulator(0)
    count_iter = 0
    cumElapsedTime = 0

    while accum.value > 0 or first_iteration:
        first_iteration = False
        count_iter += 1
        print(f"Started iteration {count_iter}...")
        start_time = time.time()

        # Re initialize the accumulator
        accum = sc.accumulator(0)

        # Maps input RDD
        mapped_rdd = iterate_mapper(input_rdd, collect=False)

        # Sorts and groups by key
        sorted_grouped_rdd = iterate_sort_group_by_key(mapped_rdd,
collect=False)

        # Performs reduce task
        reduced_rdd = iterate_reducer(sorted_grouped_rdd, accum,
collect=False)

        # Deduplicates
        dedup_rdd = dedup(reduced_rdd, collect=False)

        if accum.value != 0:
            # Set input for new iteration
            input_rdd = dedup_rdd

        elapsedTime = time.time()-start_time
        cumElapsedTime += elapsedTime

        print(f"Finished iteration {count_iter} in
{time.time()-start_time:.2f} seconds")
        print(f"Accumulator is at {accum.value}")
        print("="*80)

    print(f"""Job done in {count_iter} iterations and
{cumElapsedTime:.2f} seconds, {cumElapsedTime//60:.0f} min
{cumElapsedTime%60:.0f} seconds """)
```

```
    if not collect:
        return dedup_rdd
    else:
        return dedup_rdd.collect()
```

RDD with secondary sort (python)

(only function has changed from RDD approach)

```
def iterate_sort_group_by_key(input_rdd, collect=False):
    """Takes RDD as input. Outputs an RDD.
    Sorts and groups by key.
    """
    # Groups by key
    grouped_rdd = input_rdd.groupByKey()

    # Groups by key
    sorted_rdd = grouped_rdd.sortBy(lambda x: x[0])

    # Sorts by value
    sec_sorted_rdd = sorted_rdd.map(lambda x: (x[0], sorted(x[1])))

    if not collect:
        return sec_sorted_rdd
    else:
        return sec_sorted_rdd.collect()


def one_record_iterate_reducer(record, accum):
    """Takes as input a record of input RDD. Outputs a record.
    For a record, does CCF-iterate job as in figure 2.
    """
    output = []

    key = record[0]
    values = record[1]
    #valueList = []
    mini = values[0]
    #for value in values:
    #    if value < mini:
    #        mini = value
    #    valueList.append(value)
    if mini < key:
        output.append((key, mini))
        for value in values:
            if mini != value:
                accum += 1
                output.append((value, mini))

    if output:
        return output
```

<u>RDD with secondary sort (scala)</u>
When we first tried to code the scala algorithm that could solve our problem, we wanted to translate the python code into a scala code. However, because we had to indicate the types of input and output, we found out that it was too complex to do this translation. Therefore, we totally changed our approach to find this solution.

```
class CCF {

    var accumulator = 0

    def ccf_iterate(rdd: RDD[(Int, Int)]): RDD[(Int, Int)] = {
      val processed_rdd = rdd
                          .flatMap(x => List((x._1, x._2), (x._2, x._1)))
// flatmapper emits both < a; b >, and < b; a > pairs
                          .groupByKey()  // we group by key
                          .map(x => (x._1, x._2, x._2.min))  // we
extract the min value
                          .filter(x => x._1 > x._3)  //  if vid is
smaller than amin, we do not emit any pair so we filter these pairs out
                          .map(x=> (x._1, x._2.filter(_ != x._3), x._3))
// we remove the min from list by filtering nit out

      accumulator = processed_rdd.map(_._2.size).sum().toInt  // we
count the number of pairs at this iteration
      println(s"Acumulator is at $accumulator")

      return processed_rdd.flatMap(x => List((x._1, x._3)) ++ x._2.map(y
=> (y, x._3))).distinct()  // the function finishes by returning all
pairs and removing the duplicates
    }

    def process_iteration(rdd: RDD[(Int, Int)]): RDD[(Int, Int)] = {
      var i = 0 // we count the number of iterations
      var current_rdd = rdd
      do {
        println(s"Iteration $i")
        current_rdd = this.ccf_iterate(current_rdd) // we apply our
function
        i+=1 //
      }
      while(accumulator > 0) // we keep working until our graph does not
move anymore

      println(s"\nJob done in $i iterations")

      return current_rdd.sortBy(_._2).sortBy(_._1) // the function
returns the result
    }
  }


val ccf = new CCF()
val output_example = ccf.process_iteration(graph)
output_example.collect()
```

Here, you can see some traces of research on how we tried to translate the python code into scala (we only show the functions that actually worked) :

```
def iterate_mapper_RDD(input_rdd: RDD[(Int, Int)]): RDD[(Int, Int)] = {
  var duplicated_rdd = input_rdd.map(x => List(x, (x._2, x._1)))

//    # Flattens the result
  var mapped_rdd = duplicated_rdd.flatMap(x => x)

  return mapped_rdd
}

def iterate_sort_group_by_key_RDD(input_rdd: RDD[(Int, Int)]): RDD[(Int,
Iterable[Int])] = {


//    """Takes RDD as input. Outputs an RDD.
//    Sorts and groups by key.
//    """
    // Groups by key
  var grouped_rdd = input_rdd.groupByKey()

    // Groups by key
  var sorted_rdd = grouped_rdd.sortBy(_._1)
  return sorted_rdd
}

def dedup_RDD(input_rdd: RDD[(Int, Int)]): RDD[(Int, Int)] = {
//    """Takes RDD as input. Outputs same RDD without duplicates.
//    """
    var output_rdd = input_rdd.map(x => ((x._1, x._2),
None)).groupByKey().map(x => x._1).sortBy(x => x._2)

    return output_rdd
}
```