Advanced topics – Part II

RDDs and Partitions

RDDs and Partitions

- The content of each RDD is split in partitions
  - The number of partitions and the content of each partition depend on how RDDs are defined/created
- The number of partitions impacts on the maximum parallelization degree of the Spark application
  - But pay attention that the amount of resources is limited (there is a maximum number of executors and parallel tasks)

How many Partitions are good ?

- Disadvantages of too few partitions
  - Less concurrency/parallelism
    - There could be worker nodes that are idle and could be used to speed up the execution of your application
  - Data skewing and improper resource utilization
    - Data might be skewed on one partition
      - One partition with many data
      - Many partitions with few data
    - The worker node that processes that large partition needs more time than the other workers
      - It becomes the bottleneck of your application

How many Partitions are good ?

- Disadvantages of too many partitions
  - Task scheduling may take more time than actual execution time if the amount of data in some partitions is too small

RDDs and Partitions

- Only some specific transformations set the number of partitions of the returned RDD
  - parallelize(), textFile(), repartition(), coalesce()
- The majority of the Spark transformations do not change the number of partitions
  - Those transformations preserve the number of partitions of the input RDD
    - i.e., the returned RDD has the same number of partitions of the input RDD

RDDs and Partitions

- parallelize(collection)
  - The number of partitions of the returned RDD is equal to ***sc.defaultParallelism***
  - Sparks tries to balance the number of elements per partition in the returned RDD
    - Elements are not assigned to partitions based on their value
- parallelize(collection, numSlices)
  - The number of partitions of the returned RDD is equal to **numSlices**
  - Sparks tries to balance the number of elements per partition in the returned RDD
    - Elements are not assigned to partitions based on their value

RDDs and Partitions

- textFile(pathInputData)
  - The number of partitions of the returned RDD is equal to the **number of input chunks/blocks** of the input HDFS data
  - Each partition contains the content of **one of the input blocks**
- textFile(pathInputData, minPartitions)
  - The user specified number of partitions must be greater than the number of input blocks
  - The number of partitions of the returned RDD is greater than or equal to the specified value **minPartitions**
  - Each partition contains **a part of one input blocks**

RDDs and Partitions

- repartition(numPartitions)
  - **numPartitions** can be greater or smaller than the number of partitions of the input RDD
  - The number of partitions of the returned RDD is equal to **numPartitions**
  - Sparks tries to balance the number of elements per partition in the returned RDD
    - Elements are not assigned to partitions based on their value
  - A **shuffle** operation is executed to assign input elements to the partitions of the returned RDD

RDDs and Partitions

- coalesce(numPartitions)
  - **numPartitions <** number of partitions of the input RDD
  - The number of partitions of the returned RDD is equal to **numPartitions**
  - Sparks tries to balance the number of elements per partition in the returned RDD
    - Elements are not assigned to partitions based on their value
  - Usually **no shuffle operation** is executed to assign input elements to the partitions of the returned RDD
  - coalesce is more efficient than repartition to reduce the number of partitions

Partitioning of Pair RDDs

- Spark allows specifying how to partition the content of RDDs of key-value pairs
  - The input pairs are grouped in partitions based on the integer value returned by a **function applied on the key** of each input pair
  - This operation can be useful to improve the efficiency of the next transformations by reducing the amount of shuffle operations and the amount of data sent on the network in the next steps of the application
    - Spark can optimize the execution of the transformations if the input RDDs of pairs are properly partitioned

partitionBy

- Partitioning is based on the **partitionBy()** transformation
- partitionBy(numPartitions)
  - The input pairs are grouped in partitions based on the integer value returned by a **default hash function** applied on the key of each input pair
  - A **shuffle** operation is executed to assign input elements to the partitions of the returned RDD

partitionBy

- Suppose that
  - The number of partition of the returned Pair RDD is **numPart**
  - The default partition function is **portable_hash**
  - Given an input pair **(key, value)** a copy of that pair will be stored in the partition number **n** of the returned RDD, where

n = portable_hash(key) % numPart

partitionBy

- Suppose that
  - The number of partition of the returned Pair RDD is **numPart**
  - The default partition function is **portable_hash**
  - Given an input pair **(key, value)** a copy of that pair will be stored in the partition number **n** of the returned RDD, where

n = portable_hash(key) % numPart

This function returns and integer

partitionBy: Custom function

- partitionBy(numPartitions, partitionFunc)
  - The input pairs are grouped in partitions based on the **integer** value returned by the user provided **partitionFunc** function
  - A **shuffle** operation is executed to assign input elements to the partitions of the returned RDD

partitionBy: Custom function

- Suppose that
  - The number of partition of the returned Pair RDD is **numPart**
  - The partition function is **partitionFunc**
  - Given an input pair **(key, value)** a copy of that pair will be stored in the partition number **n** of the returned RDD, where

n = partitionFunc(key) % numPart

partitionBy: Custom function

- Suppose that
  - The number of partition of the returned Pair RDD is **numPart**
  - The partition function is **partitionFunc**
  - Given an input pair **(key, value)** a copy of that pair will be stored in the partition number **n** of the returned RDD, where

$$n = \text{partitionFunc}(key) \% \text{numPart}$$

Custom partition function

partitionBy: Use case scenario

- Partitioning Pair RDDs by using partitionBy() is useful only when the same partitioned RDD is cached and **reused multiple times** in the application **in time and network consuming key-oriented transformations**
  - E.g., the same partitioned RDD is used in many join(), cogroup, groupyByKey(), .. transformations in different paths/branches of the application (different paths/branches of the DAG)
- **Pay attention** to the amount of data that is actually sent on the network
  - **partitionBy() can slow down your application** instead of speeding it up

partitionBy: Example

- Create an RDD from a textual file containing a list of pairs (pageID, list of linked pages)
- Implement the (simplified) PageRank algorithm and compute the pageRank of each input page
- Print the result on the standard output

partitionBy: Example

```
# Read the input file with the structure of the web graph
inputData = sc.textFile("links.txt")

# Format of each input line
# PageId,LinksToOtherPages - e.g., P3 [P1,P2,P4,P5]
def mapToPairPageIDLinks(line):
   fields = line.split(' ')
   pageID = fields[0]
   links = fields[1].split(',')

   return (pageID, links)


links = inputData.map(mapToPairPageIDLinks)\
        .partitionBy(inputData.getNumPartitions())\
        .cache()
```

partitionBy: Example

```
# Read the input file with the structure of the web graph
inputData = sc.textFile("links.txt")

# Format of each input line
# PageId,LinksToOtherPages - e.g., P3 [P1,P2,P4,P5]
def mapToPairPageIDLinks(line):
   fields = line.split(' ')
   pageID = fields[0]
   links = fields[1].split(',')
```

Note that the returned Pair RDD is partitioned and cached

```
links = inputData.map(mapToPairPageIDLinks)\
      .partitionBy(inputData.getNumPartitions())\
      .cache()
```

partitionBy: Example

```
# Initialize each page's rank to 1.0; since we use mapValues,
# the resulting RDD will have the same partitioner as links
ranks = links.mapValues(lambda v: 1.0)
```

partitionBy: Example

```
# Function that returns a set of pairs from each input pair
# input pair: (pageid, (linked pages, current page rank of pageid) )
# one output pair for each linked page. Output pairs:
# (pageid linked page,
#   current page rank of the linking page pageid / number of linked pages)
def computeContributions(pageIDLinksPageRank):
    pagesContributions = []
    currentPageRank = pageIDLinksPageRank[1][1]
    linkedPages = pageIDLinksPageRank[1][0]
    numLinkedPages = len(linkedPages)
    contribution = currentPageRank/numLinkedPages

    for pageidLinkedPage in linkedPages:
        pagesContributions.append( (pageidLinkedPage, contribution))

    return pagesContributions
```

partitionBy: Example

```
# Run 30 iterations of PageRank
for x in range(30):
   # Retrieve for each page its current pagerank and
   # the list of linked pages by using the join transformation
   pageRankLinks = links.join(ranks)

   # Compute contributions from linking pages to linked pages
   # for this iteration
   contributions = pageRankLinks.flatMap(computeContributions)

   # Update current pagerank of all pages for this iteration
   ranks = contributions\
            .reduceByKey(lambda contrib1, contrib2: contrib1+contrib2)

# Print the result
 ranks.collect()
```

partitionBy: Example

```
# Run 30 iterations of PageRank
for x in range(30):
   # Retrieve for each page its current pagerank and
   # the list of linked pages by using the join transformation
   pageRankLinks = links.join(ranks)
```

The join transformation is invoked many times on the links Pair RDD.
The content of links is constant (it does not change during the loop interations.
Hence, caching it and also partitioning its content by key is useful.
- Its content is computed one time and cached in the main memory of the executors
- Its is shuffled and sent on the network only one time because we applied partitionBy on it.

```
# Print the result
 ranks.collect()
```

Default partitioning behavior of the main transformations

Transformation	Number of partitions	Partitioner
sc.parallelize(…)	sc.defaultParallelism	NONE
sc.textFile(…)	sc.defaultParallelism or number of file blocks , whichever is greater	NONE
filter(),map(),flatMap(), distinct()	same as parent RDD	NONE **except filter** preserve parent RDD's partitioner
rdd.union(otherRDD)	rdd.partitions.size + otherRDD. partitions.size	
rdd.intersection(otherRDD)	max(rdd.partitions.size, otherRDD. partitions.size)	
rdd.subtract(otherRDD)	rdd.partitions.size	
rdd.cartesian(otherRDD)	rdd.partitions.size * otherRDD. partitions.size	

Default partitioning behavior of the main transformations

Transformation	Number of partitions	Partitioner
reduceByKey(),foldByKey(), combineByKey(), groupByKey()	same as parent RDD	HashPartitioner
sortByKey()	same as parent RDD	RangePartitioner
mapValues(), flatMapValues()	same as parent RDD	parent RDD's partitioner
cogroup(), join(), ,leftOuterJoin(), rightOuterJoin()	depends upon input properties of two involved RDDs	HashPartitioner

Broadcast join

Broadcast join

- The join transformation is expensive in terms of execution time and amount of data sent on the network
- If one of the two input RDDs of key-value pairs is small enough to be stored in the main memory when can use a more efficient solution based on a broadcast variable
  - Broadcast hash join (or map-side join)
  - The smaller the small RDD, the higher the speed up

Broadcast join: Example

- Create a large RDD from a textual file containing a list of pairs (userID, post)
  - Each user can be associated to several posts
- Create a small RDD from a textual file containing a list of pairs (userID, (name, surname, age))
  - Each user can be associated to one single line in this second file
- Compute the join between these two files

Broadcast join: Example

```
# Read the first input file
largeRDD = sc.textFile("post.txt")
.map(lambda line: (int(line.split(',')[0]), line.split(',')[1]) )

# Read the second input file
smallRDD = sc.textFile("profiles.txt")
.map(lambda line: (int(line.split(',')[0]), line.split(',')[1]) )

# Broadcast join version
# Store the "small" RDD in a local python variable in the driver
# and broadcast it
localSmallTable = smallRDD.collectAsMap()
localSmallTableBroadcast = sc.broadcast(localSmallTable)
```

Broadcast join: Example

```
# Function for joining a record of the large RDD with the matching
#  record of the small one
def joinRecords(largeTableRecord):
   returnedRecords = []
   key = largeTableRecord[0]
   valueLargeRecord = largeTableRecord[1]

   if key in localSmallTableBroadcast.value:
       returnedRecords.append( (key, (valueLargeRecord,\
       localSmallTableBroadcast.value[key]) ) )

   return returnedRecords

# Execute the broadcast join operation by using a flatMap
# transformation on the "large" RDD
userPostProfileRDDBroadcatJoin = largeRDD.flatMap(joinRecords)
```