Table of contents

1 Persistence and Cache

Spark computes the content of an RDD each time an action is invoked on it. If the same RDD is used multiple times in an application, Spark recomputes its content every time an action is invoked on the RDD, or on one of its descendants, but this is expensive, especially for iterative applications.

So, it is possible to ask Spark to persist/cache RDDs: in this way, each node stores the content of its partitions in memory and reuses them in other actions on that RDD/dataset (or RDDs derived from it).

- The first time the content of a persistent/cached RDD is computed in an action, it will be kept in the main memory of the nodes;
- The next actions on the same RDD will read its content from memory (i.e., Spark persists/caches the content of the RDD across operations). This allows future actions to be much faster, often by more than ten times faster.

Spark supports several storage levels, which are used to specify if the content of the RDD is stored

- In the main memory of the nodes
- On the local disks of the nodes
- Partially in the main memory and partially on disk

Table 1: 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.
`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`, but store the data in off-heap memory. This requires off-heap memory to be enabled.

See here for more details.

It is possible to mark an RDD to be persisted by using the `persist(storageLevel)` method of the `RDD` class. The parameter of persist can assume the following values

- `pyspark.StorageLevel.MEMORY_ONLY`
- `pyspark.StorageLevel.MEMORY_AND_DISK`
- `pyspark.StorageLevel.DISK_ONLY`
- `pyspark.StorageLevel.NONE`
- `pyspark.StorageLevel.OFF_HEAP`
- `pyspark.StorageLevel.MEMORY_ONLY_2`
- `pyspark.StorageLevel.MEMORY_AND_DISK_2`

The storage level `*_2` replicate each partition on two cluster nodes, so that, ff one node fails, the other one can be used to perform the actions on the RDD without recomputing the content of the RDD.

It is possible to cache an RDD by using the `cache()` method of the `RDD` class: it corresponds to persist the RDD with the storage level `'MEMORY_ONLY'` (i.e., it is equivalent to `inRDD.persist(pyspark.StorageLevel.MEMORY_O`

Important

Notice that both persist and cache return a new RDD, since RDDs are immutable.

The use of the persist/cache mechanism on an RDD provides an advantage if the same RDD is used multiple times (i.e., multiples actions are applied on it or on its descendants).

The storage levels that store RDDs on disk are useful if and only if

- the size of the RDD is significantly smaller than the size of the input dataset;
- the functions that are used to compute the content of the RDD are expensive.

Otherwise, recomputing a partition may be as fast as reading it from disk.

1.1 Remove data from cache

Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion. It is also possible to manually remove an RDD from the cache by using the `unpersist()` method of the `RDD` class.

Example

1. Create an RDD from a textual file containing a list of words (one word for each line);
2. Print on the standard output
   - The number of lines of the input file
   - The number of distinct words

```python
# Read the content of a textual file
# and cache the associated RDD
inputRDD = sc.textFile("words.txt").cache()

print("Number of words: ",inputRDD.count())
print("Number of distinct words: ", inputRDD.distinct().count())
```

`.cache()`	The cache method is invoked, hence `inputRDD` is a cached RDD
`inputRDD.count()`	This is the first time an action is invoked on the `inputRDD` RDD. The content of the RDD is computed by reading the lines of the words.txt file and the result of the count action is returned. The content of `inputRDD` is also stored in the main memory of the nodes of the cluster.
`inputRDD.distinct().count()`	The content of `inputRDD` is in the main memory if the nodes of the cluster. Hence the computation of `distinct()` and `count()` is performed by reading the data from the main memory and not from the input (HDFS) file `words.txt`.

2 Accumulators

When a function passed to a Spark operation is executed on a remote cluster node, it works on separate copies of all the variables used in the function. These variables are copied to each node of the cluster, and no updates to the variables on the nodes are propagated back to the driver program.

Spark provides a type of shared variables called **accumulators**: accumulators are shared variables that are only "added" to through an associative operation and can therefore be efficiently supported in parallel, and they can be used to implement counters or sums.

Accumulators are usually used to compute simple statistics while performing some other actions on the input RDD, avoiding to use actions like `reduce()` to compute simple statistics (e.g., count the number of lines with some characteristics).

2.1 How to use accumulators

1. The driver defines and initializes the accumulator
2. The code executed in the worker nodes increases the value of the accumulator (i.e., the code in the functions associated with the transformations)
3. The final value of the accumulator is returned to the driver node
    - Only the driver node can access the final value of the accumulator
    - The worker nodes cannot access the value of the accumulator: they can only add values to it

! Important

Pay attention that the value of the accumulator is increased in the functions associated with transformations, and, since transformations are lazily evaluated, the value of the accumulator is computed only when an action is executed on the RDD on which the transformations increasing the accumulator are applied.

Spark natively supports numerical accumulators (integers and floats), but programmers can add support for new data types: accumulators are `pyspark.accumulators.Accumulator` objects.

Accumulators are defined and initialized by using the `accumulator(value)` method of the `SparkContext` class: the value of an accumulator can be increased by using the `add(value)` method of the `Accumulator` class, that adds `value` to the current value of the accumulator. The final value of an accumulator can be retrieved in the driver program by using `value` of the `Accumulator` class.

i Example

1. Create an RDD from a textual file containing a list of email addresses (one email for each line);
2. Select the lines containing a valid email and store them in an HDFS file (in this example, an email is considered a valid email if it contains the @ symbol);
3. Print also, on the standard output, the number of invalid emails.

```python
# Define an accumulator. Initialize it to 0
invalidEmails = sc.accumulator(0)

# Read the content of the input textual file
emailsRDD = sc.textFile("emails.txt")

#Define the filtering function
def validEmailFunc(line):
    if (line.find('@')<0):
        invalidEmails.add(1)
        return False
    else:
        return True

# Select only valid emails
# Count also the number of invalid emails
validEmailsRDD = emailsRDD.filter(validEmailFunc)

# Store valid emails in the output file
validEmailsRDD.saveAsTextFile(outputPath)

# Print the number of invalid emails
print("Invalid email addresses: ", invalidEmails.value)
```

`invalidEmails = sc.accumulator(0)`	Definition of an accumulator of type integer.
`invalidEmails.add(1)`	This function increments the value of the `invalidEmails` accumulator if the email is invalid.
`invalidEmails.value`	Read the final value of the accumulator. Pay attention that the value of the accumulator is correct only because an action (`saveAsTextFile`) has been executed on the `validEmailsRDD` and its content has been computed (the function `validEmailFunc` has been executed on each element of `emailsRDD`)

2.2 Personalized accumulators

Programmers can define accumulators based on new data types (different from integers and floats): to define a new accumulator data type of type T, the programmer must define a class subclassing the `AccumulatorParam` interface. The `AccumulatorParam` interface has two methods

- `zero` for providing a zero value for your data type
- `addInPlace` for adding two values together

3 Broadcast variables

Spark supports broadcast variables. A broadcast variable is a read-only (small/medium) shared variable

- It is instantiated in the driver: the broadcast variable is stored in the main memory of the driver in a local variable;
- It is sent to all worker nodes that use it in one or more Spark operations: the broadcast variable is also stored in the main memory of the executors (which are instantiated in the used worker nodes).

A copy each broadcast variable is sent to all executors that are used to run a task executing a Spark operation based on that variable (i.e., the variable is sent `num.executors` times). A broadcast variable is sent only one time to each executor that uses that variable in at least one Spark operation (i.e., in at least one of its tasks). Each executor can run multiples tasks associated with the same broadcast variable, but the broadcast variable is sent only one time for each executor, hence the amount of data sent on the network is limited by using broadcast variables instead of standard variables.

Broadcast variables are usually used to share (small/medium) lookup-tables, and, since they are stored in local variables, they must the small enough to be stored in the main memory of the driver and also in the main memory of the executors.

Broadcast variables are objects of type `Broadcast`: a broadcast variable (of type T) is defined in the driver by using the `broadcast(value)` method of the `SparkContext` class. The value of a broadcast variable (of type T) is retrieved (usually in transformations) by using `value` of the `Broadcast` class.

Example

1. Create an RDD from a textual file containing a dictionary of pairs (word, integer value), one pair for each line. Suppose the content of this first file is large but can be stored in main-memory;
2. Create an RDD from a textual file containing a set of words, in particular a sentence (set of words) for each line; Transform the content of the second file mapping each word to an integer based on the dictionary contained in the first file; then, store the result in an HDFS file.

- First file (dictionary)

```
java 1
spark 2
test 3
```

- Second file (the text to transform)

```
java spark
spark test java
```

- Output file

```
12
231
```

```python
# Read the content of the dictionary from the first file and
# map each line to a pair (word, integer value)
dictionaryRDD = sc \
    .textFile("dictionary.txt") \
    .map(lambda line: (
        line.split(" ")[0],
        line.split(" ")[1]
    ))

# Create a broadcast variable based on the content of dictionaryRDD.
# Pay attention that a broadcast variable can be instantiated only
# by passing as parameter a local variable and not an RDD.
# Hence, the collectAsMap method is used to retrieve the content of the
# RDD and store it in the dictionary variable
dictionary = dictionaryRDD.collectAsMap()

# Broadcast dictionary
dictionaryBroadcast = sc.broadcast(dictionary)

# Read the content of the second file
textRDD = sc.textFile("document.txt")

# Define the function that is used to map strings to integers
def myMapFunc(line):
    transformedLine=''
    for word in line.split(' '):
        intValue = dictionaryBroadcast.value[word]
        transformedLine = transformedLine+intValue+' '
    return transformedLine.strip()

# Map words in textRDD to the corresponding integers and concatenate
# them
mappedTextRDD= textRDD.map(myMapFunc)

# Store the result in an HDFS file
mappedTextRDD.saveAsTextFile(outputPath)
```

`sc.broadcast(dictionary)`	Define a broadcast variable.
`dictionaryBroadcast.value[word]`	Retrieve the content of the broadcast variable and use it.

4 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 (i.e., one partition with many data, many partitions with few data). The worker node that processes the largest partition needs more time than the other workers, becoming the bottleneck of your application.

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

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, preserving the number of partitions of the input RDD (i.e., the returned RDD has the same number of partitions of the input RDD).

Transformation	Effect
`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; notice that 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; notice that elements are not assigned to partitions based on their value.
`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`, and each partition contains a part of one input blocks.

Transformation	Effect
`repartition(numPartitions)`	`numPartitions` can be greater or smaller than the number of partitions of the input RDD, and 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; notice that 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.
`coalesce(numPartitions)`	`numPartitions` is smaller than the number of partitions of the input RDD, and 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; notice that 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, so coalesce is more efficient than repartition to reduce the number of partitions.

Spark allows specifying how to partition the content of RDDs of key-value pairs: he input tpairs 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).

4.1 `partitionBy(numPartitions)`

Partitioning is based on the `partitionBy()` transformation. 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.

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

$$n = portable\_hash(key)\%numPart$$

Notice that $portable\_hash(key)$ returns and integer.

4.1.1 `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.

Suppose that the number of partition of the returned Pair RDD is **numPart**. The custom 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

$$n = partitionFunc(key)\%numPart$$

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.
For example, 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 the application instead of speeding it up.

Example

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

```python
# 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()

# 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)

# 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

# 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()...cache()` `links`	Notice that the returned Pair RDD is partitioned and cached. 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 iterations). 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, and it is shuffled and sent on the network only one time because `partitionBy` was applied on it

4.2 Default partitioning behavior of the main transformations

Transformation	Number of partitions	Partitioner
`sc.parallelize()`	`sc.defaultParallelism`	NONE
`sc.textFile()`	the maximum between `sc.defaultParallelism` and the number of file blocks	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`	NONE except filter preserve parent RDD's partitioner
`rdd.intersection(otherRDD)`	`max(rdd.partitions.size, otherRDD.partitions.size)`	NONE except filter preserve parent RDD's partitioner
`rdd.subtract(otherRDD)`	`rdd.partitions.size`	NONE except filter preserve parent RDD's partitioner
`rdd.cartesian(otherRDD)`	`rdd.partitions.size * otherRDD. partitions.size`	NONE except filter preserve parent RDD's 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

5 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, then it is possible to 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

Example

1. Create a large RDD from a textual file containing a list of pairs `(userID, post)`; notice that each user can be associated to several posts;
2. Create a small RDD from a textual file containing a list of pairs `(userID, (name, surname, age))`; notice that each user can be associated to one single line in this second file;
3. Compute the join between these two files.

```python
# 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)

# 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)
```