Big data for internet applications

Cache, Accumulators, Broadcast variables

Persistence and Cache

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”
- This is expensive, especially for iterative applications
- We can ask Spark to persist/cache RDDs

Persistence and Cache

- When you ask Spark to persist/cache an RDD, 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 10x

Persistence and Cache

- Spark supports several storage levels
  - The storage level is 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

Persistence 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.
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_ONL, but store the data in off-heap memory. This requires off-heap memory to be enabled.

http://spark.apache.org/docs/2.4.0/rdd-programming-guide.html#rdd-persistence

Persistence and Cache

- You can 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

Persistence and Cache

  - pyspark.StorageLevel.MEMORY_ONLY_2
  - pyspark.StorageLevel.MEMORY_AND_DISK_2
- The storage level *_2 replicate each partition on two cluster nodes
  - If one node fails, the other one can be used to perform the actions on the RDD without recomputing the content of the RDD

Persistence and Cache

- You can 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_ONLY)
- Note that both persist and cache **return** a **new RDD**
  - Because RDDs are immutable

Persistence and Cache

- 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

Persistence and Cache

- 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
  - Or 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

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
- You can manually remove an RDD from the cache by using the **unpersist()** method of the **RDD** class

Cache: Example

- Create an RDD from a textual file containing a list of words
  - One word for each line
- Print on the standard output
  - The number of lines of the input file
  - The number of distinct words

Cache: Example

```
# 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: Example

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

print("Number of word
print("Number of distir
```

The cache method is invoked.
Hence, inputRDD is a “cached” RDD

Cache: Example

```
# 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())
```

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.

Cache: Example

```
# 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())
```

The content of inputRDD is in the main memory if the nodes of the cluster.
Hence the computation of distinct() + count() is performed by reading the data from the main memory and not from the input (HDFS) file words.txt

Accumulators

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

Accumulators

- 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
- They can be used to implement counters or sums

Accumulators

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

Accumulators

- The driver defines and initializes the accumulator
- The code executed in the worker nodes increments the local value of the accumulator
  - i.e., the code in the "functions" associated with the transformations
- The local increments of the accumulator are sent to the driver and the final value of the accumulator is computed in 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

Accumulators

- Pay attention that the value of the accumulator is incremented in the functions associated with transformations
- 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

Accumulators

- Spark natively supports numerical accumulators
  - Integers and floats
- But programmers can add support for new data types
- Accumulators are **pyspark.accumulators.Accumulator** objects

Accumulators

- 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
  - Add "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

Accumulators: Example

- Create an RDD from a textual file containing a list of email addresses
  - One email for each line
- 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
- Print also, on the standard output, the number of invalid emails

Accumulators: Example

```
# 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)
```

Accumulators: Example

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

# Read the content of the input textual file
emailsRD
```

Definition of an accumulator of type integer

```
#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)
```

Accumulators: Example

```
# 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
```

This function increments the value of the invalidEmails accumulator if the email is invalid

```
validEmailsRDD = emailsRDD.filter(validEmailFunc)
```

Accumulators: Example

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

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

Accumulators: Example

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

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

Read the final value of the accumulator

Accumulators: Example

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

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

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)

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

Broadcast variables

Broadcast variables

- Spark supports also broadcast variables
- A broadcast variable is a read-only (medium/large) shared variable
  - That is instantiated in the driver
  - And it is sent to all worker nodes that use it in one or more Spark actions

Broadcast variables

- A copy of each “standard” variable is sent to all the tasks executing a Spark action using that variable
  - i.e., the variable is sent “num. tasks” times
- A broadcast variable is sent only one time to each executor using it in at least one Spark action (i.e., in at least one of its tasks)
  - Each executor can run multiples tasks using that variable and the broadcast variable is sent only one time
  - Hence, the amount of data sent on the network is limited by using broadcast variables instead of “standard” variables

Broadcast variables

- Broadcast variables are usually used to share (large) lookup-tables

Broadcast variables

- 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

Broadcast variables: Example

- 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**
- Create an RDD from a textual file containing a set of words
  - 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
  - Store the result in an HDFS file

Broadcast variables: Example

- First file (dictionary)

  java 1

  spark 2

  test 3

- Second file (the text to transform)

  java spark

  spark test java

- Output file

  1 2

  2 3 1

Broadcast variables: Example

```
# 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)
```

Broadcast variables: Example

```
# 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)
```

Define a broadcast variable

Broadcast variables: Example

```
# 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)
```

Broadcast variables: Example

```
# 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)
```

Retrieve the content of the broadcast variable and use it

Broadcast variables: Example

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