Big data: architectures and data analytics

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

4

Persistence and Cache

- When you ask Spark to persist/cache an RDD, each node stores the content of any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it)
  - The first time the content of a persistent/cached RDD is computed in an action, it will be kept in memory on 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

5

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

6

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.
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.

http://spark.apache.org/docs/1.5.0/programming-guide.html#rdd-persistence 7

Persistence and Cache: Storage levels

Storage Level	Meaning
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.

http://spark.apache.org/docs/1.5.0/programming-guide.html#rdd-persistence 8

Persistence and Cache: Storage levels

Storage Level	Meaning
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon. Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory. If you plan to use Tachyon as the off heap store, Spark is compatible with Tachyon out-of-the-box. Please refer to this page for the suggested version pairings.

http://spark.apache.org/docs/1.5.0/programming-guide.html#rdd-persistence 9

Persistence and Cache

- You can mark an RDD to be persisted by using the **JavaRDD<T> persist(StorageLevel level)** method of the **JavaRDD<T>** class
- The parameter of persist can assume the following values
  - StorageLevel.MEMORY_ONLY()
  - StorageLevel.MEMORY_AND_DISK()
  - StorageLevel.MEMORY_ONLY_SER()
  - StorageLevel.MEMORY_AND_DISK_SER()
  - StorageLevel.DISK_ONLY()
  - StorageLevel.NONE()
  - StorageLevel.OFF_HEAP()

10

Persistence and Cache

- StorageLevel.MEMORY_ONLY_2()
- StorageLevel.MEMORY_AND_DISK_2()
- StorageLevel.MEMORY_ONLY_SER_2()
- StorageLevel.MEMORY_AND_DISK_SER_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

11

Persistence and Cache

- You can cache an RDD by using the **JavaRDD<T> cache()** method of the **JavaRDD<T>** class
  - It corresponds to persist the RDD with the storage level `MEMORY_ONLY`
  - i.e., it is equivalent to inRDD.persist(StorageLevel.MEMORY_ONLY())
- Note that both persist and cache return a new JavaRDD
  - Because RDDs are immutable

12

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

13

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

14

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 **JavaRDD<T> unpersist()** method of the **JavaRDD<T>** class

15

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

16

Cache: Example

```
// Read the content of a textual file
// and cache the associated RDD
JavaRDD<String> inputRDD = sc.textFile("words.txt").cache();

System.out.println("Number of words: "+inputRDD.count());
System.out.println("Number of distinct words: "
                                    +inRDD.distinct().count());
```

17

Cache: Example

```
// Read the content of a textual file
// and cache the associated RDD
JavaRDD<String> inputRDD = sc.textFile("words.txt").cache();

System.out.println("Number of words: "+inputRDD.count());
System.out.println("Number of dist
```

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

18

Cache: Example

```
// Read the content of a textual file
// and cache the associated RDD
JavaRDD<String> inputRDD = sc.textFile("words.txt").cache();

System.out.println("Number of words: "+inputRDD.count());
System.out.println("Number of distinct words: "
                                    +inRDD.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.

19

Cache: Example

```
// Read the content of a textual file
// and cache the associated RDD
JavaRDD<String> inputRDD = sc.textFile("words.txt").cache();

System.out.println("Number of words: "+inputRDD.count());
System.out.println("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() is performed by reading the data from the main memory and not from the input (HDFS) file words.txt

20

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

22

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 (as in MapReduce) or sums

23

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)

24

Accumulators

- The driver defines and initializes the accumulator
- The code executed in the worker nodes increases the value of the accumulator
  - I.e., the code in the "functions" associated with the transformations
- 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

25

Accumulators

- Pay attention that the value of the accumulator is increased in the call method of 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

26

Accumulators

- Spark natively supports accumulators of numeric types
- But programmers can add support for new data types

27

Accumulators

- Accumulators are objects of type **org.apache.spark.Accumulator**
  - An Integer accumulator can be defined and initialized in the driver by using the **Accumulator<Integer> accumulator(int value)** method of the **JavaSparkContext** class
    - **value** is the initial value of the accumulator
  - A Double accumulator can be defined and initialized in the driver by using the **Accumulator<Double> accumulator(double value)** method of the **JavaSparkContext** class
    - **value** is the initial value of the accumulator

28

Accumulators

- The value of an accumulator can be "increased" by using the **void add(T value) method** of the **Accumulator<T>** 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 the **T value() method** of the **Accumulator<T>** class

29

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 as valid if it contains the @ symbol
- Print also, on the standard output, the number of invalid emails

30

Accumulators: Example

```
// Read the content of the input textual file
// This class is used to filter the input emails
class ValidEmail implements Function<String, Boolean> {
      // Call method. It returns true if the input String
      // contains the symbol @
      public Boolean call(String line) {
            // If the line contains an invalid email increase
            // the accumulator invalidEmails
            if (line.contains("@")==false) {
                  invalidEmails.add(1);
            }

            return line.contains("@");
      }
}
```


Accumulators: Example

```
// Read the content of the input textual file
// This class is used to filter the input emails
class ValidEmail implements Function<String, Boolean> {
      // Call method. It returns true if the input String
      // contains the symbol @
      public Boolean call(String line) {
            // If the line contains an invalid email increase
            // the accumulator invalidEmails
            if (line.contains("@")==false) {
                  invalidEmails.add(1);
            }

            return line.contains("@");
      }
}
```

The call method increases also the value of invalidEmails that is an accumulator

Accumulators: Example

```
....
// Define an accumulator and initialize it to o
final Accumulator<Integer> invalidEmails=sc.accumulator(0);

// Read the content of the input textual file
JavaRDD<String> emailsRDD = sc.textFile("emails.txt");

// Select only valid emails
JavaRDD<String> validEmailsRDD = emailsRDD.filter(new ValidEmail());

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

// Print the number of invalid emails
System.out.println("Invalid emails: "+invalidEmails.value());
```


Accumulators: Example

```
....
// Define an accumulator and initialize it to o
final Accumulator<Integer> invalidEmails=sc.accumulator(0);

// Read t
JavaRDD<String> emailsRDD = sc.textFile("emails.txt");

// Select only valid emails
JavaRDD<String> validEmailsRDD = emailsRDD.filter(new ValidEmail());

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

// Print the number of invalid emails
System.out.println("Invalid emails: "+invalidEmails.value());
```

Definition of an accumulator of type Integer

Accumulators: Example

```
....
// Define an accumulator and initialize it to o
final Accumulator<Integer> invalidEmails=sc.accumulator(0);

// Read the content of the input textual file
JavaRDD<String> emailsRDD = sc.textFile("emails.txt");

// Select only valid emails
JavaRDD<String> validEmailsRDD = emailsRDD.filter(new ValidEmail());

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

// Print the number of invalid emails
System.out.println("Invalid emails: "+invalidEmails.value());
```

Read the final value of the accumulator

Accumulators: Example

```
....
// Define an accumula
final Accumulator<In

// Read the content o
JavaRDD<String> emailsRDD = sc.textFile("emails.txt");

// Select only valid emails
JavaRDD<String> validEmailsRDD = emailsRDD.filter(new ValidEmail());

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

// Print the number of invalid emails
System.out.println("Invalid emails: "+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 (and hence the call method of the ValidEmail class has been executed on each element of emailsRDD)

Personalized accumulators

- Programmers can define accumulators based on new data types (different from Integer and Double)
- To define a new accumulator data type of type T, the programmer must define a class implementing the org.apache.spark.AccumulatorParam<T> interface
  - The following methods must be implemented
    - public T zero(T initialValue)
      - Return the "zero" (identity) value for an accumulator type
    - public T addInPlace(T v1, T v2)
      - Merge two accumulated values together
    - public T addAccumulator(T v1, T v2)
      - Merge two accumulated values together

37

Personalized accumulators

- Then, a new accumulator of type T can be instantiated by using the Accumulator<T> accumulator(T initialValue, AccumulatorParam<T> param) method of the JavaSparkContext class
  - initialValue is the initial value of the accumulator
  - param is the class specifying how the value of accumulators of type T can be incremented

38

Broadcast variables

Broadcast variables

- Spark supports also broadcast variables
- A broadcast variable is a read-only (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

40

Broadcast variables

- A copy 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

41

Broadcast variables

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

42

Broadcast variables

- Broadcast variables are objects of type **Broadcast<T>**
- A broadcast variable of type T is defined in the driver by using the **Broadcast<T> broadcast(T value)** method of the **JavaSparkContext** class
- The value of a broadcast variable of type T is retrieved (usually in transformations) by using the **T value()** method of the **Broadcast<T>** class

43

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 file 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

44

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

45

Broadcast variables: Example

```
// Read the content of the input textual file
class MapToPair implements PairFunction<String, String, Integer> {
        public Tuple2<String, Integer> call(String line) {
                String[] fields = line.split(" ");

                String word=fields[0];
                Integer intWord=Integer.parseInt(fields[1]);

                return new Tuple2<String, Integer>(word, intWord) ;
        }
}

// Read the content of the dictionary from the first file
// Map each line to a pair (word, integer value)
JavaPairRDD<String, Integer> dictionaryRDD =
                sc.textFile(inputDictionary).mapToPair(new MapToPair());
```

46

Broadcast variables: Example

```
// Create a local HashMap object that will be used to store the
// mapping word -> integer
HashMap<String, Integer> dictionary=new HashMap<String, Integer>();

// 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 java variable and not an RDD
// Hence, the collect method is used to retrieve the content of the
// RDD and store it in the dictionary HashMap<String, Integer> variable
for (Tuple2<String, Integer> pair: dictionaryRDD.collect()) {
        dictionary.put(pair._1(), pair._2());
}

final Broadcast<HashMap<String, Integer>> dictionaryBroadcast =
                                                sc.broadcast(dictionary);
```

47

Broadcast variables: Example

```
// Create a local HashMap object that will be used to store the
// mapping word -> integer
HashMap<String, Integer> dictionary=new HashMap<String, Integer>();

// 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 java variable and not an RDD
// Hence, the collect method is used to retrieve the content of the
// RDD and store it in the dictionary HashMap<String, Integer> variable
for (Tuple2<String, Integer> pair: dictionaryRDD.collect()) {
        dictionary.put(pair._1(), pair._2());
}

final Broadcast<HashMap<String, Integer>> dictionaryBroadcast =
                                                sc.broadcast(dictionary);
```

Define a broadcast variable

48

Broadcast variables: Example

```
class MapToIntegers implements Function<String, String> {
        public String call(String line) {
                String[] words=line.split(" ");
                String transformedLine=new String("");
                Integer intValue;

                // map each word to the corresponding integer
                for (int i=0; i<words.length-1; i++) {
                        intValue=dictionaryBroadcast.value().get(words[i]);
                        transformedLine=transformedLine.concat(intValue+" ");
                }

                intValue=dictionaryBroadcast.value().get(words[words.length-1]);
                transformedLine=transformedLine.concat(intValue.toString());

                return transformedLine;
        }
}
```

49

Broadcast variables: Example

```
class MapToIntegers implements Function<String, String> {
        public String call(String line) {
                String[] words=line.split(" ");
                String transformedLine=new String("");
                Integer intValue;

                // map each word to the corresponding integer
                for (int i=0; i<words.length-1; i++) {
                        intValue=dictionaryBroadcast.value().get(words[i]);
                        transformedLine=transformedLine.concat(intValue+" ");
                }

                intValue=dictionaryBroadcast.value().get(words[words.length-1]);
                transformedLine=transformedLine.concat(intValue.toString());

                return transformedLine;
        }
}
```

Retrieve the content of the broadcast variable and use it

50

Broadcast variables: Example

```
// Read the content of the second file
JavaRDD<String> textRDD = sc.textFile(inputText);

// Map each word in textRDD to the corresponding integer
// Each input element is a string. Also the output elements are strings
JavaRDD<String> mappedTextRDD=
                        textRDD.map(new MapToIntegers());

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

51