Big data: architectures and data analytics

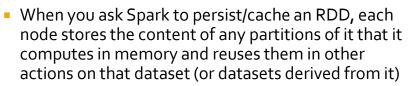
Cache, Accumulators, Broadcast variables

Persistence and Cache

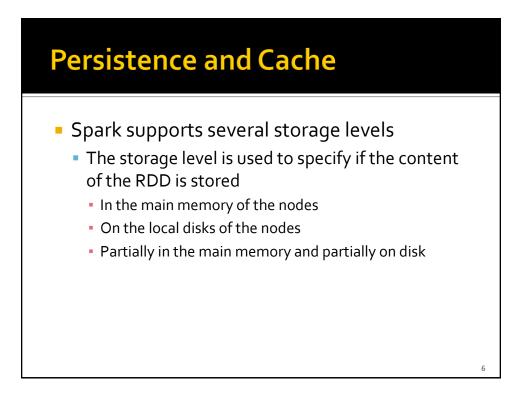
Persistence and Cache

- 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

Persistence and Cache



- 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



Persistence and Cache: Storage levels

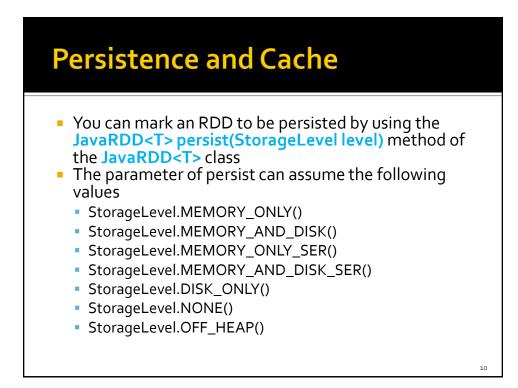
Storage Level	Meaning	
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.	
MEMORY_AND_DISK	Store RDD as deserialized Java objects in 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/doc	s/1.5.0/programming-guide.html#rdd-persistence	

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

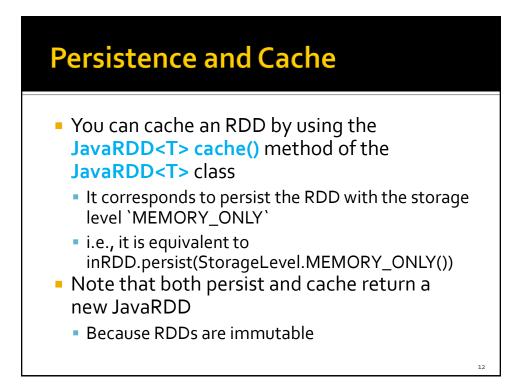
Persistence and Cache: Storage levels

ar er ar th m di re	IEMORY_ONLY_SER, OFF_HEAP reduces garbage ollection overhead and allows executors to be smaller nd to share a pool of memory, making it attractive in nvironments with large heaps or multiple concurrent pplications. Furthermore, as the RDDs reside in Tachyon, he crash of an executor does not lead to losing the in- hemory cache. In this mode, the memory in Tachyon is iscardable. Thus, Tachyon does not attempt to econstruct a block that it evicts from memory. If you plan b use Tachyon as the off heap store, Spark is compatible with Tachyon out-of-the-box. Please refer to this page for
---------------------------------------	--



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



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

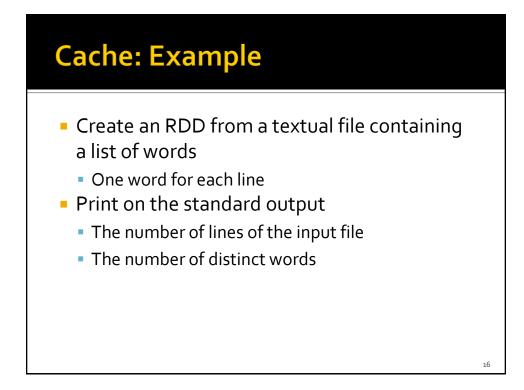


- The storage levels that store RDDs on disk are useful if and only if
 - The "size" of the RDD is significantly smaller than the size of the input dataset
 - Or the functions that are used to compute the content of the RDD are expensive
 - Otherwise, recomputing a partition may be as fast as reading 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

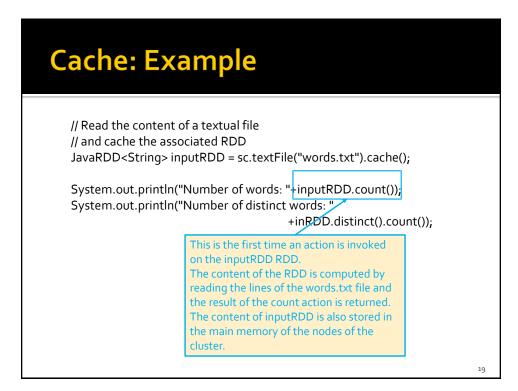


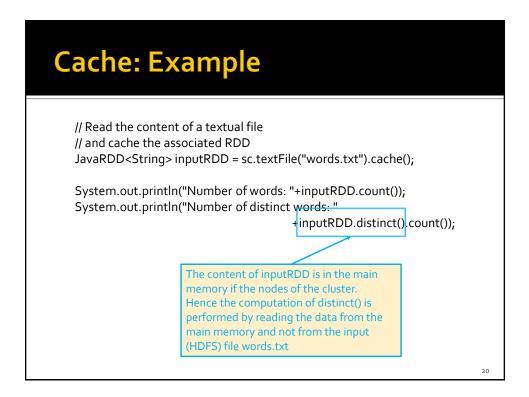
Cache: Example

// 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());

<section-header><code-block><code-block><code-block><code-block></code></code></code></code>





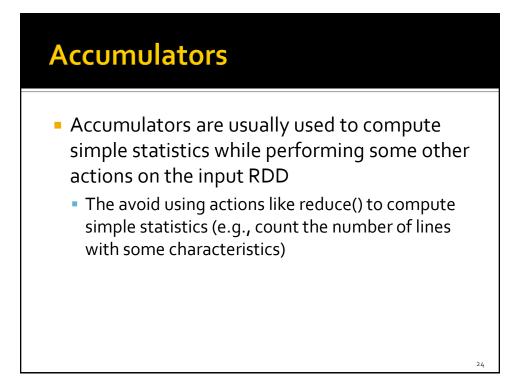


Accumulators

- 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

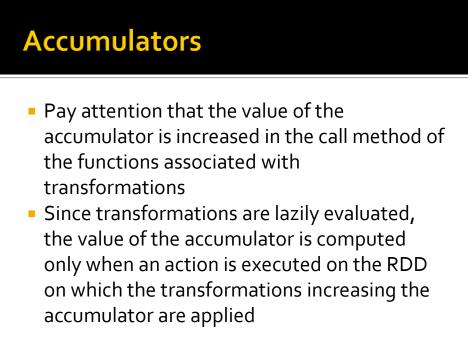
Accumulators

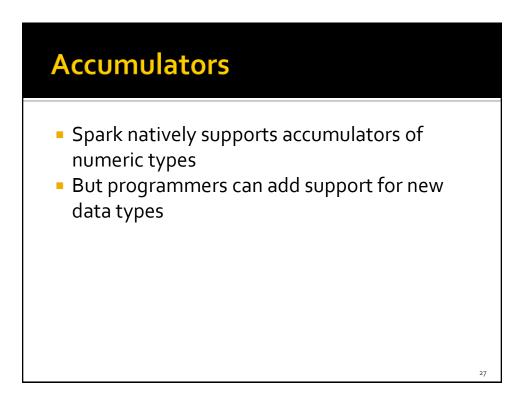
- Spark provides a type of shared variables called accumulators
- Accumulators are shared variables that are only "added" to through an associative operation and can therefore be efficiently supported in parallel
- They can be used to implement counters (as in MapReduce) or sums

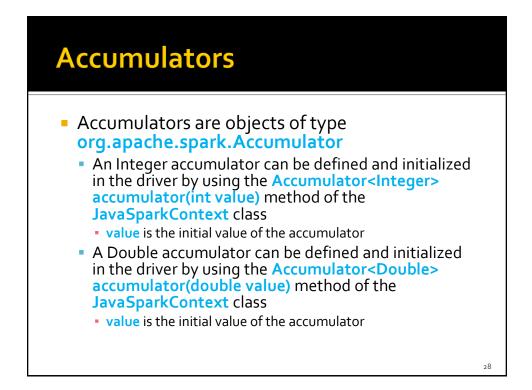


Accumulators

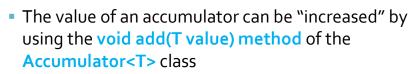
- The driver defines and initializes the accumulator
- The code executed in the worker nodes increases the value of the accumulator
 - I.e., the code in the "functions" associated with the transformations
- The final value of the accumulator is returned to the driver node
 - Only the driver node can access the final value of the accumulator
 - The worker nodes cannot access the value of the accumulator
 - They can only add values to it



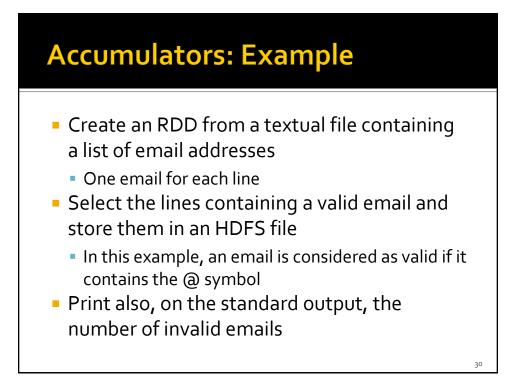




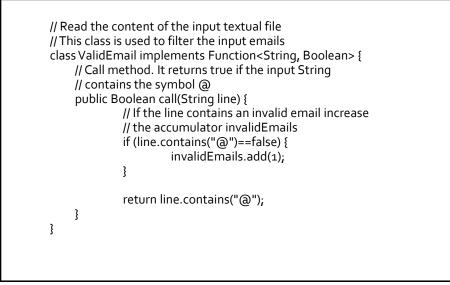
Accumulators

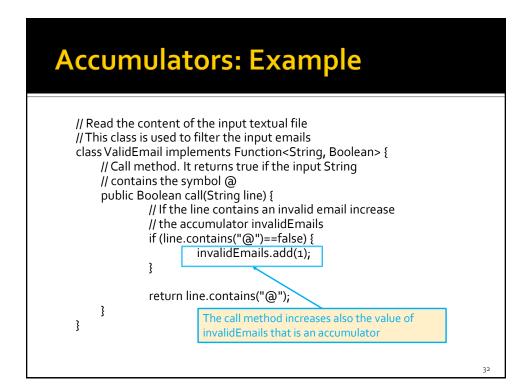


- Add "value" to the current value of the accumulator
- The final value of an accumulator can be retrieved in the driver program by using the T value() method of the Accumulator<T> class









Accumulators: Example

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

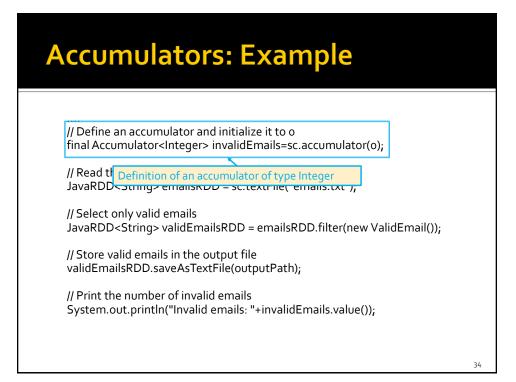
// Read the content of the input textual file
JavaRDD<String> emailsRDD = sc.textFile("emails.txt");

// 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

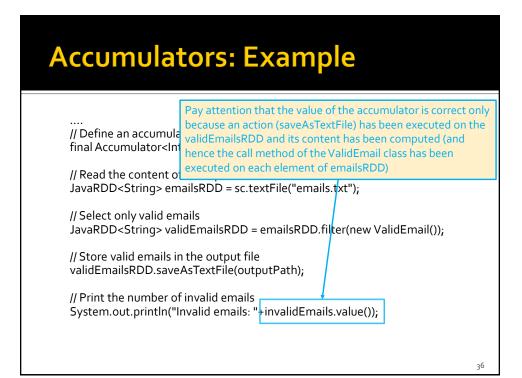
// Define an accumulator and initialize it to o final Accumulator<Integer> invalidEmails=sc.accumulator(o);

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

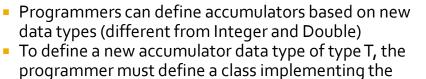
// Select only valid emails Read the final value of the accumulator JavaRDD<String> validEmailsRDD = emailsRDD.filter(new ValidEmail());

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

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



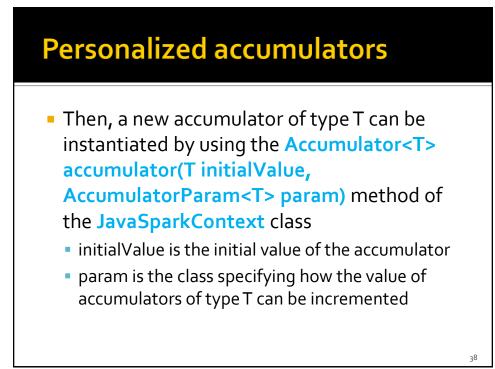
Personalized accumulators

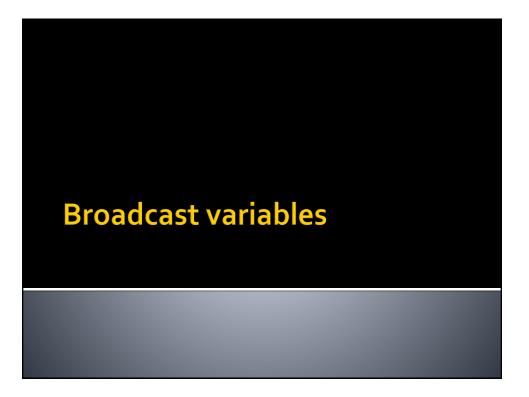


org.apache.spark.AccumulatorParam<T> interface

- 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





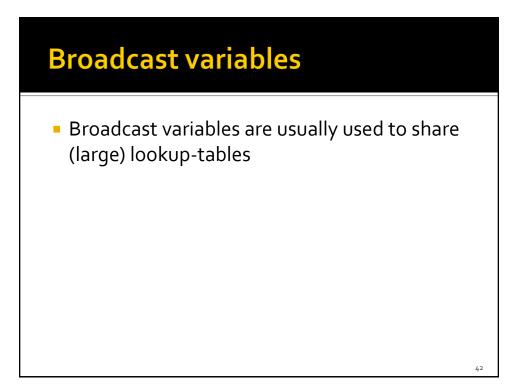




- Spark supports also broadcast variables
- A broadcast variable is a read-only (large) shared variable
 - That is instantiated in the driver
 - And it is sent to all worker nodes that use it in one or more Spark actions

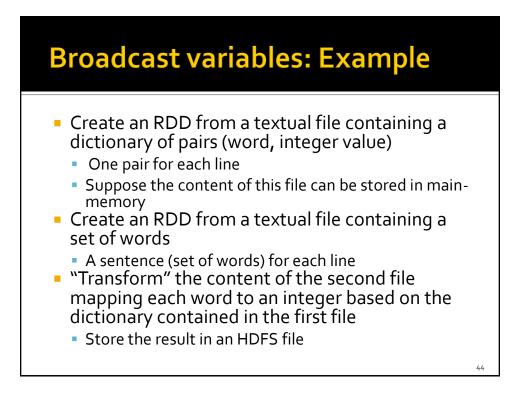
Broadcast variables

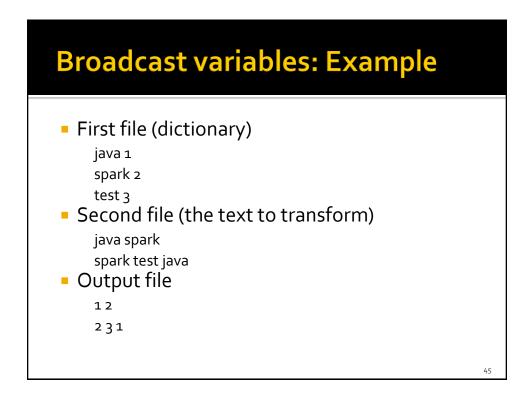
- A copy each "standard" variable is sent to all the tasks executing a Spark action using that variable
 - i.e., the variable is sent "num. tasks" times
- A broadcast variable is sent only one time to each executor using it in at least one Spark action (i.e., in at least one of its tasks)
 - Each executor 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

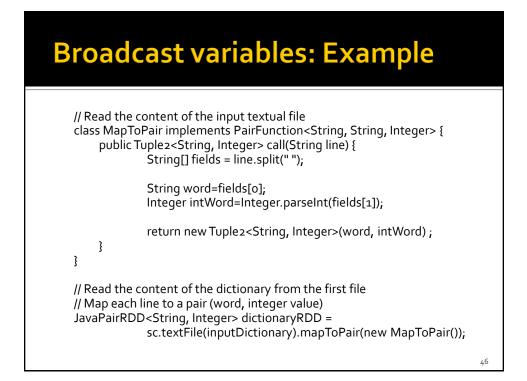


Broadcast variables

- 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







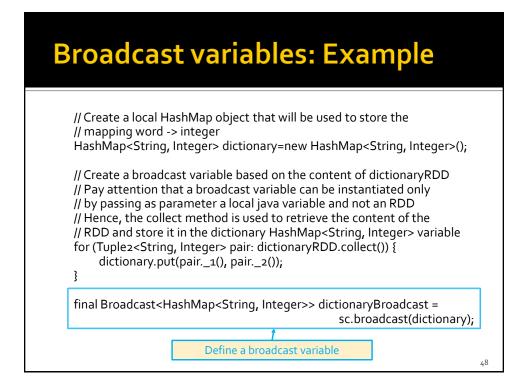
Broadcast variables: Example

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

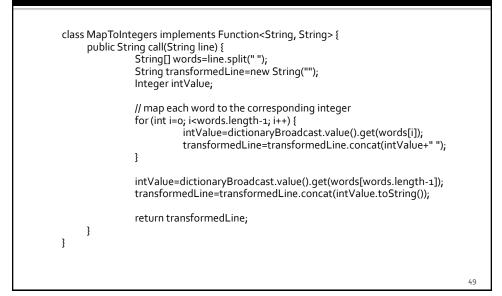
// Create a broadcast variable based on the content of dictionaryRDD // Pay attention that a broadcast variable can be instantiated only // by passing as parameter a local java variable and not an RDD // Hence, the collect method is used to retrieve the content of the // RDD and store it in the dictionary HashMap<String, Integer> variable for (Tuple2<String, Integer> pair: dictionaryRDD.collect()) { dictionary.put(pair._1(), pair._2());

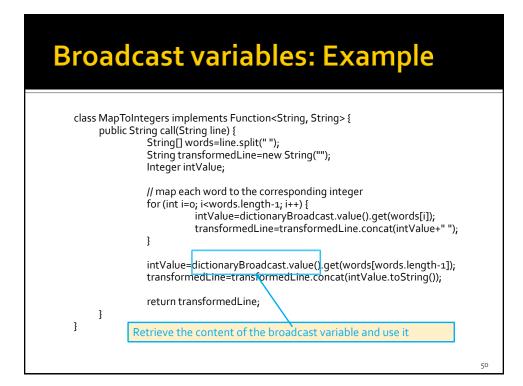
}

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



Broadcast variables: Example





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);