RDD Operations

Contents

1. RDD Transformations

- Mapping and filtering transformations
- Set-based transformations
- Combinatorial transformations
- Grouping and sorting transformations
- Additional transformations

- Demos for this section are in the following directory:
  - `transformations`

Mapping and Filtering Transformations (1 of 2)

- `map(func)`
  - Returns a new RDD, formed by passing each element of the source RDD through the supplied func

- `filter(func)`
  - Returns a new RDD, formed by selecting those elements of the source RDD where the supplied func returns true

Mapping and Filtering Transformations (2 of 2)

- `flatMap(func)`
  - Similar to `map`, but each input item can be mapped to 0 or more output items
  - So the func should return a sequence rather than a single item

- `mapPartitions(func)`
  - Similar to `map`, but runs separately on each partition (i.e. block) of the RDD
  - The function must be of type Iterator<T> => Iterator<U> when running on an RDD of type T

Set-Based Transformations

- `union(otherRDD)`
  - Returns a new RDD that contains the union of the elements in the source RDD and the argument
- `intersection(otherRDD)`
  - Returns a new RDD that contains the intersection of the elements in the source RDD and the argument
- `subtract(otherRDD)`
  - Returns a new RDD that contains the elements in the source RDD but not in the argument
- `distinct`
  - Returns a new RDD that contains the distinct elements in the source RDD

Combinatorial Transformations

- `cartesian(otherRDD)`
  - Returns a new RDD that contains the cartesian product of all the elements in both RDDs

- `zip(otherRDD)`
  - Returns an RDD of pairs
  - The 1st element of each pair comes from the source RDD
  - The 2nd element of each pair comes from the argument RDD

- `zipWithIndex`
  - Returns an RDD of pairs
  - The 1st element of each pair comes from the source RDD
  - The 2nd element of each pair is the index

Grouping and Sorting Transformations

- `groupBy(func)`
  - Returns an RDD of pairs
  - The 1st element of each pair is a key, generated by the func
  - The 2nd element of each pair is a collection of items with that key

- `keyBy(func)`
  - Returns an RDD of pairs, one pair for each item in the source RDD
  - The 1st element of each pair is a key, generated by the func
  - The 2nd element of each pair is the item with that key

- `sortBy(func, optionalAscFlag)`
  - Sorts elements in the source RDD, according to the func
  - The 2nd arg is an optional boolean, indicating whether to sort in ascending order (default is true)

Additional Transformations

- `randomSplit(weightingsArray)`
  - Splits the source RDD into an array of RDDs (randomly-populated)
  - The relative sizes of the array of RDDs is determined by the weightings-array argument
- `coalesce(number)`
  - Reduces number of partitions in an RDD, according to number arg
  - Useful for consolidating small RDDs, may improve performance
- `repartition(number)`
  - Returns an RDD with the specified number of partitions
  - Useful for increasing parallelism
- `sample(replacementStrategy, sizeRatio, seed)`
  - Returns a sampled subset of the source RDD

2. RDD Transformations on Key-Value Pairs

- Overview
- Working with keys and values
- Joins
- Key-based operations

- Demos for this section are in the following directory:
  - `transformationsKV`

Overview

- Spark provides RDD transformation operations specifically for RDDs of key-value pairs
  - In addition to the general transformation operations we discussed in the previous section

- We'll take a look at the common key-pair transformation operations in this section

Working with Keys and Values

- `keys`
  - Returns an RDD containing just the keys from the source RDD
- `values`
  - Returns an RDD containing just the values from the source RDD
- `mapValues(func)`
  - Applies the func to all the values in the source RDD
  - Returns an RDD containing original keys and transformed values

Joins

- `join(otherKvRdd)`
  - Takes another RDD of key-value pairs
  - Performs an inner join between the source RDD and the arg RDD
- `leftOuterJoin(otherKvRdd)`
  - Similar to `join`, except it performs a left outer join
- `rightOuterJoin(otherKvRdd)`
  - Similar to `join`, except it performs a right outer join
- `fullOuterJoin(otherKvRdd)`
  - Similar to `join`, except it performs a full outer join

Key-Based Operations

- `sampleByKey(replaceFlag, fractionsMap)`
  - Takes a fractions map as an argument, indicating the approximate fractional sample size desired for each key
  - Returns an RDD that contains random items from the source RDD

- `sourceRdd.subtractByKey(otherKvRdd)`
  - Takes another RDD of key-value pairs
  - Returns an RDD of key-value pairs, containing items in the source RDD but not in the arg RDD

- `groupByKey`
  - Returns an RDD of key-value pairs, grouped by key

- `reduceByKey(associativeBinaryOperator)`
  - Takes an associative binary operator as an argument
  - Reduces values with the same key, via that operator

3. Actions

- Overview
- Aggregation
- Taking items
- Collection and reduction
- Key-based actions
- Actions on numeric types
- Saving an RDD

- Demos for this section are in the following directory:
  - `actions`

Overview

- Actions are RDD methods that return a result to the program, or which save the RDD to a storage system
  - It's only when you call an action method that Spark will perform pending transformation methods
  - This is the "lazy processing" concept we discussed earlier - an important optimization feature in Spark

- We'll take a look at the common RDD action methods in this section

Aggregation

- `first`
  - Returns the first element in the source RDD
- `min, max`
  - Return the minimum and maximum elements in the source RDD
- `count`
  - Returns a count of elements in the source RDD
- `countByValue`
  - Returns the count of each unique element in the source RDD
  - Returns the info as a Map (key is the element, value is the count)

Taking Items

- `take(number)`
  - Returns an array containing the first few items, as specified by number argument
- `takeOrdered(number)`
  - Returns an array containing the smallest few items, as specified by number argument
- `takeTop(number)`
  - Returns an array containing the largest few items, as specified by number arg

Collection and Reduction

- `collect`
  - Returns the elements in the source RDD as an array
  - Use with caution - it moves data from all the worker nodes to the driver program (could crash the driver program if very large RDD)

- `reduce(associativeBinaryOperator)`
  - Takes an associative binary operator as an argument
  - Reduces items in the source RDD, via that operator

Key-Based Actions

- `countByKey`
  - Operates on an RDD of key-value pairs
  - Returns a map of key-count pairs

- `lookup(key)`
  - Operates on an RDD of key-value pairs
  - Takes a key as an argument, and returns a sequence of all the values mapped to that key

Actions on Numeric Types

- `sum`
  - Takes a numeric RDD and returns the sum
- `mean`
  - Takes a numeric RDD and returns the average
- `stdev`
  - Takes a numeric RDD and returns the standard deviation
- `variance`
  - Takes a numeric RDD and returns the variance

Saving an RDD

- `saveAsTextFile(directory)`
  - Converts each element in the source RDD to a string, and stores each string on a separate line
- `saveAsObjectFile(directory)`
  - As above, except each element is saved as a serialized Java object in the specified directory
- `saveAsSequenceFile(directory)`
  - Saves an RDD of key-value pairs in SequenceFile format
- Note:
  - All of these methods take a directory name as input, and create one file for each RDD partition in the specified directory

4. Caching

- Recap of lazy operations
- The need for caching
- Caching RDDs
- The cache() method
- The persist() method
- Storage levels for the persist() method

- Demos for this section are in the following directory:
  - `caching`

Recap of Lazy Operations

- As discussed previously, RDD creation and transformation methods are lazy operations
  - Spark just keeps track of the hierarchy of operations required

- When an action method is called on an RDD…
  - Spark creates that RDD from its parents, which might require creation of the parent RDDs, and so on
  - This process continues until Spark gets to the root RDD, which Spark creates by reading data from a storage system

The Need for Caching

- The process on the previous slide happens every time you call an action method
  - Every time you call an action method on an RDD, Spark traverses the RDD hierarchy and computes all the transformations in the chain

- This can cause undesirable re-computations
  - E.g. the following code calls the `count()` action method twice
  - This will cause the `filter()` operation to be performed twice!

```
// This code is in nocache.scala.

val logs = sc.textFile("log_files")

val errorLogs = logs filter { line => line.contains("ERROR") }

val errorCount      = errorLogs.count
val errorCountAgain = errorLogs.count
```

Caching RDDs

- Spark allows you to cache an RDD
  - Via the `cache()` and `persist()` methods
  - See following slides for examples

- When you cache an RDD:
  - Nothing happens immediately…
  - The first time you call an action method on the RDD, Spark will cache the result of all transformations up to that point

The cache() Method

- The `cache()` method stores an RDD in memory on the executors across a cluster

```
// This code is in cache.scala.

val logs = sc.textFile("log_files")

val errorLogs = logs filter { line => line.contains("ERROR") }
errorLogs.cache

val errorCount      = errorLogs.count
val errorCountAgain = errorLogs.count
```

The persist() Method

- The `persist()` method allows you to specify where and how you'd like the dataset to be persisted

```
// This code is in persist.scala.

import org.apache.spark.storage.StorageLevel._

val logs = sc.textFile("log_files")

val errorLogs = logs filter { line => line.contains("ERROR") }
errorLogs.persist(DISK_ONLY)

val errorCount = errorLogs.count
```

Storage Levels for the persist() Method

Storage level	Description
MEMORY_ONLY	Store RDD as raw Java objects in the JVM. If the RDD doesn't fit in memory, some partitions will not be cached and will be recomputed each time they're needed.
MEMORY_AND_DISK	Store RDD as raw Java objects in the JVM. If the RDD doesn't fit in memory, store partitions that don't fit on disk.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte-array per partition). Generally more space-efficient than raw 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 each time 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.

5. Spark Jobs - The Big Picture

- Jobs
- Stages and tasks

Jobs

- Recap: What is a job?
  - A job is a set of computations that Spark performs to return the results of an action to a driver program
- An application can launch one or more jobs
  - An application launches a job by calling an action method of an RDD (i.e. an action method triggers a job)
  - Spark applies the transformations required to create the RDD whose action method was called
  - Finally, Spark performs the computations specified by the action
- A job is completed when a result is returned to a driver program

Stages and Tasks

- When an application calls an RDD action method, Spark creates a DAG of task stages
  - It groups tasks into stages using shuffle boundaries
  - Tasks that don't require a shuffle are grouped into the same stage
  - A task that requires its input data to be shuffled begins a new stage

- A stage can be executed on multiple concurrent threads
  - Spark submits tasks to the executors, which run the tasks in parallel
  - If a node fails while working on a task, Spark resubmits task to another node

Any Questions?