Transformations and Actions

operations on RDDs.

Big Data Analysis with Scala and Spark

Heather Miller

Transformations and Actions

Recall ***transformers*** and ***accessors*** from Scala sequential and parallel collections.

Transformations and Actions

Recall ***transformers*** and ***accessors*** from Scala sequential and parallel collections.

Transformers. Return new collections as results. (Not single values.)
Examples: `map`, `filter`, `flatMap`, `groupBy`

```
map(f: A => B): Traversable[B]
```

Transformations and Actions

Recall ***transformers*** and ***accessors*** from Scala sequential and parallel collections.

Transformers. Return new collections as results. (Not single values.)
Examples: `map`, `filter`, `flatMap`, `groupBy`

```
map(f: A => B): Traversable[B]
```

Accessors: Return single values as results. (Not collections.)
Examples: `reduce`, `fold`, `aggregate`.

```
reduce(op: (A, A) => A): A
```

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

Transformations. Return new ~~collections~~ RDDs as results.

Actions. Compute a result based on an RDD, and either returned or saved to an external storage system (e.g., HDFS).

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

Transformations. Return new ~~collections~~ RDDs as results.

They are lazy, their result RDD is not immediately computed.

Actions. Compute a result based on an RDD, and either returned or saved to an external storage system (e.g., HDFS).

They are eager, their result is immediately computed.

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

Transformations. Return new ~~collections~~ RDDs as results.

They are lazy, their result RDD is not immediately computed.

Actions. Compute a result based on an RDD, and either returned or saved to an external storage system (e.g., HDFS).

They are eager, their result is immediately computed.

Laziness/eagerness is how we can limit network communication using the programming model.

Example

Consider the following simple example:

sc → SparkContext

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
```

RDD[String]

RDD[Int]

What has happened on the cluster at this point?

Example

Consider the following simple example:

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
```

What has happened on the cluster at this point?

Nothing. Execution of `map` (a transformation) is deferred.

To kick off the computation and wait for its result...

Example

Consider the following simple example:

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
val totalChars = lengthsRdd.reduce(_ + _)
```

...we can add an action

Common Transformations in the Wild

LAZY!!

map

`map[B](f: A => B): RDD[B]`

Apply function to each element in the RDD and retrun an RDD of the result.

flatMap

`flatMap[B](f: A => TraversableOnce[B]): RDD[B]`

Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned.

filter

`filter(pred: A => Boolean): RDD[A]`

Apply predicate function to each element in the RDD and return an RDD of elements that have passed the predicate condition, `pred`.

distinct

`distinct(): RDD[B]`

Return RDD with duplicates removed.

Common Actions in the Wild

EAGER!

collect

```
collect(): Array[T]
```

Return all elements from RDD.

count

```
count(): Long
```

Return the number of elements in the RDD.

take

```
take(num: Int): Array[T]
```

Return the first `num` elements of the RDD.

reduce

```
reduce(op: (A, A) => A): A
```

Combine the elements in the RDD together using `op` function and return result.

foreach

```
foreach(f: T => Unit): Unit
```

Apply function to each element in the RDD.

Another Example

Let's assume that we have an `RDD[String]` which contains gigabytes of logs collected over the previous year. Each element of this RDD represents one line of logging.

Assuming that dates come in the form, `YYYY-MM-DD:HH:MM:SS`, and errors are logged with a prefix that includes the word "error"...

How would you determine the number of errors that were logged in December 2016?

```
val lastYearsLogs: RDD[String] = ...
```

Another Example

Let's assume that we have an `RDD[String]` which contains gigabytes of logs collected over the previous year. Each element of this RDD represents one line of logging.

Assuming that dates come in the form, `YYYY-MM-DD:HH:MM:SS`, and errors are logged with a prefix that includes the word "error"...

How would you determine the number of errors that were logged in December 2016?

```
val lastYearsLogs: RDD[String] = ...
val numDecErrorLogs
  = lastYearsLogs.filter(lg => lg.contains("2016-12") && lg.contains("error"))
                 .count()
```

Benefits of Laziness for Large-Scale Data

Spark computes RDDs the first time they are used in an action.

This helps when processing large amounts of data.

filter
↓
take

Example:

```
val lastYearsLogs: RDD[String] = ...
val firstLogsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).take(10)
```

The execution of `filter` is deferred until the `take` action is applied.

Spark leverages this by analyzing and optimizing the **chain of operations** before executing it.

Spark will not compute intermediate RDDs. Instead, as soon as 10 elements of the filtered RDD have been computed, `firstLogsWithErrors` is done. At this point Spark stops working, saving time and space computing elements of the unused result of `filter`.

Transformations on Two RDDs

LAZY!!

rdd1 rdd2

val rdd3 = rdd1.union(rdd2)

RDDs also support set-like operations, like `union` and `intersection`.

Two-RDD transformations combine two RDDs are combined into one.

union	**union(other: RDD[T]): RDD[T]** ← Return an RDD containing elements from both RDDs.
intersection	**intersection(other: RDD[T]): RDD[T]** ← Return an RDD containing elements only found in both RDDs.
subtract	**subtract(other: RDD[T]): RDD[T]** ← Return an RDD with the contents of the other RDD removed.
cartesian	**cartesian[U](other: RDD[U]): RDD[(T, U)]** ← Cartesian product with the other RDD.

Other Useful RDD Actions

EAGER!!

RDDs also contain other important actions unrelated to regular Scala collections, but which are useful when dealing with distributed data.

takeSample

```
takeSample(withRepl: Boolean, num: Int): Array[T]
```

Return an array with a random sample of num elements of the dataset, with or without replacement.

takeOrdered

```
takeOrdered(num: Int)(implicit
ord: Ordering[T]): Array[T]
```

Return the first n elements of the RDD using either their natural order or a custom comparator.

saveAsTextFile

```
saveAsTextFile(path: String): Unit
```

Write the elements of the dataset as a text file in the local filesystem or HDFS.

saveAsSequenceFile

```
saveAsSequenceFile(path: String): Unit
```

Write the elements of the dataset as a Hadoop SequenceFile in the local filesystem or HDFS.