

IIT KHARAGPUR

IIT GANDHINAGAR


NPTEL ONLINE
CERTIFICATION COURSES

Scalable Data Science

Lecture 18: Spark

Sourangshu Bhattacharya

Computer Science and Engineering

IIT KHARAGPUR

In the previous lectures:

- Outline:
  - What is Big Data and Hadoop?
  - What is Map Reduce ?
  - Map Reduce programmming.
  - Map Reduce implementation details

In this Lecture:

- Outline:
  - Scala
    - Var and Val
    - Classes and objects
    - Functions and higher order functions
    - Lists

SCALA

Scala

- Scala is both functional and object-oriented
  - every value is an object
  - every function is a value--including methods
- Scala is interoperable with java.
- Scala is statically typed
  - includes a local type inference system:

Var and Val

- Use `var` to declare variables:
  - `var x = 3;`
  - `x += 4;`
- Use `val` to declare values (final vars)
  - `val y = 3;`
  - `y += 4; // error`
- Notice no types, but it is statically typed
  - `var x = 3;`
  - `x = "hello world"; // error`
- Type annotations:
  - `var x : Int = 3;`

Class definition

```
class Point(val xc: Int, val yc: Int) {
  var x: Int = xc
  var y: Int = yc

  def move(dx: Int, dy: Int) {
    x = x + dx
    y = y + dy
    println ("Point x location : " + x);
    println ("Point y location : " + y);
  }
}
```

Scala

- Class instances
  - `val c = new IntCounter[String];`
- Accessing members
  - `println(c.size); // same as c.size()`
- Defining functions:
  - `def foo(x : Int) { println(x == 42); }`
  - `def bar(y : Int): Int = y + 42; // no braces needed!`
  - `def return42 = 42; // No parameters either!`

Functions are first-class objects

- Functions are **values** (like integers, etc.) and can be assigned to variables, passed to and returned from functions, and so on
- Wherever you see the => symbol, it's a literal function
- Example (assigning a literal function to the variable foo):
  - ```
    scala> val foo =
      (x: Int) => if (x % 2 == 0) x / 2 else 3 * x + 1
    foo: (Int) => Int = <function1>

    scala> foo(7)
    res28: Int = 22
    ```
- The basic syntax of a function literal is
  parameter_list => ***function_body***
- In this example, foreach is a function that takes a function as a parameter:
  - myList.foreach(i => println(2 * i))
    ```

Functions as parameters

- To have a function parameter, you must know how to write its type:
  - `(type1, type2, ..., typeN) => return_type`
  - `type => return_type // if only one parameter`
  - `() => return_type // if no parameters`

- Example:

```
scala> def doTwice(f: Int => Int, n: Int) = f(f(n))
doTwice: (f: (Int) => Int,n: Int)Int

scala> def collatz(n: Int) = if (n % 2 == 0) n / 2 else 3 * n + 1
collatz: (n: Int)Int

scala> doTwice(collatz, 7)
res2: Int = 11

scala> doTwice(a => 101 * a, 3)
res4: Int = 30603
```

Scala

- Defining lambdas – nameless functions (types sometimes needed)
  - `val f = x :Int => x + 42;`
- Closures (context sensitive functions)
  - `var y = 3;`
  - `val g = {x : Int => y += 1; x+y; }`
- Maps (and a cool way to do some functions)
  - `List(1,2,3).map(_+10).foreach(println)`
- Filtering (and ranges!)
  - `1 to 100 filter (_ % 7 == 3) foreach (println)`

Lists

- Scala's Lists are more useful, and used more often, than Arrays
  - `val list1 = List(3, 1, 4, 1, 6)`
  - `val list2 = List[Int]() // An empty list must have an explicit type`

- By default, Lists, like Strings, are **immutable**
  - Operations on an immutable List return a new List

- Basic operations:
  - ***list***.head (or ***list*** head) returns the first element in the list
  - ***list***.tail (or ***list*** tail) returns a list with the first element removed
  - ***list***(***i***) returns the i^{th} element (starting from 0) of the list
  - ***list***(***i***) = ***value*** is **illegal** (immutable, remember?)

- There are over 150 built-in operations on Lists—use the API!

Higher-order methods on Lists

- `map` applies a one-parameter function to every element of a List, returning a new List
  - `scala> def double(n: Int) = 2 * n`
    `double: (n: Int)Int`
  - `scala> val ll = List(2, 3, 5, 7, 11)`
    `ll: List[Int] = List(2, 3, 5, 7, 11)`
  - `scala> ll map double`
    `res5: List[Int] = List(4, 6, 10, 14, 22)`
  - `scala> ll map (n => 3 * n)`
    `res6: List[Int] = List(6, 9, 15, 21, 33)`

- `filter` applies a one-parameter test to every element of a List, returning a List of those elements that pass the test
  - `scala> ll filter(n => n < 5)`
    `res10: List[Int] = List(2, 3)`
  - `scala> ll filter (_ < 5)` // abbreviated function where parameter is used once
    `res11: List[Int] = List(2, 3)`

More higher-order methods

- `def filterNot(p: (A) => Boolean): List[A]`
  - Selects all elements of this list which do not satisfy a predicate
- `def count(p: (A) => Boolean): Int`
  - Counts the number of elements in the list which satisfy a predicate
- `def forall(p: (A) => Boolean): Boolean`
  - Tests whether a predicate holds for every element of this list
- `def exists(p: (A) => Boolean): Boolean`
  - Tests whether a predicate holds for at least one of the elements of this list
- `def find(p: (A) => Boolean): Option[A]`
  - Finds the first element of the list satisfying a predicate, if any
- `def sortWith(lt: (A, A) => Boolean): List[A]`
  - Sorts this list according to a comparison function

SPARK

Spark

Spark is an In-Memory Cluster Computing platform for Iterative and Interactive Applications.

http://spark.apache.org

Spark

- Started in AMPLab at UC Berkeley.
- Resilient Distributed Datasets.
- Data and/or Computation Intensive.
- Scalable – fault tolerant.
- Integrated with SCALA.
- Straggler handling.
- Data locality.
- Easy to use.

Background

- Commodity clusters have become an important computing platform for a variety of applications
  - **In industry:** search, machine translation, ad targeting, …
  - **In research:** bioinformatics, NLP, climate simulation, …

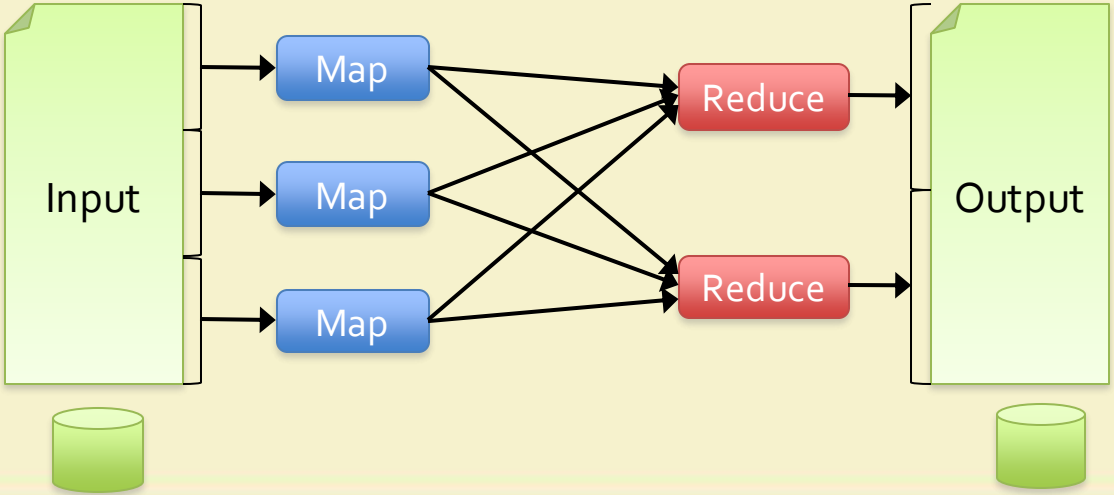
- High-level cluster programming models like MapReduce power many of these apps

- *Theme of this work: provide similarly powerful abstractions for a broader class of applications*

Motivation

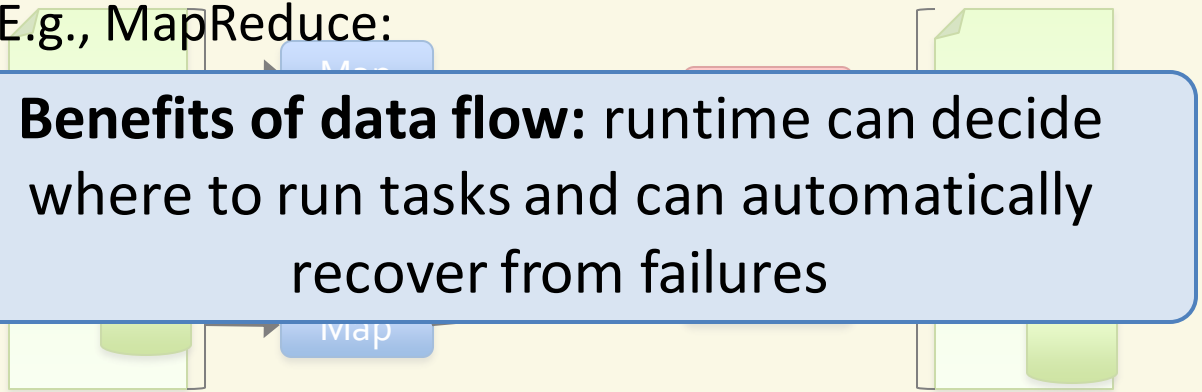
Current popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:



Motivation

- Current popular programming models for clusters transform data flowing from stable storage to stable storage
- E.g., MapReduce:



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Motivation

- Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (many in machine learning)
  - **Interactive** data mining tools (R, Excel, Python)
- Spark makes working sets a first-class concept to efficiently support these apps

Spark Goal

- Provide distributed memory abstractions for clusters to support apps with working sets
- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes & stragglers)
  - Data locality
  - Scalability

Solution: augment data flow model with "resilient distributed datasets" (RDDs)

Conclusion:

- We have seen:
  - Scala
    - Var and Val
    - Classes and objects
    - Functions and higher order functions
    - Lists

References:

- Any book on scala.

Thank You!!