Introduction to Big Data

Alexey Svyatkovskiy

Princeton University

October 21, 2016

Spark: a data analysis framework
(like e.g. Tensorflow, but with different strengths and weaknesses).

Scala: Spark's native language, used as a command prompt
(the way that Python is used in Tensorflow).

Python and R: also supported, but with some limitations that are not presented in Scala

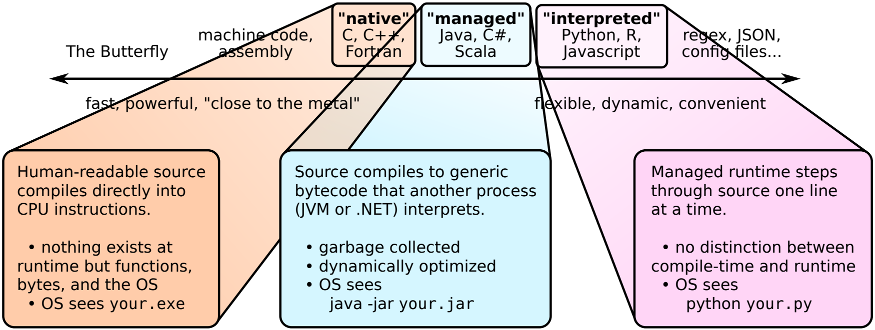
Outline

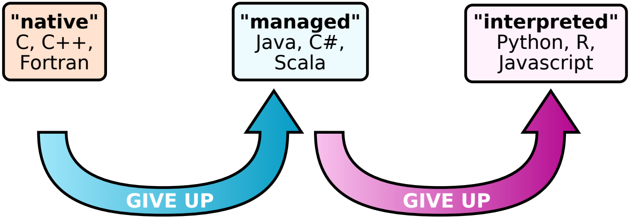
1. 5 minute talk on programming languages: Python, Scala
2. 5 minute talk on Spark and Big Data in general
3. Move on to Spark hands-on exercises

		"native"	**"managed"**	**"interpreted"**	
The Butterfly	machine code, assembly	C, C++, Fortran	Java, C#, Scala	Python, R, Javascript	regex, JSON, config files...

fast, powerful, "close to the metal"

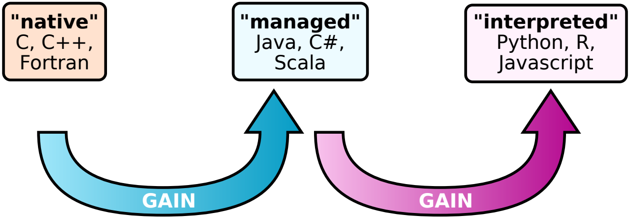
flexible, dynamic, convenient


"native"
C, C++,
Fortran
"managed"
Java, C#,
Scala
"interpreted"
Python, R,
Javascript
The Butterfly
machine code,
assembly
regex, JSON,
config files...
fast, powerful, "close to the metal"
flexible, dynamic, convenient
Human-readable source compiles directly into CPU instructions.
• nothing exists at runtime but functions, bytes, and the OS
• OS sees your.exe
Source compiles to generic bytecode that another process (JVM or .NET) interprets.
• garbage collected
• dynamically optimized
• OS sees java -jar your.jar
Managed runtime steps through source one line at a time.
• no distinction between compile-time and runtime
• OS sees python your.py



- memory management
- byte-level control
- some speed

- compilation step
- type safety
- some more speed



- high-level abstractions
- hardware portability
- easier debugging

- simplicity, flexibility
- rapid prototyping of incomplete ideas

What happens in the compilation step?

- Whole program is interpreted and turned into machine instructions (possibly for a virtual machine).
- All variables are interpreted as belonging to specific types: `int`, `string`, `MissileController`...
- Uses of these variables are checked for validity:
  - can't pass a `MissileController` into the cosine function;
  - can't call `launchAllMissiles()` on a `string`.

Interpreted languages do none of these things; you find out about misuses of variables at runtime (can be good, can be bad).

Why should you care?

- Compilation step can get in the way of testing a program one piece at a time.
- The type check is a *formal proof* that the program is free of certain types of errors; it won't fail after hours of running.

Why should you care?

- Compilation step can get in the way of testing a program one piece at a time.
- The type check is a *formal proof* that the program is free of certain types of errors; it won't fail after hours of running.

Python

- Dynamically typed, flexible, simple
- No distinction between compile-time and runtime
- Everything is a pointer

Scala

- Scala compiles to bytecode that runs on the Java Virtual Machine (JVM).
- It emphasizes type safety (even more than C++).
- and an interactive prompt for testing small components or interacting with a running program.

https://en.wikipedia.org/wiki/List_of_JVM_languages

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

List of JVM languages

From Wikipedia, the free encyclopedia

This **list of JVM Languages** comprises notable computer programming languages that are used to produce software that runs on the Java Virtual Machine (JVM). Some of these languages are interpreted by a Java program, and some are compiled to Java bytecode and JIT-compiled during execution as regular Java programs to improve performance.

The JVM was initially designed to support only the Java programming language. However, as time passed, ever more languages were adapted or designed to run on the Java platform.

Contents [hide]

High-profile languages [edit]

Apart from the Java language itself, the most common or well-known JVM languages are:

- Clojure, a functional Lisp dialect
- Groovy, a programming and scripting language
- Scala, an object-oriented and functional programming language[1]
- JRuby, an implementation of Ruby
- Jython, an implementation of Python

BLACKDUCK | Open HUB

Follow @OH | SIGN IN | JOIN NOW

PROJECTS | PEOPLE | ORGANIZATIONS | TOOLS | CODE | BLOG

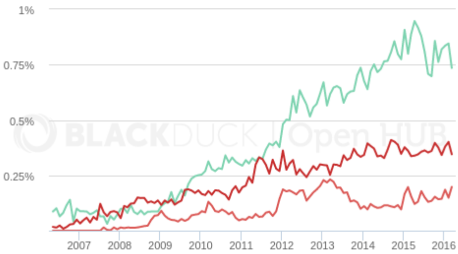
Projects | Search...

Compare Languages

Monthly Commits | Monthly Contributors | Monthly Lines of Code Changed | Monthly Projects

Monthly Commits (Percent of Total)

The lines show the count of monthly commits made by source code developers. Commits including multiple languages are counted once for each language. More



Clojure

Groovy

Scala

[None]

Update

BLACKDUCK | Open HUB

Follow @OH SIGN IN JOIN NOW

PROJECTS PEOPLE ORGANIZATIONS TOOLS CODE BLOG

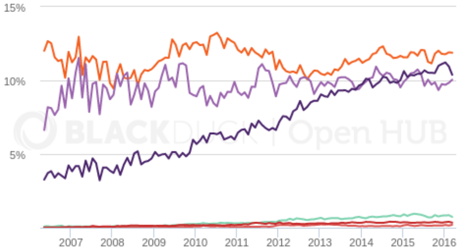
Projects Search...

Compare Languages

Monthly Commits | Monthly Contributors | Monthly Lines of Code Changed | Monthly Projects

Monthly Commits (Percent of Total)

The lines show the count of monthly commits made by source code developers. Commits including multiple languages are counted once for each language. More



- Clojure
- C++
- Groovy
- Java
- Python
- Scala
- [None]

Update

Big Data tools

Big Data tools

Distributed analysis frameworks:

- Apache Hadoop
- Apache Spark

Query engines:

- Elasticsearch
- Apache Impala
- Spark SQL
- Apache Hive

Data pipelines:

- Apache Kafka
- Apache Flume

Streaming & micro-batch tools:

- Apache Storm
- Apache Flink
- Spark Streaming

Cluster managers:

- Apache YARN
- Slurm

Machine learning & Deep learning:

- Scikit-Learn
- Tensorflow, Theano, Torch
- Keras, Lasagne
- Spark ML, MLlib

Big Data tools

Distributed analysis frameworks:

- Apache Hadoop
- **Apache Spark**

Query engines:

- Elasticsearch
- Apache Impala
- **Spark SQL**
- Apache Hive

Data pipelines:

- Apache Kafka
- Apache Flume

Streaming & micro-batch tools:

- Apache Storm
- Apache Flink
- **Spark Streaming**

Cluster managers:

- Apache YARN
- Slurm

Machine learning & Deep learning:

- Scikit-Learn
- Tensorflow, Theano, Torch
- Keras, Lasagne
- **Spark ML, MLlib**

From Hadoop to Spark

2003–2004

Google published *The Google File System* and *MapReduce: Simplified Data Processing on Large Clusters.*

2006

HDFS and Hadoop-MapReduce projects started at Yahoo! but within Apache, fully open-source.

2008–2009

Hadoop sorted TB–PB of data in record time. Started getting contributions from Facebook, LinkedIn, eBay, and IBM.

2009

Spark began as a class project at Berkley, targeting *iterative* map-reduce for machine learning.

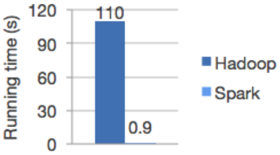
2013

Spark became an Apache project and Databricks founded. In 2014, Spark won records for TB–PB sorting.

Google Trends search results

Spark's specialty: persisting data in RAM for iterative algorithms.

Hadoop (and every other distributed batch system I've heard of) has to load data from disk for each pass.



Beyond this killer app, Spark was designed for exploratory data analysis; Hadoop was designed for large applications.

- Native Spark runs on a command line in Scala (natively), Python, and R (through a bridge).
- Hadoop asks the user to extend Mapper and Reducer classes.
- Spark has many minor conveniences.
- Spark generalizes on the map-reduce concept to chains of functional primitives.

Chains of functional primitives

Functional programming style is common among data analysts using R (inherited from Scheme).

```
for (i = 0;  i < nEvents;  i++) {
    event = events(i);
    if (condition(event))
        continue;
    add_to_output(calculation(event));
}
```

```
output =
  map(calculation,
    filter(condition,
                events))
```

Chains of functional primitives

Functional programming style is common among data analysts using R (inherited from Scheme).

Scala's object orientation lets us chain functors without nesting.

```
for (i = 0;  i < nEvents;  i++) {
    event = events(i);
    if (condition(event))
        continue;
    add_to_output(calculation(event));
}
```

```
output =
  events.filter(condition)
        .map(calculation)
```

Chains of functional primitives

Functional programming style is common among data analysts using R (inherited from Scheme).

Scala's object orientation lets us chain functors without nesting.

```
for (i = 0;  i < nEvents;  i++) {
    event = events(i);
    if (condition(event))
        continue;
    add_to_output(calculation(event));
}
```

```
output =
  events.filter(condition)
        .map(calculation)
```

- The "map" functor *says less* than "for"— it doesn't specify an order in which events must be processed.
- Underlying system can distribute and collect however it likes.
- Also hides index arithmetic from the user: datasets can be spliced automatically.

"Monad-like" functional primitives:

Transforming one data table into another.

	input	function	output	operation
map	table of A	$f : A \to B$	table of B	apply f to each row A, get a table of the same number of rows B
	a.k.a. "lapply" (R), "SELECT" (SQL), list comprehension (Python)			
filter	table of A	$f : A \to$ boolean	table of A	get a shorter table with the same type of rows
	a.k.a. single brackets (R), "WHERE" (SQL), list comprehension (Python)			
flatMap	table of A	$f : A \to$ table of B	table of B	compose map and flatten, get a table of any length
	a.k.a. "map" (Hadoop), "EXPLODE" (SQL), $>>=$ (Haskell)			

“Monoid-like” functional primitives:

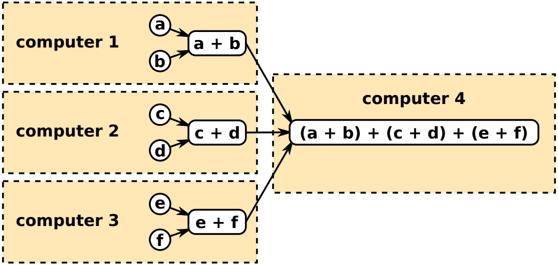
Summarizing an data table using a counter, summation, or histogram.

	input	function(s)	output	operation
reduce	table of A	$f : (A, A) \to A$	single A	apply f to the running sum and one more element
aggregate	table of A, initial value B (“zero”)	$f : (A, B) \to B$ $f : (B, B) \to B$ (increment and combine)	single value B	accumulate a counter with a different data type from the input
aggregate by key	table of $\langle K, A \rangle$, initial value B	$f : (A, B) \to B$ $f : (B, B) \to B$	pairs $\langle K, B \rangle$	aggregate independently for each key

a.k.a. “reduce” (Hadoop), “GROUP BY” (SQL)

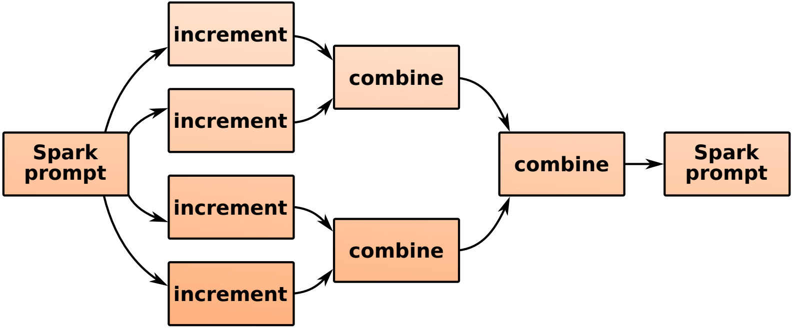
Associativity is key

"Monad" and "monoid" refer to mathematical properties of the operator, the most important being associativity, which allows the user's function to be dispatched arbitrarily.



Aggregate functional

`RDD.aggregate(initialize)(increment, combine)`



(Hadoop equivalent: reduce; SQL equivalent: "`GROUP BY`")

Spark Exercises: Day 1

Histogrammar

We want to avoid downloading the whole dataset to a laptop for a traditional ntuple-analysis.

Spark has a functional for reducing data in a distributed way:

```
RDD.aggregate(initialize)(increment, combine)
```

where

- `RDD` is a collection of data of type $\mathcal{D}$ (end of skimming chain)
- `initialize` creates a counter of type $\mathcal{C}$
- `increment` is a function from $(\mathcal{C}, \mathcal{D}) \to \mathcal{C}$
- `combine` is a function from $(\mathcal{C}, \mathcal{C}) \to \mathcal{C}$

First idea:

Move the logic of histogram-filling into the booking stage.

```
val h = Histogram("pt", 100, 0, 20,
                  {d => sqrt(d.px**2 + d.py**2)})
```

“fill rule” $f : \mathcal{D} \to \mathbb{R}$

This functional design allows the filling and merging to be automatic: no user input required.

```
RDD.aggregate(h)(auto_increment(), auto_combine())
```

Second idea:

Collect histograms into a container that also has automated filling and merging.

```
val pack_o_histograms = Label(
      "pt" -> Histogram(100, 0, 20, fill_pt),
      "Emiss" -> Histogram(100, 0, 50, fill_Emiss),
      ...)

RDD.aggregate(pack_o_histograms)(auto_increment(),
                                 auto_combine())
```

(`Label` and `Histogram` share a superclass; `auto_increment()` and `auto_combine()` call them the same way.)

Third idea:

Let all of these pieces be composable.

```
val directories =
      Label("dir1" ->
                 Label("pt" -> Histogram(...),
                       "Emiss" -> Histogram(...)),
            "dir2" ->
                 Label("pass" -> Count(...),
                       "maxpt" -> Maximize(...)))
```

(Combining directories of histograms is similar to ROOT's `hadd`.)