Spark Workshop

Daniel Hinojosa

Conventions in the slides

The following typographical conventions are used in this material:

Italic
Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold` Shows commands or other text that should be typed literally by the user.

`Constant width italic`
Shows text that should be replaced with user-supplied values or by values determined by context.

Shell Conventions

All shells (bash, zsh, Windows Shell) are represented as `%`

```
% calendar
```

All Spark shells are represented as `scala>`

```
scala> spark.range(1,100)
```

Big Data

About Big Data

- Google predicting how and when the flu would spread using large amounts of data
- In 2003, Oren Etzioni, after spurned from a high price airline ticket sought large amounts of data to predict ticket prices over time

Source: Big Data "A Revolution That Will Transform The Way we Live, Work, and Think"

The Process of Big Data

- We must store large amounts of Data
- Human Generated Data Types
- Derived from large, albeit inconsistent, user base (Facebook: 1.13 billion daily active users)
- Computationally Complex Aggregations and Interpretations.

Storage: Google

- Has approximately 10 EB of active storage in hard drives
- Is the single biggest consumer of "cold" tape storage, purchasing 200,000 per year
- Has data centers around the globe

Source: https://what-if.xkcd.com/63/

Storage: Facebook

- Has large exabyte datacenters
- Stores more than 240 billion photos
- 350 million new photos every single day
- Deploys 7 PB of storage gear every month
- Doesn’t make money of purchases by their likes and other data

Source: http://www.datacenterknowledge.com/archives/2013/01/18/facebook-builds-new-data-centers-for-cold-storage

Storage: Twitter

- Multiple Hadoop clusters storing over 500 PB divided in four groups (real time, processing, data warehouse and cold storage).

- Manhattan - the backend for Tweets, Direct Messages, Twitter accounts, and more
- Legacy Gizzard/MySQL based sharded cluster for storing our graphs
- Blobstore - Image, video and large file store where we store hundreds of billions objects.
- Redis and Memcache clusters: caching our users, timelines, tweets and more
- SQL: This includes MySQL, PostgreSQL and Vertica. MySQL/PosgreSQL are used where we need strong consistency, managing ads campaign, ads exchange as well as internal tools.
- Vertica is a column store often used as a backend for Tableau supporting sales and user organizations

Source: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html

Storage for Big Data

- The previous storage reality shows one thing
- RDBMS is slow and a complete scan would take a long time to aggregate
- To solve the problem, data is distributed and redundant for fault tolerance

The actual aggregation of data

- Once stored, how do we aggregate the data and make sense of it?
- This process is what we will talk about, particularly with Apache Spark
- Other competitors want to vie for big data analysis and aggregation

Spark's Competitors

- Apache Storm
  - Distributed stream processing computation framework written in Java and Clojure.
  - Applies real-time analytics, machine learning and continuous monitoring of operations.
- Ray Project (https://github.com/ray-project/ray)
  - Python based
  - High-performance distributed execution framework
  - Includes optimization and learning libraries

This is a very contentious race, there are other competitors

Batch Processing vs. Real Time Processing

- Batch Processing

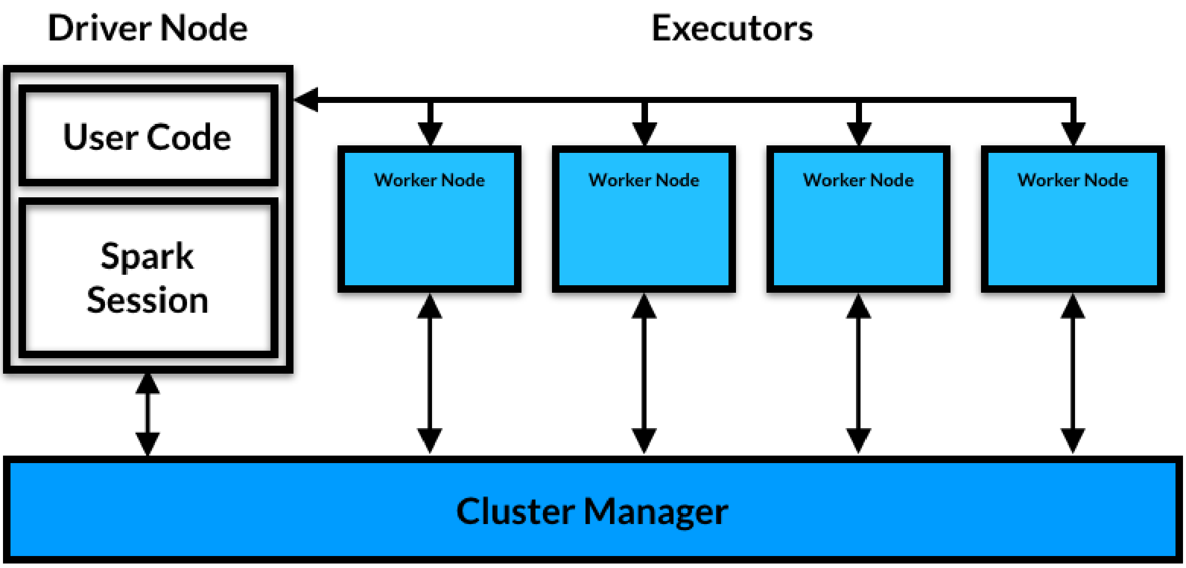
    - In typical scenarios, information is place into datastores in batches
    - ETL (Extract-Transform-Load) is a typical batch processing scenarion
- Real Time Processing
    - Any process that doesn’t wait to gather a batch of data
    - A typical instance of Real Time Processing is Streaming
- Hybrid
    - Spark Streaming for example uses Batched Streaming
    - Batched Streaming operates on batch intervals

Spark Intro

Spark Intro

- Big data processing framework
- Variety of packages built upon Spark engine
- Contains two APIs
  - Unstructured API
    - Lower Level
    - RDD
    - Accumulators
    - Broadcast Variables
  - Structured API
    - Higher Level
    - Optimized
    - DataFrames
    - Datasets
    - Spark SQL

Spark Architecture



Reason for Spark's Existence

- CPU went from single to multi-core
- Hard Drive storage became cheap over time
- This allows for processing of data without expense

Spark Architecture

- The reason for existence is that one computer is too slow for processing data
- A cluster can provide faster processing in parallel.
- Spark is separated by:
  - A `driver` process
  - An `executor` process

The Driver

- The driver node for your application
- Maintains information about the application
- Responds to external programs
- Analyzes work across executors
- Distributes work across executors
- Schedules work across executors

The Executor

- Executes code assigned to it by the driver
- Reports the state of the computation back to the driver

Spark Extras

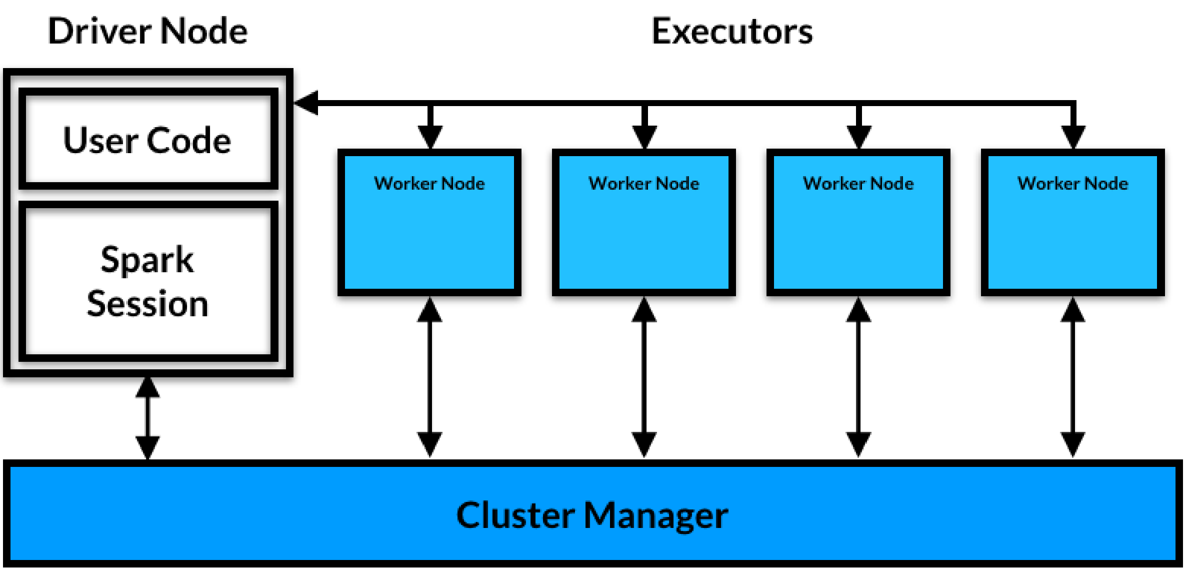
- MLlib - Machine Learning with Spark
- GraphX - for Graph Processing
- SparkR - for working with Clusters using R

Cluster Manager

- Controls the Physical Machines
- Allocates resources to Spark applications

- Cluster Managers can either be:
    - Sparks in-house cluster manager
    - YARN
    - Mesos
- Known as Cluster Mode

Spark Architecture Reviewed



Local Mode

- Instead of remote machines this will run on your internal box
- Easy for testing, in house demonstrations

Languages

- Scala (Spark's default language)
- Python (Does nearly everything that Scala does)
- Java (Louder than Scala)
- SQL (Spark SQL is compliant SQL to interact with querying data)
- R/Spark (The classic Big Data language)

> For this class/workshop we will be using Scala since it is less verbose and has other features that Java does not have.

Spark Overview

Overview of Abstractions and Terms

The following are the main abstractions of Spark

- `DataFrame`
- `Dataset`
- SQL, called SparkSQL
- `RDD` Resilient Distributed Datasets

`DataFrame`

- Are the most efficient due to catalyst optimizer
- Are available in all languages
- A table with data rows and columns
- Analogous to a spreadsheet or table
- **Distributed and spans over multiple machines!**
- Easiest to use, particularly for non-functional programmers

`DataSet`

- Rows optimized by the catalyst optimizer
- Fully functional programmable
  - `map`
  - `filter`
  - `flatMap`
- A `DataFrame` is actually a `Set[Row]`

The "actors" in Spark

Task

- A task is a command sent to the executor by the driver
- Gets processed within a stage

Stages

- Stages are logical separation of processes

- Contains tasks
- If a stage reads from a source it is given its own stage

The "storage" in Spark

Partitions

- For management, Spark breaks up data into atomic chunks
- A Partition is a collection of *rows* that sit on *one machine* in a cluster
- Therefore a `DataFrame`, a `DataSet`, or an `RDD` contains 0 or more partitions
- `DataFrame` is the interface to all the computations and data stored on remote machines
- In local mode they logical partitions are laid across a single instance on each core
- Data can be repartitioned or coalesced down to a certain number of partitions

The "functional data manipulation" in Spark

Transformations

- All data structures are *immutable*
- Any change receives a copy
- Therefore, any change will be done via a transformation
- Should be very familiar if you do functional programming like Scala
- Transformation of a `DataFrame` returns a `DataFrame`

Lazy Evaluation

- All changes do not run right away
- Transformations to DataFrames are calculated and evaluated only when needed
- Before execution a *plan* is automatically created before evaluation
- If a `DataFrame` (or `DataSet`) is used, it is optimized.

Actions

- To trigger the series of transformation we would need an *action* or *terminal operation*
- There are three kinds of actions:
    - View data in the console
    - Collect data

    - Output data to a file system or database
- Many terminal operations include:
    - `reduce`
    - `collect`
    - `count`

Setup

Typical Setup Instructions

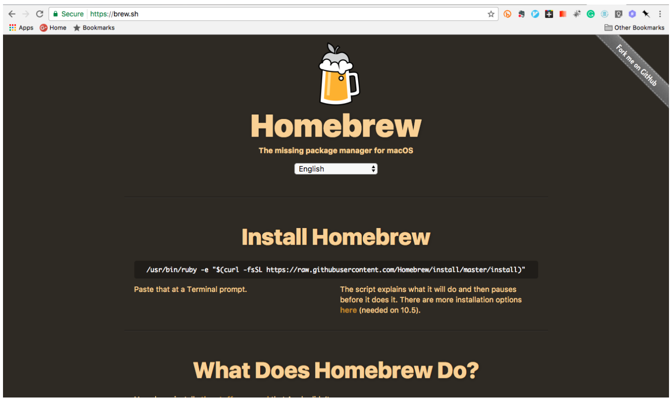
Before we begin it is assumed that all of you have the following tools installed:

- JDK 1.8 (latest java is 1.8.0_161)
- Scala 2.12.3
- SBT 1.0.2
- Spark 2.2.1
- `winutils` (Windows Only)

To verify that all your tools work as expected

```
% javac -version
javac 1.8.0_161

% scala -version
Scala code runner version 2.12.3 -- Copyright 2002-2018, LAMP/EPFL

% java -version
java version "1.8.0_161"
Java(TM) SE Runtime Environment (build 1.8.0_1.8.0_161-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% sbt sbtVersion
[info] Set current project to scala (in build file:/<folder_location>)
[info] 1.0.2

% spark-submit -version

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.1
      /_/

Using Scala version 2.12.3, Java HotSpot(TM) 64-Bit Server VM, 1.8.0_161
Branch
Compiled by user jenkins on 2017-04-25T23:51:10Z
Revision
Url
Type --help for more information.
```

Installing Java, Scala, Spark, SBT on a Mac Automatically with Brew



If you have a mac and brew installed, you can run the following ***and be done!***:

```
% brew update
% brew cask install java
% brew install scala
% brew install sbt
% brew install apache-spark
```

This will require an install of Homebrew. Visit https://brew.sh/ for details of installation if you want to use brew.

Depending on your company's software and security constraints, you may not be able to use brew

If you don't have Java 8 installed

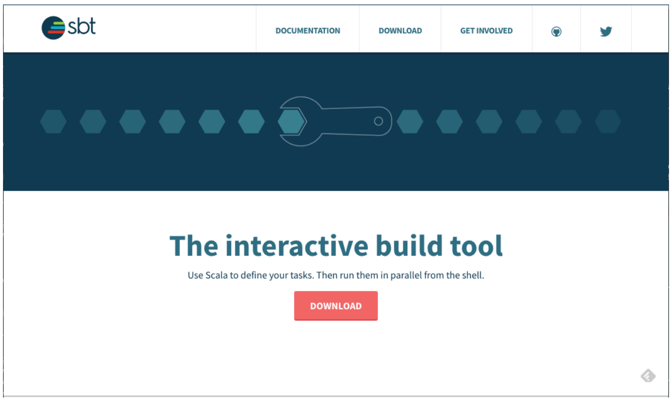
- Visit: http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html
- Select: *Accept License Agreement*
- Download the appropriate Java version based on your architecture.

Linux ARM 32 Hard Float ABI	Linux ARM 64 Hard Float ABI
Linux x86	Linux x86
Linux x64	Linux x64
Mac OS X	Solaris SPARC 64-bit
Solaris SPARC 64-bit	Solaris x64
Solaris x64	Windows x86

If you do not have Scala installed

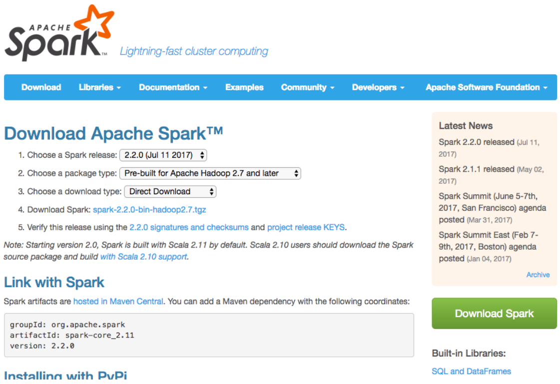
- Visit http://scala-lang.org
- Click the *Download* Button
- Download the appropriate binary for your system:
  - Mac and Linux will load a *.tgz* file
  - Windows will download an *.msi* executable
- For Mac and Linux you can expand with `tar -xvfz scala-2.12.3.tgz`

If you do not have SBT installed



- Visit http://scala-sbt.org
- Click the *Download* Button
- Download the appropriate binary for your system:
  - Mac and Linux will load a *.tgz*, or a *.zip* file
  - Windows will download an *.msi* executable
- For Mac and Linux you can expand with `tar -xvfz scala-2.12.3.tgz`

If you do not have Spark installed



- Visit https://spark.apache.org/downloads.html
- Click the spark-2.2.1-bin-hadoop2.8.1.tgz link to download
- For Mac and Linux, you can expand with `tar -xvfz spark-2.2.1-bin-hadoop2.8.1.tgz` to folder of your choosing
- For Windows, you will need a utility like WinZip to extract a tar.gz file to a folder of your choosing

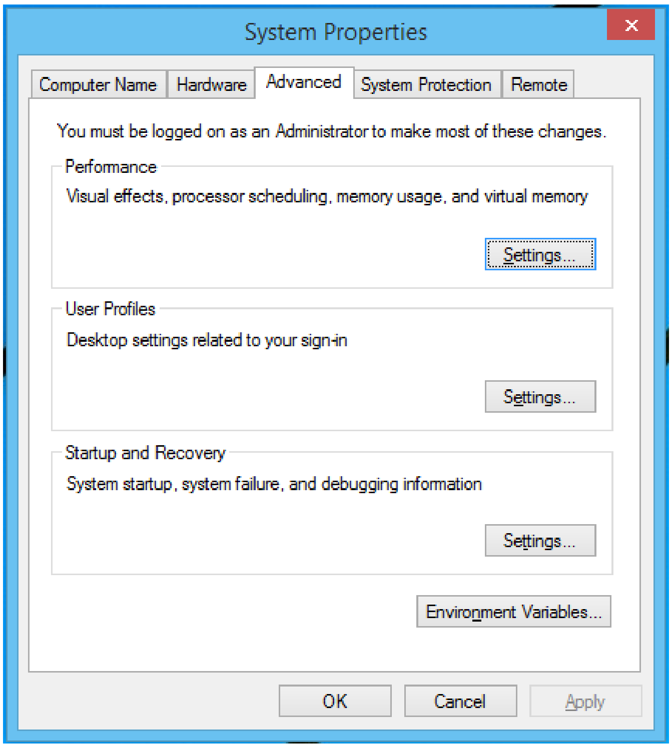
Windows Users Only: Download `winutils`

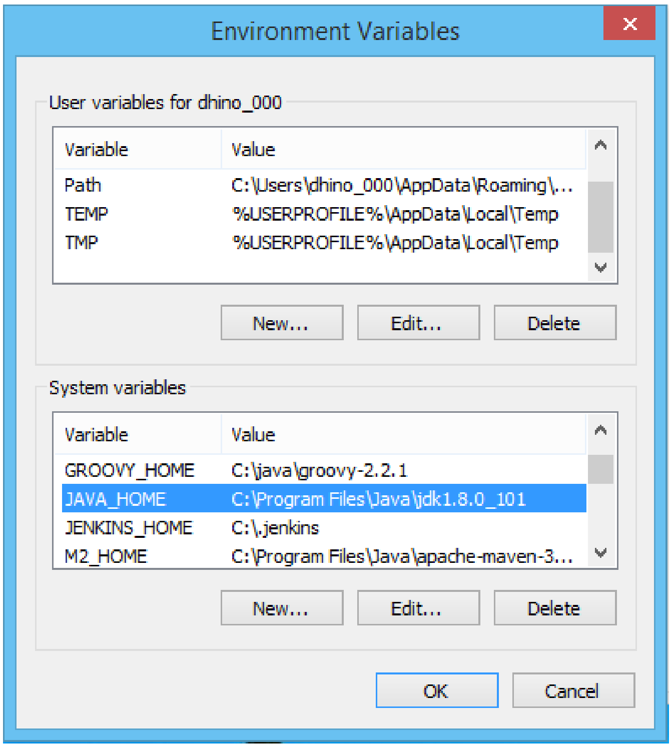
- Download *winutils.exe* from https://github.com/steveloughran/winutils/tree/master/hadoop-2.7.1/bin
- Place *winutils.exe* in a folder named *hadoop* anywhere you would like *C:\Program File\hadoop* or *C:\hadoop*.
- Note the location, since this will be your `HADOOP_HOME`

More about the installation at this link: https://hernandezpaul.wordpress.com/2016/01/24/apache-spark-installation-on-windows-10/

Windows Users Only: Setting up the Windows Environment Variables for Java

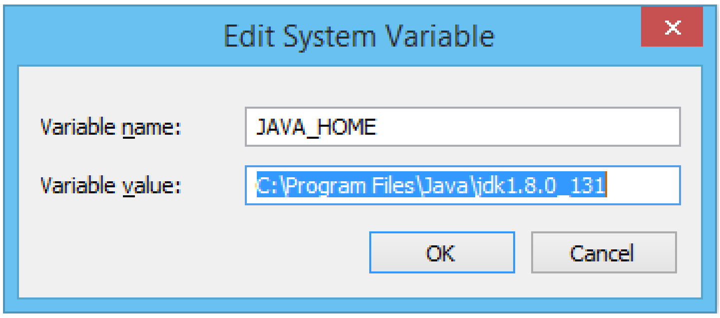
- Go to your *Environment Variables*, typically done by typing the Windows key() and type `env`





Windows Users Only: Setting up `JAVA_HOME`

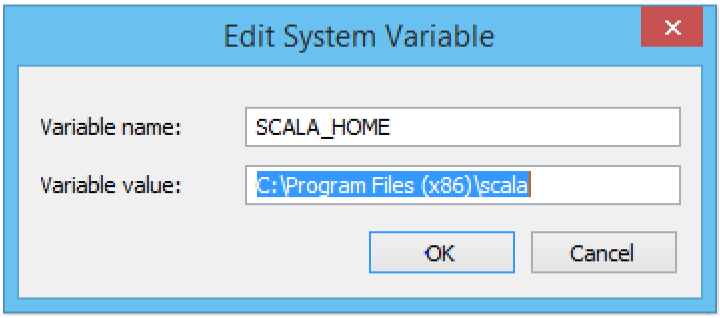
- Edit `JAVA_HOME` in the System Environment Variable window with the location of your JDK



Using `jdk1.8.0_131` in the image. Your version may vary.

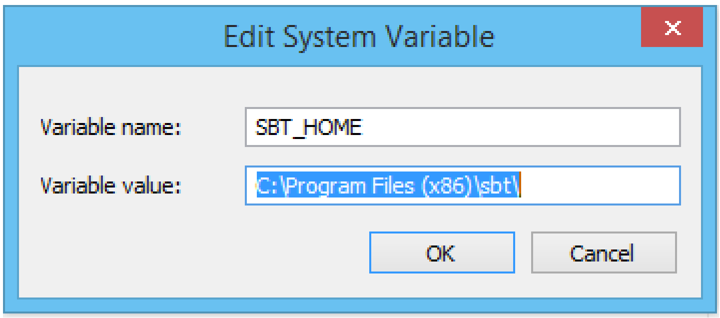
Windows Users Only (Optional): Setting up `SCALA_HOME`

- This setting is not necessary with Scala on Windows since the .msi file installs everything required
- If you do have problems where a tool is unable to locate Scala, set up an environment variable `SCALA_HOME`



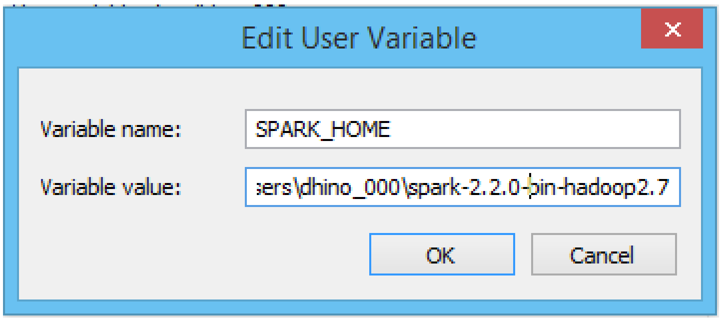
Windows Users Only: Setting up `SBT_HOME`

- This setting is not necessary since SBT on Windows since the .msi file installs everything required
- If you do have problems where a tool is unable to locate SBT, set up an environment variable `SBT_HOME`



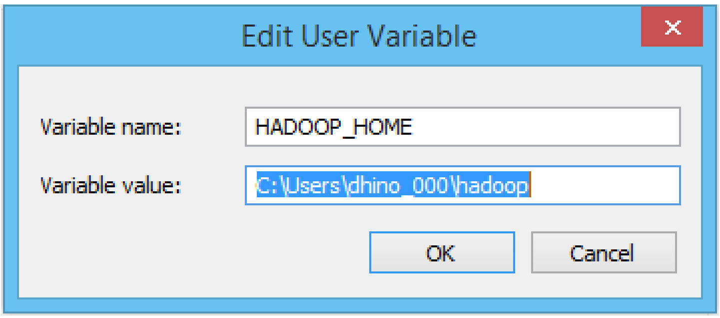
Windows Users Only: Setting up `SPARK_HOME`

- Set up an environment variable `SPARK_HOME` and setting it to the unpackaged spark folder from your download.
- **Do not include bin**
- **Do not use the `%USERPROFILE%` variable as it may cause side effects**



Windows Users Only: Setting up `HADOOP_HOME`

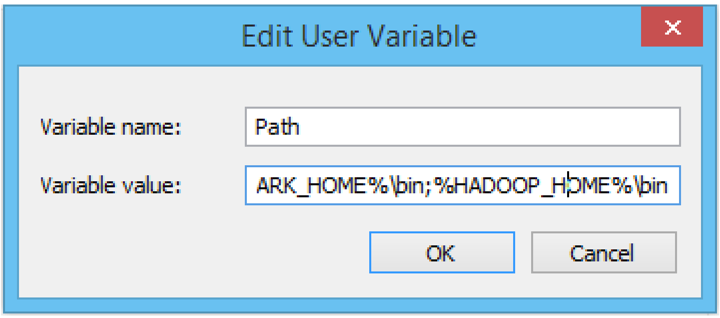
- Set up an environment variable `HADOOP_HOME` and setting it where you created your hadoop directory
- **Do not include bin**
- **Do not use the `%USERPROFILE%` variable as it may cause side effects**



Windows Users Only: Setting up `PATH`

- Once you establish `JAVA_HOME`, possibly `SCALA_HOME`, `SPARK_HOME`, `HADOOP_HOME`, ***append*** to the `PATH` setting the following:

```
;%JAVA_HOME%\bin;%SCALA_HOME%\bin;%SPARK_HOME%\bin;%HADOOP_HOME%\bin
```

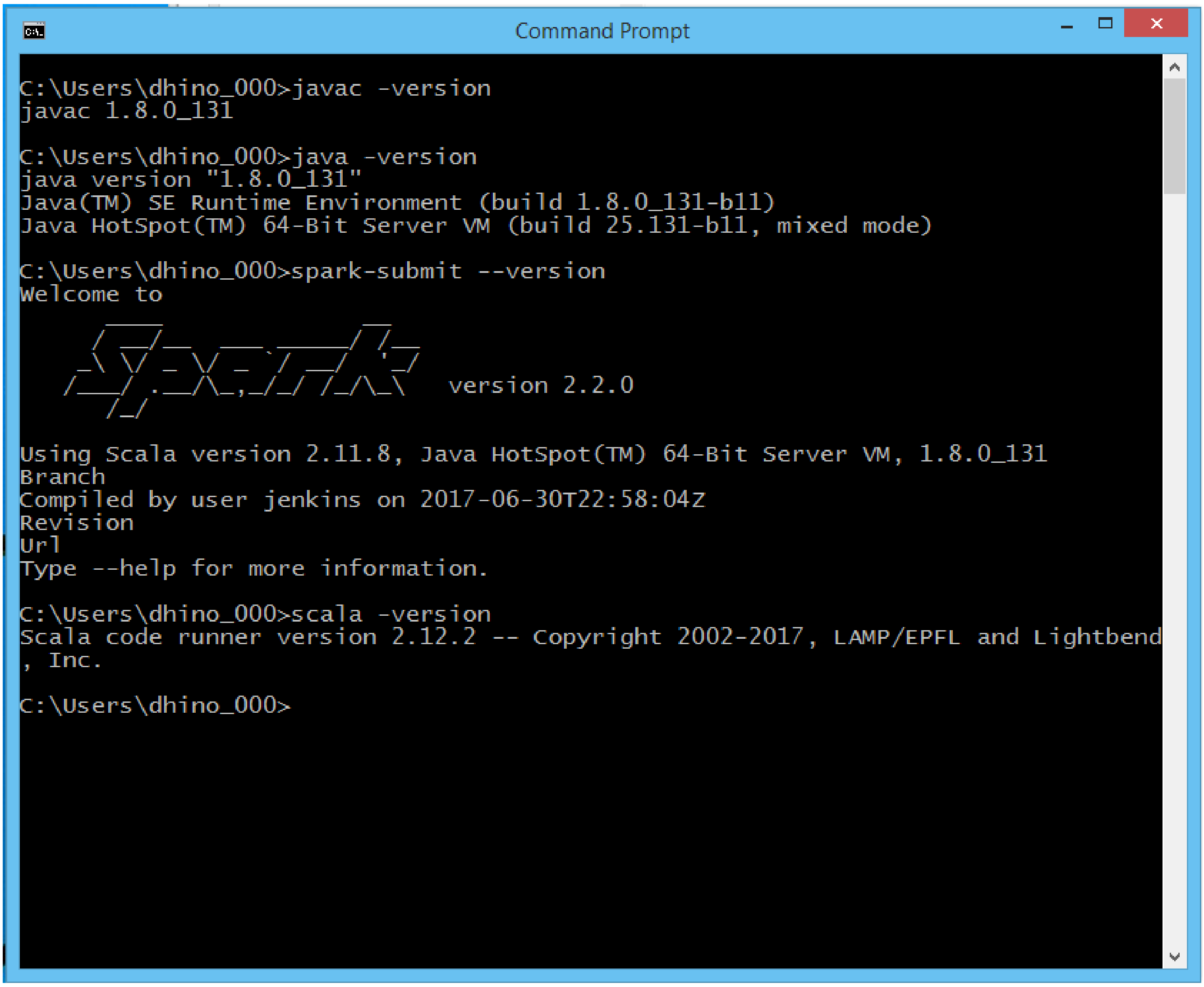


Windows Users Only: Permissions for the folder `C:\tmp\hive`

- Unfortunately, there will be issues with Windows users when the run `spark-shell`
- Attempt to run `spark-shell`
- Notice if you receive an error stating that there is not enough permission on `/tmp/hive`
- Use `winutils` to change the permission to `C:\tmp\hive` by using the command

```
winutils.exe chmod 777 \tmp\hive
```

Windows Users Only: Restart All Command Prompts And Try Again



Changes won't take effect until you open a new command prompt!

Mac Users Only: Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using nano

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

- Make sure the following contents are in your *.bash_profile*

- If you already have a `PATH`, append the new values to the end.

```
export SPARK_HOME= <location_of_spark>
export SCALA_HOME= <location_of_scala>
export SBT_HOME= <location_of_sbt>
export JAVA_HOME=$(/usr/libexec/java_home)
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$SBT_HOME
/bin:$SPARK_HOME/bin
```

If you used `brew`, many of these application will not require their `PATH` setup.

You can locate where `scala` and `spark` is by either doing

```
% which scala
% whereis scala
% which spark
% whereis spark
```

When done open a new terminal or if already on an open terminal type:

- For bash: **`source .bash_profile`**
- For zsh: **`source .zshrc`**

Linux Users Only: Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using `nano`

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

- Make sure the following contents are in your *.bash_profile*
- If you already have a `PATH`, append the new values to the end.

```
export SPARK_HOME= <location_of_spark>
export SCALA_HOME= <location_of_scala>
export SBT_HOME= <location_of_sbt>
export JAVA_HOME= <location_of_jdk>
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$SBT_HOME
/bin:$SPARK_HOME/bin
```

When done open a new terminal or if already on an open terminal type:

- For bash: `source .bash_profile`
- For zsh: `source .zshrc`

Spark Shell

Running the Spark Shell

- You can run the Spark Shell using `spark-shell` at the command line
- The spark shell provides access to a *SparkSession*
- *SparkSession*
  - Starting point to the execution of Spark
  - Can be retrieved by calling `spark` in the SparkShell

Lab: Starting up the Spark Shell

Step 1: Invoke the spark-shell for Scala in a terminal:

```
% spark-shell
```

Step 2: Verify that you get the `SparkSession` by calling `spark` in the Spark shell:

```
> spark
```

Step 3: Where you would get something like the following:

```
res1: org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@2a8081f5
```

Lab: Creating our first job

Step 1: Create some data with `spark.range` from 1 to 100 and process it with map with finally return a `DataFrame`

```
val dataFrame = spark.range(1, 100)
                     .toDF("mappedRange")
```

The above example creates data in a raw form and then puts it into a `DataFrame`

Lab: Evaluating a series of Data using Spark:

Step 1: Open up the spark shell.

Step 2: Enter the following which will create a range from 1 to 100, map, then filter, and then create a `DataFrame`

```
> val df = spark.range(1,100).map(x => x + 10).filter(x => x % 2 !=
0).toDF("numbers")

df: org.apache.spark.sql.DataFrame = [numbers: bigint]
```

Step 3: Next run `count` and this will evaluate the `count`. You may also see some extra process by doing so.

```
> df.count
res1: Long = 50
```

Lab: `show()` the data

`show()` will show the `DataFrame` by default of the first 20 rows

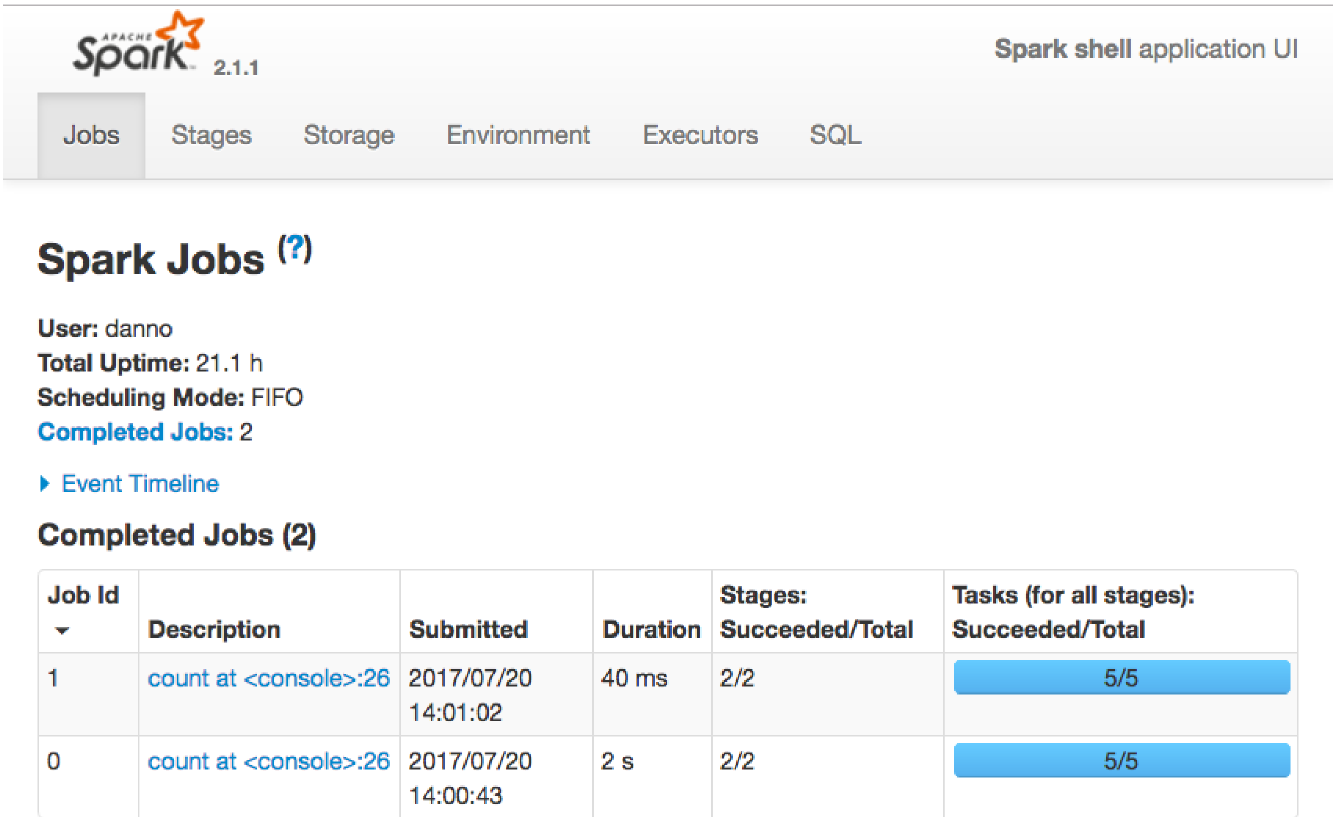
Step 1: Next `show()` the data to see what is left

```
> df.show()
```

Spark UI

- At anytime, you can go to the Spark UI for a local node setup by going to http://localhost:4040
- The Spark UI contains information about Spark:
  - Environment
  - Jobs
  - Cluster Configuration and Performance
  - Storage

Spark UI Example



Spark 2.1.1

Spark shell application UI

Jobs Stages Storage Environment Executors SQL

Spark Jobs (?)

User: danno
Total Uptime: 21.1 h
Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

Completed Jobs (2)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	count at <console>:26	2017/07/20 14:01:02	40 ms	2/2	5/5
0	count at <console>:26	2017/07/20 14:00:43	2 s	2/2	5/5

Opening Project

Lab: Getting started with a Spark Project

Step 1: Open IntelliJ within your virtual machine

Step 2: A project called ***spark-training*** should be open and already for you to use.

> To make this virtual machine light, only IntelliJ was chosen.

About the spark_training project

- Runs in SBT (Simple Build Tool)
- All Scala Based Project
- We will be running mostly tests using `ScalaTest`
- With the exception of the streaming projects

What is Scala?

- Multi-paradigm programming language
    - Functional
        - Every function is an object and a value
        - Capable of anonymous and higher order functions
    - Object Oriented
        - Everything can considered an object, including integers, floats
        - Inheritance through mixins and subclasses

Statically Typed Language

- Every value contains a type
- Expressive type system
- Types can be inferred
    - Cleaner
    - Less Physical Typing

Why bother?

- Contains all the features that Java 8 has now.
- Fully vetted by community
- Large Community
- Financial Backing
- It is good to learn a new language every year – The Pragmatic Programmer

What are the advantages of Scala?

- JVM Based
- Highly Productive Language
- Expressive Language
- Concise Language
    - Type Inference
    - No `return` required
    - No semicolons (`;`) required
- Above all, highly functional!

What are the disadvantages of Scala

- Hiring pool can be constrained
- Higher learning curve, until function programming becomes more prominent
- Non-backwards compatibility

Non-backwards compatibility

com.typesafe.akka	akka-actor_2.12	2.5.2	all (14)	24-May-2017
com.typesafe.akka	akka-actor_2.11	2.5.2	all (59)	24-May-2017

From: search.maven.org

Lab: Introduction to some Scala concepts

- `val` and `var`
- `class`
- `def` (methods)
- `object`

The Magical `apply` method

Considering the following code once again:

```
class Foo(x:Int) {
   def bar(y:Int) = x + y
}
```

If we run it, it would like this and return 50

```
val foo = new Foo(40)
foo.bar(10)
```

Replacing `bar` with `apply`

Let's take the previous code and use `apply` instead of `bar`:

```
class Foo(x:Int) {
   def apply(y:Int) = x + y
}
```

If we run it, it would like this and return 50

```
val foo = new Foo(40)
foo.apply(10)
```

Where thing are different, is that **apply is not required method call and you can leave the word out!**

Therefore...since we used apply it looks like this:

```
val foo = new Foo(40)
foo(10)
```

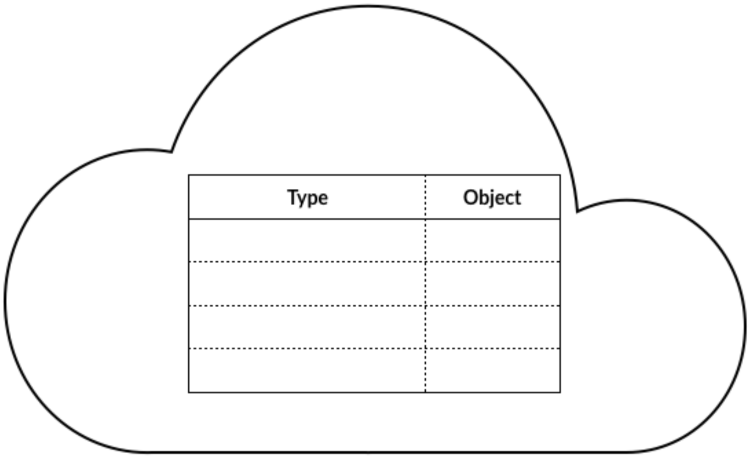
Conclusion

- If the method is called apply, *no matter where it was defined*, you can leave the explict call out!
- This is probably one of the most important aspects to the language that few know about since it too many it is too obvious too mention

`implicit`

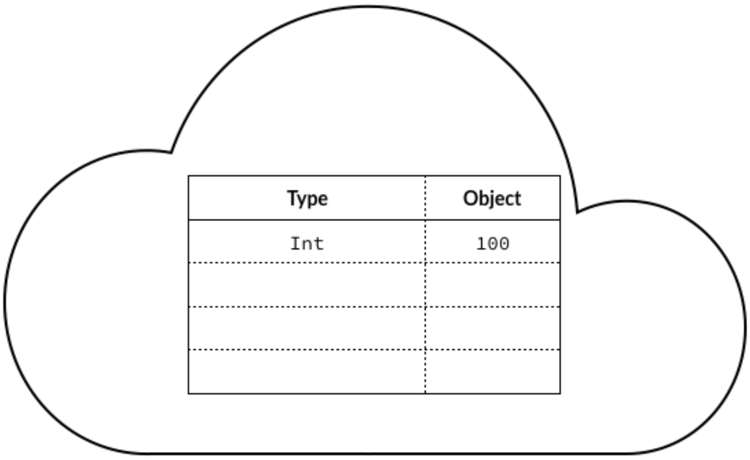
`implicit`

- `implicit` is like an invisible `Map[Class[A], A]` where `A` is any object and it is tied into the scope
- Whatever type is required the object that it corresponds to that type will be injected automatically.



`implicit`

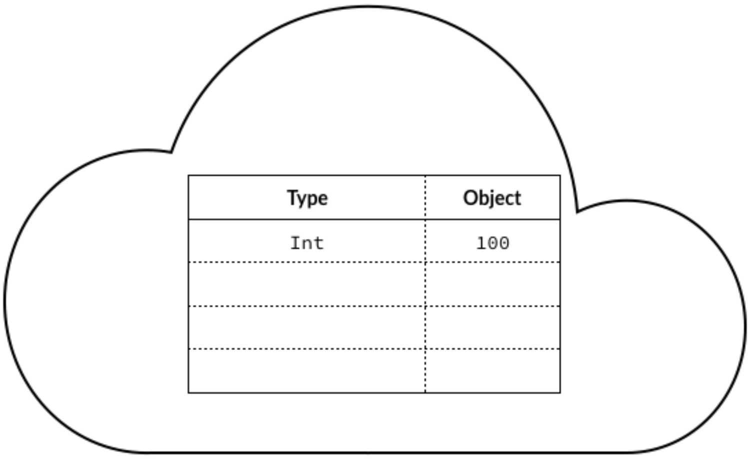
- `implicit` in this case bounds an `Int` of `100`
- Once established we can call upon the `implicit` binding to a binding parameter group




```
implicit val hourlyRate = 100

def calcPayment(hours: Int)(implicit rate: Int) = hours * rate

calcPayment(50) should be(5000)
```

Overriding an `implicit` manually

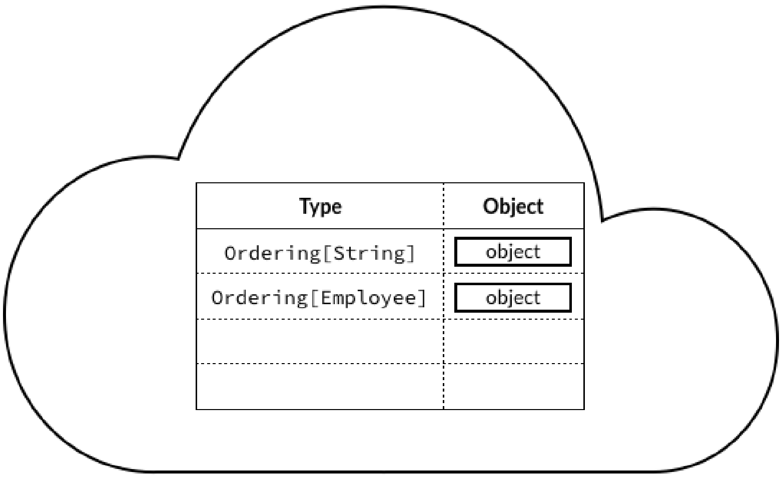
- You can always override the any `implicit` manually




```scala
implicit val hourlyRate = 100

def calcPayment(hours: Int)(implicit rate: Int) = hours * rate

calcPayment(50)(200) should be(10000)
```

Setting up an `implicit` for `Ordering[T]`

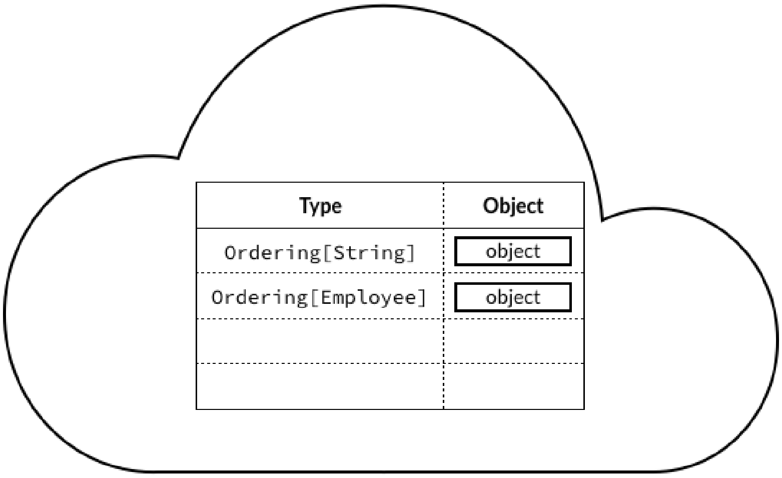
- You can establish an `implicit` for ordering inside of a collection or any construct with `Ordering[T]`
- This is also known as a *Type Class*



```scala
case class Employee(firstName:String, lastName:String)

implicit val employeeOrderingByLastName: Ordering[Employee] =
  new Ordering[Employee] {
    override def compare(x: Employee, y: Employee): Int = {
      x.lastName.compareToIgnoreCase(y.lastName)
    }
  }
```

Applying the `implicit` for `Ordering[T]`

Once this `implicit` is established there is an `implicit` bound on how to order an `Employee`




```scala
List(new Employee("Eric", "Clapton"),
        new Employee("Jeff", "Beck"),
        new Employee("Ringo", "Starr"),
        new Employee("Paul", "McCartney"),
        new Employee("John", "Lennon"),
        new Employee("George", "Harrison")).sorted
```

Which yields the result:

```scala
List(new Employee("Jeff", "Beck"),
        new Employee("Eric", "Clapton"),
        new Employee("George", "Harrison"),
        new Employee("John", "Lennon"),
        new Employee("Paul", "McCartney"),
        new Employee("Ringo", "Starr"))
```

In order to avoid any conflicts, it is up to you, the programmer, to decide in what scope `implicit` are applied

Why did `implicit Ordering[Employee]` work?

- Here is the Scala API signature for `sorted`
- Notice the `implicit` definition in the method signature
- It requires that an implicit bound be available in order to process

```
def sorted[B >: A](implicit ord: math.Ordering[B]): List[A]
```

Sorts this sequence according to an Ordering.

The sort is stable. That is, elements that are equal (as determined by lt) appear in the same order in the sorted sequence as in the original.

ord	the ordering to be used to compare elements.
returns	a sequence consisting of the elements of this sequence sorted according to the ordering ord.

Definition Classes	SeqLike
See also	scala.math.Ordering

Running a Spark Job

Lab: Running a Spark Instance from Docker

Step 1: From the ***spark-training*** folder locate the ***docker*** folder and `cd` into it

```
spark-training % cd docker
```

Step 2: In one terminal window, run the combination of the spark master and two node by running `docker-compose`

```
spark-training/docker % docker-compose up
```

Step 3: In a new terminal window, run sbt in the ***spark-training*** project, `cd ..` if you have to.

```
spark-training % sbt
```

Step 4: Once in the SBT terminal, activate the ***app*** project by invoking `project app` within SBT

```
> project app
```

- Then, create an all in one jar file, with the `assemble` task

```
> assembly
```

Lab: Submitting the Spark Uberjar

- Open yet another terminal, and enter the following in the root of the ***spark_training*** application.

```
% spark-submit \
spark-submit \
  --master spark://localhost:7077 \
  --class com.xyzcorp.SparkPi \
  spark-app/target/scala-2.11/app-assembly-1.0-SNAPSHOT.jar
```

- Analyze the Results
- Stop the Docker Compose Container

```
% docker-compose down
```

Value Types

Spark Value Types

- Again, all of Spark is based on types
- To work with types in Scala, you must import `import org.apache.spark.sql.types._`
- To work with types in Java, you must import `import org.apache.spark.sql.types.DataTypes`

Scala Table of Types

Spark Type	Scala Value Type	Scala API
`ByteType`	`Byte`	`ByteType`
`ShortType`	`Short`	`ShortType`
`IntegerType`	`Int`	`IntegerType`
`LongType`	`Long`	`LongType`
`FloatType`	`Float`	`FloatType`
`DoubleType`	`Double`	`DoubleType`
`DecimalType`	`java.math.BigDecimal`	`DecimalType`
`StringType`	`String`	`StringType`
`BinaryType`	`Array[Byte]`	`BinaryType`
`TimestampType`	`java.sql.Timestamp`	`TimestampType`
`DateType`	`java.sql.Date`	`DateType`
`ArrayType`	`scala.collection.Seq`	`ArrayType(elementType, [valueContainsNull]) **`
`MapType`	`scala.collection.Map`	`MapType(keyType, valueType, [valueContainsNull]) **`
`StructType`	`org.apache.spark.sql.Row`	`StructType(Seq(StructFields)) *`
`StructField`	`StructField with DataType contents.`	`StructField(name, dataType, nullable)`

DataFrames

Defining `DataFrame`

- Table of data with rows and columns
- The list of columns and the types are called *schemas*
- **Important** A spark data frame can span multiple machines.
- The distribution for `DataFrame` on multiple machines is for performance
- The *partitioning scheme* is how the data is broken and can either be by:
  - column
  - non-deterministically

`DataFrame` is Transformable

- Due to the `DataFrame` not actually holding data they are transformable
- You can:
  - Remove columns
  - Turn a column to a row
  - Turn a row into a column
  - Add columns
  - Add rows
  - Sort by columns
  - Sort by rows

`DataSet`

`DataSet` Overview

- An extension of the DataFrame API that provides a type-safe, object-oriented/functional programming interface
- Datasets take advantage of Spark’s Catalyst optimizer by exposing expressions and data fields to a query planner
- Strongly-typed, immutable collection of objects that are mapped to a relational schema
- Much more "performant" than a raw RDD counterpart in terms of processing speed and memory usage

Spark SQL

About Spark SQL

- Spark SQL execute queries against views and tables organized into databases
- Use system functions, define user functions, analyze query plans
- Spark SQL implements a subset of ANSI SQL :2003
- Competes against Hive, which is Facebook's implementation of Big Data SQL
- Facebook started putting efforts behind Spark SQL

Key facts about Spark SQL

- SQL analysts can leverage Spark's computation abilities
- Data Scientists can use Sparks SQL interface to derive data
- Use of the `sql` method on the `SparkSession` object

Lab: Testing the samples of Spark SQL

Step 1: In a new terminal window, run sbt in the ***spark-training*** project if you haven't already.

```
spark-training % sbt
```

Step 2: Once in the SBT terminal, activate the ***api*** project by invoking `project api` within SBT

```
> project api
```

Step 3: Compile all the test and ensure we are ready to go.

```
> test:compile
```

Step 4: Run any of the tests by running something like the following.

```
> testOnly com.xyzcorp.SparkSQLSpec -- -z "Case 1:"
```

////TODO: Be sure that people can run this

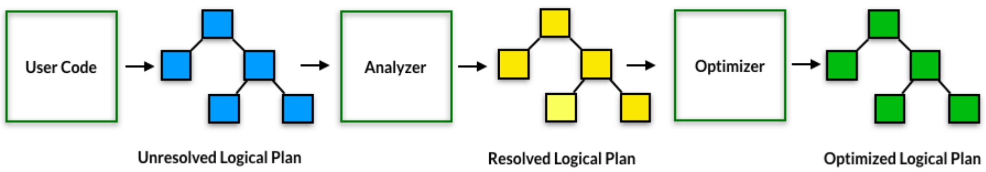
Plans

Understanding Plans

The ordering of how Spark operates is as follows:

- Write DataFrame/Dataset/SQL Code:
    - If valid code, Spark converts this to a Logical Plan
    - Spark transforms this Logical Plan to a Physical Plan
    - Spark then executes this Physical Plan on the cluster

Plans Diagram



- Unresolved Logical Plan - Taking the code and and creating a plan without consideration to actual table data
- Resolved Logical Plan - Takes the unresolved logical plan, and the *catalog* of actual data and analyzes it into a *resolved logical plan*
- Optimized Logical Plan - After it is resolved, it uses an optimizer to determine the best course to aggregate and operate on the data

Physical Planning

- After optimization of the plan, comes the physical planning
- This is called the *Spark Plan*
- Specifies how and where the Optimized Logical Plan will run by analyzing:
    - Costs
    - Best Physical Plan
- Becomes a series of RDD (Resilient Distributed Datasets) and Transformations

Explaining Plans

- With `DataFrame`/`Dataset`/`Spark SQL`, you can run:

```
val df:DataFrame = ...
df.explain(true) //true will provide a full display of optimization
```

Resilient Distributed Dataset

Defining RDD

- Resilient distributed dataset (RDD)
- Fault-tolerant collection of elements that can be operated on in parallel.
- Can be represented by
  - parallelized collections
  - externalized datasets
- Originally how Spark had always been processed without Catalyst Optimizations

Broadcast Variables

- Intend to share an immutable value efficiently around the cluster.
- Avoids having to serialize and deserialize within a function
- Are shared, immutable, cached on every machine

Accumulators Defined

- Variables that are accumulated by an associative and commutative operation (like addition)
- Accumulation is done in parallel
- Perfect for counters or debugging
- The can be named or unnamed
- They *do not change the lazy evaluation of Spark*
- Supports numbers out of the box, but can also support custom types

Spark Streaming

Streaming

- Many companies have no embraced streaming as a solution for real-time processing
- Streaming APIs can handle terabytes of data in small time span
- Continuous operation of data
- Wide range of applications
    - Credit Card Transactions
    - Fraud Detection
    - Sensors and IoT (Internet of Things)
    - System Monitoring

Spark Streaming Overview

- Spark Streaming is backed by DStreams
- DStreams
    - Represent a continuous stream of data
    - Stands for *Discretized Stream*
    - Can have input from Kafka, Flume, or Amazon Kinesis
    - Implemented by RDDs, which are a lower level API
    - Time based
    - Considered Stable as of Spark 2.2

Spark Streaming APIs

- Meant to be *simple*
- Micro-batch API
    - Lower level API
- Structured Streaming API
    - Higher level optimization

Grabbing the library

```
"org.apache.spark" % "spark-streaming_2.11" % "2.2.0"
```

Various Sources

- **Basic sources:** Sources directly available in the `StreamingContext` API
    - File systems
    - Socket connections
- **Advanced sources:** Extra Utility Classes, require extra downloading. Sources like
    - Kafka
    - Flume
    - Kinesis

Grabbing the specialized advanced libraries

```
"org.apache.spark" % "spark-streaming-kafka-0-10_2.11" % "2.2.0"
```

```
"org.apache.spark" % "spark-streaming-flume_2.11" % "2.2.0"
```

```
"org.apache.spark" % "spark-streaming-kinesis-asl_2.11" % "2.2.0"
```

Establishing a Stream Connection

- Establish a `StreamingContext` with a `SparkConf`
- The establish a `TimeUnit` that listens to the Stream based framework (Kafka, Flume, Kinesis)

```
val conf = new SparkConf().setMaster(master).setAppName("appname")
val ssc = new StreamingContext(conf, Seconds(1))
```

`DStream` under the hood

- Basic Abstraction of Spark Streaming
- Used to transform the Stream
- Internally is a series of RDDs
- Operates in microbatch
- With the exception of a `FileStream`, every `DStream` is associated with a *receiver*

Sources

- Sockets
- Distributed File Systems (S3, HDFS)
- Kafka

Sinks

- Functions
- Consoles
- Memory
- Kafka
- Distributed File Systems (S3, HDFS)
- Non-Distributed File Systems

Output Mode in Structured Streaming

- How do we wish to write information into the sink
  - Append - Append information to the end
  - Update - Update records in place
  - Complete - Rewrite the entire output

> Some outputs only operate on certain output modes

Triggers

When do we want to input data, we have two choices:

- When ever the data is processed and we need a new batch
  - Advantage: Low latency
  - Disadvantage: Writing many small files
- Based on a processing time (time interval)
  - Advantage: Manipulating Larger Sets of Data
  - Disadvantage: Higher latency

Event Time

- Data in your source that determines when data was created or update

- Structured streaming uses that data for processing rather the *system processing time*

Watermarks

- How late to see the data in the event time
- How long should the data be considered for calculations

Closing the connection

```
ssc.start()                // Start the computation
ssc.awaitTermination()     // Wait for the computation to terminate
```

Stateful Processing

- Storing State from an API
- Many state operation in Spark are automatic
- Stateful operations in Spark using an internal *state store*

Receiver Reliability

- Reliable Receiver - The receiver can acknowledge the data when received by messaging systems that have or require acknowledgment
  - Apache Kafka
  - Apache Flume
- Unreliable Receiver - The receiver does not acknowledge the source that data is received, or perhaps optionally do not want to acknowledge

Window Operations

As time goes on you may want to operate on a subset of data.

Checkpointing

- Checkpointing is used to backup information as it is being processed
- Can store information about where it was before it crashed
- Can store information based on a stateful transformations as it is continuing to process
- To configure checkpointing:

```
val ssc = new StreamingContext(...)    // new context
ssc.checkpoint(checkpointDirectory)
```

Lab: Listening to a Web Socket

Step 1: On the terminal, run the following:

```
nc -lk 10150
```

Step 2: In a new terminal window, run sbt in the ***spark-training*** project.

```
spark-training % sbt
```

Step 3: Once in the SBT terminal, activate the ***streaming*** project by invoking `project streaming` within SBT

```
> project streaming
```

Step 4: Each of the examples can be run in SBT by using `run`

```
> run
```

Netcat (`nc`) is a simple Unix utility that reads and writes data across network connections, using the TCP or UDP protocol

Step 2: == Parallelization

- To parallelize `DStreams`, you will need to create multiple inputs
- This will also require multiple receivers
- A Spark worker/executor is a long-running task, hence it occupies one of the cores allocated to the Spark Streaming application.
- Therefore, an application needs to be allocated enough cores
- **You will need more than one thread**: `local`, `local[1]` will not suffice

Setting the Right Batch Interval

For a Spark Streaming application running on a cluster to be stable:

- The system should be able to process data as fast as it is being received.
- Batches of data should be processed as fast as they are being generated.

- Whether this is true for an application can be found by monitoring the processing times in the streaming web UI, where the batch processing time should be less than the batch interval.
- So the batch interval needs to be set such that the expected data rate in production can be sustained.
- Start slow and titrate with a faster rate
- Check the logs to see if it gets progressively worse over time.

Graph X

About Graphs and Graph Parallel Computation

- Graphs and Graph Parallel Computation
- *Nodes*
  - *Vertices* which are just objects
  - *Edges* that describe the relationships between *nodes* and *vertices*
- Data is stored in both the vertices and the edges

About Graph Analytics

- Graph Analytics is the process of analyzing these relationships
- Graphs can be used to determine relationships between vertices with a weight

Directed and Undirected Graphs

- A *directed* graph has a direction between *vertices*
- An *undirected* graph does not have a direction between *vertices*

Standard Imports

```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```

Establishing Vertices

- Vertices are the endpoint data themselves
- These vertices are created in an `RDD`

```
val users: RDD[(VertexId, (String, String))] =
  sparkContext.parallelize(Array(
    (3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
    (5L, ("franklin", "prof")), (2L, ("istoica", "prof")),
    (4L, ("peter", "student"))))
```

Establishing Relationships

- The relationships are the edges and how they tie the vertices together
- The combination of left vertice, the edge, and the right vertice is a *triplet*

```
val relationships: RDD[Edge[String]] =
  sparkContext.parallelize(Array(
    Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
    Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"),
    Edge(4L, 0L, "student"),   Edge(5L, 0L, "colleague"))
  )
```

Creating a Graph

- Graphs using the data can be created

```
val defaultUser = ("John Doe", "Missing")
val graph = Graph(users, relationships, defaultUser)
```

Using functional programming to view the existing graph

```
graph.vertices.collect.foreach(println(_))
```

`collect` in this example just returns an `Array` with all the elements

Lab: Run a Sample Graph

Step 1: Be sure that in SBT you are in the `graphx` project:

```
sbt:spark-training> project graphx
```

Step 2: Compile the tests

```
sbt:graphx> test:compile
```

Step 3: Run the test with the test case you just wrote

```
sbt:graphx> testOnly com.xyzcorp.SparkGraphXSpec -- -z "Case 1:"
```

Page Rank

- Graph Algorithm created by Larry Page from Google
- Basic Formula:
  - Count number and quality of links to a page
  - More important websites receive more links

Lab: Write some users for page rank

Step 1: In the ***spark-graphx*** subproject, and in the *src/test/resources* folder open the *users.txt* file **Step 2:** Create some notable people. The first column is an integer id, the second is the person's name, and third is the person's title. Separate them by a column.

```
1,Chloe Kim, Olympic Snowboarder
2,James Gosling, Java Creator
3,Elon Musk,Tesla CEO
4,Mary Barra,GM CEO
6,Majora Carter,Activist
7,Nora Roberts,Author
8,anononymous,Anonymous
```

Lab: Write some relationships

Step 1: In the ***spark-graphx*** subproject, and in the *src/test/resources* folder open the *followers.txt* file **Step 2:** Create some relationships using the notable people you just created. **Step 3:** Determine who follows who in the same way Facebook and Twitter will follow each other. It will look something like the following, but make your own

```
2 1
4 1
1 2
6 3
7 3
7 6
6 7
3 7
```

Lab: Writing the code to evaluate the page rank

Step 1: In the ***spark-graphx*** subproject, and in the *src/test/scala* folder, locate and open *SparkGraphXSpec.scala*

Step 2: Locate test case that starts `"Case 5: Running followers and users and determine the weight…"`

Step 3: Start with setting up the file locations you just entered:

```
val followersPath = getClass.getResource("/followers.txt").getPath
val usersPath = getClass.getResource("/users.txt").getPath
```

Step 4: Create an edge list file with the followers information

```
val graph = GraphLoader.edgeListFile(sparkContext, followersPath)
```

Lab: Continuing create a Graph

Step 1: Continuing in the same file *SparkGraphXSpec.scala*, create a page rank on the graph using ε which is called the reset probability (0.0001) which is used in Page Rank computation. See https://en.wikipedia.org/wiki/PageRank for more details

```
val ranks = graph.pageRank(0.0001).vertices
```

Step 2: Read in the users and create a tuple of the users

```
val users = sparkContext.textFile(usersPath).map { line =>
val fields = line.split(",")
   (fields(0).toLong, fields(1)) //A tuple of 2
}
```

Lab: Continuing by joining the users and their page ranks.

Step 1: Bringing it all together lets join the users and ranks

```
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank) //Pattern Matching
}
```

Step 2: Print the results

```
println(ranksByUsername.collect().mkString("\n"))
```

Tuples, and Pattern Matching, `mkString` are one of the many reasons why using Scala is preferable to Java when it comes to Spark

Lab: Running the results

Step 1: Be sure that in SBT you are in the `graphx` project:

```
sbt:spark-training> project graphx
```

Step 2: Compile the tests

```
sbt:graphx> test:compile
```

Step 3: Run the test with the test case you just wrote

```
sbt:graphx> testOnly com.xyzcorp.SparkGraphXSpec -- -z "Case 6:"
```

In case the lab didn't work:

- Here are some of the imports that were required

```
import org.apache.spark.graphx.{Edge, Graph, GraphLoader, VertexId}
import org.apache.spark.rdd.RDD
```

Conclusion

Apache Spark Rundown

- Uses multiple machines on multiple cores
- Supports multiple languages
- Separates work by partitions, stages, and tasks to perform work
- Contains a wealth of APIs
  - Spark Streaming
  - Spark MLLib
  - Spark GraphX

What do we look forward to.

- Spark is a preferred framework today
- Spark may or may not be the preferred framework in the future
- Be on the lookout for competitors in this space

What should be in your knowledge portfolio?

- Understand Big Data
- Why big data aggregation and analysis works for companies big and small
- We accumulate data faster and en masse, storage is cheap.
- Every company and strategy may be different based on the data.

Contact Me

Daniel Hinojosa

Twitter: @dhinojosa

Email: dhinojosa@evolutionnext.com