Advanced Spark

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)
```

Changes in Spark 2.0

Notable Changes in Spark 2.0

- Unifying `DataFrame` and `DataSet`, `DataFrame` is a `DataSet[Row]`
- `SparkSession` created to replace old `SQLContext` and `Hive Context`
- Better `Accumulator` API
- Native CSV Source
- Improved Parquet and ORC Performance
- Improved SQL Handling
- `DataFrame` is now the primary API
- Structured Streaming using the same sources and sinks as the `DataFrame`/`DataSet` API

`SparkSession` in Spark 2.0

- `SparkSession`
  - We were earlier developing `SparkContext` and `SqlContext` separately but in the Spark 2.0 migrate to `SparkSession` which contains:
    - The `Dataset` API
    - The `DataFrame` API.
  - We can get `SparkContext` and `SqlContext` both in the `SparkSession`
    - No need for a `SqlContext`

`DataFrame` is alias for `DataSet[Row]`

In the Spark API, the `DataFrame` is now an alias for:

```
type DataFrame = Dataset[Row]
```

SparkSQL

- SparkSQL updated to SQL2003 support
- Improved Parsing
- Subquery Support with:
  - Uncorrelated Scalar Subqueries
  - Correlated Scalar Subqueries
  - NOT IN predicate Subqueries (in WHERE/HAVING clauses)

    - IN predicate subqueries (in WHERE/HAVING clauses)
    - (NOT) EXISTS predicate subqueries (in WHERE/HAVING clauses)

Streaming

- Spark 2.0 ships the initial experimental release for Structured Streaming,
- A high level streaming API built on top of Spark SQL and the Catalyst optimizer.
- Structured Streaming enables users to program against streaming sources and sinks using the same DataFrame/Dataset API as in static data sources, leveraging the Catalyst optimizer to automatically incrementalize the query plans.

Important Items Added and Removed

- Scala 2.11 now based
- Removed Support for Java 7

Review of Scala

Any harder concepts in Scala to review

- Particularly those concepts that are relevant to Spark:

Setup

Setup

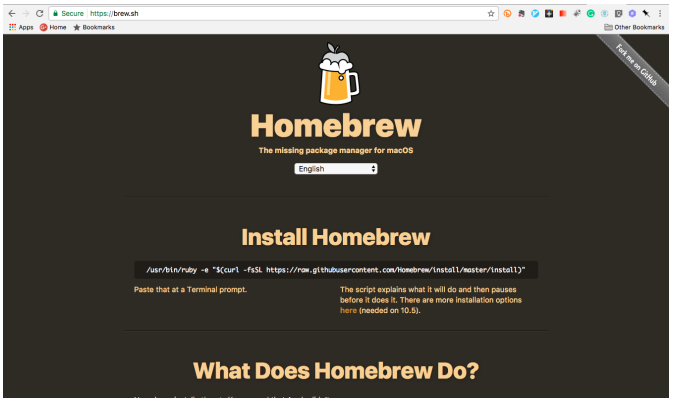
Before we begin it is assumed that all of you have the following tools installed:

- JDK 1.8 (latest java is 1.8.0_144)
- Scala 2.12.3
- SBT 1.0.2
- Spark 2.2.0
- `winutils` (Windows Only)

To verify that all your tools work as expected

```
% javac -version
javac 1.8.0_144

% scala -version
Scala code runner version 2.12.3 -- Copyright 2002-2017, LAMP/EPFL

% java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_1.8.0_144-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 2.2.0
      /_/

Using Scala version 2.12.3, Java HotSpot(TM) 64-Bit Server VM, 1.8.0_144
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

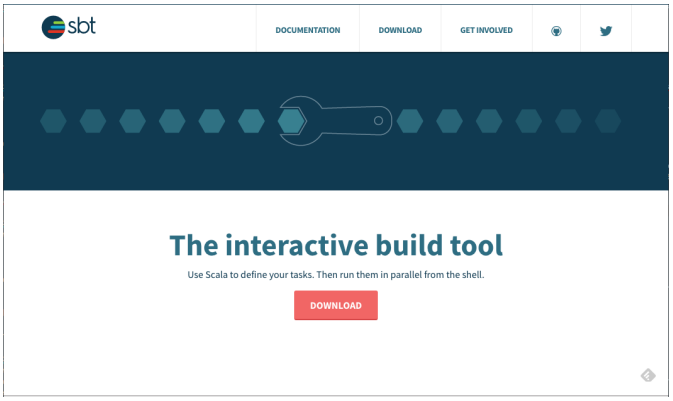
- Visit: http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html
- Select: *Accept License Agreement*
- Download the appropriate Java version based on your architecture.

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

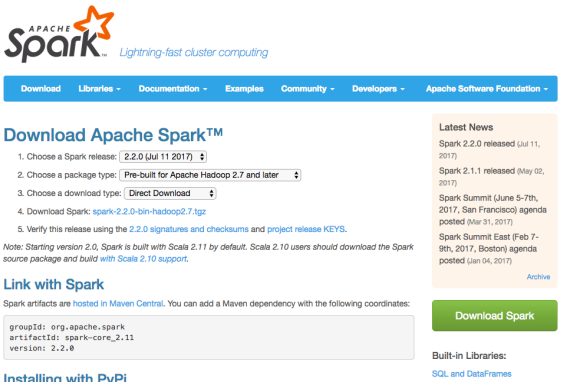
- Visit http://scala-lang.org
- Click the *Download* Button
- Download the appropriate binary for your system:
  - Mac and Linux will load a *.tgz* file
  - Windows will download an *.msi* executable
- For Mac and Linux you can expand with `tar -xvfz scala-2.12.3.tgz`

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-2.2.0-bin-hadoop2.8.1.tgz link to download
- For Mac and Linux, you can expand with `tar -xvfz spark-2.2.0-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

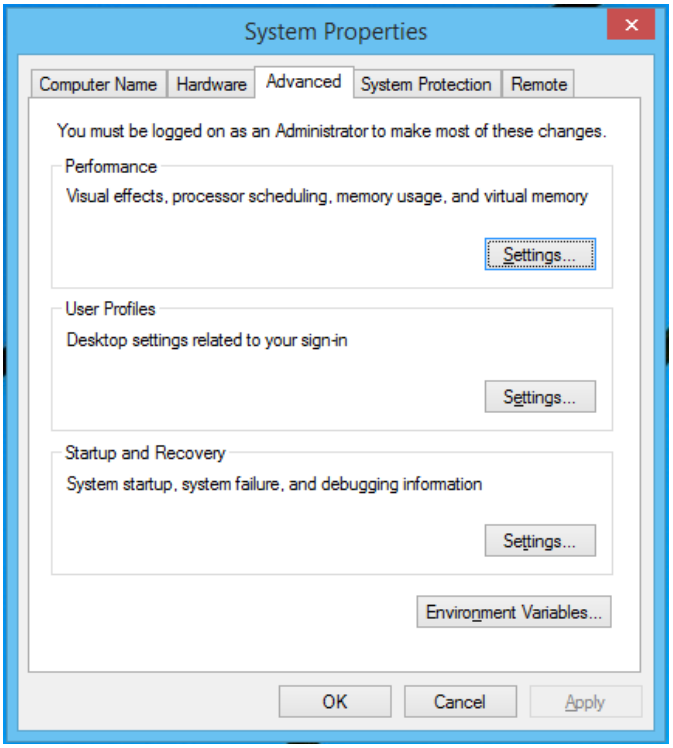
Windows Users Only: Download `winutils`

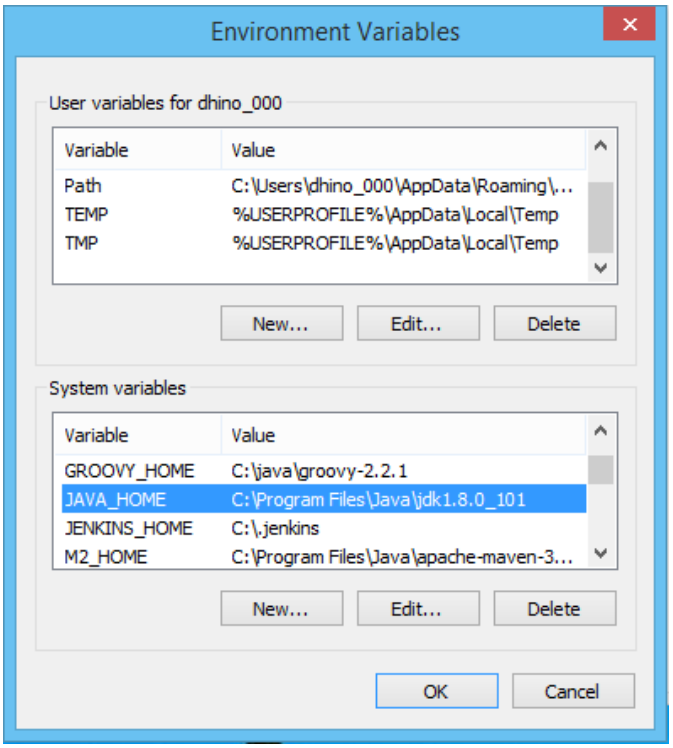
- Download *winutils.exe* from https://github.com/steveloughran/winutils/tree/master/hadoop-2.7.1/bin
- Place *winutils.exe* in a folder named *hadoop* anywhere you would like *C:\Program File\hadoop* or *C:\hadoop*.
- Note the location, since this will be your `HADOOP_HOME`

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

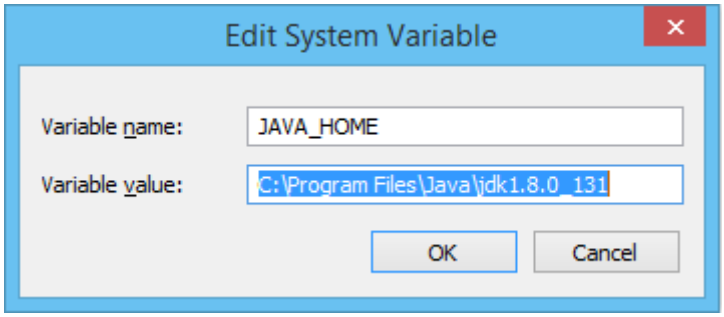
- Go to your *Environment Variables*, typically done by typing the Windows key(⊞) and type `env`





Windows Users Only: Setting up `JAVA_HOME`

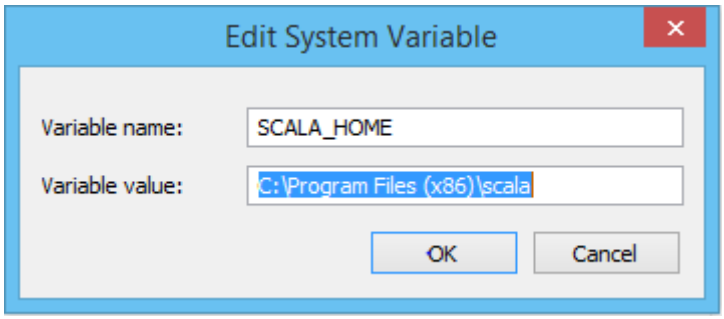
- Edit `JAVA_HOME` in the System Environment Variable window with the location of your JDK



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

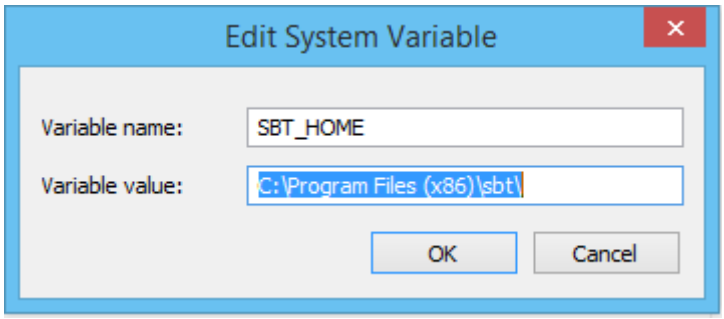
Windows Users Only (Optional): Setting up SCALA_HOME

- 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`



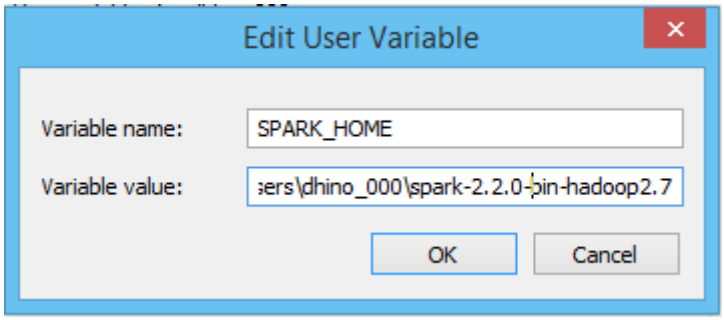
Windows Users Only: Setting up SBT_HOME

- 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`



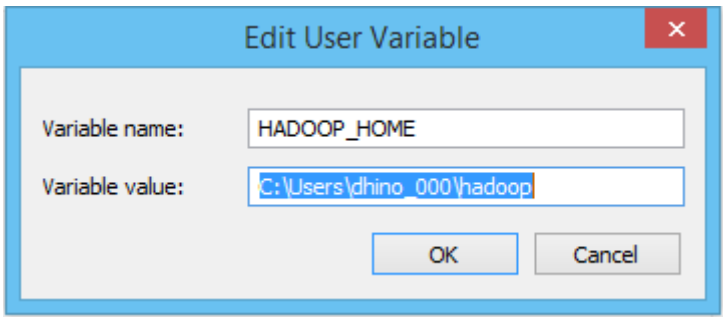
Windows Users Only: Setting up SPARK_HOME

- 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

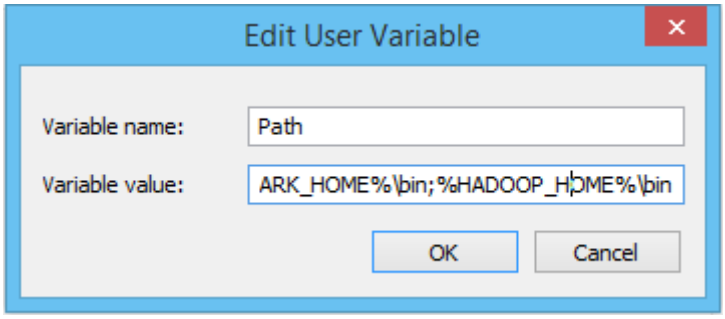
- Set up an environment variable `HADOOP_HOME` and setting it where you created your hadoop directory
- **Do not include bin**
- **Do not use the `%USERPROFILE%` variable as it may cause side effects**



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
```

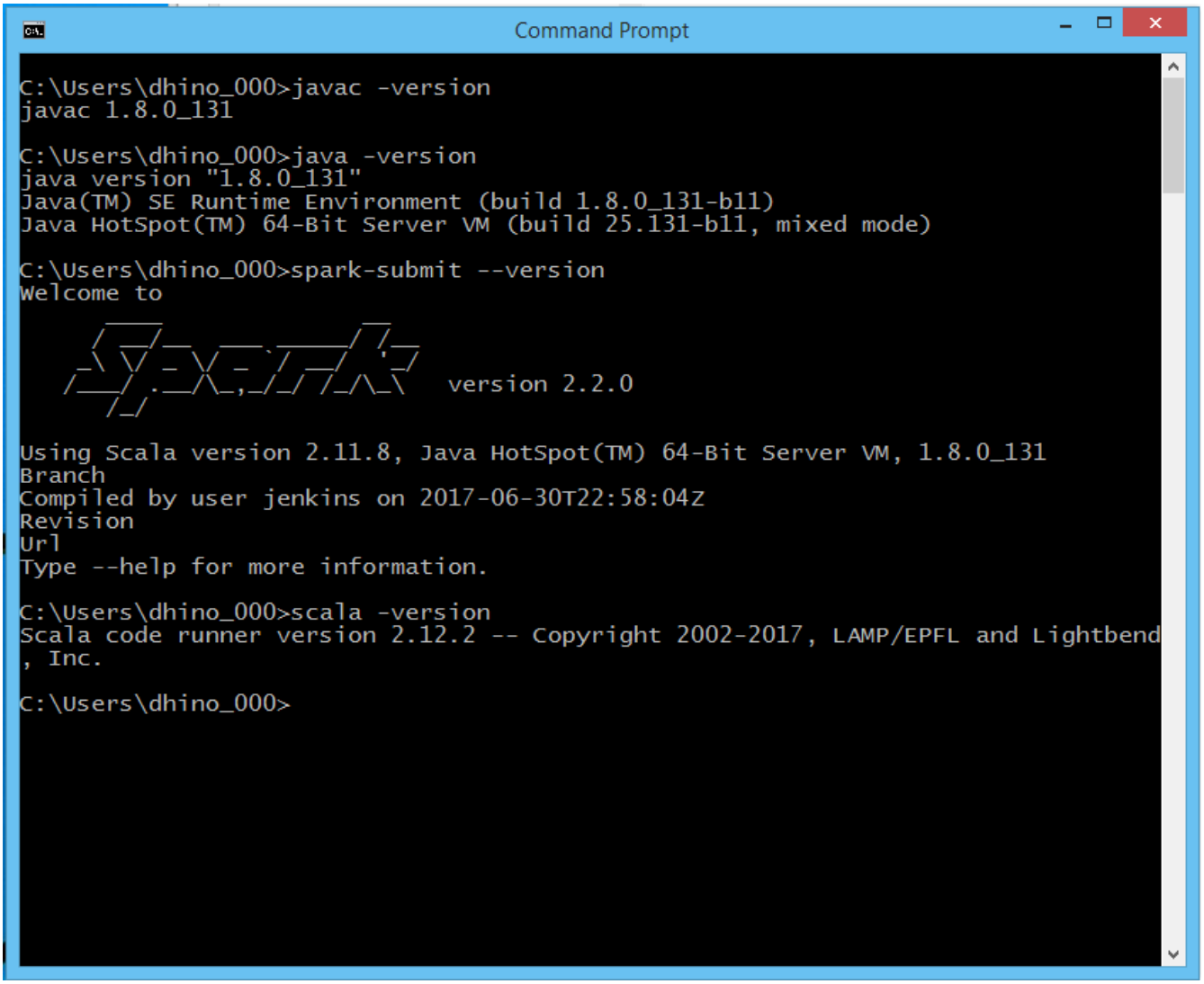


Windows Users Only: Permissions for the folder `C:\tmp\hive`

- 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`**

Review Abstractions

Overview of Abstractions

The following are the main abstractions of Spark

- DataFrames
- Datasets
- SQL Tables
- Resilient Distributed Datasets

DataFrames

- Are the most efficient
- 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

Partitions

- For management, Spark breaks up data into chunks
- A Partition is a collection of *rows* that sit on *one machine* in a cluster
- Therefore a DataFrame contains 0 or more partitions
- DataFrame is the interface to all the computations and data stored on remote machines
- In local mode they are laid across a single instance

Partition parallelism

- Partitions are operated on in parallel
- Unless they undergo a process called *shuffling*

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

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`

Rows and Columns

- Dataframes are described as rows and columns
- Rows and columns are established as objects in Spark

Columns

- Embodied in the API as a `Column` type

Rows

- Embodied in the API as a `Row` type

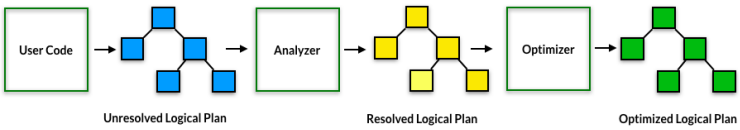
Plans

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

Lab: Explain all the plans

Step 1: Using spark-shell, call `explain(true)` to explain `badDays`

Step 2: View the analysis which should look like the following

```
scala> badDays.explain(true)
== Parsed Logical Plan ==
'Sort ['Date DESC NULLS LAST], false
+- 'Project ['Date, 'Open, 'Close]
   +- 'Filter ('Close < 'Open)
      +- 'UnresolvedRelation <code>google_stocks</code>

== Analyzed Logical Plan ==
Date: string, Open: double, Close: double
Sort [Date#21 DESC NULLS LAST], false
+- Project [Date#21, Open#22, Close#25]
   +- Filter (Close#25 < Open#22)
      +- SubqueryAlias google_stocks
         +- Relation[Date#21,Open#22,High#23,Low#24,Close#25,Volume#26]
csv

== Optimized Logical Plan ==
Sort [Date#21 DESC NULLS LAST], false
+- Project [Date#21, Open#22, Close#25]
   +- Filter ((isnotnull(Close#25) && isnotnull(Open#22)) && (Close#25 <
Open#22))
      +- Relation[Date#21,Open#22,High#23,Low#24,Close#25,Volume#26] csv

== Physical Plan ==
*Sort [Date#21 DESC NULLS LAST], false, 0
+- *Project [Date#21, Open#22, Close#25]
   +- *Filter ((isnotnull(Close#25) && isnotnull(Open#22)) && (Close#25
< Open#22))
      +- *FileScan csv [Date#21,Open#22,Close#25] Batched: false,
Format: CSV, Location:
InMemoryFileIndex[file:/Users/danno/Downloads/goog.csv],
PartitionFilters: [], PushedFilters: [IsNotNull(Close),
IsNotNull(Open)], ReadSchema:
struct<Date:string,Open:double,Close:double>
```

Step 3: Notice the differences between each of the plans

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`

Lab: Scala Value Types

Step 1: Start up the spark-shell

Step 2: Import `import org.apache.spark.sql.types._`

```
import org.apache.spark.sql.types._
```

Step 3: Create a `ByteType`

```
val b = ByteType()
```

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

- 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

Schemas

- Schemas have by default are assumed by the structure of our tables
- We can view the schemas of each of these `DataFrame` by calling `schema`
- A schema is a `StructType` made up of a number of fields called `StructFields`
- A `StructField` has:
  - A name,
  - A type
  - A boolean that specifies whether the column is nullable
- A schema can also contain other `StructType` (Spark complex types).
- Can also be overridden by your own custom schema which is preferred for production

View the Schema of a `DataFrame`

A schema for a `DataFrame` can viewed with:

```
df.printSchema()
```

Customizing A Schema

- import the types that you are requiring for Spark
- `import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}`
- Include them when calling `read` to get specific types

Lab: Override our read with our own customized schema

Step 1: In the `spark-shell`, copy the following, and paste it into the spark-shell using `:paste` mode

```
val mySchema = new StructType(Array(
   new StructField("VOLUME", LongType, false),
   new StructField("HIGH", DoubleType, false),
   new StructField("LOW", DoubleType, false),
   new StructField("DATE", StringType, false),
   new StructField("CLOSE", DoubleType, false),
   new StructField("OPEN", DoubleType, false)))
```

Step 2: In the `spark-shell`, read in the csv once more only this time, using our custom schema

```
val googleHistoryCSV = spark.read.schema(mySchema).csv
("/Users/danno/Downloads/goog.json")
```

Step 3: Analyze the schema using `printSchema` and the `schema` method on the `DataFrame`

```
googleHistoryCSV.printSchema
```

```
googleHistoryCSV.schema
```

Columns

- Embodied in the API as a `Column` type
- Can be obtained by either `col` or `column` function residing in `org.apache.spark.sql.functions`
- **IMPORTANT** We can program what we want and those columns don't really need to exist

```
import org.apache.spark.sql.functions.{col, column}

col("someColumnName")
column("someColumnName")
$"someColumnName"
'someColumnName
```

Columns direct from `DataFrame`

- Columns can also be called upon from the `DataFrame` directly

```
dataFrame.col("count")
```

Access all the columns from a `DataFrame`

- All the columns can be accessed from a `DataFrame` using `columns`

```
df.columns
```

Access all the columns from a `googleHistoryCSV`

Step 1: In spark-shell, determine all the column names that are currently in `googleHistoryCSV`

```
> googleHistoryCSV.columns
```

- All the columns can be accessed from a `DataFrame` using `columns`

```
df.columns
```

Expressions

- Transformations on one or more values of records on a `DataFrame`
- Is a function that can be imported `import org.apache.spark.sql.functions.expr`

Obtaining a single column

- There is more than one way to get a column, and you can use an expression

```
import org.apache.spark.sql.functions.expr

expr("someColumn")
```

Making complex expressions

- Expressions are dynamic, and you can do varying things
- For example: `expr(col("High") + 5 < col("Low") - 2)`
- This creates directed acyclic graph
- You can also place the entire expression into a String
- `expr("HIGH + 5 < LOW - 2")`
- This creates the foundation as to why SparkSQL works

Lab: Find all the rows using expressions

Step 1: In the spark-shell and given `googleHistoryCSV` already established enter the following:

```
val badDays3 = googleHistoryCSV.where(expr("CLOSE < OPEN"))
```

Step 2: `show` the results of `badDays3`

```
badDays3.show
```

Rows

- Embodied in the API as a `Row` type
- You can add a row after the fact to a `DataFrame`

```
val newRow = Row("24-Jul-17", 967.84, 967.84, 960.33, 961.08, 1493955)
```

Getting the first row from a `DataFrame`

- You can get the first row of a `DataFrame` by calling `first` or `head`

```
df.first
df.head
```

`parallelize`

- When creating `DataFrames` on the fly we can use `parallelize`
- `parallelize`:
  - Takes a `Seq` with `Row`
  - Returns an RDD (Resilient Distributed Dataset) which is a lower level API for data manipulation

Lab: Create your own DataFrame using and `Row`

Step 1: In spark-shell, copy and paste the following imports:

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.{StructField, StructType,
                                   StringType, IntegerType}
```

Step 2: Create a schema

```
val employeeSchema = new StructType(Array(
  new StructField("firstName", StringType, false),
  new StructField("middleName", StringType, true),
  new StructField("lastName", StringType, false),
  new StructField("salaryPerYear", IntegerType, false)
))
```

Step 3: Create some rows in a `Seq`

```
val employees = Seq(Row("Abe", null, "Lincoln", 40000),
                    Row("Martin", "Luther", "King", 80000),
                    Row("Ben", null , "Franklin", 82000),
                    Row("Toni", null , "Morrisson", 82000))
```

Step 4: Create a `DataFrame` using an alternate means using `spark.createDataFrame` and verify using `show`

```
val employeeDF = spark.createDataFrame(employees, employeeSchema)
employeeDF.show
```

Creating a `DataFrame` on the cheap using `toDF`

- You can create a `DataFrame` on the spot using `toDF` from a `Seq`
- Doesn't work well with `null`
- Uses `implicit` in Scala to create the `DataFrame`

```
val afcNorth = Seq(("Bengals", "Cincinnati", "Paul Brown Stadium"),
                   ("Steelers", "Pittsburgh", "Heinz Field"),
                   ("Browns", "Cleveland", "FirstEnergy Field"),
                   ("Ravens", "Baltimore", "M&T Bank Stadium"))
val afcNorthDataFrame = afcNorth.toDF("NAME", "CITY", "STADIUM")
afcNorthDataFrame.show
```

Sample Data

All example in this chapter use the `googleHistoryCSV` which we will rename for all example with `dataFrame` which was derived from:

```
val mySchema = new StructType(Array(
   new StructField("DATE", StringType, false),
   new StructField("OPEN", DoubleType, false),
   new StructField("HIGH", DoubleType, false),
   new StructField("LOW", DoubleType, false),
   new StructField("CLOSE", DoubleType, false),
   new StructField("VOLUME", LongType, false)))

val dataFrame = spark.read.schema(mySchema).option("header", true).csv
("/Users/danno/Downloads/goog.csv")

dataFrame.createOrReplaceTempView("google_data")
```

Sample Data Results

```
scala> dataFrame.show
+---------+------+------+------+------+-------+
|     DATE|  OPEN|  HIGH|   LOW| CLOSE| VOLUME|
+---------+------+------+------+------+-------+
|19-Jul-17|967.84|973.04|964.03|970.89|1224540|
|18-Jul-17| 953.0|968.04| 950.6| 965.4|1153964|
|17-Jul-17| 957.0|960.74|949.24|953.42|1165537|
|14-Jul-17| 952.0|956.91| 948.0|955.99|1053774|
|13-Jul-17|946.29|954.45|943.01|947.16|1294687|
|12-Jul-17|938.68| 946.3|934.47|943.83|1532144|
|11-Jul-17|929.54|931.43| 922.0|930.09|1113235|
|10-Jul-17|921.77|930.38|919.59| 928.8|1192825|
| 7-Jul-17|908.85|921.54|908.85|918.59|1637785|
| 6-Jul-17|904.12|914.94| 899.7|906.69|1424503|
| 5-Jul-17|901.76|914.51| 898.5|911.71|1813884|
| 3-Jul-17|912.18|913.94|894.79| 898.7|1710373|
|30-Jun-17|926.05|926.05|908.31|908.73|2090226|
|29-Jun-17|929.92|931.26|910.62|917.79|3299176|
|28-Jun-17| 929.0|942.75| 916.0|940.49|2721406|
|27-Jun-17|942.46|948.29|926.85|927.33|2579930|
|26-Jun-17| 969.9|973.31|950.79|952.27|1598355|
|23-Jun-17|956.83| 966.0| 954.2|965.59|1527856|
|22-Jun-17| 958.7|960.72|954.55|957.09| 941958|
|21-Jun-17|953.64| 960.1|950.76|959.45|1202233|
+---------+------+------+------+------+-------+
only showing top 20 rows
```

Labs all the way!

- Feel free to try out none, some, or all of the following to get a feel for what they do.
- Experiment using spark-shell

`select`

- `select` allows us to manipulate `DataFrame` to another `DataFrame`
- Easiest to pass the columns you wish to transform or use

```
scala> dataFrame.select("DATE").show(5)
+---------+
|     DATE|
+---------+
|19-Jul-17|
|18-Jul-17|
|17-Jul-17|
|14-Jul-17|
|13-Jul-17|
-----------
```

Spark SQL Equivalent:

```
scala> spark.sql("SELECT DATE FROM google_data").show(5)
```

`select` multiple columns

- `select` can do multiple columns

```
scala> dataFrame.select("DATE", "VOLUME").show(5)
+---------+-------+
|     DATE| VOLUME|
+---------+-------+
|19-Jul-17|1224540|
|18-Jul-17|1153964|
|17-Jul-17|1165537|
|14-Jul-17|1053774|
|13-Jul-17|1294687|
+---------+-------+
only showing top 5 rows
```

Spark SQL Equivalent:

```
scala> spark.sql("SELECT DATE, VOLUME FROM google_data").show(5)
```

`select` Column Alternatives

All variants for selecting a column

```
import org.apache.spark.sql.functions.{expr, col, column}

df.select(
  df.col("DATE"),
  col("DATE"),
  column("DATE"),
  'DATE,
  $"DATE",
  expr("DATE")
).show(2)
```

`selectExpr`

- Combines both `select` and `expr`
- Accepts a list of String as expressions
- No need to include `expr`

```
scala> dataFrame.selectExpr("DATE as TRADEDATE", "DATE").show(5)
+---------+---------+
|TRADEDATE|     DATE|
+---------+---------+
|19-Jul-17|19-Jul-17|
|18-Jul-17|18-Jul-17|
|17-Jul-17|17-Jul-17|
|14-Jul-17|14-Jul-17|
|13-Jul-17|13-Jul-17|
+---------+---------+
only showing top 5 rows
```

Showing all the columns using `*` in `selectExpr`

- A `*` can be used to show all the columns in a `selectExpr`

```
scala> dataFrame.selectExpr("*", "DATE as TRADEDATE").show(5)
+---------+------+------+------+------+-------+---------+
|     DATE|  OPEN|  HIGH|   LOW| CLOSE| VOLUME|TRADEDATE|
+---------+------+------+------+------+-------+---------+
|19-Jul-17|967.84|973.04|964.03|970.89|1224540|19-Jul-17|
|18-Jul-17| 953.0|968.04| 950.6| 965.4|1153964|18-Jul-17|
|17-Jul-17| 957.0|960.74|949.24|953.42|1165537|17-Jul-17|
|14-Jul-17| 952.0|956.91| 948.0|955.99|1053774|14-Jul-17|
|13-Jul-17|946.29|954.45|943.01|947.16|1294687|13-Jul-17|
+---------+------+------+------+------+-------+---------+
only showing top 5 rows
```

Spark SQL Equivalent:

```
scala> spark.sql("SELECT *, DATE as TRADEDATE FROM google_data").show(5)
```

Literals

- Literals are explicit values made to be included in a `DataFrame`
- This will inevitably be created into your preferred languages type

```
scala> dataFrame.select(expr("*"), lit(30).as("CONSTANT")).show(5)
+---------+------+------+------+------+-------+--------+
|     DATE|  OPEN|  HIGH|   LOW| CLOSE| VOLUME|CONSTANT|
+---------+------+------+------+------+-------+--------+
|19-Jul-17|967.84|973.04|964.03|970.89|1224540|      30|
|18-Jul-17| 953.0|968.04| 950.6| 965.4|1153964|      30|
|17-Jul-17| 957.0|960.74|949.24|953.42|1165537|      30|
|14-Jul-17| 952.0|956.91| 948.0|955.99|1053774|      30|
|13-Jul-17|946.29|954.45|943.01|947.16|1294687|      30|
+---------+------+------+------+------+-------+--------+
only showing top 5 rows
```

For Spark SQL, there is no `lit` function, just express the value

Spark SQL Equivalent:

```
scala> spark.sql("SELECT *, 30 as CONSTANT FROM google_data").show(5)
```

Adding a column

- An alternative way to add a column is with `withColumn`
- Adds a column or replacing the existing column that has the same name.

- `withColumn` takes a name, and a column definition or function

```
scala> dataFrame.withColumn("CONSTANT", lit(30)).show(5)
+---------+------+------+------+------+-------+--------+
|     DATE|  OPEN|  HIGH|   LOW| CLOSE| VOLUME|CONSTANT|
+---------+------+------+------+------+-------+--------+
|19-Jul-17|967.84|973.04|964.03|970.89|1224540|      30|
|18-Jul-17| 953.0|968.04| 950.6| 965.4|1153964|      30|
|17-Jul-17| 957.0|960.74|949.24|953.42|1165537|      30|
|14-Jul-17| 952.0|956.91| 948.0|955.99|1053774|      30|
|13-Jul-17|946.29|954.45|943.01|947.16|1294687|      30|
+---------+------+------+------+------+-------+--------+
only showing top 5 rows
```

Renaming Columns with `withColumnRenamed`

- A column can also be renamed with `withColumnRenamed`
- `withColumnRenamed` takes the old column name first, then the new name

```
scala> dataFrame.withColumnRenamed("DATE", "TRADEDATE").show(5)
+---------+------+------+------+------+-------+
|TRADEDATE|  OPEN|  HIGH|   LOW| CLOSE| VOLUME|
+---------+------+------+------+------+-------+
|19-Jul-17|967.84|973.04|964.03|970.89|1224540|
|18-Jul-17| 953.0|968.04| 950.6| 965.4|1153964|
|17-Jul-17| 957.0|960.74|949.24|953.42|1165537|
|14-Jul-17| 952.0|956.91| 948.0|955.99|1053774|
|13-Jul-17|946.29|954.45|943.01|947.16|1294687|
+---------+------+------+------+------+-------+
only showing top 5 rows
```

Removing Columns

- Removing columns is done with `drop`
- The function can take 1 or more Strings for the column names

```
scala> dataFrame.drop("OPEN", "LOW").show(5)
+---------+------+------+-------+
|     DATE|  HIGH| CLOSE| VOLUME|
+---------+------+------+-------+
|19-Jul-17|973.04|970.89|1224540|
|18-Jul-17|968.04| 965.4|1153964|
|17-Jul-17|960.74|953.42|1165537|
|14-Jul-17|956.91|955.99|1053774|
|13-Jul-17|954.45|947.16|1294687|
+---------+------+------+-------+
only showing top 5 rows
```

Casting

- You can cast to a type by using the `cast` function
- Available in Scala and Spark SQL

Given the schema currently is all double:

```
scala> dataFrame.printSchema
root
 |-- DATE: string (nullable = true)
 |-- OPEN: double (nullable = true)
 |-- HIGH: double (nullable = true)
 |-- LOW: double (nullable = true)
 |-- CLOSE: double (nullable = true)
 |-- VOLUME: long (nullable = true)
```

We can convert say LOW to an int using `cast:

```
scala> dataFrame.withColumn("LOW", col("LOW").cast("int")).printSchema
root
 |-- DATE: string (nullable = true)
 |-- OPEN: double (nullable = true)
 |-- HIGH: double (nullable = true)
 |-- LOW: integer (nullable = true)
 |-- CLOSE: double (nullable = true)
 |-- VOLUME: long (nullable = true)
```

Spark SQL Equivalent:

```
scala> spark.sql("SELECT CAST(LOW as int) FROM google_data").printSchema
root
 |-- LOW: integer (nullable = true)
```

Filtering Rows

- Filtering is a common functional programming construct
- It "weeds out" data from a container that doesn't meet the requirements
- Comes with two forms: `where` and `filter`
  - `where` takes either a `Column` or a `String` expression
  - `filter` takes a predicate `A ⇒ Boolean`

```
scala> dataFrame.filter(col("OPEN") < col("CLOSE")).show(5)
```

```
scala> dataFrame.where("OPEN < CLOSE").show(5)
```

Both the above will return ...

```
+---------+------+------+------+------+-------+
|     DATE|  OPEN|  HIGH|   LOW| CLOSE| VOLUME|
+---------+------+------+------+------+-------+
|19-Jul-17|967.84|973.04|964.03|970.89|1224540|
|18-Jul-17| 953.0|968.04| 950.6| 965.4|1153964|
|14-Jul-17| 952.0|956.91| 948.0|955.99|1053774|
|13-Jul-17|946.29|954.45|943.01|947.16|1294687|
|12-Jul-17|938.68| 946.3|934.47|943.83|1532144|
+---------+------+------+------+------+-------+
only showing top 5 rows
```

Spark SQL Equivalent:

```
scala> spark.sql("SELECT * from google_data WHERE CLOSE < OPEN").show(5)
```

There is no benefit to putting all filter logic in one block since Spark will inevitably calculate the best process regardless

Distinct Data

- Intuitively use `distinct` to obtain distinct data from where required

Given:

```
val countriesMedalCountSeq =
    Seq(("United States", "100m Freestyle", 1, 0, 3),
        ("Spain", "100m Butterfly", 2, 1, 1),
        ("Japan", "100m Butterfly", 0, 3, 0),
        ("Spain", "100m Freestyle", 0, 0, 3),
        ("Uruguay", "100m Breaststroke", 0, 1, 0),
        ("United States", "100m Breaststroke", 2, 2, 0))
val countriesMedalCountDF = countriesMedalCountSeq
                              .toDF("Country", "Event", "Gold",
                                    "Silver", "Bronze")
```

You can retreive the distinct data by doing the following:

```
scala> countriesMedalCountDF.selectExpr("Country").distinct.show
+-------------+
|      Country|
+-------------+
|United States|
|        Spain|
|      Uruguay|
|        Japan|
---------------
```

Spark SQL Equivalent:

```
scala> spark.sql("SELECT DISTINCT(Country) from country_medal_count"
).show
```

Appending Rows to Existing Data

- Data can be appended from one DataFrame into another using 'union'
- This is one the great features of Spark, two `DataFrame` can come from two different data sources
- Consider multiple datasource with similar data where you want to structure the data in the same way

Starting with Two Datasources

Consider one DataSource:

```
val countriesMedalCountDF =
    Seq(("United States", "100m Freestyle", 1, 0, 3),
        ("Spain", "100m Butterfly", 2, 1, 1),
        ("Japan", "100m Butterfly", 0, 3, 0),
        ("Spain", "100m Freestyle", 0, 0, 3),
        ("Uruguay", "100m Breaststroke", 0, 1, 0),
        ("United States", "100m Breaststroke", 2, 2, 0))
           .toDF("Country", "Event", "Gold", "Silver", "Bronze")
```

And another that is somewhat different that came from a different source:

```
val countriesMedalCountDF2 =
    Seq(("United States", "100m Freestyle", 1, 0, 3),
        ("Spain", "100m Backstroke", 2, 1, 1),
        ("Spain", "200m Breaststroke", 1, 0, 0),
        ("Spain", "500m Freestyle", 3, 0, 0),
        ("Spain", "1000m Freestyle", 2, 1, 0),
        ("United States", "100m Breaststroke", 2, 2, 0))
           .toDF("Country", "Event", "Gold", "Silver", "Bronze")
```

You also notice that the second you only need `Spain`. You can merge the following:

Using `union` to bring them together

```
scala> countriesMedalCountDF.union(countriesMedalCountDF2.where("country
== 'Spain'")).show(20)
+-------------+-----------------+----+------+------+
|      Country|            Event|Gold|Silver|Bronze|
+-------------+-----------------+----+------+------+
|United States|   100m Freestyle|   1|     0|     3|
|        Spain|   100m Butterfly|   2|     1|     1|
|        Japan|   100m Butterfly|   0|     3|     0|
|        Spain|   100m Freestyle|   0|     0|     3|
|      Uruguay|100m Breaststroke|   0|     1|     0|
|United States|100m Breaststroke|   2|     2|     0|
|        Spain|  100m Backstroke|   2|     1|     1|
|        Spain|200m Breaststroke|   1|     0|     0|
|        Spain|   500m Freestyle|   3|     0|     0|
|        Spain|  1000m Freestyle|   2|     1|     0|
+-------------+-----------------+----+------+------+
```

Sorting

- Sorting is done with `sort` or `orderBy`
- `sort` can either take a list of expressions or `String` that represents the `Column`

- `orderBy` is an alias so that you can express yourself differently
- The default is to sort in ascending order

```
dataFrame.sort("VOLUME").show(5)
dataFrame.orderBy("VOLUME", "HIGH").show(5)
dataFrame.orderBy(col("VOLUME"), col("HIGH")).show(5)
```

```
scala> dataFrame.orderBy("VOLUME", "HIGH").show(5)
+---------+------+------+------+------+------+
|     DATE|  OPEN|  HIGH|   LOW| CLOSE|VOLUME|
+---------+------+------+------+------+------+
|25-Nov-16|764.26| 765.0|760.52|761.68|587421|
|23-Dec-16| 790.9|792.74|787.28|789.91|623944|
|18-Aug-16|780.01|782.86| 777.0| 777.5|719429|
|12-Aug-16| 781.5| 783.4| 780.4|783.22|740498|
|29-Dec-16|783.33|785.93|778.92|782.79|744272|
+---------+------+------+------+------+------+
only showing top 5 rows
```

Sorting using functions `asc` and `desc`

- Both `asc` and `desc` are methods that accept a `String` and return `Column`
- Therefore can be used for sorting as a `Column`

```
scala> dataFrame.selectExpr("*").orderBy(desc("VOLUME")).show(5)
+---------+------+------+------+------+-------+
|     DATE|  OPEN|  HIGH|   LOW| CLOSE| VOLUME|
+---------+------+------+------+------+-------+
|10-Nov-16|791.17|791.17|752.18|762.56|4745183|
|28-Oct-16|808.35|815.49|793.59|795.37|4269902|
|29-Jul-16|772.71|778.55|766.77|768.79|3841482|
|12-Jun-17|939.56|949.36|915.23| 942.9|3763529|
|14-Nov-16| 755.6|757.85|727.54|736.08|3654385|
+---------+------+------+------+------+-------+
only showing top 5 rows
```

Getting the `limit` of the results

- The difference between this function and `head` is that `head` is an action
- `limit` on the other hand is lazy and returns a new `Dataset`.

```
scala> dataFrame.selectExpr("*").limit(5).show(5)
+---------+------+------+------+------+-------+
|     DATE|  OPEN|  HIGH|   LOW| CLOSE| VOLUME|
+---------+------+------+------+------+-------+
|19-Jul-17|967.84|973.04|964.03|970.89|1224540|
|18-Jul-17| 953.0|968.04| 950.6| 965.4|1153964|
|17-Jul-17| 957.0|960.74|949.24|953.42|1165537|
|14-Jul-17| 952.0|956.91| 948.0|955.99|1053774|
|13-Jul-17|946.29|954.45|943.01|947.16|1294687|
+---------+------+------+------+------+-------+
```

Repartition

- Just like indexing in RDBS, it would be a good idea in time to repartition often used columns into their own partitions
- This is done for performance and minimizing network traffic
- `repartition` can be set with either:
  - The number of partitions
  - The `Column`
  - Both

```
scala> val largeRange = spark.range(1, 1000000).toDF
largeRange: org.apache.spark.sql.DataFrame = [id: bigint]

scala> largeRange.rdd.getNumPartitions
res145: Int = 4
```

Repartitioning to 10 partitions

```
scala> val largeRangeDistributed = largeRange.repartition(10)
largeRangeDistributed: org.apache.spark.sql.Dataset
[org.apache.spark.sql.Row] = [id: bigint]

scala> largeRangeDistributed.rdd.getNumPartitions
res146: Int = 10
```

Coalesce

- To `coalesce` will also rearrange to a number of partitions
- It will make an attempt to bring down the number of columns where possible
- In the following example, the number of columns will likely be brought down to 4.

```
scala> val largeRange = spark.range(1, 1000000).toDF
largeRange: org.apache.spark.sql.DataFrame = [id: bigint]

scala> val coalescedRange = largeRange.repartition(10).coalesce(5)
coalescedRange: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[id: bigint]

scala> coalescedRange.rdd.getNumPartitions
res148: Int = 4
```

Collect

- `collect()` will get all the rows from the `DataFrame`
- May come at a cost depending on the size of the result
- Will return an `Array[Row]` of your data

```
scala> val collectedData = dataFrame.collect
collectedData: Array[org.apache.spark.sql.Row] = Array([19-Jul-17,967.
84,973.04,964.03,970.89,1224540], [18-Jul-17,953.0,968.04,950.6,965.4
,1153964], [17-Jul-17,957.0,960.74,949.24,953.42,1165537], [14-Jul-17
,952.0,956.91,948.0,955.99,1053774], [13-Jul-17,946.29,954.45,943.01,
947.16,1294687], [12-Jul-17,938.68,946.3,934.47,943.83,1532144], [11-Jul
-17,929.54,931.43,922.0,930.09,1113235], [10-Jul-17,921.77,930.38,919.
59,928.8,1192825], [7-Jul-17,908.85,921.54,908.85,918.59,1637785], [6
-Jul-17,904.12,914.94,899.7,906.69,1424503], [5-Jul-17,901.76,914.51,
898.5,911.71,1813884], [3-Jul-17,912.18,913.94,894.79,898.7,1710373],
[30-Jun-17,926.05,926.05,908.31,908.73,2090226], [29-Jun-17,929.92,931
.26,910.62,917.79,3299176], [28-Jun-17,929.0,942.75,916.0,940.49,
2721406], [27-Jun-17,942.46,948.29,...
```

User Defined Functions

- If you don't have enough functions to work with? Create your own!
- Start with a standard function

```
def is_odd(x:Int):Boolean = x % 2 != 0
```

- Wrap it in a `udf` (User defined function)

```
val is_odd_udf = udf(is_odd(_:Int):Boolean)
```

- Use the `udf`

```
scala> dataFrame.withColumn("IS_ODD_VOLUME", is_odd_udf('VOLUME)).show(
5)
+---------+------+------+------+------+-------+-------------+
|     DATE|  OPEN|  HIGH|   LOW| CLOSE| VOLUME|IS_ODD_VOLUME|
+---------+------+------+------+------+-------+-------------+
|19-Jul-17|967.84|973.04|964.03|970.89|1224540|        false|
|18-Jul-17| 953.0|968.04| 950.6| 965.4|1153964|        false|
|17-Jul-17| 957.0|960.74|949.24|953.42|1165537|         true|
|14-Jul-17| 952.0|956.91| 948.0|955.99|1053774|        false|
|13-Jul-17|946.29|954.45|943.01|947.16|1294687|         true|
+---------+------+------+------+------+-------+-------------+
```

Joins

- Spark can bring in separate DataFrames/Datasets and join them together
- Joins are *left* and *right*
- Matched by a *key*

Join Types

inner joins	Keep rows with keys that exist in the left and right `Dataframe`
outer joins	Keep rows with keys in either the left or right `DataFrame`
left outer joins	Keep rows with keys in the left `DataFrame`
right outer joins	Keep rows with keys in the right `DataFrame`
left semi joins	Keep the rows in the left and only left `DataFrame` where the key appears in the right `DataFrame`
left anti joins	Keep the rows in the left and only the left `DataFrame` where they does not appear in the right `DataFrame`
cross joins	Match every row in the left `DataFrame` with every row in the right `DataFrame`

Setting up the tables to join

Given the following:

```
val cities = Seq(
                  (1, "San Francisco", "CA"),
                  (2, "Dallas", "TX"),
                  (3, "Pittsburgh", "PA"),
                  (4, "Buffalo", "NY"),
                  (5, "Oklahoma City", "OK"),
                  (6, "New York City", "NY"),
                  (7, "Los Angeles", "CA"),
                  (8, "Omaha", "NE")).toDF("id", "city", "state")

val teams = Seq(
                  (1, 7, "Rams", "Football"),
                  (2, 7, "Dodgers", "Baseball"),
                  (3, 6, "Giants", "Football"),
                  (4, 1, "Giants", "Baseball"),
                  (5, 4, "Bills", "Football"),
                  (6, 3, "Pirates", "Baseball"),
                  (7, 1, "49ers", "Football"),
                  (8, 3, "Steelers", "Football")).toDF("id", "city_id",
"team", "sport_type")
```

Inner Join

Create an inner join with the following:

```
scala> val innerjoin = cities.join(teams, cities.col("id") === teams.
col("city_id"))
```

The above returns the following:

```
scala> innerjoin.show
+---+-------------+-----+---+-------+--------+----------+
| id|         city|state| id|city_id|    team|sport_type|
+---+-------------+-----+---+-------+--------+----------+
|  1|San Francisco|   CA|  7|      1|   49ers|  Football|
|  1|San Francisco|   CA|  4|      1|  Giants|  Baseball|
|  3|   Pittsburgh|   PA|  8|      3|Steelers|  Football|
|  3|   Pittsburgh|   PA|  6|      3| Pirates|  Baseball|
|  4|      Buffalo|   NY|  5|      4|   Bills|  Football|
|  6|New York City|   NY|  3|      6|  Giants|  Football|
|  7|  Los Angeles|   CA|  2|      7| Dodgers|  Baseball|
|  7|  Los Angeles|   CA|  1|      7|    Rams|  Football|
+---+-------------+-----+---+-------+--------+----------+
```

Outer Join

Create an outer join with the following:

```
scala> val outerjoin = cities.join(teams, cities.col("id") === teams.
col("city_id"), "outer")
```

The above returns the following:

```
scala> outerjoin.show
+---+-------------+-----+----+-------+--------+----------+
| id|         city|state|  id|city_id|    team|sport_type|
+---+-------------+-----+----+-------+--------+----------+
|  1|San Francisco|   CA|   4|      1|  Giants|  Baseball|
|  1|San Francisco|   CA|   7|      1|   49ers|  Football|
|  6|New York City|   NY|   3|      6|  Giants|  Football|
|  3|   Pittsburgh|   PA|   6|      3| Pirates|  Baseball|
|  3|   Pittsburgh|   PA|   8|      3|Steelers|  Football|
|  5|Oklahoma City|   OK|null|   null|    null|      null|
|  4|      Buffalo|   NY|   5|      4|   Bills|  Football|
|  8|        Omaha|   NE|null|   null|    null|      null|
|  7|  Los Angeles|   CA|   1|      7|    Rams|  Football|
|  7|  Los Angeles|   CA|   2|      7| Dodgers|  Baseball|
|  2|       Dallas|   TX|null|   null|    null|      null|
+---+-------------+-----+----+-------+--------+----------+
```

Duplicate Keys

- In the other example, keys have also been duplicated, `id` and `id`
- One `id` is for the `city`, the other is for `team`
- Use `withColumnRenamed` to establish desired field names

```
val outerjoin = cities.join(teams.withColumnRenamed("id", "team_id"),
cities.col("id") === teams.col("city_id"), "outer").withColumnRenamed
("id", "city_id").show
```

This gives the result of:

```
+-------+-------------+-----+-------+-------+--------+----------+
|city_id|         city|state|team_id|city_id|    team|sport_type|
+-------+-------------+-----+-------+-------+--------+----------+
|      1|San Francisco|   CA|      4|      1|  Giants|  Baseball|
|      1|San Francisco|   CA|      7|      1|   49ers|  Football|
|      6|New York City|   NY|      3|      6|  Giants|  Football|
|      3|   Pittsburgh|   PA|      6|      3| Pirates|  Baseball|
|      3|   Pittsburgh|   PA|      8|      3|Steelers|  Football|
|      5|Oklahoma City|   OK|   null|   null|    null|      null|
|      4|      Buffalo|   NY|      5|      4|   Bills|  Football|
|      8|        Omaha|   NE|   null|   null|    null|      null|
|      7|  Los Angeles|   CA|      1|      7|    Rams|  Football|
|      7|  Los Angeles|   CA|      2|      7| Dodgers|  Baseball|
|      2|       Dallas|   TX|   null|   null|    null|      null|
+-------+-------------+-----+-------+-------+--------+----------+
```

Advanced RDD

Advanced RDD

View the content on `AdvancedRDDSpec` for samples of running some other exercises

Spark in Maven

Creating an archetype

```
mvn archetype:generate
```

Selecting the quick start project

In the following example, `1057` is the latest id

```
1959: remote -> us.fatehi:schemacrawler-archetype-maven-project (-)
1960: remote -> us.fatehi:schemacrawler-archetype-plugin-command (-)
1961: remote -> us.fatehi:schemacrawler-archetype-plugin-dbconnector (-)
1962: remote -> us.fatehi:schemacrawler-archetype-plugin-lint (-)
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 1057:
```

There are some archetypes available with spark

Enter project values

- Select the latest version for the archetype
- Enter the `groupId`, `artifactId`, and `version`
- Create the project
- Change to the project directory, named for the `artifactId` and edit the `pom.xml` file

Configure Properties

- To make it easier, enter the following properties in the `<properties>` section

```
<properties>
    <spark.groupId>org.apache.spark</spark.groupId>
    <spark.version>2.2.0</spark.version>
    <scala.major.version>2.11</scala.major.version>
    <scala.version>${scala.major.version}.8</scala.version>
</properties>
```

Logging Dependencies

For Logging:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.25</version>
</dependency>
```

Spark Dependencies

For Spark Dependencies:

```
<dependency>
      <groupId>${spark.groupId}</groupId>
      <artifactId>spark-core_${scala.major.version}</artifactId>
      <version>${spark.version}</version>
</dependency>
<dependency>
      <groupId>${spark.groupId}</groupId>
      <artifactId>spark-sql_${scala.major.version}</artifactId>
      <version>${spark.version}</version>
</dependency>
<dependency>
      <groupId>${spark.groupId}</groupId>
      <artifactId>spark-streaming_${scala.major.version}</artifactId>
      <version>${spark.version}</version>
</dependency>
```

Create a Plugins Section

```
<build>
  <plugins>
     ...plugins go here
  </plugins>
</build>
```

Add and Fix the `maven-compiler-plugin` to use Java 8

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.7.0</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
    <executions>
        <execution>
            <phase>compile</phase>
            <goals>
                <goal>compile</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Add the `scala-maven-plugin` into maven

```
<plugin>
    <groupId>net.alchim31.maven</groupId>
    <artifactId>scala-maven-plugin</artifactId>
    <version>3.3.1</version>
    <executions>
        <execution>
            <id>scala-compile-first</id>
            <phase>process-resources</phase>
            <goals>
                <goal>add-source</goal>
                <goal>compile</goal>
            </goals>
        </execution>
        <execution>
            <id>scala-test-compile</id>
            <phase>process-test-resources</phase>
            <goals>
                <goal>testCompile</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```