Introduction to Spark

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

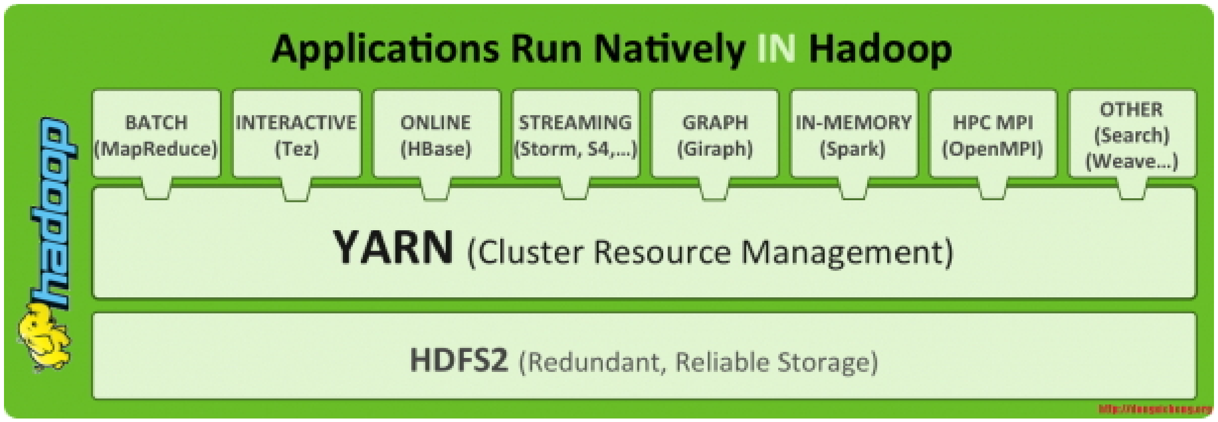
Thank Y. Kang (MIS/NSYSU) for providing slides.

Topic Overview

- Introduction to Spark
- The Overall Architecture
- Cluster Mode Overview
- Resilient Distributed Datasets
- Spark Shell
- Writing and Submitting a Spark Application
- A bit about Sparklyr
- Hands on Spark MLlib
- Scala Spark

Where We Stand Now

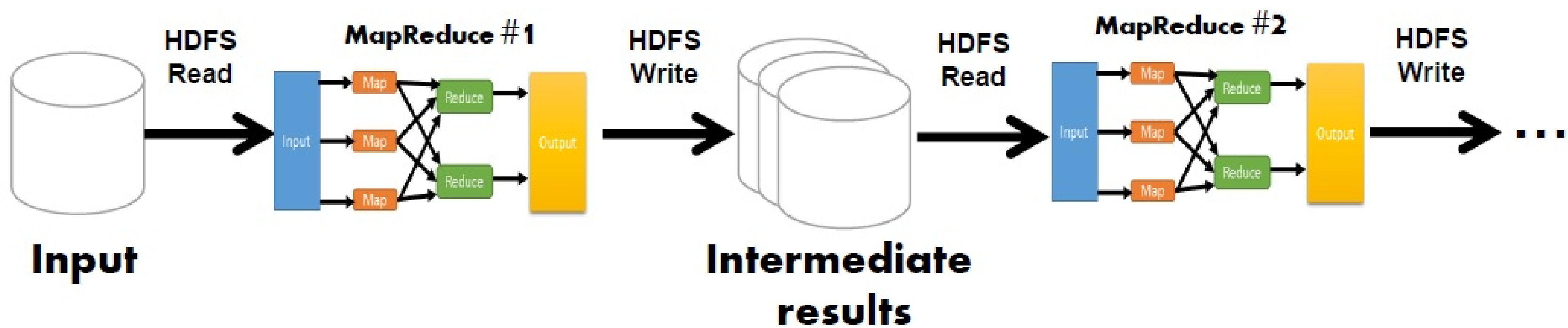
- We have learned MPI, Hadoop, MapReduce, Postgre.
- It is Spark(ling) time.
- Applications on the top of HDFS



Going beyond Hadoop & MapReduce

- Hadoop and MapReduce have been very successful in recent decade due to its easiness of use, linear scalability, and high availability. However, data analytics processes, such as feature engineering, often involves iterative and interactive data pipelining/sharing.

- Data reuse in Hadoop MapReduce is slow simply because it requires a series of replications, serializations, and disk I/O–chains of MapReduce tasks. The lack of data abstraction and data sharing mechanisms make Hadoop and MapReduce unable to leverage distributed memory of multiple computing nodes in emerging data applications that reuse intermediate analysis results.



Introduction to Apache Spark

- Apache Spark, an open-source cluster computing framework, is developed in 2009 at UC Berkeley AMP Lab to solve the problems.

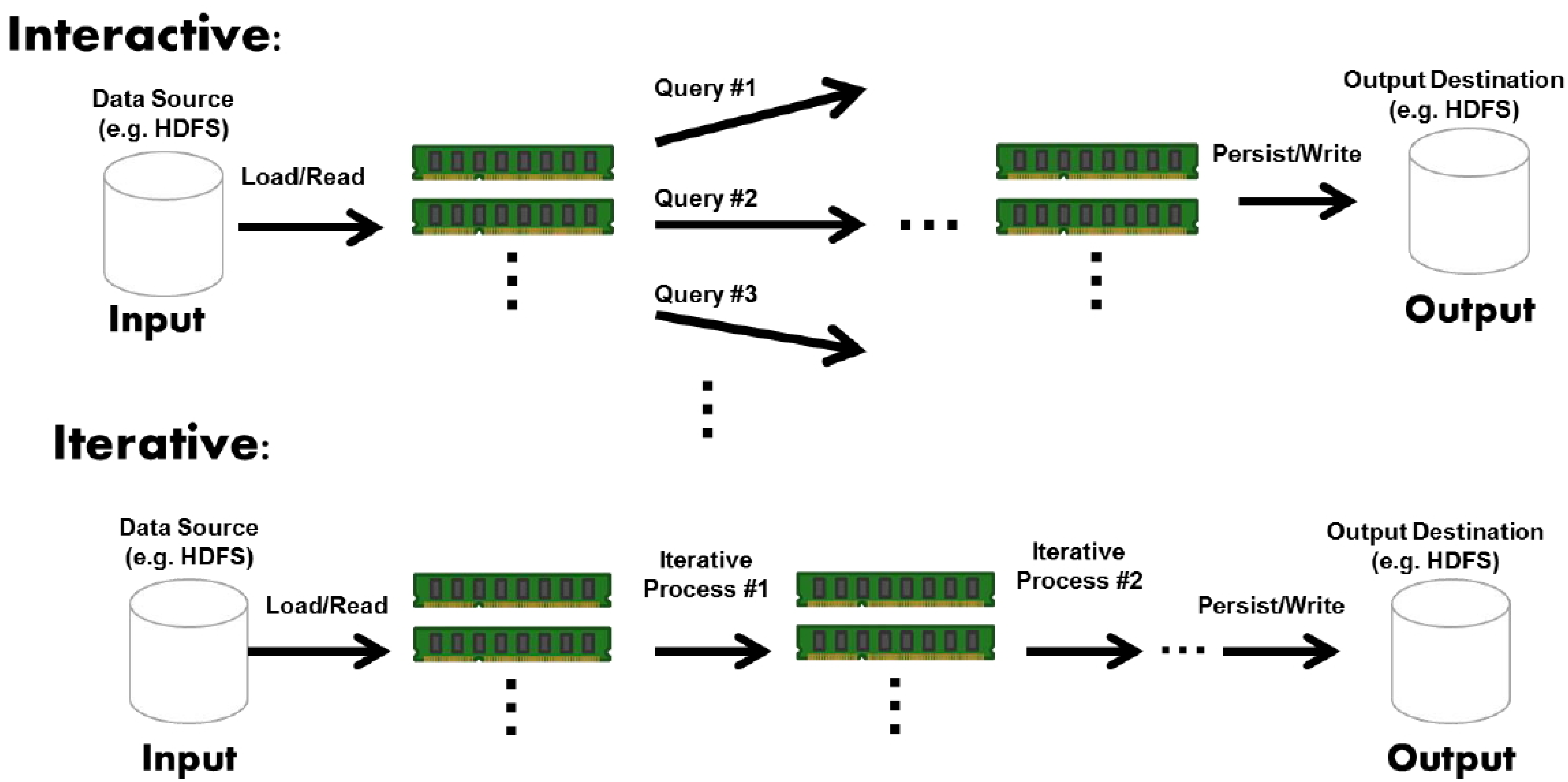
- Spark provides an unified engine with a stack of libraries that allow for complex analytics, including batch, streaming, interactive, and graphical computing.

- It is not proposed to replace but considered a generalization of Hadoop & MapReduce.

- Spark is one of the largest OSS(Open Source Software) communities in big data analytics, and its applications range from business, finance, healthcare, and other scientific computing.

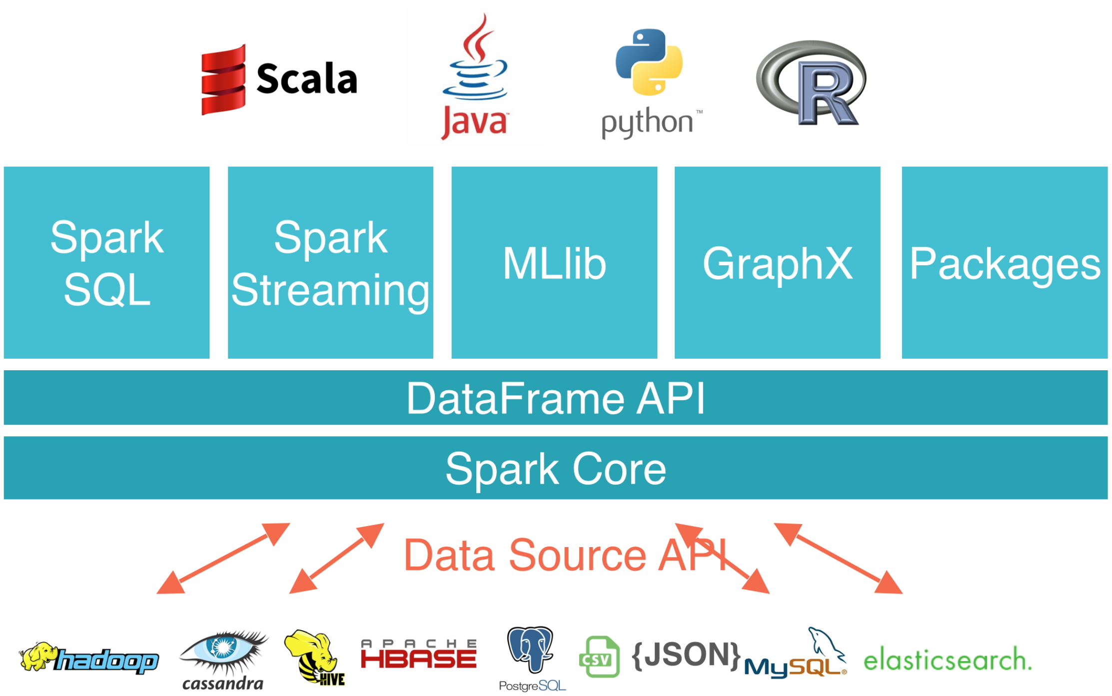
Need for Speed

- Growing main memory capacity has fueled the development of in-memory big data analytics.
- Unlike Hadoop MapReduce, Spark supports interactive and iterative data processing by eliminating disk I/O bottleneck.



The Overall Architecture

- Spark architecture.



Turning on Spark Cluster Standalone Mode

- Start the cluster

```
[hadoop@fisher sbin]$ /usr/local/spark/spark-2.1.0-bin-hadoop2.7/sbin/start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /usr/local/spark/spark-2.1.0-...
ps: /usr/local/greenplum-db/lib/libz.so.1: no version information available (required by...
ps: /usr/local/greenplum-db/lib/libz.so.1: no version information available (required by...
ps: /usr/local/greenplum-db/lib/libz.so.1: no version information available (required by...
hdatanode12: starting org.apache.spark.deploy.worker.Worker, logging to /usr/local/spark...
hdatanode21: starting org.apache.spark.deploy.worker.Worker, logging to /usr/local/spark...
hdatanode23: starting org.apache.spark.deploy.worker.Worker, logging to /usr/local/spark...
hdatanode11: starting org.apache.spark.deploy.worker.Worker, logging to /usr/local/spark...
hdatanode24: starting org.apache.spark.deploy.worker.Worker, logging to /usr/local/spark...
hdatanode13: starting org.apache.spark.deploy.worker.Worker, logging to /usr/local/spark...
hdatanode14: starting org.apache.spark.deploy.worker.Worker, logging to /usr/local/spark...
hdatanode22: starting org.apache.spark.deploy.worker.Worker, logging to /usr/local/spark...
```

Spark 2.1.0 **Spark Master at spark://hnamenode:7077**

URL: spark://hnamenode:7077
REST URL: spark://hnamenode:6066 *(cluster mode)*
Alive Workers: 8
Cores in use: 64 Total, 0 Used
Memory in use: 241.5 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20180416022913-192.168.30.11-19772	192.168.30.11:19772	ALIVE	8 (0 Used)	30.2 GB (0.0 B Used)
worker-20180416022913-192.168.30.12-4007	192.168.30.12:4007	ALIVE	8 (0 Used)	30.2 GB (0.0 B Used)
worker-20180416022913-192.168.30.13-17999	192.168.30.13:17999	ALIVE	8 (0 Used)	30.2 GB (0.0 B Used)
worker-20180428023028-192.168.30.22-15920	192.168.30.22:15920	ALIVE	8 (0 Used)	30.2 GB (0.0 B Used)
worker-20180428143028-192.168.30.14-32534	192.168.30.14:32534	ALIVE	8 (0 Used)	30.2 GB (0.0 B Used)
worker-20180428143028-192.168.30.21-1574	192.168.30.21:1574	ALIVE	8 (0 Used)	30.2 GB (0.0 B Used)
worker-20180428143028-192.168.30.23-14437	192.168.30.23:14437	ALIVE	8 (0 Used)	30.2 GB (0.0 B Used)
worker-20180428143028-192.168.30.24-2198	192.168.30.24:2198	ALIVE	8 (0 Used)	30.2 GB (0.0 B Used)

Cluster Mode Overview

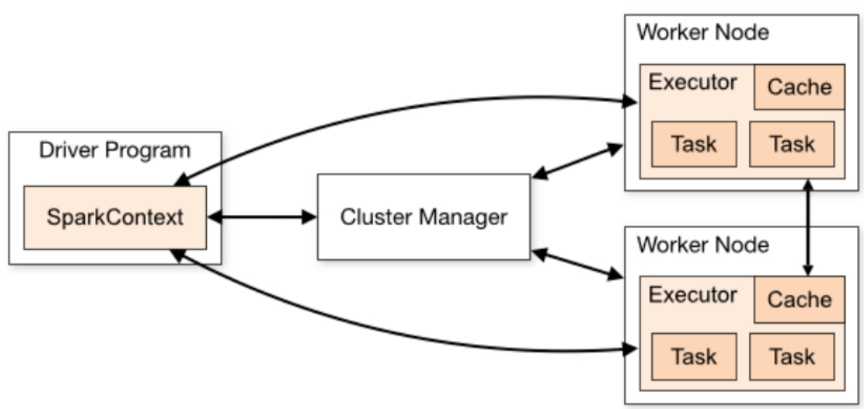
- Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the driver program).

- SparkContext can connect to several types of cluster managers (either Spark's own standalone cluster manager, Mesos or YARN), which allocate resources across applications.

- Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application.

- Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors.

- Finally, SparkContext sends tasks to the executors to run.



Cluster Manager Types

The system currently supports three cluster managers:

- Standalone –a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos –a general cluster manager that can also run Hadoop MapReduce and service applications.
- Hadoop YARN –the resource manager in Hadoop 2.
- Kubernetes –an open-source system for automating deployment, scaling, and management of containerized applications.

Submitting Spark Applications

- The spark-submit script in Sparks bin directory is used to launch applications on a cluster.
- ./bin/run-example utilizes spark-submit
- 2 here is number of distributed tasks/partitions/slices/threads

```
./run-example SparkPi 2
18/04/28 16:30:09 INFO scheduler.TaskSetManager: Finished task 1.0
in stage 0.0 (TID 1) in 398 ms on 192.168.30.11 (executor 0) (1/2)
18/04/28 16:30:09 INFO scheduler.TaskSetManager: Finished task 0.0
in  stage 0.0 (TID 0) in 452 ms on 192.168.30.11 (executor 0) (2/2)
18/04/28 16:30:09 INFO scheduler.TaskSchedulerImpl: Removed TaskSet
0.0, whose tasks have all completed, from pool
18/04/28 16:30:09 INFO scheduler.DAGScheduler: ResultStage 0
(reduce at SparkPi.scala:38) finished in 0.801 s
18/04/28 16:30:09 INFO scheduler.DAGScheduler: Job 0 finished:
reduce at SparkPi.scala:38, took 1.041968 s
Pi is roughly 3.142435712178561
```

Submitting Spark Applications

- Each slice uses a fixed number of smapling points (100000) of uniform random points with unit square, and compute the probablity of the sampling points falling with the circle to approximat value of π.

```
./run-example SparkPi 4
18/04/28 16:34:17 INFO scheduler.TaskSetManager: Finished task 3.0 in
stage 0.0 (TID 3) in 433 ms on 192.168.30.21 (executor 0) (1/4)
18/04/28 16:34:17 INFO scheduler.TaskSetManager: Finished task 0.0 in
stage 0.0 (TID 0) in 485 ms on 192.168.30.21 (executor 0) (2/4)
18/04/28 16:34:17 INFO scheduler.TaskSetManager: Finished task 2.0 in
stage 0.0 (TID 2) in 444 ms on 192.168.30.21 (executor 0) (3/4)
18/04/28 16:34:17 INFO scheduler.TaskSetManager: Finished task 1.0 in
stage 0.0 (TID 1) in 448 ms on 192.168.30.21 (executor 0) (4/4)
18/04/28 16:34:17 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 0.0,
whose tasks have all completed, from pool
18/04/28 16:34:17 INFO scheduler.DAGScheduler: ResultStage 0
(reduce at SparkPi.scala:38) finished in 0.790 s
18/04/28 16:34:17 INFO scheduler.DAGScheduler: Job 0 finished:
reduce at SparkPi.scala:38, took 1.051772 s
Pi is roughly 3.143527858819647

[hadoop@fisher bin]$ ./run-example SparkPi 100
18/04/28 16:41:14 INFO scheduler.DAGScheduler:
Job 0 finished: reduce at SparkPi.scala:38, took 3.874493 s
Pi is roughly 3.1420995142099515

[hadoop@fisher bin]$ ./run-example SparkPi 1000
18/04/28 16:41:59 INFO scheduler.DAGScheduler:
Job 0 finished: reduce at SparkPi.scala:38, took 14.953784 s
Pi is roughly 3.1413859514138593
```

Resilient Distributed Datasets

- The main abstraction Spark provides is the Resilient Distributed Dataset (RDD).
- It is an immutable (persistent) distributed collection of data, which can be partitioned across a cluster of machines.
- RDDs are created by starting with a file in the Hadoop file system, or an existing Scala collection in the driver program, and transforming it.
- Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations.
- Also, RDDs automatically recover from node failures.
- Note that most RDD operations are lazy. Think of an RDD as a description of a series of operations. An RDD is “NOT” data.

Resilient Distributed Datasets

- Let's read in a CSV file of size 15.9 GBs.

```
#Link to a new file
>data2_sc <- spark_read_csv(sc, "data2", "hdfs:///user/huangcs/stock.csv")
>object.size(data2_sc)
15968 bytes
```

Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
16	In-memory table `data2`	Memory Deserialized 1x Replicated	127	100%	1516.5 MB	0.0 B

- After read, the system creates an RDD that says "we will need to load this file". The file is not loaded at this point.

- After "action", the file will be read and action will be executed.

- Moreover, RDD.cache is also a lazy operation. The file is still not read if RDD.cache is run. But now the RDD says "read this file and then cache the contents".

- Again, after "action", the file will be read and action will be executed.

Resilient Distributed Datasets

- The cache behavior depends on the available memory. If the file does not fit in the memory, for example, then RDD.action will fall back to the usual behavior and re-read the file.

- It facilitates two types of data operations: transformation and action.

- A transformation is an dataset operation, such as filter() and map() on an RDD that create another RDD.

- An action, on the other hand, is a computation on RDDs, such as count() and collect(). An action actually triggers a creation/computation of RDDs, returns a value back to the Master node, or writes RDDs to a persistent storage system.

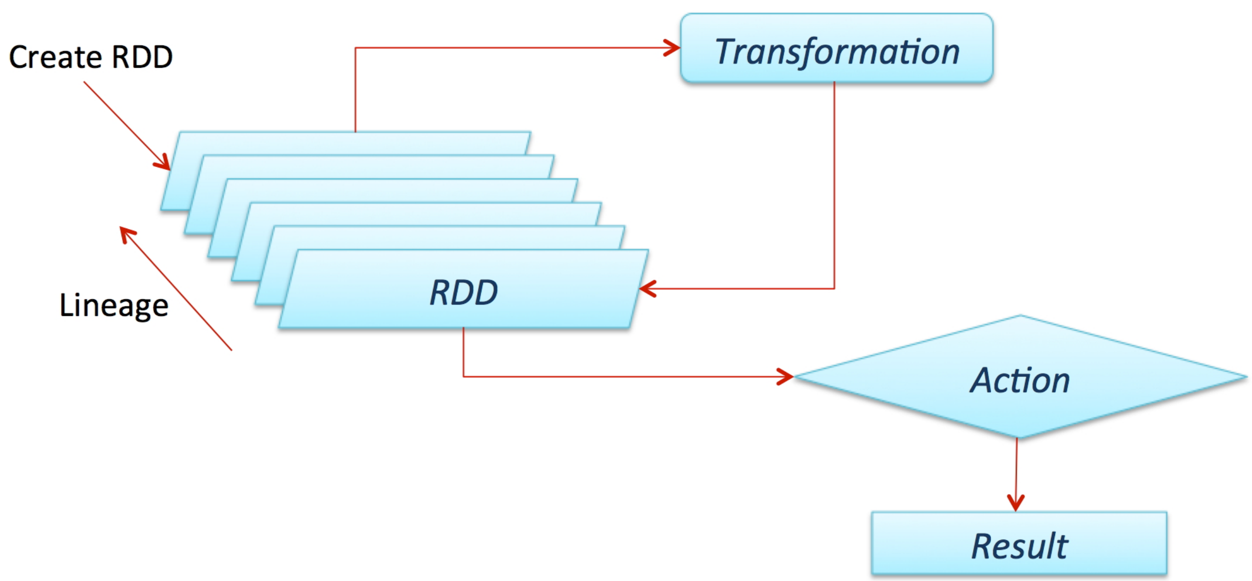
- Transformations are lazily evaluated, in that they do not run until an action performs it. Spark Master nodes remembers all the transformations applied to an RDD, a lineage of RDDs.

- RDDs would be less suitable for applications that make asynchronous fine grained updates to shared state.

RDD Operations in Spark

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	: RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	: RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	: RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	: RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	: RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	: RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	: (RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	: (RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	: (RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	: RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	: RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	: RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	: RDD[T] ⇒ Long
	$collect()$	: RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	: RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	: RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	: Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.



Spark Shell

- This section introduces Spark using a modified Scala shell, `spark-shell`. Since Spark is built using Scala languages, the examples shown here are also in Scala.

- Spark's shell provides powerful APIs to interactively analyze datasets. `spark-shell` launches a `scala>` prompt which pre-loads the Spark jar and other dependencies.

```
huangcs@fisher:~$ spark_shell
Spark context Web UI available at http://hnamenode:4040
Spark context available as 'sc' (master = spark://hnamenode:7077,
                                 app id = app-20180513172334-0026).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.0
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_101)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20180513172334-0026 (kill)	Spark shell	8	1024.0 MB	2018/05/13 17:23:34	huangcs	RUNNING	1.1 min

Running Drivers (0)

Submission ID	Submitted Time	Worker	State	Cores	Memory	Main Class

Spark Shell

- Spark context available as ’sc’.
- Create an RDD using /user/huangcs/derby.log file.

```
scala> val textFile = sc.textFile("/user/huangcs/derby.log")
textFile: org.apache.spark.rdd.RDD[String] = /user/huangcs/derby.log
MapPartitionsRDD[1] at textFile at <console>:24
```

- RDDs are bundled with actions that return values and transformations to new RDDs.
- Counting the number of items in an textFile RDD.

```
scala> textFile.count()
res1: Long = 4916
```

- Returning the first item of textFile RDD.

```
scala> textFile.first()
res2: String = Wed Apr 18 16:54:30 CST 2018 Thread[main,5,main] Cleanup action starting
```

Spark Shell

- Applying `filter` transformation points to a new RDD, which is a subset of the original RDD.

- Filter items in `textFile` RDD containing *"Thread"* and count the occurrences.

```
scala> val linesWithThread = textFile.filter(line => line.contains("Thread"))
linesWithThread: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3]at filter at <console>:25

scala> linesWithThread.count()
res3: Long = 8
```

- RDD transformations can be chained.

```
scala> val linesWithThread = textFile.filter(line => line.contains("Thread")).count()
linesWithThread: Long = 8

scala> :q
huangcs@fisher:~/Spark_App/simple$
```

- Check point and Cache can be used to raise efficiency.

  Homework: Use "sc" to read in a text file, and count the number of lines.

Writing and Submitting a Spark Application

- This section takes you through
  1. creating a Spark application,
  2. submitting it to a Spark cluster for execution,
  3. and reviewing the example bundled with Apache Spark.

Step 1: Create a directory structure for the application src and target directories.

```
fisher/home/huangcs/Spark_App : mkdir simple
fisher/home/huangcs/Spark_App/simple : mkdir -p src/main/scala
```

Step 2: Edit an sbt configuration file, `simple.sbt` which explains that Spark is a dependency. This file also adds a repository that Spark depends on:

```
fisher/home/huangcs/Spark_App/simple 65: cat simple.sbt
name := "Simple Project"

version := "1.0"

scalaVersion := "2.11.4"

libraryDependencies += "org.apache.spark" %% "spark-core" %
        "2.3.0"

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

Writing and Submitting a Spark Application

- set `SBT_HOME` and `PATH`

```
[huangcs@fisher]$ export SBT_HOME=/usr/local/sbt
[huangcs@fisher]$ export PATH=$SBT_HOME/bin:$PATH
```

- Make sure http://repo.akka.io/releases/ contains the right versions.

Step 3: Write a spark program to count the number of a's and b's in a user-defined text file. Create a file `SimpleApp.scala` in `src/main/scala` directory with the following code:

```
/*** SimpleApp.scala ***/
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
 def main(args: Array[String]) {
   val logFile = "/user/huangcs/stock.csv"
   val conf = new SparkConf().setAppName("SimpleApp")
   val sc = new SparkContext(conf)
   val logData = sc.textFile(logFile, 2).cache()
   val numAs = logData.filter(line =>
                      line.contains("a")).count()
   val numBs = logData.filter(line => line.contains("b")).count()
   println("Lines with a: %s, Lines with b: %s".format(numAs, numBs)) }}
```

- `/user/huangcs/stock.csv` is a file in Hadoop file system, so is the output file, if there is any.

Writing and Submitting a Spark Application

Step 4: The application is packaged in to a jar file that is deployed across the clusters and run.

```
huangcs@fisher:~/Spark_App/simple$ sbt package
[info] Set current project to Simple Project (in build file:/home/huangcs/Spark_App/simple/)
[warn] Scala version was updated by one of library dependencies:
[warn]  * org.scala-lang:scala-library:(2.11.4, 2.11.0) -> 2.11.8
[warn] To force scalaVersion, add the following:
[warn]  ivyScala := ivyScala.value map { _.copy(overrideScalaVersion = true) }
[warn] Run 'evicted' to see detailed eviction warnings
[success] Total time: 1 s, completed May 13, 2018 4:30:41 PM
```

Step 5: Submit the application to the Spark cluster using the bin/spark-submit script.

```
huangcs@fisher:~/Spark_App/simple$ more myspark_submit
/usr/local/spark/spark-2.3.0-bin-hadoop2.7/bin/spark-submit  \
      --class "SimpleApp"\
      --master spark://192.168.30.2:7077 \
      --executor-memory 30G  \
  /home/huangcs/Spark_App/simple/target/scala-2.11/simple-project_2.11-1.0.jar
```

Homework: Write a spark application to read in a CVS file and do some analysis on it.

Completed Applications (25)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20180511161913-0024	SparkWordCount	8	4.0 GB	2018/05/11 16:19:13	huangcs	FINISHED	4 s
app-20180511161806-0023	SparkWordCount	8	4.0 GB	2018/05/11 16:18:06	huangcs	FINISHED	3 s
app-20180511161514-0022	SparkWordCount	8	4.0 GB	2018/05/11 16:15:14	huangcs	FINISHED	3 s
app-20180511161349-0021	SparkWordCount	8	4.0 GB	2018/05/11 16:13:49	huangcs	FINISHED	3 s
app-20180511161012-0020	SparkWordCount	8	4.0 GB	2018/05/11 16:10:12	huangcs	FINISHED	1 s
app-20180511154303-0019	SparkWordCount	8	4.0 GB	2018/05/11 15:43:03	huangcs	FINISHED	4 s
app-20180511152644-0018	SimpleApp	64	30.0 GB	2018/05/11 15:26:44	huangcs	FINISHED	31 s

Introduction to Sparklyr

- The sparklyr package provides a ***dplyr*** and ***DBI*** backends with wrapper functions to Spark DataFrames and Spark's distributed machine learning library (MLlib).
- Make the Spark connection using `spark_connect`

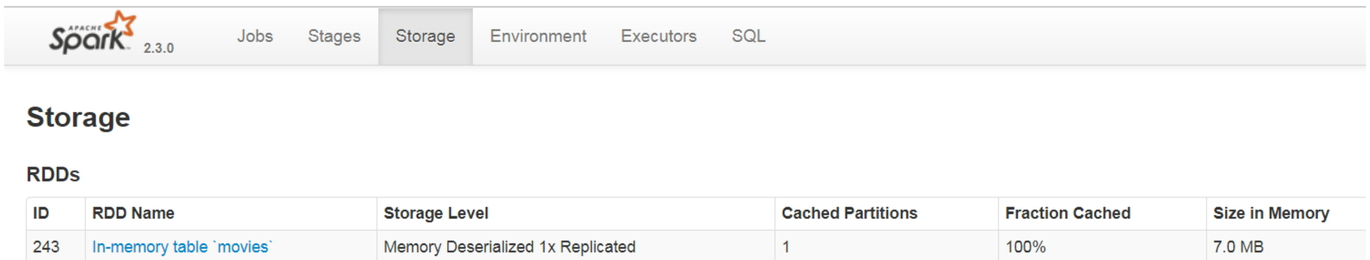
```
library(sparklyr); library(DBI); library(dplyr); library(ggplot2)
# Setting Config

conf <- spark_config()
# Need to be less than the memory in each node
conf$spark.executor.memory <- "15GB"
# Need to be less than the number of core in each node
conf$spark.executor.cores <- 8
# Also, (the number of executor)X(executor memory) < memory in each node

# Please do NOT turn this off,
# You can get all cores if you turn this off
#conf$spark.dynamicAllocation.enabled <- "false"

sc <- spark_connect(master="spark://192.168.30.2:7077",
      version = "2.3.0",
      config = conf,
      spark_home = "/usr/local/spark/spark-2.3.0-bin-hadoop2.7/")
```

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Submitted Time
app-20180522164830-0007 (kill)	sparklyr	8	15.0 GB	2018/05/22 16:48:30

Sparklyr in Action

```
library(ggplot2movies) #load ggplot2movies
# Move R dataframe to Spark. Movies dataset as the example.
# An "movies" RDD is created in Spark memory system
# View it at http://192.168.30.2:4040/storage/
movies_sdf = copy_to(sc, df = ggplot2movies::movies, name = "movies",
                     overwrite = T)
```



Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory
243	In-memory table `movies`	Memory Deserialized 1x Replicated	1	100%	7.0 MB

```
> head(movies_sdf) # first few observations
# Source:   lazy query [?? x 24]
# Database: spark_connection
                     title  year length budget rating votes    r1    r2
                     <chr> <int>  <int>  <int>  <dbl> <int> <dbl> <dbl>
1                        $  1971    121     NA    6.4   348   4.5   4.5
2         $1000 a Touchdown  1939     71     NA    6.0    20   0.0  14.5
3   $21 a Day Once a Month  1941      7     NA    8.2     5   0.0   0.0
4                  $40,000  1996     70     NA    8.2     6  14.5   0.0
5 $50,000 Climax Show, The  1975     71     NA    3.4    17  24.5   4.5
6                   $pent  2000     91     NA    4.3    45   4.5   4.5
# ... with 16 more variables: r3 <dbl>, r4 <dbl>, r5 <dbl>, r6 <dbl>,
#   r7 <dbl>, r8 <dbl>, r9 <dbl>, r10 <dbl>, mpaa <chr>, Action <int>,
#   Animation <int>, Comedy <int>, Drama <int>, Documentary <int>,
#   Romance <int>, Short <int>
```

Sparklyr in Action

```
# List all available Spark data frames in Spark memory
dbListTables(sc)
[1] "data"   "movies"

# Check Spark log file for monitoring purposes
>spark_log(sc, n = 2)

2018-05-23 14:11:46 WARN  SparkSession$Builder:66 - Using an existing
SparkSession; some configuration may not take effect.
2018-05-23 14:11:46 WARN  SparkSession$Builder:66 - Using an existing
SparkSession; some configuration may not take effect.


# "Explain" complex query plan before you actually hit "run"
>movies_sdf %>% select(title, rating) %>% explain()
<SQL>
SELECT `title` AS `title`, `rating` AS `rating`
FROM `movies`

<PLAN>

> movies_sdf %>% select(title, rating)
# Source:   lazy query [?? x 2]
# Database: spark_connection
                      title rating
                      <chr>  <dbl>
 1                        $    6.4
 2        $1000 a Touchdown    6.0
 3   $21 a Day Once a Month    8.2
 4                  $40,000    8.2
 5 $50,000 Climax Show, The    3.4
```

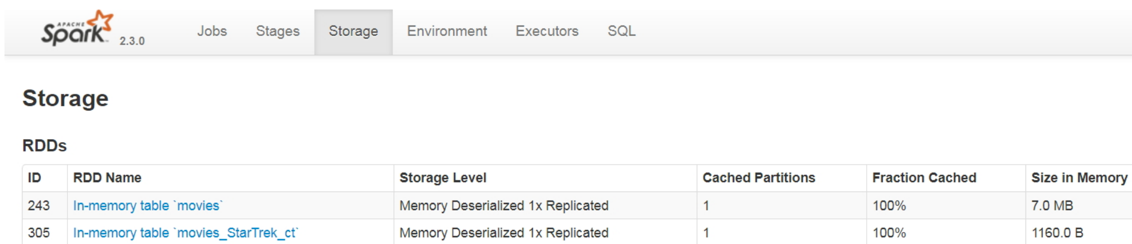
Sparklyr in Action

```
# The good, the bad, and the movie title < 10?
>movies_sdf %>% select(title, length, rating ) %>%
  filter((rating > 9 | rating < 2) &  nchar(title) < 10) %>%
  arrange(desc(nchar(title)))

# Source:     lazy query [?? x 3]
# Database:   spark_connection
# Ordered by: desc(nchar(title))
       title length rating
       <chr>  <int>  <dbl>
 1 Cerebrium      8    9.7
 2 Dvoynikat     98    9.2
 3 Challenge    152    9.8
 4 Causality      9    9.7
# ... with more rows

# Give me "Star Trek"!
> movies_sdf_StarTrek = movies_sdf %>%
   select(title, rating, budget) %>% filter(title %like% "%Star Trek%")

# Source:   lazy query [?? x 3]
# Database: spark_connection
                                             title rating   budget
                                             <chr>  <dbl>    <int>
 1              Star Trek III: The Search for Spock    6.2 17000000
 2                    Star Trek IV: The Voyage Home    7.1 25000000
 3                  Star Trek V: The Final Frontier    4.8 27800000
 4           Star Trek VI: The Undiscovered Country    6.9 30000000
 5 Star Trek the Experience: The Klingon Encounter    7.0 70000000
# ... with more rows
```

Sparklyr in Action

- Notice that previous SDF operations did not actually create named SDFs/RDDs in Spark, as most of them are transformations and thus lazily evaluated by Spark.

```
# Create the temporary table "movies_StarTrek_ct"
>compute(movies_sdf_StarTrek, "movies_StarTrek_ct")

dbListTables(sc) # It's now in Spark as a temporary RDD
> dbListTables(sc)
[1] "data"                    "movies"
[3] "movies_startrek_ct"
```



Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory
243	In-memory table `movies`	Memory Deserialized 1x Replicated	1	100%	7.0 MB
305	In-memory table `movies_StarTrek_ct`	Memory Deserialized 1x Replicated	1	100%	1160.0 B

```
# You can actually create (cache) a Spark DataFramein memory.
# It may improve performance. Remember to check your Spark UI/Tab.
> tbl_cache(sc, "movies_StarTrek_ct")

# Create an SDF pointer to existing RDD
>movies_StarTrek_ct_sdf= tbl(sc, from = "movies_StarTrek_ct")

# For small SDFs, we may save the result as local R data frames
>movies_StarTrek_ct_df = collect(movies_StarTrek_ct_sdf)

# We can surely save or remove RDDs
>tbl_uncache(sc, "movies_StarTrek_ct")
```

A bit about Sparklyr

```
> sc
$master
[1] "spark://192.168.30.2:7077"
$method
[1] "shell"
$app_name
[1] "sparklyr"
$config
$config$spark.env.SPARK_LOCAL_IP.local
[1] "127.0.0.1"
.................
..................
$spark_context
<jobj[9]>
  org.apache.spark.SparkContext
  org.apache.spark.SparkContext@4d13208e
$java_context
<jobj[10]>
  org.apache.spark.api.java.JavaSparkContext
  org.apache.spark.api.java.JavaSparkContext@413a9cc4
attr(,"class")
[1] "spark_connection"       "spark_shell_connection" "DBIConnection"
```

The dplyr package provides a flexible grammar of data manipulation. It's the next iteration of plyr, focused on tools for working with data frames (hence the d in the name).

A bit about Sparklyr

```
# The Spark Context represents the connection to a Spark cluster,
# and can be used to create RDDs, accumulators and broadcast
# variables on that cluster.

# Invoke a Method on a JVM Object
count_lines <- function(sc, file) {
>  spark_context(sc) %>%
    invoke("textFile", file, 1L) %>%
    invoke("count")
}

>count_lines(sc, "/user/huangcs/input.txt")

[1] 1000
```

- Read a CSV file

```
# test.csv
#   ID   sex
#    1   male
#    2  female
#    3   male
#    4  female

>data1_sc <- spark_read_csv(sc, "data","hdfs:///user/d012040001/test.csv")
# sdf_pivot construct a pivot table over a Spark Dataframe
>sdf_pivot(data1_sc, sex~ID, list(sex = "count"))

# Source:   table<sparklyr_tmp_75986d4a84be> [?? x 5]
# Database: spark_connection
     sex   `1`   `2`   `3`   `4`
   <chr> <dbl> <dbl> <dbl> <dbl>
1 female   NaN     1   NaN     1
2   male     1   NaN     1   NaN
```

A bit about Sparklyr

```
>group_by(data1_sc, sex) %>% count
# Source:   lazy query [?? x 2]
# Database: spark_connection
# Groups:   sex
     sex     n
   <chr> <dbl>
1 female     2
2   male     2

#Link to a new file
>data2_sc <- spark_read_csv(sc, "data2", "hdfs:///user/huangcs/stock.csv")
>object.size(data2_sc)
15968 bytes
```

- Stages

Stages for All Jobs

Completed Stages: 7

Completed Stages (7)

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input
6	collect at utils.scala:196	+details	2018/05/24 10:52:41	96 ms	1/1	
5	collect at utils.scala:196	+details	2018/05/24 10:52:40	0.1 s	127/127	1516.5 MB
4	sql at NativeMethodAccessorImpl.java:0	+details	2018/05/24 10:52:40	0.1 s	1/1	
3	sql at NativeMethodAccessorImpl.java:0	+details	2018/05/24 10:52:16	24 s	127/127	15.9 GB
2	csv at NativeMethodAccessorImpl.java:0	+details	2018/05/24 10:51:49	27 s	127/127	15.9 GB
1	csv at NativeMethodAccessorImpl.java:0	+details	2018/05/24 10:51:47	2 s	1/1	128.1 MB
0	collect at utils.scala:43	+details	2018/05/24 10:51:45	1 s	1/1	

A bit about Sparklyr

- Storage

Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
16	In-memory table `data2`	Memory Deserialized 1x Replicated	127	100%	1516.5 MB	0.0 B

```
# Count the number of transactions each day and order them descendingly
>a1 = Sys.time()
>group_by(data2_sc, day) %>% count %>% arrange(desc(day))
>a2 = Sys.time()
> a2 - a1

# Source:     lazy query [?? x 2]
# Database:   spark_connection
# Groups:     day
# Ordered by: desc(day)
         day      n
       <chr>  <dbl>
 1 2008/03/20 377387
 2 2008/03/19 476265
 3 2008/03/18 376580
 4 2008/03/17 542230
 5 2008/03/14 626088
# ... with more rows

Time difference of 0.9095645 secs
```

A bit about Spark SQL in Sparklyr

- Sparklyr supports Spark SQL via functions in DBI. However, SQL via DBI is not intended to be used to create new SDFs/RDDs. It should only be used in query purposes.

```
# Save query result as local R data frame
movies_10obs = dbGetQuery(sc, "select * from movies limit 10")
# Or a query result in Spark
rs= dbSendQuery(sc, "select title, rating from movies limit 100")
dbFetch(rs, n = 5)
# Remove objects
dbClearResult(rs); rm(rs)
# Use dbWriteTable() to create a new RDD, diamonds
# with "log of Price"
dbWriteTable(sc, "diamonds", mutate(diamonds,
             logPrice= log(price)))
# Create a tibble pointing to the cached RDD
diamonds_sdf= tbl(sc, "diamonds")
tbl_cache(sc, "diamonds"); diamonds_sdf
```

Hands on Spark MLlib

- Although some up-to-date functions are not available, sparklyr provides wrapper functions that allows us to access the power of Spark MLlib.

```
# 70% as the training set
diamonds_train_sdf= sdf_sample(diamonds_sdf, 0.7, seed = 1 )
nrow(diamonds_train_sdf)
# 30% as the testing set
diamonds_test_sdf= setdiff(diamonds_sdf, diamonds_train_sdf)
nrow(diamonds_test_sdf)
# Or we can just use sdf_partition() to make your life easier
diamonds_train_test= sdf_partition(diamonds_sdf, training = 0.7,
                                   test = 0.3, seed =1)
# Actually create RDDs to faciliatemodel fitting
compute(diamonds_train_sdf, "diamonds_train", temporary = F)
tbl_cache(sc, "diamonds_train", force = T)
compute(diamonds_test_sdf, "diamonds_test", temporary = F)
tbl_cache(sc, "diamonds_test", force = T)
# General linear model
diamonds_lm= ml_linear_regression(logPrice˜ carat + cut + clarity +
                                + y + z, data = diamonds_train_sdf)
summary(diamonds_lm)
```

Hands on Spark MLlib

```
# We can also save model to local filesystem.
# It is actually a folder instead of file
ml_save(diamonds_lm, file = "./diamonds_spark_lm.RData")
rm(diamonds_lm) # delete the model
# Then load it back later.
# Unfortunately, until sparklyr 0.55, only model coefficients
# can be serialized.
# The reloaded model can only be used for prediction
diamonds_lm= ml_load(sc, "./diamonds_spark_lm.RData")
# Predicted response/outcome column is called "prediction"
diamonds_lm_testpred= sdf_predict(diamonds_lm,
                        newdata = diamonds_test_sdf)
diamonds_lm_testpred %>% select(actualLogPrice= logPrice,
                     %>% predictedLogPrice = prediction)
                     %>% head()
# Compute RMSE
diamonds_lm_testpred %>%
   transmute(MSE = mean((logPrice-prediction)^2) %>%
   head(1) %>%
   transmute(RMSE = round(sqrt(MSE), 4) ) %>% collect()
```

Hands on Spark MLlib

- An example on Logistic Regression

```
install.packages("titanic")
library(titanic)

## Load the datasets
data("titanic_train")
data("titanic_test")

## Setting Survived column for test data to NA
titanic_test$Survived <- NA

## Combining Training and Testing dataset
complete_data <- rbind(titanic_train, titanic_test)

## Check data structure
str(complete_data)

## Let's check for any missing values in the data
colSums(is.na(complete_data))

colSums(complete_data=='')

## Check number of uniques values for each of the column to find out
## columns which we can convert to factors
sapply(complete_data, function(x) length(unique(x)))

## Missing values imputation
complete_data$Embarked[complete_data$Embarked==""] <- "S"
complete_data$Age[is.na(complete_data$Age)] <- median(complete_data$Age,na.rm=T)
```

Hands on Spark MLlib

```
## Removing Cabin as it has very high missing values, passengerId,
## Ticket and Name are not required
library(dplyr)
titanic_data <- complete_data %>% select(-c(Cabin, PassengerId, Ticket, Name))

## Converting "Survived","Pclass","Sex","Embarked" to factors
for (i in c("Survived","Pclass","Sex","Embarked")){
  titanic_data[,i]=as.factor(titanic_data[,i])
}

#install.packages("dummies")

## Create dummy variables for categorical variables
library(dummies)
titanic_data <- dummy.data.frame(titanic_data,
  names=c("Pclass","Sex","Embarked") , sep="_")

## Splitting training and test data
train <- titanic_data[1:667,]
test <- titanic_data[668:889,]

## Logestic Regression Model Creation
model <- glm(Survived ~.,family=binomial(link='logit'),data=train)

## Model Summary
summary(model)
```

Hands on Spark MLlib

```
Call:
glm(formula = Survived ~ ., family = binomial(link = "logit"),  data = train)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.3804  -0.6562  -0.4300   0.6392   2.3950

Coefficients: (3 not defined because of singularities)
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.373105   0.319779  -4.294 1.76e-05 ***
Pclass_1     2.175104   0.359365   6.053 1.42e-09 ***
Pclass_2     1.302268   0.271680   4.793 1.64e-06 ***
Pclass_3           NA         NA      NA       NA
Sex_female   2.677814   0.226863  11.804  < 2e-16 ***
Sex_male           NA         NA      NA       NA
Age         -0.031671   0.008945  -3.540 0.000399 ***
SibSp       -0.248975   0.123365  -2.018 0.043570 *
Parch       -0.091603   0.141950  -0.645 0.518718
Fare        -0.001397   0.003179  -0.440 0.660254
Embarked_C   0.431447   0.271693   1.588 0.112288
Embarked_Q   0.533193   0.369337   1.444 0.148837
Embarked_S         NA         NA      NA       NA
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 891.99  on 666  degrees of freedom
Residual deviance: 605.78  on 657  degrees of freedom
AIC: 625.78
```

Hands on Spark MLlib

- An example on Logistic Regression

```
>train_tbl <- copy_to(sc, train)

>logistic_train=ml_logistic_regression(
            formula = Survived ~ .,x = train_tbl,family="binomial")


Formula: Survived ~ .

Coefficients:
 (Intercept)     Pclass_1     Pclass_2     Pclass_3    Sex_female      Sex_male          Age
-0.209291549  1.546872086  0.674036620   -0.628231038  1.853785141  -0.824027540   -0.031670887
    SibSp         Parch         Fare          Embarked_C    Embarked_Q    Embarked_S
-0.248973819   -0.091603564  -0.001397229  0.719891103  0.821636898
0.288444229


>logistic_train1=ml_logistic_regression(
    formula = Survived ~ Pclass_1 + Pclass_2 + Sex_female
     + Age+ SibSp + Parch + Fare + Embarked_C + Embarked_Q
                                ,x = train_tbl,family="binomial")
Coefficients:
 (Intercept)     Pclass_1     Pclass_2      Sex_female     Age           SibSp
-1.373105934   2.175106131   1.302268994   2.677814646  -0.031670924  -0.248974498
  Parch          Fare         Embarked_C     Embarked_Q
-0.091603424    -0.001397245  0.431446006  0.533193502
```

Hands on Spark MLlib

- An example on Random Forest

```
# Copy iris to Spark system
>iris_tbl <- copy_to(sc, iris, "iris", overwrite = TRUE)
# Run the ml_random_forest function,
# Using two variables, Petal_Length,
#  Petal_Width to classify Species
>rf_model <- iris_tbl %>%
  ml_random_forest(Species ~ Petal_Length + Petal_Width,
  type = "classification")

# Output the performance
>rf_predict <- sdf_predict(model=rf_model, x=iris_tbl) %>%
  ft_string_indexer("Species", "Species_idx") %>%
  collect

>table(rf_predict$Species_idx, rf_predict$prediction)

     0  1  2
  0 49  1  0
  1  0 50  0
  2  0  0 50
# We only missed 1 classification, i.e. we miss 0->1.
```

Homeworks:

- Load Hitters and Default datasets in package ISLR. Save them to Spark memory.
- Perform some statistical analysis on those RDDs.

Spark Reads a CSV File

- readCSV.sbt

```
name := "Read CSV Project"

version := "1.0"

scalaVersion := "2.11.4"

libraryDependencies ++= Seq(
      "org.apache.spark" %% "spark-core" % "2.3.0",
      "org.apache.spark" %% "spark-sql"  % "2.3.0"
)

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

Spark Reads a CSV File

- verb—src/main/scala/readCSV.scala—

```
/*** readCSVApp.scala ***/
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.sql.SQLContext

object readCSVApp {
  def main(args: Array[String]) {
    val logFile = "/user/huangcs/matrix.csv"
    val conf = new SparkConf().setAppName("readCSVApp")
    val sc = new SparkContext(conf)
    val sqlContext: SQLContext = new SQLContext(sc)
    val matrix = sqlContext.read.format("csv")
                  .option("header", "true")
                  .option("inferSchema", "true")
                  .load(logFile)
    matrix.show()
    matrix.printSchema()
}}

+---+---+---+
|  1|  2|  3|
+---+---+---+
|  a|  b|  c|
+---+---+---+

root
 |-- 1: string (nullable = true)
 |-- 2: string (nullable = true)
 |-- 3: string (nullable = true)
```

Spark Linear Regression

```
#Preparing Data, with polynomial as y = a + bx.
a = -3; b = 2;
#number of samples, x = var1
data_size<-1000
# var is random variable which comes from a uniform distribution
var1 <- runif(data_size, 1, 10)

#res is function of x, 1 and noise
res <- b*var1 + a + rnorm(data_size)
sum_data=cbind(var1, res)
write.table(sum_data, file = "D:\\input.txt", sep = ",",
            row.names = FALSE, col.names = FALSE)
#Data are arranged as {x_i, y_i}
```

- verb—src/main/scala/linReg.scala—

```
/*** linRegApp.scala ***/
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object linRegApp {
  def main(args: Array[String]) {
      val logFile = "/user/huangcs/input.txt"
      val conf = new SparkConf().setAppName("linRegApp")
      val sc = new SparkContext(conf)
      val input = sc.textFile(logFile, 2).cache()

      val xy = input.map( line => {
                            val parts = line.split(',')
                            (parts(0).toDouble, parts(1).toDouble)
                          })
```

Spark Linear Regression

```
      var m = input.count().toDouble
      var theta0 : Double = -2.0
      var theta1 : Double = 4.0
      var e1sum : Double = 0.0
      var e2sum : Double = 0.0

      for( i <- 1 to 1000){
        e1sum = xy.map(
          line => {
            var prediction = theta0 + theta1 * line._1
            var diff = prediction - line._2
             (diff)
            }).sum()
        e2sum = xy.map(
          line => {
            var prediction = theta0 + theta1 * line._1
            var diff = (prediction - line._2) * line._1
             (diff)
            }).sum()
        theta0 = theta0 - 0.01 * ( 1.0 / m ) * e1sum
        theta1 = theta1 - 0.01 * ( 1.0 / m ) * e2sum
        }
        println("Answer:=", theta0, theta1)
}}

(Answer:=,-2.8965035694582157,1.9906901135401143)
```

Goodbye Spark

I'll be back.

"We" will work on H_2O, Sparklying Water and/or Deep water.

H_2O

Deep Water

Sparkling Water