Data Science
&
Big Data

Agenda

- **Data Cleaning, Profiling, Performance**
- String Processing
- Trends & Outliers
- Big Data Processing with Spark
- Final Project

lecture07.1.clean.data.ipynb

Agenda

- Data Cleaning, Profiling, Performance
- **String Processing**
- Trends & Outliers
- Big Data Processing with Spark
- Final Project

lecture07.2.strings.ipynb

Agenda

- Data Cleaning, Profiling, Performance
- String Processing
- **Trends & Outliers**
- Big Data Processing with Spark
- Final Project

lecture07.3.trends.ipynb

Agenda

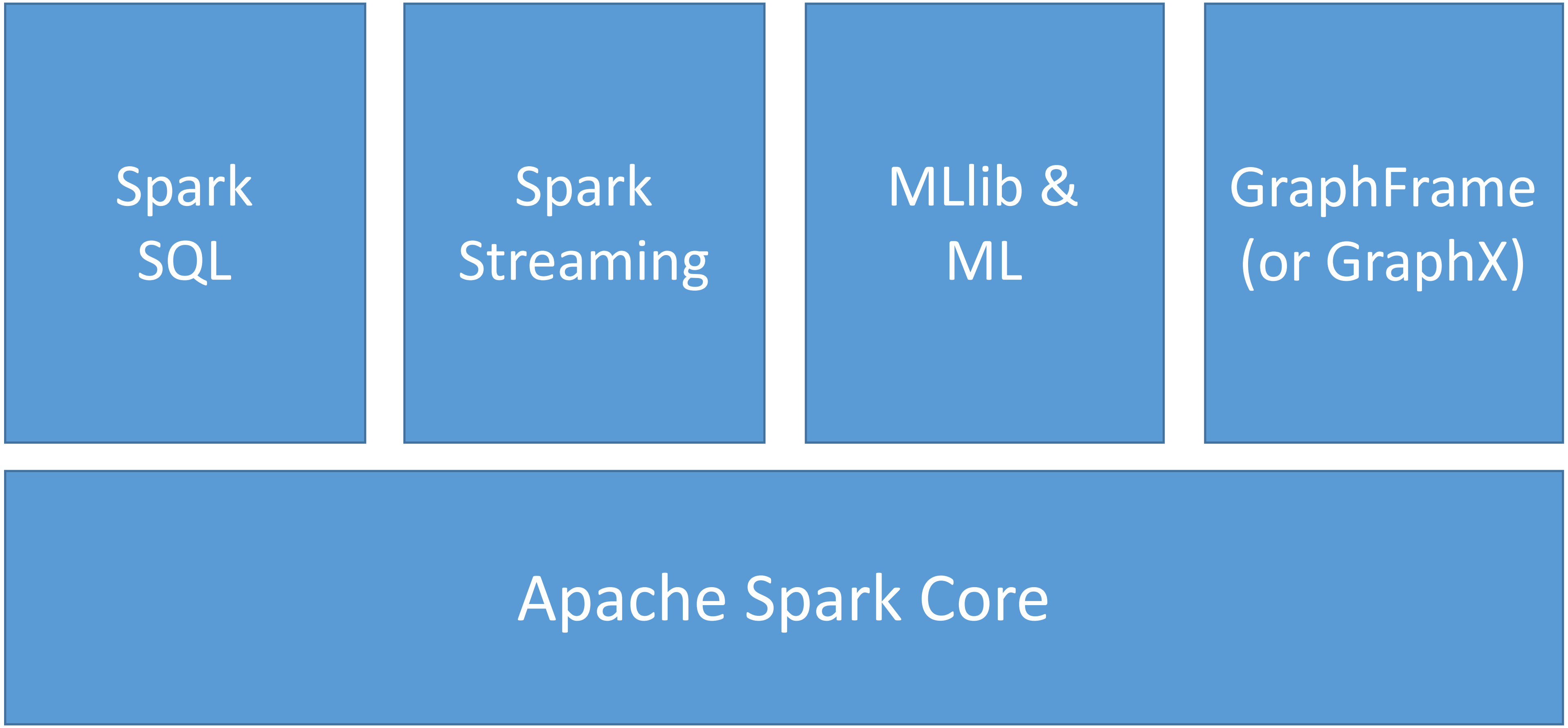
- Data Cleaning, Profiling, Performance
- String Processing
- Trends & Outliers
- **Big Data Processing with Spark**
- Final Project

Big Data Processing with

APACHE Spark™

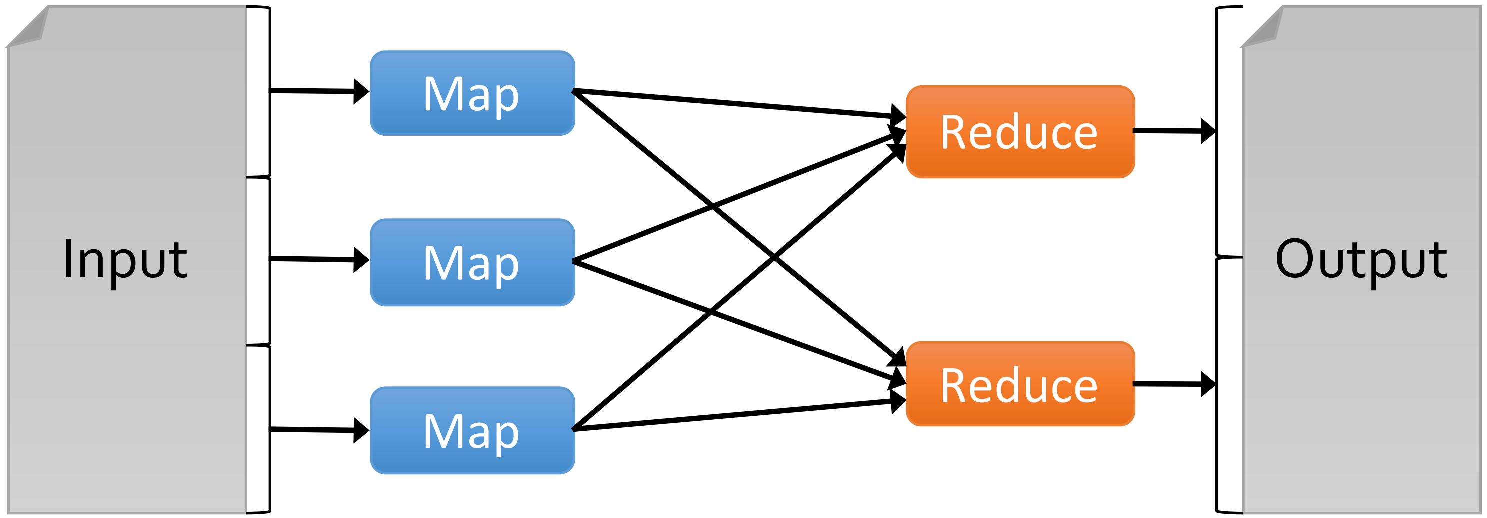
https://databricks.com/ce

Apache Spark



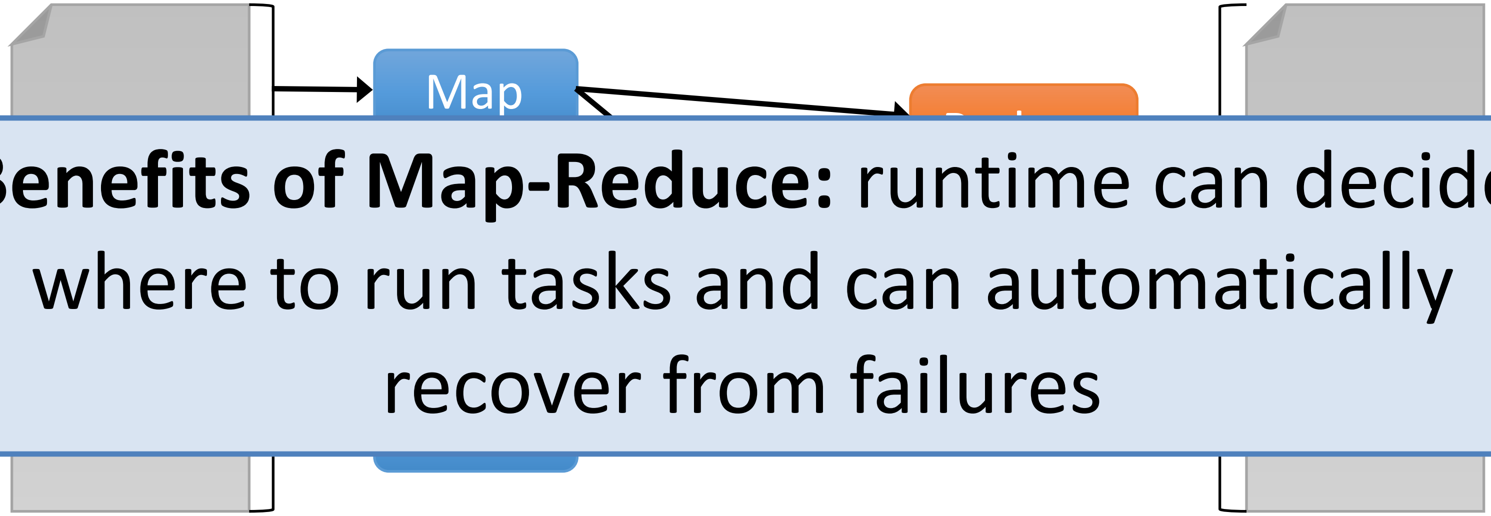
Map-Reduce

- Commodity clusters have become an important computing platform for a variety of applications
- Current popular programming models for clusters transform data flowing from stable storage to stable storage
- Great for “acyclic” data flow



Map-Reduce

- Commodity clusters have become an important computing platform for a variety of applications
- Current popular programming models for clusters transform data flowing from stable storage to stable storage
- Great for "acyclic" data flow



Benefits of Map-Reduce: runtime can decide where to run tasks and can automatically recover from failures

Why Spark?

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

Spark Goals

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

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

RDDs

- The primary data abstraction in Spark
- An **immutable**, **partitioned**, **logical** collection of records
- Need **not be materialized**, but rather contains information to rebuild a dataset from stable storage
- Track **lineage** information to efficiently recompute lost data
- Built using bulk **transformations** on other RDDs
- Can be **cached** for future reuse

RDDs : 3 ways to construct

- by parallelizing existing **Python collections** (lists)
- by transforming an **existing RDDs**
- from **files** in HDFS or any other storage system

RDDs : examples

```
> data = range(10)
  data
```

```
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Command took 0.04s
```

```
> rdd = sc.parallelize(data, 5)
  rdd
```

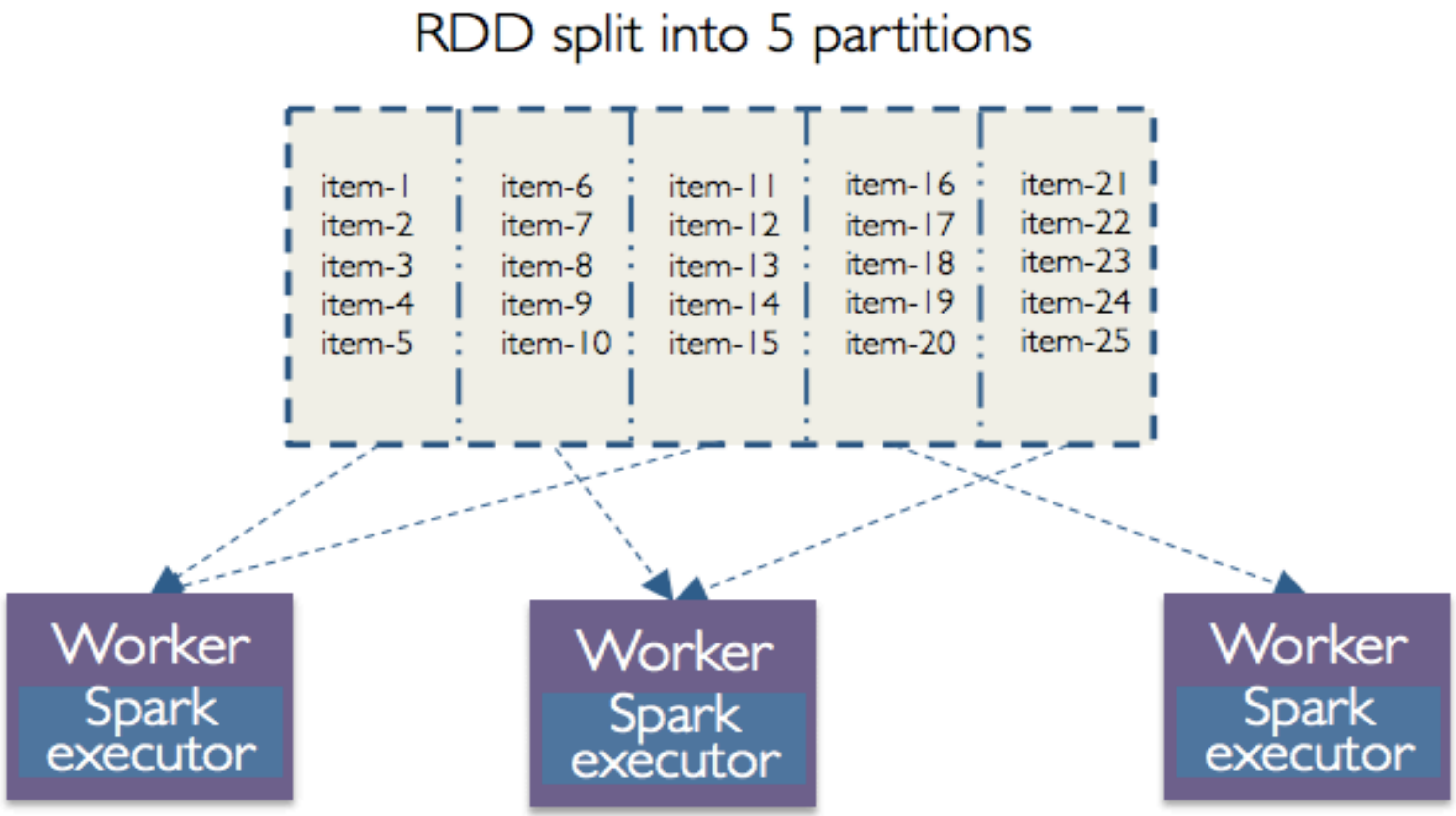
```
Out[3]: ParallelCollectionRDD[88] at parallelize at PythonRDD.scala:423
Command took 0.07s
```

```
> rdd = sc.textFile("spark.joins", 5)
  rdd
```

```
Out[4]: spark.joins MapPartitionsRDD[137] at textFile at NativeMethodAccessorImpl.java:-2
Command took 0.13s
```

RDDs : partitions

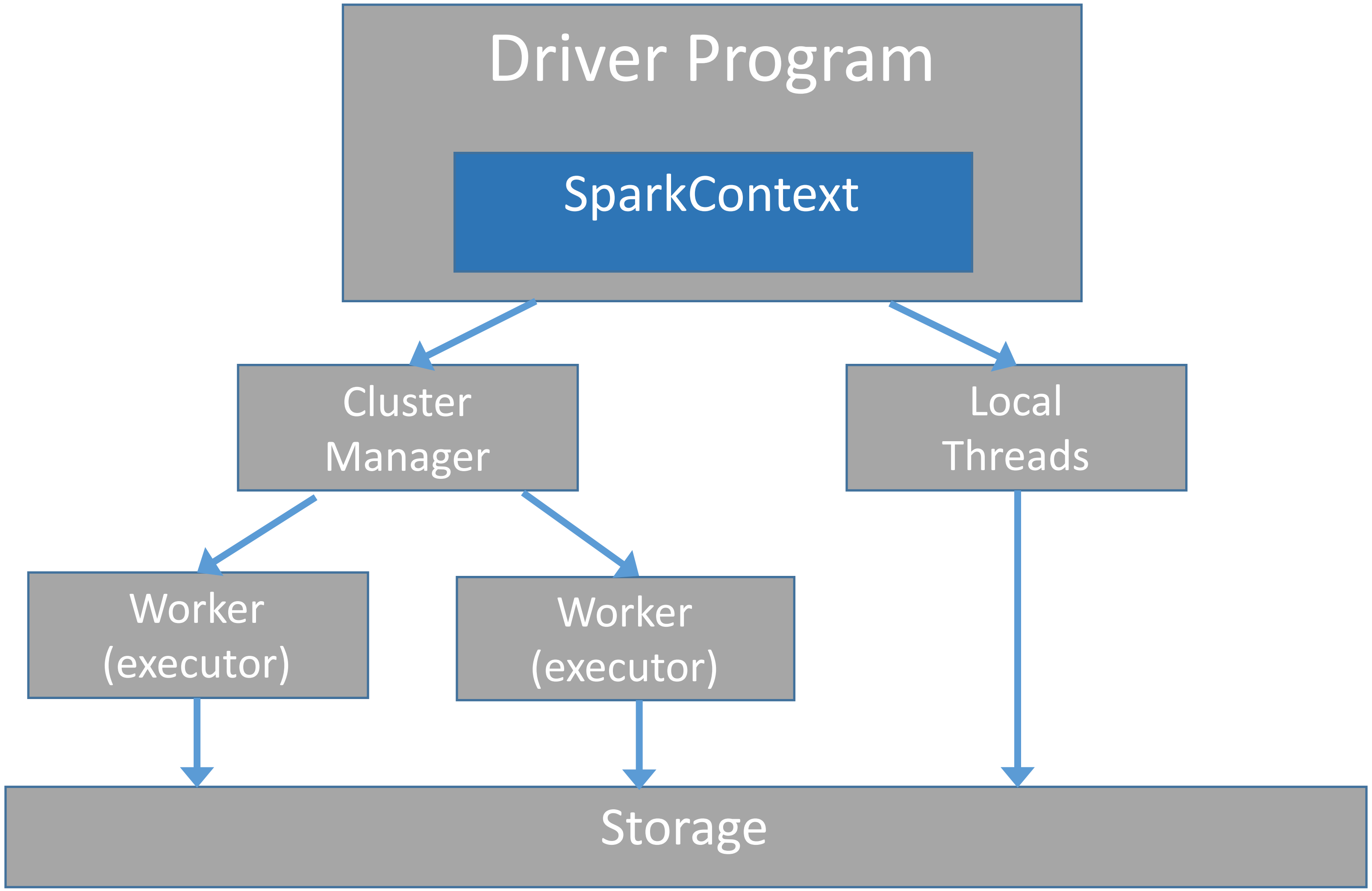
- User specified number of partitions
- More partitions = more parallelism



RDD vs. Shared Memory Model

Concern	RDDs	Distr. Shared Mem.
Reads	Fine-grained	Fine-grained
Writes	Bulk transformations	Fine-grained
Consistency	Guaranteed (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using speculative execution	Difficult
Work placement	Automatic based on data locality	Up to app (but runtime aims for transparency)

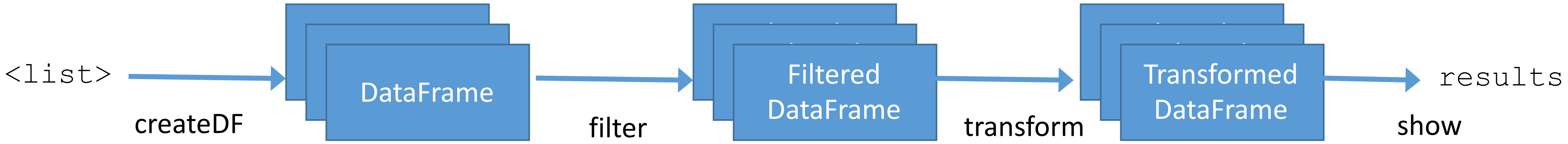
Spark Driver & Workers



2 types of operations

- **Transformations**
  - Transformations are lazy (*not computed immediately*)
  - Transformed DF is executed when action runs on it
  - Persist (cache) DFs in memory or disk

- **Actions** : collect, show, reduce, …



http://spark.apache.org/docs/1.6.2/api/python/pyspark.sql.html#module-pyspark.sql.functions

RDD: Operations

Transformations (define a new RDD)	Parallel operations (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	
…	

Transformations

Transformation	Description
`map(`*`func`*`)`	return a new distributed dataset formed by passing each element of the source through a function *func*
`filter(`*`func`*`)`	return a new dataset formed by selecting those elements of the source on which *func* returns true
`distinct([`*`numTasks`*`]))`	return a new dataset that contains the distinct elements of the source dataset
`flatMap(`*`func`*`)`	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)

Transformation: map & filter

```
> data = range(8)
  rdd = sc.parallelize(data, 8)
  rdd.map(lambda x: x * 2).collect()
```

▸ (1) Spark Jobs

```
Out[12]: [0, 2, 4, 6, 8, 10, 12, 14]
```

```
> rdd.filter(lambda x: x % 2 == 0).collect()
```

▸ (1) Spark Jobs

```
Out[13]: [0, 2, 4, 6]
```

Transformation: distinct

```
> rdd2 = sc.parallelize([3,2,1,4,3,5,1])
  rdd2.distinct().collect()
```

▸ (1) Spark Jobs

```
Out[11]: [1, 2, 3, 4, 5]
```

Transformation: map & flatMap

```
> rdd = sc.parallelize([1,2,3])
  rdd.map(lambda x: [x, x+10]).collect()
```

▸ (1) Spark Jobs

```
Out[14]: [[1, 11], [2, 12], [3, 13]]
```

```
> rdd.flatMap(lambda x: [x, x+10]).collect()
```

▸ (1) Spark Jobs

```
Out[15]: [1, 11, 2, 12, 3, 13]
```

Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark

Action	Description
`reduce(func)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(n)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(n, key=func)`	return n elements ordered in ascending order or as specified by the optional key function

Action: reduce, take, collect

```
> rdd = sc.parallelize([1,2,3])
  rdd.reduce(lambda a, b: a + b)
```

▸ (1) Spark Jobs

```
Out[16]: 6
```

```
> rdd.take(2)
```

▸ (3) Spark Jobs

```
Out[17]: [1, 2]
```

```
> rdd.collect()
```

▸ (1) Spark Jobs

```
Out[18]: [1, 2, 3]
```

Action: takeOrdered

```
> rdd = sc.parallelize(range(10))
  rdd.takeOrdered(5, lambda x: -1 * x)
```

▸ (1) Spark Jobs

```
Out[19]: [9, 8, 7, 6, 5]
```

Spark: Key-Value RDD

- Similar to Map Reduce, Spark supports Key-Value pairs
- Each element of a Pair RDD is a pair tuple

```
> rdd = sc.parallelize([(1,2), (3,4)])
  rdd.collect()

▸ (1) Spark Jobs

Out[20]: [(1, 2), (3, 4)]
```

Key-Value Transformations

Key-Value Transformation	Description
`reduceByKey(`*`func`*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ➔ V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Key-Value Transformation: reduceByKey

```
> rdd = sc.parallelize([(1,2), (1,10), (3,4), (3,10)])
  rdd.reduceByKey(lambda a,b : a + b).collect()
```

▸ (1) Spark Jobs

```
Out[24]: [(1, 12), (3, 14)]
```

Key-Value Transformation: sortByKey

```
> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
  rdd2.sortByKey().collect()
```

▸ (3) Spark Jobs

```
Out[26]: [(1, 'a'), (1, 'b'), (2, 'c')]
```

Key-Value Transformation: groupByKey

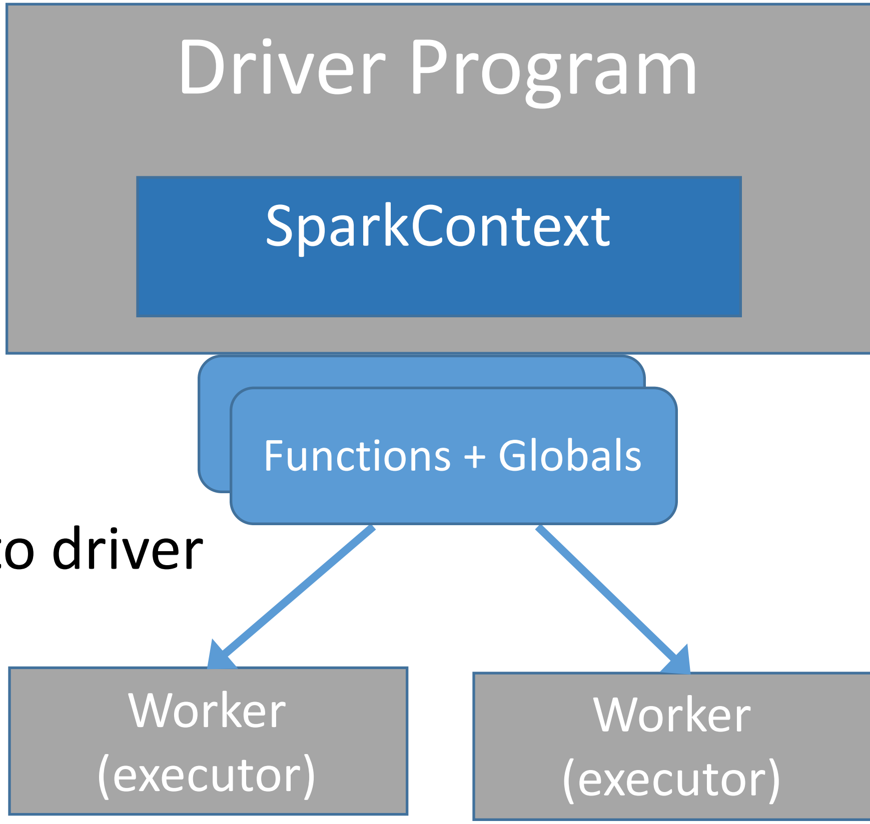
```
> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
  rdd2.groupByKey().collect()
```

▸ (1) Spark Jobs

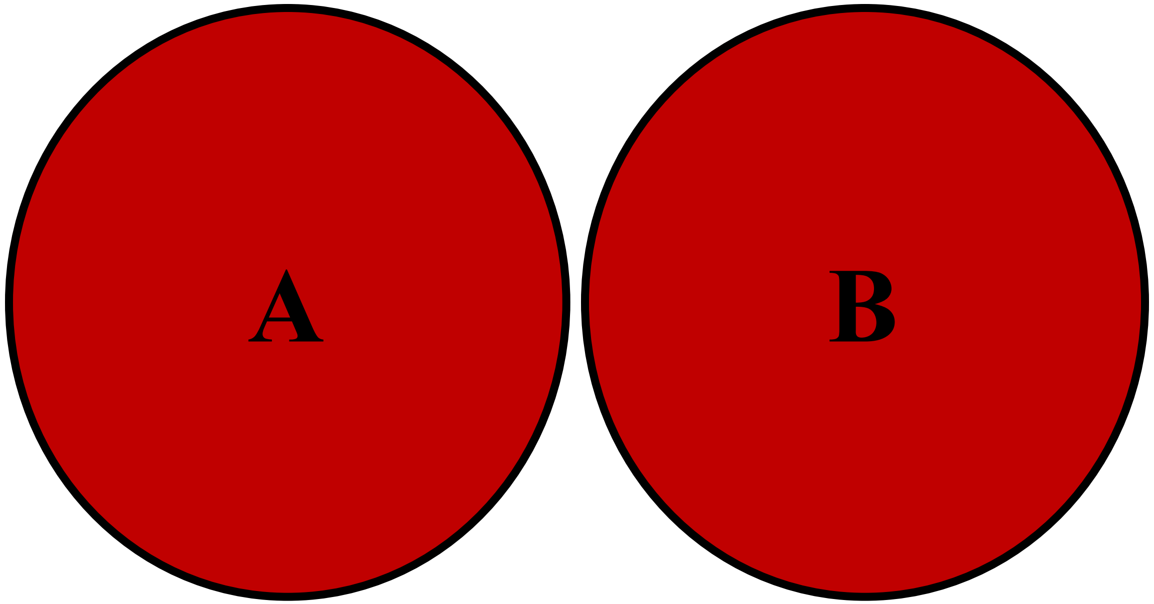
```
Out[28]:
[(1, <pyspark.resultiterable.ResultIterable at 0x7f978c887890>),
 (2, <pyspark.resultiterable.ResultIterable at 0x7f978c887e90>)]
```

Python: lambda & closure

- Spark creates "closure" with lambda function to each worker
- Closure contains:
  - Functions that run on RDDs at workers
  - Any global variables used by those workers
- One closure per worker
  - Sent for every task
  - No communication between workers
  - Changes to global variables at workers are not sent to driver



Joining Datasets in Spark!

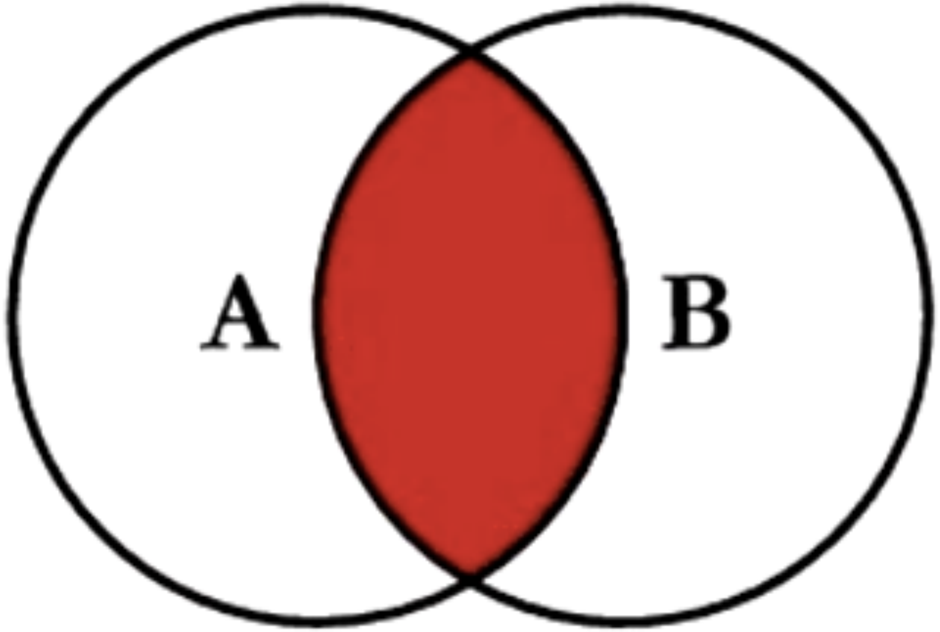



```
data = [['Alice', 1], ['Bob', 2]]
A = sqlContext.createDataFrame(data, ['name', 'age'])
data2 = [['Chris', 80], ['Bob', 85]]
B = sqlContext.createDataFrame(data2, ['name', 'height'])
```

```
+-----+---+
| name|age|
+-----+---+
|Alice|  1|
|  Bob|  2|
+-----+---+
```

```
+-----+------+
| name|height|
+-----+------+
|Chris|    80|
|  Bob|    85|
+-----+------+
```

Joining Spark Datasets : Inner Join




```
+-----+---+
| name|age|
+-----+---+
|Alice|  1|
|  Bob|  2|
+-----+---+
```

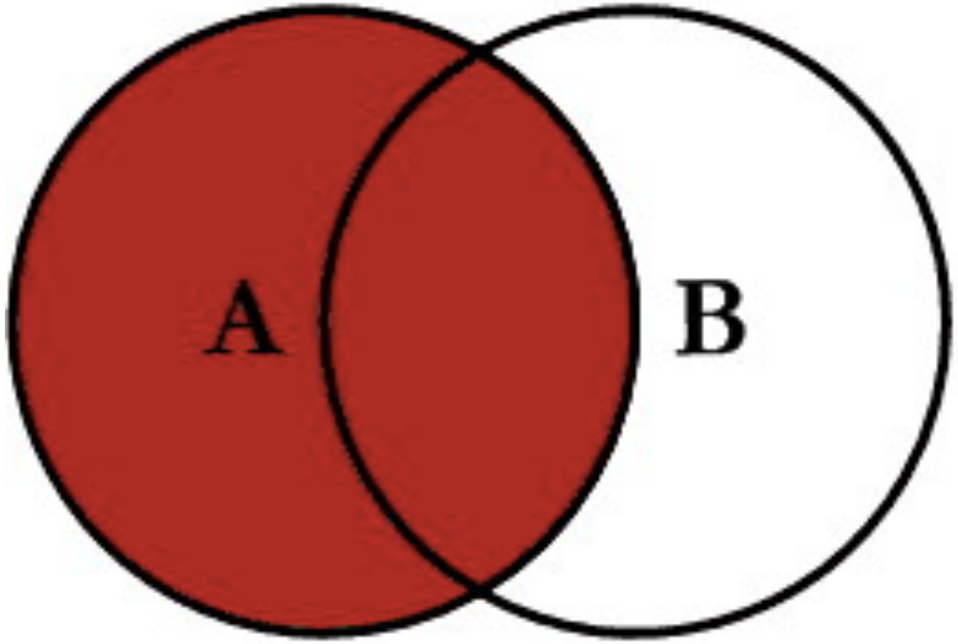
```
+-----+------+
| name|height|
+-----+------+
|Chris|    80|
|  Bob|    85|
+-----+------+
```

```
A.join(B, 'name').show()
```

▸ (2) Spark Jobs

```
+----+---+------+
|name|age|height|
+----+---+------+
| Bob|  2|    85|
+----+---+------+
```

Joining Spark Datasets : Left Outer Join




```
+-----+---+
| name|age|
+-----+---+
|Alice|  1|
|  Bob|  2|
+-----+---+
```

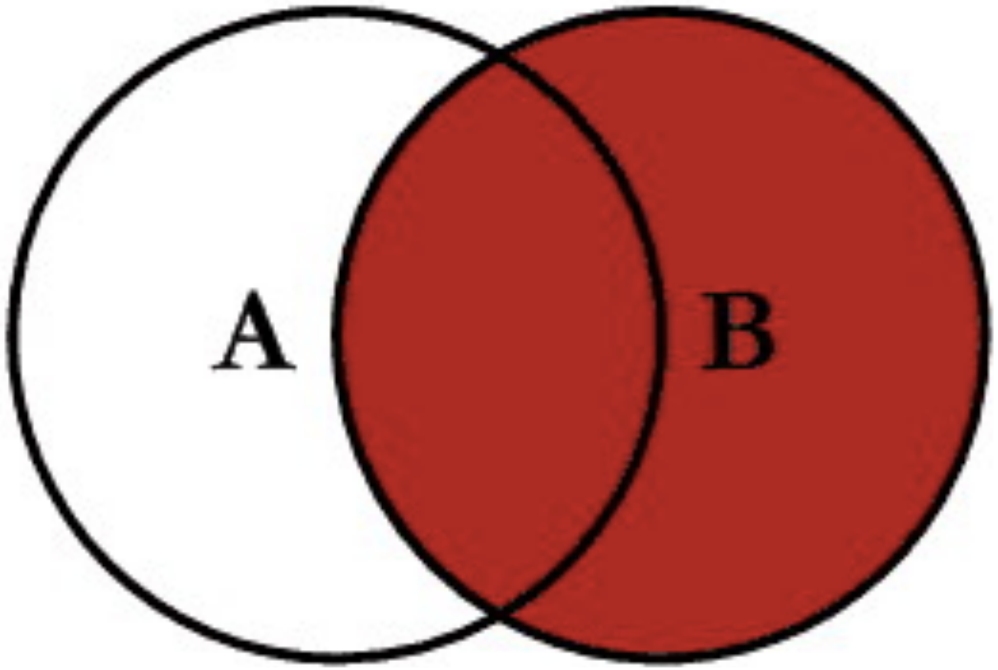
```
+-----+------+
| name|height|
+-----+------+
|Chris|    80|
|  Bob|    85|
+-----+------+
```

```
A.join(B, 'name', 'left_outer').show()
```

▸ (2) Spark Jobs

```
+-----+---+------+
| name|age|height|
+-----+---+------+
|Alice|  1|  null|
|  Bob|  2|    85|
+-----+---+------+
```

Joining Spark Datasets : Right Outer Join




```
+-----+---+
| name|age|
+-----+---+
|Alice|  1|
|  Bob|  2|
+-----+---+
```

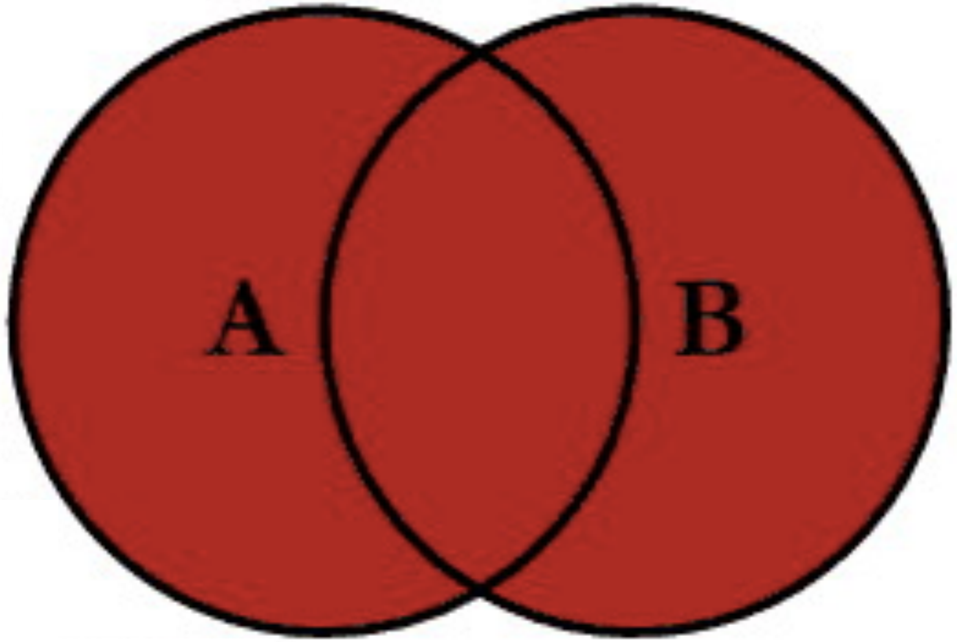
```
+-----+------+
| name|height|
+-----+------+
|Chris|    80|
|  Bob|    85|
+-----+------+
```

```
A.join(B, 'name', 'right_outer').show()
```

▸ (2) Spark Jobs

```
+-----+----+------+
| name| age|height|
+-----+----+------+
|Chris|null|    80|
|  Bob|   2|    85|
+-----+----+------+
```

Joining Spark Datasets : Full Outer Join




```
+-----+---+
| name|age|
+-----+---+
|Alice|  1|
|  Bob|  2|
+-----+---+
```

```
+-----+------+
| name|height|
+-----+------+
|Chris|    80|
|  Bob|    85|
+-----+------+
```

```
A.join(B, 'name', 'outer').show()
```

(2) Spark Jobs

```
+-----+----+------+
| name| age|height|
+-----+----+------+
|Chris|null|    80|
|Alice|   1|  null|
|  Bob|   2|    85|
+-----+----+------+
```

Exercise: Shakespeare Complete Work

- ShakespeareStart.dbc
  - Your exercise project to examine Shakespeare’s complete work
- ShakespeareSolution.dbc
  - Solution file containing a possible solution

Agenda

- Data Cleaning, Profiling, Performance
- String Processing
- Trends & Outliers
- Big Data Processing with Spark
- **Final Project**

Your final project : guidelines

- Goal: **apply** what you have learned in this class to a realistic data science challenge + exercise your creativity + have fun!
- This is meant to be a significant **individual effort** to learn by practicing what you are learning to a real-world data science problem.
- The **writeup** of your final project is in the form of a Jupyter notebook and associated data – to be uploaded to the final project assignment in Camino.
- You are to submit your final notebook by September 3 @ 11:59pm.

Your final project : topic selection

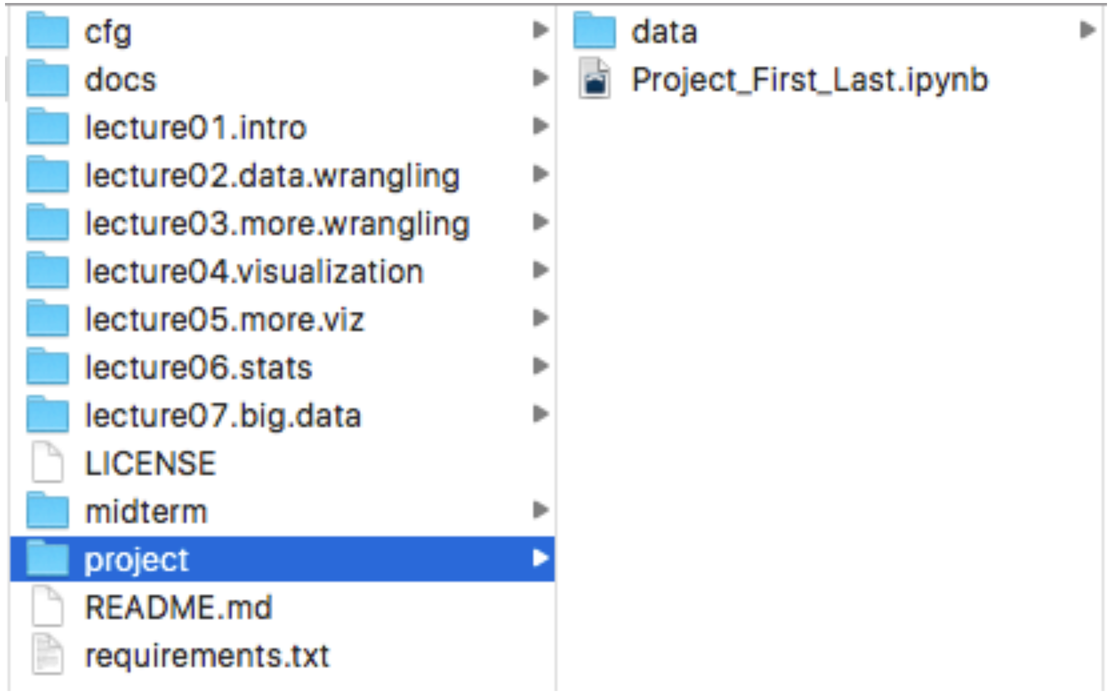
- Goal: **apply** what you have learned in this class to a realistic data science challenge + exercise your creativity + have fun!
- You can choose any ”significant” data set via downloadable sites, APIs, or use any of the datasets from the class.
- You need to propose an interesting data insight investigation that you would like to explore, analyze the data, visualize the data, and finally write up your conclusion on what insights you have reached.
- Grading of your final project will be based on the following rubric.

Your final project : grading rubric

Area	Details	Grading %
Topic Selection	Did you create a reasonably interesting data insight hypothesis for your investigation?	10%
Packaging	Did you create a Jupyter project packaging that looks professional and understandable?	10%
Analysis Competence	Does your notebook show competence in using the data science tools we learned in class?	40%
Insight	Does your project show useful or interesting insights from the data analysis you have done?	40%

Submitting your notebook to github

- In the “datascience” folder, there is now a “project” subfolder:



- Your notebook needs to be submitted to this “project” folder

Submitting your notebook to github

- Your notebook should be named: “Project_First_Last.ipynb” where:
  - First : your first name
  - Last : your last name

- Any dataset that you are using should be submitted to:
  - datascience/project/data

Submitting your notebook to github

- When you are in the folder “datascience” in the console, your first check-in should look like:

```
git add project/Project_First_Last.ipynb
git add project/data/Project_First_Last_data.txt
git commit –m “Checking in my awesome project”
git push
```

Submitting your notebook to github

- After your first check, and you modified your notebook or data,

```
git add project/Project_First_Last.ipynb
git add project/data/Project_First_Last_data.txt
git commit –m "Checking in my awesome project"
git push
```

See you on September 10!