CS 398 ACC
Spark

Prof. Robert J. Brunner

Ben Congdon
Tyler Kim

MP2

How’s it going?

Final Autograder run:

- Tonight ~9pm
- Tomorrow ~3pm

- Due tomorrow at 11:59 pm.
- Latest Commit to the repo at the time will be graded.
- Last Office Hours today after the lecture until 7pm.

Fun MP2 Facts:

Shortest Succeeded Job: 32 seconds

Longest Succeeded Job: 2.5 Hours

Longest Failed Job: 35 minutes

Longest Running Job: 4 days (still running... please stop)

Applications Submitted: ~350

Announcements

- Quizzes will now be due on Sundays.
  - Grading method will remain last attempt

- Cluster address will be changing for MP3
  - SSH keys will stay the same
  - Old cluster will be terminated after MP2 due date
  - Copy any data off that you care about

Outline

- Spark Overview
- Spark Core
- Related Frameworks
- Spark vs. Hadoop
- Spark Use Cases
- Spark Programming

Outline

- **Spark Overview**
- Spark Core
- Related Frameworks
- Spark vs. Hadoop
- Spark Use Cases
- Spark Programming

Motivations / History

- MapReduce began to show it’s downsides:
  - It isn’t fast enough
  - It’s inefficient on iterative workloads
  - It relies too heavily on on-disk operations

- Research group at UC Berkeley develop Spark
  - Started in 2009
  - Initial versions outperformed MapReduce by 10-20x

Apache Spark

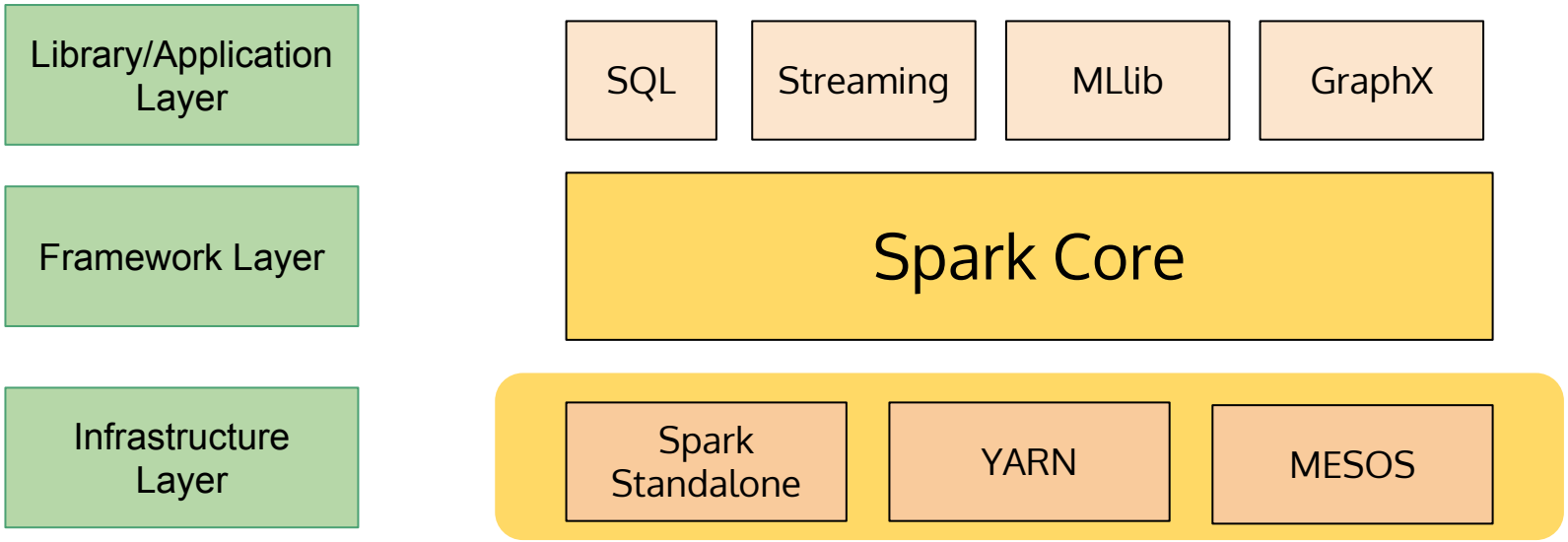
Open Source, Distributed general-purpose computing framework.

- Managed by Apache Foundation
- Written in Scala
- Robust high-level APIs for different languages
- Allows iterative computations
    - Graph algorithms, ML, and more

Computational Framework

- You write framework code, as distinct from “normal code”
- Code that tells the framework ***what*** to do, not ***how*** to do it
  - The framework can handle optimization / data transfer internally
- Other computational frameworks:
  - Tensorflow, Caffe, etc.

Much more flexibility than Hadoop



- Aims for easy and interactive analytics
- Distributed Platform for complex multi-stage applications, (e.g. real-time ML)

Resilient Distributed Dataset (RDD)

Data Abstraction in Spark.

- **Resilient**
  - Fault-tolerant using a data lineage graph
  - RDDs know where they came from, and how they were computed
- **Distributed**
  - Data lives on multiple nodes
  - RDDs know where they're stored, so computation can be done "close" to the data
- **Dataset**
  - A collection of partitioned data

Resilient Distributed Dataset (RDD)

- What does an RDD look like?
  - A large set of arbitrary data (tuples, objects, values, etc)

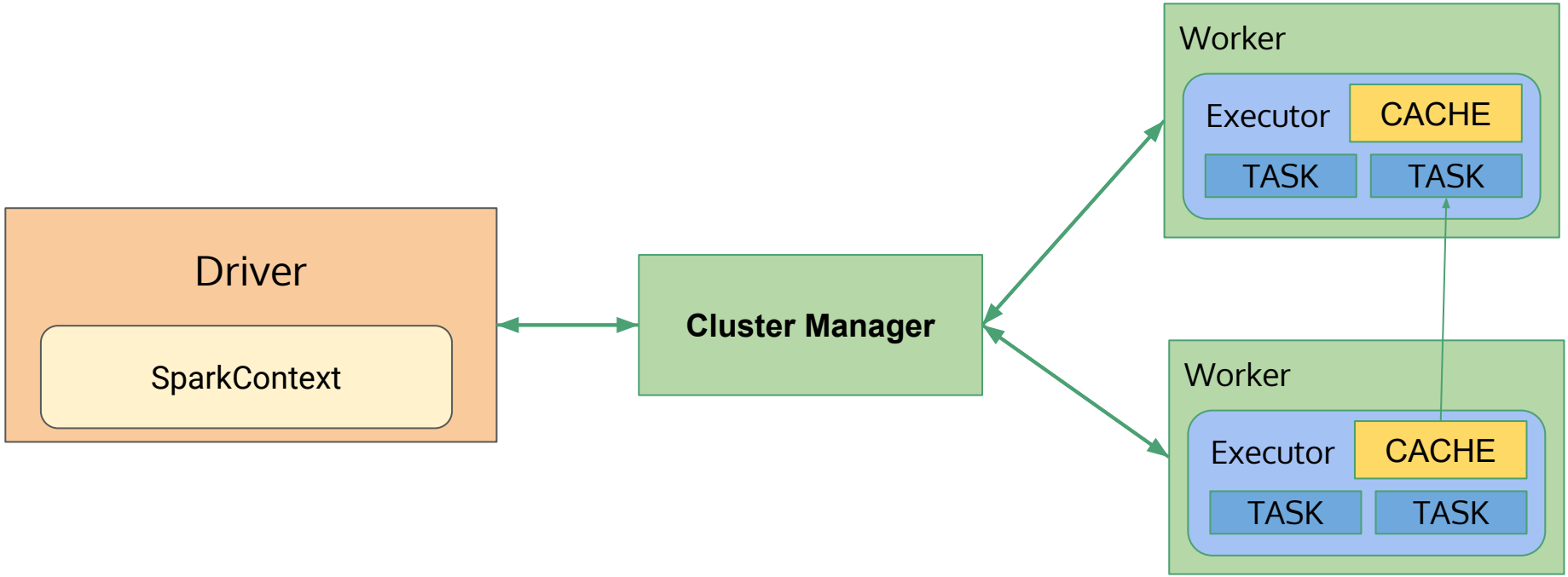
- Features of RDDs:
  - Stored in-memory, **Cacheable**
    - Stored on executors
  - **Immutable**
    - Once created, cannot be edited
    - Must be *transformed* into a new descendent RDD
  - **Parallel** via partitioning
    - Similar to how Hadoop partitions map inputs

RDD Operations

- **Transformations**
  - The process of taking an input RDD, doing some computation, and producing a new RDD
  - Done lazily (only ever executed if an “action” depends on the output)
  - i.e. Higher order function like Map, ReduceByKey, FlatMap

- **Actions**
  - Triggers computation by asking for some type of output
  - i.e. Output to text file, Count of RDD items, Min, Max

Architecture



Architecture

- **Driver**
  - One per job
  - Handles DAG scheduling, schedules tasks on executors
  - Tracks data as it flows through the job

- **Executor**
  - Possibly many per job
  - Possibly many per worker node
  - Stores RDD data in memory
  - Performs tasks (operations on RDDs), and transfers data as needed

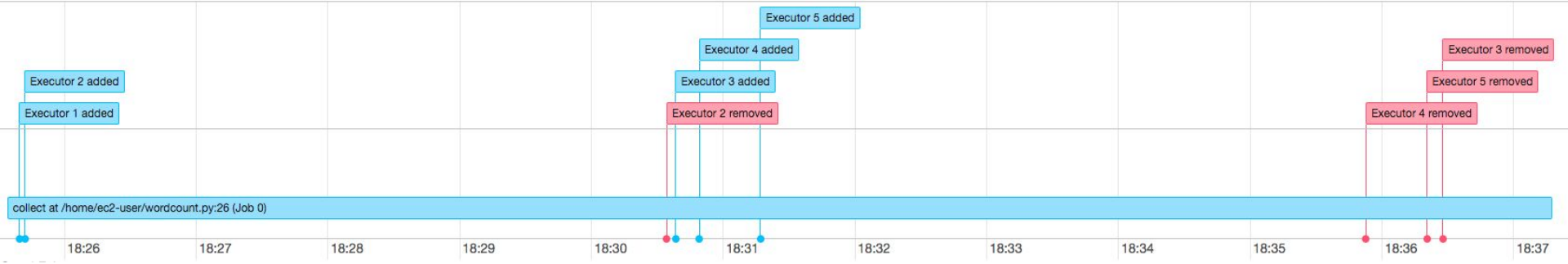
Executor Allocation

- **Traditional Static Allocation**
  - Create all Executors at beginning of job
  - Executors are online until end of job
  - Only option in early versions of Spark

- **Dynamic Executor Allocation**
  - Jobs can scale up/down number of executors as needed
  - More efficient for clusters running multiple apps concurrently

Executor Allocation

- **Dynamic Executor Allocation**



Spark Application Architecture

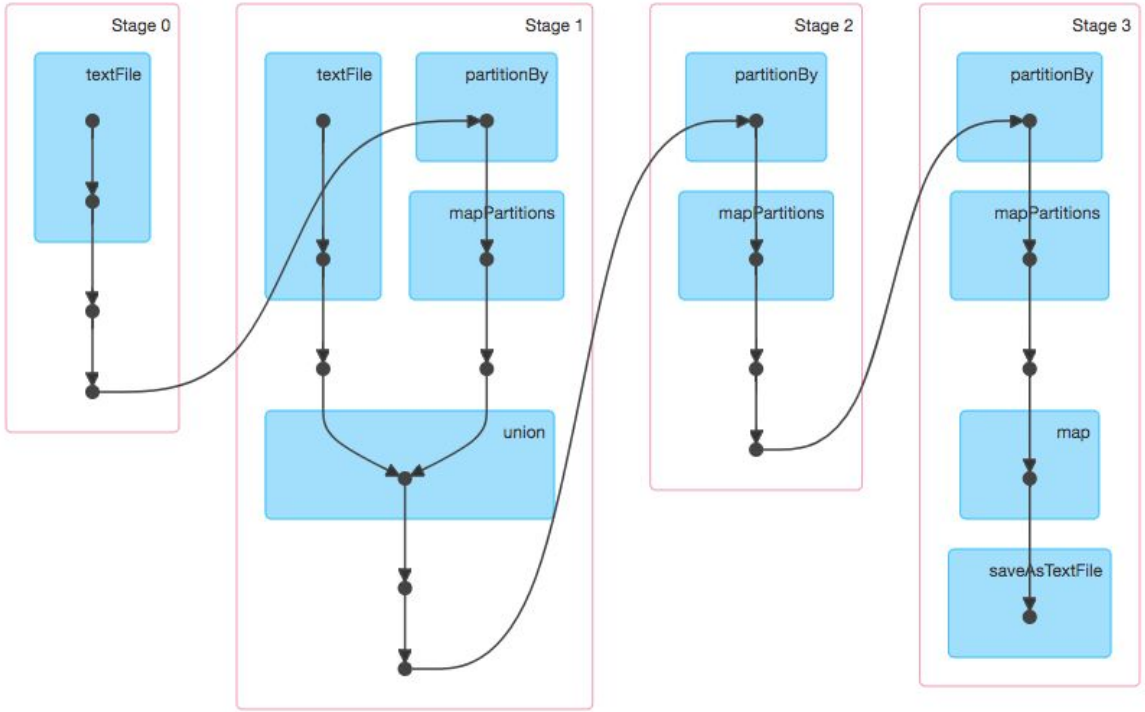
- **Job**
  - An application can have multiple jobs
    - For our purposes, we'll usually have just one job per application
  - Created by a RDD action (e.g. collect)
- **Stage**
  - A group of potentially many operations
  - Many executors work on tasks in a single stage
  - A stage is made up of many tasks
- **Task**
  - The "simplest unit of work" in Spark
  - One operation on a partition of an RDD

DAGScheduler

What does it do?

- Computes an execution DAG
- Determines the preferred locations (Executors) to run each task
- Handles failure due to lost shuffle output files
- Performs operation optimizing
    - Groups multiple operations (e.g. maps and filters) into the same stage

DAGScheduler



Storage for Spark

Again, much more flexibility.

- HDFS
- S3
- Cassandra
- HBase
- Etc.

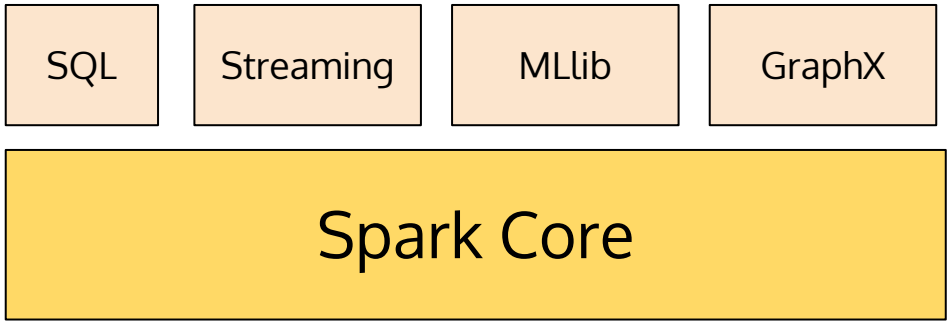
While Hadoop (mostly) limited to HDFS, Spark can bring in data from anywhere

Outline

- Spark Overview
- Spark Core
- **Related Framework**
- Spark vs. Hadoop
- Spark Use Cases
- Spark Programming

Related Frameworks

Spark Core is tightly integrated with several key libraries



Related Frameworks

- **Spark Streaming**
  - Stream live data into Spark cluster
  - Send it out to databases or HDFS
- **Spark SQL**
  - Integrates relational database programming (SQL) with Spark
- **Spark MLlib**
  - Large-Scale Machine Learning
- **GraphX**
  - Graph and graph-parallel computations

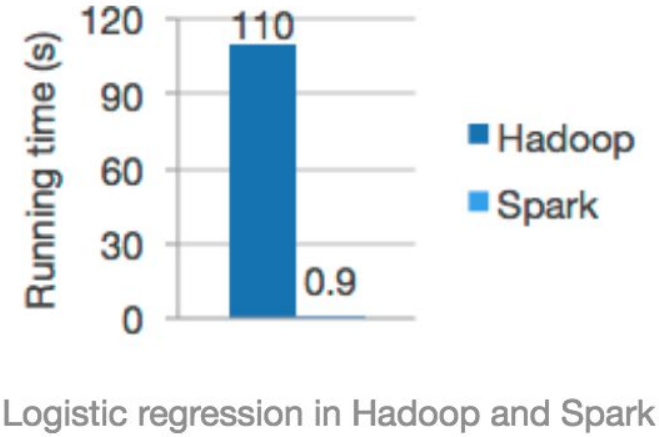
Interactions between the frameworks allow multi-stage data applications.

Outline

- Spark Overview
- Spark Core
- Related Framework
- **Spark vs. Hadoop**
- Spark Use Cases
- Spark Programming

Hadoop vs. Spark

Spark can be more than 100x faster, especially when performing computationally intensive tasks.



Logistic regression in Hadoop and Spark

Hadoop vs. Spark

Step 1. Build something

Step 2. Prove its 100x faster than Hadoop

Step 3. ???

Step 4. Profit!

Hadoop + Spark

Spark on HDFS

What can they bring to the table for each other?

- **Hadoop**
  - Huge Datasets under control by commodity hardware.
    - Low cost operations
- **Spark**
  - Real-time, in-memory processing for those data sets.
    - High-speed, advanced analytics to a multiple stage operations.

Spark cannot yet completely replace Hadoop.

Outline

- Spark Overview
- Spark Core
- Related Framework
- Spark vs. Hadoop
- **Spark Use Cases**
- Deploying Spark

When to use Spark

Building a data pipeline

Interactive analysis and **multi-stage** data application.

- Allows real-time interaction / experimentation with data

Streaming Data

- Spark Streaming

Machine Learning

- Spark MLlib

Spark in Industry

e-Commerce Industry

- eBay
  - Provide targeted offers, enhance customer experience, etc.
  - eBay runs Spark on top of YARN.
    - 2000 nodes, 20,000 cores, and 100TB of RAM
- Alibaba
  - Feature extraction on image data, Aggregate data on the platform
  - Millions of Merchant-User Interaction is represented in graphs.

Spark in Industry

Finance Industry

- Real-Time Fraud Detection (stolen credit card swipe or stolen card number)
  - Check with previous fraud footprint
  - Triggers call center, etc.
  - Validate incoming transactions
- Risk-based assessment
  - Collecting and archiving logs
  - Spark can easily be combined with external data source and pipelines.

Outline

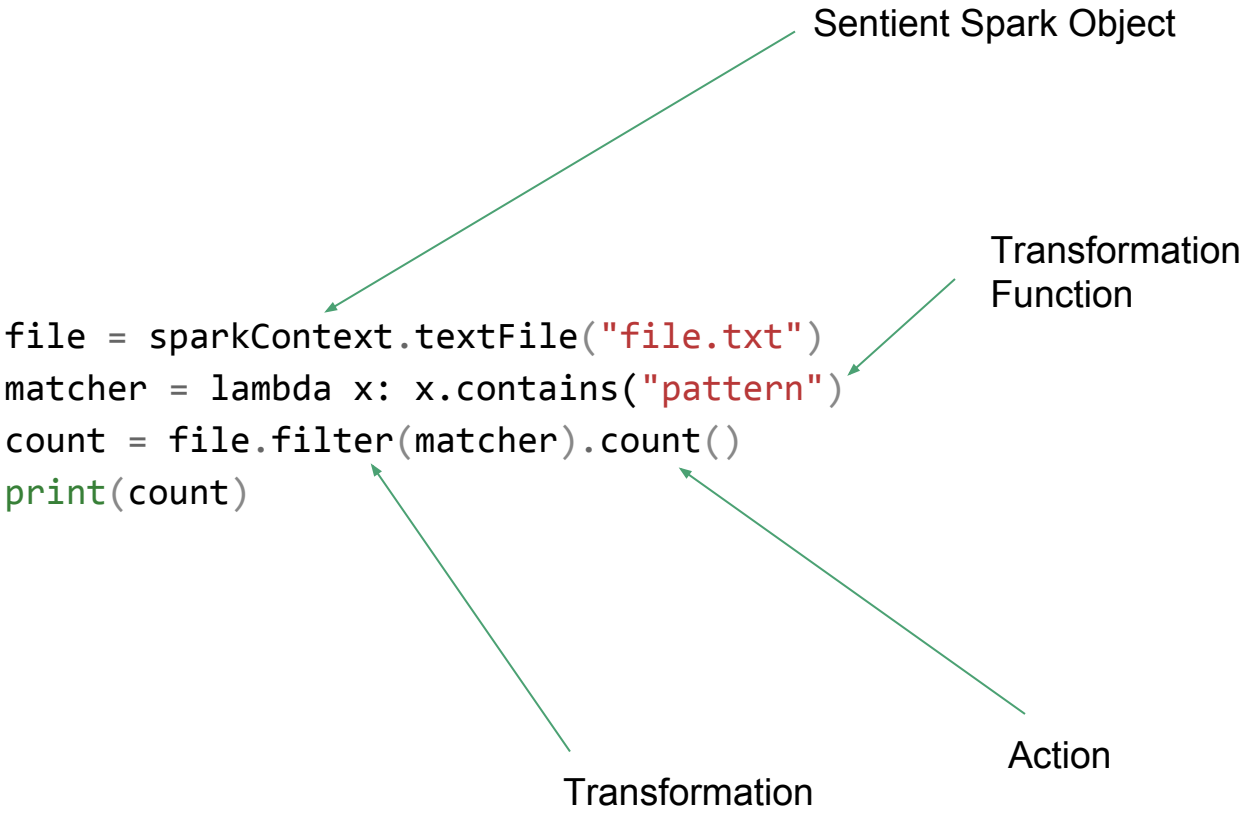
- Spark Overview
- Spark Core
- Related Framework
- Spark vs. Hadoop
- Spark Use Cases
- **Spark Programming**

What Does Spark Code Look Like?

```
count = len([line for line in \
    open('file.txt') \
    if 'pattern' in line])
print(count)
```

```
file = sparkContext.textFile("file.txt")
matcher = lambda x: x.contains("pattern")
count = file.filter(matcher).count()
print(count)
```

Example


Sentient Spark Object
Transformation
Function
file = sparkContext.textFile("file.txt")
matcher = lambda x: x.contains("pattern")
count = file.filter(matcher).count()
print(count)
Transformation
Action

Word Count Example

```
# Load data
textData = sparkContext.textFile("input.txt")

# Split into words
WORD_RE = re.compile(r"[\w']+")
words = textData.flatMap(lambda line: WORD_RE.findall(line))

# Get count by word
counts = words.map(lambda w: (w, 1)).countByKey()

print(counts.collect())
```

Join Example

```
favColors = sc.parallelize([('bob', 'red'), ('alice', 'blue')])
favNumbers = sc.parallelize([('bob', 1), ('alice', 2)])

joined = favColors.join(favNumbers)
joined.collect()

# [('bob', ('red', 1)), ('alice', ('blue', 2))]
```

ReduceByKey Example

```
nums = sc.parallelize([('a', 1), ('b', 2), ('a', 3), ('b', 4)])
reduced = nums.reduceByKey(lambda v1, v2: v1 + v2)
reduced.collect()

# [('b', 6), ('a', 4)]
```

Wednesday

- (Optional)
- Spark Demo
- Office Hours

MP3 - Spark Core

- Will be released Tonight.
- One of the more difficult MPs
- Will involve a lot of documentation reading. (This is expected)