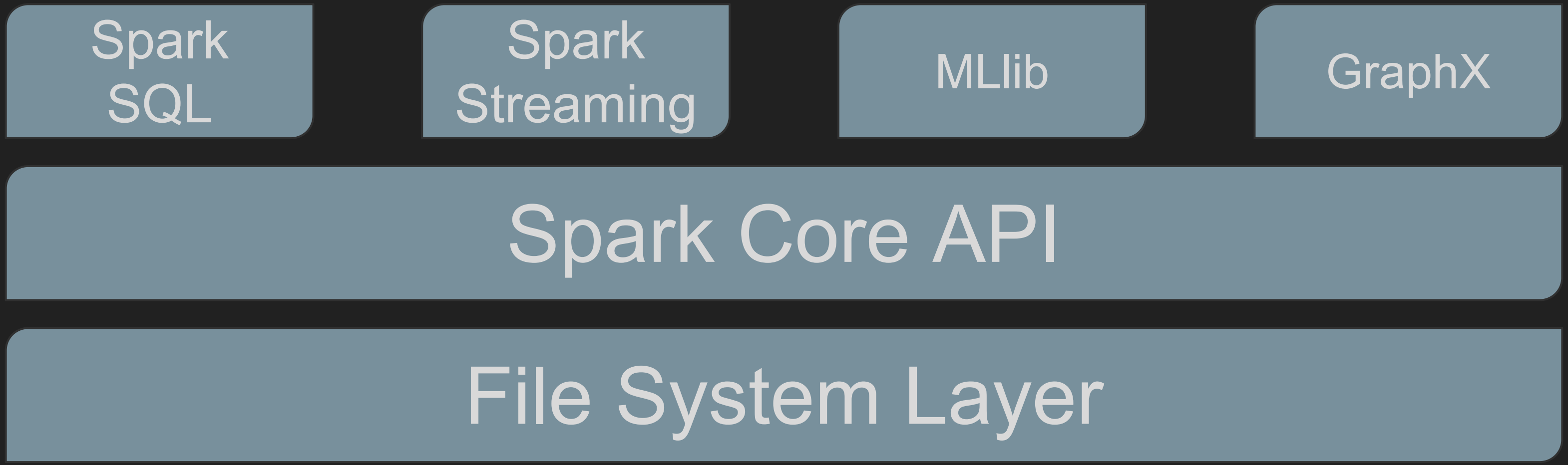
Spark

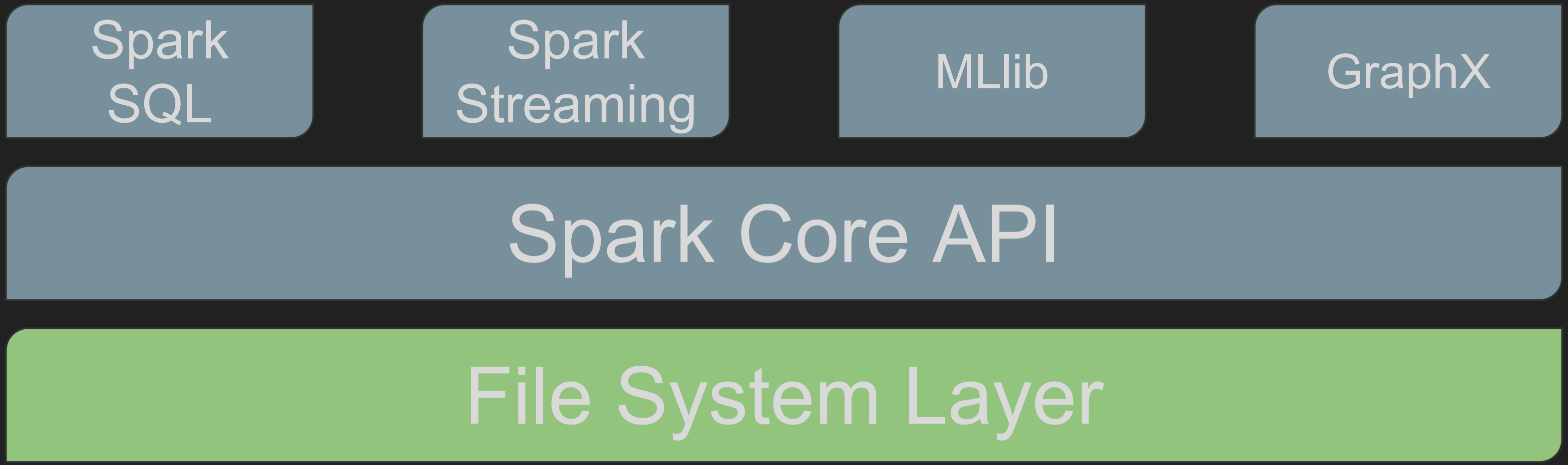
What is spark?

Spark is an extension of the MapReduce framework, designed to work in-memory and have a simpler API



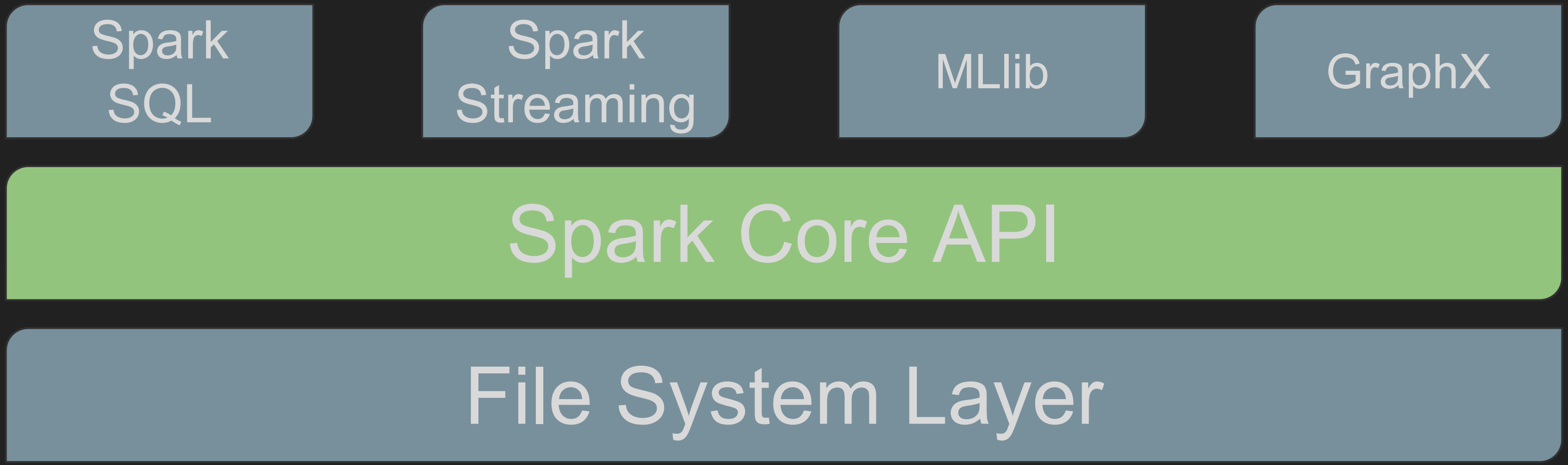
What is spark?

Spark is an extension of the MapReduce framework, designed to work in-memory and have a simpler API



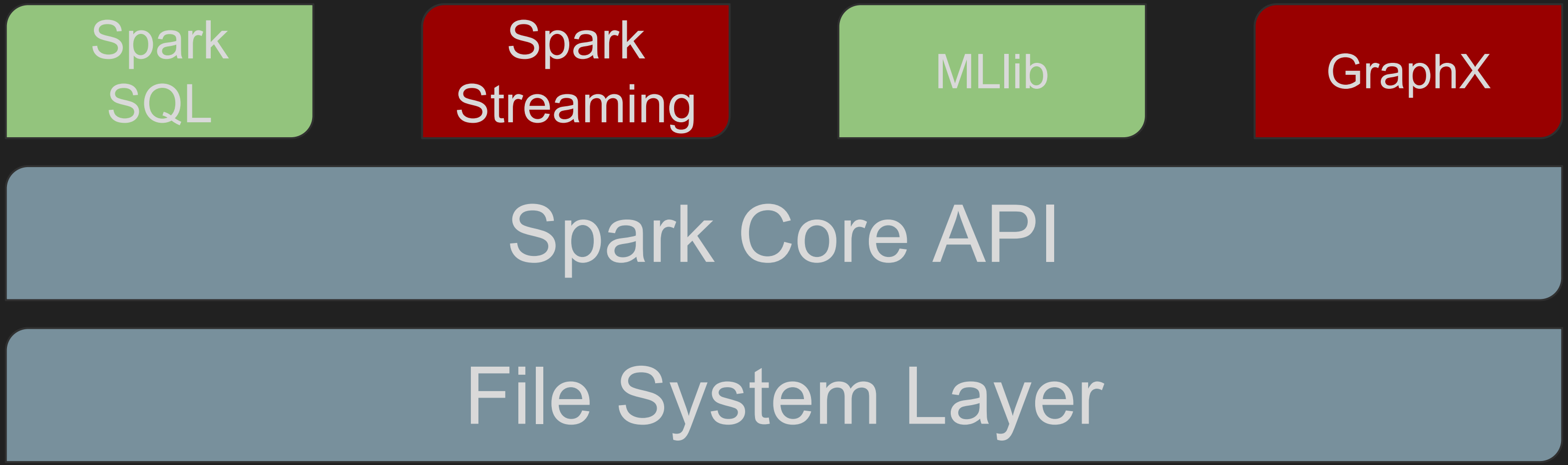
What is spark?

Spark is an extension of the MapReduce framework, designed to work in-memory and have a simpler API



What is spark?

Spark is an extension of the MapReduce framework, designed to work in-memory and have a simpler API



How does Spark compare to MapReduce

Category	Spark	MapReduce	Winner
Method	Works in memory	Writes to disk after each operation	Spark (??)
Speed	Up to 100X faster in memory/10X faster on disk	Inverse of that	Spark
Reliability	Been around for ~7 years, has 1000's of active contributors	Been around for ~15 years, also lots of contributors	MapReduce
Compatibility	Can be built on any filesystem	Must be built on top of HDFS	Spark (??)
Overall Winner:			Depends on what you are doing

Spark basics

- At the heart of spark is the RDD (Resilient Distributed Dataset)
  - Dataset
    - It’s a container for data
  - Distributed
    - Spreads data across multiple workers in the cluster
    - No built in redundancy, relies on the underlying file system
    - Data is split into partitions, generally 2-4 partitions per machine
  - Resilient
    - Remembers where it came from
    - In event of a failure, will recompute data based on this history

RDD’s

Three keys of RDD’s

- ➔ Immutable
- ➔ Lazily evaluated
- ➔ Cacheable

RDD’s

Three keys of RDD’s

- ➔ Immutable
  - ◆ Cannot change an RDD, can only **transform** into a new RDD
- ➔ Lazily evaluated
- ➔ Cacheable

RDD’s

Three keys of RDD’s

- ➔ Immutable
  - ◆ Cannot change an RDD, can only <u>**transform**</u> into a new RDD
- ➔ Lazily evaluated
  - ◆ Transformations are not performed when called
  - ◆ The chain of transformations is evaluated when a result is asked for
    - Known as an <u>**action**</u> in spark lingo
- ➔ Cacheable

RDD’s

Three keys of RDD’s

- ➔ Immutable
  - ◆ Cannot change an RDD, can only <u>**transform**</u> into a new RDD
- ➔ Lazily evaluated
  - ◆ Transformations are not performed when called
  - ◆ The chain of transformations is evaluated when a result is asked for
    - ● Known as an <u>**action**</u> in spark lingo
- ➔ Cacheable
  - ◆ Can prevent recomputation of entire chain by caching intermediate results

RUN SOME CODE HERE

Congratulations, you just learned functional programming

- ➔ The immutable object being transformed via a series of lazily evaluated functions resulting in a separate output is a large component of functional programming
- ➔ The combination of immutability, static typing, and lazy evaluation allows for internal optimizations
- ➔ MapReduce roughly follows this framework as well, but is limited by its structure

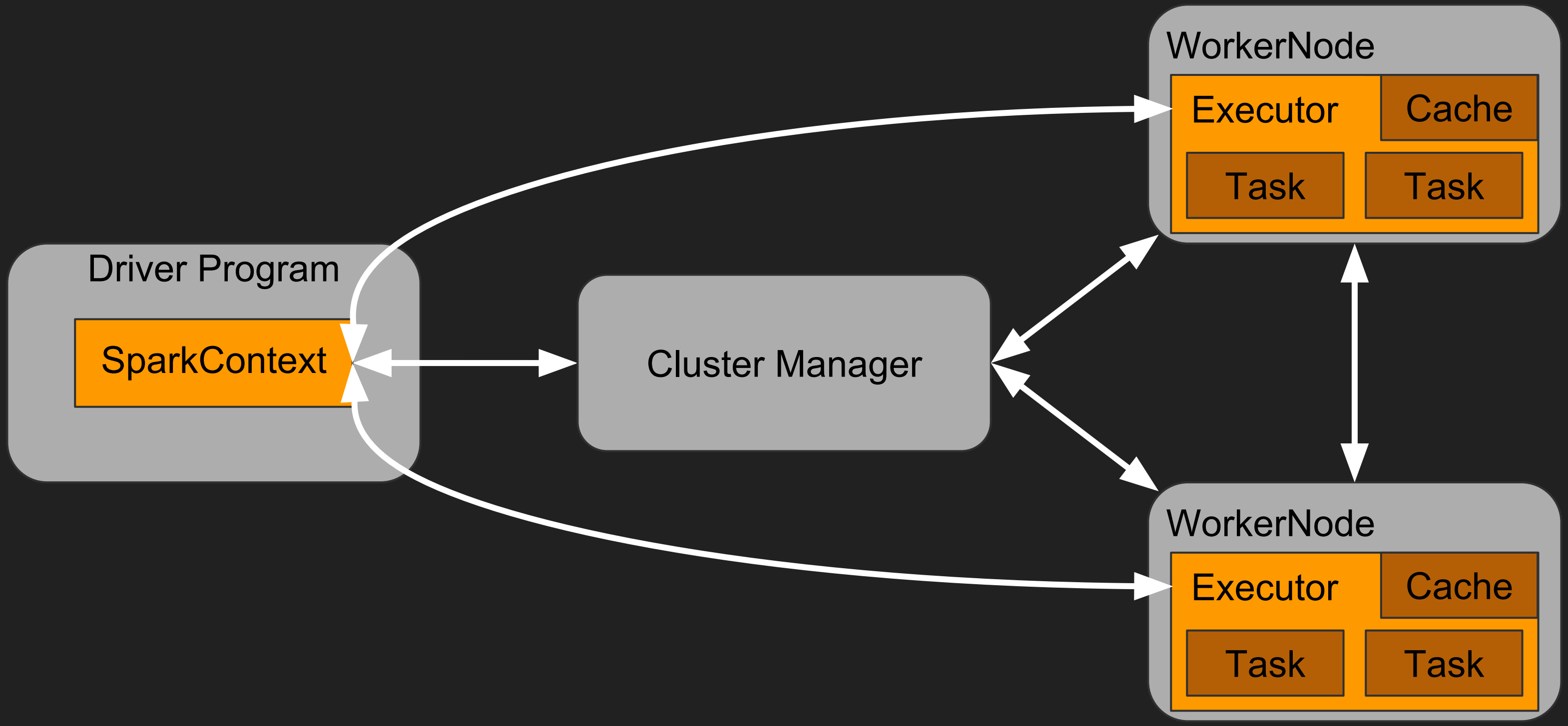
Back to RDD’s

- ➔ Why doesn’t Spark cache everything for us?
- ➔ What are the drawbacks of caching?
- ➔ Do we have an alternative to cache?

Spark Architecture

- ➔ Spark requires a lot of infrastructure to work
- ➔ Spark was written in Scala, and runs on the JVM (Java Virtual Machine)
  - ◆ Python API is very thorough, but there are drawbacks to using it
    - Because Python is duck-typed Spark has to work out what type everything is as it is passed on to the JVM
    - This slows down execution, PySpark is slower than Spark code written in Java
    - We will see a workaround to this problem tomorrow
- ➔ Designed around working on a cluster
  - ◆ Even when running spark locally, you are using a fake cluster under the hood

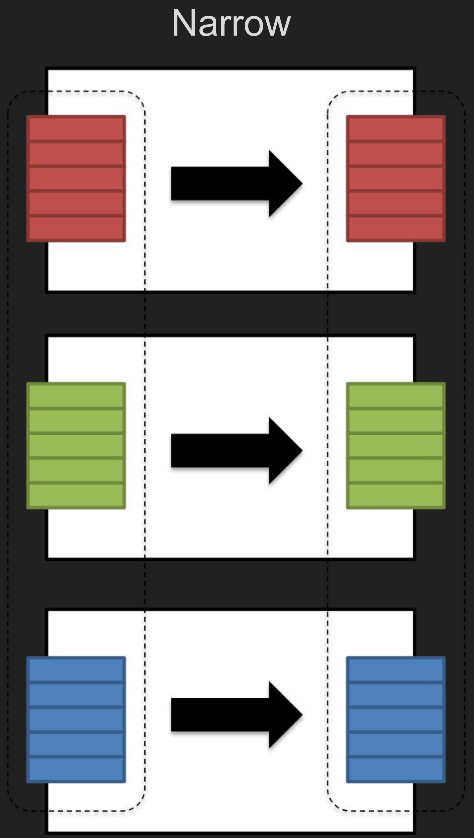
Spark Architecture

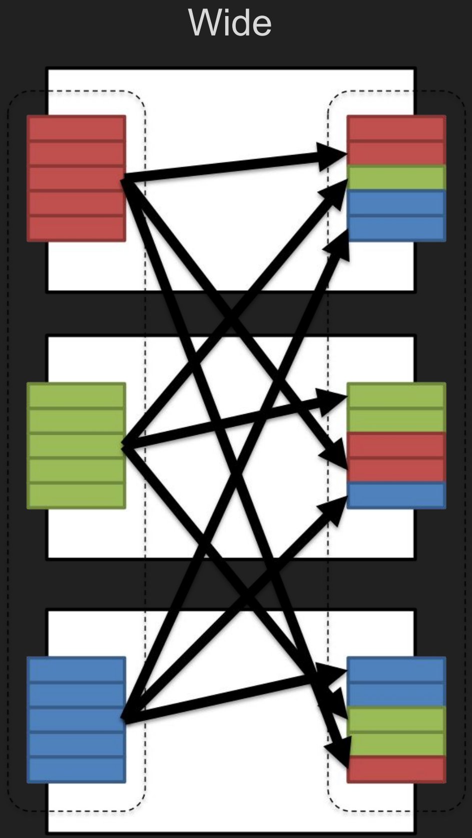


Spark Architecture

➔ Driver program passes code to the Cluster Manager (also called MasterNode)
➔ Cluster Manager distributes code out the WorkerNodes
➔ WorkerNodes perform calculations (tasks) and pass results back to MasterNode and/or driver
➔ Workers can communicate between each other, and back to the

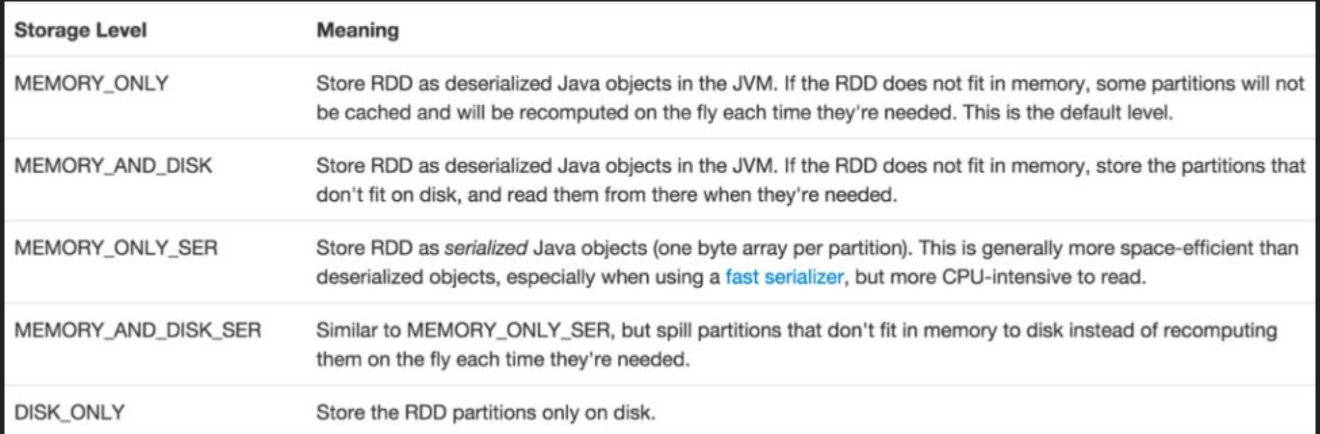
Two types of transformation





Some practical notes

- Useful Transformations
  - map(), flatMap(), sortBy(), join(), filter(), reduceByKey(), groupByKey()
- Useful Actions
  - first(), take(), top(), count(), reduce(), countByKey(), collect(), saveAsTextFile()
- persist() is an alternative to cache() that can use the disk
- Can unpersist() to remove cached rdd from memory/disk



Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.

More practical notes

- Remember, Spark is designed to be run on clusters
  - We will do more with this tomorrow
- Can track the progress of spark jobs, investigate the DAG, and other useful things in the SparkUI
  - Will work with this explicitly in the individual assignment
- Looking at types of transformations can be useful for large jobs
  - Want to cache after wide transformations, narrow ones can generally be recalculated quickly
- In general Spark is not designed for iterative testing and updating in way we have been doing so far
  - Develop on small fraction of data
  - Make sure the DAG is working and doing what you want it to do
  - Run job on huge dataset only once

STOP LECTURING DUMMY

Just a note to myself that this is the endpoint for the morning

Partitioning

- Remember Monday
  - Parallelization speeds things up, but has overhead
- Can alter how Spark parallelizes by changing number of partitions
  - More partitions means more parallel tasks
    - Means each one goes faster
    - But there is more overhead
  - Need to balance between overhead and speed
- Spark recommends 2-4 partitions per core in cluster
-

Partitioning

- ➔ Can set number of partitions when creating RDD
  - ◆ sc.parallelize(input_data, 16)
  - ◆ sc.textFile(‘path/to/data’, 16)
- ➔ Update after the fact
  - ◆ rdd.repartition(16)
- ➔ Can also partition by key
  - ◆ Remember all those (key, value) tuples from MapReduce
  - ◆ Same idea here
  - ◆ Partitioning by key will put tuples with the same key on the same partition
  - ◆ Keeping keys together can potentially promote more narrow transformations
    - ● But again, there is overhead in the initial partitioning process
- ➔ In practice only partition by key when you will perform multiple key based transformations
  - ◆ Groupbykey, reducebykey, and so forth

Joins

- ➔ (key, value) RDDs can be joined
  - ◆ Can use SQL style joins (inner, left outer, right outer, full outer) to combine based on common keys
  - ◆ rdd1 = sc.parallelize([(100156, 1),(100156, 2), (100157, 3)])

    rdd2 = sc.parallelize([(1, “REI”), (2, “Sports!”), (3, “Target”)])

    joined_rdd = rdd1.map(lambda (key, value) = (value, key).join(rdd2)

    joined_rdd.collect()
    > [(1, (100156, “REI), (2, (100156, “Sports!”)), (3, (100157, “Target”))]

Accumulators

- ➔ Remember counters from MapReduce (I don’t)
  - ◆ Spark has them too, but they are called Accumulators
- ➔ Allows for aggregating data back to the MasterNode from the Workers
  - ◆ Useful for counting events you want to track, for example, errors

```
file = sc.textFile(inputFile)
blank_lines = sc.Accumulator(0) # initialize to zero, could start with other value

def process_line(line):
    global blank_lines
    if line == “”:
        blank_lines += 1
    return line.split()

data = file.flatMap(process_line)
```

Broadcast variables

- A different type of shared variable
- In general spark sends any variable referenced in a function to the workers
  - But we might reference the same variable in multiple functions
  - Spark would send this multiple times
  - Since communication is the slowest part, this is bad
- If we know this is going to happen, can declare a broadcast variable
  - This variable is only sent once
  - And is sent using the workers as a p2p network
  - Reduces number of times sent, and the time to send the first time
- Broadcast variables are read only
  - cannot be updated on the workers
- Use sc.broadcast(object) to create

MLlib

- Spark has support for many different machine learning libraries
- A lot of the ones you would expect
  - Linear/Logistic Regression, SVM, NaiveBayes, RandomForest, GradientBoostedTrees
  - NMF, SVD, PCA
  - Kmeans, LDA
- More are being added fairly regularly
  - I almost took a job where IBM was going to hire me to implement ML algorithms for Spark
  - Still not sure why that job exists, but it does
- Spark demands you pass data to its ML functions in a certain way

Labeled points

- ➔ LabeledPoint(target, feature)
  - ◆ Think of X and y
  - ◆ Feature is a row of X, y is associated label
- ➔ MLlib has lots of documentation