Advanced Spark Programming

Advanced Spark Programming
Game Plan

- Partitioning
- Joins
- Advanced features

    Accumulators
    Broadcast variables

- MLlib

    Sparks library of machine learning functions

Advanced Spark Programming
Goals

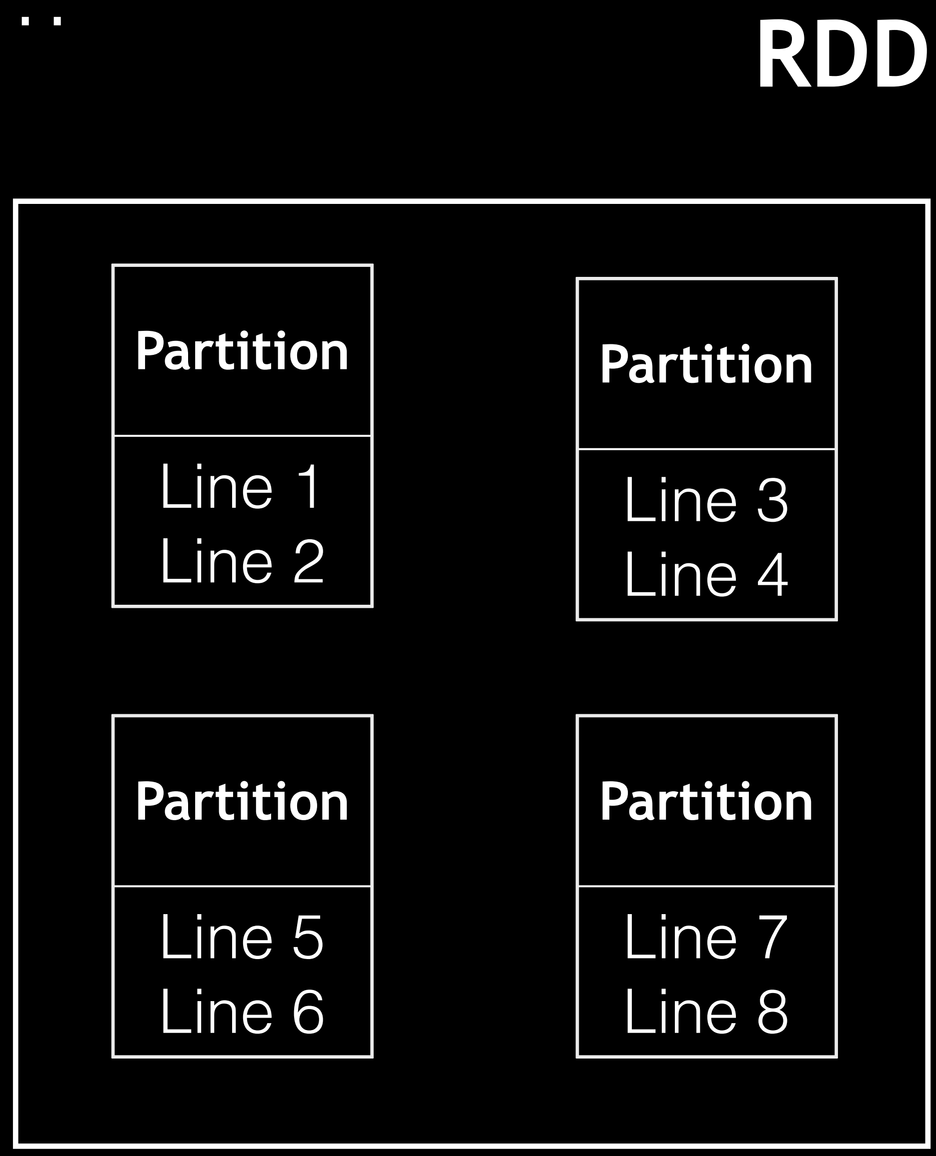
- Know how many partitions we should have for a Spark RDD

- Define what accumulators and broadcast variables are, and use cases for each

- Describe what type of input format MLlib machine learning algorithms typically expect

Resilient Distributed Datasets (RDD)

Recall our old friend…



Resilient Distributed Datasets (RDD)
Number of Partitions

- By default, Spark chooses the number of partitions based off the size of your cluster

- You have the option of parallelizing your RDD over more partitions…

- More partitions means more parallel processes, but more overhead. Choose k partitions based on:

  Gain in parallelization > Loss in overhead

- Spark documentation recommends 2-4 partitions per CPU (core) in your cluster

Resilient Distributed Datasets (RDD)

Number of Partitions

- Set **when initializing** your RDD:

```
rdd = sc.parallelize([1, 3, 4, 5,6], 16)
rdd = sc.textFile('path/to/file', 16)
```

- Set **after initializing** your RDD:

```
rdd.repartition(16)
```

Resilient Distributed Datasets (RDD)

Partitioning By Key

- In a distributed program, communication between different machines is often very expensive
- We can control the way that Spark partitions our RDDs when they are composed of key/value pairs
- We can assure that all key/value pairs for a given key will end up on the same machine. This can reduce the communication that is necessary between machines and greatly speed up our programs.

Resilient Distributed Datasets (RDD)
Partitioning By Key

- In practice:

```
rdd.partitionBy(100)
```

- This will partition the data into 100 partitions by the current key

- Only useful when a dataset is reused multiple times in key-oriented operations (groupByKey, reduceByKey, join, etc.)

Resilient Distributed Datasets (RDD)

Joins

- Spark RDDs offer all of our standard SQL joins: inner(default), left outer , right outer, full outer

- Each RDD in the join must be in the format of (key, value) pairs, where the key in each corresponds to the same variable

Resilient Distributed Datasets (RDD)

Joins

Transactions table

User ID	Store ID
100156	1
100156	2
100157	1
. . .	. . .

Store lookup Table

Store ID	Name
1	REI
2	Sports!
3	Target
4	Hippy

Resilient Distributed Datasets (RDD)

Joins

```
transactions_rdd = sc.parallelize([(100156, 1), (100156, 2), (100157, 1)])

store_lookup_rdd = sc.parallelize([(1, "REI"), (2, "Sports!"), (3, "Target"), (4, "Hippy")])
```

```
joined_rdd = transactions_rdd.map(lambda (key, value): \
    (value, key)).join(store_lookup_rdd)
```

```
[(1, (100156, 'REI')), (1, (100157, 'REI')), (2, (100156, 'Sports!'))]
```

Advanced Features
Accumulators

- A type of shared variable across all worker machines
- Provide a simple syntax for **aggregating** values from worker nodes back to the driver program
- Most common use is to count events that occur during the job for debugging purposes

Advanced Features
Accumulators

- In practice:

```
file = sc.textFile(inputFile)
blank_lines = sc.accumulator(0)  Initialization value

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

call_signs = file.flatMap(extract_call_signs)
```

Advanced Features
Broadcast Variables

- Another type of shared variable across all worker machines
- Allow the program to efficiently send a large, read-only value (or values) to all the worker nodes
- By default, Spark automatically sends all variables referenced in our functions to the worker nodes for each task, which **can be highly inefficient**. We might end up sending **multiple copies** of the same variables to the same workers!
- Broadcast variables are a solution to this problem

Advanced Features
Broadcast Variables

- Can be particularly useful to **broadcast** a small lookup table across our worker nodes

Transactions table

User ID	Store ID
100156	1
100156	2
100157	1
⋮	⋮

Store lookup Table

Store ID	Name
1	REI
2	Sports!
3	Target
4	Hippy

Advanced Features

Broadcast Variables

- In practice:

```
transactions_rdd = sc.parallelize([(100156, 1), (100156, 2), (100157, 1)])

store_lookup_broadcasted = sc.broadcast({1: "REI", 2:"Sports!", 3: "Target", 4: "Hippy"})

def process_transactions(transaction, store_lookup_broadcasted):
    store_id = transaction[0]
    store_name = store_lookup_broadcasted.value.get(store_id)
    user_id = transaction[1]
    return (store_id, (user_id, store_name))
```

Tell spark to broadcast these values

Lookup broadcasted values

```
transactions_rdd = transactions_rdd.map(lambda(key, value): (value, key))

lookedup_rdd = transactions_rdd.map(lambda transaction:  \
process_transactions(transaction, store_lookup_broadcasted))
```

```
[(1, (100156, 'REI')), (1, (100157, 'REI')), (2, (100156, 'Sports!'))]
```

Spark MLlib
Algorithms

- Classification/Regression

  Logistic Regression, SVM, Naive Bayes,
  Gradient Boosted Trees, Random Forests,
  Multilayer Perceptron (Java and Scala only, for now),
  Generalized linear regression (GLM)

- Recommenders/Collaborative Filtering

  NMF (ALS)

- Decomposition

  SVD, PCA, NMF

- Clustering

  K-Means

Spark MLlib
Conventions

- For Supervised Learning

  LabeledPoint(target, feature)

    target(numeric)

    feature(numeric vector)