HI!

CS 744: RESILIENT DISTRIBUTED DATASETS

Shivaram Venkataraman

Fall 2019

Administrivia

- Assignment 1: Due Sep 24
- Project details
    - Ideas posted on Piazza by Sat.
    - Come up with your own ideas!
    - Submit groups, topics by *9/30*
    - Meet? Office hours 9/23 or 9/30

MOTIVATION: PROGRAMMABILITY

Most real applications require multiple MR steps

- Google indexing pipeline: 21 steps
- Analytics queries (e.g. sessions, top K): 2-5 steps
- Iterative algorithms (e.g. PageRank): 10's of steps

Multi-step jobs create spaghetti code

- 21 MR steps → 21 mapper and reducer classes

MOTIVATION: PERFORMANCE

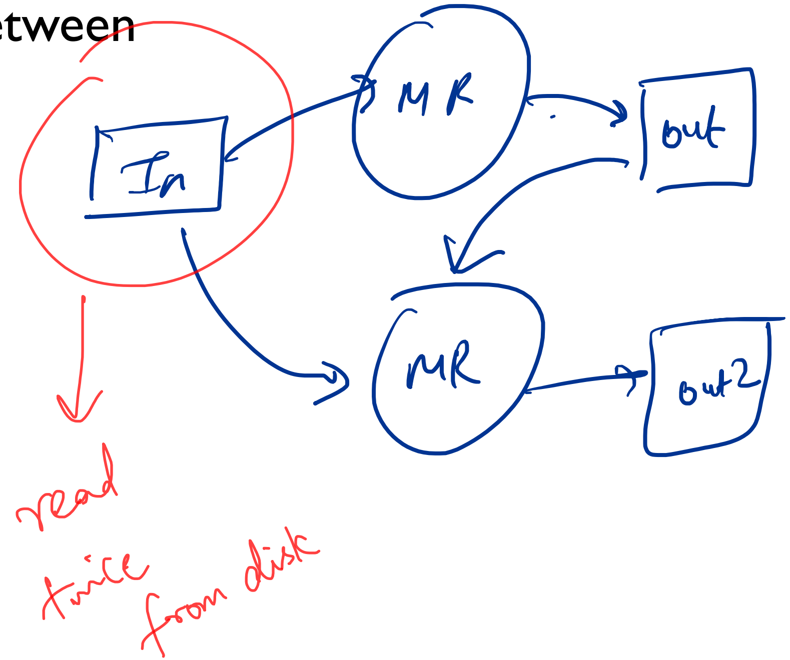
MR only provides one pass of computation

– Must write out data to file system in-between

Expensive for apps that need to *reuse* data

– Multi-step algorithms (e.g. PageRank)

– Interactive data mining



Programmability

Google MapReduce WordCount:

```
#include "mapreduce/mapreduce.h"

// User's map function
class SplitWords: public Mapper {
  public:
  virtual void Map(const MapInput& input)
  {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while (i < n && isspace(text[i]))
        i++;
      // Find word end
      int start = i;
      while (i < n && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(
            start,i-start),"1");
    }
  }
};

REGISTER_MAPPER(SplitWords);
```

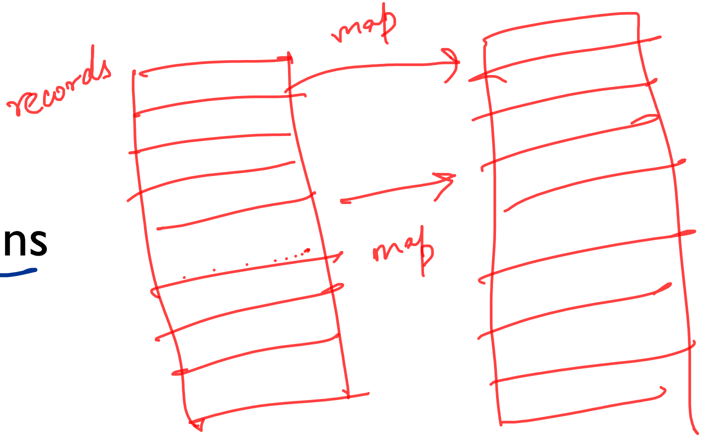
```
// User's reduce function
class Sum: public Reducer {
  public:
  virtual void Reduce(ReduceInput* input)
  {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(
                 input->value());
      input->NextValue();
    }
    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};

REGISTER_REDUCER(Sum);
```

```
int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
  MapReduceSpecification spec;
  for (int i = 1; i < argc; i++) {
    MapReduceInput* in= spec.add_input();
    in->set_format("text");
    in->set_filepattern(argv[i]);
    in->set_mapper_class("SplitWords");
  }

  // Specify the output files
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Sum");

  // Do partial sums within map
  out->set_combiner_class("Sum");

  // Tuning parameters
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();
  return 0;
 }
```

APACHE SPARK PROGRAMMABILITY

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

counts.save("out.txt")
```

APACHE SPARK



RDD

Programmability: clean, functional API

- Parallel transformations on collections
- 5-10x less code than MR
- Available in Scala, Java, Python and R

Performance

- In-memory computing primitives
- Optimization across operators

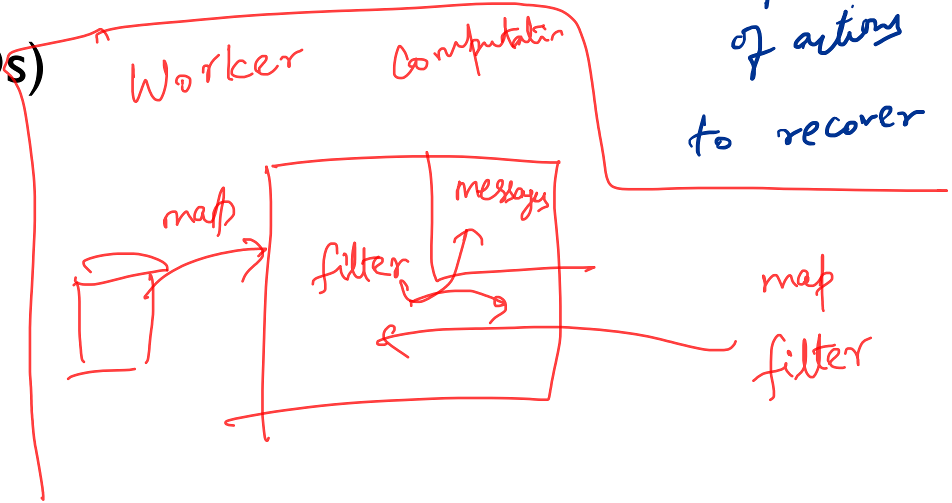
SPARK CONCEPTS

Coordination

Consistency

failure ⇒ repeat fix set of actions to recover

- Resilient distributed datasets (RDDs)
  - Immutable, partitioned collections of objects
  - May be cached in memory for fast reuse
- Operations on RDDs
  - *Transformations* (build RDDs)
  - *Actions* (compute results)
- Restricted shared variables
  - Broadcast, accumulators



Lazy execution model

⇒ Optimizations

EXAMPLE: LOG MINING

Find error messages present in log files interactively

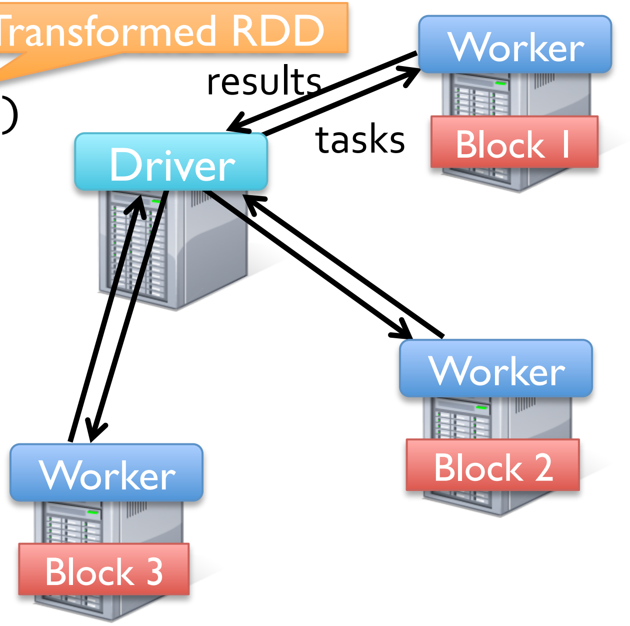
(Example: HTTP server logs)

Base RDD → read input from HDFS

Transformed RDD

Action

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.cache()
messages.filter(_.contains("foo")).count
```



INFO
ERROR
WARN
INFO

filter →

ERROR:
1912 Blah
foo

messages RDD

map →

Blah
foo

→ 1

EXAMPLE: LOG MINING

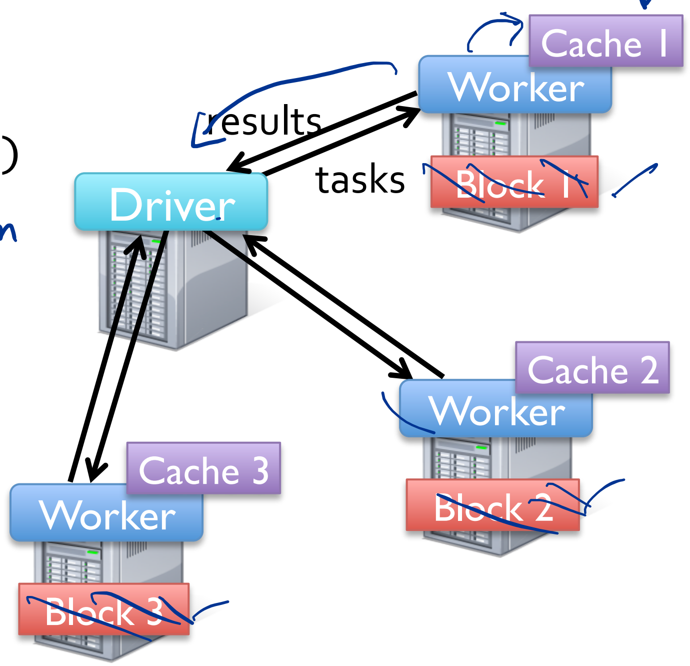
locality

Find error messages present in log files interactively

(Example: HTTP server logs)

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.cache()
messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
. . .
```

Action



Result: search 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

FAULT RECOVERY

lines =
lines.persist()
lines.take(10)

Machine1 ~~M1~~
Machine2 M2
Machine3 M3
Machine4 M1

re-execute
map, filter

```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```

.filter( ) . count

HDFS File $\xrightarrow{\text{filter}}$ Filtered RDD $\xrightarrow{\text{map}}$ Mapped RDD

filter
(func = _.contains(...))

map
(func = _.split(...))

batch
records
≃ 4K

filter → map → filter → count

Iterator model

SHARED VARIABLES

Program Driver

local variables

"Spark" variables

```
val data = spark.textFile(...).map(readPoint).cache()

// Random Projection
val M = Matrix.random(N)

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w.dot(p.x.dot(M)))) - 1)
    * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

= RDD

Large Matrix

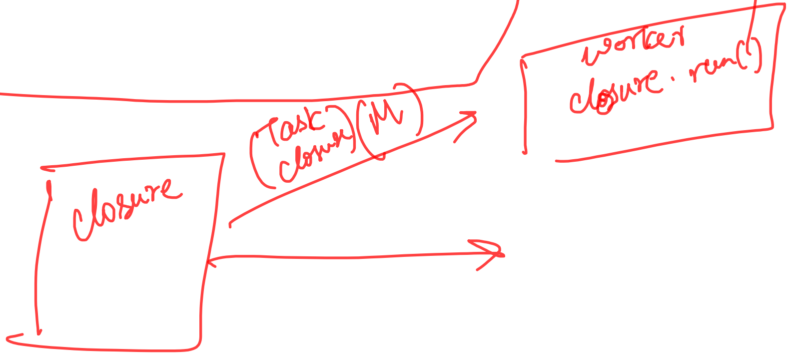
input variables

shared variables

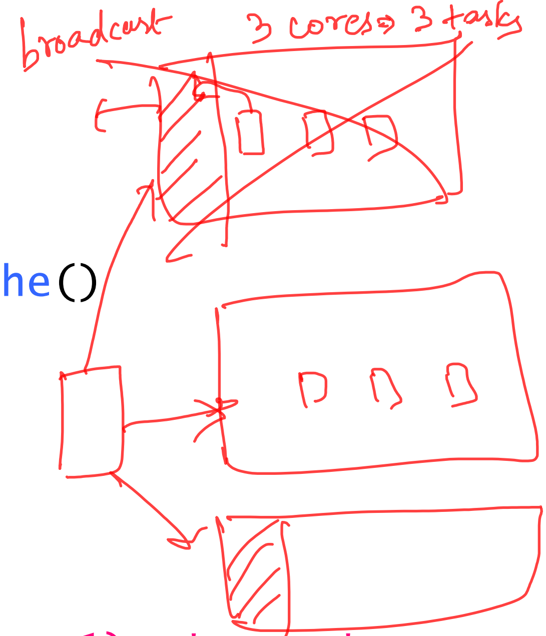
closure

adds up results

local data



BROADCAST VARIABLES




```
val data = spark.textFile(...).map(readPoint).cache()

// Random Projection
Val M = spark.broadcast(Matrix.random(N))

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w.dot(p.x.dot(M.value))) - 1)  * p.y *
   p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

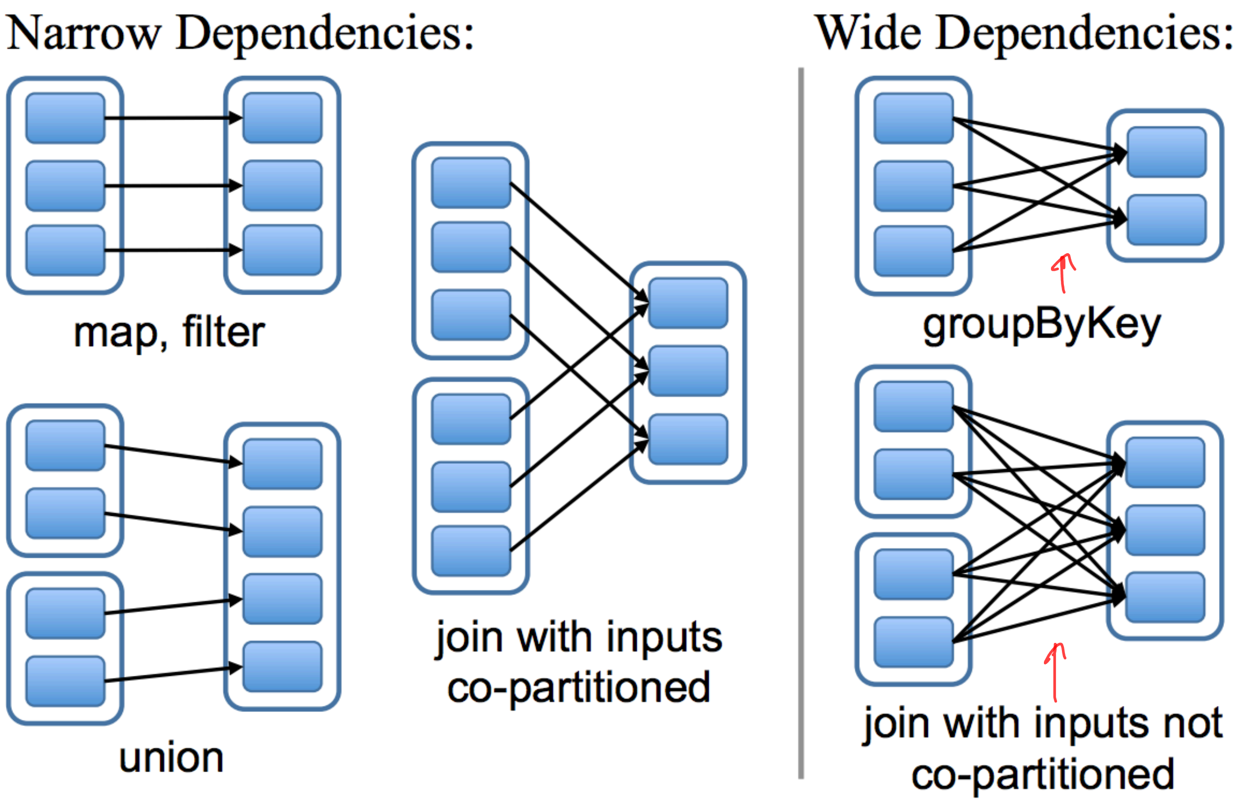
M. destroy()

OTHER RDD OPERATIONS

flat Map
reduce ByKey
save As Hadop

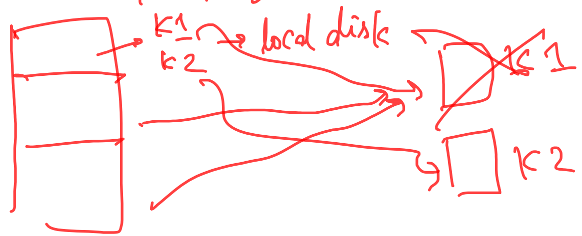
Transformations (define a new RDD)	map filter sample groupByKey reduceByKey cogroup	flatMap union join cross mapValues ...
Actions (output a result)	collect reduce take fold	count saveAsTextFile saveAsHadoopFile ...

DEPENDENCIES



JOB SCHEDULER

B = A.groupByKey()

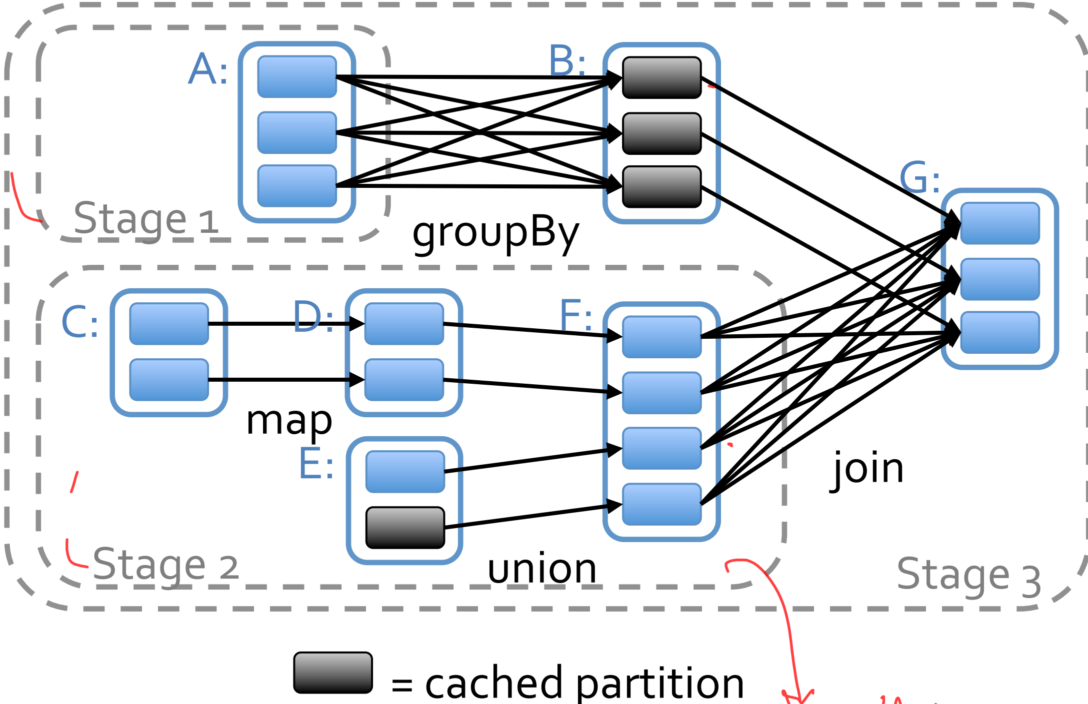


Captures RDD dependency graph

Pipelines functions into "stages"

Cache-aware for data reuse, locality

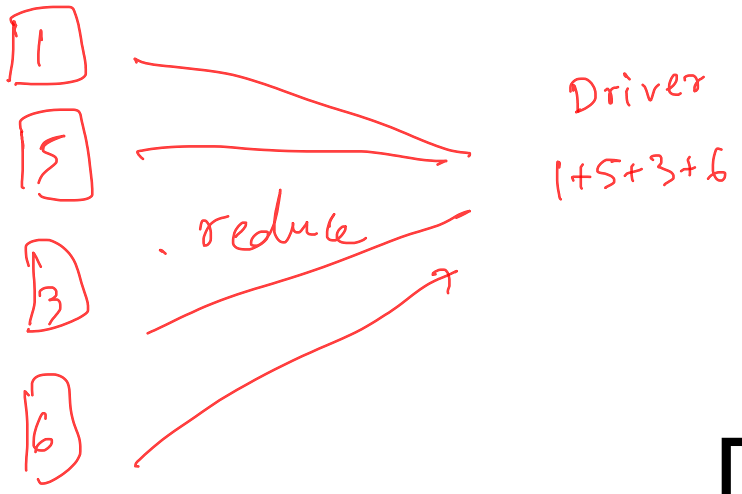
Partitioning-aware to avoid shuffles

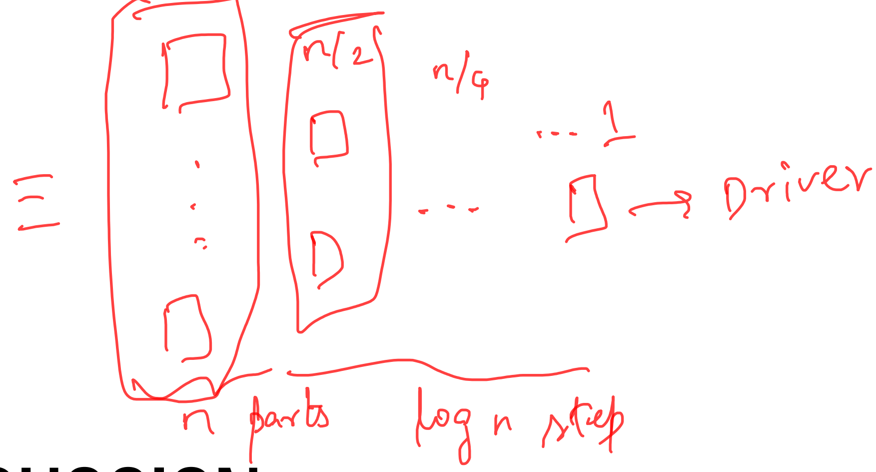


narrow deps

CHECKPOINTING

```
rdd = sc.parallelize(1 to 100, 2).map(x → 2*x)
rdd.checkpoint()
```





DISCUSSION

Co group   2 RDDs

https://forms.gle/Gg2K1hsGFJpFbmSj9

reduce By Key   n/2 keys

Partitioner to control keyspace

```
while ( ) {
rdd = rdd.mapPartitions { (r, idx) =>
    (idx/2, r)
}.reduceByKey(n/2)
n=n/2
}
```

SPARK ADOPTION

Open source Apache Project, > 1000 contributors

Extensions to SQL, Streaming, Graph processing

Unified Platform for Big Data Applications

NEXT STEPS

- Next week: Resource Management
  - Mesos, YARN
  - DRF
- Assignment 1 is due soon!