CS 744: RESILIENT DISTRIBUTED DATASETS

Shivaram Venkataraman

Fall 2020

ADMINISTRIVIA

- Assignment 1: Due Sep 21, Monday at 10pm! Posted some submission notes to Piazza!
- Assignment 2: ML will be released Sep 22
- Course ~~Final~~ project details: Next week

MOTIVATION: PROGRAMMABILITY

Discussion

All of our inputs and outputs to Disk

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

Grep

Latency or Time to run MR has a lower bound in time to read input write output to disk

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

① fewer lines of code

② Is this because of Scala? Lambda function

Map
Reduce } → functions

inline functions

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

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

Scala

Python

④ Type checking allows for chaining safely

③ Scala to operate on collections (local)

Same API to compute in a distributed setting as local

APACHE SPARK

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

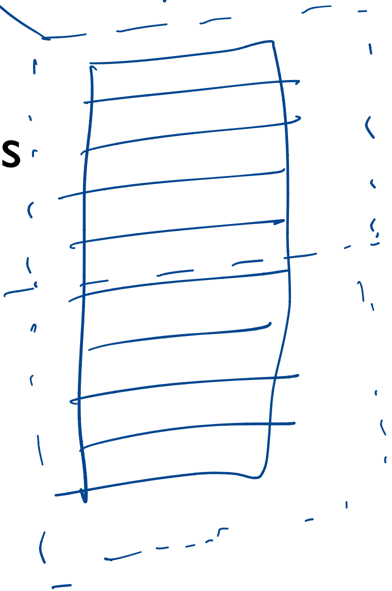
once we create this we cannot change its contents

track changes using lineage

List class <Type>

collection of records

- 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



split or partitioned

RDD

RDD → RDD

List <Int> → List <String>

Int, Type

File

EXAMPLE: LOG MINING

for ( ) {
read a line
filter, map, map, inc count
}
put this in cache

RDD [String]

Find error messages present in log files interactively

(Example: HTTP server logs)

create an RDD from a file

Base RDD

Transformed RDD

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

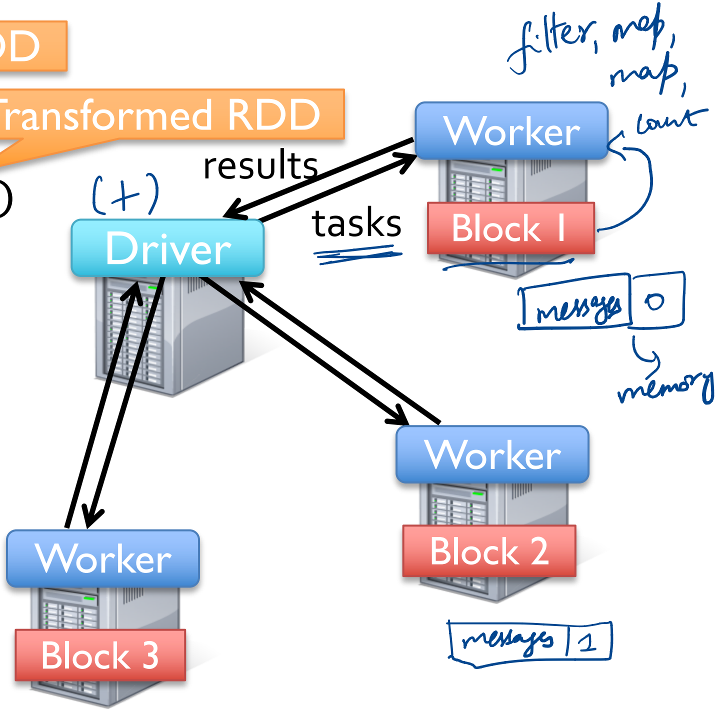
Action

are handle to data

SparkDriver

memory

nothing happens till here



EXAMPLE: LOG MINING

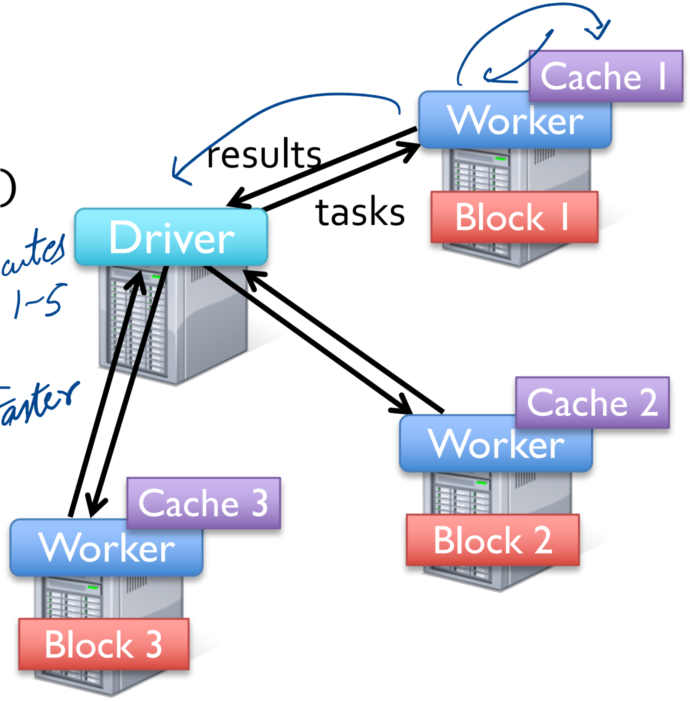
Find error messages present in log files interactively

(Example: HTTP server logs)

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

executes 1-5

Faster



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

FAULT RECOVERY

Count

cache (messages)

on Count

messages

read from HDFS

Base RDD (HDFS)

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

HDFS File → Filtered RDD → Mapped RDD

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

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

"narrow deps"

1 → 1 transform

2 → 2

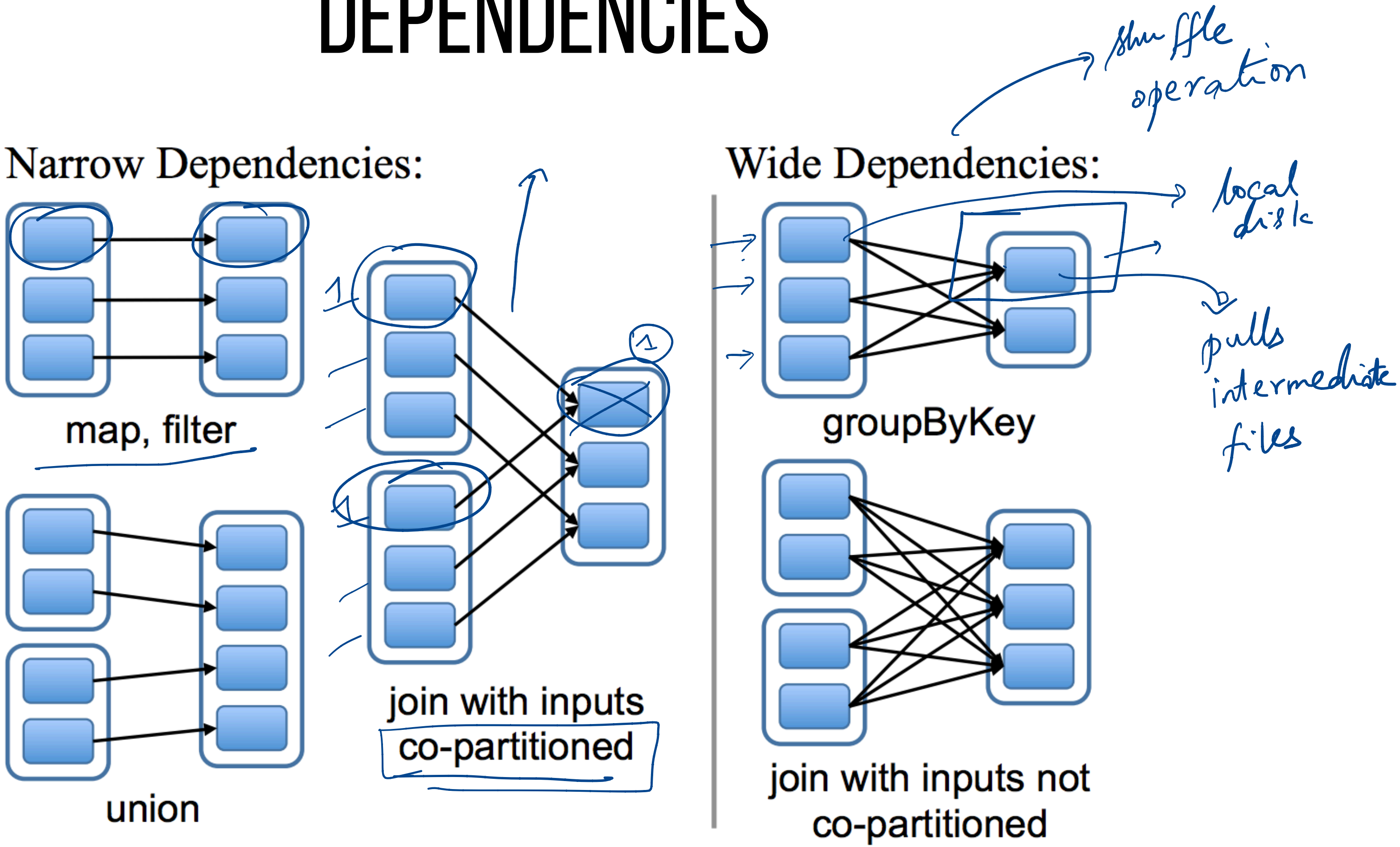
Parent: HDFS file
Transform: filter function

Parent: Filtered RDD
Transform: map function

OTHER RDD OPERATIONS

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



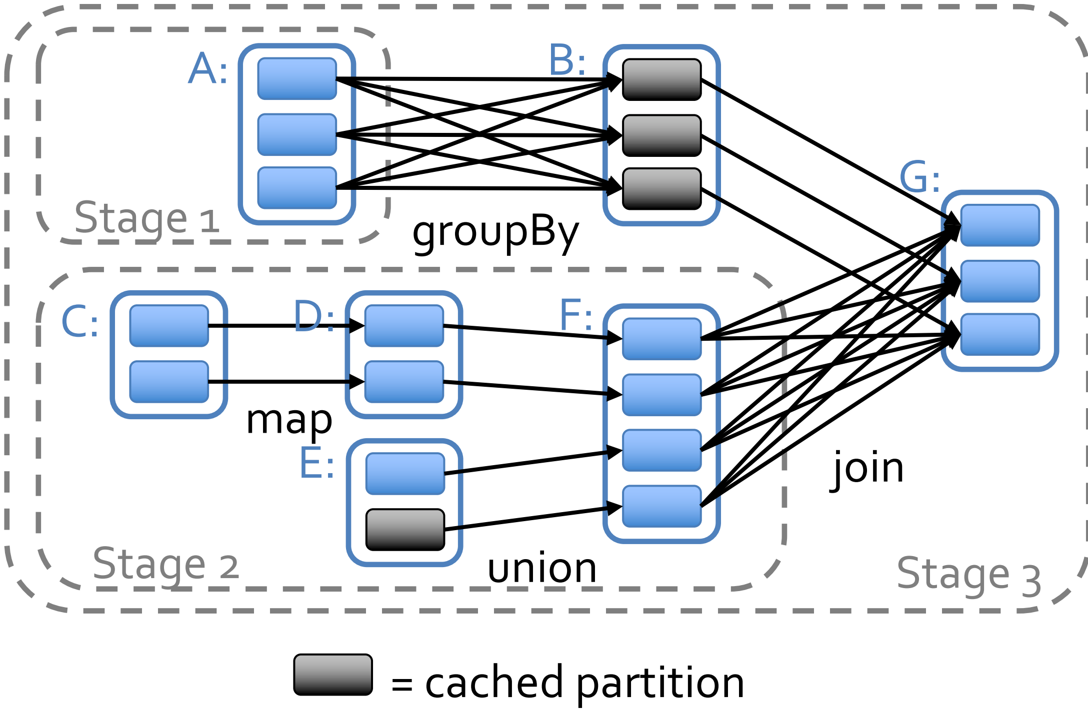
JOB SCHEDULER (1)

Driver → W1, W2 → Stage 0 → Driver Stage 1 → ~~Stage 0~~ W1, W2

Captures RDD dependency graph

Pipelines functions into "stages"

$\underbrace{\text{filter, map, map}}_{\text{narrow}}$ → One Task



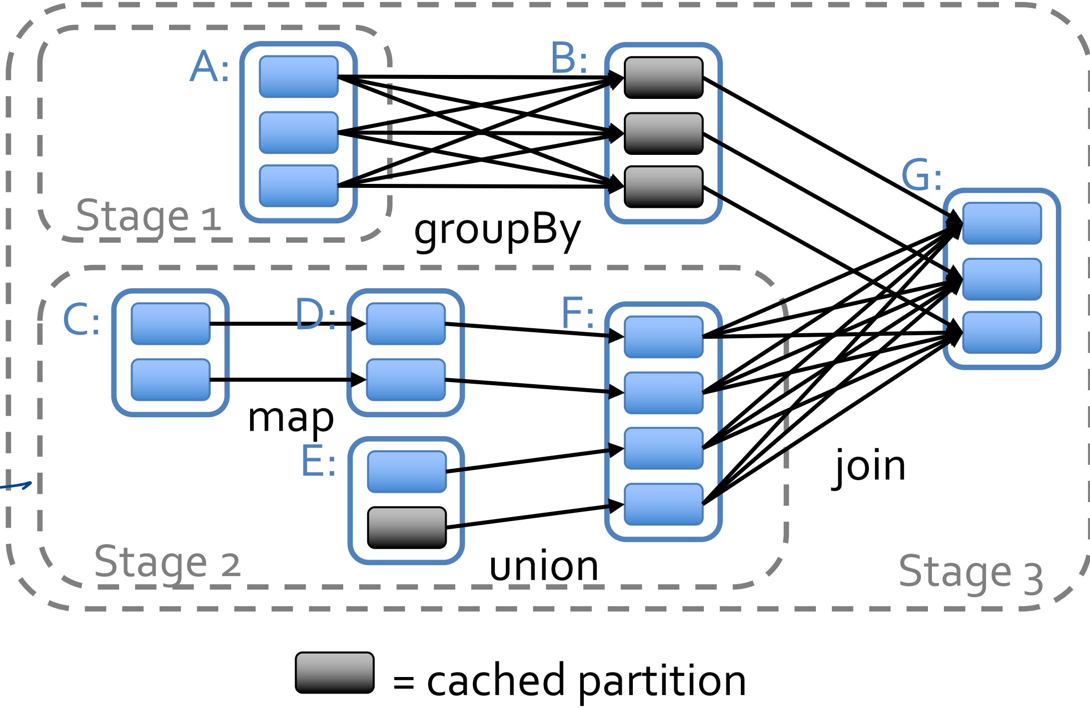
JOB SCHEDULER (2)

also in MR

Cache-aware for data reuse, locality

Partitioning-aware to avoid shuffles

co-partition



Scheduler fails??

*

Job aborts

CHECKPOINTING

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

SUMMARY

reduce → why at Driver?

Map Reduce

↳ map

reduceByKey

equivalent in Spark

Spark: Generalize MR programming model

Support in-memory computations with RDDs

Job Scheduler: Pipelining, locality-aware

mimic Scala API

reduce → Spark behaves like this

DISCUSSION

https://forms.gle/4JDXfpRuVaXmQHxD8

tree reduction

16 → 4 → 1

Workers

1

5

3

6

network

Driver (1+5+3+6) = 15

→ Computation

→ bottleneck

→ p partitions

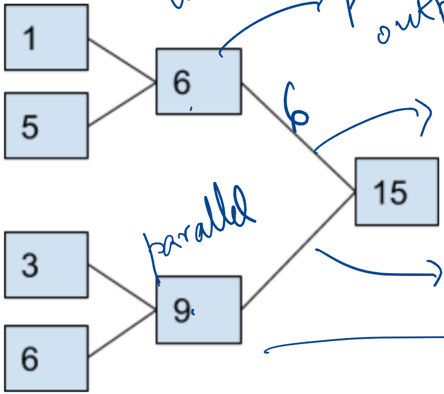
d bytes

p × d bytes = Driver

→ action (like, count)

```
for (i <- 1 to numIters) {
        val modelBC = sc.broadcast(model)
        val grad = data.mapPartitions(iter => gradient(iter, modelBC.value))
        val aggGrad = grad.reduce(case(x, y) => add(x, y))
        model = computeUpdate(aggGrad, model)
}
```

workers → partial outputs



parallel

6 → reduction in network at Driver

Similar in spirit to Combiner in MapReduce!

How can we do this?

→ (0, ...) reduceByKey

→ (0, ...) → (0,

n partition

→ (1,) → (0,

→ (1, ...)

reduceBy key

n/2 partition

→ reduceBy Key

→ treeReduce

RDD.scala

When would reduction trees be better than using `reduce` in Spark?
When would they not be ?

Overhead with doing work in stages

→ scheduling, task creation
shuffle overheads

→ Compute & data transmitted is small

→ treeReduce might be slow.

Device is full error

↳ Disk is full

↳ Shuffle data → local disk

/ partition

/mnt → big!

worker

" df -h "

Python

Python → Scala → Scala → Python

Python → Scala → Scala → Python

dstat → Tool

↳ CPU, network disk util

event logging enabled

↓

save job info

NEXT STEPS

Next week: Resource Management

- Mesos, YARN
- DRF

Assignment 1 is due soon!

FT → memory replication
vs
lineage
↓
Frequency of failure

Trade-offs
network speed
vs
memory speed

Review form
↳ when is MR better Spark
vice versa

→ Spark is better when multiple passes over data & data fits in memory

MR is better when → single pass
→ scheduling overhead ?