Good morning!

CS 744: RESILIENT DISTRIBUTED DATASETS

Shivaram Venkataraman
Fall 2021
- Assignment 1: Due Sep 28, Tuesday at 10pm!
- Assignment 2: ML will be released Sep 29

- REMINDER: Submit your discussions
  - Within 24 hrs after end of class (11am next day)
  - Each student needs to submit

- Course project details: Next week
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
#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) {
    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

- Fewer lines of code
- Trace how operations are chained. Type checked!
- Use of inline functions
  - Minics local programs

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

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

Create from a file
- Integer on screen
- File save out

Counters
EXAMPLE: LOG MINING

Find error messages present in log files interactively
(Example: HTTP server logs)

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.cache()  # when messages is processed, save it
messages.filter(_.contains("foo")).count = 400
lines.filter(_.starts("INFO"))
```

create an RDD

```
msg1
msg2
msg3
```

results
tasks

Worker
Worker
Worker

Driver

Block 1
Block 2
Block 3
EXAMPLE: LOG MINING

Find error messages present in log files interactively
(Example: HTTP server logs)

```scala
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
...
```

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

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

HDFS File

Filtered RDD

Mapped RDD

• Lineage: History of transformations that created this RDD

Assumption: Input file is still available
# Other RDD Operations

### Transformations
(Define a new RDD)

- `map`
- `filter`
- `sample`
- `groupByKey`
- `reduceByKey`
- `cogroup`

### Actions
(Output a result)

- `collect`
- `reduce`
- `take`
- `fold`

- `flatMap`
- `union`
- `join`
- `cross`
- `mapValues`
- `...`

- `count`
- `saveAsTextFile`
- `saveAsHadoopFile`
- `...`
Intermediate files are on local disk.

Save

Narrow Dependencies:
- map, filter
- union

Wide Dependencies:
- groupByKey
- join with inputs not co-partitioned
- join with inputs co-partitioned

shuffle
Captures RDD dependency graph

Pipelines functions into “stages”

All narrow deps are coalesced inside a stage
Cache-aware for data reuse, locality

Partitioning-aware to avoid shuffles

Stage 1

Stage 2

Stage 3

Join

map

groupBy

union

A:  
B:  
C:  
D:  
E:  
F:  
G:  

Output → a b c 5 8 9

= cached partition
SUMMARY

Spark: Generalize MR programming model

Support in-memory computations with RDDs

Job Scheduler: Pipelining, locality-aware
DISCUSSION

https://forms.gle/nPdjYq9D4nE4gtAD9
How can we implement binary reduction tree in Spark?

```scala
for (i <- 1 to numIters) {
    val modelBC = sc.broadcast(model)
    val grad = data.mapPartitions(iter => gradient(iter, modelBC.value))
    val aggGrad = grad.reduce((x, y) => add(x, y))
    model = computeUpdate(aggGrad, model)
}
```
for (i <- 1 to numIter) {
    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)
}
When would reduction trees be better than using `reduce` in Spark? When would they not be?

- Very large number of partitions
- If shuffles are expensive

→ good for treeReduce
NEXT STEPS

Next week: Resource Management
  - Mesos
  - DRF
Assignment 1 is due soon!
rdd = sc.parallelize(1 to 100, 2).map(x → 2*x)
rdd.checkpoint()