CS 744: RESILIENT DISTRIBUTED DATASETS

Shivaram Venkataraman
Fall 2020
- Assignment 1: Due Sep 21, Monday at 10pm!
- Assignment 2: ML will be released Sep 22
- Final 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 \(\rightarrow\) 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

\( \rightarrow \) Grep
# 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) {
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }
    // Emit sum for input->key()
    Emit(IntToString(value));
  }
  REGISTER_REDDUCE(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");
  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;`
val file = spark.textFile("hdfs://...")
val counts = fileflatMap(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

Once we create this, we cannot change its contents.

Track changes using lineage.
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
```
EXAMPLE: LOG MINING

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

```python
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

filter (func = _.contains(...))
map (func = _.split(...)
## OTHER RDD OPERATIONS

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Actions (output a result)</th>
<th>Other Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>collect</td>
<td>flatMap</td>
</tr>
<tr>
<td>filter</td>
<td>reduce</td>
<td>union</td>
</tr>
<tr>
<td>sample</td>
<td>take</td>
<td>join</td>
</tr>
<tr>
<td>sampleByKey</td>
<td>fold</td>
<td>cross</td>
</tr>
<tr>
<td>reduceByKey</td>
<td></td>
<td>mapValues</td>
</tr>
<tr>
<td>cogroup</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

**Actions** (output a result)
- collect
- reduce
- take
- fold
- count
- saveAsTextFile
- saveAsHadoopFile
DEPENDENCIES

Narrow Dependencies:
- map, filter
- union

Wide Dependencies:
- groupByKey

join with inputs co-partitioned

join with inputs not co-partitioned
Captures RDD dependency graph

Pipelines functions into “stages”
Cache-aware for data reuse, locality

Partitioning-aware to avoid shuffles

Also in MR

Stage 1

Stage 2

Stage 3

Scheduler fails ??

Job aborts

= cached partition
CHECKPOINTING

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

Spark: Generalize MR programming model

Support in-memory computations with RDDs

Job Scheduler: Pipelining, locality-aware
DISCUSSION

https://forms.gle/4JDXfpRuVaXmQHxD8
```scala
for (i <- 1 to numIter) {
  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)
}
```
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
  - tree reduce might be slow.
Device is full error

 Device is full

 Disk is full

 HDFS data → local disk

 "df -h" worker

 "df -h"

 Python

 Python → Scala

 Scala → Python

 Scala → Python

 Scala → Python

 Scala → Python

 Scala

 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!

Review form
- When is MR better?
  - Spark is better when multiple passes over data & data fits in memory
  - MR is better when single pass
- scheduling overhead?

Trade-offs
- Lineage vs. network speed
- Memory vs. memory speed

Frequency of failure vs. memory speed