Announcements

- HW8 due next Fri
- Extra office hours today: Rajiv @ 6pm in CSE 220
- No lecture Monday (holiday)
- Guest lecture Wednesday
  - Kris Hildrum from Google will be here
  - she works on technologies related to Spark etc.
  - whatever she talks about will be on the final

Spark

- Open source system from Berkeley
- Distributed processing over HDFS
- Differences from MapReduce:
  - Multiple steps, including iterations
  - Stores intermediate results in main memory
  - Supports SQL
- Details: [http://spark.apache.org/examples.html](http://spark.apache.org/examples.html)

Spark Interface

- Spark supports a Scala interface
- Scala = ext of Java with functions/closures
  - will show Scala/Spark examples shortly...
- Spark also supports a SQL interface
- It compiles SQL into Scala
- For HW8: you only need the SQL interface!

RDD

- RDD = Resilient Distributed Datasets
  - A distributed relation, together with its lineage
  - Lineage = expression that says how that relation was computed = a relational algebra plan
- Spark stores intermediate results as RDD
- If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply recompute the lost partition of the RDD

Programming in Spark

- A Spark/Scala program consists of:
  - Transformations (map, reduce, join…). Lazy
  - Actions (count, reduce, save…). Eager
- RDD[T] = an RDD collection of type T
  - Partitioned, recoverable (through lineage), not nested
- Seq[T] = a Scala sequence
  - Local to a server, may be nested
Example

Given a large log file `hdfs://logfile.log`
retrieve all lines that:
• Start with “ERROR”
• Contain the string “sqlite”

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```

MapReduce Again…

Steps in Spark resemble MapReduce:
• `col.filter(p)` applies in parallel the predicate `p` to all elements `x` of the partitioned collection, and returns those `x` where `p(x) = true`
• `col.map(f)` applies in parallel the function `f` to all elements `x` of the partitioned collection, and returns a new partitioned collection

Scala Primer

• Functions with one argument:
  `_`.contains("sqlite")
  `_ > 6`
• Functions with more arguments
  `(x => x.contains("sqlite"))` (x => x > 6)
  `((x,y) => x+3*y)`
• Closures (functions with variable references):
  `var x = 5; rdd.filter(_ > x)`
  `var s = "sqlite"; rdd.filter(x => x.contains(s))`

Persistence

If any server fails before the end, then Spark must restart

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```
Persistence

If any server fails before the end, then Spark must restart

lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.persist()

Spark can recompute the result from errors

Example

R(A,B) SELECT count(*) FROM R, S
WHERE R.B > 200 and S.C < 100 and R.A = S.A

Programming in Spark

• A Spark/Scala program consists of:
  – Transformations (map, reduce, join…). Lazy
  – Actions (count, reduce, save…). Eager

• RDD[T] = an RDD collection of type T
  – Partitioned, recoverable (through lineage), not nested

• Seq[T] = a Scala sequence
  – Local to a server, may be nested

MapReduce ~> Spark

• input into an RDD
• map phase becomes .flatMap
• shuffle & sort becomes .groupByKey
• reduce becomes another .flatMap
• save output to HDFS
SQL -> Spark

• You know enough to execute SQL on Spark!
  
  Idea: (1) SQL to RA + (2) RA on Spark

  - \( \sigma \) = filter
  - \( \pi \) = map
  - \( \gamma \) = groupByKey
  - \( \times \) = crossProduct
  - \( \bowtie \) = join

  Spark SQL does small optimizations to RA
  Also chooses between broadcast and parallel joins

PageRank

• Page Rank is an algorithm that assigns to each page a score such that pages have higher scores if more pages with high scores link to them

• Page Rank was introduced by Google, and, essentially, defined Google

PageRank toy example

Input graph

Superstep 0

Superstep 1

Superstep 2

PageRank

for \( i = 1 \) to \( n \):
  \( r[i] = 1/n \)
repeat
  for \( j = 1 \) to \( n \):
    \( contribs[j] = 0 \)
  for \( i = 1 \) to \( n \):
    \( k = \text{links}[i].\text{length()} \)
    for \( j \) in \( \text{links}[i] \):
      \( contribs[j] += r[i] / k \)
  for \( i = 1 \) to \( n \):
    \( r[i] = contribs[i] \)
  until convergence

\( ^{\text{usually 10-20 iterations}} \)

PageRank

Random walk interpretation:

Start at a random node
At each step, randomly choose an outgoing link and follow it.
Repeat for a very long time

\[ r[i] = \text{prob. that we are at node } i \]

\( \text{for } i = 1 \) to \( n \): \( r[i] = 1/n \)
repeat
  for \( j = 1 \) to \( n \):
    \( contribs[j] = 0 \)
  for \( i = 1 \) to \( n \):
    \( k = \text{links}[i].\text{length()} \)
    for \( j \) in \( \text{links}[i] \):
      \( contribs[j] += r[i] / k \)
  for \( i = 1 \) to \( n \):
    \( r[i] = contribs[i] \)
  until convergence

\( ^{\text{usually 10-20 iterations}} \)

\( \text{Random walk interpretation:} \)

\text{Start at a random node } i
\text{At each step, randomly choose an outgoing link and follow it.}
\text{Improvement: with small prob. a restart at a random node.}

PageRank

Random walk interpretation:

Start at a random node
At each step, randomly choose an outgoing link and follow it.
Repeat for a very long time

\( r[i] = \text{prob. that we are at node } i \)

\( \text{for } i = 1 \) to \( n \):
  \( r[i] = 1/n \)
repeat
  for \( j = 1 \) to \( n \):
    \( contribs[j] = 0 \)
  for \( i = 1 \) to \( n \):
    \( k = \text{links}[i].\text{length()} \)
    for \( j \) in \( \text{links}[i] \):
      \( contribs[j] += r[i] / k \)
  for \( i = 1 \) to \( n \):
    \( r[i] = contribs[i] \)
  until convergence

\( ^{\text{usually 10-20 iterations}} \)

PageRank

Random walk interpretation:

Start at a random node
At each step, randomly choose an outgoing link and follow it.
Repeat for a very long time

\( r[i] = \text{prob. that we are at node } i \)

\( \text{for } i = 1 \) to \( n \):
  \( r[i] = 1/n \)
repeat
  for \( j = 1 \) to \( n \):
    \( contribs[j] = 0 \)
  for \( i = 1 \) to \( n \):
    \( k = \text{links}[i].\text{length()} \)
    for \( j \) in \( \text{links}[i] \):
      \( contribs[j] += r[i] / k \)
  for \( i = 1 \) to \( n \):
    \( r[i] = contribs[i] \)
  until convergence

\( ^{\text{usually 10-20 iterations}} \)

\( \text{Random walk interpretation:} \)

\text{Start at a random node } i
\text{At each step, randomly choose an outgoing link and follow it.}
\text{Improvement: with small prob. a restart at a random node.}

PageRank

Random walk interpretation:

Start at a random node
At each step, randomly choose an outgoing link and follow it.
Repeat for a very long time

\( r[i] = \text{prob. that we are at node } i \)

\( \text{for } i = 1 \) to \( n \):
  \( r[i] = 1/n \)
repeat
  for \( j = 1 \) to \( n \):
    \( contribs[j] = 0 \)
  for \( i = 1 \) to \( n \):
    \( k = \text{links}[i].\text{length()} \)
    for \( j \) in \( \text{links}[i] \):
      \( contribs[j] += r[i] / k \)
  for \( i = 1 \) to \( n \):
    \( r[i] = contribs[i] \)
  until convergence

\( ^{\text{usually 10-20 iterations}} \)
Google Dataflow

- Similar to Spark/Scala
- Allows you to lazily build pipelines and then execute them
- Much simpler than multi-job MapReduce

Dataflow Example Pipeline

Counting words again...

Dataflow Example Code

```java
Pipeline p = Pipeline.create(options);
p.apply(TextIO.Read.from("gs://dataflow-samples/shakespeare/kinglear.txt"))
  .apply(ParDo.named("ExtractWords").of(new DoFn<String, String>() {
    @Override
    public void processElement(ProcessContext c) {
      for (String word : c.element().split("[^a-zA-Z]+")) {
        if (!word.isEmpty()) {
          c.output(word);
        }
      }
    }
  })))
  .apply(TextIO.Write.to("gs://my-bucket/counts.txt"));
p.run();
```

Summary

- Parallel databases
  - Predefined relational operators
  - Optimization
  - Transactions
- MapReduce
  - User-defined map and reduce functions
  - Must implement/optimize manually relational ops
  - No updates/transactions
- Spark
  - Predefined relational operators
  - Must optimize manually
  - No updates/transactions

Summary cont.

- All of these technologies use dataflow engines:
  - Google Dataflow (on top of MapReduce)
  - Spark (on top of Hadoop)
  - AsterixDB (on top of Hyracks)
- Spark & AsterixDB map SQL to a dataflow pipeline
  - SQL => RA => dataflow operators (group, join, map)
  - could do the same thing for Google Dataflow
- None of these systems optimize RA very well (as of 2015)
  - Spark has no indexes
  - AsterixDB has indexes but no statistics
- Future work should improve that