CSE 344: Section 7
MapReduce (HW6)

May 10th, 2018

Apache

Cluster-computing framework

Apache Hadoop Mapreduce vs. Apache Spark

https://www.datamation.com/data-center/hadoop-vs.-spark-the-new-age-of-big-data.html

“Hadoop MapReduce”

Distributed File System (DFS)

MapReduce Job:

- Map Task (EmitIntermediate)
- Reduce Task (Emit)

Fault Tolerance (replicated chunks, write intermediate files to disk)

“Spark” (HW6)

- Resilient Distributed Datasets (RDD)
  - A distributed, immutable 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

“Spark” (HW6)

A Spark program consists of :

- Transformations (map, join, sort…) -> **Lazy**
- Actions (count, reduce, save...) -> **Eager**
    - Eager: operators are executed immediately
    - Lazy: operators are not executed immediately
        - A operator tree is constructed in memory instead
        - Similar to a relational algebra tree

Spark Objects for HW6

```
Row // Represents one row of output from a relational operator

RowFactory.create(Objects...)

Dataset<Row>

JavaRDD<Row>

JavaPairRDD<K, V>

Tuple2<Type1, Type2> // you can leave the generics empty
```

Spark Methods for HW6

`spark.sql(“SELECT ... FROM ...”)` spark must be a SparkSession

`d.filter(t -> f(t) == true/false)`

`d.distinct()`

`d.map()` d must be a JavaRDD

`d.mapToPair(t -> new Tuple2<>(K, V))`

`d.reduceByKey((v1, v2) -> f(v1, v2))` d must be a JavaPairRDD

Spark Javadoc!!!

Collections in Spark

- RDD<T> = an RDD collection of type T
  - Partitioned, recoverable (through lineage), not nested
- Seq<T> = a 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"

lines, errors, sqlerrors
have type JavaRDD<String>

```
s = SparkSession.build           reate();

lines = s.read().tex              ile.log");

errors = lines.filter(l    > l.startsWith("ERROR"));

sqlerrors = errors.filter(                 lite"));

sqlerrors.collect();
```

Transformation:
Not executed yet…

Action:
triggers execution
of entire program

Example

Recall: anonymous functions (lambda expressions) starting in Java 8

```
errors = lines.filter(l -> l.startsWith(“ERROR”));
```

is the same as:

```
class FilterFn implements Function<Row, Boolean>{
  Boolean call (Row r)
  { return l.startsWith(“ERROR”); }
}

errors = lines.filter(new FilterFn());
```

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 collection with 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

Persistence

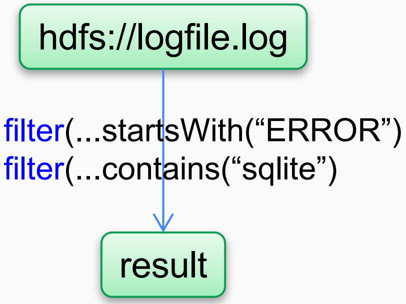
```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```

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

Persistence

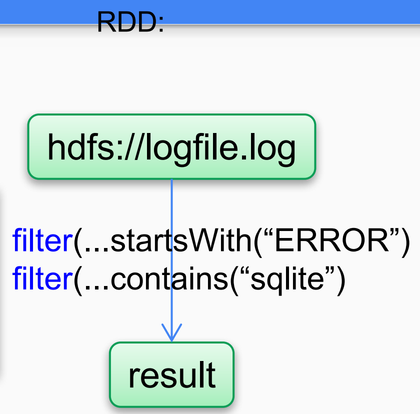
```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```

RDD:



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

Persistence



```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```

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

```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
errors.persist();
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect()
```

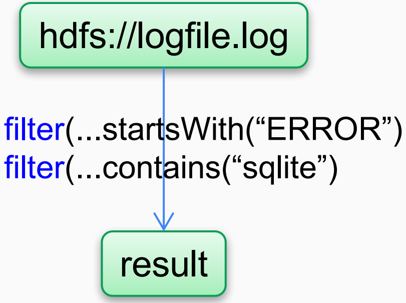
New RDD

Spark can recompute the result from errors

Persistence

RDD:

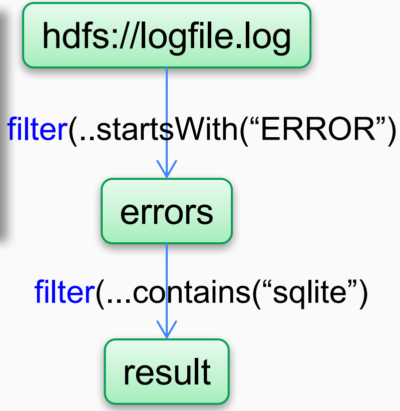
```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```



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

```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
errors.persist();
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect()
```

New RDD



Spark can recompute the result from errors

Example

R(A,B)
S(A,C)

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

```
R = s.read().textFile(“R.csv”).map(parseRecord).persist();
S = s.read().textFile(“S.csv”).map(parseRecord).persist();
```

Parses each line into an object

persisting on disk

Example

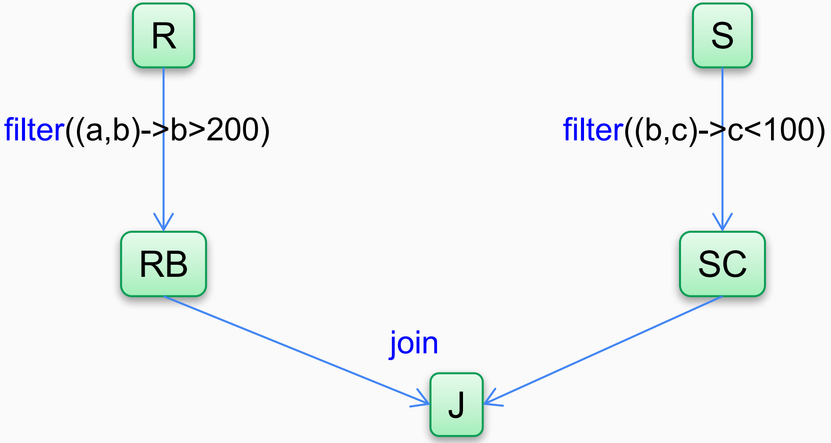
R(A,B)
S(A,C)

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

```
R = s.read().textFile(“R.csv”).map(parseRecord).persist();
S = s.read().textFile(“S.csv”).map(parseRecord).persist();
RB = R.filter(t -> t.b > 200).persist();
SC = S.filter(t -> t.c < 100).persist();
J = RB.join(SC).persist();
J.count();
```

transformations

action



Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U>
flatMap(f: T -> Seq(U)):	RDD<T> -> RDD<U>
filter(f:T->Bool):	RDD<T> -> RDD<T>
groupByKey():	RDD<(K,V)> -> RDD<(K,Seq[V])>
reduceByKey(F:(V,V)-> V):	RDD<(K,V)> -> RDD<(K,V)>
union():	(RDD<T>,RDD<T>) -> RDD<T>
join():	(RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,(V,W))>
cogroup():	(RDD<(K,V)>,RDD<(K,W)>)-> RDD<(K,(Seq<V>,Seq<W>))>
crossProduct():	(RDD<T>,RDD<U>) -> RDD<(T,U)>

Actions:	
count():	RDD<T> -> Long
collect():	RDD<T> -> Seq<T>
reduce(f:(T,T)->T):	RDD<T> -> T
save(path:String):	Outputs RDD to a storage system e.g., HDFS

DataFrames

- Like RDD, also an immutable distributed collection of data
- Organized into *named columns* rather than individual objects
    - Just like a relation
    - Elements are untyped objects called `Row`’s
- Similar API as RDDs with additional methods

    -
      ```
      people = spark.read().textFile(…);
      ageCol = people.col(“age”);
      ageCol.plus(10); // creates a new DataFrame
      ```

Datasets

- Similar to DataFrames, except that elements must be typed objects
- E.g.: `Dataset<People>` rather than `Dataset<Row>`
- Can detect errors during compilation time
- **DataFrames are aliased as `Dataset<Row>`** (as of Spark 2.0)
- You will use both Datasets and RDD APIs in HW6

Datasets API: Sample Methods

- Functional API
  - `agg(Column expr, Column... exprs)`
    Aggregates on the entire Dataset without groups.
  - `groupBy(String col1, String... cols)`
    Groups the Dataset using the specified columns, so that we can run aggregation on them.
  - `join(Dataset<?> right)`
    Join with another DataFrame.
  - `orderBy(Column... sortExprs)`
    Returns a new Dataset sorted by the given expressions.
  - `select(Column... cols)`
    Selects a set of column based expressions.
- “SQL” API
  - `SparkSession.sql(“select * from R”);`

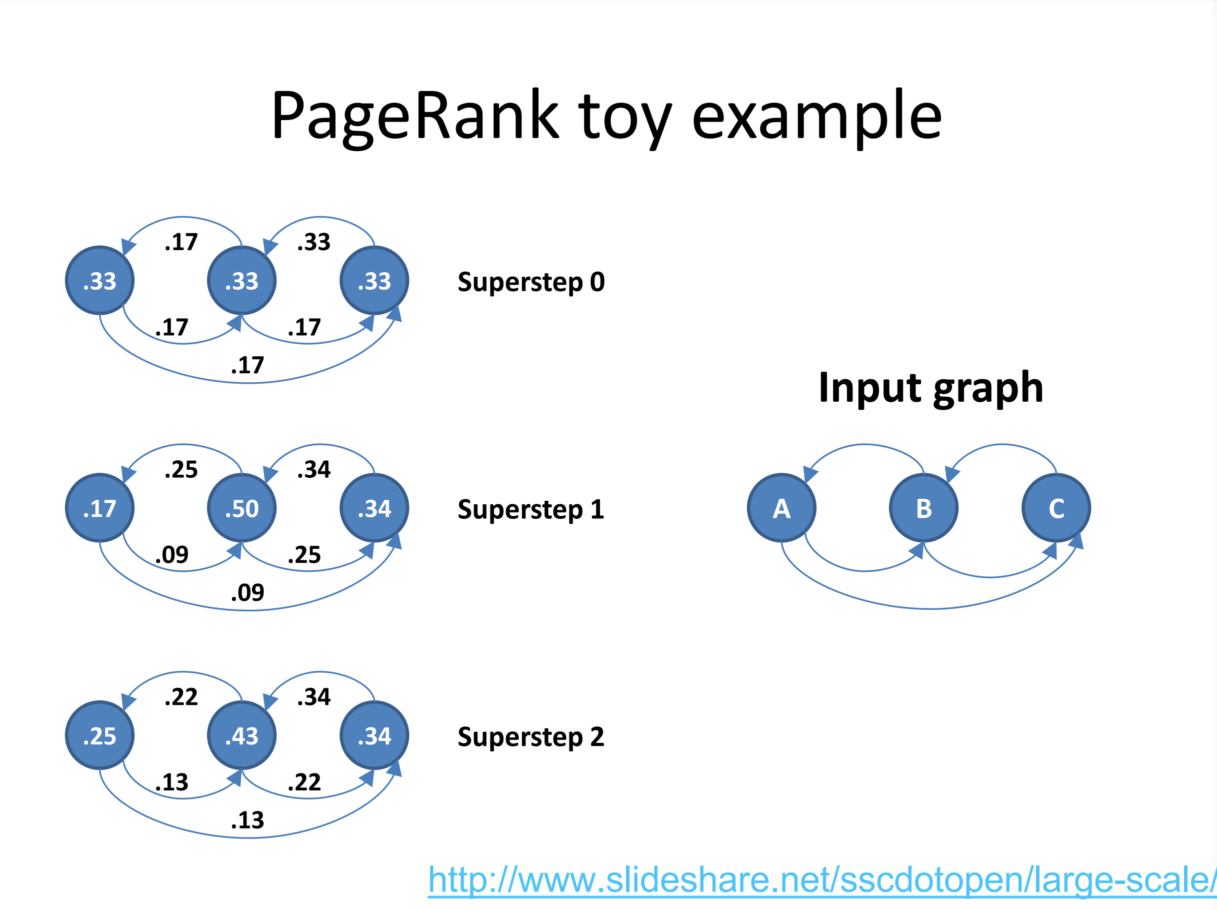
An Example Application

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



PageRank toy example

Superstep 0

Superstep 1

Superstep 2

Input graph

http://www.slideshare.net/sscdotopen/large-scale/20

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 = links[i].length()
    for j in links[i]:
       contribs[j] += r[i] / k
   for i = 1 to n: r[i] = contribs[i]
until convergence
/* usually 10-20 iterations */
```

Random walk interpretation:

Start at a random node i
At each step, randomly choose
an outgoing link and follow it.

Repeat for a very long time

r[i] = prob. that we are at node i

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 = links[i].length()
      for j in links[i]:
          contribs[j] += r[i] / k
    for i = 1 to n: r[i] = contribs[i]
until convergence
/* usually 10-20 iterations */
```

r[i] = a/N + (1-a)*contribs[i]

Random walk interpretation:

Start at a random node i
At each step, randomly choose an outgoing link and follow it.

Improvement: with small prob. a restart at a random node.

where a $\in$(0,1) is the restart probability

PageRank

links: RDD<url:string, outlinks:SEQ<string>>
ranks: RDD<url:string, rank:float>

```
for i = 1 to n:
   r[i] = 1/n

repeat
   for j = 1 to n: contribs[j] = 0
   for i = 1 to n:
      k = links[i].length()
      for j in links[i]:
          contribs[j] += r[i] / k
    for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]
until convergence
/* usually 10-20 iterations */
```

```
// spark

links = spark.read().textFile(..).map(...);
ranks = // RDD of (URL, 1/n) pairs

for (k = 1 to ITERATIONS) {

  // Build RDD of (targetURL, float) pairs
  // with contributions sent by each page
  contribs = links.join(ranks).flatMap {
    (url, lr) -> // lr: a (link, rank) pair
       links.map(dest ->
                 (dest, lr._2/outlinks.size()))
   }

  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) -> x+y)
               .mapValues(sum -> a/n + (1-a)*sum)
}
```

PageRank

links: RDD<url:string, outlinks:SEQ<string>>
ranks: RDD<url:string, rank:float>

```
for i = 1 to n:
   r[i] = 1/n

repeat
   for j = 1 to n: contribs[j] = 0
   for i = 1 to n:
      k = links[i].length()
      for j in links[i]:
          contribs[j] += r[i] / k
    for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]
until convergence
/* usually 10-20 iterations */
```

```
// spark

links = spark.read().textFile(..).map(...);
ranks = // RDD of (URL, 1/n) pairs

for (k = 1 to ITERATIONS) {

  // Build RDD of (targetURL, float) pairs
  // with contributions sent by each page
  contribs = links.join(ranks).flatMap {
    (url, lr) -> // lr: a (link, rank) pair
      links.map(dest ->
                   (dest, lr._2/outlinks.size()))
   }

  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) -> x+y)
               .mapValues(sum -> a/n + (1-a)*sum)
}
```

Key: url$_1$,
Value: ([outlink$_1$, outlink$_2$, …], rank$_1$)

PageRank

links: RDD<url:string, outlinks:SEQ<string>>
ranks: RDD<url:string, rank:float>

```
for i = 1 to n:
  r[i] = 1/n

repeat
  for j = 1 to n: contribs[j] = 0
  for i = 1 to n:
    k = links[i].length()
    for j in links[i]:
       contribs[j] += r[i] / k
   for i = 1 to n: r[i] = a/N + (1-a)*contribs[i]
until convergence
/* usually 10-20 iterations */
```

```
// spark

links = spark.read().textFile(..).map(...);
ranks = // RDD of (URL, 1/n) pairs

for (k = 1 to ITERATIONS) {

  // Build RDD of (targetURL, float) pairs
  // with contributions sent by each page
  contribs = links.join(ranks).flatMap {
    (url, lr) -> // lr: a (link, rank) pair
       links.map(dest ->
                  (dest, lr._2/outlinks.size()))
  }

  // Sum contributions by URL and get new ranks
   nks = contribs.reduceByKey((x,y) -> x+y)
                .mapValues(sum -> a/n + (1-a)*sum)
}
```

Key: url$_1$,
Value: rank$_1$/outlink$_1$.size)