Database Systems
CSE 414

Lecture 20-21: Spark

(Ch. 23.1-2)

Spark

- Open source system from Berkeley
- Distributed processing over HDFS
- Differences from MapReduce:
  - Multiple steps, including a fixed number of iterations
    - E.g., running Spark SQL
  - Stores intermediate results in main memory
  - Supports SQL
- Details: http://spark.apache.org/examples.html

Spark Interface

- Spark supports a Scala interface
- Scala = ext of Java with lambda functions/closures
  - will show Scala/Spark examples shortly…

- Spark also supports a SQL interface
- It compiles SQL into Scala
- For HW6: 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 re-compute the lost partition of the RDD

Programming in Spark

- A Spark/Scala program consists of:
  - Transformations (map, reduceByKey, join…). Lazy
    - Construct a new RDD from a previous one
    - Compute the new RDD at the first time it is used in an action
  - Actions (count, reduce, save...). Eager
    - Compute a result based on an RDD, and either return it to the driver program or save it to an external storage system

- 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”

```
lines = spark.textFile(“hdfs://logfile.log”);

errors = lines.filter(_.startsWith(“ERROR”));

sqlerrors = errors.filter(_.contains(“sqlite”));

sqlerrors.collect()
```

collect(): return all elements from the RDD. Should use only on a small data set that can fit in a single machine’s memory

Example

Given a large log file hdfs://logfile.log, retrieve all lines that:

- Start with “ERROR”
- Contain the string “sqlite”

```
lines = spark.textFile(“hdfs://logfile.log”);

errors = lines.filter(_.startsWith(“ERROR”));

sqlerrors = errors.filter(_.contains(“sqlite”));

sqlerrors.collect()
```

Transformation:
Not executed yet…

Action:
triggers execution of entire program

MapReduce Again…

Steps in Spark resemble MapReduce:

- rdd.filter(p) applies in parallel the predicate p to all elements x of the partitioned collection / RDD, and returns those x where p(x) = true
  - E.g., rdd = {1, 2, 3, 3}. rdd.filter(x => x != 1) has result {2, 3, 3}
- rdd.map(f) applies in parallel the function f to all elements x of the partitioned collection / RDD, and returns a new partitioned collection
  - E.g., rdd = {1, 2, 3, 3}. rdd.map(x => x + 1) has result {2, 3, 4, 4}

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 using one or more variables declared outside the function):

```
var x = 5;  rdd.filter(_ > x)
var s = “sqlite”;  rdd.filter(x => x.contains(s))
```

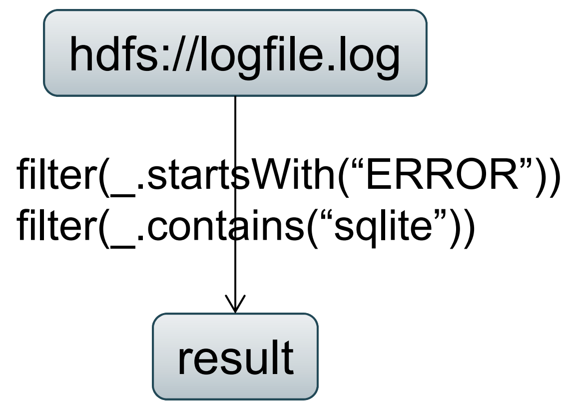
Persistence

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

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

Persistence

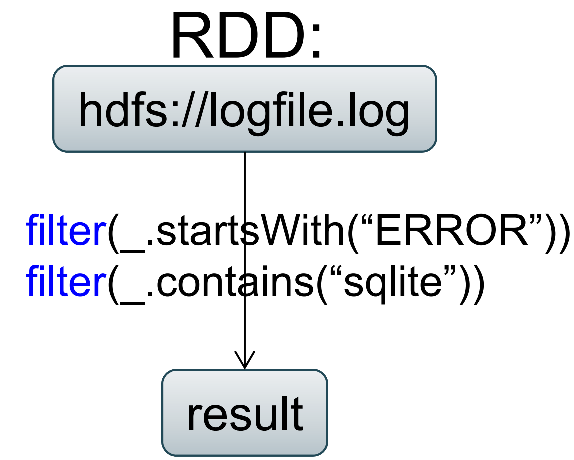
RDD:




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

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

Persistence




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

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

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

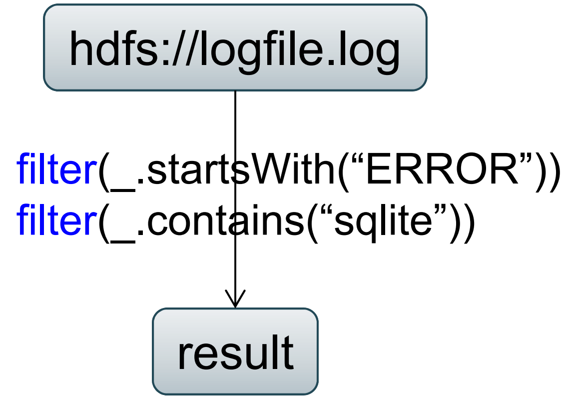
New RDD

Spark can re-compute the result from errors

By default, an RDD is re-computed each time an action is run on it. persist() can choose to store an RDD's content in memory or on disk, so the content can be reused in multiple actions.

Persistence

RDD:

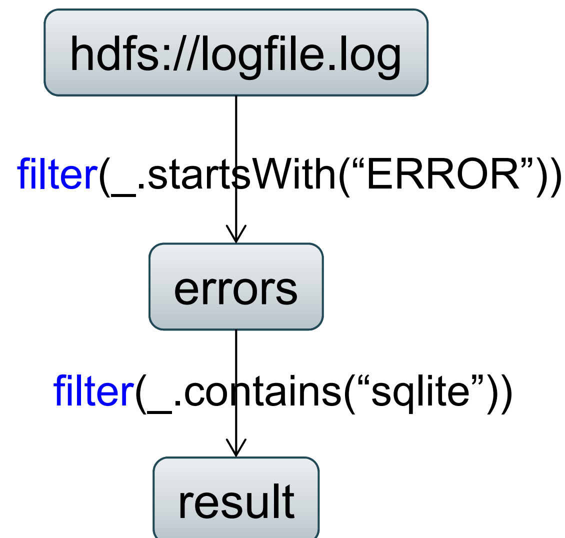



```
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```

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

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

New RDD



Spark can re-compute the result from errors

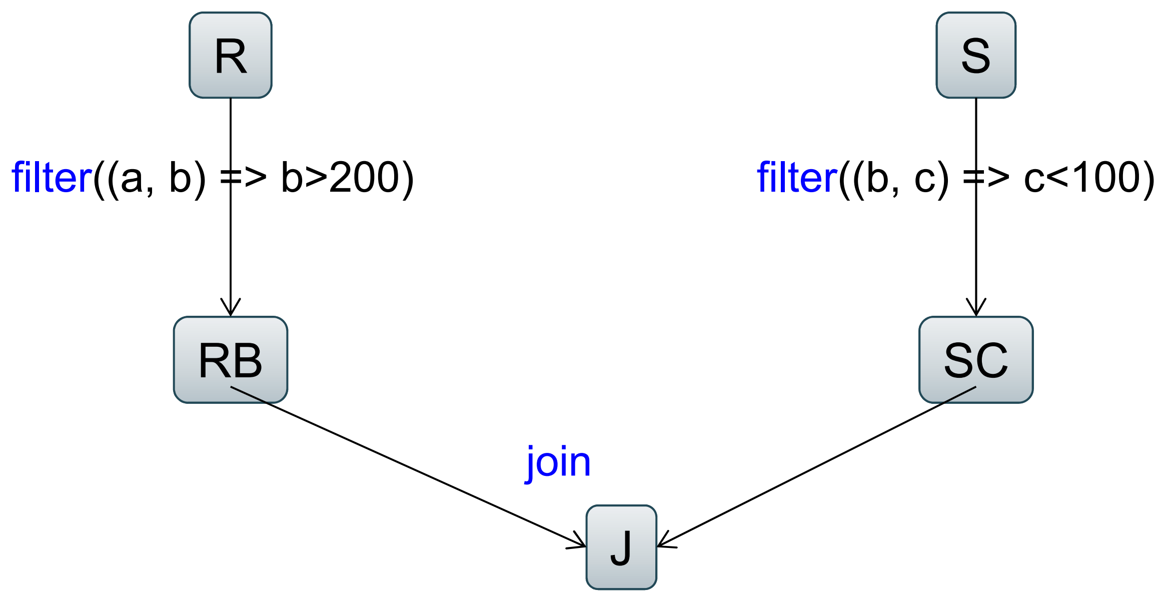
R(A, B)
S(A, C)

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

Example

```
R = spark.textFile("R.csv").map(parseRecord).persist()
S = spark.textFile("S.csv").map(parseRecord).persist()
RB = R.filter((a, b) => b > 200).persist()
SC = S.filter((a, c) => c < 100).persist()
J = RB.join(SC).persist()
J.count();
```

join(): inner join between two RDDs containing key/value pairs



Programming in Spark

- A Spark/Scala program consists of:
  - Transformations (map, reduceByKey, 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

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]))]
cartesian():	(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 like HDFS

Example Transformations

- flatMap()
  - Apply a function to each element in the RDD and return an RDD consisting of the elements from all of the iterators
  - E.g., rdd = {“a b”, “c d”}. rdd.flatMap(x => x.split(" ")) has result {“a”, “b”, “c”, “d”}

- union()
  - Produce an RDD containing elements from both RDDs
  - E.g., rdd1 = {1, 2}, rdd2 = {2, 3}, rdd1.union(rdd2) has result {1, 2, 2, 3}

- cartesian()
  - Cartesian product with the other RDD
  - rdd1.crossProduct(rdd2) has result {(1, 2), (1, 3), (2, 2), (2, 3)}

Example Transformations – Cont.

For RDDs containing key/value pairs

E.g., rdd = {(1, 2), (3, 4), (3, 6)}, rdd2 = {(3, 9)}

- groupByKey()
  - Group values with the same key
  - rdd.groupByKey() has result {(1, [2]), (3, [4, 6])}

- reduceByKey()
  - Combine values with the same key
  - rdd.reduceByKey((x, y) => x + y) has result {(1, 2), (3, 10)}

Example Transformations – Cont.

For RDDs containing key/value pairs

E.g., rdd = {(1, 2), (3, 4), (3, 6)}, rdd2 = {(3, 9)}

- mapValues()
  - Apply a function to each value of a key/value pair without changing the key
  - rdd.mapValues(x => x + 1) has result {(1, 3), (3, 5), (3, 7)}

- cogroup()
  - Group data from both RDDs sharing the same key
  - rdd.group(rdd2) has result {(1, ([2], [])), (3, ([4, 6], [9]))}

Example Actions

E.g., rdd = {1, 2, 3, 3}

- count()
  - Number of elements in the RDD
  - rdd.count() has result 4

- reduce()
  - Combine the elements of the RDD together in parallel
  - rdd.reduce((x, y) => x + y) has result 9

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
  - $\boldsymbol{\sigma}$ = filter
  - $\boldsymbol{\pi}$ = map
  - $\boldsymbol{\gamma}$ = groupByKey
  - $\times$ = cartesian
  - $\bowtie$ = join
- Spark SQL does small optimizations to RA
- Also chooses between broadcast and parallel joins

PageRank

- PageRank 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
- PageRank was introduced by Google, and essentially defined Google

Purpose of PageRank

- Compute *p*(*d*), the prior probability of the document *d* for retrieval purpose
- Not all Web pages are equally important
  - E.g., pages on popular Web sites tend to be more important
- Give weights to Web pages based on how often they are hyperlinked by other Web pages
  - Hyperlink = citation
  - More citations $\Rightarrow$ more important

Model behind PageRank: Random Walk

- Imagine a Web surfer doing a random walk on the Web
  - Start at a random page
  - At each step, go out of the current page along one of the links on the page
    - Each link is chosen with equal probability
- In the steady state, each page has a long-term visit rate
  - Called the page's PageRank
  - It does not matter where the surfer starts
- PageRank = long-term visit rate = steady state probability

Random Walk – Cont.

- A Markov chain consists of N states + an $N \times N$ transition probability matrix P
- state = page
- At each step, the Web surfer is on exactly one page, say page i
- For $1 \le i, j \le N$, the matrix entry P_{ij} is the probability of moving from page i to page j in the next step
- For every i, $\sum_{j=1}^{N} P_{ij} = 1$

Random Walk – Cont.

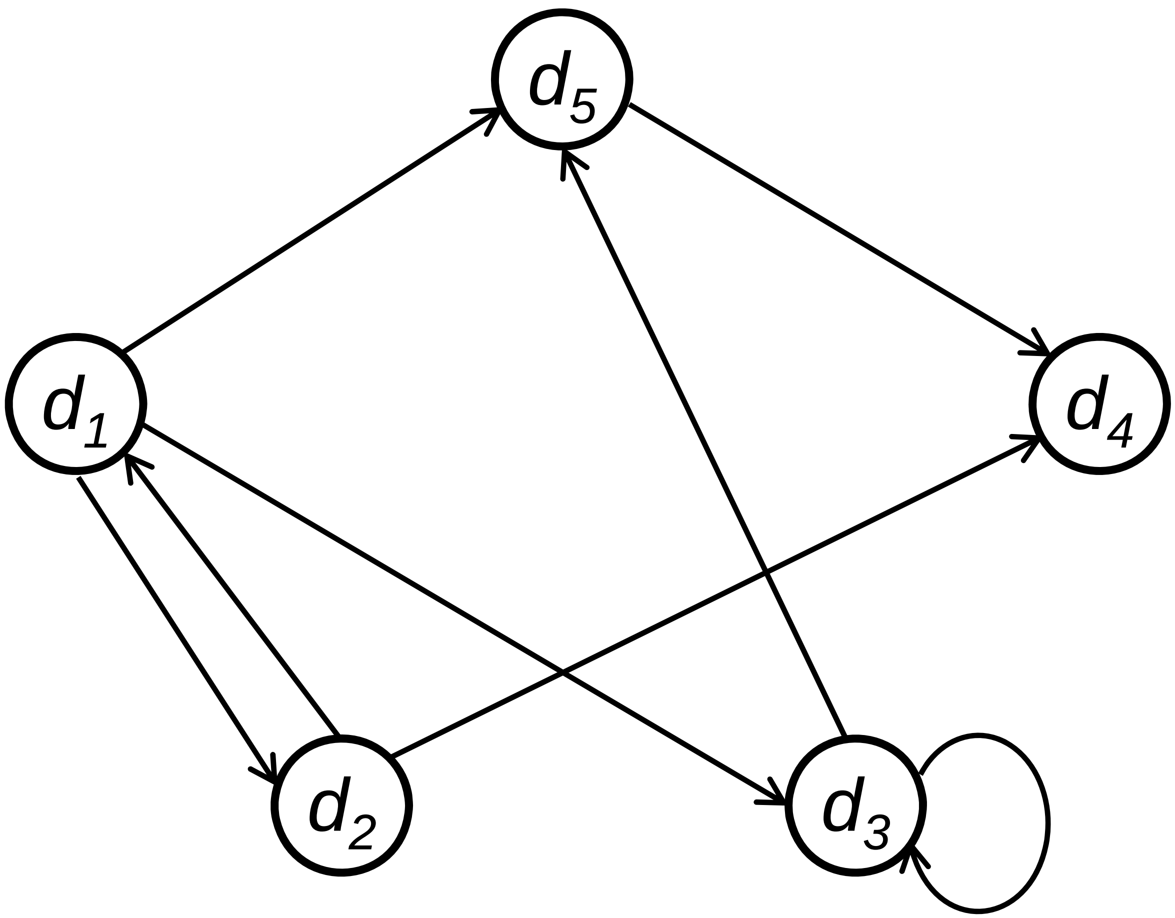
- Dead end: a Web page with no outgoing link
- *r*: the teleportation rate
  - A parameter whose value is between 0 and 1
  - Typical value: 0.15
- At a dead end (say page *i*), choose a random Web page with equal probability $1/N$ and jump to it
  - $P_{ij} = 1/N$ for every j

Random Walk – Cont.

- At a non-dead end (say page *i*),
  - With probability *r*, jump to a random web page
    - to each page with a probability of *r*/*N*
  - With the remaining probability 1-*r*, go out on a random hyperlink
    - C_i: the number of links going out of page *i*
    - Go out on each of the C_i links with a probability of (1-*r*)/C_i

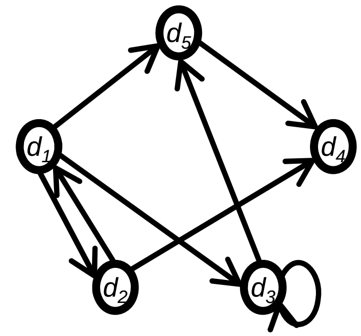
$$P_{ij} = \begin{cases} \frac{r}{N}, & \text{if there is no link going from page i to page j} \\ \frac{r}{N} + \frac{1-r}{C_i}, & \text{if there is a link going from page i to page j} \end{cases}$$

Example Web Graph



C_1=3 (d_2, d_3, d_5)
C_2=2 (d_1, d_4)
C_3=2 (d_3, d_5)
C_4=0 (dead end)
C_5=1 (d_4)

Transition Probability Matrix



	d_1	d_2	d_3	d_4	d_5
d_1	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{3}$	$\frac{r}{5}+\frac{1-r}{3}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{3}$
d_2	$\frac{r}{5}+\frac{1-r}{2}$	$\frac{r}{5}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{2}$	$\frac{r}{5}$
d_3	$\frac{r}{5}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{2}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{2}$
d_4	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$
d_5	$\frac{r}{5}$	$\frac{r}{5}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{1}$	$\frac{r}{5}$

C_1=3 (d_2, d_3, d_5)

C_2=2 (d_1, d_4)

C_3=2 (d_3, d_5)

C_4=0 (dead end)

C_5=1 (d_4)

Ergodicity Theorem

- Theorem in stochastic processes:

Web-graph+teleporting has a steady-state probability distribution

$\Rightarrow$ Each page in the Web-graph+teleporting has a PageRank

- Steady state probability vector $\Pi = (\pi_1, \pi_2, \ldots, \pi_N)$
  - π_i is the long-term visit rate (or PageRank) of page *i*

Probability Vector

- At a specific step, a probability (row) vector $X = (x_1, \ldots, x_N)$ tells us where the random walk is at
  - The random walk is on page i with probability x_i
  - $\sum_{i=1}^{N} x_i = 1$
- Example:
  - (0.1 0.2 0.3 0.15 0.25)

(0.1	0.2	0.3	0.15	0.25)
1	2	3	4	5

Change in Probability Vector

- If the probability vector in the current step is $X = (x_1, \ldots, x_N)$, the probability vector in the next step is XP
  - In the next step, the random walk is on page j with probability $\sum_{i=1}^{N} x_i \cdot P_{ij}$

$$(x_1 \quad \ldots \quad x_N)\begin{pmatrix} & P_{1j} & \\ \cdots & \vdots & \cdots \\ & P_{Nj} & \end{pmatrix}$$

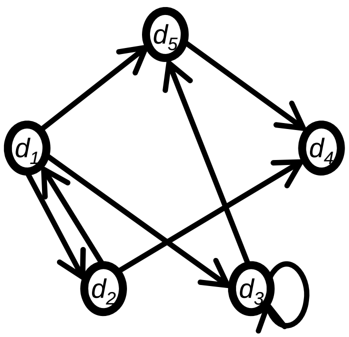
Compute the Steady State Probability Vector

- Suppose the distribution has reached the steady state $\Pi = (\pi_1, \pi_2, \ldots, \pi_N)$ in the current step
- The distribution in the next step is ΠP, which should also be in steady state
- So $\Pi = \Pi P$
- Solving this matrix equation gives us Π
  - Π is the principal left eigenvector for P
    - i.e., the left eigenvector with the largest eigenvalue

Example of $\Pi = \Pi P$

$$\pi_3 = \pi_1 \cdot \left(\frac{r}{5} + \frac{1-r}{3}\right) + \pi_2 \cdot \frac{r}{5} + \pi_3 \cdot \left(\frac{r}{5} + \frac{1-r}{2}\right) + \pi_4 \cdot \frac{1}{5} + \pi_5 \cdot \frac{r}{5}$$

	d_1	d_2	d_3	d_4	d_5
d_1	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{3}$	$\frac{r}{5}+\frac{1-r}{3}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{3}$
d_2	$\frac{r}{5}+\frac{1-r}{2}$	$\frac{r}{5}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{2}$	$\frac{r}{5}$
d_3	$\frac{r}{5}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{2}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{2}$
d_4	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$
d_5	$\frac{r}{5}$	$\frac{r}{5}$	$\frac{r}{5}$	$\frac{r}{5}+\frac{1-r}{1}$	$\frac{r}{5}$



Another Way of Writing $\Pi = \Pi P$

- Assume no dead end for now
- Suppose pages $T_1, \ldots, T_m$ have links to page A
- $C(T_j)$: the number of links going out of page T_j

$$PageRank(A) = \frac{r}{N} + (1-r)\left[\frac{PageRank(T_1)}{C(T_1)} + \cdots + \frac{PageRank(T_m)}{C(T_m)}\right]$$

One Way of Computing the PageRank Π

- Start with any distribution X
- E.g., uniform distribution
- After one step, we get XP
- After two steps, we get XP^2
- After k steps, we get XP^k
- Algorithm: multiply X by increasing powers of P until convergence
- This is called the power method

PageRank

```
for i = 1 to N:
   x[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] += x[i] / k
    for i = 1 to N: x[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

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

PageRank

```
for i = 1 to N:
   x[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] += x[i] / k
    for i = 1 to N: x[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.

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

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

where r $\in$(0,1) is the teleportation rate

links: RDD[url:string, links:SEQ[string]]
ranks: RDD[url:string, rank:float]

PageRank

```
for i = 1 to N:
   x[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] += x[i] / k
    for i = 1 to N: x[i] = r/N + (1-r)*contribs[i]
until convergence
/* usually 10-20 iterations */
```

```
// SPARK
val links = spark.textFile(..).map(..).persist()
var ranks = … // RDD of (URL, 1/n) pairs
for (k <- 1 to ITERATIONS) {
  // Build RDD of (targetURL, float) pairs
  // with contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url,  (links, rank)) =>
       links.map(dest => (dest, rank/links.size))
   }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x, y) => x+y)
       .mapValues(sum => a/n + (1-a)*sum)
}
```

Google Dataflow

- Similar to Spark/Scala
- Allows you to lazily build pipelines and then execute them

- Much simpler than multi-job MapReduce

Summary

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