15-319 / 15-619 Cloud Computing

Recitation 13

April 17th 2018

Overview

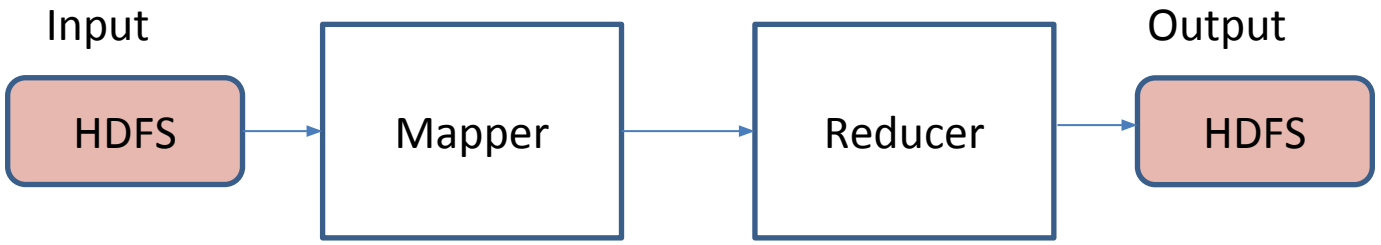
- **Last week's reflection**
  - Team Project Phase 2
  - Quiz 11
  - OLI Unit 5: Modules 21 & 22
- **This week's schedule**
  - Project 4.2
  - No more OLI modules and quizzes!
- **Twitter Analytics: The Team Project**
  - Phase 3

Project 4

- Project 4.1
  - Batch Processing with MapReduce
- Project 4.2
  - Iterative Batch Processing Using Apache Spark
- Project 4.3
  - Stream Processing with Kafka and Samza

Typical MapReduce Batch Job

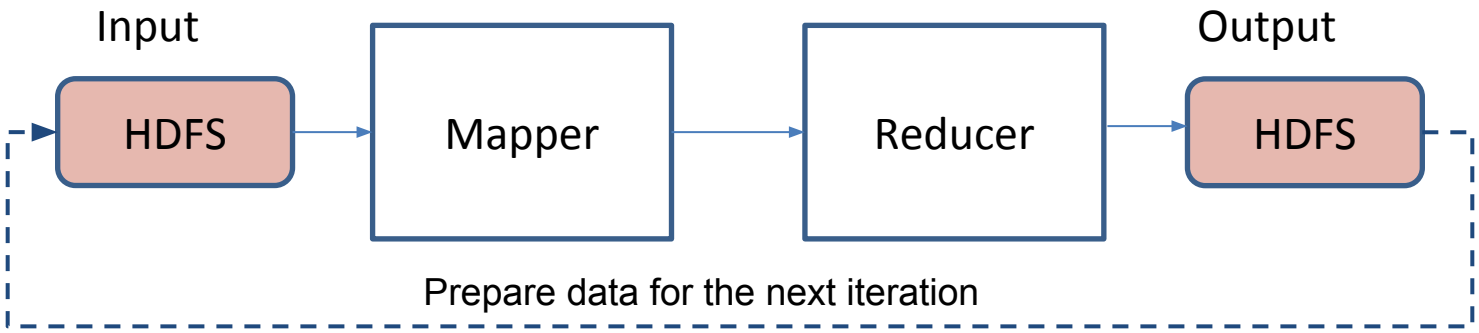
- Simplistic view of a MapReduce job



- You write code to implement the following classes
  - Mapper
  - Reducer
- Inputs are read from disk and outputs are written to disk
  - Intermediate data is spilled to local disk

Iterative MapReduce Jobs

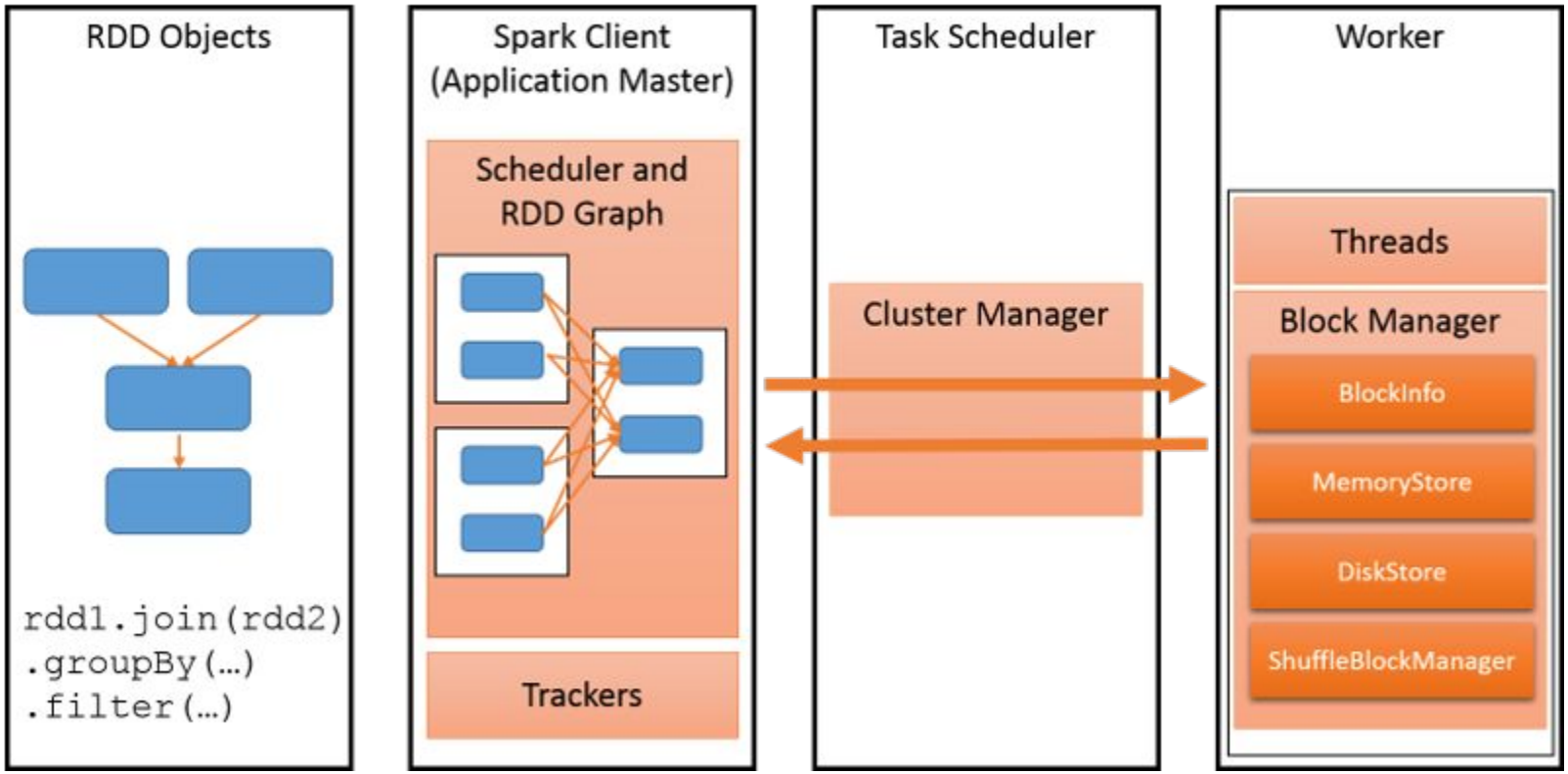
- Some applications require iterative processing
- Eg: Machine Learning



- MapReduce: Data is always **written** to disk
  - This leads to added overhead for each iteration
  - Can we keep data in memory? Across Iterations?
  - How do you manage this?

Apache Spark

- General-purpose cluster computing framework
- APIs in Python, Java, Scala and R
- Runs on Windows and UNIX-like systems



Spark Ecosystem

- Spark SQL
  - Process structured data
  - Run SQL-like queries against RDDs
- Spark Streaming
  - Ingest data from sources like Kafka
  - Process data with high level functions like map and reduce
  - Output data to live dashboards or databases
- MLlib
  - Machine learning algorithms such as regression
  - Utilities such as linear algebra and statistics
- GraphX
  - Graph-parallel framework
  - Support for graph algorithms and analysis

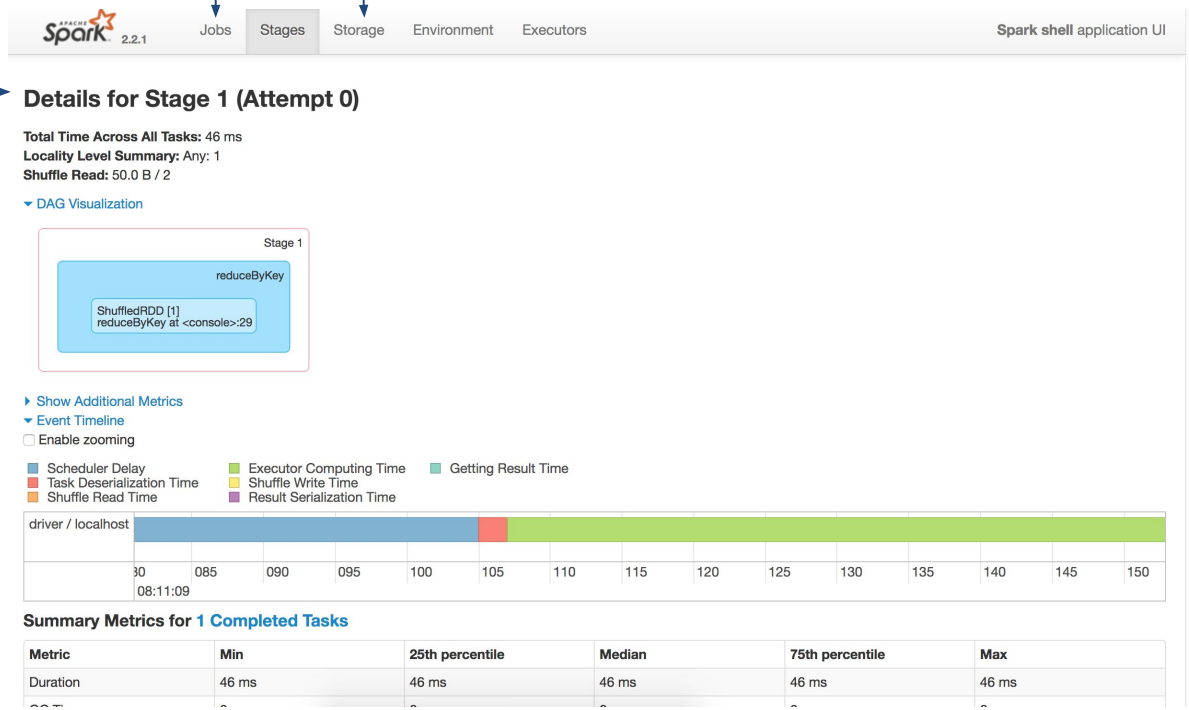
Spark Web UI

- Provides useful information on your Spark programs
- You can learn about resource utilization of your cluster
- Is a stepping stone to optimize your jobs

Status of RDD actions being computed

Info about cached RDDs and memory usage

In-depth job info



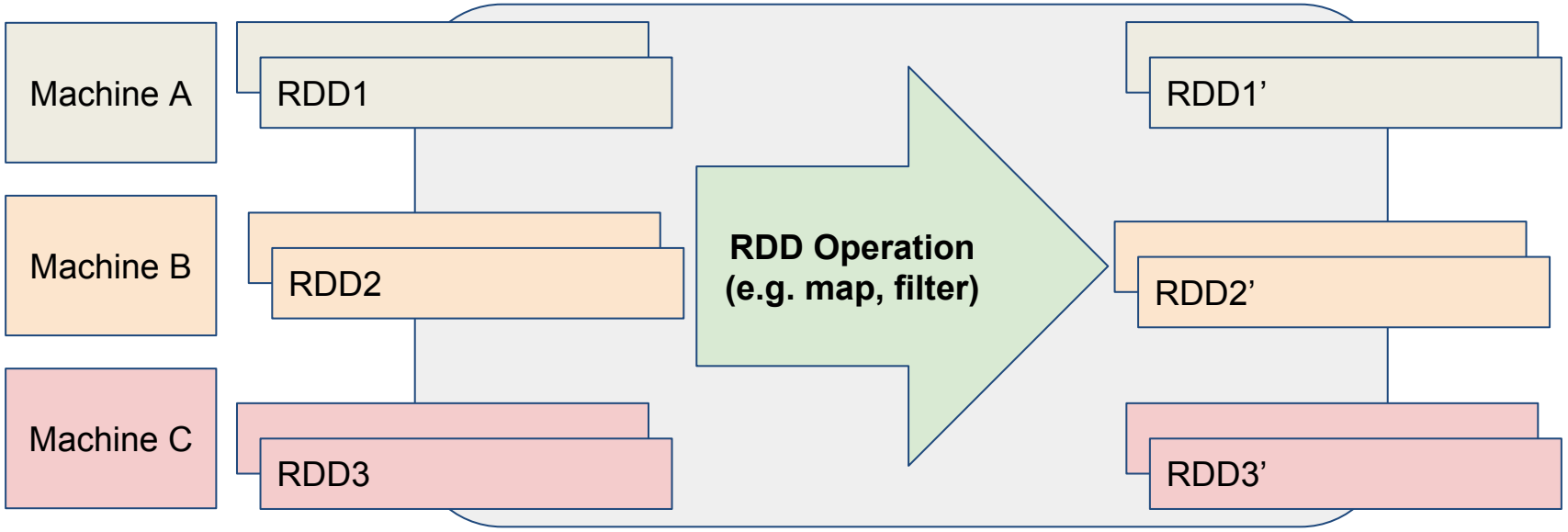
Apache Spark APIs

- There exists 3 sets of APIs for handling data in Spark

Resilient Distributed Datasets (RDD)	**DataFrame**	**DataSet**
• Distribute collection of JVM objects • Functional operators (map, filter, etc.)	• Distribute collection of Row objects • Expression-based operations • Fast, efficient internal representations	• Internally rows, externally JVM objects • Type safe and fast • Slower than dataframes

Resilient Distributed Datasets

- Focus of Project 4.2
- Can be in-memory or on disk
- Read-only objects
- Partitioned across the cluster based on a range or the hash of a key in each record



Operations on RDDs

- Loading data

```
>>> input_RDD = sc.textFile("text.file")
```

- Transformation
  - Applies an operation to derive a new RDD
  - Lazily evaluated -- may not be executed immediately

```
>>> transform_RDD =  input_RDD.filter(lambda x: "abcd" in x)
```

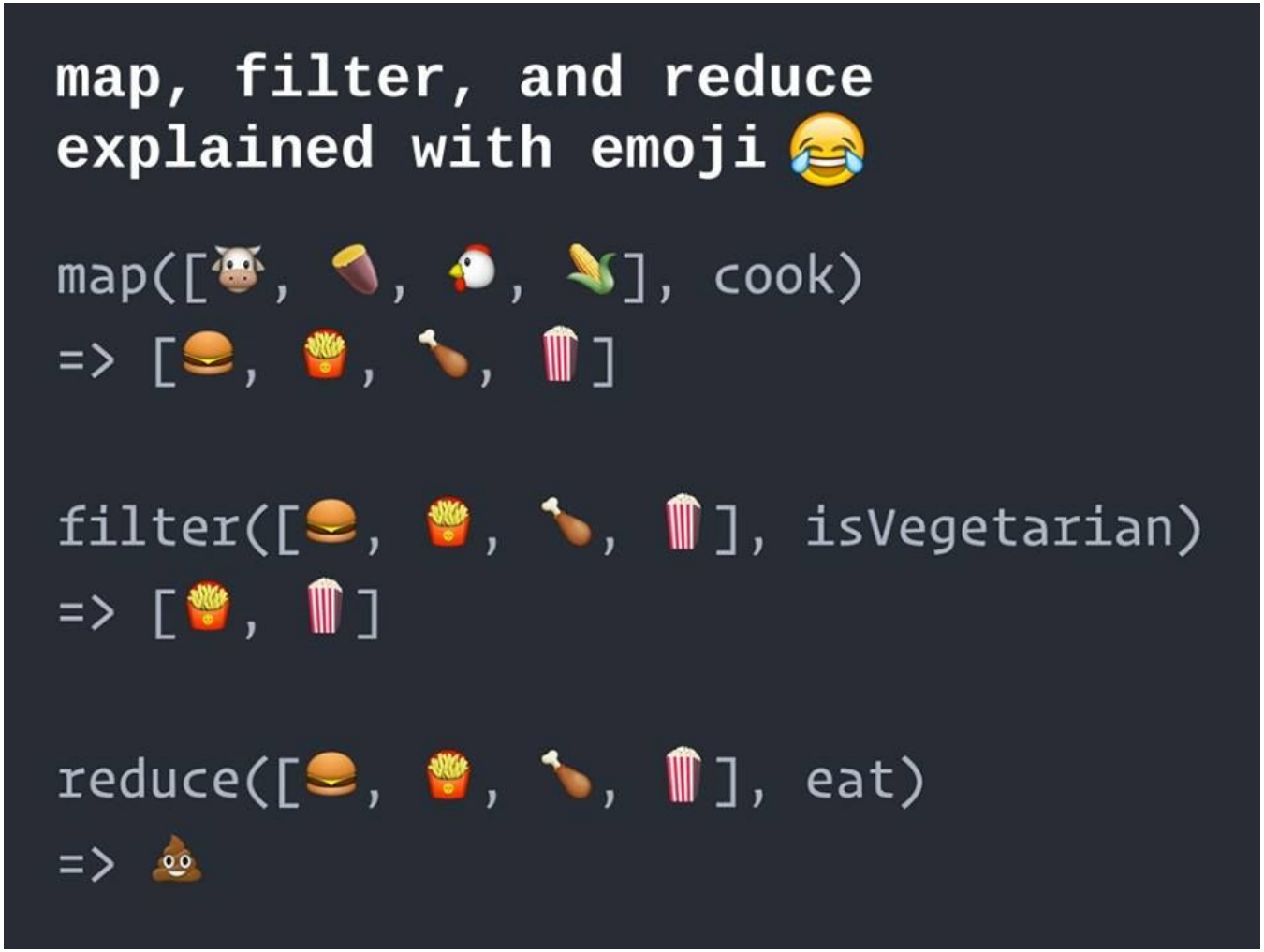
- Action
  - Forces the computation on an RDD
  - Returns a single object

```
>>> print "Number of “abcd”:" + transform_RDD.count()
```

- Saving data

```
>>> output.saveAsTextFile(“hdfs:///output”)
```

Operations on RDDs

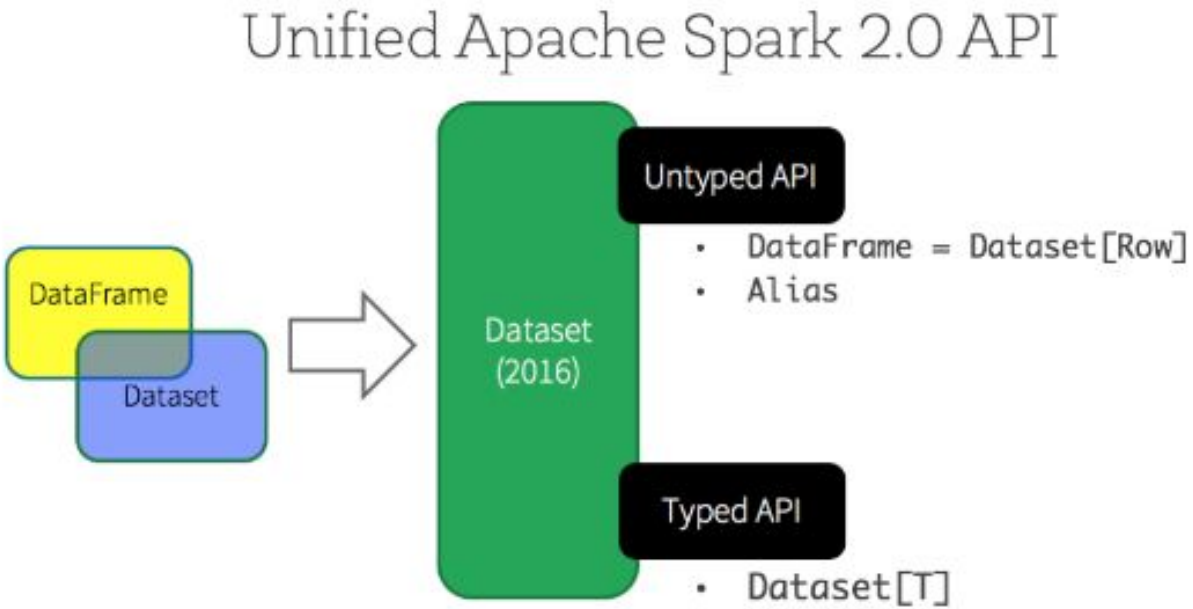


RDDs and Fault Tolerance

- Actions create new RDDs
- Uses the notion of lineage to support fault tolerance
  - Lineage is a log of transformations
  - Stores lineage on the driver node
  - Upon node failure, Spark loads data from disk to recompute the entire sequence of operations based on lineage

DataFrames

- A DataFrame is an immutable distributed collection of data
- Organized into named columns, like a table in a relational database
- A DataFrame is represented as a DataSet of rows



Operations on DataFrames

- Suppose we have a file `people.json`

```
{"name":"Michael"} {"name":"Andy", "age":30} {"name":"Justin", "age":19}
```

- Create a DataFrame with its contents

```
val df = spark.read.json("people.json")
```

- Run SQL-like queries against the data

```
val sqlDF = df.filter($"age" > 20).show()
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
```

- Save data to file

```
df.filter($"age" > 20).select(“name”).write.format(“parquet”).save(“output”)
```

Note: Parquet is a column-based storage format for Hadoop. You will need special dependencies to read this file

Project 4.2

Task	Points	Description	Language
Tutorial	15	Reimplement the data filtering portion of Project 1.2 in Apache Spark	Scala
1	10	Get familiarized with the Spark shell and running Spark programs by doing some data analysis on a Twitter graph dataset	Python, Scala or Java
2	40	Calculate the influence of users in the Twitter graph dataset with the pagerank algorithm	Python, Scala or Java
Bonus	10	Optimize the pagerank algorithm in task 2	Probably Scala
Reflection and discussion	5		-
Optional, ungraded task	0	Use the Apache Spark GraphX API to implement a second degree centrality algorithm on the graph	Scala

Tutorial Task

- Data filtering on a month's worth of Wikipedia data
- Similar to Project 1.2
- Output page titles and the number of views
- Starter code in Scala is provided
- Use `map`, `filter`
- Use `reduceByKey` and `aggregateByKey` over `groupByKey`

Twitter Social Graph

- Dataset that we will be using in Project 4.2
- ~8GB of data
- Edge list of (follower, followee) pairs
- Directed edges
  - (u, v) and (v, u) are two distinct edges

Task 1

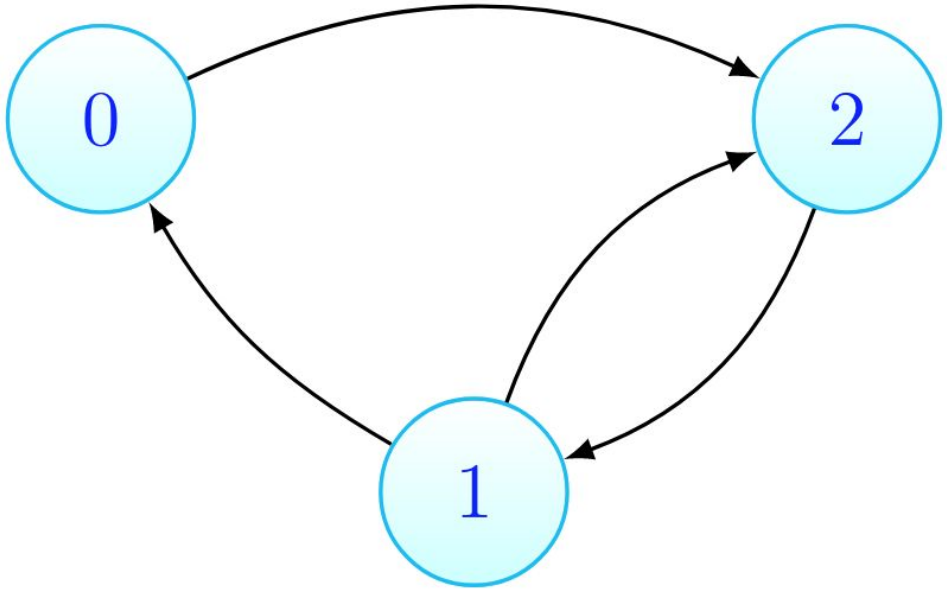
- Four parts to Task 1
  - Find the number of edges
  - Find the number of vertices
  - Find the top 100 users with the RDD API
  - Find the top 100 users with the SparkSQL API

Task 2: PageRank

- Started as an algorithm to rank websites in search engine results
- Assign ranks based on the number of links pointing to them
- A page that has links from
  - Many nodes ⇒ high rank
  - A high-ranking node ⇒ high rank
- In Task 2, we will implement pagerank to find the top 100 influential vertices in the Twitter social graph

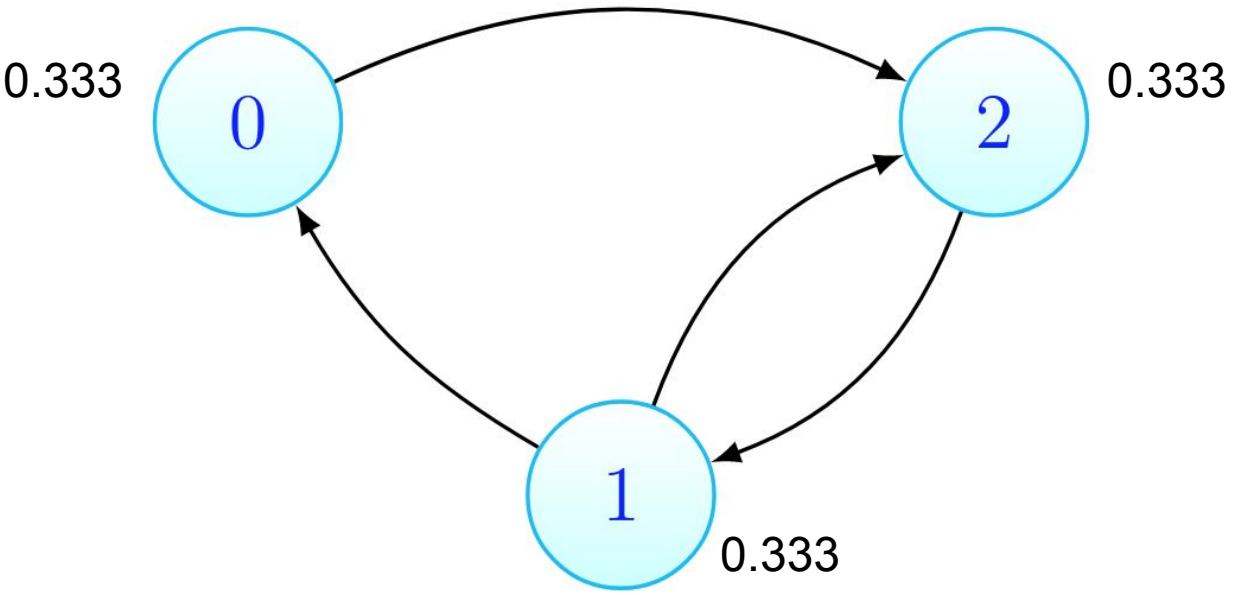
Basic PageRank

- How do we measure influence?
  - Intuitively, it should be the node with the most followers



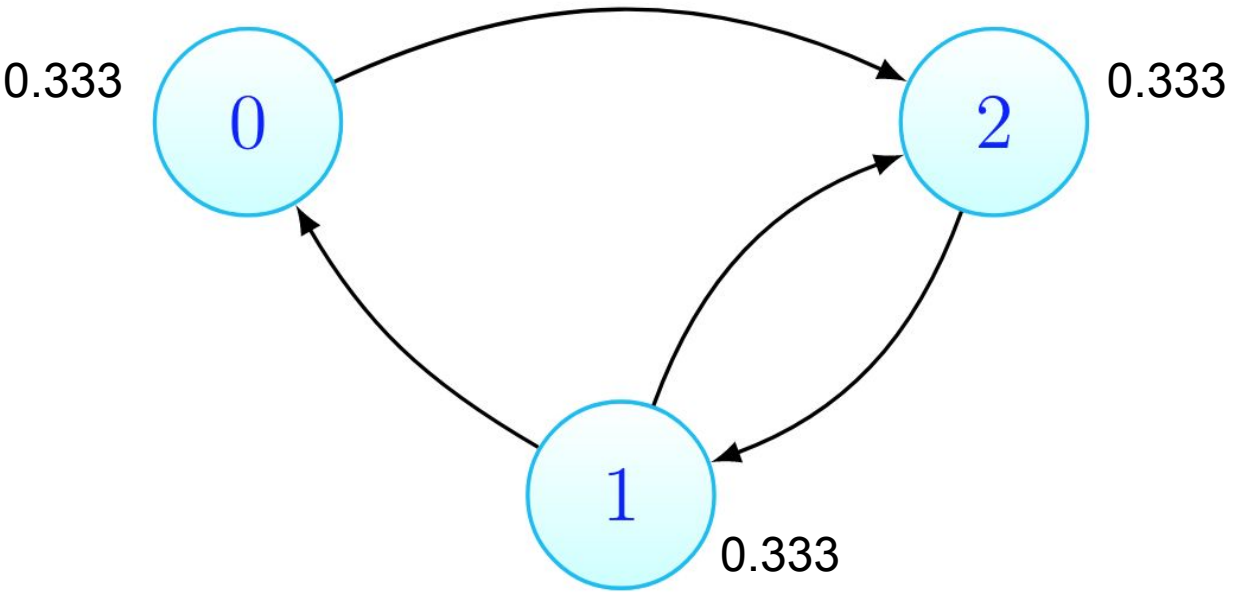
Basic PageRank

- Influence scores are initialized to 1.0/number of vertices



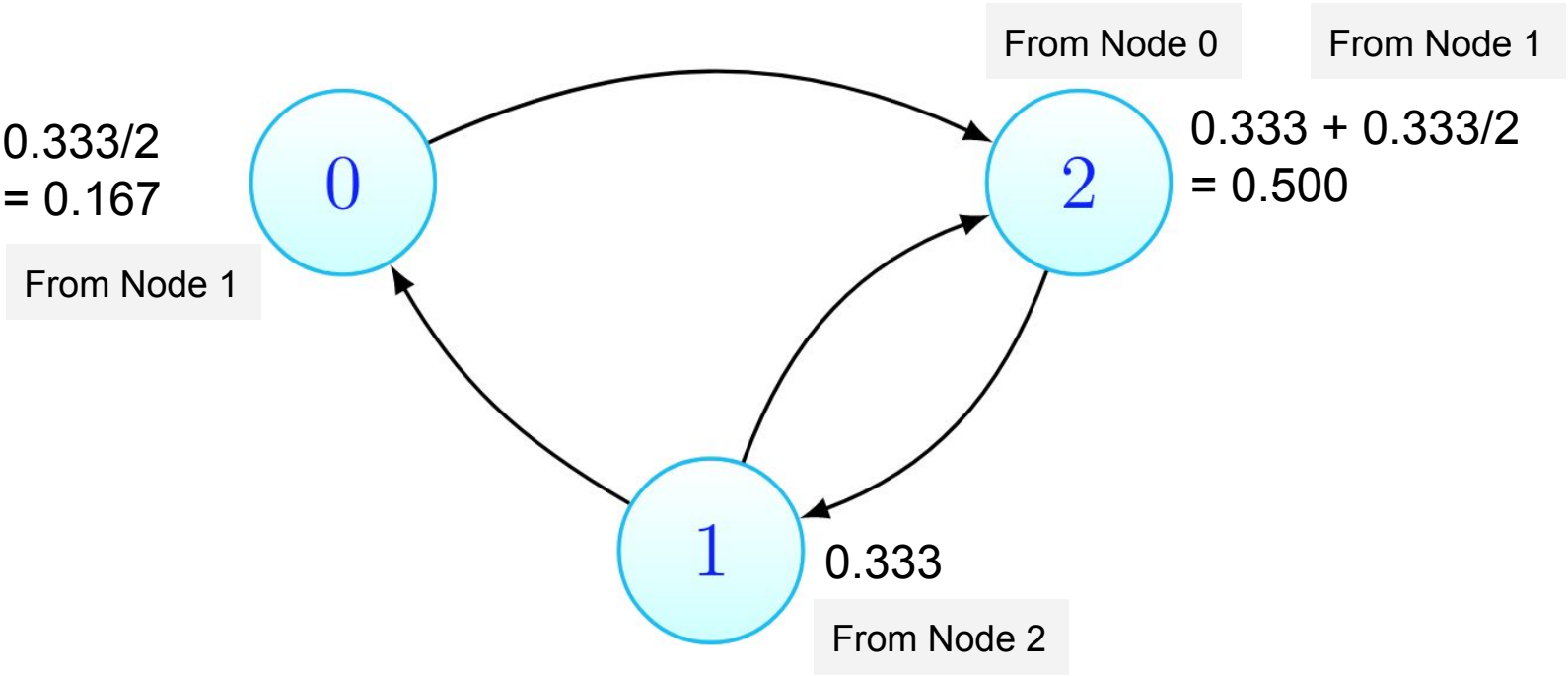
Basic PageRank

- Influence scores are initialized to 1.0/number of vertices
- In each iteration of the algorithm, scores of each user are redistributed between the users they are following



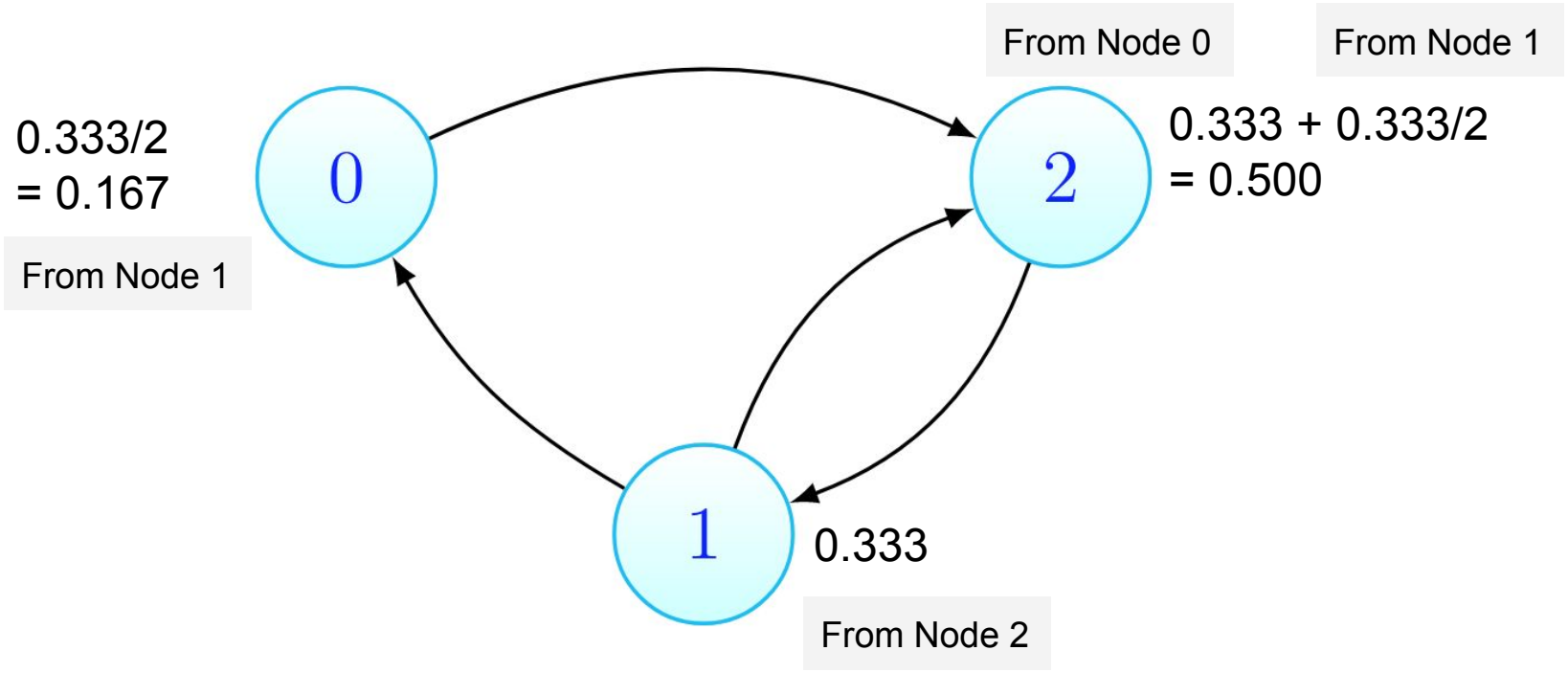
Basic PageRank

- Influence scores are initialized to 1.0/number of vertices
- In each iteration of the algorithm, scores of each user are redistributed between the users they are following



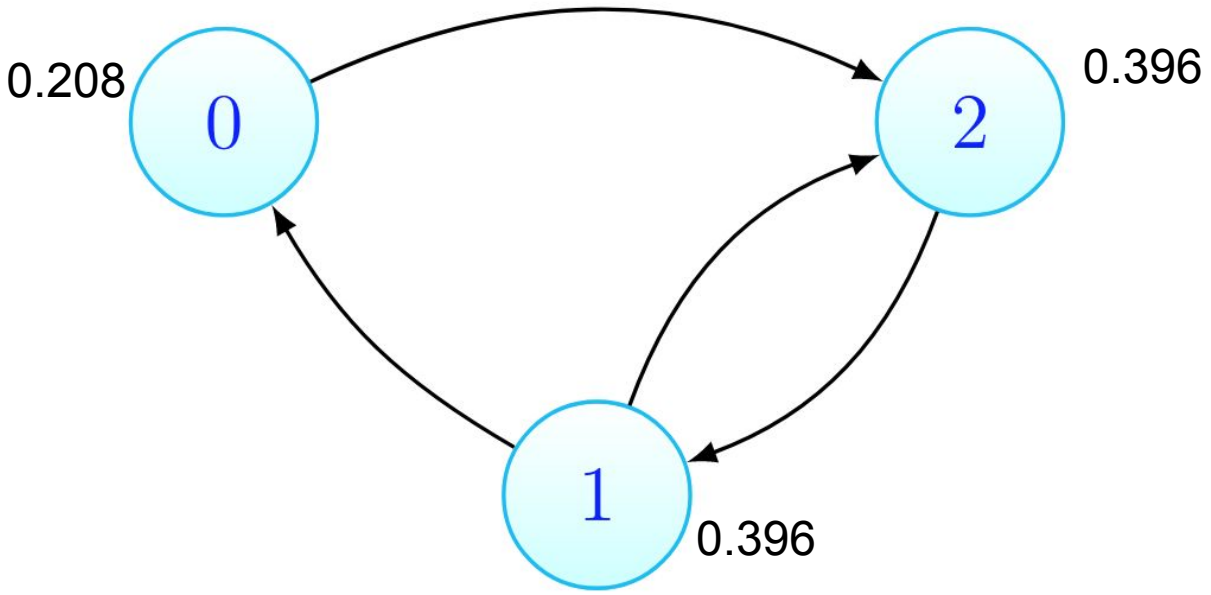
Basic PageRank

- Influence scores are initialized to 1.0/number of vertices
- In each iteration of the algorithm, scores of each user are redistributed between the users they are following
- Convergence is achieved when the scores of nodes do not change between iterations
- Pagerank is guaranteed to converge



Basic PageRank

- Influence scores are initialized to 1.0/number of vertices
- In each iteration of the algorithm, scores of each user are redistributed between the users they are following
- Convergence is achieved when the scores of nodes do not change between iterations
- Pagerank is guaranteed to converge



Basic PageRank Pseudocode

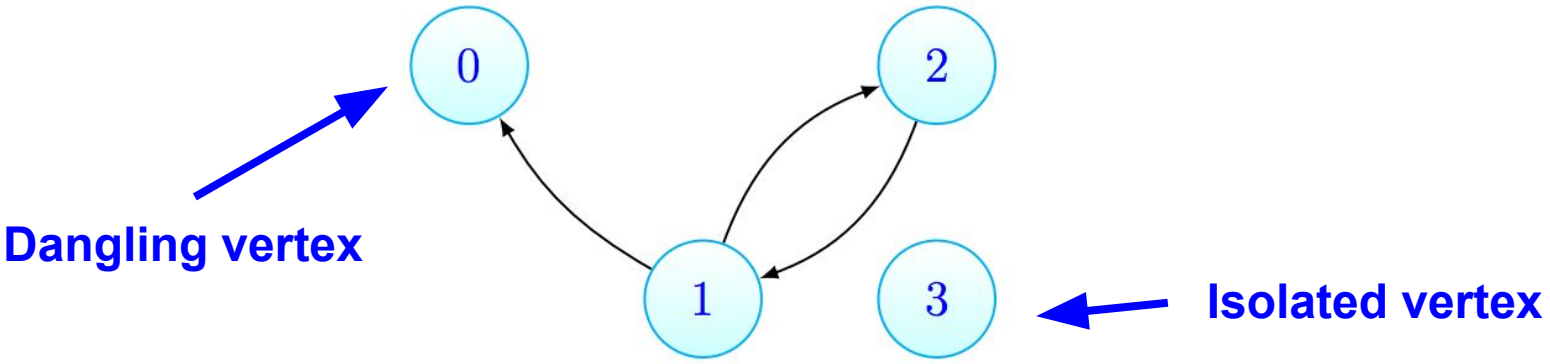
(Note: This does not meet the requirements of Task 2)

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS)
{
    // Build an RDD of (targetURL, float) pairs
    // with the 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)
}
```

PageRank Terminology

- Dangling or sink vertex
  - No outgoing edges
  - Redistribute contribution equally among all vertices
- Isolated vertex
  - No incoming and outgoing edges
  - No isolated nodes in Project 4.2 dataset
- Damping factor *d*
  - Represents the probability that a user clicking on links will continue clicking on them, traveling down an edge
  - Use d = 0.85



Visualizing Transitions

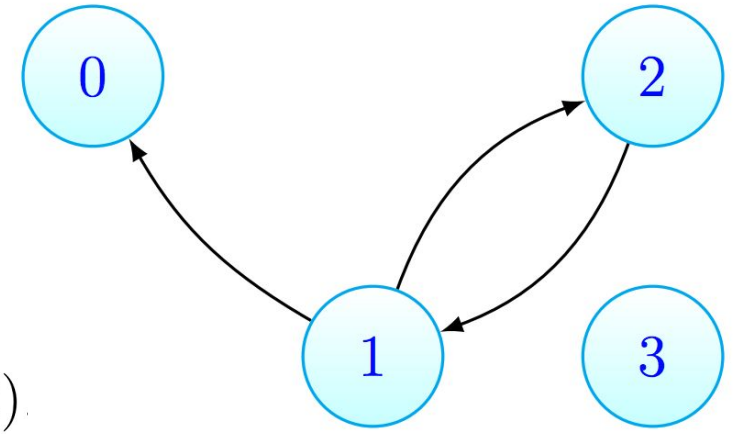
- Adjacency matrix:

$$\mathbf{G} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

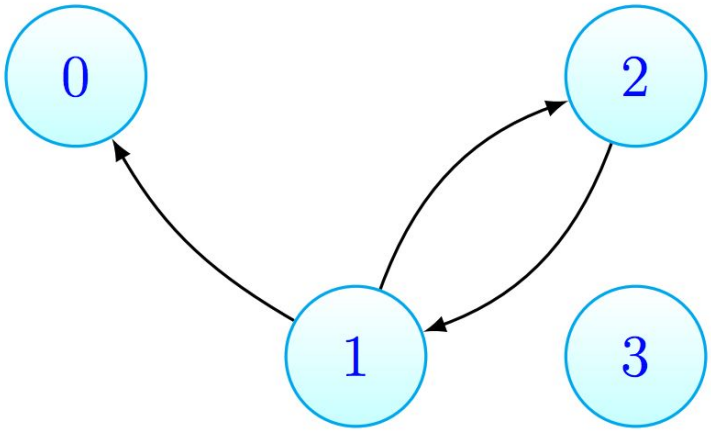
- Transition matrix: (rows sum to 1)

$$\mathbf{M} = \begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \\ 0.25 & 0.25 & 0.25 & 0.25 \end{bmatrix}$$

$$M_{ij} = \frac{G_{ij}}{\sum_{k=1}^{n} G_{ik}} \left(\text{ when } \sum_{k=1}^{n} G_{ik} \neq 0 \right)$$



Task 2: PageRank



Formula for calculating rank

$$r_i^{(k+1)} = d \sum_{v_j \in \mathcal{N}(v_i)} r_j^{(k)} M_{ji} + (1-d) r_i^{(0)}$$

d = 0.85

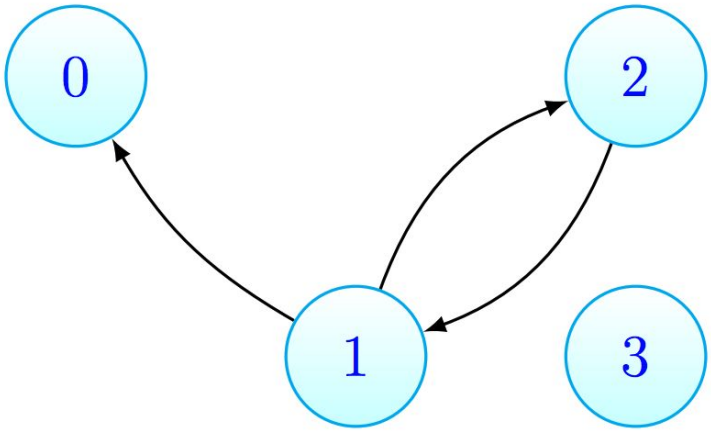
$$r_0^{(1)} = d(\frac{r_1^{(0)}}{2} + \frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}) + (1-d)\frac{1}{n}$$

$$r_1^{(1)} = d(\frac{r_2^{(0)}}{1} + \frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}) + (1-d)\frac{1}{n}$$

$$r_2^{(1)} = d(\frac{r_1^{(0)}}{2} + \frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}) + (1-d)\frac{1}{n}$$

$$r_3^{(1)} = d(\frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}) + (1-d)\frac{1}{n}$$

Task 2: PageRank



Formula for calculating rank

$$r_i^{(k+1)} = d \sum_{v_j \in \mathcal{N}(v_i)} r_j^{(k)} M_{ji} + (1-d) r_i^{(0)}$$

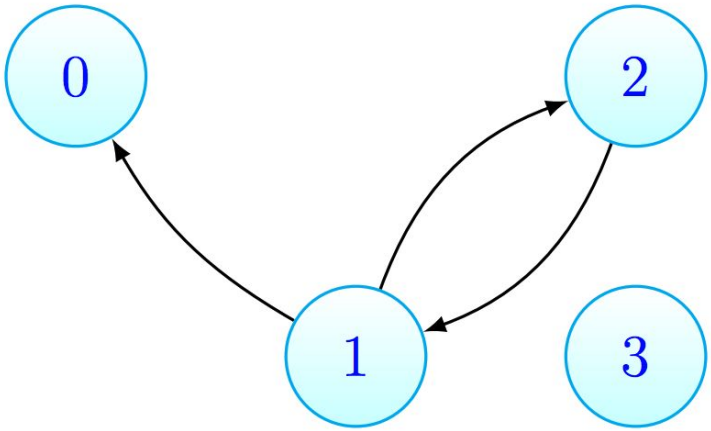
d = 0.85

Note: contributions from isolated and dangling vertices are constant in an iteration

Let

$$\epsilon = d\left(\frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}\right)$$

Task 2: PageRank



Formula for calculating rank

$$r_i^{(k+1)} = d \sum_{v_j \in \mathcal{N}(v_i)} r_j^{(k)} M_{ji} + (1-d) r_i^{(0)}$$

d = 0.85

Note: contributions from isolated and dangling vertices are constant in an iteration

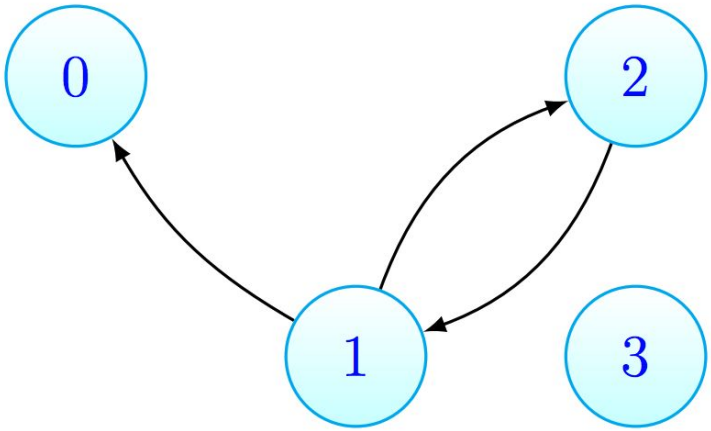
This simplifies the formula to

$$r_0^{(1)} = d\frac{r_1^{(0)}}{2} + \epsilon + (1-d)\frac{1}{n}$$
$$r_1^{(1)} = d\frac{r_2^{(0)}}{1} + \epsilon + (1-d)\frac{1}{n}$$
$$r_2^{(1)} = d\frac{r_1^{(0)}}{2} + \epsilon + (1-d)\frac{1}{n}$$
$$r_3^{(1)} = \epsilon + (1-d)\frac{1}{n}$$

Let

$$\epsilon = d\left(\frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}\right)$$

Task 2: PageRank



Formula for calculating rank

$$r_i^{(k+1)} = d \sum_{v_j \in \mathcal{N}(v_i)} r_j^{(k)} M_{ji} + (1-d) r_i^{(0)}$$

d = 0.85

$$\epsilon = 0.85 \times (0.25/4 + 0.25/4) = 0.106$$

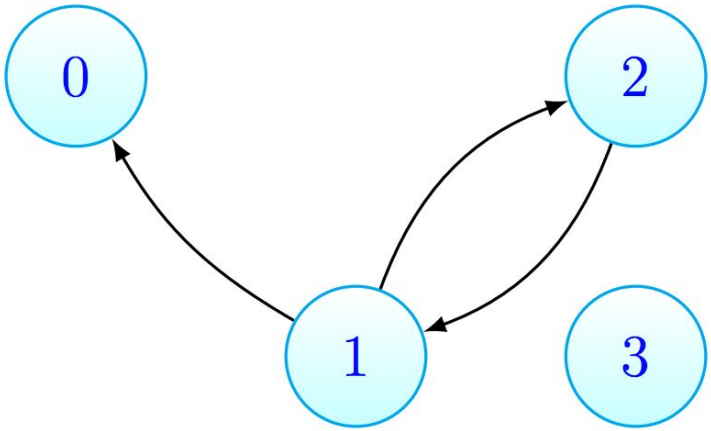
$$r_0^{(1)} = 0.85 \times 0.25/2 + 0.106 + 0.15 \times 0.25 = 0.25$$

$$r_1^{(1)} = 0.85 \times 0.25 + 0.106 + 0.15 \times 0.25 = 0.356$$

$$r_2^{(1)} = 0.85 \times 0.25/2 + 0.106 + 0.15 \times 0.25 = 0.25$$

$$r_3^{(1)} = 0.106 + 0.15 \times 0.25 = 0.144$$

Task 2: PageRank



Formula for calculating rank

$$r_i^{(k+1)} = d \sum_{v_j \in \mathcal{N}(v_i)} r_j^{(k)} M_{ji} + (1-d) r_i^{(0)}$$

d = 0.85

$$r_0^{(k)} = 0.2656$$

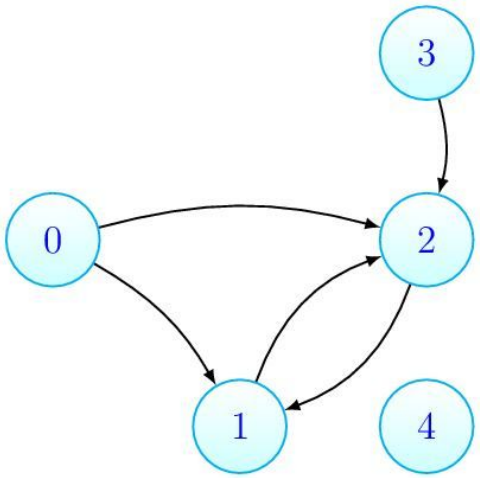
$$r_1^{(k)} = 0.3487$$

$$r_2^{(k)} = 0.2656$$

$$r_3^{(k)} = 0.1199$$

Optional Non-Graded Task

- Implement a second-degree centrality algorithm with Apache Spark's GraphX API
- PageRank score is a type of centrality score
  - Importance of a node in a graph.
  - Score for Node 0 and Node 4 is the same: 0.0306
  - Does not make sense
- The 2nd degree centrality algorithm calculates a node's importance as the average of your followees and followee's followees



General Hints

- Consider the implementation differences between `reduceByKey`, `groupByKey` and `aggregateByKey`
- Test out commands in the Spark Shell
  - REPL for Spark
  - Scala: `./spark/bin/spark-shell`
  - Python: `./spark/bin/pyspark`
- Test locally on small datasets
  - Spark can run on your local machine
  - EMR r3 clusters are expensive
  - Each task can take 45 min - 1 hour to run on EMR
- When in doubt, read the docs!
  - SparkSQL
  - RDD
- Don't forget to include in your submission
  - Updated references file
  - Shell commands used in Task 1
- Arguably the hardest P4 project. Start early!

Pagerank Hints

- Ensuring correctness
  - Make sure total scores sum to 1.0 in every iteration
  - Understand closures in Spark
    - Do not do something like this

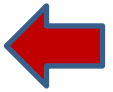
```
val data = Array(1,2,3,4,5)
var counter = 0
var rdd = sc.parallelize(data)
rdd.foreach(x => counter += x)
println("Counter value: " + counter)
```

  - Graph representation
    - Adjacency lists use less memory than matrices
  - More detailed walkthroughs and sample calculations can be found here

Pagerank Hints

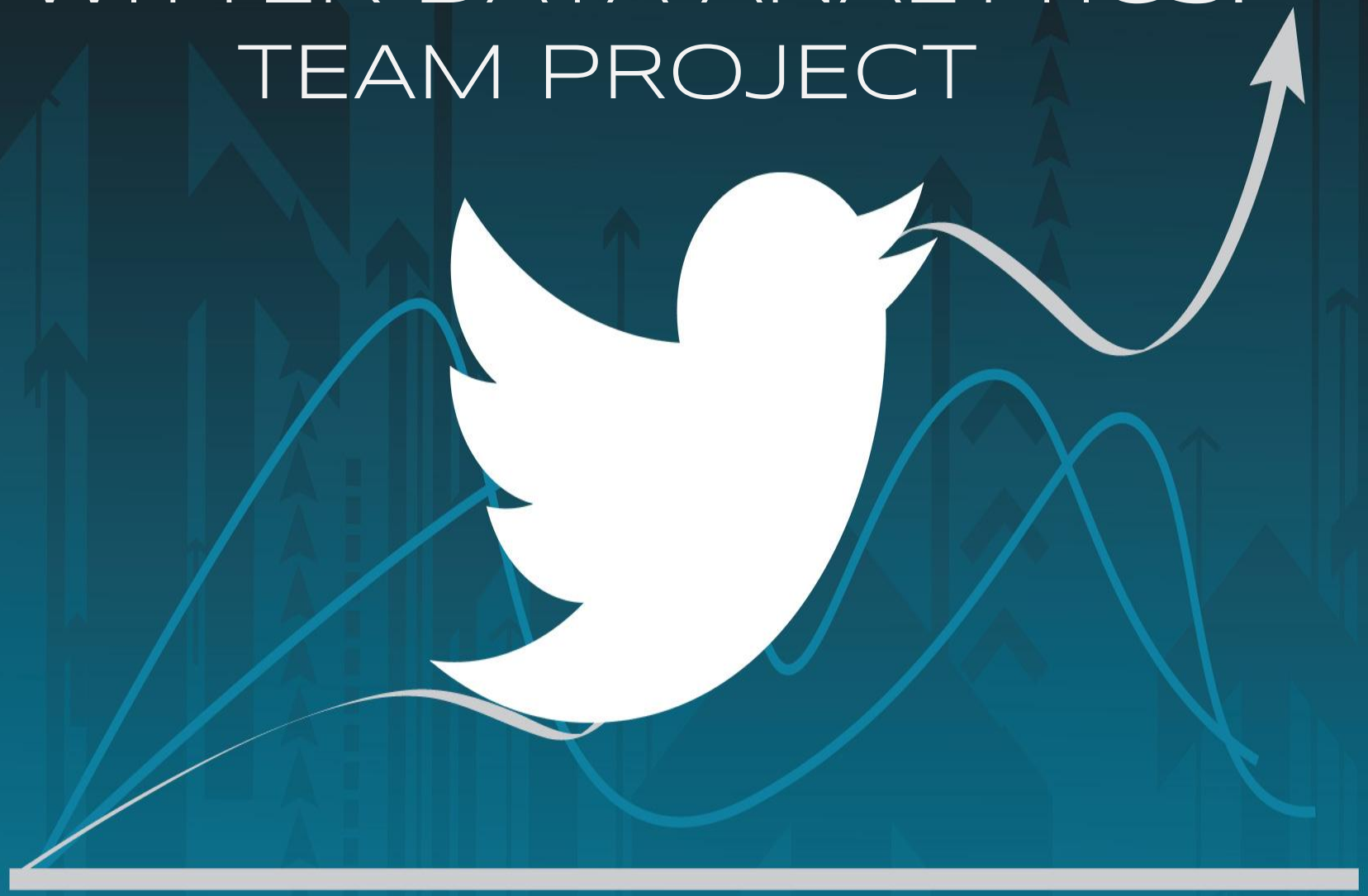
- Optimization
  - Eliminate repeated calculations
  - Use the Spark Web UI
    - Monitor your instances to make sure they are fully utilized
    - Identify bottlenecks
  - Understand RDD manipulations
    - Actions vs transformations
    - Lazy transformations
  - Explore parameter tuning to optimize resource usage
  - Be careful with repartition on your RDDs

Upcoming Deadlines

- Team Project : Phase 2
  - Code and report due: 04/17/2018 11:59 PM Pittsburgh 
- Project 4.2 : Iterative Programming with Spark
  - Due: 04/22/2018 11:59 PM Pittsburgh 
- Team Project : Phase 3
  - Live-test due: 04/29/2018 3:59 PM Pittsburgh
  - Code and report due: 05/01/2018 11:59 PM Pittsburgh

Questions?

TWITTER DATA ANALYTICS: TEAM PROJECT



Team Project Phase 2 MySQL Live Test Honor Board

Hong Kong Journalists
monoid
wen as dog
KingEdward
FatTiger
Faster Hong Kong Journalists
Cerulean
TaZoRo
liulaliula
30 centimeter

Team Project Phase 2 HBase Live Test Honor Board

Faster Hong Kong Journalists
liulaliula
30 centimeter
HLW
Hong Kong Journalists
DDLKiller
We Bare Bears
TakemezhuangbTakemefly
return no_sleep
Pyrocumulus

Team Project Phase 2 Live Test Honor Board

Faster Hong Kong Journalists	80
liulaliula	80
We Bare Bears	80
HLW	78
wen as dog	76
Hong Kong Journalists	76
TakemezhuangbTakemefly	75
monoid	74
team6412	74
Pyrocumulus	74

Query 4: Interactive Tweet Server

You are going to build a web service that supports READ, WRITE, SET and DELETE requests on tweets.

General Info:

1. Four operations:
   - write, set, read and delete
2. Operations under the same **uuid** should be executed in the order of the sequence number
3. Be wary of malformed queries
4. Only English tweets are needed

Query 4: Interactive Tweet Server

field	type	example
tweetid	long int	15213
userid	long int	156190000001
username	string	CloudComputing
timestamp	string	Mon Feb 15 19:19:57 2017
text	string	Welcome to P4!#CC15619#P3
favorite_count	int	22
retweet_count	int	33

Query 4: Interactive Tweet Server

- Write Request

  `/q4?op=write&payload=json_string&uuid=unique_id&seq=sequence_number`

- Response

  `TEAMID,TEAM_AWS_ACCOUNT_ID\n`

  `success\n`

- `payload`
  - It is a url-encoded json string;
  - It has the same structure as the original tweet json;
  - It only contains the seven fields needed. For tid and uid,
  - Don’t get them from the “id_str” field, only get them from the “id” field.

Query 4: Interactive Tweet Server

- Read Request

  `/q4?op=read&uid1=userid_1&uid2=userid_2&n=max_number_of_tweets&uuid=unique_id&seq=sequence_number`

- Response

```
TEAMID,TEAM_AWS_ACCOUNT_ID\n
tid_n\ttimestamp_n\tuid_n\tusername_n\ttext_n\tfavorite_count_n\tretweet_count_n\n
```

Query 4: Tweet Server

- Delete Request

  `/q4?op=delete&tid=tweet_id&uuid=unique_id&seq=sequence_number`

- Response

  `TEAMID,TEAM_AWS_ACCOUNT_ID\n`

  `success\n`

- Delete the whole tweet

Query 4: Tweet Server

- Set Request

  `/q4?op=set&field=field_to_set&tid=tweet_id&payload=string&uuid=unique_id&seq=sequence_number`

- Response

  `TEAMID,TEAM_AWS_ACCOUNT_ID\n`

  `success\n`

- Set one of the text, favorite_count, retweet_count of a particular tweet
- Payload is a url-encoded string

Query 4: Tweet Server

- Malformed Request

  `/q4?op=set&field=field_to_set&tid1=tweet_id&tid2=<empty>&payload=0;drop+tables+littlebobby&uuid=unique_id&seq=sequence_number`

- Response

  `TEAMID,TEAM_AWS_ACCOUNT_ID\n`

  `success\n`

Phase 2 Live Test Issues

- Cache became too large
- Used up IOPS
- Forgot to catch exceptions

Team Project General Hints

- Identify the bottlenecks using fine-grained profiling.
- Do not cache naively.
- Use logging to debug concurrent issues
- Review what we have learned in previous project modules
  - Load balancing (are requests balanced?)
  - Replication and sharding
  - Multi-threading programming
- Look at the feedback of your Phase 1 and Phase 2 reports!
- To test mixed queries, run your own load generator.
  - Use jmeter/ab/etc.

Team Project, Q4 Hints

- Start with one machine if you are not sure that your concurrency model is correct.
- Adopt a forwarding mechanism or a non-forwarding mechanism
  - You may need a custom load balancer
- Think carefully about your async/sync design
- May need many connections and threads at the same time, in the case of out of order sequence numbers.

Team Project Time Table

Phase (and query due)	Start	Deadlines	Code and Report Due
Phase 1 • Q1, Q2	Monday 02/26/2018 00:00:00 ET	**Checkpoint 1, Report: Sunday 03/11/2018 23:59:59 ET** **Checkpoint 2, Q1: Sunday 03/25/2018 23:59:59 ET** **Phase 1, Q2: Sunday 04/01/2018 23:59:59 ET**	Phase 1: Tuesday 04/03/2018 23:59:59 ET
Phase 2 • Q1, Q2,Q3	Monday 04/02/2018 00:00:00 ET	**Sunday 04/15/2018 15:59:59 ET**	
Phase 2 Live Test (Hbase **AND** MySQL) • Q1, Q2, Q3	Sunday 04/15/2018 17:00:00 ET	**Sunday 04/15/2018 23:59:59 ET**	**Tuesday 04/17/2018 23:59:59 ET**
Phase 3 • Q1, Q2, Q3, Q4	Monday 04/16/2018 00:00:00 ET	Sunday 04/29/2018 15:59:59 ET	
Phase 3 Live Test (Hbase **OR** MySQL) • Q1, Q2, Q3, Q4	Sunday 04/29/2018 17:00:00 ET	Sunday 04/29/2018 23:59:59 ET	Tuesday 05/01/2018 23:59:59 ET

Questions?