15-319 / 15-619 Cloud Computing

Recitation 12

November 14th 2017

Overview

- **Last week's reflection**
  - Team project phase 2
  - Quiz 10
- **This week's schedule**
  - Project 4.2
  - OLI Modules 19 & 20
  - Quiz 11
- **Twitter Analytics: The Team Project**
  - Phase 3

Conceptual Modules to Read on OLI

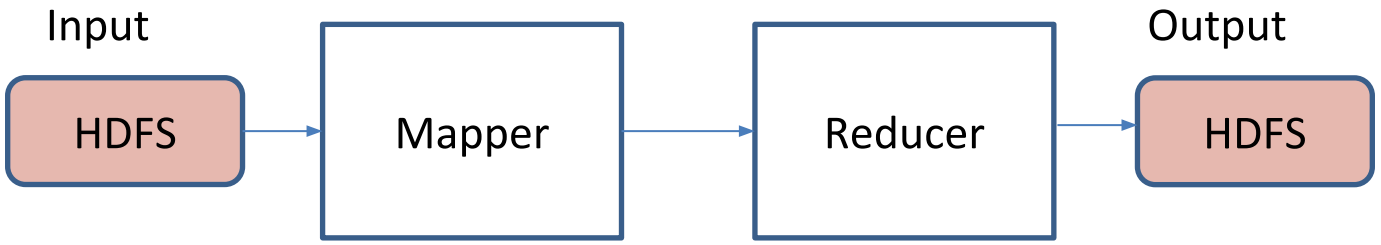
- UNIT 5: Distributed Programming and Analytics Engines for the Cloud
  - Module 18: Introduction to Distributed Programming for the Cloud
  - Module 19: Distributed Analytics Engines for the Cloud: MapReduce
  - Module 20: Distributed Analytics Engines for the Cloud: Spark

Project 4

- Project 4.1, Batch Processing with MapReduce
  - MapReduce Programming
- Project 4.2
  - Iterative Batch Processing Using Apache Spark
- Project 4.3
  - Stream Processing using Kafka/Samza

Typical MapReduce Batch Job

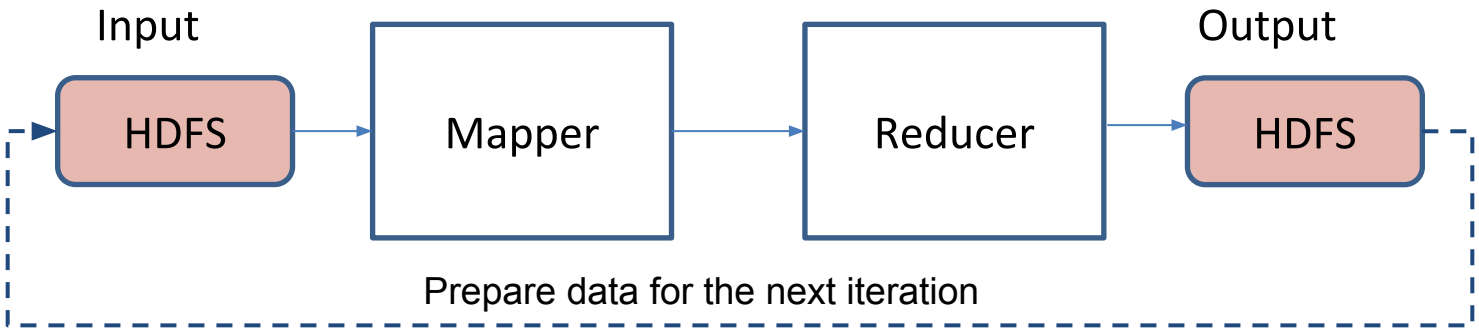
- Simplistic view of a MapReduce job



- You simply write code for the
  - 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, etc.



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

Resilient Distributed Datasets (RDDs)

- New data abstraction, RDDs
  - can be in-memory or on disk
  - are read-only objects
  - are partitioned across the cluster
    - partitioned across machines 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
  - Apply an operation and derive a new RDD

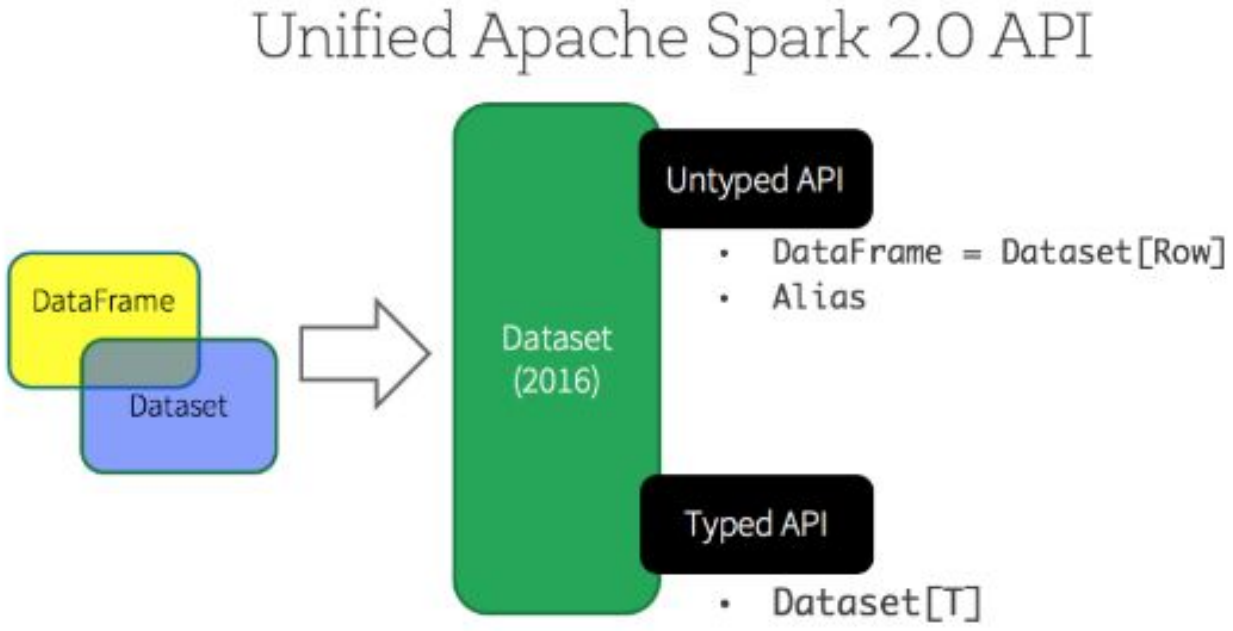
```
>>>transform_RDD =  input_RDD.filter(lambda x: "abcd" in x)
```

- Action
  - Computations on an RDD and return a single object

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

DataFrames

- Like an RDD, a DataFrame is an immutable distributed collection of data, organized into named columns, like a table in a relational database.



DataFrames

- Json:

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

- Load:

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

- Convert dataframe to dataset:

```
case class Person(name: String, age: Long)
val ds = df.as[Person]
```

- SQL:

```
val sqlDF = spark.sql("SELECT name FROM people where age >  20").show()
// +----+
// |name|
// +----+
// |Andy|
// +----+
```

DataFrames

- Json:

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

- Load:

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

- Convert dataframe to dataset:

```
case class Person(name: String, age: Long)
val ds = df.as[Person]
```

- SQL:

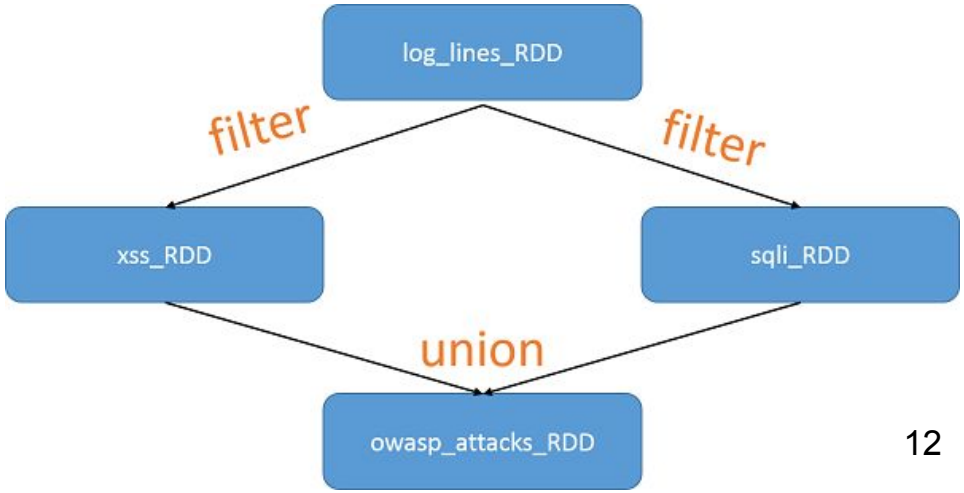
```
val sqlDF = spark.sql("SELECT name FROM people where age >  20").show()
```

- API:

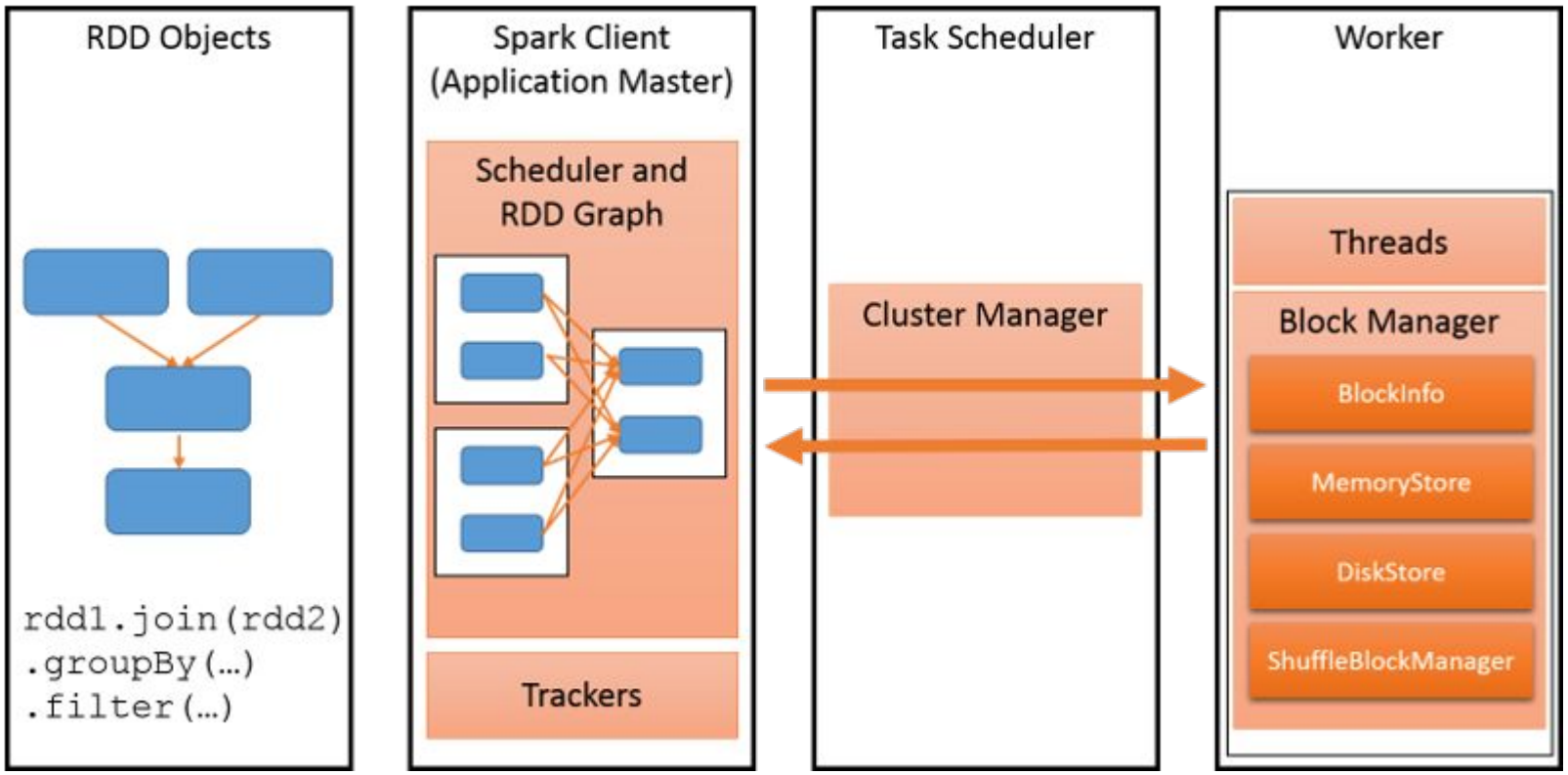
```
df.select("name").filter($"age" > 20).show()
```

RDDs and Fault Tolerance

- Actions create new RDDs
- Instead of replication, recreate RDDs on failure
- Recreate RDDs using lineage
  - RDDs store the transformations required to bring them to current state
  - Provides a form of resilience even though they can be in-memory



The Spark Framework

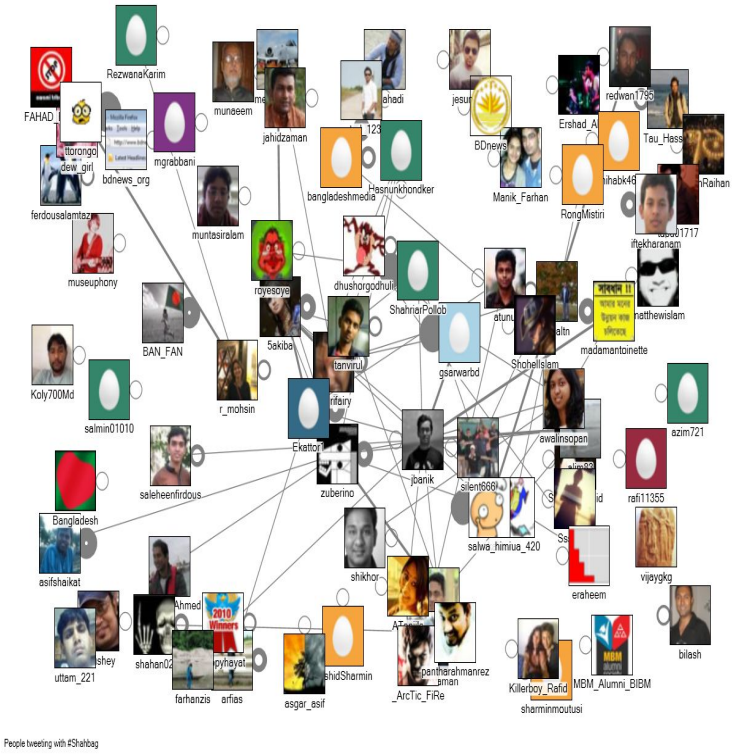


Spark Ecosystem

- Spark SQL
  - Allows running of SQL-like queries against RDDs
- Spark Streaming
  - Run spark jobs against streaming data
- MLlib
  - Machine learning library
- GraphX
  - Graph-parallel framework

Project 4.2

- Use Spark to analyze a Twitter social graph
  - **Task 0**
    - Some basic conceptual questions about Spark
  - **Task 1**
    - Number of nodes and edges
    - Number of followers for each user
    - Spark shell, RDD, Dataframe
  - **Task 2**
    - Run PageRank to compute the influence of users
    - Fast runs get a **bonus**
  - **Task 3**
    - 2nd-degree centrality on the graph using GraphX

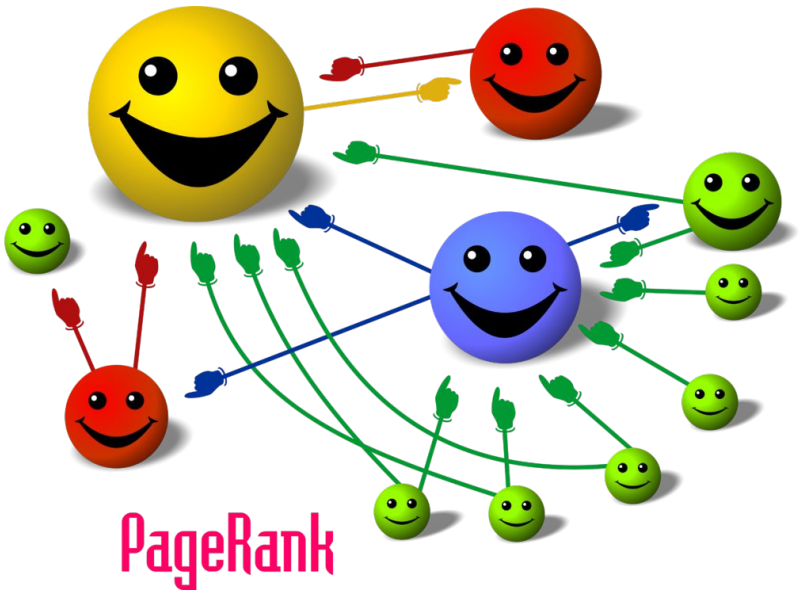


Project 4.2 - Three Main Tasks

1. Enumerate the Twitter Social Graph
   - Find the number of nodes and edges
   - Edges in the graph are directed. (u, v) and (v, u) should be counted as two edges
   - Find the number of followers for each user
   - Compare **RDD/Dataframe** implementations
2. Rank each user by influence
   - Run PageRank with **10** iterations
   - **All users' scores need to sum up to 1.0 at every iteration.**
3. Second degree centrality
   - Need to use GraphX with Scala.

Task 2: The PageRank Algorithm

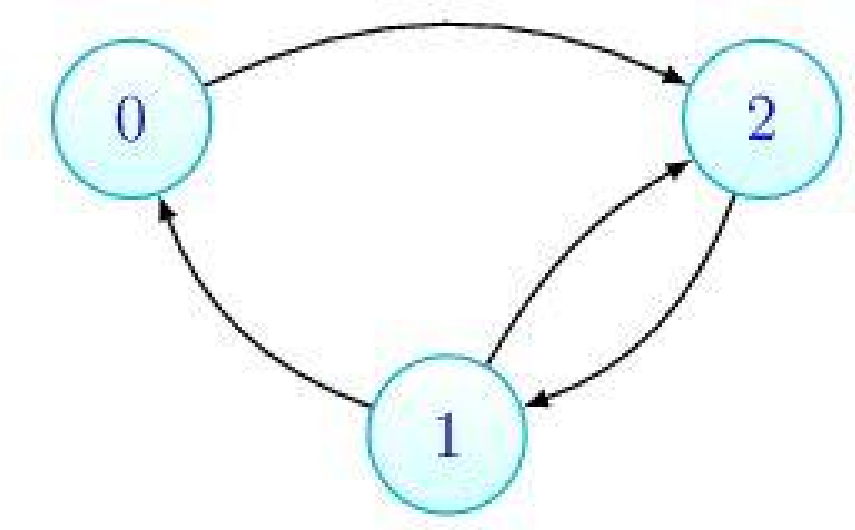
- Give ranks (scores) based on links to them
- A page that has:
  - Links from many nodes ⇒ high rank
  - Link from a high-ranking node ⇒ high rank



The PageRank Algorithm

The PageRank algorithm can find important or influential vertices in a graph.

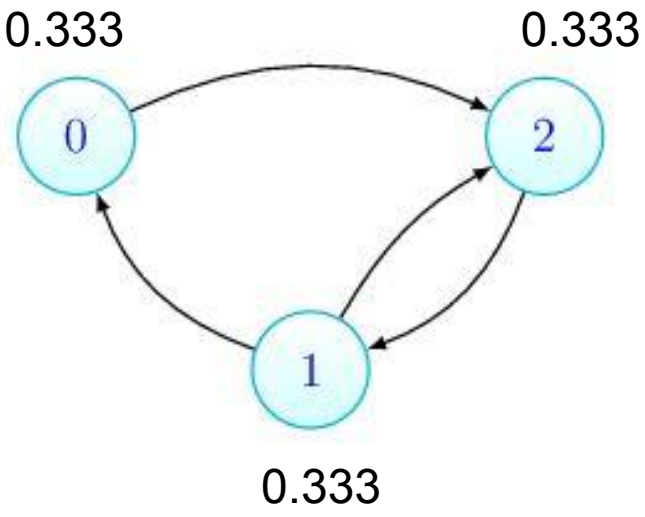
Which node is more important or influential?



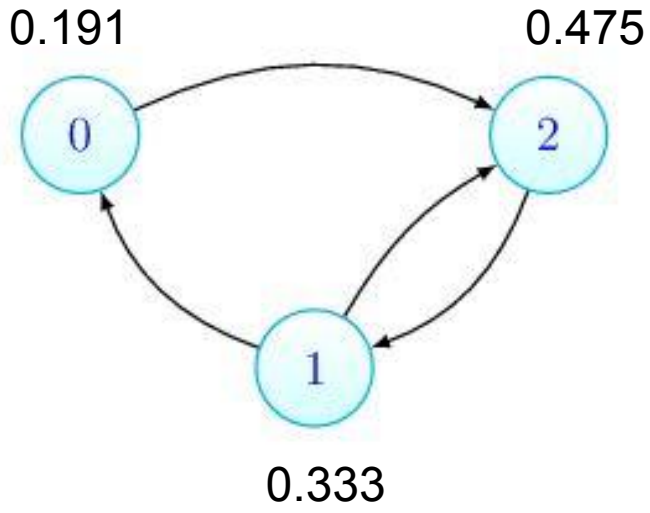
The PageRank Algorithm

Node 0 passes its score to Node 2.
Node 1 passes its score to Node 0 and Node 2.
Node 2 passes its score to Node 1.

Iteration 0

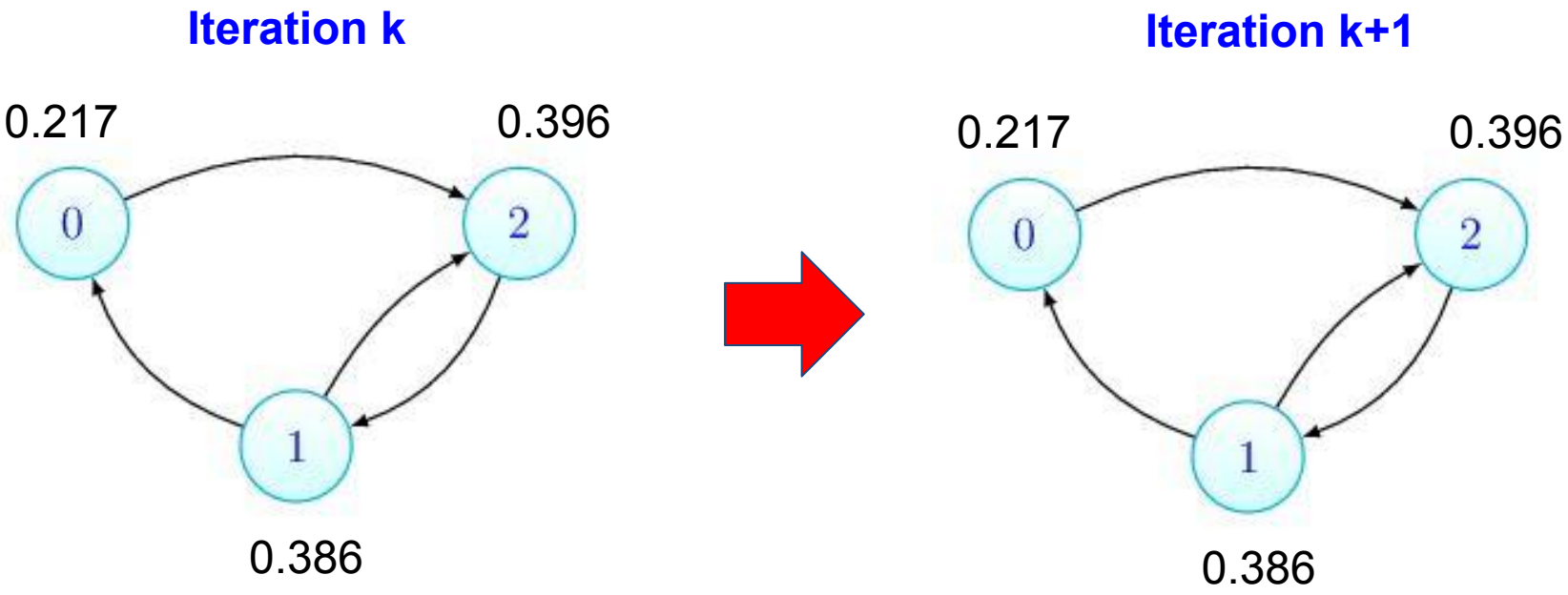


Iteration 1

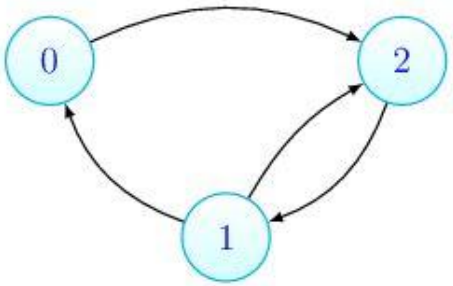


The PageRank Algorithm

When the score of every node does not change across iterations, we refer to it as the algorithm converged.



The PageRank Algorithm



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

- Transition matrix: (row sums to 1)

$$M_{ij} = \frac{G_{ij}}{\sum_{k=1}^{n} G_{ik}} \left(\text{when } \sum_{k=1}^{n} G_{ik} \neq 0 \right) \qquad \mathbf{M} = \begin{bmatrix} 0 & 0 & 1 \\ 0.5 & 0 & 0.5 \\ 0 & 1 & 0 \end{bmatrix}$$

What if a node has zero outgoing links?
Set each element of that row to 1/n,
where n is the number of nodes in the graph.

The PageRank Algorithm

Algorithm 1: PageRank algorithm

input : the transition matrix $\mathbf{M}$, the number of nodes in graph n, and the damping factor d

output: the converged PageRank score vector $\mathbf{r}$

1 $\mathbf{r}^{(0)} = [\frac{1}{n}]_{n\times 1}$;
2 **while** $\delta \geq e^{-9}$ *and* $k \leq 10$ **do**
3 update $\mathbf{r}^{(k+1)}$ using $\mathbf{r}^{(k)}$;
4 $\delta = \|\mathbf{r}^{(k+1)} - \mathbf{r}^{(k)}\|_2^2$; // calculate the difference
5 k++;
6 **end**

PageRank algorithm can usually converge in 10 iterations.
Two ways to implement Step 3:

- matrix solver
- for-loop solver (what you will be implementing)

Matrix Solver

1.1 Matrix Multiplication Solver

We can update the score by the matrix multiplication:

$$\mathbf{r}^{(k+1)} = d\mathbf{M}^{T}\mathbf{r}^{(k)} + (1-d)\mathbf{r}^{(0)}, \tag{2}$$

where $\mathbf{r}^{(k)}$ indicates the score vector at iteration k and $\mathbf{r}^{(0)} = [\frac{1}{n}]_{n\times 1}$ is the initial score vector. Recall n is the number of nodes in G. d is the damping factor used to jump out of isolated nodes or clusters during the random walk.

The interpretation of Equation (2) is that an agent has probability d to follow an edge in the graph and probability $1-d$ to jump to a random node, where d controls the frequency of the random jump. It guarantees the algorithm does not get trapped into any isolated node cluster during the walking, and thus guarantee the process will eventually converge. In this project, we set $d = 0.85$.

- Matrix implementation can be very fast… but we might not want to implement this here due to memory constraints
- Use sparse matrix implementations to store M.

For-loop Solver

1.2 For-loop Solver

In this alternative implementation, for each node v_i, we have the following update equation:

$$r_i^{(k+1)} = d \sum_{v_j \in \mathcal{N}(v_i)} r_j^{(k)} M_{ji} + (1-d) r_i^{(0)}, \tag{3}$$

where $\mathcal{N}(v_i)$ represents a set of nodes that point to v_i. Recall M_{ji} is an element in the transition matrix. See Eq. (1). Incorporating Eq. (1), we have an equivalent update equation:

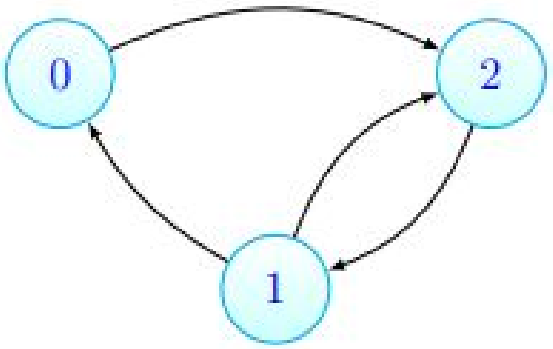
$$r_i^{(k+1)} = d \sum_{v_j \in \mathcal{N}(v_i)} \frac{r_j^{(k)}}{\sum_{k=1}^{n} G_{jk}} + (1-d)\frac{1}{n}. \tag{4}$$

Note that $\sum_{k=1}^{n} G_{jk}$ cannot become 0.

You need to implement the for-loop solver in this project. The for-loop solver cloud be slower as matrix multiplication operations are usually optimized in its system library. But anyway the for-loop solver is easier to implement for very large graphs.

- Less efficient but scalable to very large graphs.
- You **are required to** implement the for-loop solver.

A Toy Example - 1



Adjacent matrix $\mathbf{G}$:

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

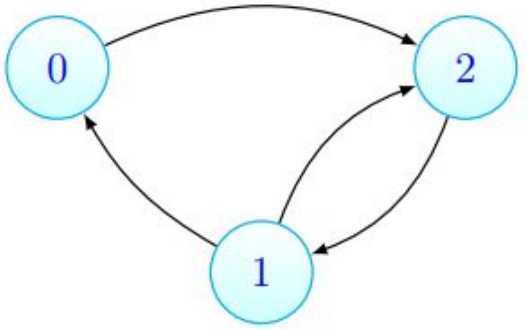
Transition matrix $\mathbf{M}$

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

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

A Toy Example - 1

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



$$d = 0.85$$

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

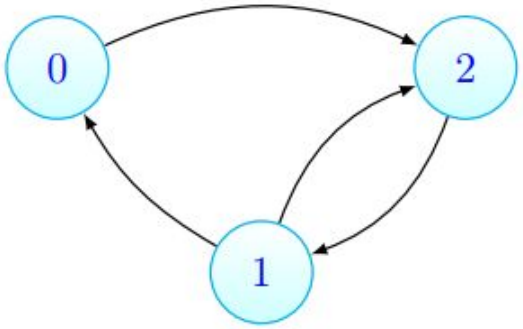
$$r_0^{(1)} = d\frac{r_1^{(0)}}{2} + (1-d)\frac{1}{n}$$

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

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

A Toy Example - 1

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



$$d = 0.85$$

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

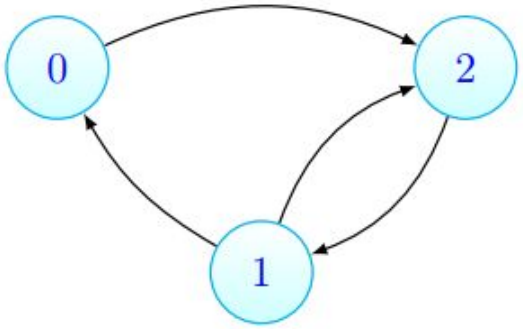
$$r_0^{(1)} = 0.85\frac{1}{6} + 0.15\frac{1}{3} = 0.191$$

$$r_1^{(1)} = 0.85\frac{1}{3} + 0.15\frac{1}{3} = 0.333$$

$$r_2^{(1)} = 0.85(\frac{1}{3} + \frac{1}{6}) + 0.15\frac{1}{3} = 0.475$$

A Toy Example - 1

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

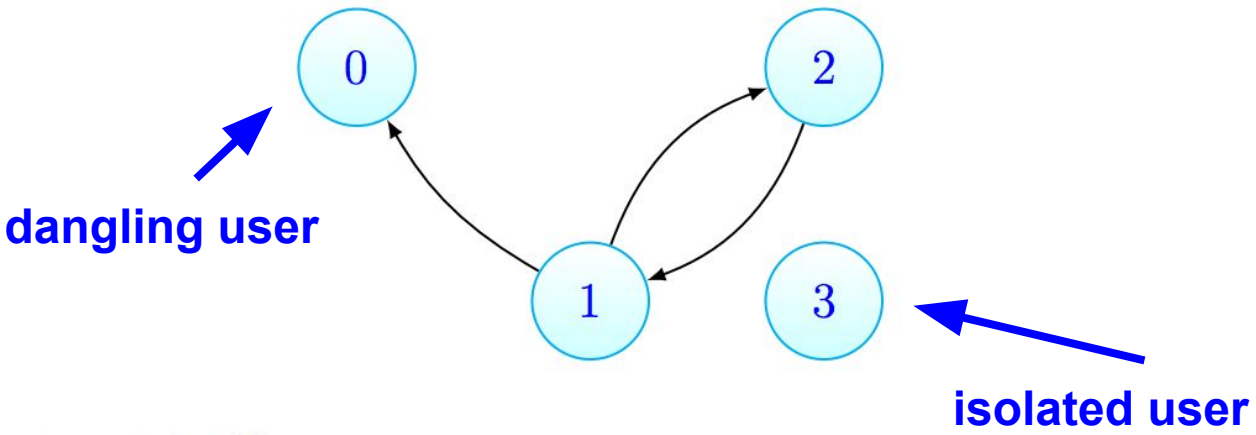


$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 1 \\ 0.5 & 0 & 0.5 \\ 0 & 1 & 0 \end{bmatrix}$$

The converged result is

$$r_0^{(k)} = 0.217$$
$$r_1^{(k)} = 0.386$$
$$r_2^{(k)} = 0.396$$

A Toy Example - 2



Adjacent matrix $\mathbf{G}$:

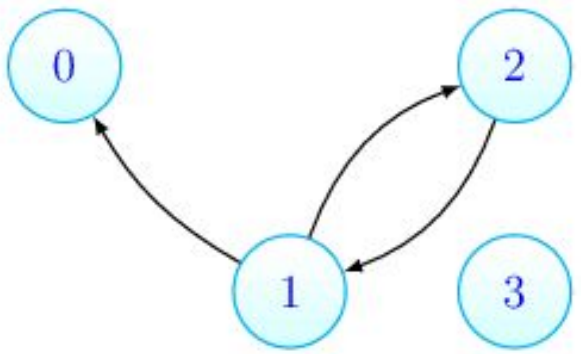
$$\mathbf{G} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Transition matrix $\mathbf{M}$

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

A Toy Example - 2

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

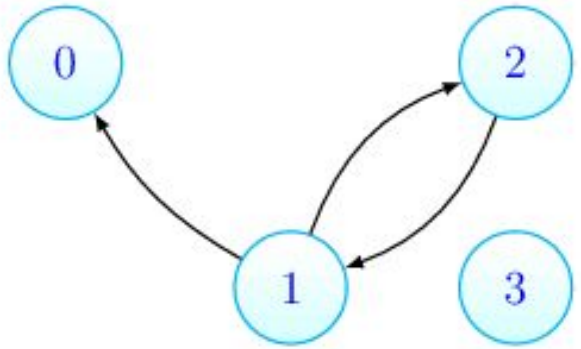


$$r_0^{(1)} = d(\frac{r_1^{(0)}}{2} + \frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}) + (1-d)\frac{1}{n}$$
$$r_1^{(1)} = d(\frac{r_2^{(0)}}{1} + \frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}) + (1-d)\frac{1}{n}$$
$$r_2^{(1)} = d(\frac{r_1^{(0)}}{2} + \frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}) + (1-d)\frac{1}{n}$$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

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

A Toy Example - 2

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



$$\epsilon = d(\frac{r_0^{(0)}}{4} + \frac{r_3^{(0)}}{4}) \qquad d = 0.85$$

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

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

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

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

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

PageRank in Spark (Scala)

(Note: This is a pseudocode of PageRank, simpler than P4.2)

```
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)
}
```

Bonus Task: Speed up Spark

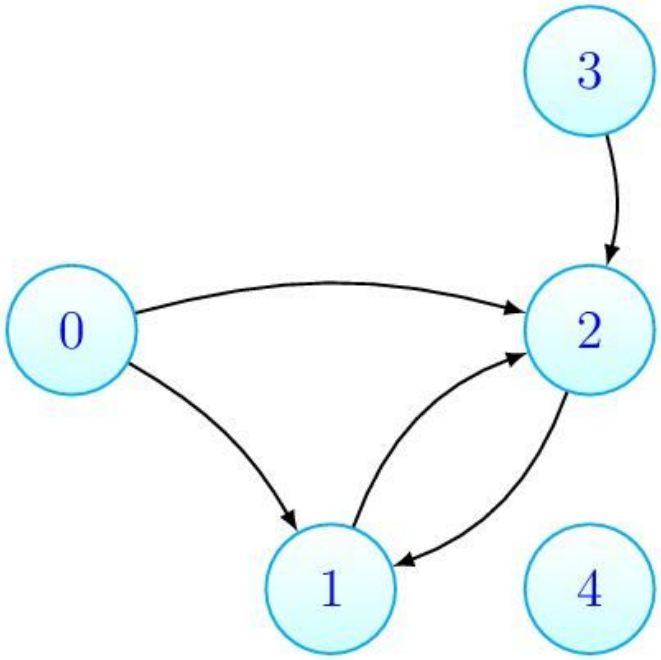
Hints:

- Eliminate repeated calculations in PageRank.
- Monitor your instances to make sure they are fully utilized.
- Develop a better understanding of RDD manipulations. Understand the "lazy" transformation in Spark.
- Spark is a tunable framework where there are many parameters that you can configure to make the best use of the resources.
- Be careful with repartition on your RDDs.

Graph Processing using GraphX

Task 3: Second-degree Centrality

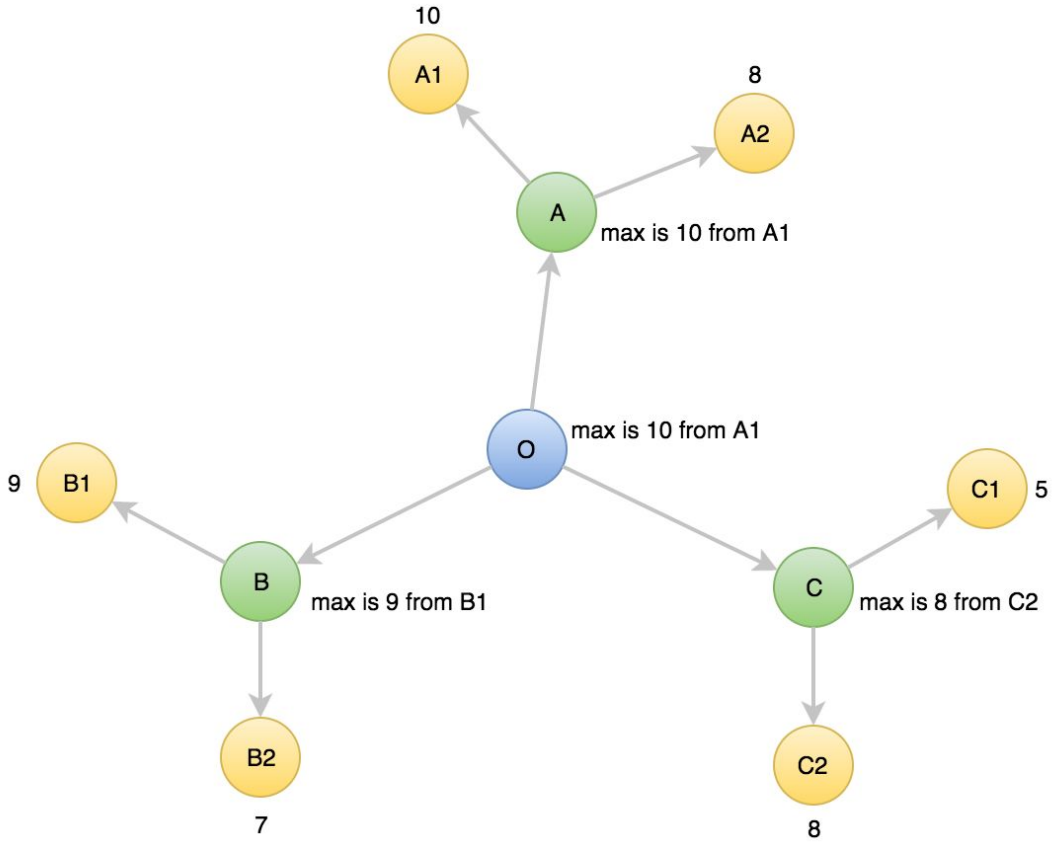
- PageRank score is a type of centrality score.
  - Importance of a node in a graph.
- However, the PageRank score for Node0 and Node4 is the same: 0.0306.
  - Does not make sense!
- Use PageRank to measure a 2nd degree centrality score.



Graph Processing using GraphX

Task 3: Second-degree Centrality

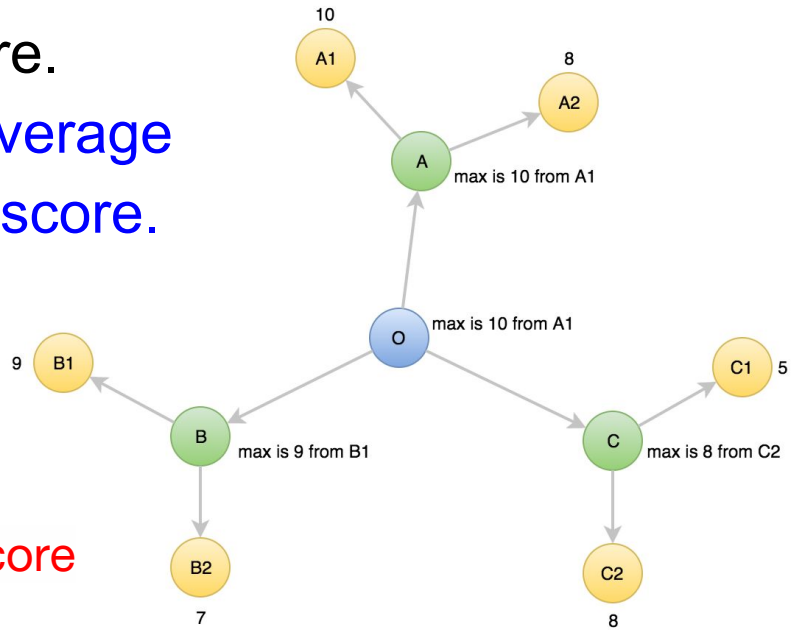
- From all the people your followees follow (i.e. your 2nd degree followees), find the 2nd-degree centrality score which is the highest PageRank score within the reach of 1 or 2 jumps.



Graph Processing using GraphX

Task 3: Second-degree Centrality

- From all the people your followees follow (i.e. your 2nd degree followees), find the one with the highest PageRank score.
- First calculate this score and then average this score with its original pagerank score.



new_influencial_score = 0.5 * pagerank_score +
0.5 * most_influential_second_degree_user_score

Hints for Launching a Spark Cluster

- Spark is an in-memory system
  - Develop and test your scripts on a portion of the dataset before launching a big cluster.
  - Look at examples in the writeups.
- A regular run in Task 2 would take 45 min using **5** r3.xlarge.
- It is possible to test Spark on your local machine

Spark Shell

- Like the python shell
- Run commands interactively
- On the master, execute (from /root)
  - ./spark/bin/spark-shell
  - ./spark/bin/pyspark

P4.2 Grading - 1

- Task 0
  - Use the student AMI
  - Authenticate via port 15319 or 15619
  - 5 questions are in the write up
  - Put your answers in the answers file on the student AMI

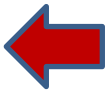
P4.2 Grading - 2

- Submit your work from the EMR instance
- Don’t forget to submit your code
- Task 1
  - Put the number of nodes and edges in the answer file
  - Put your Spark programs that count the number of followers for each user in the given folder
  - Run the submitter to submit
- Task 2
  - Put your Spark program that calculates the pagerank score for each user in the given folder
  - Run the submitter to submit
  - Check if the sum of PageRank score is 1.0 at each iteration
  - **Bonus**: tune your system to run as fast as possible

P4.2 Grading - 3

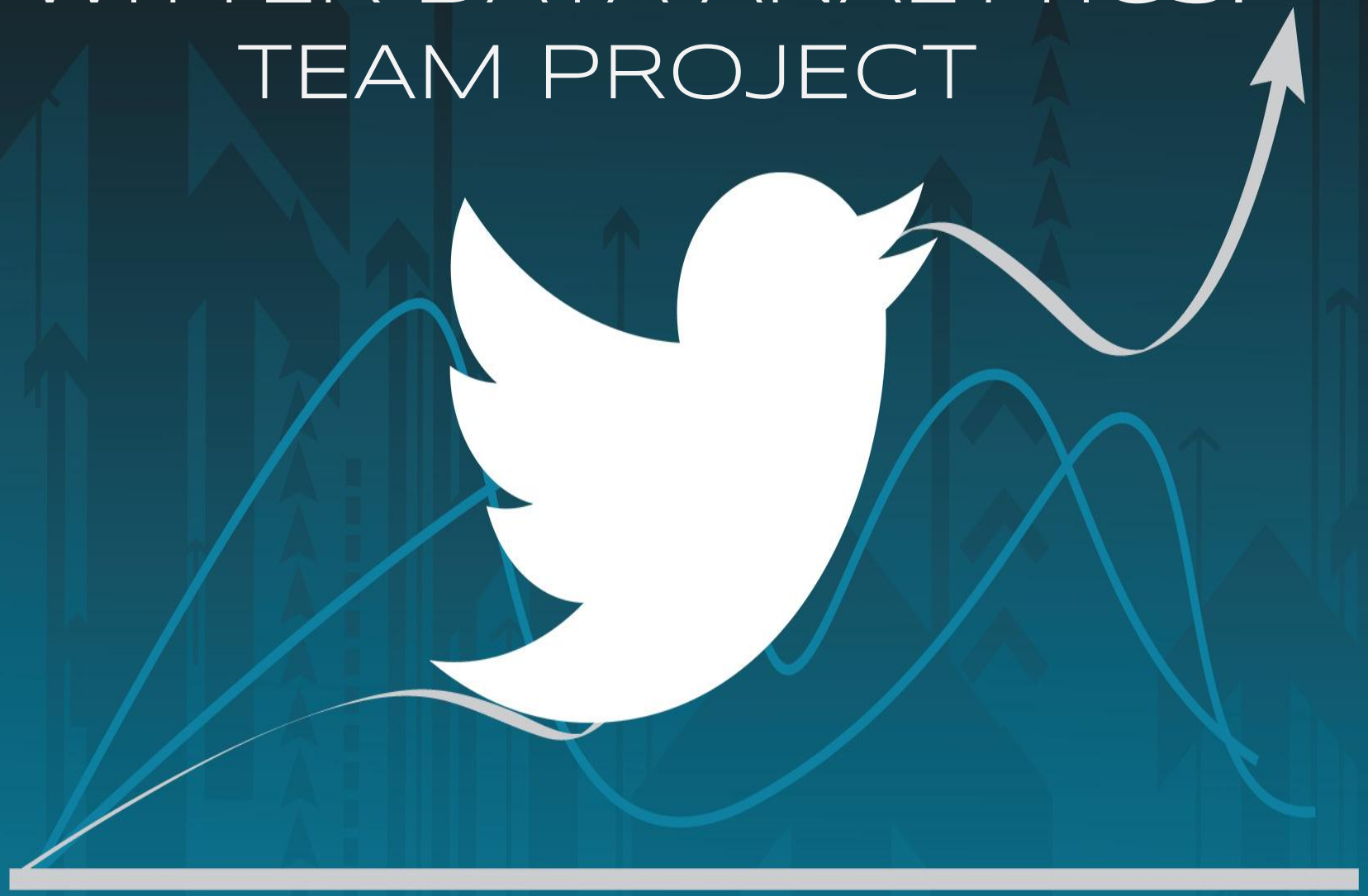
- Submit your work from the EMR instance
- Don’t forget to submit your code
- Task 3
  - Put your GraphX program to calculate the new centrality score in the given folder
  - Run the submitter to submit

Upcoming Deadlines

- Quiz 11: OLI Modules 19 and 20
  - Quiz 11 due: 11/17/2017 11:59 PM Pittsburgh 
- Team Project : Phase 2
  - Code and report due: 11/14/2017 11:59 PM Pittsburgh 
- Project 4.2 : Iterative Programming with Spark
  - Due: 11/19/2017 11:59 PM Pittsburgh 
- Team Project : Phase 3
  - Live-test due: 12/03/2017 3:59 PM Pittsburgh
  - Code and report due: 12/05/2017 11:59 PM Pittsburgh

Questions?

TWITTER DATA ANALYTICS: TEAM PROJECT



Team Project Phase 2 MySQL Live Test Honor Board

The Brogrammers	40
AotianLong	40
squirrel	40
BVW	38.92
Targaryen	38.77
CrispyChicken	38.61
Tricorn	37.27
Tourists	36.07
MyHeartIsInCC	35.02
Steelers	34.85

Team Project Phase 2 HBase Live Test Honor Board

Tricorn	40
Steelers	37.5
The Brogrammers	35.24
hkJournalist	34.46
MyHeartIsInCC	33.15
CrispyChicken	32.77
Targaryen	28.13
AotianLong	26.88
Tourists	26.58
THL	25

Team Project Phase 2 Live Test Honor Board

Tricorn	77
The Brogrammers	74
Steelers	71
CrispyChicken	69
MyHeartIsInCC	66
AotianLong	66
Targaryen	65
squirrel	62
Tourists	60
hkJournalist	60

Query 4: Interactive Tweet Server

In this query, you are going to build a web service that supports READ, WRITE, SET and DELETE requests on tweets. Our server will send a mix of read, write, set and delete requests to your web service to test for correctness and performance.

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.

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` is the url-encoded json string; same structure as the original tweet json; 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
```

- Range read

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 url-encoded

Query 4: Tweet Server

- Malformed Request

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

- Response

```
TEAMID,TEAM_AWS_ACCOUNT_ID\n
success\n
```

Team Project General Hints

- Identify the bottlenecks using fine-grained profiling.
- Do not cache naively.
- Review what we have learned in previous project modules
  - Scale out
  - Load balancing (are requests balanced?)
  - Replication and sharding
- 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. Pay attention to scalability.
- Adopt a forwarding mechanism or a non-forwarding mechanism
  - You may need a custom load balancer
- May need many connections at the same time, in the case of out of order sequence numbers.
- Consider batch writes. Writes and reads are exclusive due to the consistency model.

Team Project Time Table

Phase	Query	Start	Deadline	Code and Report Due
Phase 1	Q1	Monday 10/9/2017 00:00:01 EST	Sunday 10/22/2017 23:59:59 EST	-
	Q2	Monday 10/9/2017 00:00:01 EST	Sunday 10/29/2017 23:59:59 EST	Tuesday 10/31/2017 23:59:59 EST
Phase 2	Q1, Q2, Q3	Monday 10/30/2017 00:00:01 EST	Sunday 11/12/2017 15:59:59 EST	-
	Live Test Q1, Q2, Q3	Sunday 11/12/2017 18:00:01 ET	Sunday 11/12/2017 23:59:59 EST	**Tuesday 11/14/2017 23:59:59 EST**
Phase 3	Q1, Q2, Q3, Q4	Monday 11/13/2017 00:00:01 ET	Sunday 12/03/2017 15:59:59 EST	-
	Live Test Q1, Q2, Q3, Q4	Sunday 12/03/2017 18:00:01 ET	Sunday 12/03/2017 21:00:00 EST	Tuesday 12/05/2017 23:59:59 EST

Questions?