Mining Massive Datasets

Lecture 13

Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Final Exam: Start Learning NOW!

- Final exam is in exactly 2 weeks:
  - On Feb, 5th, 16-18 CET
- Registration is <u>mandatory</u> if you want to participate
  - From Jan 28 until on Feb 1st (separate email)
- Suggestion for learning:
  - For contents from the MMD-book use the videos:
    - http://www.mmds.org/, under „The 2nd edition …"
    - Also here: https://heibox.uni-heidelberg.de/d/ada004a39f/
  - Next week: overview of relevant topics

Note on Slides

A substantial part of these slides come (either verbatim or in a modified form) from the book
Mining of Massive Datasets
by Jure Leskovec, Anand Rajaraman, Jeff Ullman
(Stanford University).
For more information, see the website accompanying the book: http://www.mmds.org.

Infinite Data

High dim. data	Graph data	Infinite data	Machine learning	Apps
Locality sensitive hashing	PageRank, SimRank	Filtering data streams	SVM	Recommender systems
Clustering	Community Detection	Web advertising	Decision Trees	Association Rules
Dimensionality reduction	Spam Detection	Queries on streams	Perceptron, kNN	Duplicate document detection

Programming in Spark & MapReduce

PageRank:

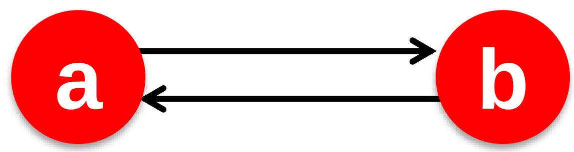
The Google Formulation

PageRank: Three Questions

$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d}_\mathrm{i}}$$ **or equivalently** $$r = Mr$$

- Does this (always) converge?
- Does it (always) converge to what we want?
- Are results reasonable?

Does this converge?



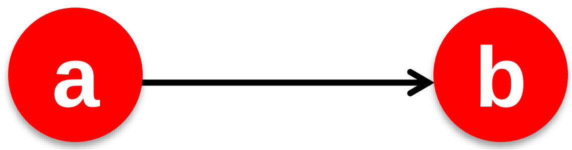
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d_i}}$$

- **Example:**

$$\begin{array}{ccccccc} r_a & \multirow{2}{*}{=} & 1 & 0 & 1 & 0 \\ r_b & & 0 & 1 & 0 & 1 \end{array}$$

Iteration 0, 1, 2, …

Does it converge to what we want?



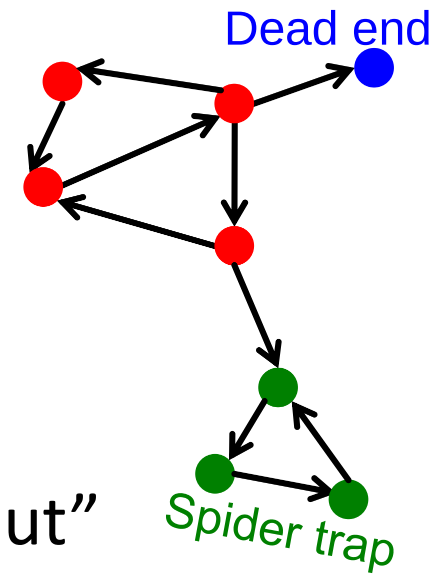
$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d}_i}$$

- **Example:**

$$\begin{array}{c} r_a \\ r_b \end{array} = \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}$$

Iteration 0, 1, 2, …

PageRank: Problems



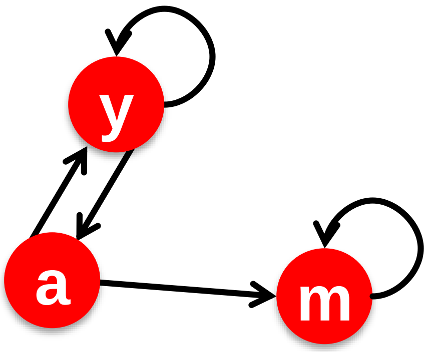
2 problems:

- **(1)** Some pages are **dead ends** (have no out-links)
  - Random walk has "nowhere" to go to
  - Such pages cause importance to "leak out"

- **(2) Spider traps:** (all out-links are within the group)
  - Random walked gets "stuck" in a trap
  - And eventually spider traps absorb all importance

Problem: Spider Traps

- **Power Iteration:**
  - Set $r_j = 1$
  - $r_j = \sum_{i \to j} \frac{r_i}{d_i}$
    - And iterate



m is a spider trap

	y	a	m
y	½	½	0
a	½	0	0
m	0	½	1

$\mathbf{r_y} = \mathbf{r_y}/2 + \mathbf{r_a}/2$

$\mathbf{r_a} = \mathbf{r_y}/2$

$\mathbf{r_m} = \mathbf{r_a}/2 + \mathbf{r_m}$

- **Example:**

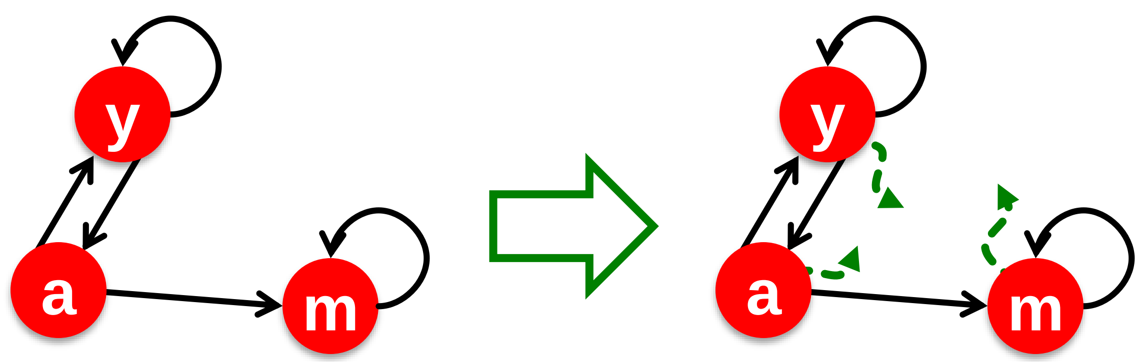
$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \quad \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \ldots & 0 \\ 1/3 & 3/6 & 7/12 & 16/24 & & 1 \end{matrix}$$

Iteration 0, 1, 2, …

All the PageRank score gets "trapped" in node m.

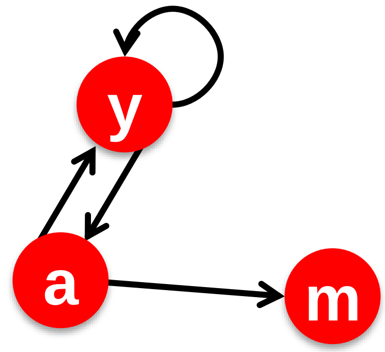
Solution: Teleports!

- **The Google solution for spider traps:** At each time step, the random surfer has two options:
  - With prob. β, follow a link at random
  - With prob. $1-\beta$, jump to some random page
  - Common values for β are in the range 0.8 to 0.9
- Surfer will teleport out of spider trap within a few time steps



Problem: Dead Ends

- **Power Iteration:**
  - Set $r_j = 1$
  - $r_j = \sum_{i \to j} \frac{r_i}{d_i}$
    - And iterate



	y	a	m
y	½	½	0
a	½	0	0
m	0	½	0

$$\mathbf{r_y = r_y/2 + r_a/2}$$
$$\mathbf{r_a = r_y/2}$$
$$\mathbf{r_m = r_a/2}$$

- Example:

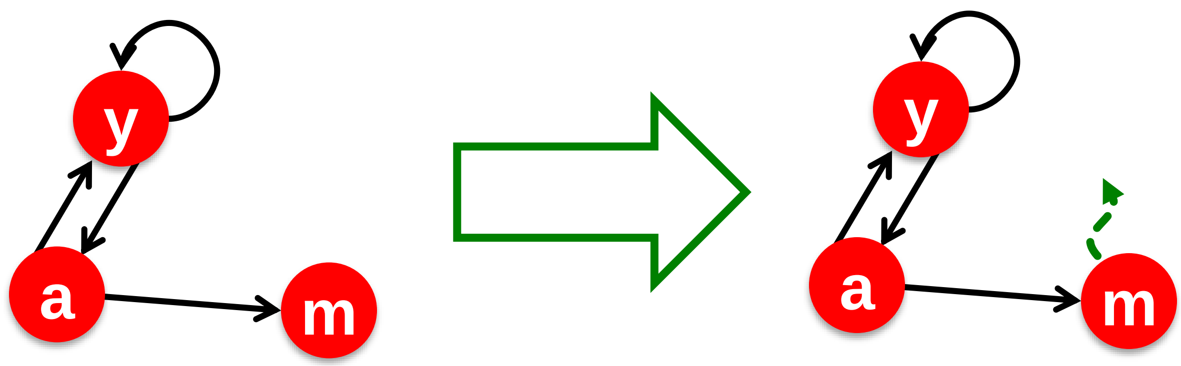
$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \ldots & 0 \\ 1/3 & 1/6 & 1/12 & 2/24 & & 0 \end{matrix}$$

Iteration 0, 1, 2, …

Here the PageRank "leaks" out since the matrix is not stochastic.

Solution: Always Teleport!

- **Teleports:** Follow random teleport links with probability 1.0 from dead-ends
  - Adjust matrix accordingly



	y	a	m
y	½	½	0
a	½	0	0
m	0	½	0

	y	a	m
y	½	½	⅓
a	½	0	⅓
m	0	½	⅓

Why Teleports Solve the Problem?

- **Spider-traps** are not a problem, but with traps PageRank scores are **not** what we want
  - **Solution:** Never get stuck in a spider trap by teleporting out of it in a finite number of steps
- **Dead-ends** are a "formal" a problem
  - The matrix is <u>not column stochastic</u> so our initial assumptions are not met
  - **Solution:** Make matrix column stochastic by always teleporting when there is nowhere else to go

Solution: Random Teleports

- **Google's solution that does it all:**
  At each step, random surfer has two options:
  - With probability β, follow a link at random
  - With probability $1-\beta$, jump to a random page
- **PageRank equation** [Brin-Page, 98]

$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$$

d_i ... out-degree of node i

This formulation assumes that M has no dead ends. We can either **preprocess matrix M to remove all dead ends** or explicitly follow random teleport links with probability 1.0 from dead-ends.

The Google Matrix

- **PageRank equation** [Brin-Page, '98]

$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$$

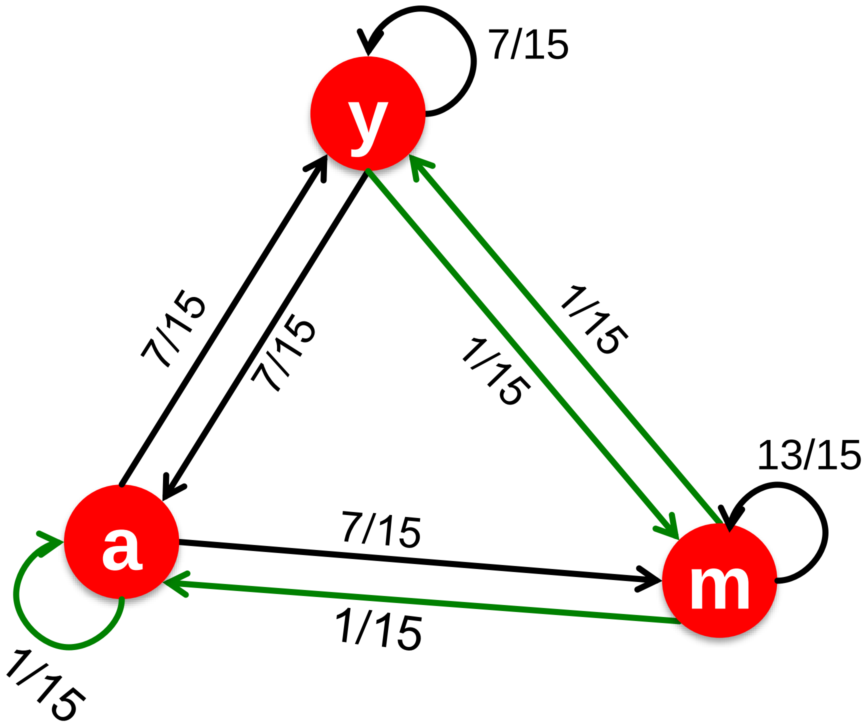
- **The Google Matrix *A*:**

[1/N]$_{NxN}$…N by N matrix where all entries are 1/N

$$A = \beta\, M + (1-\beta)\left[\frac{1}{N}\right]_{N \times N}$$

- **We have a recursive problem: $\boldsymbol{r} = \boldsymbol{A} \cdot \boldsymbol{r}$**
  And the Power method still works!
- **What is $\boldsymbol{\beta}$?**
  - In practice β =0.8,0.9 (make 5 steps on avg., jump)

Random Teleports ($\beta = 0.8$)



$$0.8 \overset{\mathbf{M}}{\begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}} + 0.2 \overset{[\mathbf{1/N}]_{\mathbf{NxN}}}{\begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}}$$

$$\begin{array}{c} \text{y} \\ \text{a} \\ \text{m} \end{array} \begin{bmatrix} 7/15 & 7/15 & 1/15 \\ 7/15 & 1/15 & 1/15 \\ 1/15 & 7/15 & 13/15 \end{bmatrix}$$

A

$$\begin{array}{c} \text{y} \\ \text{a} \\ \text{m} \end{array} = \begin{array}{c} 1/3 \\ 1/3 \\ 1/3 \end{array} \quad \begin{array}{c} 0.33 \\ 0.20 \\ 0.46 \end{array} \quad \begin{array}{c} 0.24 \\ 0.20 \\ 0.52 \end{array} \quad \begin{array}{c} 0.26 \\ 0.18 \\ 0.56 \end{array} \quad \dots \quad \begin{array}{c} 7/33 \\ 5/33 \\ 21/33 \end{array}$$

PageRank:

How do we actually compute the PageRank?

Computing Page Rank

- Key step is **matrix-vector multiplication**
  - $\boldsymbol{r}^{\text{new}} = \boldsymbol{A} \cdot \boldsymbol{r}^{\text{old}}$
- Easy if we have enough main memory to hold **A**, $\mathbf{r}^{\text{old}}$, $\mathbf{r}^{\text{new}}$
- **Say N = 1 billion pages**
  - We need 4 bytes for each entry (say)
  - 2 billion entries for vectors, approx 8GB
  - Matrix **A** has N^2 entries
    - 10^{18} is a large number!
  - Problem: **M** is sparse, **A** not!

$$\mathbf{A} = \beta \cdot \mathbf{M} + (1-\beta)\,[1/N]_{N\times N}$$

$$\boldsymbol{A} = 0.8 \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} + 0.2 \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}$$

$$= \begin{bmatrix} 7/15 & 7/15 & 1/15 \\ 7/15 & 1/15 & 1/15 \\ 1/15 & 7/15 & 13/15 \end{bmatrix}$$

Matrix Formulation

- Suppose there are N pages
- Consider page i, with $\mathbf{d}_i$ out-links
- We have $M_{ji} = 1/|d_i|$ when $i \to j$
  and $M_{ji} = 0$ otherwise
- The random teleport is equivalent to:
  - Adding a **teleport link** from i to every other page and setting transition probability to $(1-\beta)/N$
  - Reducing the probability of following each out-link from $1/|d_i|$ to $\beta/|d_i|$
  - **Equivalent:** *Tax* each page a fraction $(1-\beta)$ of its score and redistribute evenly

Rearranging the Equation

- $\boldsymbol{r} = \boldsymbol{A} \cdot \boldsymbol{r}$, **where** $\boldsymbol{A_{ji}} = \boldsymbol{\beta}\, \boldsymbol{M_{ji}} + \frac{\mathbf{1}-\boldsymbol{\beta}}{\boldsymbol{N}}$
- $r_j = \sum_{\mathrm{i}=1}^{N} A_{ji} \cdot r_i$
- $r_j = \sum_{i=1}^{N} \left[\beta\, M_{ji} + \frac{1-\beta}{N}\right] \cdot r_i$

  $= \sum_{\mathrm{i}=1}^{N} \beta\, M_{ji} \cdot r_i + \frac{1-\beta}{N} \sum_{\mathrm{i}=1}^{N} r_i$

  $= \sum_{\mathrm{i}=1}^{N} \beta\, M_{ji} \cdot r_i + \frac{1-\beta}{N}$ since $\sum r_i = 1$
- **So we get:** $\boldsymbol{r} = \boldsymbol{\beta}\, \boldsymbol{M} \cdot \boldsymbol{r} + \left[\frac{\mathbf{1}-\boldsymbol{\beta}}{\boldsymbol{N}}\right]_{\boldsymbol{N}}$

Note: Here we assumed **M** has no dead-ends

$[x]_N$ … a vector of length N with all entries x

Sparse Matrix Formulation

- We just rearranged the **PageRank equation**

$$\boldsymbol{r} = \boldsymbol{\beta M} \cdot \boldsymbol{r} + \left[\frac{\boldsymbol{1-\beta}}{\boldsymbol{N}}\right]_{\boldsymbol{N}}$$

  - where $[(1-\beta)/N]_N$ is a vector with all N entries $(1-\beta)/N$

- ***M*** is a **sparse matrix!** (with no dead-ends)
  - 10 links per node, approx 10N entries
- So in each iteration, we need to:
  - Compute $\boldsymbol{r}^{\text{new}} = \beta \boldsymbol{M} \cdot \boldsymbol{r}^{\text{old}}$
  - Add a constant value $(1-\beta)/N$ to each entry in $\boldsymbol{r}^{\text{new}}$
    - Note: if M contains dead-ends then $\sum_j r_j^{new} < 1$ and we also have to renormalize r^{new} so that it sums to 1

PageRank: The Complete Algorithm

- **Input: Graph G and parameter β**
  - Directed graph $\boldsymbol{G}$ (can have **spider traps** and **dead ends**)
  - Parameter $\boldsymbol{\beta}$
- **Output: PageRank vector $\boldsymbol{r}^{new}$**

- **Set:** $r_j^{old} = \frac{1}{N}$
- **repeat until convergence:** $\sum_j \left| r_j^{new} - r_j^{old} \right| < \varepsilon$
  - $\forall j$: $\boldsymbol{r'}_j^{new} = \sum_{i \to j} \boldsymbol{\beta} \frac{\boldsymbol{r}_i^{old}}{\boldsymbol{d}_i}$ (this is $\boldsymbol{\beta M} \cdot \boldsymbol{r}$)

    $\boldsymbol{r'}_j^{new} = \mathbf{0}$ if in-degree of $\boldsymbol{j}$ is **0**
  - Now re-insert the leaked PageRank:

    $\forall \boldsymbol{j}$: $\boldsymbol{r}_j^{new} = \boldsymbol{r'}_j^{new} + \frac{\mathbf{1} - \boldsymbol{\beta S}}{\boldsymbol{N}}$ where: $S = \sum_j r'^{new}_j$
  - $\boldsymbol{r}^{old} = \boldsymbol{r}^{new}$

If the graph has no dead-ends then the amount of leaked PageRank is **1-β**. But since we have dead-ends the amount of leaked PageRank may be larger. We have to explicitly account for it by computing **S**.

Sparse Matrix Encoding

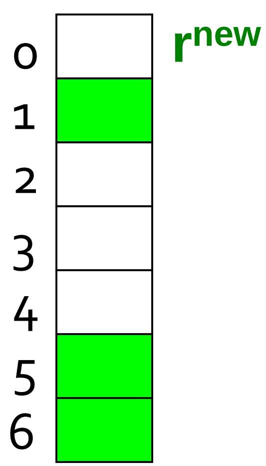
- Encode sparse matrix using only nonzero entries
  - Space proportional roughly to number of links
  - Say 10N, or 4*10*1 billion = 40GB
  - **Still won't fit in memory, but will fit on disk**

source node	degree	destination nodes
0	3	1, 5, 7
1	5	17, 64, 113, 117, 245
2	2	13, 23

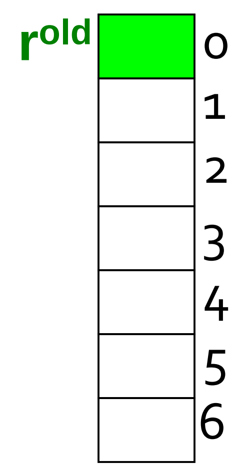
Basic Algorithm: Update Step

- **Assume enough RAM to fit *r^{new}* into memory**
  - Store *r^{old}* and matrix **M** on disk
- **1 step of power-iteration is:**

> **Initialize** all entries of $\mathbf{r}^{new} = \mathbf{(1\text{-}\beta) / N}$
> For each page $\boldsymbol{i}$ (of out-degree $\boldsymbol{d_i}$):
>   Read into memory: $\boldsymbol{i}$, $\boldsymbol{d_i}$, $\boldsymbol{dest_1, \ldots, dest_{d_i}}$, $\boldsymbol{r^{old}(i)}$
>   For $\mathbf{j = 1 \ldots d_i}$ (this is $\boldsymbol{\beta M \cdot r}$)
> $\mathbf{r^{new}(dest_j) \mathrel{+}= \beta\, r^{old}(i) / \boldsymbol{d_i}}$



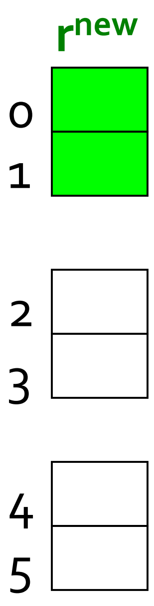
source	degree	destination
0	3	1, 5, 6
1	4	17, 64, 113, 117
2	2	13, 23



Analysis

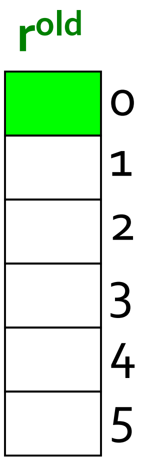
- **Assume enough RAM to fit r^{new} into memory**
  - Store r^{old} and matrix M on disk
- **In each iteration, we have to:**
  - Read r^{old} and M
  - Write r^{new} back to disk
  - **Cost per iteration of Power method:**
    $= 2|r| + |M|$
- **Question:**
  - What if we could not even fit r^{new} in memory?

Block-based Update Algorithm



src	degree	destination
0	4	0, 1, 3, 5
1	2	0, 5
2	2	3, 4

M



- Break $\boldsymbol{r}^{new}$ into $\boldsymbol{k}$ blocks that fit in memory
- Scan $\boldsymbol{M}$ and $\boldsymbol{r}^{old}$ once for each block

Analysis of Block Update

- Similar to nested-loop join in databases
  - Break $\boldsymbol{r}^{\text{new}}$ into $\boldsymbol{k}$ blocks that fit in memory
  - Scan $\boldsymbol{M}$ and $\boldsymbol{r}^{\text{old}}$ once for each block
- Total cost:
  - $\boldsymbol{k}$ scans of $\boldsymbol{M}$ and $\boldsymbol{r}^{\text{old}}$
  - **Cost per iteration of Power method:** $k(|\boldsymbol{M}| + |\boldsymbol{r}|) + |\boldsymbol{r}| = \boldsymbol{k|M| + (k+1)|r|}$
- Can we do better?
  - **Hint:** $\boldsymbol{M}$ is much bigger than $\boldsymbol{r}$ (approx 10-20x), so we must avoid reading it $\boldsymbol{k}$ times per iteration

Block-Stripe Update Algorithm

r^{new}: 0, 1 | 2, 3 | 4, 5

src	degree	destination
0	4	0, 1
1	3	0
2	2	1

0	4	3
2	2	3

0	4	5
1	3	5
2	2	4

r^{old}: 0, 1, 2, 3, 4, 5

Break *M* into stripes! Each stripe contains only destination nodes in the corresponding block of $\boldsymbol{r}^{new}$

Block-Stripe Analysis

- Break $\boldsymbol{M}$ into stripes
  - Each stripe contains only destination nodes in the corresponding block of $\boldsymbol{r}^{\text{new}}$
- Some additional overhead per stripe
  - But it is usually worth it
- => Cost per iteration of Power method:
  $=|M|(1+\varepsilon) + (k+1)|r|$

Some Problems with Page Rank

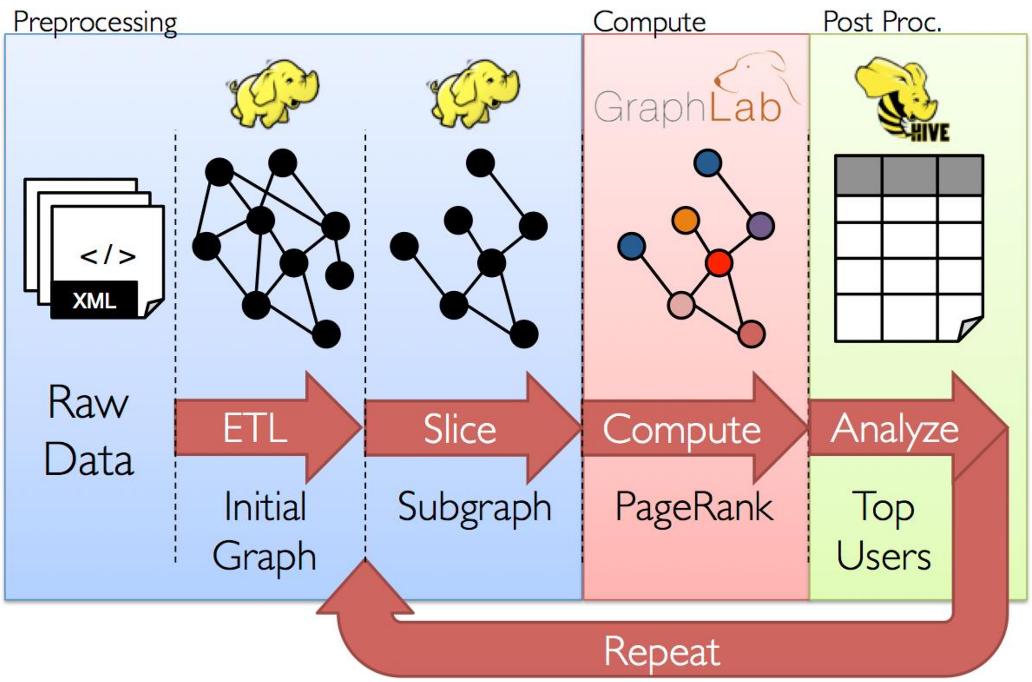
- Measures generic popularity of a page
  - Biased against topic-specific authorities
  - **Solution:** Topic-Specific PageRank
- Uses a single measure of importance
  - Other models of importance
  - **Solution:** Hubs-and-Authorities
- Susceptible to Link spam
  - Artificial link topographies created in order to boost page rank
  - **Solution:** TrustRank

GraphX

Graphs: Multiple Processing Steps

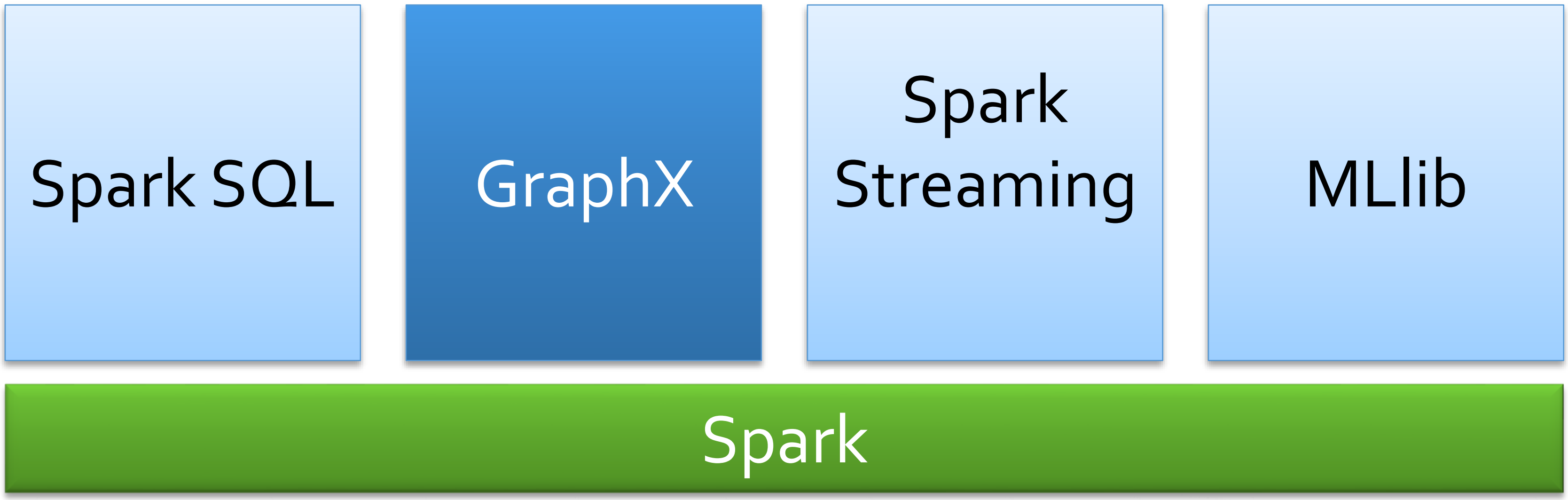
- Graph processing requires pre- and post-processing, or spanning multiple graphs

http://spark.apache.org/docs/latest/graphx-programming-guide.html



- Traditionally, multiple frameworks are combined
- Spark with **GraphX** allows integrated, efficient pipeline

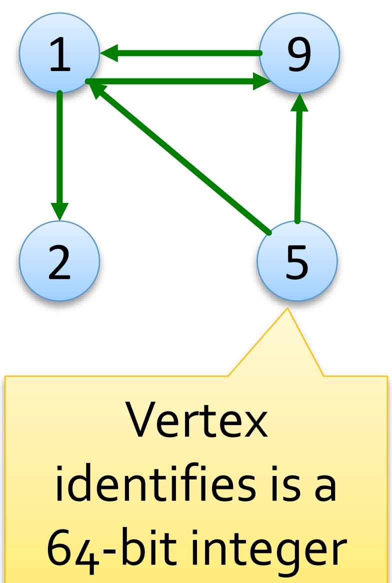
What is GraphX?



- Spark library / module: integrates graph analysis and processing in Spark
- Based on previous work on Pregel and GraphLab
- Bindings since Spark version 1.2
  - Only Scala

GraphX Data Structure

- The **property graph** is a directed multigraph with user-defined objects at vertices & edges



Vertex ID	Property (V)
1	("Dr. Evil", 39)
2	("Number 2", 45)
9	("Mini me", 12)
5	("Austin P.", 33)

SrcID	DstID	Property (E)
1	9	"takes care"
9	1	"loves"
1	2	"directs"
5	1	"chases"
5	9	"likes"

Graph Building - Scala

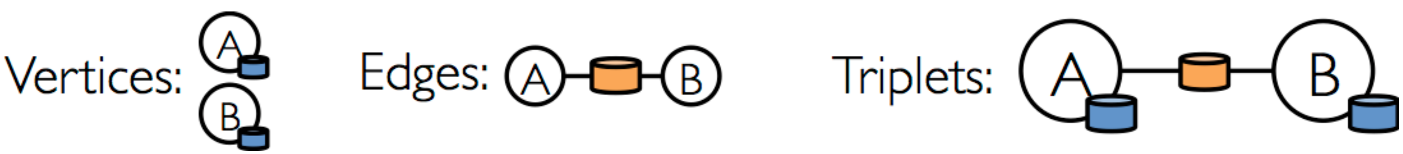
- Our graph would have the type signature:
  - val evilGraph: Graph[(String, Integer), String]
- Graph building in Scala
  - val figures: RDD[(VertexId, (String, Integer))] = sc.parallelize(
    Array((1L, ("**Dr. Evil**", **39**)), (2L, ("**Number 2**", **45**)),
    (9L, ("**Mini me**", **12**)), (5L, ("**Austin P.**", **33**))))
  - val relations: RDD[Edge[String]] = sc.parallelize(
    Array(Edge(1L, 9L, "**takes care**"), Edge(1L, 2L, "**directs**"),
    ...))
  - val defaultFigure = ("Mike Meyers", 0)
  - val evilGraph = Graph(figures, relations, defaultFigure)

Graph Views /1

- We obtain vertex and edge views by using graph.vertices and graph.edges respectively
  - // Count all users older than 35
  - graph.vertices.filter { case (id, (name, age)) => age > 35 }.count
  - // Count all the edges where src > dst
  - graph.edges.filter{ e => e.srcId > e.dstId }.count

Graph Views - Triplets/2

- We also have a **triplet** view: for each edge e = (src, dst), a triplet comprises properties of e, src, dst
  - “The EdgeTriplet class extends the Edge class by srcAttr and dstAttr = source and destination properties“



http://spark.apache.org/docs/latest/graphx-programming-guide.html

- Example: print all relationships

```
val facts: RDD[String] =  graph.triplets.map( triplet =>
  triplet.srcAttr._1+" " + triplet.attr + " for "+ triplet.dstAttr._1)
facts.collect.foreach( println(_) )
```

Message Aggregation

- A core operation is **aggregateMessages**:
  - Phase 1 ("map"): for each triplet, generate a message
  - Phase 2 ("reduce"): any two messages for the same destination vertex V are reduced to a single message
  - The result contains (for each vertex) an aggregate message (after "reduce")
- Scala:

```
class Graph[VD, ED] {
  def aggregateMessages [Msg: ClassTag](
    sendMsg: EdgeContext[VD, ED, Msg] => Unit,
    mergeMsg: (Msg, Msg) => Msg,
    tripletFields: TripletFields = TripletFields.All)
   : VertexRDD[Msg] }
```

Message Aggregation - Example /1

- Compute the average age of older followers of each user

```
// Create a random graph with "age" as the vertex property
val graph: Graph[Double, Int] =
        GraphGenerators.logNormalGraph(sc, numVertices = 100).
                                  mapVertices( (id, _) => id.toDouble )

// Compute the number of older followers and their total age
val olderFollowers: VertexRDD[(Int, Double)] =
                    graph.aggregateMessages[(Int, Double)](
 triplet => {
  // Map function (sendMsg)
  if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst(1, triplet.srcAttr)
  }
 },
 // Reduce function (mergeMsg): add counter and age
 (a, b) => (a._1 + b._1, a._2 + b._2)
)
```

Message Aggregation - Example /2

- Compute the average age of older followers of each user (cont.)

```
...
// Divide total age by number of older followers (over all follower vertices) to get average age of older followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
 olderFollowers.mapValues(
                    (id, value) => value match {
                              case (count, totalAge) => totalAge / count } )
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

Many Other GraphX Operators

```
// Information about the Graph
val numEdges: Long
val numVertices: Long
val inDegrees: VertexRDD[Int]
val outDegrees: VertexRDD[Int]
val degrees: VertexRDD[Int]

// Views of the graph as collections
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
val triplets: RDD[EdgeTriplet[VD, ED]]

// Transform vertex and edge attributes
def mapVertices [VD2](map: (VertexID,
VD) => VD2): Graph[VD2, ED]
def mapEdges [ED2](map: Edge[ED] =>
ED2): Graph[VD, ED2]
def mapTriplets [ED2](map:
EdgeTriplet[VD, ED] => ED2): Graph[VD,
ED2]

// Modify the graph structure
def reverse: Graph[VD, ED]
def subgraph( …) : Graph[VD, ED]
def mask [VD2, ED2](other: Graph[VD2,
ED2]): Graph[VD, ED]
def groupEdges (merge: (ED, ED) => ED):
Graph[VD, ED]

….
// Basic graph algorithms
def pageRank (tol: Double, resetProb:
Double = 0.15): Graph[Double, Double]
 def connectedComponents():
Graph[VertexID, ED]
def triangleCount(): Graph[Int, ED]
def stronglyConnectedComponents
(numIter: Int): Graph[VertexID, ED]
```

Thank you.

Questions?