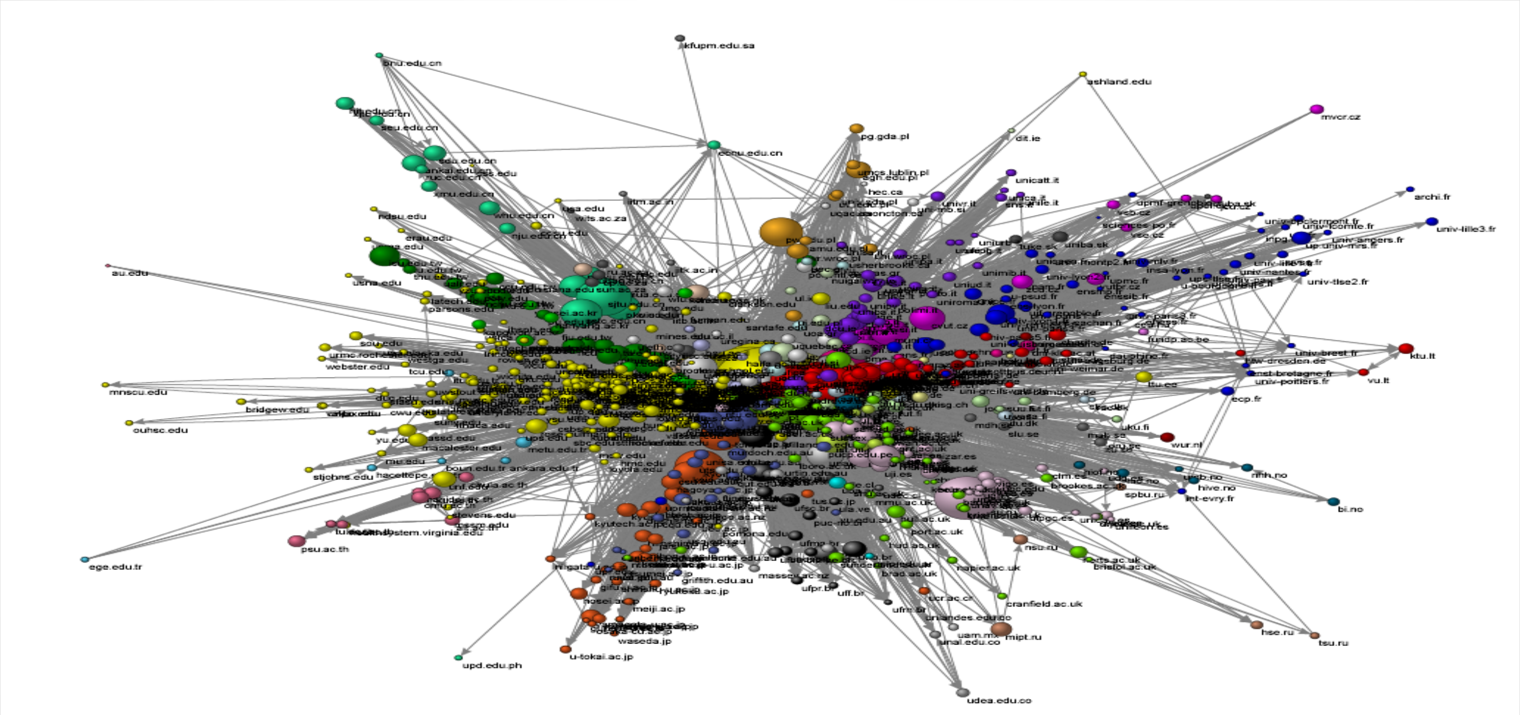
Web Graph Analysis: Bring Order to the Web(Implementation)



Presented by:

Sahil Mutneja(B11031) Ankit Srivastava(B11108) Sachin Tyagi(B11030)

Introduction to HDFS

Big Data

Wikipedia Definition:

In information technology, big data is a loosely-defined term used to describe data sets so ***large*** and ***complex*** that they become ***awkward*** to work with using on-hand database management tools.

How Big is Big Data?

- 2008: Google processed 20 PB a day
- 2009: Facebook had 2.5 PB user data + 15 TB/day
- 2009: eBay had 6.5 PB user data + 50 TB/day
- 2011: Yahoo! had 180-200 PB of data
- 2012: Facebook ingests 500 TB/day

Hadoop Distributed File System

HDFS is a fault tolerant and self-healing distributed file system designed to turn a cluster of industry standard servers into a massively scalable pool of storage. Developed specifically for large-scale data processing workloads where scalability, flexibility and throughput are critical, HDFS accepts data in any format regardless of schema, optimizes for high bandwidth streaming, and scales to proven deployments of 100PB and beyond.

Key Features

- **Accessible**
  - Hadoop runs on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- **Robust**
  - As Hadoop is intended to run on commodity hardware, It is architected with the assumption of frequent hardware malfunctions. It can gracefully handle most such failures.
- **Scalable**
  - Hadoop scales linearly to handle larger data by adding more nodes to the cluster.
- **Simple**
  - Hadoop allows users to quickly write efficient parallel code.

HDFS Scaling Out

Performs a task in
45 minutes

Performs a task in
~45/4 minutes

Hadoop Modes of Operation

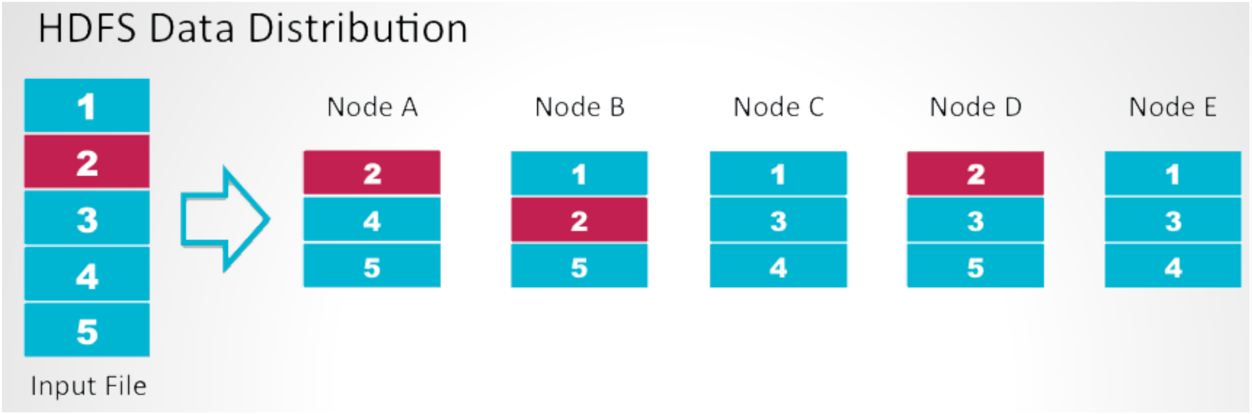
- Standalone (or local) mode
  - There are no daemons running and everything runs in a single JVM (Java Virtual Machine). Standalone mode is suitable for running MapReduce programs during development, since it is easy to test and debug them.
- Pseudo-distributed mode
  - The Hadoop daemons run on the local machine, thus simulating a cluster on a small scale.
- Fully distributed mode
  - The Hadoop daemons run on a cluster of machines.

Master-Slave Architecture

- HDFS has a master-slave architecture.
- The **master node** or the **name node** governs the cluster. It takes care of tasks and resource allocation.
- It stores all the metadata related to file breakage, block storage, block replication and task execution status.
- The **slave nodes** or the **data nodes** are the one which stores all the data blocks and perform task executions
- **Tasktracker** is the program which runs on each individual data node and monitors the task execution over each node.
- **Jobtracker** runs on name node and monitors the complete job execution.

HDFS file Distribution

- ❖ Name node stores metadata related to:
  - ➢ File split
  - ➢ Block allocation
  - ➢ Task allocation
- ❖ Each file is split into data blocks. Default size is 64 MB
- ❖ Each data block is replicated on different data node. The replication factor in configurable. Default value is 3



Data in HDFS is replicated across multiple nodes(factor of 3) for getting performance and data protection.

3 main configuration files

- Core-site.xml
  - Contains configuration information that overrides the default core Hadoop properties
- Mapred-site.xml
  - Contains configuration information that overrides the default core Mapreduce properties
  - Also defines the host and port that the MapReduce job tracker runs at
- Hdfs-site.xml
  - Mainly, to set the block replication factor

Introduction to Spark

- Open Source data analytics cluster computing framework
- Spark is not tied to the two-stage MapReduce paradigm, and promises performance up to 100 times faster than Hadoop MapReduce, for certain applications.
- A low latency cluster computing system
- Spark provides primitives for in-memory cluster computing that allows user programs to load data into a cluster's memory and query it repeatedly, making it well suited to machine learning algorithms.

How does it work?

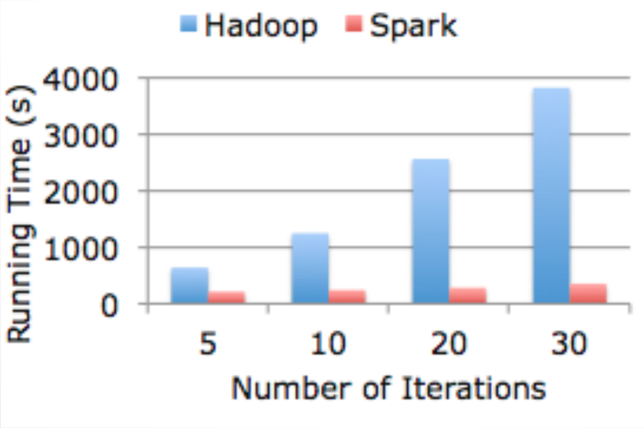
- Uses in memory cluster computing
- Memory access faster than disk access
- Has API's written in
  - Scala
  - Java
  - Python
- Can be accessed from Scala and Python shells
- Currently an Apache incubator project

Benefits

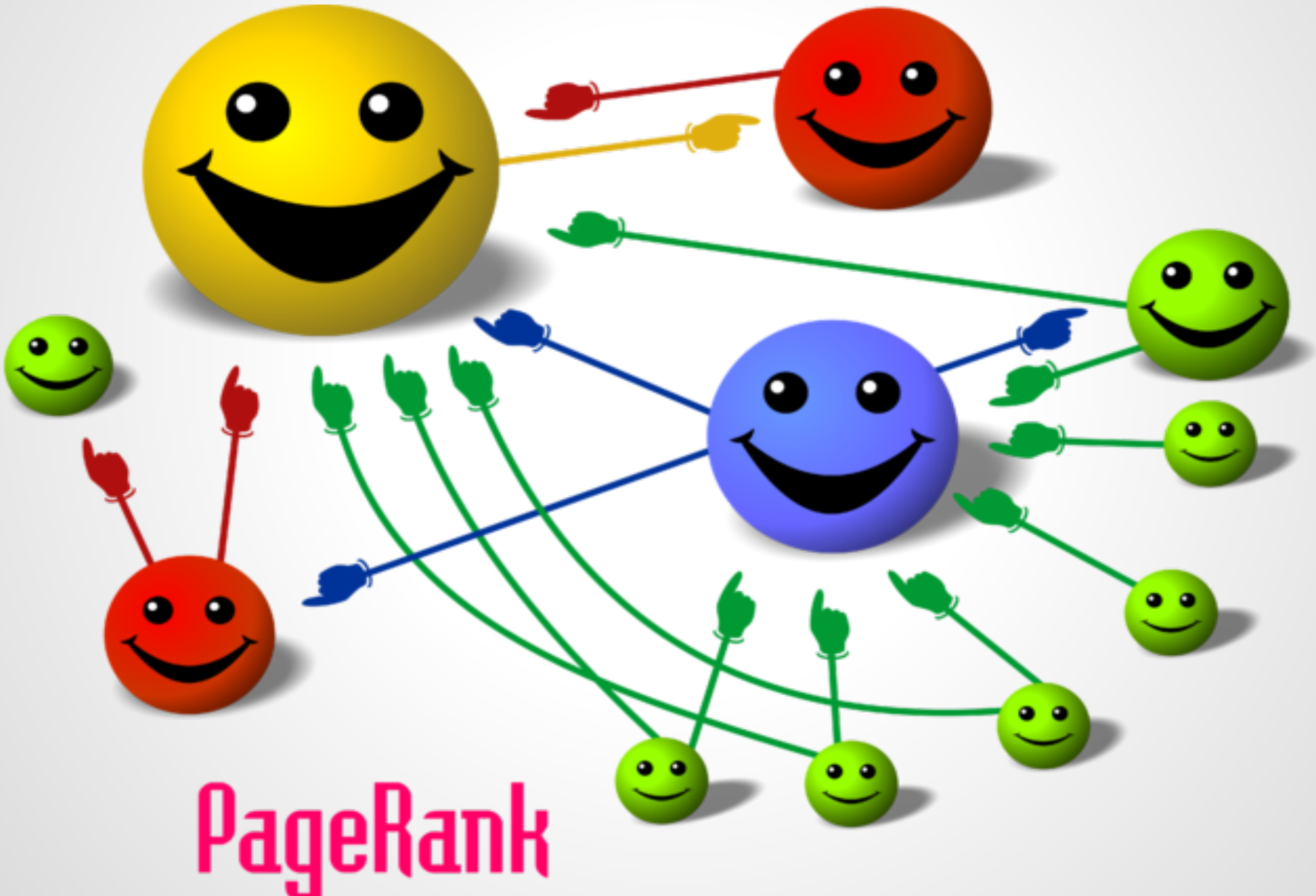
- Scales to very large clusters
- Uses in memory processing for increased speed
- High Level API's
  - Java, Scala, Python
- Low latency shell access

Spark vs Hadoop

- Hadoop and Spark are completely different things.
- Hadoop is a Distributed file system and Spark is an in-memory computing engine.
- Often when you're using Spark, you want to have HDFS (the Hadoop distributed file system) installed as well.



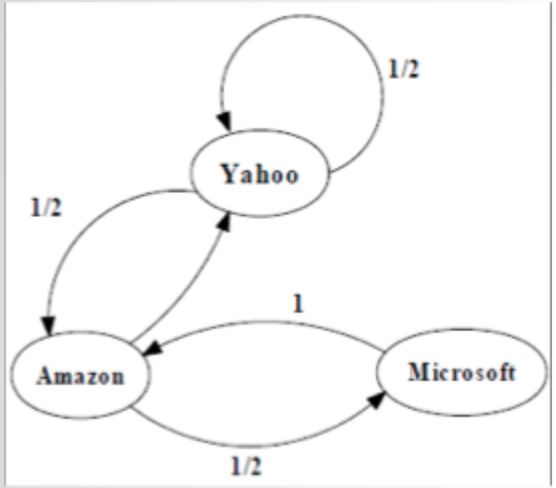
Implementation of PageRank



Implementing PageRank using MapReduce-Pseudocode

```
map( key : [url, pagerank], value : outlink_list )
   for each outlink in outlink_list :
      emit( key : outlink, value : pagerank / sizeof(outlink_list) )
   emit( key : url, value : outlink_list )

reducer( key : url, value : list_pr_or_urls )
   outlink = [ ]
   pagerank = 0
   for each pr_or_urls in list_pr_or_urls :
      if is_list(pr_or_urls)
         outlink = pr_or_urls
      else
         pagerank += pr_or_urls
   pagerank = 1 - DAMPING_FACTOR + ( DAMPING_FACTOR * pagerank )
   emit( key : [url, pagerank], value : outlink_list )
```

An Example of PageRank



Input File given to the Program (1::Yahoo, 2:: Amazon, 3::Microsoft)

1 1
1 2
2 1
2 3
3 2

Here first number represents the *from* Link and second the *to* Link.

Showing results for Iteration No. 1

Loads all URL's from input file and initialize their neighbors

[(u'1', [u'2', u'1']), (u'3', [u'2']), (u'2', [u'1', u'3'])]

Loads all URLs with other URLs link to from input file and initialize ranks of them to one.

[(u'1', 1.0), (u'3', 1.0), (u'2', 1.0)]

Calculates URL contributions to the rank of other URLs.

[(u'2', 0.5), (u'1', 0.5), (u'2', 1.0), (u'1', 0.5), (u'3', 0.5)]

Re-calculates URL ranks based on neighbor contributions.

[(u'1', 1.0), (u'3', 0.575), (u'2', 1.425)]

PageRank- Code

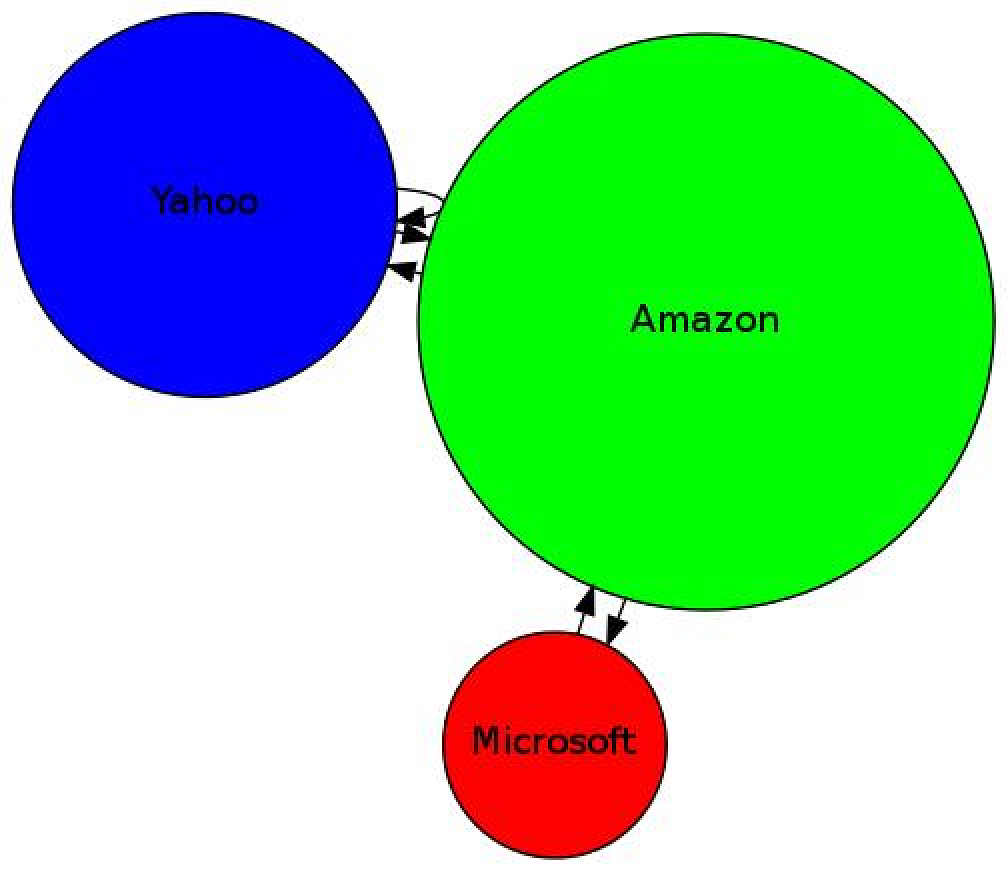
```
#Showing the snippet of the main function
links=lines.map(lambda urls: parseNeighbors(urls)).distinct().groupByKey().
cache()
ranks = links.map(lambda (url, neighbors): (url, 1.0))
for iteration in xrange(int(sys.argv[3])):
  contribs = links.join(ranks).flatMap(lambda (url, (urls, rank)):
    computeContribs(urls, rank))
  ranks = contribs.reduceByKey(add).mapValues(lambda rank: rank * 0.85 + 0.15)

def computeContribs(urls, rank):
  num_urls = len(urls)
  for url in urls: yield (url, rank / num_urls)
```

Similarly, iterating 15 times in the same manner we get

1 has rank: 1.14474367709

3 has rank: 0.657800480237

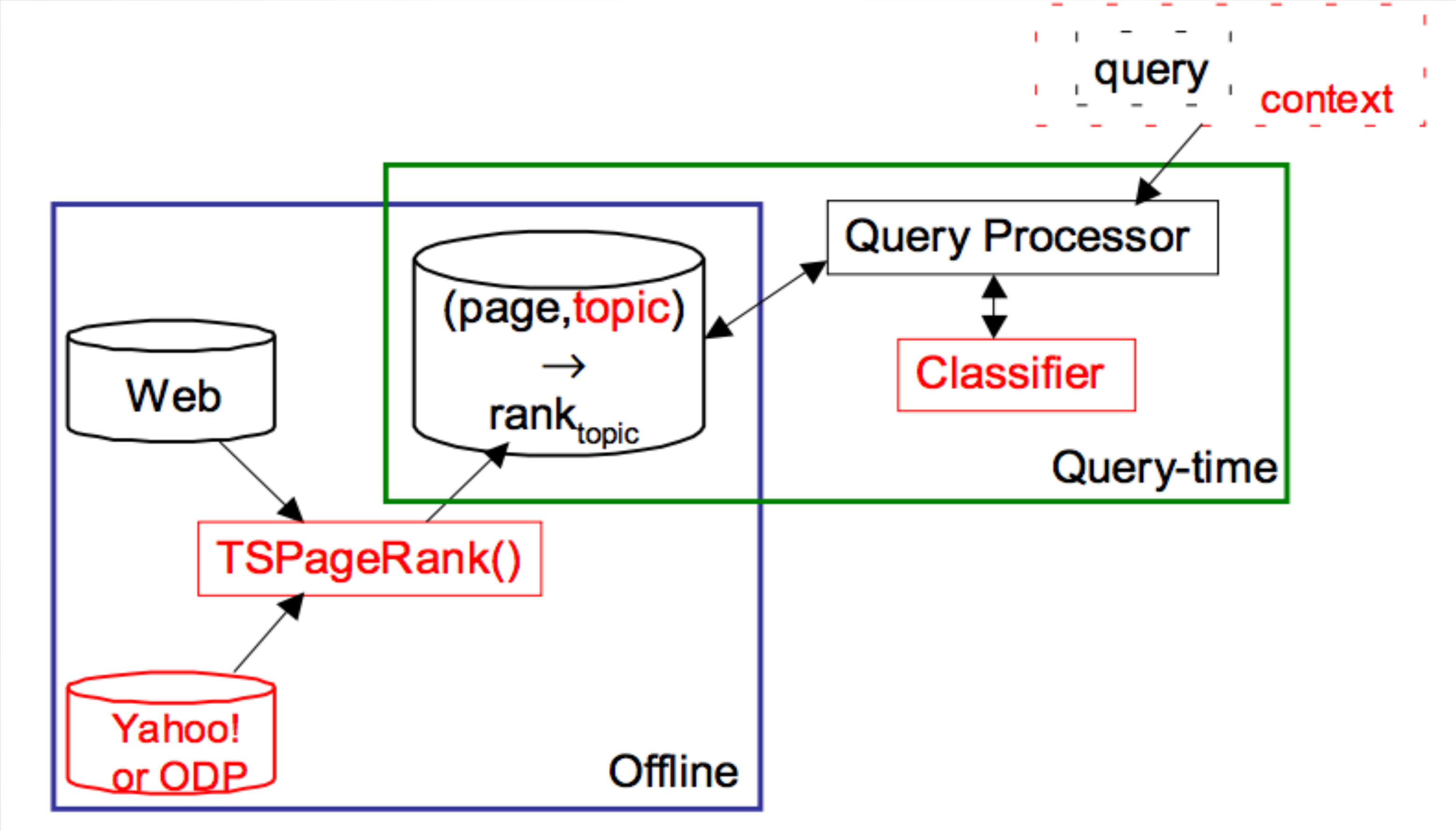
2 has rank: 1.19745584268


Yahoo
Amazon
Microsoft

Time Analysis for a single node

No. of Link to Link data	Time(s)	Size of Input file
1000	16sec	9.23KB
37500	30sec	365.65KB
40,00,000	12min59sec	51MB
70,50,000	17min21sec	94MB
1,01,00000	26min17sec	138MB

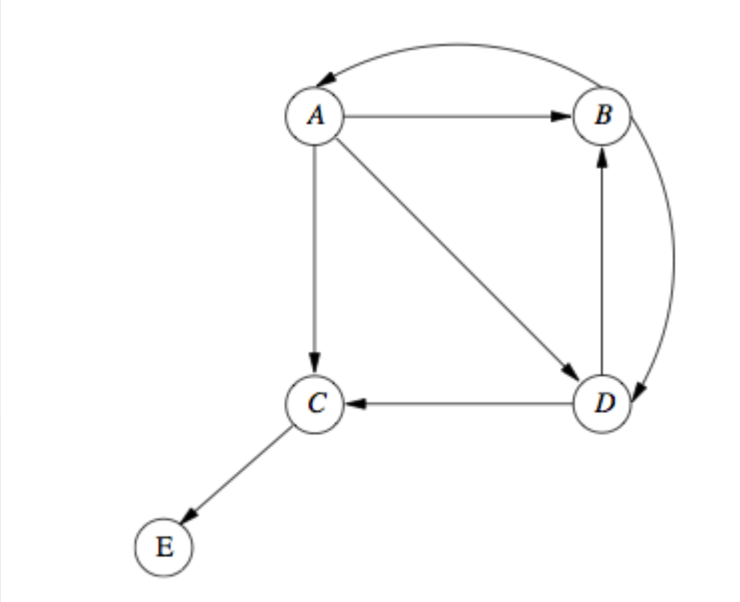
Implementation of Topic Sensitive PageRank



Topic sensitive PageRank Algorithm - Pseudocode

```
procedure PageRank(G,iteration,teleport_set)     #G:inlink file, iteration: # of iteration
   d <- 0.85          #damping factor
   oh <- G            #get out link count hash from G
   ih <- G            #get inlink hash from G
   N <- G             #get number of pages from G
   for all p in the graph do
      opg[p] <- 1          #initialise PageRank
   end for
   while iteration > 0 do
      for all p in the graph do
         if p ϵ teleport_set
            ngp[p] <- (1-d)        #get PageRank from random jump
         for all ip in in ih[p] do
            ngp[p] <- ngp[p] + (d*opg[ip])/oh[ip]   #get PageRank from inlinks
         end for
      end for
      opg <- ngp                                     #update PageRank
      iteration <- iteration - 1
   end while
end procedure
```

An Example of TSPR Algorithm



Input File given to the Program
1 2
1 3
1 4
2 1
2 4
3 5
4 2
4 3
First no. represents the *from* Link and second the *to* Link.

Topics file given to the Program
2
3
For a given searched query, this is the list of sensitive links which will be given some preference.

Showing results for Iteration No. 1

Loads all URL's from input file and initialize their neighbors

[(u'1', [u'4', u'2', u'3']), (u'3', [u'5']), (u'2', [u'1', u'4']), (u'4', [u'3', u'2'])]

Loads all URLs with other URLs link to from input file and initialize ranks of them to one.

[(u'1', 1.0), (u'3', 1.0), (u'2', 1.0), (u'4', 1.0)]

Calculates URL contributions to the rank of other URLs.

[(u'4', 0.33), (u'2', 0.33), (u'3', 0.33), (u'5', 1.0), (u'1', 0.5), (u'4', 0.5), (u'3', 0.5), (u'2', 0.5)]

Re-calculates URL ranks based on neighbor contributions.

[(u'1', 0.425), (u'3', 0.70), (u'2', 0.70), (u'5', 0.85), (u'4', 0.70)]

Adding the damping factor to the topics set only

[(u'1', 0.425), (u'3', 0.85), (u'2', 0.858), (u'5', 0.85), (u'4', 0.70)]

Topic Sensitive PageRank- Code

```
#Showing the snippet of the main function
for iteration in xrange(int(sys.argv[3])):
   contribs = links.join(ranks).flatMap(lambda (url, (urls, rank)):
      computeContribs(urls, rank))
ranks = contribs.reduceByKey(add).mapValues(lambda rank: rank * 0.85)
new_ranks = [(v[0], v[1]) for i, v in enumerate(ranks.collect())]
for number, i in enumerate(new_ranks):
   if i[0] in topics.collect():
      new_ranks[number] = (i[0], i[1]+0.15)
ranks = sc.parallelize(new_ranks)
```

Similarly, iterating 15 times in the same manner we get

1 has rank: 0.0988970037809

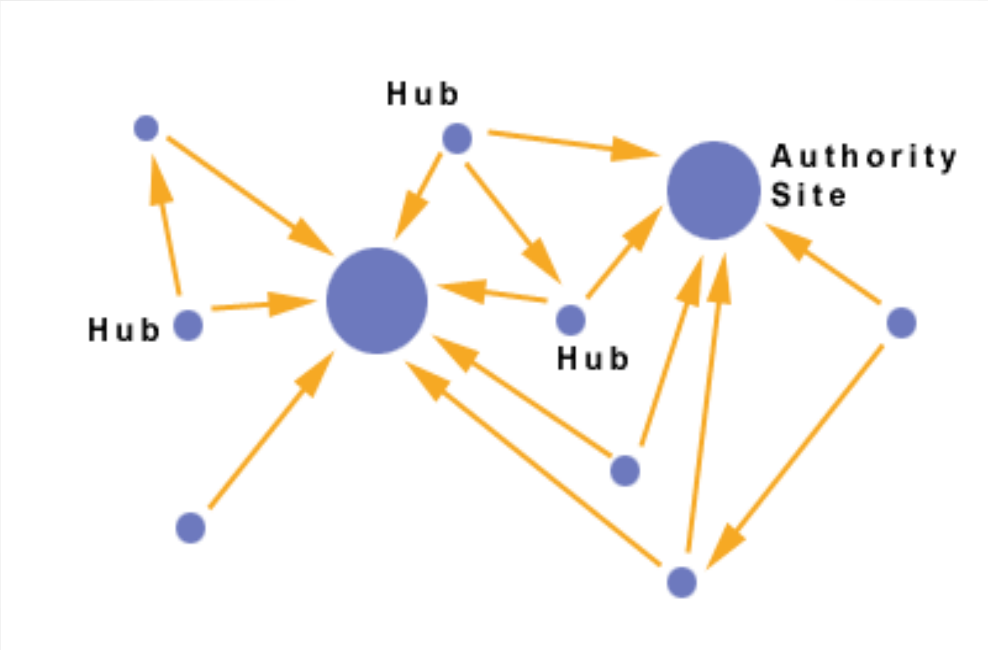
3 has rank: 0.232265784501

2 has rank: 0.232265784501

5 has rank: 0.197794007562

4 has rank: 0.127002345992

Implementation of Hubs and Authorities- HITS(Hyperlink-Induced Topic Search)



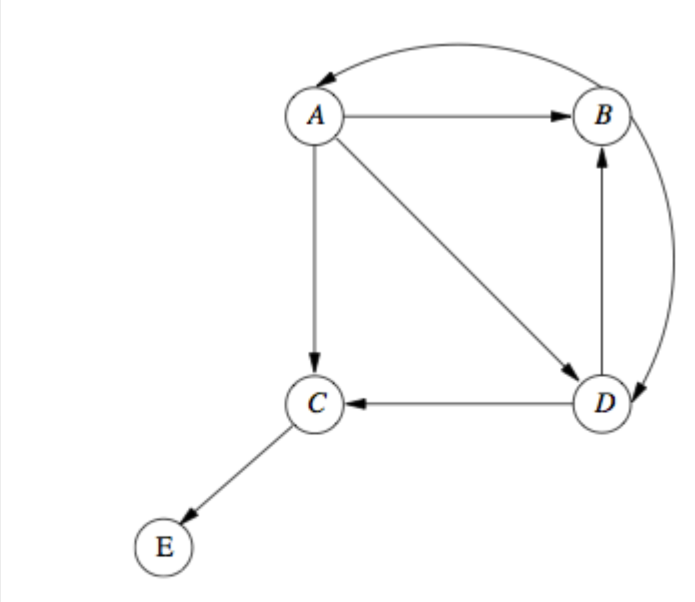
HITS - Pseudocode

```
G := set of pages
for each page p in G do
  p.auth = 1    // p.auth is the authority score of the page p
  p.hub = 1    // p.hub is the hub score of the page p
function HubsAndAuthorities(G)
  for step from 1 to k do    // run the algorithm for k steps
    norm = 0
    for each page p in G do     // update all authority values first
      p.auth = 0
      for each page q in p.incomingNeighbors do    // p.incomingNeighbors is the set of pages that link to p
        p.auth += q.hub
        norm += square(p.auth)    // calculate the sum of the squared auth values to normalise
      norm = sqrt(norm)
      for each page p in G do     // update the auth scores
        p.auth = p.auth / norm     // normalise the auth values
      norm = 0
      for each page p in G do     // then update all hub values
        p.hub = 0
        for each page r in p.outgoingNeighbors do    // p.outgoingNeighbors is the set of pages that p links to
          p.hub += r.auth
          norm += square(p.hub)    // calculate the sum of the squared hub values to normalise
      norm = sqrt(norm)
      for each page p in G do     // then update all hub values
        p.hub = p.hub / norm      // normalise the hub values
```

Computing auth and hub from the hub and auth of a link resp.

```
def computeAuth(urls, hub):
    """Calculates hub contributions to the auth of other URLs."""
    """Outgoing list of a link and its hub is given"""
    for url in urls: yield (url, hub)

def computeHub(urls, auth):
    """Calculates auth contributions to the hub of other URLs."""
    """Incoming list of a link and its auth is given"""
    for url in urls: yield (url, auth)
```

An Example of Hits Algorithm



Input File given to the Program

1 2
1 3
1 4
2 1
2 4
3 5
4 2
4 3

Here first number represents the *from* Link and second the *to* Link.

Showing results for the Iteration No. 1

Outgoing List after loading URLs from the input file

[(u'1', [u'4', u'2', u'3']), (u'3', [u'5']), (u'2', [u'1', u'4']), (u'4', [u'3', u'2'])]

Incoming List

[(u'1', [u'2']), (u'3', [u'1', u'4']), (u'2', [u'1', u'4']), (u'5', [u'3']), (u'4', [u'1', u'2'])]

Initialising the auths of every URL

[(u'1', 1.0), (u'3', 1.0), (u'2', 1.0), (u'5', 1.0), (u'4', 1.0)]

Initialising the hub of every URL

[(u'1', 1.0), (u'3', 1.0), (u'2', 1.0), (u'4', 1.0)]

HITS- Code

```
for iteration in xrange(int(sys.argv[3])):
    auth_contribs = out_links.join(hubs).flatMap(lambda (url, (urls, hub)):
        computeAuth(urls, hub))
    auths = auth_contribs.reduceByKey(add)
    max_value = max(auths.collect(), key=lambda x:x[1])[1]
    auths = auths.mapValues(lambda rank: rank/(max_value))
    hub_contribs = in_links.join(auths).flatMap(lambda (url, (urls, auth)):
        computeHub(urls, auth))
    hubs = hub_contribs.reduceByKey(add)
    max_value = max(hubs.collect(), key=lambda x:x[1])[1]
    hubs = hubs.mapValues(lambda rank:rank/(max_value))
```

Finding Contribution to the auth of all links present in the outgoing list of a link whose hub is given

[(u'4', 1.0), (u'2', 1.0), (u'3', 1.0), (u'5', 1.0), (u'1', 1.0), (u'4', 1.0), (u'3', 1.0), (u'2', 1.0)]

Reducing the obtained auth values

[(u'1', 1.0), (u'3', 2.0), (u'2', 2.0), (u'5', 1.0), (u'4', 2.0)]

Normalising the obtained values via max value, Hence finalising the first iteration of the auth values

[(u'1', 0.5), (u'3', 1.0), (u'2', 1.0), (u'5', 0.5), (u'4', 1.0)]

```
# Finding Contribution to the hub of all links present in the incoming list of a link whose new auth is given
[(u'2', 0.5), (u'1', 1.0), (u'4', 1.0), (u'1', 1.0), (u'4', 1.0), (u'3', 0.5), (u'1', 1.0), (u'2', 1.0)]
# Reducing the obtained auth values
[(u'1', 3.0), (u'3', 0.5), (u'2', 1.5), (u'4', 2.0)]
# Normalising the obtained values via max value, Hence finalising the first iteration of the auth values
[(u'1', 1.0), (u'3', 0.16), (u'2', 0.5), (u'4', 0.66)]
```

Similarly, iterating 15 times in the same manner we get

1 has auth: 0.128348981338

3 has auth: 0.614953988757

2 has auth: 0.614953988757

5 has auth: 8.4371932348e-11

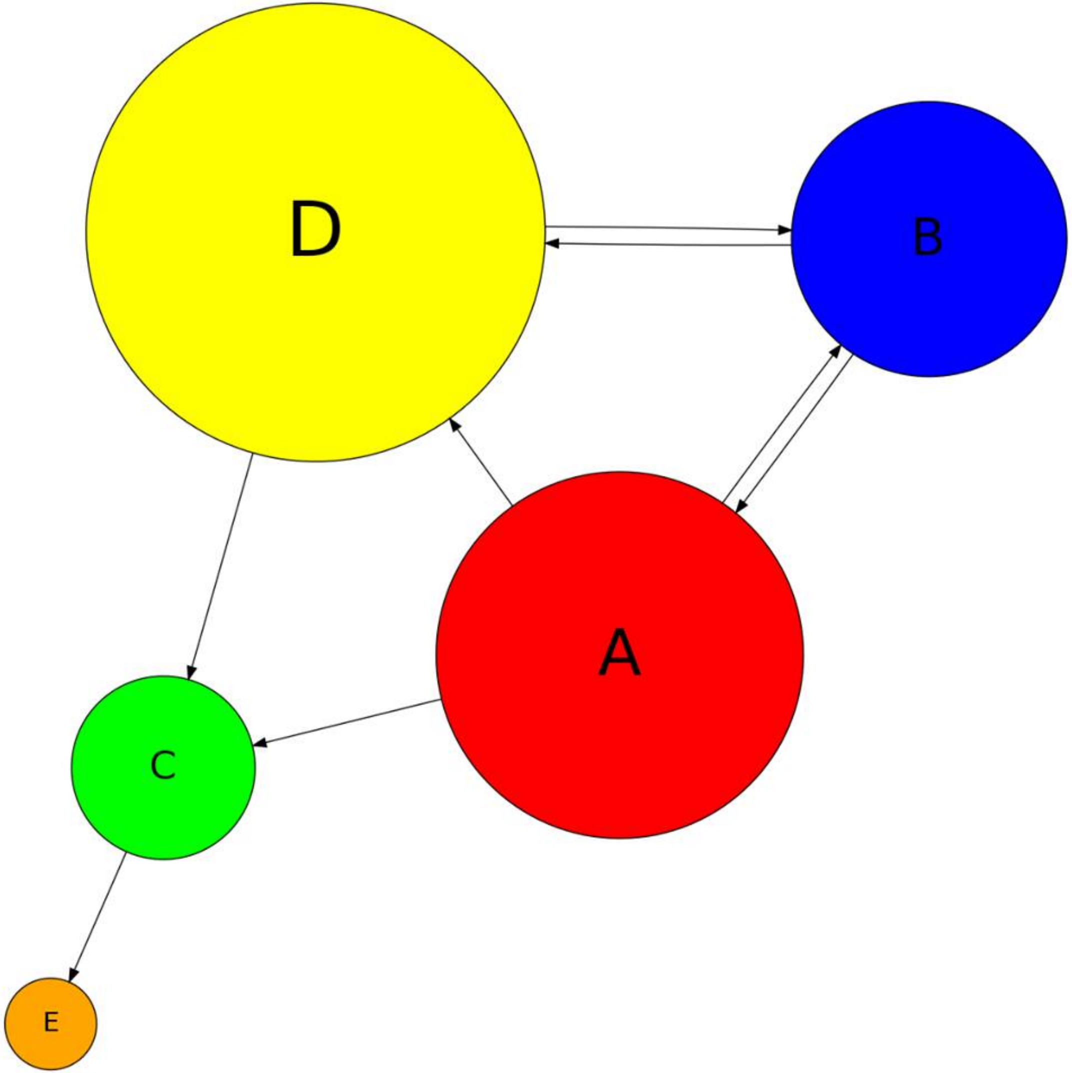
4 has auth: 0.486606389769

1 has hub: 1.0

3 has hub: 4.91530592205e-11

2 has hub: 0.358258213755

4 has hub: 0.716514816862


D
B
A
C
E