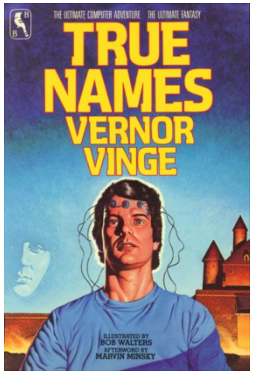


In the once upon a time days of the First Age of Magic, the prudent sorcerer regarded his own true name as his most valued possession but also the greatest threat to his continued good health, for--the stories go--once an enemy, even a weak unskilled enemy, learned the sorcerer's true name, then routine and widely known spells could destroy or enslave even the most powerful. As times passed, and we graduated to the Age of Reason and thence to the first and second industrial revolutions, such notions were discredited. Now it seems that the Wheel has turned full circle (even if there never really was a First Age) and we are back to worrying about true names again:

The first hint Mr. Slippery had that his own True Name might be known--and, for that matter, known to the Great Enemy--came with the appearance of two black Lincolns humming up the long dirt driveway ... Roger Pollack was in his garden weeding, had been there nearly the whole morning.... Four heavy-set men and a hard-looking female piled out, started purposefully across his well-tended cabbage patch.…

This had been, of course, Roger Pollack's great fear. They had discovered Mr. Slippery's True Name and it was Roger Andrew Pollack TIN/SSAN 0959-34-2861.

Recap: soft joins/similarity joins

Input: Two Different Lists of Entity Names

Abraham Lincoln Birthplace NHS Acadia NP Adams NHS Agate Fossil Beds NM Alagnak Wild River Alaska Public Lands Inf. Center Alibates Flint Quarries NM Allegheny Portage Railroad NHS American Memorial Park Amistad NRA Andersonville NHS Andrew Johnson NHS Aniakchak NM & NPRES Antietam NB Apostle Islands NL Appalachian National Scenic Trail Appomattox Courthouse NHP Arches NP Arkansas Post NM ...	Acadia NP Allegheny Portage Railroad NHS American Memorial Park Amistad NRA Andersonville NHP Aniakchak NM Antietam NB Apostle Islands NL Appomattox Court House NHP Arches NP Arkansas Post N. Mem. Assateague Island NS Aztec Ruins NM Badlands NP Bandelier NM Bent's Old Fort NHS Bering Land Bridge N. Preserve Big Bend NP Big Cypress N. Preserve ...

Recap: soft joins/similarity joins

Output: Pairs of Names Ranked by Similarity

identical

Chickamauga & Chattanooga NMP:d445	Chickamauga & Chattanooga NMP:d72
George Washington Carver NM:d499	George Washington Carver NM:d153
Salinas Pueblo Missions NM:d597	Salinas Pueblo Missions NM:d329
Florissant Fossil Beds NM:d473	Florissant Fossil Beds NM:d116
Hagerman Fossil Beds NM:d517	Hagerman Fossil Beds NM:d177
Gila Cliff Dwellings NM:d502	Gila Cliff Dwellings NM:d156
Booker T. Washington NM:d423	Booker T. Washington NM:d38

similar

…

Obed Wild & Scenic River:d570	Obed Wild and Scenic River:d283
Andersonville NHP:d401	Andersonville NHS:d11
Sitka NHP:d606	Sitka NHS:d342
Bering Land Bridge N. Preserve:d413	Bering Land Bridge NPRES:d26
Sequoia & Kings Canyon NP:d603	Sequoia and Kings Canyon NP:d339
Glacier Bay NP & Preserve:d643	Glacier Bay NP & NPRES:d157
NP of American Samoa:d561	National Park Of American Samoa:d267
Kalaupapa NHS:d538	Kalaupapa NHP:d210

less similar

…

Lake Mead NRA:d545	Lake Mead NRA (Nevada):d224
Upper Delaware Scenic & Rec. River:d617	Upper Delaware Scenic & Recreational River:d368

Example: soft joins/similarity joins

Output: Pairs of Names Ranked by Similarity

A surprisingly good similarity score is TFIDF cosine distance.

- Mismatches on frequent terms ("&" vs "and", "N.", "Preserve", "NHP", …) are discounted
- Matches on rare term ("Kalaupapa", "Samoa") are rewarded.

…

Obed Wild & Scenic River:d570	Obed Wild and Scenic River:d283
Andersonville NHP:d401	Andersonville NHS:d11
Sitka NHP:d606	Sitka NHS:d342
Bering Land Bridge N. Preserve:d413	Bering Land Bridge NPRES:d26
Sequoia & Kings Canyon NP:d603	Sequoia and Kings Canyon NP:d339
Glacier Bay NP & Preserve:d643	Glacier Bay NP & NPRES:d157
NP of American Samoa:d561	National Park Of American Samoa:d267
Kalaupapa NHS:d538	Kalaupapa NHP:d210

…

Lake Mead NRA:d545	Lake Mead NRA (Nevada):d224
Upper Delaware Scenic & Rec. River:d617	Upper Delaware Scenic & Recreational River:d368

Softjoin Example - 1

~ means "similar to"

FROM top500,hiTech SELECT * WHERE top500.name~hiTech.name

top500:	*hiTech:*
Abbott Laboratories	ACC CORP
Able Telcom Holding Corp	ADC TELECOMMUNICATION INC

Table VI. Pairs of Names from the Hoovers and Iontech Relations

√	Texas Instruments Incorporated	TEXAS INSTRUMENTS INC
√	The New York Times Company	NEW YORK TIMES CO
√	Campo Electronics, Appliances and Computers, Inc.	CAMPO ELECTRONICS APPLIANCES
√	Cascade Communications Corp.	CASCADE COMMUNICATION
√	The McGraw-Hill Companies, Inc.	MCGRAW-HILL CO
√	U S WEST Communications Group	U S WEST INC
×	Silicon Valley Group, Inc.	SILICON VALLEY RESEARCH INC
×	The Reynolds and Reynolds Company	REYNOLDS & REYNOLDS CO
√	InTime Systems International, Inc.	INTIME SYSTEMS INTERNATIONAL I

A useful scalable similarity metric: IDF weighting plus cosine distance!

One solution: Soft (Similarity) joins

- A similarity join of two sets A and B is
  - an ordered list of triples (s_{ij},a_i,b_j) such that
    - a_i is from A
    - b_j is from B
    - s_{ij} is the *similarity* of a_i and b_j
    - the triples are in descending order
    - the list is either the top K triples by s_{ij} or ALL triples with $s_{ij}>L$ … or sometimes some approximation of these….

How well does TFIDF work?

- **Input:** query
- **Output:** ordered list of documents

1	✓	a_1	b_1
2	✓	a_2	b_2
3	×	a_3	b_3
4	✓	a_4	b_4
5	✓	a_5	b_5
6	✓	a_6	b_6
7	×	a_7	b_7
8	✓	a_8	b_8
9	✓	a_9	b_9
10	×	a_{10}	b_{10}
11	×	a_{11}	b_{11}
12	✓	a_{12}	b_{12}

Precision at K: G_K/K

Recall at K: G_K/G

G: # good pairings

G_K: # good pairings in first K

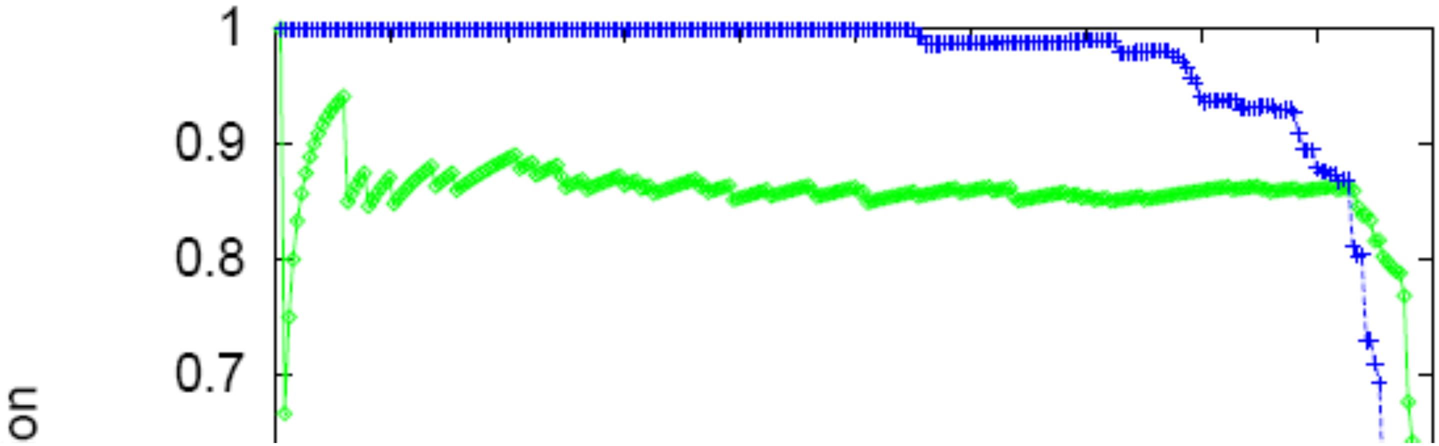


Table VI. Pairs of Names from the Hoovers and Iontech Relations

√	Texas Instruments Incorporated	TEXAS INSTRUMENTS INC
√	The New York Times Company	NEW YORK TIMES CO
√	Campo Electronics, Appliances and Computers, Inc.	CAMPO ELECTRONICS APPLIANCES
√	Cascade Communications Corp.	CASCADE COMMUNICATION
√	The McGraw-Hill Companies, Inc.	MCGRAW-HILL CO
√	U S WEST Communications Group	U S WEST INC
×	Silicon Valley Group, Inc.	SILICON VALLEY RESEARCH INC
×	The Reynolds and Reynolds Company	REYNOLDS & REYNOLDS CO
√	InTime Systems International, Inc.	INTIME SYSTEMS INTERNATIONAL I

Table V. Average Precision for Similarity Joins

Domain	Relations Joined	Average Precision
Movies	MovieLink/Review	100.0%
Animals	IntFact1/SWFact	100.0%
	IntFact2/FWSFact	99.6%
	IntFact3/NMFSFact	97.1%
	Endanger/ParkAnim	95.2%
Birds	IntBirdPic1/DonBirdPic	100.0%
	IntBirdPic2/MBRBirdPic	99.1%
	IntBirdMap/BirdMap	91.4%
	BirdCall/BirdList	95.8%
Businesses	Fodor/Zagrat	99.5%
	HooverWeb/Iontech	84.9%
National Parks	IntPark/Park	95.7%
Computer Games	Demo/AgeList	86.1%

There are refinements to TFIDF distance – eg ones that extend with soft matching at the token level (e.g., softTFIDF)

```
distance is '[JaroWinklerTFIDF:threshold=0.9]'
Pairs: 6806 Correct: 250
Matching time: 0.278
+   1    1.00 |           Agate Fossil Beds NM |           Agate Fossil Beds NM
+   2    1.00 |                    Big Bend NP |                    Big Bend NP
...
+ 194    1.00 |                    Gateway NRA |                    Gateway NRA
+ 195    0.99 |                Gulf Islands NS |                 Gulf Island NS
+ 196    0.99 |              Rainbow Bridge NM |             Rainbow Bridges NM
+ 197    0.98 | Whiskeytown Shasta Trinity NRA |    Whiskey-Shasta-Trinity NRA
+ 198    0.97 |                Capitol Reef NP |                Capital Reef NP
+ 199    0.95 |             Timpanogos Cave NM |            Timpanogas Caves NM
+ 200    0.94 |         War in the Pacific NHP |             War in Pacific NHP
+ 201    0.94 |      Chesapeake & Ohio Canal NHP | Chesapeake and Ohio Canal NHP
+ 203    0.92 |                     Saguaro NP |                     Saguaro NM
..
+ 210    0.88 |         Aniakchak NM & NPRES   |                   Aniakchak NM
+ 211    0.86 | National Park Of American Samoa|          NP of American Samoa
..
+ 224    0.76 |       Pu'uhonua a Honaunau NHP |     Pu'uohonua O Honaunau NHP
+ 225    0.75 |      Bering Land Bridge NPRES   | Bering Land Bridge N. Preserve
+ 226    0.75 |      Yukon Charley Rivers NPRES | Yukon-Charley Rivers N. Preserve
...
+ 241    0.69 | Wolf Trap Farm Park for the Performing Arts
                                              |            Wolf Trap Farm Park
+ 242    0.69 | Fredericksburg and Spotsylvania County Battlefields Memorial NMP
                                              | Fredericksburg & Spotsylvania NMP
+ 243    0.69 |           Great Smoky Mtn. NP |      Great Smoky Mountains NP
+ 245    0.67 |             Mount Rushmore NM |       Mount Rushmore N. Mem.
+ 246    0.67 |             Chattahoochee NSR |      Chattahoochee River NRA
...
```

```
distance is '[JaroWinklerTFIDF:threshold=0.9]'
Pairs: 6806 Correct: 250
Matching time: 0.278
+   1     1.00 |             Agate Fossil Beds NM |            Agate Fossil Beds NM
+   2     1.00 |                      Big Bend NP |                     Big Bend NP
...
+ 194     1.00 |                      Gateway NRA |                     Gateway NRA
+ 195     0.99 |                  Gulf Islands NS |                  Gulf Island NS
+ 196     0.99 |                Rainbow Bridge NM |              Rainbow Bridges NM
+ 197     0.98 | Whiskeytown Shasta Trinity NRA |    Whiskey-Shasta-Trinity NRA
+ 198     0.97 |                  Capitol Reef NP |                 Capital Reef NP
+ 199     0.95 |               Timpanogos Cave NM |             Timpanogas Caves NM
+ 200     0.94 |            War in the Pacific NHP |           War in Pacific NHP
+ 201     0.94 |     Chesapeake & Ohio Canal NHP | Chesapeake and Ohio Canal NHP
+ 203     0.92 |                       Saguaro NP |                     Saguaro NM
..
+ 210     0.88 |           Aniakchak NM & NPRES   |                   Aniakchak NM
+ 211     0.86 | National Park Of American Samoa|            NP of American Samoa
..
+ 224     0.76 |         Pu'uhonua a Honaunau NHP |      Pu'uohonua O Honaunau NHP
+ 225     0.75 |        Bering Land Bridge NPRES  | Bering Land Bridge N. Preserve
+ 226     0.75 |        Yukon Charley Rivers NPRES | Yukon-Charley Rivers N. Preserve
...
+ 241     0.69 | Wolf Trap Farm Park for the Performing Arts
                                                  |              Wolf Trap Farm Park
+ 242     0.69 | Fredericksburg and Spotsylvania County Battlefields Memorial NMP
                                                  | Fredericksburg & Spotsylvania NMP
+ 243     0.69 |             Great Smoky Mtn. NP |      Great Smoky Mountains NP
+ 245     0.67 |               Mount Rushmore NM |        Mount Rushmore N. Mem.
+ 246     0.67 |               Chattahoochee NSR |       Chattahoochee River NRA
...
```

SOFT JOINS WITH TFIDF: HOW?

Rocchio's algorithm

Many variants of these formulae

$$DF(w) = \#\text{ different docs } w \text{ occurs in}$$

$$TF(w,d) = \#\text{ different times } w \text{ occurs in doc } d$$

$$IDF(w) = \frac{|D|}{DF(w)}$$

$$u(w,d) = \log(TF(w,d)+1) \cdot \log(IDF(w))$$

...as long as *u(w,d)=0* for words not in *d!*

$$\mathbf{u}(d) = \langle u(w_1,d),\ldots,u(w_{|V|},d)\rangle$$

Store only non-zeros in **u**(*d*), so size is O(|*d*|)

$$\mathbf{u}(y) = \alpha \frac{1}{|C_y|} \sum_{d \in C_y} \frac{\mathbf{u}(d)}{\|\mathbf{u}(d)\|_2} - \beta \frac{1}{|D - C_y|} \sum_{d' \in D - C_y} \frac{\mathbf{u}(d')}{\|\mathbf{u}(d')\|_2}$$

But size of **u**(*y*) is O(|n_V|)

$$f(d) = \arg\max_y \frac{\mathbf{u}(d)}{\|\mathbf{u}(d)\|_2} \cdot \frac{\mathbf{u}(y)}{\|\mathbf{u}(y)\|_2}$$

$$\|\mathbf{u}\|_2 = \sqrt{\sum_i u_i^2}$$

TFIDF similarity

$$DF(w) = \#\text{ different docs } w \text{ occurs in}$$

$$TF(w,d) = \#\text{ different times } w \text{ occurs in doc } d$$

$$IDF(w) = \frac{|D|}{DF(w)}$$

$$u(w,d) = \log(TF(w,d)+1) \cdot \log(IDF(w))$$

$$\mathbf{u}(d) = \left\langle u(w_1,d),....,u(w_{|V|},d) \right\rangle$$

$$\mathbf{v}(d) = \frac{\mathbf{u}(d)}{\|\mathbf{u}(d)\|_2}$$

$$sim(\mathbf{v}(d_1),\mathbf{v}(d_2)) = \mathbf{v}(d_1) \cdot \mathbf{v}(d_2) = \sum_w \frac{u(w,d_1)}{\|\mathbf{u}(d_1)\|_2} \frac{u(w,d_2)}{\|\mathbf{u}(d_2)\|_2}$$

TFIDF soft joins

- A similarity join of two sets of TFIDF-weighted vectors A and B is
  - an ordered list of triples (s_{ij}, a_i, b_j) such that
    - a_i is from A
    - b_j is from B
    - s_{ij} is the dot product of a_i and b_j
    - the triples are in descending order

    - the list is either the top K triples by s_{ij} or ALL triples with $s_{ij}>L$ … or sometimes some approximation of these….

PARALLEL SOFT JOINS

Efficient Parallel Set-Similarity Joins Using MapReduce

Rares Vernica
Department of Computer Science
University of California, Irvine
rares@ics.uci.edu

Michael J. Carey
Department of Computer Science
University of California, Irvine
mjcarey@ics.uci.edu

Chen Li
Department of Computer Science
University of California, Irvine
chenli@ics.uci.edu

SIGMOD 2010

TFIDF similarity: variant for joins

$$DF(A,w) = \#\text{different docs } w \text{ occurs in from A}$$

$$DF(B,w) = \#\text{different docs } w \text{ occurs in from B}$$

$$TF(w,d) = \#\text{different times } w \text{ occurs in doc } d$$

$$IDF(w,d) = \frac{|C_d|}{DF(C_d,w)},\ \text{where } C_d \in \{A,B\}$$

$$u(w,d) = \log(TF(w,d)+1) \cdot \log(IDF(w,d))$$

$$\mathbf{u}(d) = \left\langle u(w_1,d),\ldots.,u(w_{|V|},d)\right\rangle$$

$$\mathbf{v}(d) = \frac{\mathbf{u}(d)}{\|\mathbf{u}(d)\|_2}$$

$$sim(\mathbf{v}(d_1),\mathbf{v}(d_2)) = \mathbf{v}(d_1)\cdot\mathbf{v}(d_2) = \sum_w \frac{u(w,d_1)}{\|\mathbf{u}(d_1)\|_2}\frac{u(w,d_2)}{\|\mathbf{u}(d_2)\|_2}$$

Sim Joins on Product Descriptions

- Similarity can be **high** for descriptions of **distinct** items:
  - AERO TGX-Series Work Table -42" x 96" Model 1TGX-4296 All tables shipped KD AEROSPEC- 1TGX Tables are Aerospec Designed. In addition to above specifications; - All four sides have a V countertop edge ...
  - AERO TGX-Series Work Table -42" x 48" Model 1TGX-4248 All tables shipped KD AEROSPEC- 1TGX Tables are Aerospec Designed. In addition to above specifications; - All four sides have a V countertop ..
- Similarity can be **low** for descriptions of **identical** items:
  - Canon Angle Finder C 2882A002 Film Camera Angle Finders Right Angle Finder C (Includes ED-C & ED-D Adapters for All SLR Cameras) Film Camera Angle Finders & Magnifiers The Angle Finder C lets you adjust ...
  - CANON 2882A002 ANGLE FINDER C FOR EOS REBEL® SERIES PROVIDES A FULL SCREEN IMAGE SHOWS EXPOSURE DATA BUILT-IN DIOPTRIC ADJUSTMENT COMPATIBLE WITH THE CANON® REBEL, EOS & REBEL EOS SERIES.

Parallel Inverted Index Softjoin - 1

```
#compute document frequency
docFreq = Group(data, by=lambda(rel,docid,term):(rel,term), reducingTo=ReduceToCount()) \
 | ReplaceEach(by=lambda((rel,term),df):(rel,term,df))

#find total number of docs per relation
ndoc = ReplaceEach(data, by=lambda(rel,docid,term):(rel,docid)) \
       | Distinct() | Group(by=lambda(rel,docid):rel, reducingTo=ReduceToCoun

#unweighted document vectors
udocvec = Join( Jin(data,by=lambda(rel,docid,term):(rel,term)),
                Jin(docFreq,by=lambda(rel,term,df):(rel,term)) ) \
    | ReplaceEach(by=lambda((rel,doc,term),(rel_,term_,df)):(rel,doc,term,df)
    | JoinTo( Jin(ndoc,by=lambda(rel,relCount):rel), by=lambda(rel,doc,term,d
    | ReplaceEach(by=lambda((rel,doc,term,df),(rel_,relCount)):(rel,doc,term,
    | ReplaceEach(by=lambda(rel,doc,term,df,relCount):(rel,doc,term,termWeight(relCount,df)))

#normalizers
sumSquareWeights = ReduceTo(float, lambda accum,(rel,doc,term,weight): accum+weight*weight)
norm = Group( udocvec,
              by=lambda(rel,doc,term,weight):(rel,doc),
              retaining = lambda (rel,doc,term,weight): weight,
              reducingTo=ReduceToSum() ) \
       | ReplaceEach( by=lambda((rel,doc),z):(rel,doc,z))

#normalized document vector
docvec = Join( Jin(norm,by=lambda(rel,doc,z):(rel,doc)),
               Jin(udocvec,by=lambda(rel,doc,term,weight):(rel,doc)) ) \
    | ReplaceEach( by=lambda((rel,doc,z),(rel_,doc_,term,weight)): (rel,doc,term,weight/math.sqrt(z)) )
```

want this to work for long documents or short ones…and keep the relations simple

sumSquareWeights

Statistics for computing TFIDF with IDFs local to each relation

Parallel Inverted Index Softjoin - 2

```
# naive algorithm: use all pairs for finding matches
rel1Docs = Filter(docvec, by=lambda(rel,doc,term,weight):rel=='icepark')
rel2Docs = Filter(docvec, by=lambda(rel,doc,term,weight):rel=='npspark')
softjoin = Join( Jin(rel1Docs,by=lambda(rel,doc,term,weight):term),
                 Jin(rel2Docs,by=lambda(rel,doc,term,weight):term)) \
    | ReplaceEach(by=lambda((rel1,doc1,term,weight1),(rel2,doc2,term2,weight2)): (doc1,doc2,weight1*weight2)) \
    | Group(by=lambda(doc1,doc2,p):(doc1,doc2), \
            retaining=lambda(doc1,doc2,p):p, \
            reducingTo=ReduceToSum()) \
    | ReplaceEach(by=lambda((doc1,doc2),sim):(doc1,doc2,sim))

simpairs = Filter(softjoin, by=lambda(doc1,doc,sim):sim>0.75)
```

What's the algorithm?

- Step 1: create document vectors as *(C_d, d, term, weight)* tuples
- Step 2: *join* the tuples from A and B: one sort and reduce
    - Gives you tuples (*a, b, term, w(a,term)*w(b,term))*
- Step 3: *group* the common terms by (a,b) and reduce to aggregate the components of the sum

An alternative TFIDF pipeline

```python
def loadDictView(view):
    result = {}
    for (key,val) in GPig.rowsOf(view):
        result[key] = val
    return result
```

```python
class TFIDF(Planner):

    D = GPig.getArgvParams()
    data = ReadLines(D.get('corpus','idcorpus.txt')) \
        | Map(by=lambda line:line.strip().split("\t")) \
        | Map(by=lambda (docid,doc): (docid,doc.lower().split())) \
        | FlatMap(by=lambda (docid,words): map(lambda w:(docid,w),words))

    #compute document frequency and inverse doc freq
    docFreq = Distinct(data) \
        | Group(by=lambda (docid,term):term, retaining=lambda(docid,term):docid, reducingTo=ReduceToCount())

    ndoc = Map(data, by=lambda (docid,term):docid) \
        | Distinct() \
        | Group(by=lambda row:'ndoc', reducingTo=ReduceToCount())

    inverseDocFreq = Augment(docFreq, sideview=ndoc, loadedBy=lambda v:GPig.onlyRowOf(v)) \
        | Map(by=lambda((term,df),(dummy,ndoc)):(term,math.log(ndoc/df)))

    #compute unweighted document vectors
    udocvec = Augment(data, sideview=inverseDocFreq, loadedBy=loadDictView) \
        | Map(by=lambda ((docid,term),idfDict):(docid,term,idfDict[term]))

    #normalize
    norm = Group( udocvec, by=lambda(docid,term,weight):docid,
                           retaining=lambda(docid,term,weight):weight*weight,
                           reducingTo=ReduceToSum() )

    docvec = Augment(udocvec, sideview=norm, loadedBy=loadDictView) \
        | Map( by=lambda ((docid,term,weight),normDict): (docid,term,weight/math.sqrt(normDict[docid])))
```

Inverted Index Softjoin – PIG 1/3

```
-- invoke as: pig --param input=id-park --param rel=icepark ... phirl.pig

%default output sim
%default rel a
%default def_par 10

SET default_parallel $def_par;

-- load and tokenize the data as data:{rel,id,str,term}

raw = LOAD 'phirl/$input' AS (rel,docid,keyid,str);
data = FOREACH raw GENERATE rel,docid,FLATTEN(TOKENIZE(LOWER(str))) AS term;

-- compute relation-dependent document frequencies as docfreq:{rel,term,df:int}

docfreq =
  FOREACH (GROUP data by (rel,term))
  GENERATE group.rel AS rel, group.term as term, COUNT(data) as df;

-- find the total number of documents in each relation as ndoc:{rel,c:long}

ndoc1 = DISTINCT(FOREACH data GENERATE rel,docid);
ndoc = FOREACH (GROUP ndoc1 by rel) GENERATE group AS rel, COUNT(ndoc1) AS c;
```

Inverted Index Softjoin – 2/3

```
-- find the un-normalized document vectors as udocvec:{rel.docid,term,weight}
udocvec1 = JOIN data BY (rel,term), docfreq BY (rel,term);
udocvec2 = JOIN udocvec1 BY data::rel, ndoc BY rel;
udocvec =
   FOREACH udocvec2
   GENERATE data::rel, data::docid, data::term,
     LOG(2.0)*LOG(ndoc::c/(double)docfreq::df) AS weight;

-- find the square of the normalizer for each document: norm:{rel,docid,z2:double}

norm1 = FOREACH udocvec GENERATE rel,docid,term,weight*weight as w2;
norm =
   FOREACH (GROUP norm1 BY (rel,docid))
   GENERATE group.rel AS rel, group.docid AS docid, SUM(norm1.w2) AS z2;

-- compute the TFIDF weighted document vectors as: docvec:{rel,docid,term,weight:double}
docvec =
   FOREACH (JOIN udocvec BY (rel,docid), norm BY (rel,docid))
   GENERATE data::rel AS rel, data::docid AS docid, data::term AS term,
      weight/SQRT(z2) as weight;
```

Inverted Index Softjoin – 3/3

docvec:{rel,docid,term,weight:double}

```
-- naive algorithm: use all terms for finding potentil matches

docsA = FILTER docvec BY rel=='$rel';
docsB = FILTER docvec BY rel!='$rel';
softjoin1 = JOIN docsA BY term, docsB BY term;
softjoin2 =
   FOREACH softjoin1
   GENERATE docsA::docid AS idA, docsB::docid AS idB, docsA::weight*docsB::weight AS p;
softjoin =
   FOREACH (GROUP softjoin2 BY (idA,idB))
   GENERATE group.idA, group.idB, SUM(softjoin2.p) AS sim;
```

```
# naive algorithm: use all pairs for finding matches
rel1Docs = Filter(docvec, by=lambda(rel,doc,term,weight):rel=='icepark')
rel2Docs = Filter(docvec, by=lambda(rel,doc,term,weight):rel=='npspark')
softjoin = Join( Jin(rel1Docs,by=lambda(rel,doc,term,weight):term),
                 Jin(rel2Docs,by=lambda(rel,doc,term,weight):term)) \
    | ReplaceEach(by=lambda((rel1,doc1,term,weight1),(rel2,doc2,term2,weight2)): (doc1,doc2,weight1*weight2)) \
    | Group(by=lambda(doc1,doc2,p):(doc1,doc2), \
          retaining=lambda(doc1,doc2,p):p, \
          reducingTo=ReduceToSum()) \
    | ReplaceEach(by=lambda((doc1,doc2),sim):(doc1,doc2,sim))
```

Inverted Index Softjoin – 3/3

```
docvec:{rel,docid,term,weight:double}
```

```
-- naive algorithm: use all terms for finding potentil matches

docsA = FILTER docvec BY rel=='$rel';
docsB = FILTER docvec BY rel!='$rel';
softjoin1 = JOIN docsA BY term, docsB BY term;
softjoin2 =
   FOREACH softjoin1
   GENERATE docsA::docid AS idA, docsB::docid AS idB, docsA::weight*docsB::weight AS p;
softjoin =
   FOREACH (GROUP softjoin2 BY (idA,idB))
   GENERATE group.idA, group.idB, SUM(softjoin2.p) AS sim;
```

```
-- diagnostic output: look: {sim,[01],idA,idB,str1,str2}

look1 = JOIN topSimPairs BY idA, raw BY docid;
look2 = JOIN look1 BY idB, raw BY docid;
look =
   FOREACH look2
   GENERATE sim, (look1::raw::keyid==raw::keyid ? 1 : 0),
     idA,idB, look1::raw::str AS str1,raw::str AS str2;

STORE look INTO 'phirl/$output';
```

Results…..

0.99436717611623	1	d00059	d00436	Carl Sandburg Home NHS	Carl Sandburg Home NHS
0.9937688379278058	1	d00354	d00611	Theodore Roosevelt NP	Theodore Roosevelt NP
0.9920648281782544	1	d00286	d00573	Oregon Caves NM	Oregon Caves NM
0.9914077975044103	1	d00274	d00566	New River Gorge NR	New River Gorge NR
0.9881961852455996	1	d00009	d00399	American Memorial Park	American Memorial Park
0.9878514547862078	1	d00154	d00500	George Washington Memorial Parkway	George Washington Me
0.9422676645498852	1	d00376	d00623	War in the Pacific NHP	War in Pacific NHP
0.92307133361005	1	d00323	d00594	Saguaro NP	Saguaro NM
0.8914304226443976	1	d00292	d00577	Pea Ridge NHS	Pea Ridge NMP
0.890829830425262	1	d00200	d00532	Jean Lafitte NHP & NPRES	Jean Lafitte NHP & Preserve
0.8873463623037525	0	d00283	d00570	Obed Wild and Scenic River	Obed Wild & Scenic River
0.8838421147370781	1	d00342	d00606	Sitka NHS	Sitka NHP
0.8838421147370781	1	d00011	d00401	Andersonville NHS	Andersonville NHP
0.8700042867436217	1	d00026	d00413	Bering Land Bridge NPRES	Bering Land Bridge N. Preser
0.8684330615122184	1	d00157	d00643	Glacier Bay NP & NPRES	Glacier Bay NP & Preserve
0.8680495192463105	1	d00339	d00603	Sequoia and Kings Canyon NP	Sequoia & Kings Canyon NP
0.8660286476353838	1	d00267	d00561	National Park Of American Samoa	NP of American Samoa
0.8593112749780314	1	d00210	d00538	Kalaupapa NHP	Kalaupapa NHS
0.8500226387429363	1	d00208	d00536	Johnstown Flood NM	Johnstown Flood N. Mem.
0.8424859579540737	1	d00222	d00646	Lake Clark NP & NPRES	Lake Clark NP & Preserve
0.8398407018438242	1	d00187	d00523	Homestead National Monument of America	Homestead NM of Amer
0.8395526626941698	1	d00230	d00548	Lincoln Boyhood NM	Lincoln Boyhood N. Mem.
0.8390553468895996	1	d00349	d00610	Sunset Crater NM	Sunset Crater Volcano NM
0.8344604123961857	1	d00259	d00559	Mount Rushmore NM	Mount Rushmore N. Mem.
0.8313853772986841	0	d00353	d00611	Theodore Roosevelt Island	Theodore Roosevelt NP
0.8301435671019225	1	d00071	d00444	Chesapeake & Ohio Canal NHP	Chesapeake and Ohio Canal NH
0.82492593280652	1	d00019	d00407	Arkansas Post NM	Arkansas Post N. Mem.
0.8202902347497227	1	d00212	d00644	Katmai NP & NPRES	Katmai NP & Preserve
0.8202902347497227	1	d00098	d00464	Denali NP & NPRES	Denali NP & Preserve
0.7965479702996782	1	d00013	d00402	Aniakchak NM & NPRES	Aniakchak NM
0.7835432589199314	1	d00031	d00417	Big Thicket NPRES	Big Thicket N. Preserve
0.7835432589199314	1	d00028	d00415	Big Cypress NPRES	Big Cypress N. Preserve

```
raw = LOAD 'phirl/$input' AS (rel,docid,keyid,str);
data = FOREACH raw GENERATE rel,docid,FLATTEN(TOKENIZE(LOWER(str))) AS term;

-- compute relation-dependent document frequencies as docfreq:{rel,term,df:int}

docfreq =
  FOREACH (GROUP data by (rel,term))
  GENERATE group.rel AS rel, group.term as term, COUNT(data) as df;

-- find the total number of documents in each relation as ndoc:{rel,c:long}

ndoc1 = DISTINCT(FOREACH data GENERATE rel,docid);
ndoc = FOREACH (GROUP ndoc1 by rel) GENERATE group AS rel, COUNT(ndoc1) AS c;

-- find the un-normalized document vectors as udocvec:{rel.docid,term,weight}
udocvec1 = JOIN data BY (rel,term), docfreq BY (rel,term);
udocvec2 = JOIN udocvec1 BY data::rel, ndoc BY rel;
udocvec =
   FOREACH udocvec2
   GENERATE data::rel, data::docid, data::term,
     LOG(2.0)*LOG(ndoc::c/(double)docfreq::df) AS weight;

-- find the square of the normalizer for each document: norm:{rel,docid,z2:double}

norm1 = FOREACH udocvec GENERATE rel,docid,term,weight*weight as w2;
norm =
   FOREACH (GROUP norm1 BY (rel,docid))
   GENERATE group.rel AS rel, group.docid AS docid, SUM(norm1.w2) AS z2;

-- compute the TFIDF weighted document vectors as: docvec:{rel,docid,term,weight:double}
docvec =
   FOREACH (JOIN udocvec BY (rel,docid), norm BY (rel,docid))
   GENERATE data::rel AS rel, data::docid AS docid, data::term AS term,
      weight/SQRT(z2) as weight;

fs -rmr phirl/docvec
STORE docvec INTO 'phirl/docvec';

-- naive algorithm: use all terms for finding potentil matches

docsA = FILTER docvec BY rel=='$rel';
docsB = FILTER docvec BY rel!='$rel';
softjoin1 = JOIN docsA BY term, docsB BY term;
softjoin2 =
   FOREACH softjoin1
   GENERATE docsA::docid AS idA, docsB::docid AS idB, docsA::weight*docsB::weight AS p;
softjoin =
   FOREACH (GROUP softjoin2 BY (idA,idB))
   GENERATE group.idA, group.idB, SUM(softjoin2.p) AS sim;
```

Making the algorithm smarter….

Inverted Index Softjoin - 2

```
-- naive algorithm: use all terms for finding potentil matches

docsA = FILTER docvec BY rel=='$rel';
docsB = FILTER docvec BY rel!='$rel';
softjoin1 = JOIN docsA BY term, docsB BY term;
softjoin2 =
   FOREACH softjoin1
   GENERATE docsA::docid AS idA, docsB::docid AS idB, docsA::weight*docsB::weight AS p;
softjoin =
   FOREACH (GROUP softjoin2 BY (idA,idB))
   GENERATE group.idA, group.idB, SUM(softjoin2.p) AS sim;
```

we should make a smart choice about which terms to use

Adding heuristics to the soft join - 1

```
-- compute maximum weight for rel2docs as: maxweight2:{term,weight}

maxweightB =
   FOREACH (GROUP docsB BY (rel,term))
   GENERATE group.term AS term, MAX(docsB.weight) AS weight;

-- augment the docvecs for rel1 with maxweight2 and docfreq information to get
-- augdocsA: {rel,docid,term, w,df,maxw,score}

docfreqB = FILTER docfreq BY rel!='$rel';
augdocsA1 = JOIN docsA BY term, docfreqB BY term, maxweightB BY term;
augdocsA =
   FOREACH augdocsA1
   GENERATE docsA::rel, docsA::docid, docsA::term, docsA::weight AS w,
     docfreqB::df AS df, maxweightB::weight AS maxw,
     docsA::weight*maxweightB::weight AS score;
```

$$\mathbf{v}_a \mathbf{v}_b = \sum_w \mathbf{v}_a[w] * \mathbf{v}_b[w] \leq \sum_w \underbrace{\mathbf{v}_a[w] * \mathrm{maxweight2}[w]}_{\text{score for w in doc a}}$$

Adding heuristics to the soft join - 1

```
augdocsA =
   FOREACH augdocsA1
   GENERATE docsA::rel, docsA::docid, docsA::term, docsA::weight AS w,
     docfreqB::df AS df, maxweightB::weight AS maxw,
     docsA::weight*maxweightB::weight AS score;

-- filter out useful terms to join on, using the info in augdocsA.
-- the heuristics used here are:
--- (1) only use top K by maxscore w/in each document;
--- (2) filter by df<=maxDF
--- (3) filter by score>=minscore

usefulTerms1 =
   FOREACH (GROUP augdocsA BY (rel,docid))
   GENERATE group, TOP($top_k,6,augdocsA) AS top;
usefulTerms2 =
   FOREACH usefulTerms1 {
      filteredTop = FILTER top BY (df<=$max_df) AND score>$min_sim;
      topTerms = FOREACH filteredTop GENERATE term;
      GENERATE flatten(topTerms);
   };
usefulTerms = DISTINCT usefulTerms2;
```

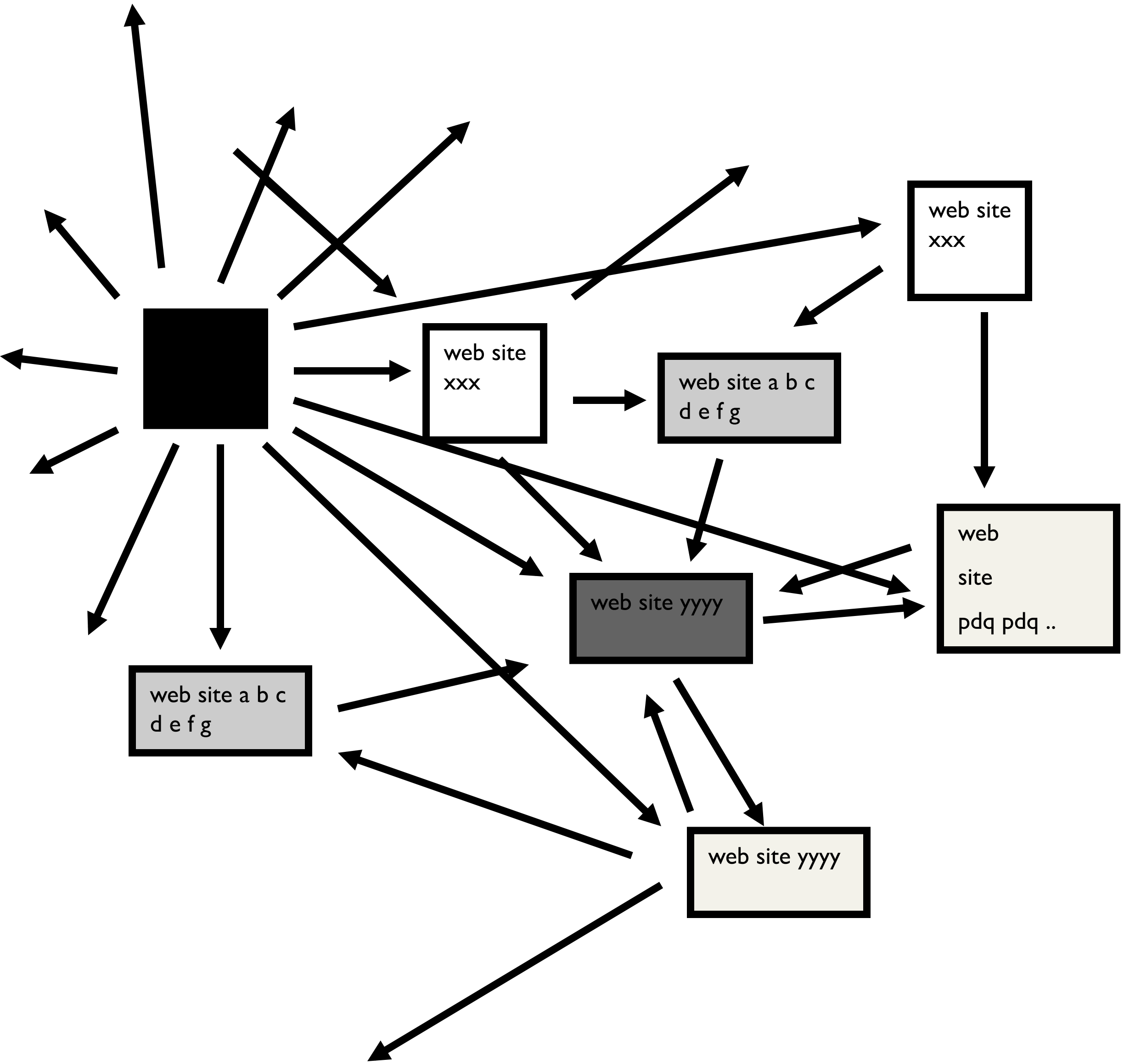
Adding heuristics to the soft join - 2

```
-- use the restricted sets of terms to get candidate pairs

pairs1 = JOIN usefulTerms BY term, docsA BY term, docsB BY term;
pairs2 = FOREACH pairs1 GENERATE docsA::docid AS idA, docsB::docid AS idB;
pairs = DISTINCT pairs2;
-- STORE pairs INTO 'phirl/pairs';

softjoin1 = JOIN pairs BY idA, docsA by docid;
softjoin2 = JOIN softjoin1 BY (idB,term), docsB by (docid,term);
softjoin3 =
   FOREACH softjoin2
   GENERATE idA, idB, docsA::term AS term, docsA::weight*docsB::weight AS p;
softjoin =
   FOREACH (GROUP softjoin3 BY (idA,idB))
   GENERATE group.idA, group.idB, SUM(softjoin3.p) AS sim;
```

```
docsA = FILTER docvec BY rel=='$rel';
docsB = FILTER docvec BY rel!='$rel';

-- compute maximum weight for rel2docs as: maxweight2:{term,weight}

maxweightB =
   FOREACH (GROUP docsB BY (rel,term))
   GENERATE group.term AS term, MAX(docsB.weight) AS weight;

-- augment the docvecs for rel1 with maxweight2 and docfreq information to get
-- augdocsA: {rel,docid,term, w,df,maxw,score}

docfreqB = FILTER docfreq BY rel!='$rel';
augdocsA1 = JOIN docsA BY term, docfreqB BY term, maxweightB BY term;
augdocsA =
   FOREACH augdocsA1
   GENERATE docsA::rel, docsA::docid, docsA::term, docsA::weight AS w,
     docfreqB::df AS df, maxweightB::weight AS maxw,
     docsA::weight*maxweightB::weight AS score;
usefulTerms1 =
   FOREACH (GROUP augdocsA BY (rel,docid))
   GENERATE group, TOP($top_k,6,augdocsA) AS top;
usefulTerms2 =
   FOREACH usefulTerms1 {
      filteredTop = FILTER top BY (df<=$max_df) AND score>$min_sim;
      topTerms = FOREACH filteredTop GENERATE term;
      GENERATE flatten(topTerms);
   };
usefulTerms = DISTINCT usefulTerms2;

pairs1 = JOIN usefulTerms BY term, docsA BY term, docsB BY term;
pairs2 = FOREACH pairs1 GENERATE docsA::docid AS idA, docsB::docid AS idB;
pairs = DISTINCT pairs2;
-- STORE pairs INTO 'phirl/pairs';

softjoin1 = JOIN pairs BY idA, docsA by docid;
softjoin2 = JOIN softjoin1 BY (idB,term), docsB by (docid,term);
softjoin3 =
   FOREACH softjoin2
   GENERATE idA, idB, docsA::term AS term, docsA::weight*docsB::weight AS p;
softjoin =
   FOREACH (GROUP softjoin3 BY (idA,idB))
   GENERATE group.idA, group.idB, SUM(softjoin3.p) AS sim;
```

PageRank at Scale

Google's PageRank



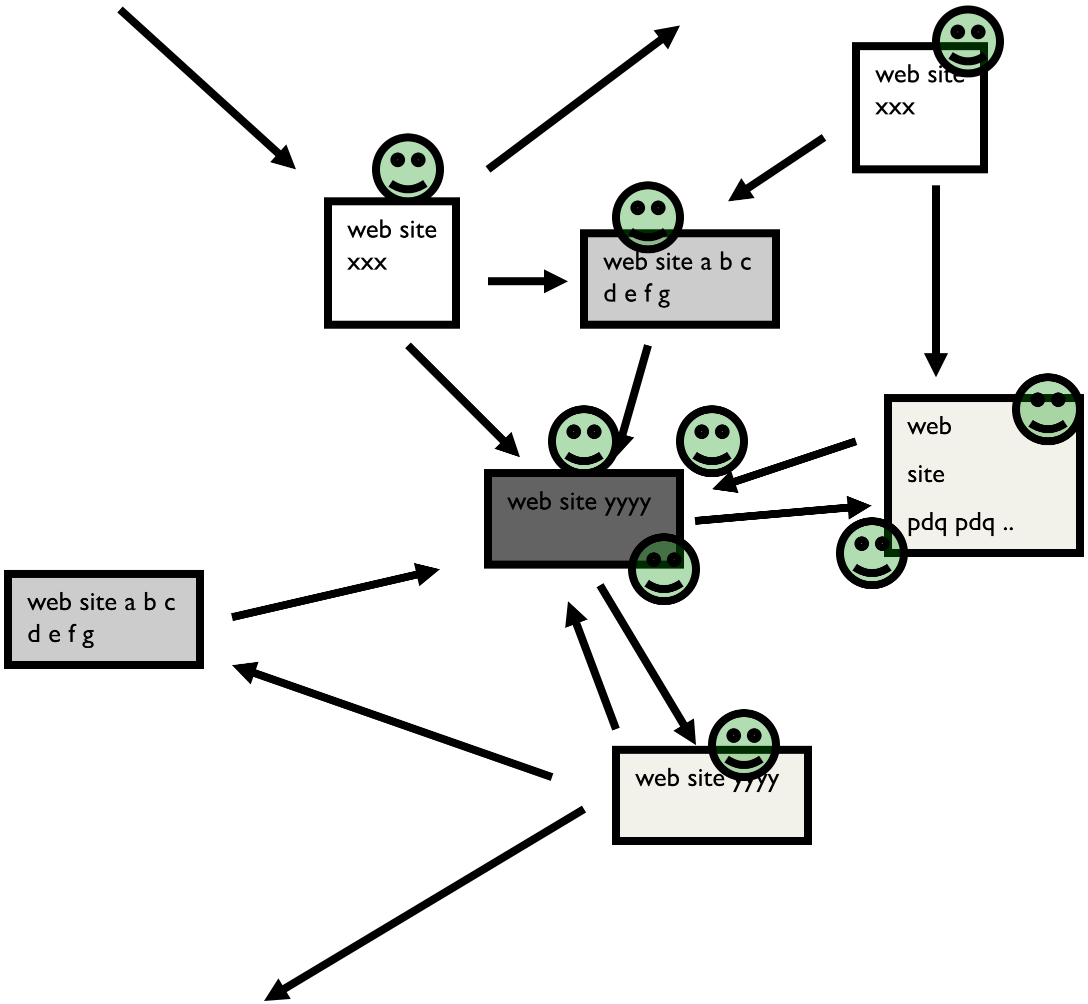
Inlinks are "good" (recommendations)

Inlinks from a "good" site are better than inlinks from a "bad" site

but inlinks from sites with many outlinks are not as "good"...

"Good" and "bad" are relative.

Google's PageRank

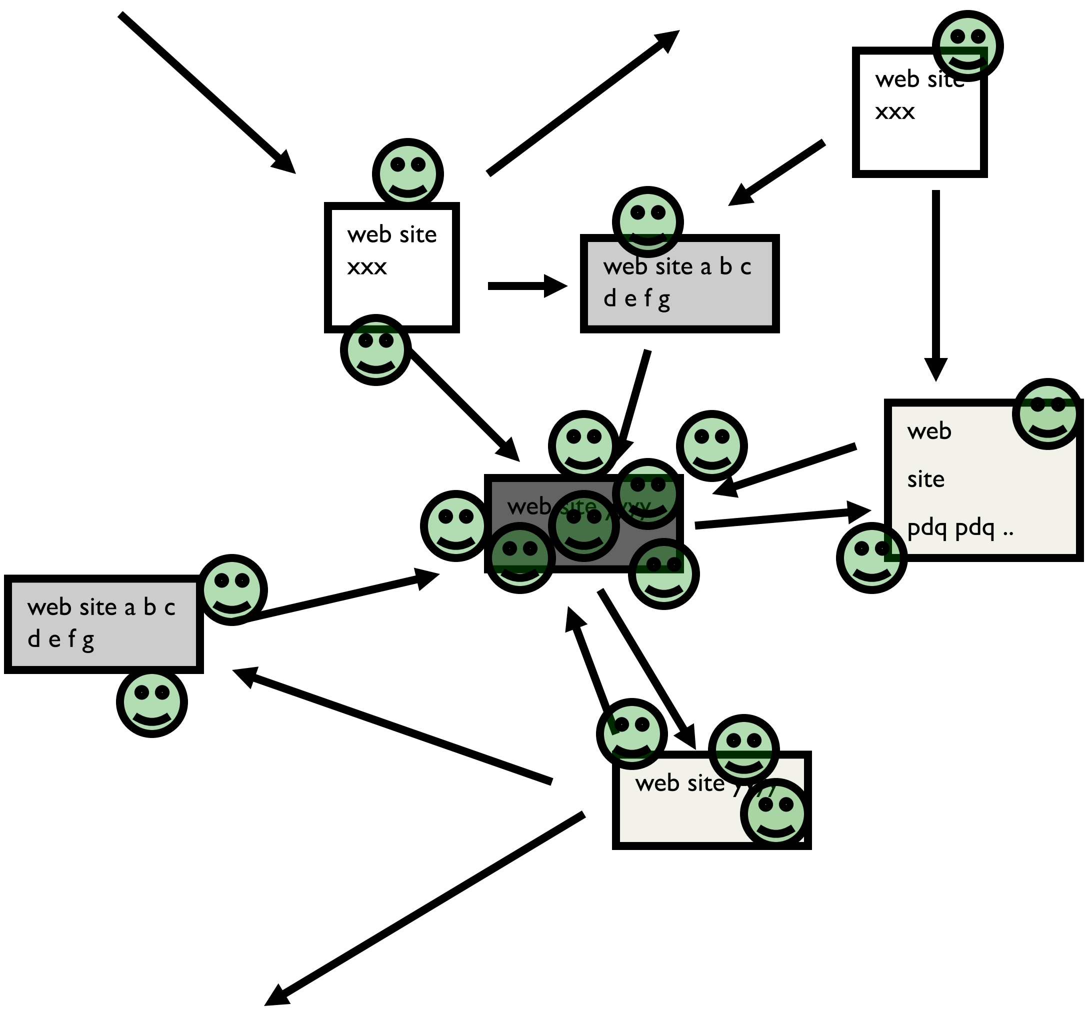


Imagine a "pagehopper" that always either

- follows a random link, or
- jumps to random page

Google's PageRank

(Brin & Page, http://www-db.stanford.edu/~backrub/google.html)



Imagine a "pagehopper" that always either

- follows a random link, or
- jumps to random page

PageRank ranks pages by the amount of time the pagehopper spends on a page:

- or, if there were many pagehoppers, PageRank is the expected "crowd size"

PageRank in Memory

- Let $\mathbf{u} = (1/N, \ldots, 1/N)$
  - dimension = #nodes N
- Let A = adjacency matrix: $[a_{ij}=1 \Leftrightarrow$ i links to j]
- Let $W = [w_{ij} = a_{ij}/\text{outdegree}(i)]$
  - w_{ij} is probability of jump from i to j
- Let $\mathbf{v}^0 = (1,1,\ldots,1)$
  - or anything else you want
- Repeat until converged:
  - Let $\mathbf{v}^{t+1} = c\mathbf{u} + (1\text{-}c)\mathbf{W}\mathbf{v}^t$
    - c is probability of jumping “anywhere randomly”

Streaming PageRank

- Assume we can store **v** but not **W** in memory
- Repeat until converged:
  - Let $\mathbf{v}^{t+1} = c\mathbf{u} + (1\text{-}c)\mathbf{W}\mathbf{v}^t$
- Store **A** as a row matrix: each line is
  - i $j_{i,1},\ldots,j_{i,d}$ [the neighbors of i]
- Store **v'** and **v** in memory: **v'** starts out as c**u**
- For each line "i $j_{i,1},\ldots,j_{i,d}$ "
  - For each j in $j_{i,1},\ldots,j_{i,d}$
    - $\mathbf{v}'[j] \mathrel{+}= (1\text{-}c)\mathbf{v}[\mathbf{i}]/d$

Everything needed for update is right there in row….

Streaming PageRank: with some long rows

- Repeat until converged:
  - Let $\mathbf{v}^{t+1} = c\mathbf{u} + (1\text{-}c)\mathbf{W}\mathbf{v}^t$

- Store **A** as a list of edges: each line is: "i d(i) j"
- Store **v'** and **v** in memory: **v'** starts out as **cu**
- For each line "i d j"
  - **v'**[j] += (1-c)**v[i]**/d

We need to get the degree of *i* and store it locally

Streaming PageRank: preprocessing

- Original encoding is edges (i,j)
- Mapper replaces i,j with i,1
- Reducer is a SumReducer
- Result is pairs (i,d(i))

- Then: join this back with edges (i,j)
- For each i,j pair:
  - send j as a message to node i in the degree table
    - messages always sorted after non-messages
  - the reducer for the degree table sees i,d(i) first
    - then j1, j2, ….
    - can output the key,value pairs with key=i, value=d(i), j

Preprocessing Control Flow: 1

I	J
i1	j1,1
i1	j1,2
…	…
i1	j1,k1
i2	j2,1
…	…
i3	j3,1
…	…

MAP

I	
i1	1
i1	1
…	…
i1	1
i2	1
…	…
i3	1
…	…

SORT

I	
i1	1
i1	1
…	…
i1	1
i2	1
…	…
i3	1
…	…

REDUCE

Summing values

I	d(i)
i1	d(i1)
..	…
i2	d(i2)
…	…
i3	d)i3)
…	…

Preprocessing Control Flow: 2

I	J
i1	j1,1
i1	j1,2
…	…
i2	j2,1
…	…

I	d(i)
i1	d(i1)
..	…
i2	d(i2)
…	…

I	J
i1	~j1,1
i1	~j1,2
…	…
i2	~j2,1
…	…

I	d(i)
i1	d(i1)
..	…
i2	d(i2)
…	…

I	
i1	d(i1)
i1	~j1,1
i1	~j1,2
..	…
i2	d(i2)
i2	~j2,1
i2	~j2,2
…	…

I		
i1	d(i1)	j1,1
i1	d(i1)	j1,2
…	…	…
i1	d(i1)	j1,n1
i2	d(i2)	j2,1
…	…	…
i3	d(i3)	j3,1
…	…	…

MAP

SORT

REDUCE

copy or convert to messages

join degree with edges

Streaming PageRank: with some long rows

- Repeat until converged:
  - Let $\mathbf{v}^{t+1} = c\mathbf{u} + (1\text{-}c)\mathbf{W}\mathbf{v}^t$

- Pure streaming: use a table of nodes→ degree+pageRank
  - Lines are *i: degree=d,pr=v*
- For each edge *i,j*
  - Send to i (in degree/pagerank) table: outlink j
- For each line *i*: degree=*d*,pr=*v*:
  - send to *i:* incrementVBy *c*
  - for each message "outlink j":
    - send to *j:* incrementVBy *(1-c)*v/d*
- For each line *i:* degree=d,pr=v
  - sum up the incrementVBy messages to compute v'
  - output new row: i: degree=*d*,pr=*v'*

One identity mapper with two inputs (edges, degree/ pr table)

Reducer outputs the incrementVBy messages

Two-input mapper + reducer

Control Flow: Streaming PR

I	J
i1	j1,1
i1	j1,2
…	…
i2	j2,1
…	…

I	d/v
i1	d(i1),v(i1)
i2	d(i2),v(i2)
…	…

I	d/v
i1	d(i1),v(i1)
i1	~j1,1
i1	~j1,2
..	…
i2	d(i2),v(i2)
i2	~j2,1
i2	~j2,2
…	…

to	delta
i1	c
j1,1	(1-c)v(i1)/d(i1)
…	…
j1,n1	i
i2	c
j2,1	…
…	…
i3	c

I	delta
i1	c
i1	(1-c)v(…)….
i1	(1-c)…
..	…
i2	c
i2	(1-c)…
i2	….
…	…

MAP → SORT

copy or convert to messages

REDUCE

send "pageRank updates " to outlinks

MAP → SORT

Control Flow: Streaming PR

to	delta
i1	c
j1,1	(1-c)v(i1)/d(i1)
…	…
j1,n1	i
i2	c
j2,1	…
…	…
i3	c

I	delta
i1	c
i1	(1-c)v(…)….
i1	(1-c)…
..	…
i2	c
i2	(1-c)…
i2	….
…	…

I	v'
i1	~v'(i1)
i2	~v'(i2)
…	…

I	d/v
i1	d(i1),v(i1)
i2	d(i2),v(i2)
…	…

I	d/v
i1	d(i1),v'(i1)
i2	d(i2),v'(i2)
…	…

REDUCE → MAP → SORT → REDUCE → MAP → SORT → REDUCE

Summing values

Replace v with v'

Control Flow: Streaming PR

I	J
i1	j1,1
i1	j1,2
…	…
i2	j2,1
…	…

I	d/v
i1	d(i1),v(i1)
i2	d(i2),v(i2)
…	…

and back around for next iteration….

copy or convert to messages

PageRank in Pig

How to use loops, conditionals, etc?

Embed PIG in a real programming language.

Julien Le Dem - Yahoo

```
#!/usr/bin/python
from org.apache.pig.scripting import *

P = Pig.compile("""
-- PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))

previous_pagerank =
    LOAD '$docs_in'
    USING PigStorage('\t')
    AS ( url: chararray, pagerank: float, links:{ link: ( url: chararray ) } );

outbound_pagerank =
    FOREACH previous_pagerank
    GENERATE
        pagerank / COUNT ( links ) AS pagerank,
        FLATTEN ( links ) AS to_url;

new_pagerank =
    FOREACH
        ( COGROUP outbound_pagerank BY to_url, previous_pagerank BY url INNER )
    GENERATE
        group AS url,
        ( 1 - $d ) + $d * SUM ( outbound_pagerank.pagerank ) AS pagerank,
        FLATTEN ( previous_pagerank.links ) AS links;

STORE new_pagerank
    INTO '$docs_out'
    USING PigStorage('\t');
""")

params = { 'd': '0.5', 'docs_in': 'data/pagerank_data_simple' }

for i in range(10):
    out = "out/pagerank_data_" + str(i + 1)
    params["docs_out"] = out
    Pig.fs("rmr " + out)
    stats = P.bind(params).runSingle()
    if not stats.isSuccessful():
        raise 'failed'
    params["docs_in"] = out
```

```
#!/usr/bin/python
from org.apache.pig.scripting import *

P = Pig.compile("""
pig script: PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))
""")

params = { 'd': '0.5', 'docs_in': 'data/pagerank_data_simple' }

for i in range(10):
    out = "out/pagerank_data_" + str(i + 1)
    params["docs_out"] = out
    Pig.fs("rmr " + out)
    stats = P.bind(params).runSingle()
    if not stats.isSuccessful():
        raise 'failed'
    params["docs_in"] = out
```

Iterate 10 times

Pass parameters as a dictionary

Just run P, that was declared above

The output becomes the new input

```
previous_pagerank =
    LOAD '$docs_in'
    USING PigStorage('\t')
    AS ( url: chararray, pagerank: float, links:{ link: ( url: chararray ) } );

outbound_pagerank =
    FOREACH previous_pagerank
    GENERATE
        pagerank / COUNT ( links ) AS pagerank,
        FLATTEN ( links ) AS to_url;

new_pagerank =
    FOREACH
        ( COGROUP outbound_pagerank BY to_url, previous_pagerank BY url INNER )
    GENERATE
        group AS url,
        ( 1 - $d ) + $d * SUM ( outbound_pagerank.pagerank ) AS pagerank,
        FLATTEN ( previous_pagerank.links ) AS links;

STORE new_pagerank
    INTO '$docs_out'
    USING PigStorage('\t');
```

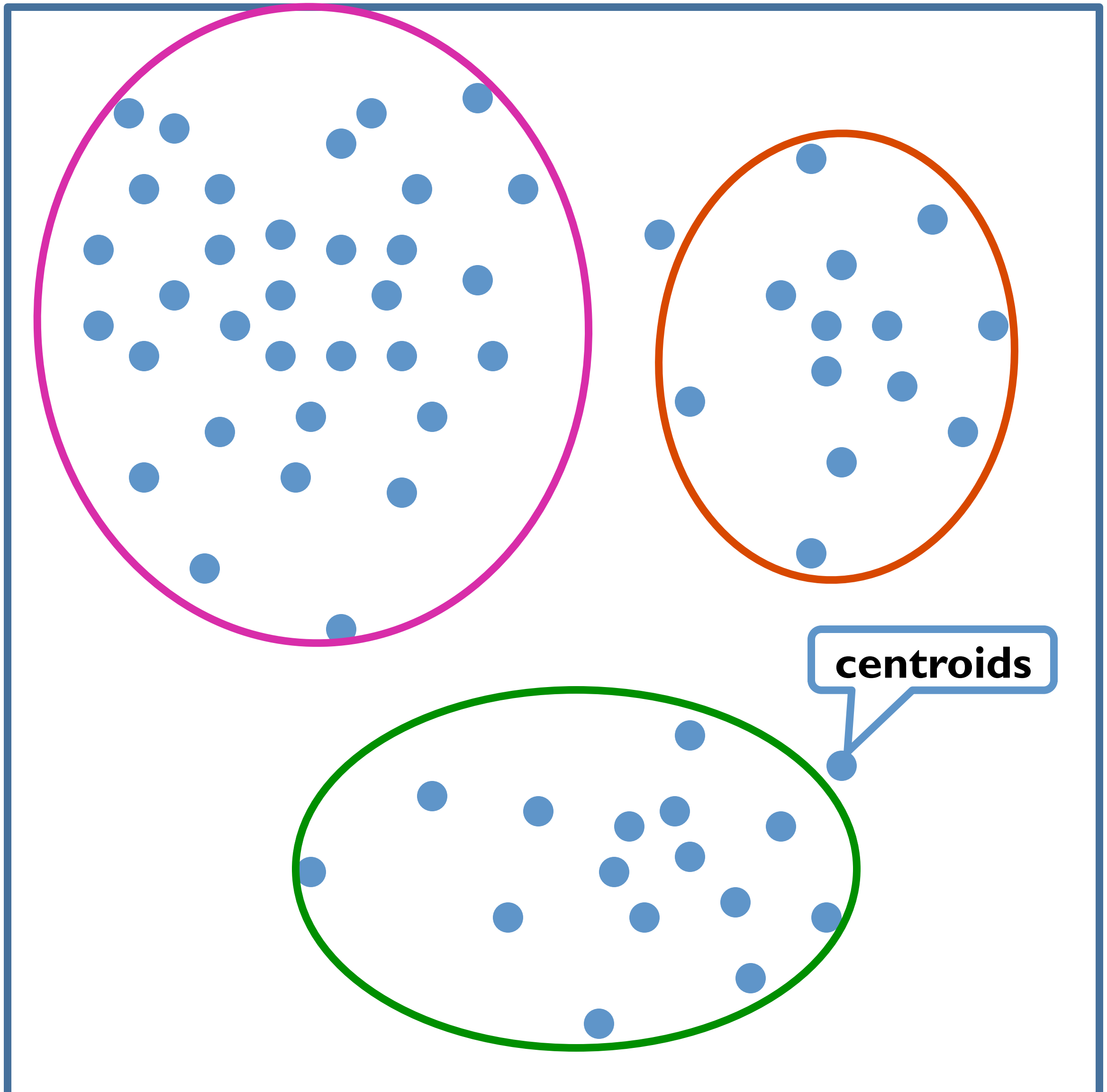
lots of i/o happening here…

An example from Ron Bekkerman

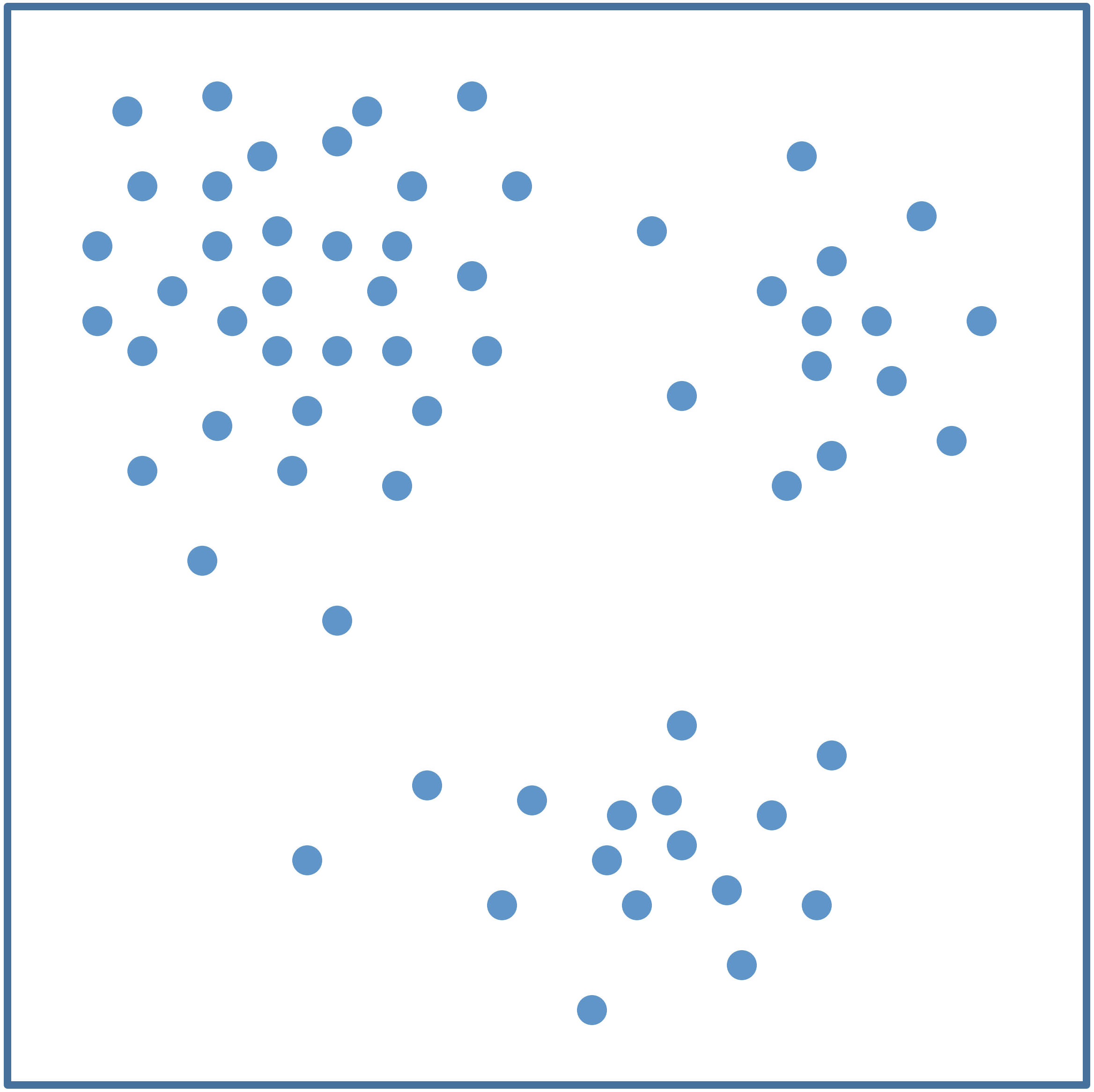
Example: *k*-means clustering

- An EM-like algorithm:
- Initialize k cluster centroids
- E-step: associate each data instance with the closest centroid
  - Find expected values of cluster assignments given the data and centroids
- M-step: recalculate centroids as an average of the associated data instances
  - Find new centroids that maximize that expectation

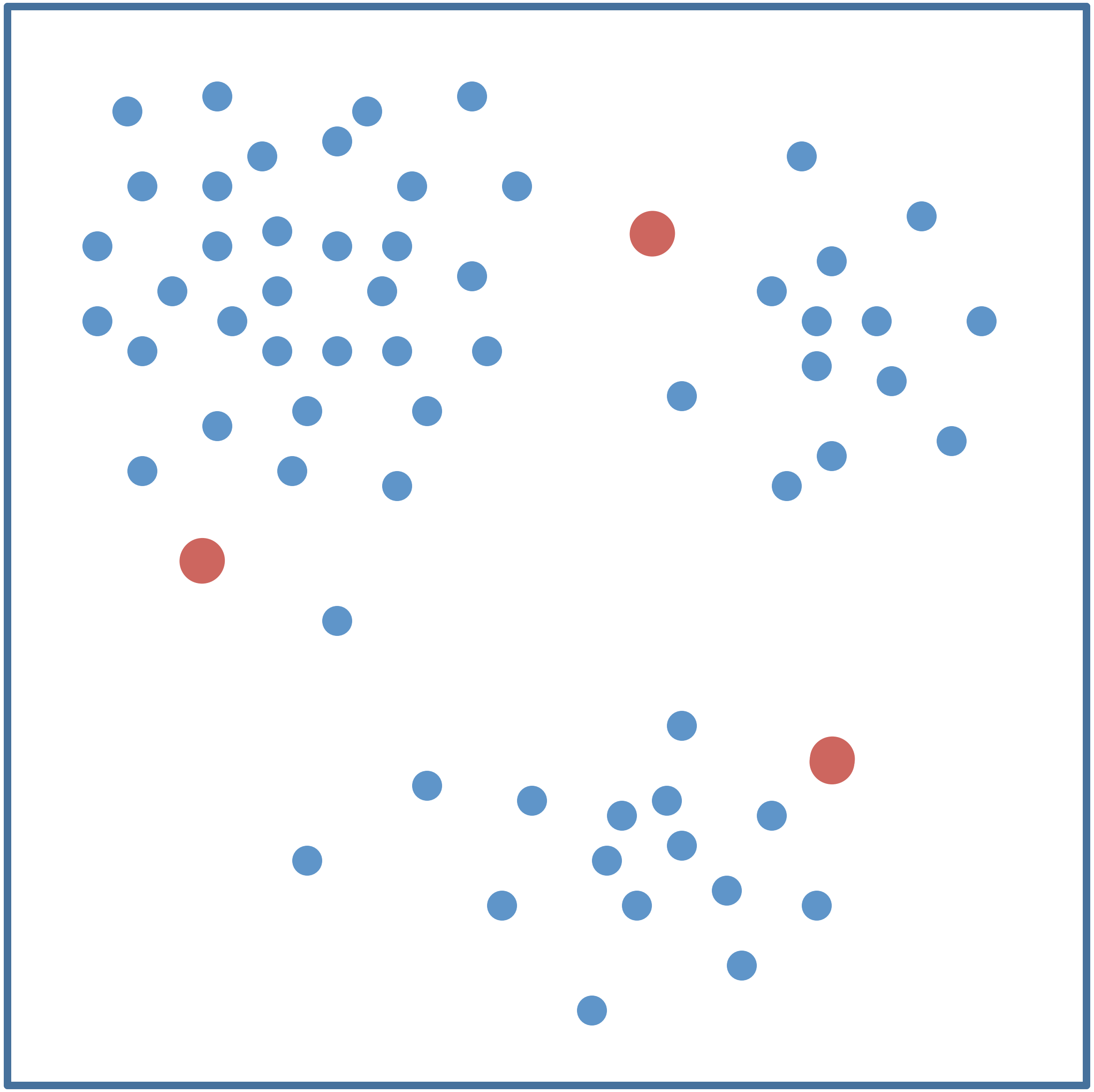
k-means Clustering



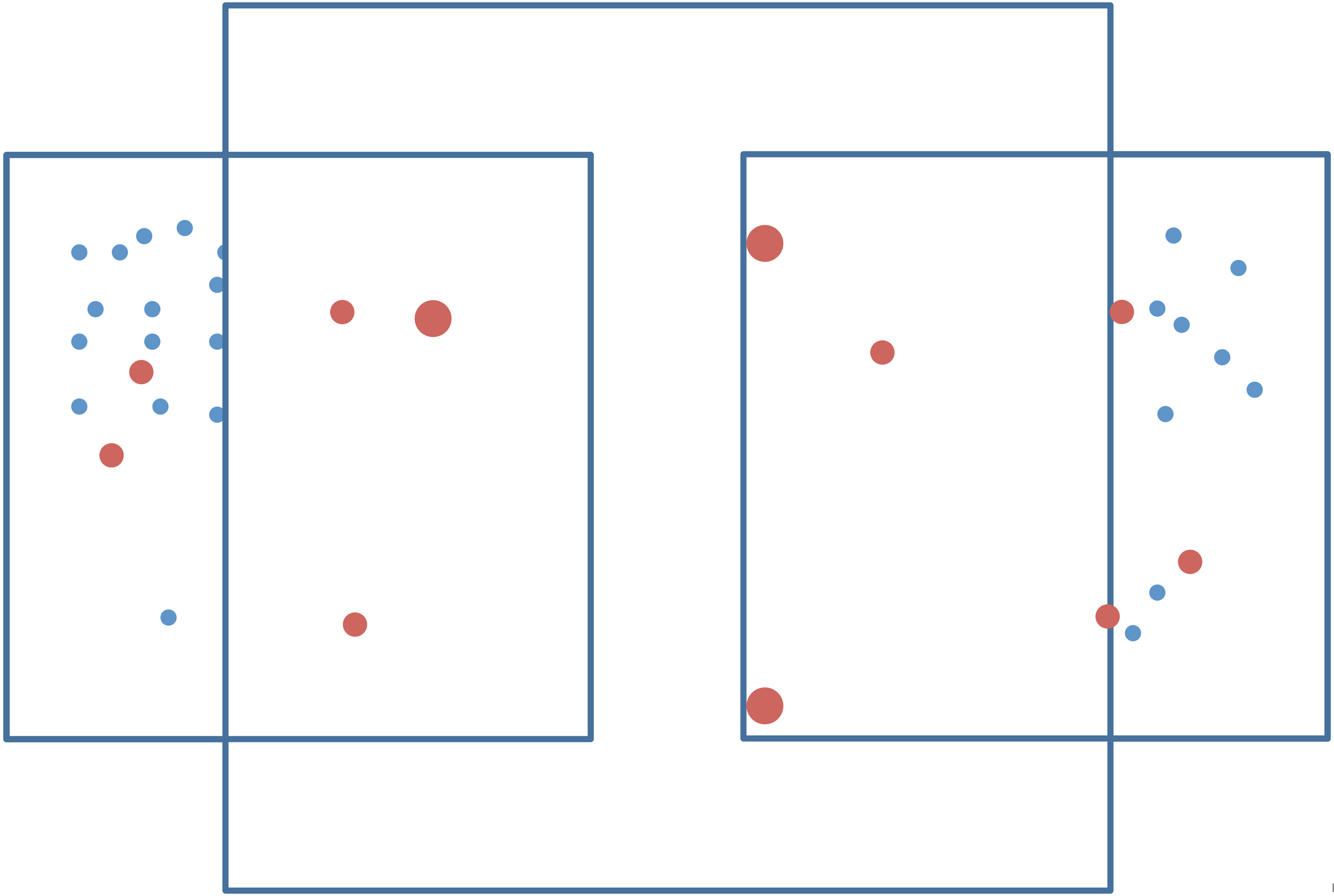
Parallelizing *k*-means



Parallelizing *k*-means



Parallelizing *k*-means



k-means on MapReduce

Panda et al, Chapter 2

- Mappers read data portions and centroids
- Mappers **assign data instances** to clusters
- Mappers **compute new local centroids** and local cluster sizes
- Reducers **aggregate local centroids** (weighted by local cluster sizes) into new global centroids
- Reducers **write the new centroids**

k-means in Apache Pig: input data

- Assume we need to cluster documents
  - Stored in a 3-column table *D*:

Document	Word	Count
doc1	Carnegie	2
doc1	Mellon	2

- Initial centroids are *k* randomly chosen docs
  - Stored in table *C* in the same format as above

k-means in Apache Pig: E-step

D_C = **JOIN** *C* **BY** *w*, *D* **BY** *w*;
PROD = **FOREACH** *D_C* **GENERATE** *d*, *c*, i_d * i_c **AS** $i_d i_c$;

$PROD_g$ = **GROUP** *PROD* **BY** (*d*, *c*);
*DOT_* … **S** *dXc*;

SQR
SQR_g
*LEN_* … n_c;

$$c_d = \arg\max_c \frac{\sum_{w\in d} i_d^w \cdot i_c^w}{\sqrt{\sum_{w\in c} \left(i_c^w\right)^2}}$$

*DOT_*
SIM = **FOREACH** *DOT_LEN* **GENERATE** *d*, *c*, *dXc* / len_c;

SIM_g = **GROUP** *SIM* **BY** *d*;
CLUSTERS = **FOREACH** SIM_g **GENERATE** TOP(1, 2, *SIM*);

k-means in Apache Pig: E-step

D_C = **JOIN** *C* **BY** *w*, *D* **BY** *w*;
PROD = **FOREACH** *D_C* **GENERATE** *d*, *c*, $i_d * i_c$ **AS** $i_d i_c$;

$PROD_g$ = **GROUP** *PROD* **BY** (*d*, *c*);
*DOT_* … **S** *dXc*;

SQR
SQR_g
*LEN_* … n_c;

*DOT_*

$$c_d = \arg\max_c \frac{\sum_{w \in d} i_d^w \cdot i_c^w}{\sqrt{\sum_{w \in c} \left(i_c^w\right)^2}}$$

SIM = **FOREACH** *DOT_LEN* **GENERATE** *d*, *c*, *dXc* / len_c;

SIM_g = **GROUP** *SIM* **BY** *d*;
CLUSTERS = **FOREACH** SIM_g **GENERATE** TOP(1, 2, *SIM*);

k-means in Apache Pig: E-step

D_C = **JOIN** *C* **BY** *w*, *D* **BY** *w*;
PROD = **FOREACH** *D_C* **GENERATE** *d*, *c*, i_d * i_c **AS** $i_d i_c$;

$PROD_g$ = **GROUP** *PROD* **BY** (*d*, *c*);
*DOT_* … **S** *dXc*;

SQR …
SQR_g …
*LEN_* … n_c;

*DOT_* …
SIM = **FOREACH** *DOT_LEN* **GENERATE** … $_c$;

SIM_g = **GROUP** *SIM* **BY** *d*;
CLUSTERS = **FOREACH** SIM_g **GENERATE** TOP(1, 2, *SIM*);

$$c_d = \arg\max_c \frac{\sum_{w\in d} i_d^w \cdot i_c^w}{\sqrt{\sum_{w\in c}\left(i_c^w\right)^2}}$$

k-means in Apache Pig: E-step

D_C = **JOIN** *C* **BY** *w*, *D* **BY** *w*;
PROD = **FOREACH** *D_C* **GENERATE** *d*, *c*, $i_d * i_c$ **AS** $i_d i_c$;

$PROD_g$ = **GROUP** *PROD* **BY** (*d*, *c*);
*DOT_* … **S** *dXc*;

SQR …
SQR_g …
*LEN_* … n_c;

*DOT_* …
SIM = **FOREACH** *DOT_LEN* **GENERATE** *d*, *c*, *dXc* / len_c;

SIM_g = **GROUP** *SIM* **BY** *d*;
CLUSTERS = **FOREACH** SIM_g **GENERATE** TOP(1, 2, *SIM*);

$$c_d = \arg\max_c \frac{\sum_{w \in d} i_d^w \cdot i_c^w}{\sqrt{\sum_{w \in c} \left(i_c^w\right)^2}}$$

k-means in Apache Pig: E-step

D_C = **JOIN** *C* **BY** *w*, *D* **BY** *w*;
PROD = **FOREACH** *D_C* **GENERATE** *d*, *c*, i_d * i_c **AS** $i_d i_c$;

$PROD_g$ = **GROUP** *PROD* **BY** (*d*, *c*);
*DOT_* … **S** *dXc*;

SQR
SQR_g
*LEN_* … n_c;

*DOT_*
SIM = **FOREACH** *DOT_LEN* **GENERATE** *d*, *c*, *dXc* / len_c;

SIM_g = **GROUP** *SIM* **BY** *d*;
CLUSTERS = **FOREACH** SIM_g **GENERATE** TOP(1, 2, *SIM*);

$$c_d = \arg\max_c \frac{\sum_{w \in d} i_d^w \cdot i_c^w}{\sqrt{\sum_{w \in c} \left(i_c^w\right)^2}}$$

k-means in Apache Pig: E-step

D_C = **JOIN** *C* **BY** *w*, *D* **BY** *w*;
PROD = **FOREACH** *D_C* **GENERATE** *d*, *c*, $i_d * i_c$ **AS** $i_d i_c$;

$PROD_g$ = **GROUP** *PROD* **BY** (*d*, *c*);
DOT_PROD = **FOREACH** $PROD_g$ **GENERATE** *d*, *c*, SUM($i_d i_c$) **AS** *dXc*;

SQR = **FOREACH** *C* **GENERATE** *c*, $i_c * i_c$ **AS** i_c^2;
SQR_g = **GROUP** *SQR* **BY** c;
LEN_C = **FOREACH** SQR_g **GENERATE** *c*, SQRT(SUM(i_c^2)) **AS** len_c;

DOT_LEN = **JOIN** *LEN_C* **BY** *c*, *DOT_PROD* **BY** *c*;
SIM = **FOREACH** *DOT_LEN* **GENERATE** *d*, *c*, *dXc* / len_c;

SIM_g = **GROUP** *SIM* **BY** *d*;
CLUSTERS = **FOREACH** SIM_g **GENERATE** TOP(1, 2, *SIM*);

k-means in Apache Pig: M-step

D_C_W = **JOIN** *CLUSTERS* **BY** *d*, *D* **BY** *d*;

$D\_C\_W_{g}$ = **GROUP** *D_C_W* **BY** (*c*, *w*);
SUMS = **FOREACH** $D\_C\_W_{g}$ **GENERATE** *c*, *w*, SUM(i_d) **AS** *sum*;

$D\_C\_W_{gg}$ = **GROUP** *D_C_W* **BY** *c*;
SIZES = **FOREACH** $D\_C\_W_{gg}$ **GENERATE** *c*, COUNT(*D_C_W*) **AS** *size*;

SUMS_SIZES = **JOIN** *SIZES* **BY** *c*, *SUMS* **BY** *c*;
C = **FOREACH** *SUMS_SIZES* **GENERATE** *c*, *w*, *sum* / *size* **AS** i_c ;

Finally - embed in Java (or Python or ….) to do the looping

How to use loops, conditionals, etc?

Embed PIG in a real programming language.

h/t Julien Le Dem - Yahoo

```
#!/usr/bin/python
from org.apache.pig.scripting import *

P = Pig.compile("""
-- PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))

previous_pagerank =
    LOAD '$docs_in'
    USING PigStorage('\t')
    AS ( url: chararray, pagerank: float, links:{ link: ( url: chararray ) } );

outbound_pagerank =
    FOREACH previous_pagerank
    GENERATE
        pagerank / COUNT ( links ) AS pagerank,
        FLATTEN ( links ) AS to_url;

new_pagerank =
    FOREACH
        ( COGROUP outbound_pagerank BY to_url, previous_pagerank BY url INNER )
    GENERATE
        group AS url,
        ( 1 - $d ) + $d * SUM ( outbound_pagerank.pagerank ) AS pagerank,
        FLATTEN ( previous_pagerank.links ) AS links;

STORE new_pagerank
    INTO '$docs_out'
    USING PigStorage('\t');
""")

params = { 'd': '0.5', 'docs_in': 'data/pagerank_data_simple' }

for i in range(10):
    out = "out/pagerank_data_" + str(i + 1)
    params["docs_out"] = out
    Pig.fs("rmr " + out)
    stats = P.bind(params).runSingle()
    if not stats.isSuccessful():
        raise 'failed'
    params["docs_in"] = out
```

```
#!/usr/bin/python
from org.apache.pig.scripting import *

P = Pig.compile("""
pig script: PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))
""")

params = { 'd': '0.5', 'docs_in': 'data/pagerank_data_simple' }

for i in range(10):
    out = "out/pagerank_data_" + str(i + 1)
    params["docs_out"] = out
    Pig.fs("rmr " + out)
    stats = P.bind(params).runSingle()
    if not stats.isSuccessful():
        raise 'failed'
    params["docs_in"] = out
```

Iterate 10 times

Pass parameters as a dictionary

Just run P, that was declared above

The output becomes the new input

The problem with k-means in Hadoop

I/O costs

Data is read, and model is written, with every iteration

Panda et al, Chapter 2

- Mappers read data portions and centroids
- Mappers **assign data instances** to clusters
- Mappers **compute new local centroids** and local cluster sizes
- Reducers **aggregate local centroids** (weighted by local cluster sizes) into new global centroids
- Reducers **write the new centroids**

Spark

Spark

- Too much typing
  - programs are not concise
- Too low level
  - missing abstractions
  - hard to specify a workflow
- Not well suited to iterative operations
  - E.g., E/M, k-means clustering, …
  - Workflow and memory-loading issues

Set of concise dataflow operations ("transformation")

Dataflow operations are embedded in an API together with "actions"

Sharded files are replaced by "RDDs" – resiliant distributed datasets

RDDs can be cached in *cluster* memory and recreated to recover from error

Spark examples

```
errors.cache()
```

spark is a *spark context* object

```
text_file = spark.textFile("hdfs://...")
errors = text_file.filter(lambda line: "ERROR" in line)
# Count all the errors
errors.count()
# Count errors mentioning MySQL
errors.filter(lambda line: "MySQL" in line).count()
# Fetch the MySQL errors as an array of strings
errors.filter(lambda line: "MySQL" in line).collect()
```

Spark examples

```
errors.cache()

text_file = spark.textFile("hdf
errors = text_file.filter(lambda line: "ERROR" in line)
# Count all the errors
errors.count()
# Count errors mentioning MySQL
errors.filter(lambda line: "MySQL" in line).count()
# Fetch the MySQL errors as an array of strings
errors.filter(lambda line: "MySQL" in line).collect()
```

errors is a *transformation*, and thus a
that exp
do s

count() is an *action*: it will actually execute the plan for **errors** and return a value.

everything is **sharded**, like in Hadoop and GuineaPig

errors.filter() is a *transformation*

collect() is an *action*

Spark examples

everything is **sharded** … and the shards are stored in *memory* of worker machines not local *disk* (if possible)

```
text_file = spark.textFile("hdfs://...")
errors = text_file.filter(lambda line: "ERROR" in line)
errors.cache()
errors.count()
# Count errors mentioning MySQL
errors.filter(lambda line: "MySQL" in line).count()
# Fetch the MySQL errors as an array of strings
errors.filter(lambda line: "MySQL
```

\# modify **errors** to be stored in *cluster memory*

subsequent actions will be much faster

You can also **persist()** an RDD on disk, which is like marking it as opts(stored=True) in GuineaPig. Spark's *not* smart about persisting data.

Spark examples: wordcount

```
text_file = spark.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

the action

transformation on (key,value) pairs , which are special

Spark examples: batch logistic regression

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

reduce is an action – it produces a numby vector

p.x and **w** are *vectors*, from the numpy package. Python overloads operations like * and + for vectors.

Spark examples: batch logistic regression

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

Important note: numpy vectors/matrices are not just "syntactic sugar".

- They are *much more compact* than something like a list of python floats.
- numpy operations like **dot, *, +** are calls to *optimized C code*
- a little python logic around a lot of numpy calls is pretty efficient

Spark examples: batch logistic regression

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

w is defined *outside* the lambda function, but used *inside* it

So: python builds a *closure* – code including the *current value* of **w** – and Spark ships it off to each worker. So **w** is *copied*, and must be *read-only*.

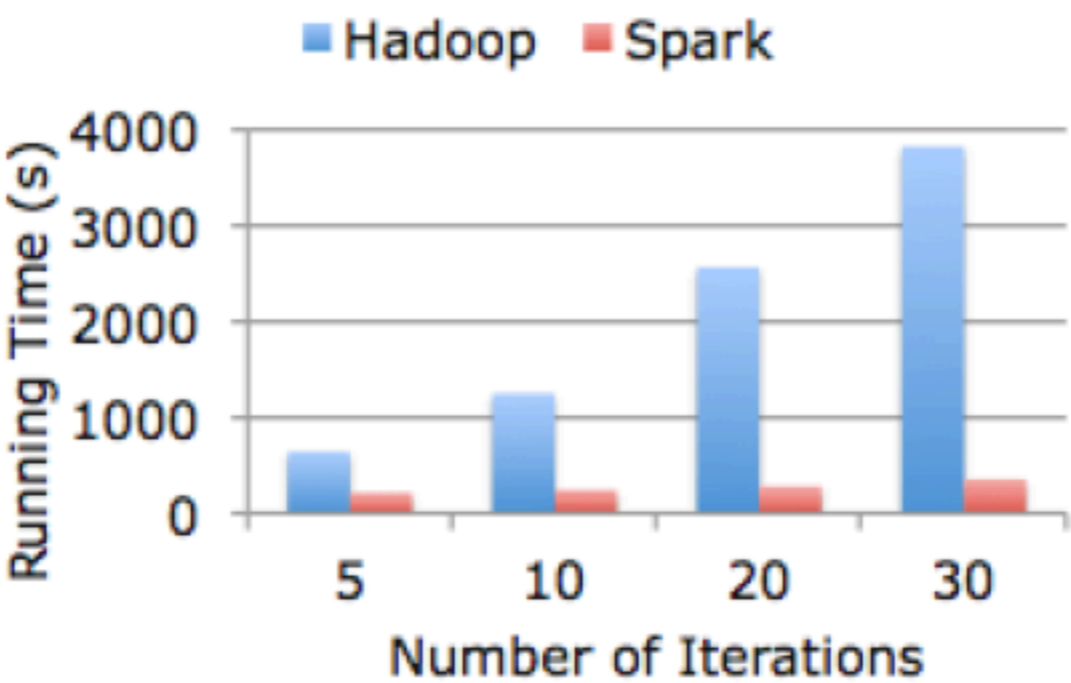
Spark examples: batch logistic regression

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

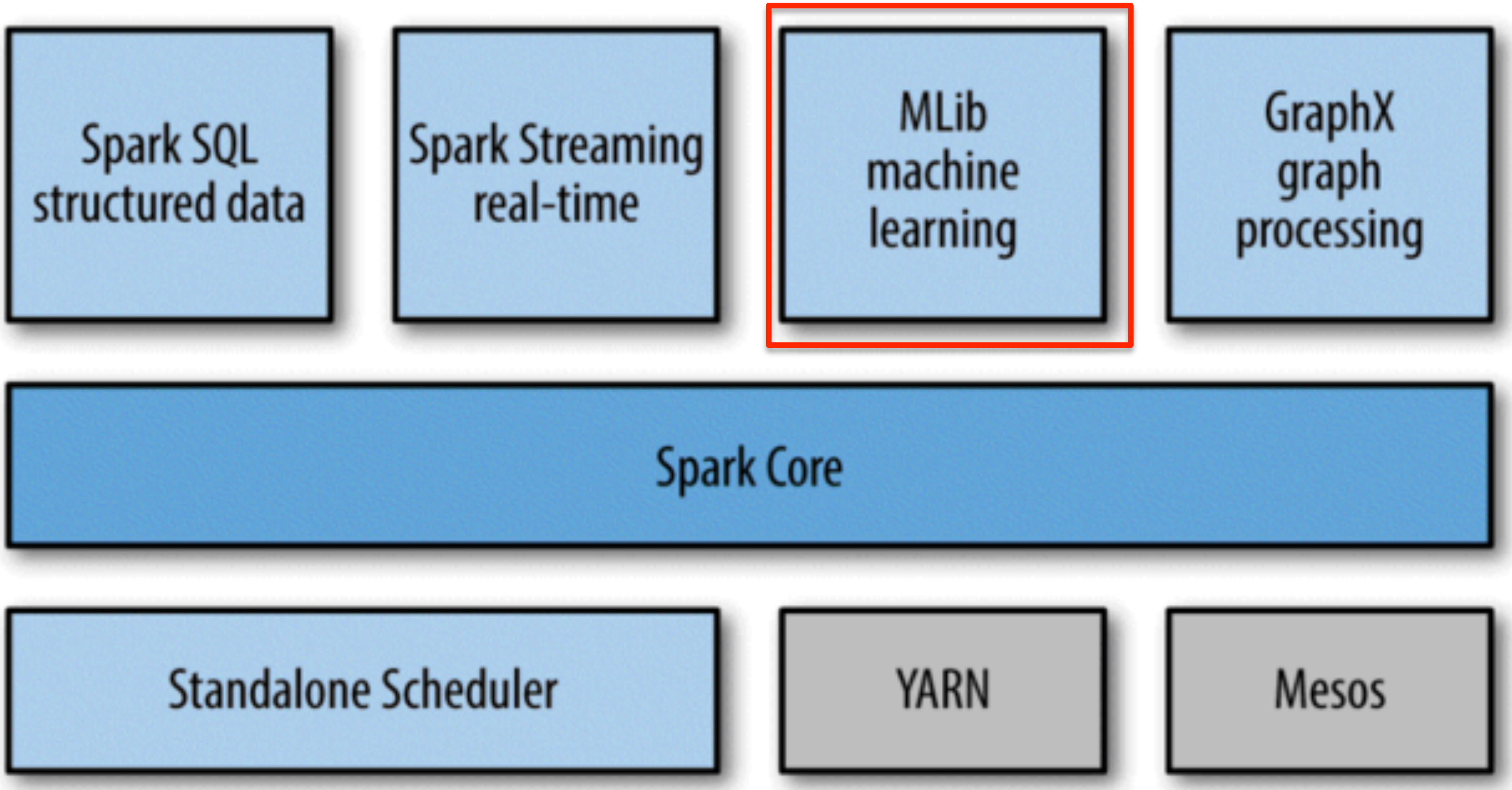
dataset of points is cached in cluster memory to reduce i/o

Spark logistic regression example

The graph below compares the performance of this Spark program against a Hadoop implementation on 30 GB of data on an 80-core cluster, showing the benefit of in-memory caching:



Spark



Spark SQL
structured data
Spark Streaming
real-time
MLib
machine
learning
GraphX
graph
processing
Spark Core
Standalone Scheduler
YARN
Mesos

Spark details: broadcast

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

So: python builds a *closure* – code including the *current value* of **w** – and Spark ships it off to each worker. So **w** is *copied*, and must be *read-only*.

Spark details: broadcast

little penalty for distributing something that's not used by all workers

```
points = spark.textFile(...).map(parsePoint).cac
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %
```

alternative: create a *broadcast variable*, e.g.,

- w_broad = spark.broadcast(w)

which is accessed by the worker via

- w_broad.value()

what's sent is a *small pointer* to **w** (e.g., the name of a file containing a serialized version of **w**) and when **value** is called, some clever all-reduce like machinery is used to reduce network load.

Spark details: mapPartitions

```
class WordProb(Planner):

    wc = ReadLines('corpus.txt') | Flatten(by=tokens) \
         | Group(by=lambda x:x, reducingTo=ReduceToCount())
    total = ...
    wcWithTotal = Augment(wc, sideview=total,loadedBy=lambda v:GPig.onlyRowOf(v))
    prob = ReplaceEach(wcWithTotal, by=lambda ((word,count),n): (word,count,n,float(count)/n))
```

Common issue:

- map task requires loading in some small shared value
- more generally, map task requires some sort of *initialization* before processing a shard
- GuineaPig:
  - special *Augment* … *sideview* … pattern for shared values
  - can kludge up any initializer using Augment
- Raw Hadoop: **mapper.configure()** and **mapper.close()** methods

Spark details: mapPartitions

```
class WordProb(Planner):

    wc = ReadLines('corpus.txt') | Flatten(by=tokens) \
         | Group(by=lambda x:x, reducingTo=ReduceToCount())
    total = ...
    wcWithTotal = Augment(wc, sideview=total,loadedBy=lambda v:GPig.onlyRowOf(v))
    prob = ReplaceEach(wcWithTotal, by=lambda ((word,count),n): (word,count,n,float(count)/n))
```

Spark:

- **rdd.mapPartitions(f)**: will call **f(iteratorOverShard)** once per shard, and return an iterator over the mapped values.

- **f()** can do any setup/close steps it needs

Also:

- there are transformations to partition an RDD with a user-selected function, like in Hadoop. Usually you partition and persist/cache.