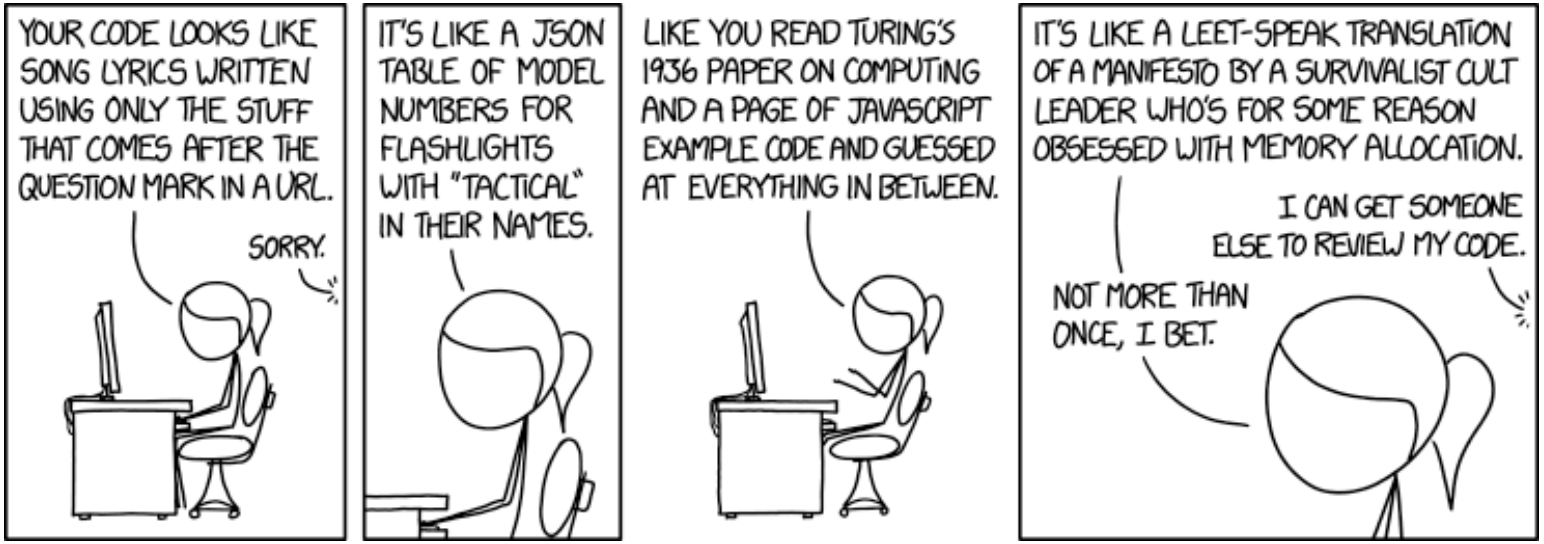


Workflows and Abstractions for Map-Reduce

GUINEA PIG: A WORKFLOW PACKAGE FOR PYTHON

A wordcount example

```
class WordCount(Planner):
    lines = ReadLines('corpus.txt')
    words = Flatten(lines,by=tokens)
    wordCount = Group(words, by=lambda x:x, reducingTo=ReduceToCount())
```

class variables in the planner are data structures

```
wordCount = Group(words,by=<function <lambda> at
| words = Flatten(lines, by=<function tokens at 0
| | lines = ReadLines("corpus.txt")
```

Semantics

- A program is converted to a data structure
- The data structure can be converted to a series of "abstract map-reduce tasks" and then shell commands

```
=====================================================================
map-reduce task 1: corpus.txt => wordCount
 - +-------------------- explanation --------------------
 - |  read corpus.txt with lines
 - |  flatten to words
 - |  group to wordCount
 - +-------------------- commands --------------------
 - |  python longer-wordcount.py --view=wordCount --do=doGroupl                    rt -k1 |
```

steps in the compiled plan invoke your script with special args

```
python longer-wordcount.py --view=wordCount --do=doGroupMap < corpus.txt \
    | LC_COLLATE=C sort -k1 \
    | python longer-wordcount.py --view=wordCount --do=doStoreRows \
    > gpig_views/wordCount.gp
```

Grouping

Full Syntax for Group

```
Group(wc,  by=lambda (word,count):word,
           retaining=lambda (word,count):count,
           combiningTo=ReduceToSum(),
           reducingTo=ReduceToSum())
```

(today, 1)
(i, 1)
..
(farmville, 1)
…

aardvark	(aardvark, 1)
aardvark	(aardvark, 1)
	..
zymurgy	(zymurgy, 1)
zymurgy	(zymurgy, 1)

Grouping

```
Group(wc,  by=lambda (word,count):word,
           reducingTo=ReduceTo(
                    lambda: 0,                          #init accum
                    lambda accum,(_,c):accum+c)  #update
```

(today, 1)
(i, 1)
..
(farmville, 1)
…

aardvark	(aardvark, 1)
aardvark	(aardvark, 1)
	..
zymurgy	(zymurgy, 1)
zymurgy	(zymurgy, 1)

(aardvark, 214)
(absolute, 1)
..
(zymurgy, 11)

aardvark	[(aardvark, 1),(aardvark,1),…]
	..
zymurgy	[(zymurgy, 1),….]

Grouping

```
Group(wc,  by=lambda (word,count):word,
           retaining=lambda (word,count):count,
           reducingTo=ReduceTo(
                        lambda: 0,                          #init accum
                        lambda accum,c:accum+c )            #update
```

(today, 1)
(i, 1)
..
(farmville, 1)
…

aardvark	[(~~aardvark,~~ 1),(~~aardvark~~,1),…]
	..
zymurgy	[(~~zymurgy~~, 1),….]

(aardvark, 214)
(absolute, 1)
..
(zymurgy, 11)

aardvark	[1,1,…]
	..
zymurgy	[1,1,….]

Grouping

```
Group(wc,  by=lambda (word,count):word,
           retaining=lambda (word,count):count,
           combiningTo=ReduceToSum(),
           reducingTo=ReduceToSum())
```

(today, 1)
(i, 1)
..
(farmville, 1)
…

aardvark	[1,1,…]
	..
zymurgy	[1,1,….]

(aardvark, 214)
(absolute, 1)
..
(zymurgy, 11)

aardvark	[4,13,…]
	..
zymurgy	[7,4]

Joins

```
class WordCmp(Planner):

    def wcPipe(fileName):
        return ReadLines(fileName) | Flatten(by=tokens) | Group(by=lambda x:x, reducingTo=

    wc1 = wcPipe('bluecorpus.txt')
    wc2 = wcPipe('redcorpus.txt')

    cmp = Join( Jin(wc1, by=lambda(word,n):word), Jin(wc2, by=lambda(word,n):word) ) \
          | ReplaceEach(by=lambda((word1,n1),(word2,n2)):(word1, score(n1,n2)))

    result = Format(cmp, by=lambda(word,blueScore):'%6.4f %s' % (blueScore,word))
```

Semantics – Hadoop backend

- Data structure can be converted to commands for streaming hadoop

```
(hadoop fs -test -e /user/wcohen/gpig_views/wordCount.gp \
        && hadoop fs -rmr /user/wcohen/gpig_views/wordCount.gp) \
|| echo no need to remove /user/wcohen/gpig_views/wordCount.gp

echo ...

hadoop jar /opt/cloudera/parcels/CDH/lib/hadoop-mapreduce/hadoop-streaming.jar  \
   -D mapred.reduce.tasks=5 \
   -file /Users/wcohen/Documents/code/GuineaPig/tutorial/guineapig.py \
   -file /Users/wcohen/Documents/code/GuineaPig/tutorial/longer-wordcount.py \
   -cmdenv PYTHONPATH=. \
   -input corpus.txt -output /user/wcohen/gpig_views/wordCount.gp \
   -mapper 'python longer-wordcount.py --view=wordCount --do=doGroupMap \
                --opts viewdir:/user/wcohen/gpig_views,target:hadoop'\
   -reducer 'python longer-wordcount.py --view=wordCount --do=doStoreRows \
                --opts viewdir:/user/wcohen/gpig_views,target:hadoop'
```

EXTENDED EXAMPLE: COMPUTING TFIDF IN GUINEA PIG

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 $\mathbf{u}(d)$, 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 $\mathbf{u}(y)$ 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

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

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

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

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

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

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

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

Implementation

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

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

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

#unweighted document vectors

udocvec1 = Join( Jin(data,by=lambda(docid,term):term), Jin(docFreq,by=lambda(term,df):term) )
udocvec2 = Map(udocvec1, by=lambda((docid,term1),(term2,df)):(docid,term1,df))
udocvec3 = Augment(udocvec2, sideview=ndoc, loadedBy=lambda v:GPig.onlyRowOf(v))
udocvec = Map(udocvec3, by=lambda((docid,term,df),(dummy,ndoc)):(docid,term,math.log(ndoc/df)))

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

docvec = Join( Jin(norm,by=lambda(docid,z):docid), Jin(udocvec,by=lambda(docid,term,weight):docid) ) \
         | Map( by=lambda((docid1,z),(docid2,term,weight)): (docid1,term,weight/math.sqrt(z)) )
```

Implementation

```
D = GPig.getArgvParams()
idDoc = ReadLines(D.get('corpus','idcorpus.txt')) | Map(by=lambda line:l
idWords = Map(idDoc, by=lambda (docid,doc): (docid,doc.lower().split()))
data = FlatMap(idWords, by=lambda (docid,words): map(lambda w:(docid,w),
```

docId	term
d123	found
d123	aardvark
…	…

(d123,found)
(d123,aardvark)
…

Implementation

```
D = GPig.getArgvParams()
idDoc = ReadLines(D.get('corpus','idcorpus.txt')) | Map(by=lambda line:l
idWords = Map(idDoc, by=lambda (docid,doc): (docid,doc.lower().split()))
data = FlatMap(idWords, by=lambda (docid,words): map(lambda w:(docid,w),
```

docId	term
d123	found
d123	aardvark

```
docFreq = Distinct(data) \
    | Group(by=lambda (docid,term):term, retaining=lambda(docid,term):docid,

                                        , reducingTo=ReduceToCount()
```

key	value
found	~~(d123,found),(d134,found),~~… 2456
aardvark	~~(d123,aardvark),~~… 7

Implementation

```
udocvec1 = Join( Jin(data,by=lambda(docid,term):term), Jin(docFreq,by=lambda(term,df):term) )
udocvec2 = Map(udocvec1, by=lambda((docid,term1),(term2,df)):(docid,term1,df))
```

docId	term
d123	found
d123	aardvark

key	value
found	2456
aardvark	7

('1', 'quite')
('1', 'a')
('1', 'difference.')
…
('3', 'alcohol')
…

('"94", 1)
('"94,", 1)
('"a", 1)
('"alcohol", 1)
…

(('2', '"alcohol"), ('"alcohol", 1))
(('550', '"cause"), ('"cause", 1))
…

Implementation

```
udocvec1 = Join( Jin(data,by=lambda(docid,term):term), Jin(docFreq,by=lambda(term,df):term) )
udocvec2 = Map(udocvec1, by=lambda((docid,term1),(term2,df)):(docid,term1,df))
```

docId	term	df
d123	found	2456
d123	aardvark	7

```
('2', '"confabulation'.", 2)
('3', '"confabulation'.", 2)
('209', '"controversy'", 1)
('181', '"em", 3)
('434', '"em", 3)
('452', '"em", 3)
('113', '"fancy", 1)
('212', '"franchise',", 1)
('352', '"honest,", 1)
```

Implementation: Map-side join

```
udocvec1 = Join( Jin(data,by=lambda(docid,term):term), Jin(docFreq,by=lambda(term,df):term) )
udocvec2 = Map(udocvec1, by=lambda((docid,term1),(term2,df)):(docid,term1,df))
udocvec3 = Augment(udocvec2, sideview=ndoc, loadedBy=lambda v:GPig.onlyRowOf(v))
udocvec = Map(udocvec3, by=lambda((docid,term,df),(dummy,ndoc)):(docid,term,math.log(ndoc/df))
```

Augment: loads a preloaded object b at mapper initialization time, cycles thru the input, and generates pairs (a,b)

docId	term	df	
d123	found	2456	Arbitrary python object
d123	aardvark	7	Arbitrary python object
…			

Implementation

```
udocvec1 = Join( Jin(data,by=lambda(docid,term):term), Jin(docFreq,by=lambda(term,df):term) )
udocvec2 = Map(udocvec1, by=lambda((docid,term1),(term2,df)):(docid,term1,df))
udocvec3 = Augment(udocvec2, sideview=ndoc, loadedBy=lambda v:GPig.onlyRowOf(v))
udocvec = Map(udocvec3, by=lambda((docid,term,df),(dummy,ndoc)):(docid,term,math.log(ndoc/df))
```

Augment: loads a preloaded object b at mapper initialization time, cycles thru the input, and generates pairs (a,b), where b **points** to the preloaded object

docId	term	df	
d123	found	2456	ptr
d123	aardvark	7	ptr
…			…

```
(('2', '"confabulation'.", 2), ('ndoc', 964))
(('3', '"confabulation'.", 2), ('ndoc', 964))
(('209', '"controversy"', 1), ('ndoc', 964))
(('181', '"em", 3), ('ndoc', 964))
(('434', '"em", 3), ('ndoc', 964))
```

Arbitrary python object

Implementation

```
udocvec1 = Join( Jin(data,by=lambda(docid,term):term), Jin(docFreq,by=lambda(term,df):term) )
udocvec2 = Map(udocvec1, by=lambda((docid,term1),(term2,df)):(docid,term1,df))
udocvec3 = Augment(udocvec2, sideview=ndoc, loadedBy=lambda v:GPig.onlyRowOf(v))
udocvec = Map(udocvec3, by=lambda((docid,term,df),(dummy,ndoc)):(docid,term,math.log(ndoc/df))
```

Augment: loads a preloaded object b at mapper initialization time, cycles thru the input, and generates pairs (a,b), where b **points** to the preloaded object

This looks like a join. But it's different.

- It's a single map, not a map-shuffle/sort-reduce
- The loaded object is paired with *every* a, not just ones where the join keys match (but you can use it for a map-side join!)
- The loaded object has to be *distributed* to every mapper (so, copied!)

```
(('2', "'confabulation'.", 2), ('ndoc', 964))
(('3', "'confabulation'.", 2), ('ndoc', 964))
(('209', "'controversy'", 1), ('ndoc', 964))
(('181', "'em", 3), ('ndoc', 964))
(('434', "'em", 3), ('ndoc', 964))
```

Implementation

```
udocvec1 = Join( Jin(data,by=lambda(docid,term):term), Jin(docFreq,by=lambda(term,df):term) )
udocvec2 = Map(udocvec1, by=lambda((docid,term1),(term2,df)):(docid,term1,df))
udocvec3 = Augment(udocvec2, sideview=ndoc, loadedBy=lambda v:GPig.onlyRowOf(v))
udocvec = Map(udocvec3, by=lambda((docid,term,df),(dummy,ndoc)):(docid,term,math.log(ndoc/df))
```

Gotcha: if you **store** an augment, it's printed on disk, and Python writes the **object pointed to**, not the pointer. So when you **store** you make a copy of the object *for every row*.

docId	term	df	
d123	found	2456	ptr
d123	aardvark	7	ptr
…			…

(('2', "'confabulation'.", 2), *printed-object*)
(('3', "'confabulation'.", 2), *printed-object*)
(('209', "'controversy'", 1), *printed-object*)
(('181', "'em", 3), *printed-object*)
(('434', "'em", 3), *printed-object*)

Arbitrary python object

```python
rom guineapig import *

# compute TFIDF in Guineapig

import sys
import math

class TFIDF(Planner):

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

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

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

    #unweighted document vectors

    udocvec1 = Join( Jin(data,by=lambda(docid,term):term), Jin(docFreq,by=lambda(term,df):term) )
    udocvec2 = Map(udocvec1, by=lambda((docid,term1),(term2,df)):(docid,term1,df))
    udocvec3 = Augment(udocvec2, sideview=ndoc, loadedBy=lambda v:GPig.onlyRowOf(v))
    udocvec = Map(udocvec3, by=lambda((docid,term,df),(dummy,ndoc)):(docid,term,math.log(ndoc/df)))

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

    docvec = Join( Jin(norm,by=lambda(docid,z):docid), Jin(udocvec,by=lambda(docid,term,weight):docid) ) \
             | Map( by=lambda((docid1,z),(docid2,term,weight)): (docid1,term,weight/math.sqrt(z)) )

# always end like this
if __name__ == "__main__":
    p = TFIDF()
    p.main(sys.argv)
```

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

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

TFIDF with map-side joins

```
class TFIDF(Planner):

    data = ReadLines('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 x:1, \
                reducingTo=ReduceToSum())

    # definitely use combiners when you aggregate
    ndoc = Map(data, by=lambda (docid,term):docid) \
        | Distinct() \
        | Group(by=lambda row:'ndoc', retaining=lambda x:1, combiningTo=ReduceToSum(), reducingTo=ReduceToSum())

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

TFIDF with map-side joins

```
class TFIDF(Planner):

    data = ReadLines('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 x:1, \
                reducingTo=ReduceToSum())

    # definitely use combiners when you aggregate
    ndoc = Map(data, by=lambda (docid,term):docid) \
        | Distinct() \
        | Group(by=lambda row:'ndoc', retaining=lambda x:1, combiningTo=ReduceToSum(), reducingTo=ReduceToSum())

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

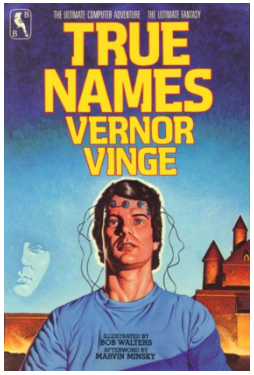
```
def loadAsDict(view):
    result = {}
    for (key,val) in GPig.rowsOf(view):
        result[key] = val
    return result
```

```
#compute unweighted document vectors with a map-side join
udocvec = Augment(data, sideview=inverseDocFreq, loadedBy=loadAsDict) \
    | 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=loadAsDict) \
    | Map( by=lambda ((docid,term,weight),normDict): (docid,term,weight/math.sqrt(normDict[docid])))
```

Similarity Joins



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.

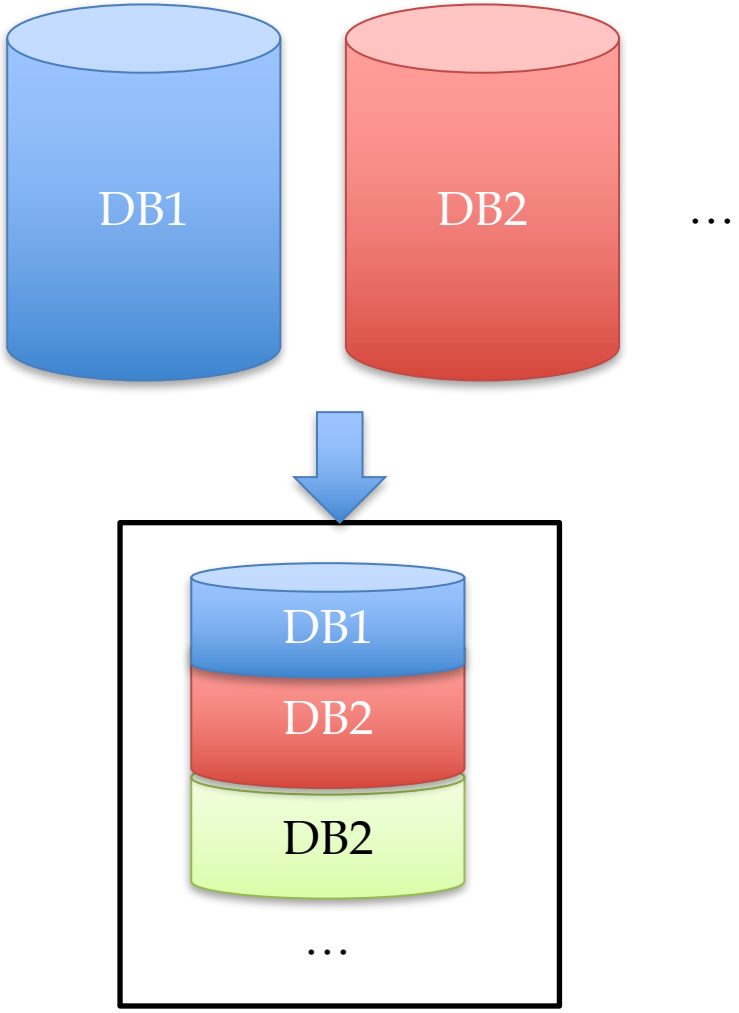
Outline: Soft Joins with TFIDF

- Why similarity joins are important
- Useful similarity metrics for sets and strings
- Fast methods for K-NN and similarity joins
  - ~~Blocking~~
  - ~~Indexing~~
  - ~~Short-cut algorithms~~
  - Parallel implementation

Motivation

- Integrating data is important
- Data from different sources may not have consistent *object identifiers*
  - Especially automatically-constructed ones
- But databases will have human-readable names and/or descriptions for the objects

- But matching names and descriptions is tricky….



Sim Joins on Product Descriptions

TFIDF weighting is often useful for matching 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.

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….

Example: 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
    ...
```

Example: 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
```

How well does TFIDF work?

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

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

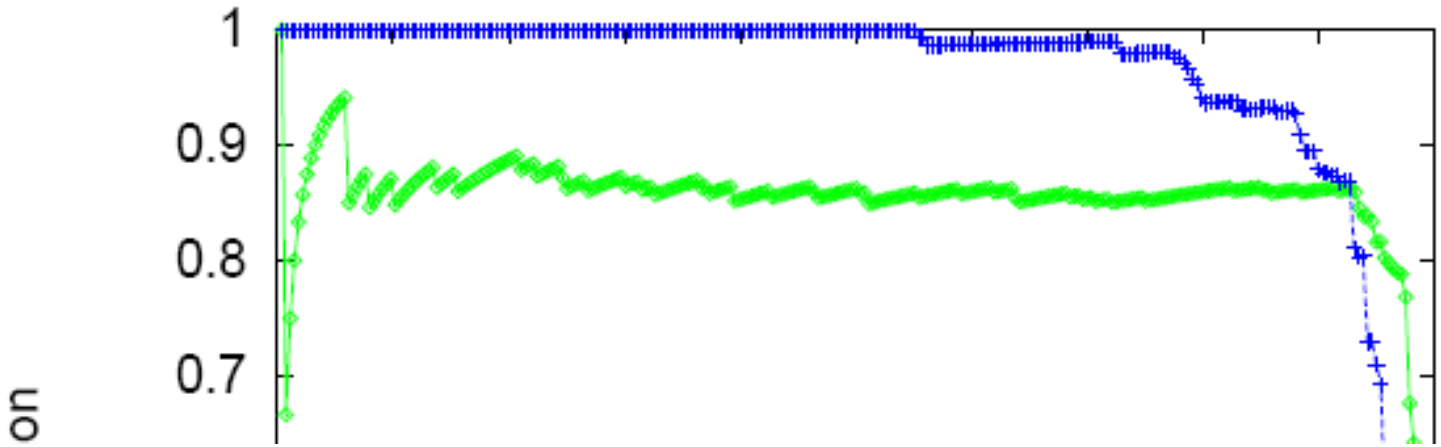


Table VI. Pairs of Names from the Hoovers and Iontech Relations

√	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
...
```

A parallel workflow for TFIDF similarity joins

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=ReduceToCount())

#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,df):rel ) \
    | ReplaceEach(by=lambda((rel,doc,term,df),(rel_,relCount)):(rel,doc,term,df,relCount)) \
    | 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)) )
```

sumSquareWeights

Statistics for computing TFIDF with IDFs local to each relation

$sim(\mathbf{v}(d_1),\mathbf{v}(d_2)) = \mathbf{v}(d_1)\cdot\mathbf{v}(d_2)$ 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

Making the algorithm smarter….

Inverted Index Softjoin - 2

```
# naive algorithm for the soft joint will 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,d
    | ReplaceEach(by=lambda((rel1,doc1,term,weight1),(rel2,doc2,term_,weight2)): (doc1,doc2,weig
    | Group(by=lambda(doc1,doc2,p):(doc1,doc2), reducingTo=sumOfP) \
    | ReplaceEach(by=lambda((doc1,doc2),sim):(doc1,doc2,sim))
```

we should make a **smart** choice about which terms to use

Adding heuristics to the soft join - 1

```
# 1) pick only top terms in each document
topTermsInEachDocForRel1 = Group(rel1Docs,
                                 by=lambda(rel,doc,term,weight):doc,
                                 retaining=lambda(rel,doc,term,weight):(weight,term)) \
    | ReplaceEach(by=lambda(doc,termList):sorted(termList,reverse=True)[0:NUM_TOP_TERMS]) \
    | Flatten(by=lambda x:x) | ReplaceEach(by=lambda(weight,term):term)


# 2) pick terms that have some minimal weight in their documents
highWeightTermsForRel1 = Filter(rel1Docs, by=lambda(rel,doc,term,weight):weight>=MIN_TERM_WEIGHT) \
    | ReplaceEach(by=lambda(rel,doc,term,weight):term)


# 3) pick terms with some maximal DF
lowDocFreqTerms = Filter(docFreq,by=lambda(rel,term,df):df<=MAX_TERM_DF)
    | ReplaceEach(by=lambda(rel,term,df):term)


# terms we will join on should pass all of the tests above
usefulTerms = Join( Jin(topTermsInEachDocForRel1), Jin(highWeightTermsForRel1)) \
    | ReplaceEach(by=lambda(term1,term2):term1) \
    | JoinTo( Jin(lowDocFreqTerms)) \
    | ReplaceEach(by=lambda(term1,term2):term1) | Distinct()
```

Adding heuristics to the soft join - 2

```
softjoin = Join( Jin(rel1Docs,by=lambda(rel,doc,term,weight):term),
                 Jin(usefulTerms)) \
    | ReplaceEach(by=lambda(rel1doc,term):rel1doc) \
    | JoinTo( Jin(rel2Docs,by=lambda(rel,doc,term,weight):term),
              by=lambda(rel,doc,term,weight):term)\
    | ReplaceEach( \
        by=lambda((rel1,doc1,term,weight1),(rel2,doc2,term_,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))
```

Adding heuristics

- Parks:
  - input 40k
  - data 60k
  - docvec 102k
  - softjoin
    - 539k tokens
    - 508k documents
    - 0 errors in top 50
- w/ heuristics:
  - input 40k
  - data 60k
  - docvec 102k
  - softjoin
    - 32k tokens
    - 24k documents
    - 3 errors in top 50
    - < 400 useful terms

Adding heuristics

- SO vs Wikipedia:
  - input 612M
  - docvec 1050M
  - softjoin 67M
- with heuristics
  - input 612M
  - docvec 1050M
  - softjoin 9.1M