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())

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

class variables in the planner are data structures
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

```bash
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=doGroupMap < corpus.txt
  | `LC_COLLATE=C sort -k1` 
  | python longer-wordcount.py --view=wordCount --do=doStoreRows > gpg_views/wordCount.gp
deploy steps in the compiled plan
invoke your script with special args
```
Grouping

Full Syntax for Group

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

<table>
<thead>
<tr>
<th>(today, 1)</th>
<th>aardvark</th>
<th>(aardvark, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, 1)</td>
<td>aardvark</td>
<td>(aardvark, 1)</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>(farmville, 1)</td>
<td>zymurgy</td>
<td>(zymurgy, 1)</td>
</tr>
<tr>
<td>...</td>
<td>zymurgy</td>
<td>(zymurgy, 1)</td>
</tr>
</tbody>
</table>
Grouping

Group(wc,  \textbf{by}=\texttt{lambda (word,count):word,}
\textbf{reducingTo}=\texttt{ReduceTo(}
\texttt{lambda: 0,}
\texttt{lambda accum,(_c):accum+c)} \quad \#init accum
\texttt{lambda accum,(_c):accum+c}) \quad \#update

<table>
<thead>
<tr>
<th>(today, 1)</th>
<th>aardvark</th>
<th>(aardvark, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i, 1)</td>
<td>aardvark</td>
<td>(aardvark, 1)</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td></td>
</tr>
<tr>
<td>(farmville, 1)</td>
<td>zymurgy</td>
<td>(zymurgy, 1)</td>
</tr>
<tr>
<td>..</td>
<td>zymurgy</td>
<td>(zymurgy, 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>aardvark</th>
<th>[(aardvark, 1),(aardvark,1),...]</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td></td>
</tr>
<tr>
<td>zymurgy</td>
<td>[(zymurgy, 1),... ]</td>
</tr>
</tbody>
</table>
Grouping

\[
\text{Group}(wc, \ \textbf{by}=\lambda (\text{word},\text{count}):\text{word}, \\
\text{retaining}=\lambda (\text{word},\text{count}):\text{count}, \\
\text{reducingTo}=\text{ReduceTo}( \\
\quad \lambda:\text{0}, \\
\quad \lambda \text{accum}, c: \text{accum}+c) \\
\]

| (today, 1)  
| (i, 1)     
| ...        
| (farmville, 1) 
| ...        
| aardvark   | [\textbf{aardvark}, 1, \textbf{aardvark}, 1, \ldots] | (aardvark, 214) 
| ..          | ...                                              | (absolute, 1)   
| zymurgy    | [\textbf{zymurgy}, 1, \ldots]                   | (zymurgy, 11)   
| aardvark   | [1, 1, \ldots]                                   
| ..          | ...                                              
| zymurgy    | [1, 1, \ldots]                                   

\#init accum 
\#update
Grouping

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

(today, 1)
(i, 1)
.. 
(farmville, 1)
...

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

<table>
<thead>
<tr>
<th>aardvark</th>
<th>[1,1,...]</th>
<th>aardvark</th>
<th>[4,13,...]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>..</td>
<td></td>
<td>..</td>
</tr>
<tr>
<td>zymurgy</td>
<td>[1,1,....]</td>
<td>zymurgy</td>
<td>[7,4]</td>
</tr>
</tbody>
</table>
class WordCmp(Planner):
    def wcPipe(fileName):
        return ReadLines(fileName) | Flatten(by=tokens) | Group(by=lambda x:x, reducingToDoc=True)
    wc1 = wcPipe('bluecorpus.txt')
    wc2 = wcPipe('redcorpus.txt')
    cmp = Join( Join(wc1, by=lambda(word,n):word), Join(wc2, by=lambda(word,n):word) ) \  
    | ReplaceEach(by=lambda((word1,n1),(word2,n2)):(word1, score(n1,n2)))
    result = Format(cmp, by=lambda(word,blueScore):'%.4f %s' % (blueScore,word))
Semantics – Hadoop backend

- Data structure can be converted to commands for streaming hadoop

```bash
(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

\[ DF(w) = \# \text{different docs } w \text{ occurs in} \]
\[ TF(w, d) = \# \text{different times } w \text{ occurs in doc } d \]

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

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

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

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

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

\[ \|u\|_2 = \sqrt{\sum_{i} u_i^2} \]
**TFIDF similarity**

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

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

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

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

\[
u(d) = \langle u(w_1, d), ..., u(w_{|V|}, d) \rangle
\]

\[
v(d) = \frac{u(d)}{\| u(d) \|_2}
\]

\[
sim(v(d_1), v(d_2)) = v(d_1) \cdot v(d_2) = \sum_w \frac{u(w, d_1)}{\| u(d_1) \|_2} \frac{u(w, d_2)}{\| u(d_2) \|_2}
\]
Implementation

D = GPin.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( Join(data,by=lambda (docid,term):term), Join(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:GPin.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( Join(norm,by=lambda (docid,z):docid), Join(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:
    idWords = Map(idDoc, by=lambda (docid,doc): (docid,doc.lower().split())))
data = FlatMap(idWords, by=lambda (docid,words): map(lambda w:(docid,w),

<table>
<thead>
<tr>
<th>docId</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td>d123</td>
<td>found</td>
</tr>
<tr>
<td>d123</td>
<td>aardvark</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(d123,found)
(d123,aardvark)
...


Implementation

D = GPig.getArgvParams()

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

docFreq = Distinct(data) |
    Group(by=lambda (docid,term):term, retaining=lambda (docid,term):docid,
          , reducingTo=ReduceToCount() 

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>found</td>
<td>(d123,found),(d134,found),... 2456</td>
</tr>
<tr>
<td>aardvark</td>
<td>(d123,aardvark),... 7</td>
</tr>
</tbody>
</table>
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))
```

<table>
<thead>
<tr>
<th>docId</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td>d123</td>
<td>found</td>
</tr>
<tr>
<td>d123</td>
<td>aardvark</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>found</td>
<td>2456</td>
</tr>
<tr>
<td>aardvark</td>
<td>7</td>
</tr>
</tbody>
</table>

('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))

<table>
<thead>
<tr>
<th>docId</th>
<th>term</th>
<th>df</th>
</tr>
</thead>
<tbody>
<tr>
<td>d123</td>
<td>found</td>
<td>2456</td>
</tr>
<tr>
<td>d123</td>
<td>aardvark</td>
<td>7</td>
</tr>
</tbody>
</table>

('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

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

<table>
<thead>
<tr>
<th>docId</th>
<th>term</th>
<th>df</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d123</td>
<td>found</td>
<td>2456</td>
<td>Arbitrary python object</td>
</tr>
<tr>
<td>d123</td>
<td>aardvark</td>
<td>7</td>
<td>Arbitrary python object</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>docId</th>
<th>term</th>
<th>df</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d123</td>
<td>found</td>
<td>2456</td>
<td>ptr</td>
</tr>
<tr>
<td>d123</td>
<td>aardvark</td>
<td>7</td>
<td>ptr</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| ('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

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

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.

<table>
<thead>
<tr>
<th>docId</th>
<th>term</th>
<th>df</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d123</td>
<td>found</td>
<td>2456</td>
<td>ptr</td>
</tr>
<tr>
<td>d123</td>
<td>aardvark</td>
<td>7</td>
<td>ptr</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(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)
from 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)
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.onlyRow0f(v)) \
    | Map(by=lambda((term,df),(dummy,ndoc)): (term,math.log(ndoc/df)))
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.onlyRow0f(v)) \
    | Map(by=lambda (term,df),(dummy,ndoc):(term,math.log(ndoc/df))

#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

- 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.

TFIDF weighting is often useful for matching descriptions.
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
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
  George Washington Carver NM:d499
  Salinas Pueblo Missions NM:d597
  Florissant Fossil Beds NM:d473
  Hagerman Fossil Beds NM:d517
  Gila Cliff Dwellings NM:d502
  Booker T. Washington NM:d423

similar

Obed Wild & Scenic River:d570
  Andersonville NHP:d401
  Sitka NHP:d606
Bering Land Bridge N. Preserve:d413
  Sequoia & Kings Canyon NP:d603
  Glacier Bay NP & Preserve:d643
  NP of American Samoa:d561
  Kalaupapa NHS:d538

less similar

Lake Mead NRA:d545
  Upper Delaware Scenic & Rec. River:d617

Lake Mead NRA (Nevada):d224
  Upper Delaware Scenic & Recreational River:d368
How well does TFIDF work?

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td>$a_1$</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>$a_2$</td>
</tr>
<tr>
<td>3</td>
<td>✗</td>
<td>$a_3$</td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>$a_4$</td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>$a_5$</td>
</tr>
<tr>
<td>6</td>
<td>✓</td>
<td>$a_6$</td>
</tr>
<tr>
<td>7</td>
<td>✗</td>
<td>$a_7$</td>
</tr>
<tr>
<td>8</td>
<td>✓</td>
<td>$a_8$</td>
</tr>
<tr>
<td>9</td>
<td>✓</td>
<td>$a_9$</td>
</tr>
<tr>
<td>10</td>
<td>✗</td>
<td>$a_{10}$</td>
</tr>
<tr>
<td>11</td>
<td>✗</td>
<td>$a_{11}$</td>
</tr>
<tr>
<td>12</td>
<td>✓</td>
<td>$a_{12}$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>✔</th>
<th>Texas Instruments Incorporated</th>
<th>TEXAS INSTRUMENTS INC</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>The New York Times Company</td>
<td>NEW YORK TIMES CO</td>
</tr>
<tr>
<td>✔</td>
<td>Campo Electronics, Appliances and Computers, Inc.</td>
<td>CAMPO ELECTRONICS APPLIANCES</td>
</tr>
<tr>
<td>✔</td>
<td>Cascade Communications Corp.</td>
<td>CASCADE COMMUNICATION</td>
</tr>
<tr>
<td>✔</td>
<td>The McGraw-Hill Companies, Inc.</td>
<td>MCGRAW-HILL CO</td>
</tr>
<tr>
<td>✔</td>
<td>U S WEST Communications Group</td>
<td>U S WEST INC</td>
</tr>
<tr>
<td>✗</td>
<td>Silicon Valley Group, Inc.</td>
<td>SILICON VALLEY RESEARCH INC</td>
</tr>
<tr>
<td>✗</td>
<td>The Reynolds and Reynolds Company</td>
<td>REYNOLDS &amp; REYNOLDS CO</td>
</tr>
<tr>
<td>✔</td>
<td>InTime Systems International, Inc.</td>
<td>INTIME SYSTEMS INTERNATIONAL</td>
</tr>
</tbody>
</table>
Table V. Average Precision for Similarity Joins

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

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

<table>
<thead>
<tr>
<th>Rank</th>
<th>Similarity</th>
<th>Left Name</th>
<th>Right Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>Agate Fossil Beds NM</td>
<td>Agate Fossil Beds NM</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
<td>Big Bend NP</td>
<td>Big Bend NP</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Gateway NRA</td>
<td>Gateway NRA</td>
</tr>
<tr>
<td>194</td>
<td>1.00</td>
<td>Gulf Islands NS</td>
<td>Gulf Island NS</td>
</tr>
<tr>
<td>195</td>
<td>0.99</td>
<td>Rainbow Bridge NM</td>
<td>Rainbow Bridges NM</td>
</tr>
<tr>
<td>196</td>
<td>0.99</td>
<td>Whiskeytown Shasta Trinity NRA</td>
<td>Whiskey-Shasta-Trinity NRA</td>
</tr>
<tr>
<td>197</td>
<td>0.98</td>
<td>Capitol Reef NP</td>
<td>Capital Reef NP</td>
</tr>
<tr>
<td>199</td>
<td>0.95</td>
<td>Timpanogos Cave NM</td>
<td>Timpanogas Caves NM</td>
</tr>
<tr>
<td>200</td>
<td>0.94</td>
<td>War in the Pacific NHP</td>
<td>War in Pacific NHP</td>
</tr>
<tr>
<td>201</td>
<td>0.94</td>
<td>Chesapeake &amp; Ohio Canal NHP</td>
<td>Chesapeake and Ohio Canal NHP</td>
</tr>
<tr>
<td>203</td>
<td>0.92</td>
<td>Saguaro NP</td>
<td>Saguaro NM</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Aniakchak NM &amp; NPRES</td>
<td>Aniakchak NM</td>
</tr>
<tr>
<td>210</td>
<td>0.88</td>
<td>National Park Of American Samoa</td>
<td>NP of American Samoa</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Pu'uhonua a Honaunau NHP</td>
<td>Pu'uhonua O Honaunau NHP</td>
</tr>
<tr>
<td>224</td>
<td>0.76</td>
<td>Bering Land Bridge NPRES</td>
<td>Bering Land Bridge N. Preserve</td>
</tr>
<tr>
<td>225</td>
<td>0.75</td>
<td>Yukon Charley Rivers NPRES</td>
<td>Yukon-Charley Rivers N. Preserve</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Wolf Trap Farm Park for the Performing Arts</td>
<td>Wolf Trap Farm Park</td>
</tr>
<tr>
<td>241</td>
<td>0.69</td>
<td>Fredericksburg and Spotsylvania County Battlefields Memorial NMP</td>
<td>Fredericksburg &amp; Spotsylvania NMP</td>
</tr>
<tr>
<td>242</td>
<td>0.69</td>
<td>Great Smoky Mtn. NP</td>
<td>Great Smoky Mountains NP</td>
</tr>
<tr>
<td>243</td>
<td>0.67</td>
<td>Mount Rushmore NM</td>
<td>Mount Rushmore N. Mem.</td>
</tr>
<tr>
<td>245</td>
<td>0.67</td>
<td>Chattahoochee NSR</td>
<td>Chattahoochee River NRA</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A parallel workflow for TFIDF similarity joins
#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( Join(data, by=lambda(rel,docid,term):(rel,term)), 
    Join(docFreq, by=lambda(rel,term,df):(rel,term))) \\
    | ReplaceEach(by=lambda((rel,doc,term),(rel_,term_,df)):(rel,doc,term,df)) \\
    | JoinTo( Join(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 acc,rel,doc,term,weight: acc+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( Join(norm,by=lambda(rel,doc,z):(rel,doc)), 
    Join(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))))

Statistics for computing TFIDF with IDFs local to each relation
What’s the algorithm?

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

\[
sim(v(d_1), v(d_2)) = v(d_1) \cdot v(d_2)
\]
Making the algorithm smarter....
we should make a **smart** choice about which terms to use
# 1) pick only top terms in each document

```python

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

```python

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

```python

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

```python

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

```c
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