Your code looks like song lyrics written using only the stuff that comes after the question mark in a URL.

It's like a JSON table of model numbers for flashlights with "tactical" in their names.

Like you read Turing's 1936 paper on computing and a page of Javascript example code and guessed at everything in between.

It's like a leet-speak translation of a manifesto by a survivalist cult leader who's for some reason obsessed with memory allocation.

I can get someone else to review my code.

Not more than once, I bet.

Workflows and Abstractions for Map-Reduce
Recap

• Map-reduce ✔
• Algorithms with multiple map-reduce steps
  – Naïve bayes test routine for large datasets and large models
• Cleanly describing these algorithms
  – workflow (or dataflow) languages
  – abstract operations: map, filter, flatten, group, join, ...
  – PIG: one such language
PIG

• Released in 2008
• Wordcount program:

```bash
A = load '/tmp/bible+shakes.nopunc';
B = foreach A generate flatten(TOKENIZE((chararray)$0)) as word;
C = filter B by word matches '\w+';
D = group C by word;
E = foreach D generate COUNT(C) as count, group as word;
F = order E by count desc;
store F into '/tmp/wc';
```
GUINEA PIG
GuineaPig: PIG in Python

- Pure Python (< 1500 lines)
- Streams Python data structures
  - strings, numbers, tuples (a,b), lists [a,b,c]
  - No records: operations defined functionally
- Compiles to Hadoop streaming pipeline
  - Optimizes sequences of MAPs
- Runs locally without Hadoop
  - compiles to stream-and-sort pipeline
  - intermediate results can be viewed
- Can easily run parts of a pipeline
- [http://curtis.ml.cmu.edu/w/courses/index.php/Guinea_Pig](http://curtis.ml.cmu.edu/w/courses/index.php/Guinea_Pig)
GuineaPig: PIG in Python

- Pure Python, streams Python data structures
  - not too much new to learn (eg field/record notation, special string operations, UDFs, ...)
  - codebase is small and readable
- Compiles to Hadoop or stream-and-sort, can easily run parts of a pipeline
  - intermediate results often are (and always can be) stored and inspected
  - plan is fairly visible
- Syntax includes high-level operations but also fairly detailed description of an optimized map-reduce step
  - Flatten | Group(by=..., retaining=..., reducingTo=...)
A wordcount example

```python
# always start like this
from guineapig import *
import sys

# supporting rou
def tokens(line):
    for tok in line:
        yield tok

#always subclass
class WordCount(Planner):

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

# always end like this
if __name__ == '__main__':
    WordCount().main(sys.argv)
```

Class variables in the planner are data structures.
Wordcount example ....

- 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

```plaintext
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

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 \
> gpg_views/wordCount.gp
```
Wordcount example ....

- 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'
```
Wordcount example ....

- Of course you won’t access local files with Hadoop, so you need to specify an HDFS location for inputs and outputs

```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'
```
Wordcount example ....

- Of course you won’t access local files with Hadoop, so you need to specify an HDFS location for inputs and outputs.

def __main__(sys.argv):
    WordCount().main(sys.argv)

    python param-wordcount.py
      --plan wc
      --opts target:hadoop,viewdir:/user/wcohen/gpviews
      --params corpus:/user/wcohen/sharded-corpus
More examples of GuineaPig

Join syntax, macros, Format command

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

wc1 = wcPipe('bluecorpus.txt')
w2 = wcPipe('redcorpus.txt')

cmp = Join( Join(w1, by=lambda(word,n):word), Join(w2, 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))

Incremental debugging, when intermediate views are stored:

% python wrdcmp.py –store result
...
% python wrdcmp.py –store result –reuse cmp
More examples of GuineaPig

Full Syntax for Group

```
Group(wc,  by=lambda (word,count):word[:k],
       retaining=lambda (word,count):count,
       combiningTo=ReduceToSum(),
       reducingTo=ReduceToSum())
equiv to:
Group(wc,  by=lambda (word,count):word[:k],
       reducingTo=
       ReduceTo(int,
               lambda accum,word,count): accum+count))
```
ANOTHER EXAMPLE: COMPUTING TFIDF IN GUINEA PIG
Implementation

D = GPig.getArgvParams()
dDoc = ReadLines(D.get('corpus', 'idcorpus.txt')) | Map(by=lambda line: line.strip().split('\\t'))
dIDWords = Map(dIDoc, by=lambda (docid, doc): (docid, doc.lower().split()))
data = FlatMap(dIDWords, 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())
dDocIds = Map(data, by=lambda (docid, term): docid) | Distinct()
ndoc = Group(dDocIds, 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: 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( 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 = GPin.getArgvParam()
idooc = 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>docld</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:l)
idWords = Map(idDoc, by=lambda (docid,doc): (docid,doc.lower().split()))
data = FlatMap(idWords, by=lambda (docid,words): map(lambda w:(docid,w),

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

<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>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>d123</td>
<td>found</td>
<td>found</td>
<td>2456</td>
</tr>
<tr>
<td>d123</td>
<td>aardvark</td>
<td>aardvark</td>
<td>7</td>
</tr>
</tbody>
</table>

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

('2', '"alcohol"'), ('"alcohol", 1))
('550', '"cause"'), ('"cause", 1))
...
### Implementation

```python
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>Arbiary python object</th>
</tr>
</thead>
<tbody>
<tr>
<td>d123</td>
<td>found</td>
<td>2456</td>
<td></td>
</tr>
<tr>
<td>d123</td>
<td>aardvark</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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

<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>
<tr>
<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))

Arbitrary python object
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))
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.
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('
'))
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.onlyRow0f(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)
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, \n    retaining=lambda x:1, \n    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))
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
doc = 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:GPinG.rowsOf(view)) 
  | 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])))

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

Another problem to attack with dataflow
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
SOFT JOINS WITH TFIDF: WHY AND WHAT
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 for the objects
• But names are tricky....
<table>
<thead>
<tr>
<th>Humongous</th>
<th>Humongous Entertainment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Headbone</td>
<td>Headbone Interactive</td>
</tr>
<tr>
<td>The Lion King:</td>
<td>Lion King Animated StoryBook</td>
</tr>
<tr>
<td>Storybook</td>
<td></td>
</tr>
<tr>
<td>Disney’s Activity</td>
<td>The Lion King Activity Center</td>
</tr>
<tr>
<td>Center, The</td>
<td></td>
</tr>
<tr>
<td>Lion King</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Microsoft</td>
</tr>
<tr>
<td></td>
<td>Microsoft Kids</td>
</tr>
<tr>
<td></td>
<td>Microsoft/Scholastic</td>
</tr>
<tr>
<td></td>
<td>American Kestrel</td>
</tr>
<tr>
<td>Kestrel</td>
<td>Eurasian Kestrel</td>
</tr>
<tr>
<td>Canada Goose</td>
<td>Goose, Aleutian Canada</td>
</tr>
<tr>
<td>Mallard</td>
<td>Mallard, Mariana</td>
</tr>
</tbody>
</table>
Sim Joins on Product Descriptions

• Similarity can be **high** for descriptions of **distinct** items:
  
  o 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 ...
  
  o 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:
  
  o 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 ...
  
  o 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 NHP
Aniakchak NM
Antietam NB
Apostle Islands NL
Appomattox Courthouse 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

Acadia NP
Allegheny Portage Railroad NHS
American Memorial Park
Amistad NRA
Andersonville NHP
Aniakchak NM
Antietam NB
Apostle Islands NL
Appomattox Courthouse 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

lesser similar

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

Lake Mead NRA (Nevada):d224
  Upper Delaware Scenic & Recreational River:d368

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

top500:
Abbott Laboratories
Able Telcom Holding Corp.
Access Health, Inc.
Acclaim Entertainment, Inc.
Ace Hardware Corporation
ACS Communications, Inc.
ACT Manufacturing, Inc.
Active Voice Corporation
Adams Media Corporation
Adolph Coors Company
...

hiTech:
ACC CORP
ADC TELECOMMUNICATION INC
ADELPHIA COMMUNICATIONS CORP
ADT LTD
ADTRAN INC
AIRTOUCH COMMUNICATIONS
AMATI COMMUNICATIONS CORP
AMERITECH CORP
APERATUS TECHNOLOGIES INC
APPLIED DIGITAL ACCESS INC
APPLIED INNOVATION INC

A useful scalable similarity metric: IDF weighting plus cosine distance!
How well does TFIDF work?

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>$a_2$</td>
<td>$b_2$</td>
</tr>
<tr>
<td>3</td>
<td>×</td>
<td>$a_3$</td>
<td>$b_3$</td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>$a_4$</td>
<td>$b_4$</td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>$a_5$</td>
<td>$b_5$</td>
</tr>
<tr>
<td>6</td>
<td>✓</td>
<td>$a_6$</td>
<td>$b_6$</td>
</tr>
<tr>
<td>7</td>
<td>×</td>
<td>$a_7$</td>
<td>$b_7$</td>
</tr>
<tr>
<td>8</td>
<td>✓</td>
<td>$a_8$</td>
<td>$b_8$</td>
</tr>
<tr>
<td>9</td>
<td>✓</td>
<td>$a_9$</td>
<td>$b_9$</td>
</tr>
<tr>
<td>10</td>
<td>×</td>
<td>$a_{10}$</td>
<td>$b_{10}$</td>
</tr>
<tr>
<td>11</td>
<td>×</td>
<td>$a_{11}$</td>
<td>$b_{11}$</td>
</tr>
<tr>
<td>12</td>
<td>✓</td>
<td>$a_{12}$</td>
<td>$b_{12}$</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</td>
<td>CAMPO ELECTRONICS</td>
</tr>
<tr>
<td></td>
<td>and Computers, Inc.</td>
<td>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 I</td>
</tr>
</tbody>
</table>
There are refinements to TFIDF distance – e.g., ones that extend with soft matching at the token level (e.g., softTFIDF)

<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>
distance is '[JaroWinklerTFIDF:threshold=0.9]'  
Pairs: 6806 Correct: 250  
Matching time: 0.278

| +   | 1.00 | Agate Fossil Beds NM | Agate Fossil Beds NM |
|     |      | Big Bend NP          | Big Bend NP          |
| ... |
| +   | 1.00 | Gateway NRA          | Gateway NRA          |
| +   | 0.99 | Gulf Islands NS      | Gulf Island NS       |
| +   | 0.99 | Rainbow Bridge NM    | Rainbow Bridges NM   |
| +   | 0.98 | Whiskeytown Shasta Trinity NRA | Whiskey-Shasta-Trinity NRA |
| +   | 0.97 | Capitol Reef NP      | Capital Reef NP      |
| +   | 0.95 | Timpanogos Cave NM   | Timpanogas Caves NM  |
| +   | 0.94 | War in the Pacific NHP | War in Pacific NHP   |
| +   | 0.94 | Chesapeake & Ohio Canal NHP | Chesapeake and Ohio Canal NHP |
| +   | 0.92 | Saguaro NP           | Saguaro NM           |
| ... |
| +   | 0.88 | Aniakchak NM & NPRES | Aniakchak NM         |
| +   | 0.86 | National Park Of American Samoa | NP of American Samoa |
| ... |
| +   | 0.76 | Pu'uhonua a Honaunau NHP | Pu'uhonua O Honaunau NHP |
| +   | 0.75 | Bering Land Bridge NPRES | Bering Land Bridge N. Preserve |
| +   | 0.75 | Yukon Charley Rivers NPRES | Yukon-Charley Rivers N. Preserve |
| ... |
| +   | 0.69 | Wolf Trap Farm Park for the Performing Arts | Wolf Trap Farm Park |
| +   | 0.69 | Fredericksburg and Spotsylvania County Battlefields Memorial NMP | Fredericksburg & Spotsylvania NMP |
| +   | 0.69 | Great Smoky Mtn. NP | Great Smoky Mountains NP |
| +   | 0.67 | Mount Rushmore NM | Mount Rushmore N. Mem. |
| +   | 0.67 | Chattahoochee NSR | Chattahoochee River NRA |
William W. Cohen

Carnegie Mellon University
machine learning, information integration, information extraction, intelligent tutoring, natural language processing
Verified email at cs.cmu.edu - Homepage
My profile is public

Fast Effective Rule Induction

- WW Cohen
- Proceedings of the Twelfth International Conference on Machine Learning...

A comparison of string metrics for matching names and records

- W Cohen, P Ravikumar, S Fienberg
- Kdd workshop on data cleaning and object consolidation 3, 73-78

Recommendation as classification: Using social and content-based information in recommendation

- C Basu, H Hirsh, W Cohen
- AAAI/IAAI, 714-720
SOFT JOINS WITH TFIDF: HOW?
Rocchio’s algorithm

\[ DF(w) = \# \text{ different docs } w \text{ occurs in} \]
\[ TF(w, d) = \# \text{ different times } w \text{ occurs in doc } d \]
\[ IDF(w) = \frac{|D|}{DF(w)} \]
\[ u(w, d) = \log(TF(w, d) + 1) \cdot \log(IDF(w)) \]
\[ u(d) = \langle u(w_1, d), \ldots, u(w_{|V|}, d) \rangle \]
\[ u(y) = \alpha \frac{1}{|C_y|} \sum_{d \in C_y} \frac{u(d)}{\|u(d)\|_2} - \beta \frac{1}{|D - C_y|} \sum_{d' \in D - C_y} \frac{u(d')}{\|u(d')\|_2} \]
\[ f(d) = \arg \max_{y} \frac{u(d)}{\|u(d)\|_2} \cdot \frac{u(y)}{\|u(y)\|_2} \]
\[ \|u\|_2 = \sqrt{\sum_{i} u_i^2} \]

Many variants of these formulae

…as long as \( u(w, d) = 0 \) for words not in \( d \! \)

Store only non-zeros in \( u(d) \), so size is \( \mathcal{O}(|d|) \)

But size of \( u(y) \) is \( \mathcal{O}(|n_v|) \)
TFIDF similarity

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

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

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

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

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

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

\[ sim(v(d_1), v(d_2)) = v(d_1) \cdot v(d_2) = \sum_{w} \frac{u(w,d_1)}{\| u(d_1) \|_2} \frac{u(w,d_2)}{\| u(d_2) \|_2} \]
Soft TFIDF 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

loosely adapted
Parallel Inverted Index Softjoin - 1

Statistics for computing TFIDF with IDFs local to each relation

want this to work for long documents or short ones...and keep the relations simple
What’s the algorithm?

• Step 1: create document vectors as \((C_d, d, \text{term, weight})\) 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
An alternative TFIDF pipeline

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

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])))
```

51
Parallel versus serial

• Names of tags from Stackoverflow vs Wikipedia concepts:
  – input 612M, 7.1M entities
  – docvec 1050M
  – softjoin 67M, 1.5M pairs
  – wallclock time 24min
    • 25 processes on in-memory map-reduce
    • called ”mrs_gp”
  – wall clock time for SecondString
    • 3-4 days

  – (preliminary experiments)
The same code in PIG
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);
doc = 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

-- naive algorithm: use all terms for finding potential 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';
<table>
<thead>
<tr>
<th>Score</th>
<th>Code</th>
<th>Name</th>
<th>Code</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.99436717611623</td>
<td>d00059</td>
<td>Carl Sandburg Home NHS</td>
<td>d00436</td>
<td>Carl Sandburg Home NHS</td>
</tr>
<tr>
<td>0.9937688379278058</td>
<td>d00354</td>
<td>Theodore Roosevelt NP</td>
<td>d00611</td>
<td>Theodore Roosevelt NP</td>
</tr>
<tr>
<td>0.9920648281782544</td>
<td>d00286</td>
<td>Oregon Caves NM</td>
<td>d00573</td>
<td>Oregon Caves NM</td>
</tr>
<tr>
<td>0.99140777975044103</td>
<td>d00274</td>
<td>New River Gorge NR</td>
<td>d00566</td>
<td>New River Gorge NR</td>
</tr>
<tr>
<td>0.9881961852455996</td>
<td>d00009</td>
<td>American Memorial Park</td>
<td>d00399</td>
<td>American Memorial Park</td>
</tr>
<tr>
<td>0.9878514547862078</td>
<td>d00154</td>
<td>George Washington Memorial Parkway</td>
<td>d00500</td>
<td>George Washington Memorial Parkway</td>
</tr>
<tr>
<td>0.9422676645498852</td>
<td>d00376</td>
<td>War in the Pacific NHP</td>
<td>d00623</td>
<td>War in the Pacific NHP</td>
</tr>
<tr>
<td>0.92307133361005</td>
<td>d00323</td>
<td>Saguaro NP</td>
<td>d00594</td>
<td>Saguaro NP</td>
</tr>
<tr>
<td>0.8914304226443976</td>
<td>d00292</td>
<td>Pea Ridge NHS</td>
<td>d00577</td>
<td>Pea Ridge NMP</td>
</tr>
<tr>
<td>0.890829830425262</td>
<td>d00200</td>
<td>Jean Lafitte NHP &amp; NPRES</td>
<td>d00532</td>
<td>Jean Lafitte NHP &amp; Preserve</td>
</tr>
<tr>
<td>0.8873463623037525</td>
<td>d00283</td>
<td>Obed Wild and Scenic River</td>
<td>d00570</td>
<td>Obed Wild &amp; Scenic River</td>
</tr>
<tr>
<td>0.8838421147370811</td>
<td>d00342</td>
<td>Sitka NHS</td>
<td>d00606</td>
<td>Sitka NHS</td>
</tr>
<tr>
<td>0.88342114737081</td>
<td>d00011</td>
<td>Andersonville NHS</td>
<td>d00401</td>
<td>Andersonville NHP</td>
</tr>
<tr>
<td>0.870042867436217</td>
<td>d00026</td>
<td>Bering Land Bridge NPRES</td>
<td>d00413</td>
<td>Bering Land Bridge N. Preserve</td>
</tr>
<tr>
<td>0.8684330361522184</td>
<td>d00157</td>
<td>Glacier Bay NP &amp; N PRES</td>
<td>d00643</td>
<td>Glacier Bay N. &amp; N.PRES</td>
</tr>
<tr>
<td>0.8680495192463105</td>
<td>d00339</td>
<td>Sequoia and Kings Canyon NP</td>
<td>d00603</td>
<td>Sequoia &amp; Kings Canyon NP</td>
</tr>
<tr>
<td>0.8660286476353838</td>
<td>d00267</td>
<td>National Park Of American Samoa NP</td>
<td>d00561</td>
<td>National Park Of American Samoa NP</td>
</tr>
<tr>
<td>0.859311749780314</td>
<td>d00210</td>
<td>Kalaupapa NHP</td>
<td>d00538</td>
<td>Kalaupapa NHP</td>
</tr>
<tr>
<td>0.850226387429363</td>
<td>d00208</td>
<td>Johnstown Flood NM</td>
<td>d00536</td>
<td>Johnstown Flood N. Mem.</td>
</tr>
<tr>
<td>0.8424859579540737</td>
<td>d00222</td>
<td>Lake Clark NP &amp; N PRES</td>
<td>d00646</td>
<td>Lake Clark NP &amp; N Preserve</td>
</tr>
<tr>
<td>0.8398407018438242</td>
<td>d00187</td>
<td>Homestead National Monument of America N.</td>
<td>d00523</td>
<td>Homestead National Monument of America N.</td>
</tr>
<tr>
<td>0.8395526626941698</td>
<td>d00230</td>
<td>Lincoln Boyhood NM</td>
<td>d00548</td>
<td>Lincoln Boyhood N. Mem.</td>
</tr>
<tr>
<td>0.8390553468895996</td>
<td>d00349</td>
<td>Sunset Crater NM</td>
<td>d00610</td>
<td>Sunset Crater Volcano NM</td>
</tr>
<tr>
<td>0.8344604123961857</td>
<td>d00259</td>
<td>Mount Rushmore NM</td>
<td>d00559</td>
<td>Mount Rushmore N. Mem.</td>
</tr>
<tr>
<td>0.8313853772896841</td>
<td>d00353</td>
<td>Theodore Roosevelt Island</td>
<td>d00611</td>
<td>Theodore Roosevelt Island</td>
</tr>
<tr>
<td>0.8301435671019225</td>
<td>d00071</td>
<td>Chesapeake &amp; Ohio Canal NHP</td>
<td>d00444</td>
<td>Chesapeake and Ohio Canal NHP</td>
</tr>
<tr>
<td>0.82492593820652</td>
<td>d00019</td>
<td>Arkansas Post NM</td>
<td>d00407</td>
<td>Arkansas Post N. Mem.</td>
</tr>
<tr>
<td>0.8282902347497227</td>
<td>d00212</td>
<td>Katmai NP &amp; N PRES</td>
<td>d00644</td>
<td>Katmai NP &amp; N Preserve</td>
</tr>
<tr>
<td>0.8202902347497227</td>
<td>d00029</td>
<td>Denali NP &amp; N PRES</td>
<td>d00464</td>
<td>Denali NP &amp; N Preserve</td>
</tr>
<tr>
<td>0.796547902996782</td>
<td>d00013</td>
<td>Aniakchak NM &amp; N PRES</td>
<td>d00402</td>
<td>Aniakchak NM</td>
</tr>
<tr>
<td>0.7835432589199314</td>
<td>d00031</td>
<td>Big Thicket N PRES</td>
<td>d00417</td>
<td>Big Thicket N. Preserve</td>
</tr>
<tr>
<td>0.7835432589199314</td>
<td>d00028</td>
<td>Big Cypress N PRES</td>
<td>d00415</td>
<td>Big Cypress N. Preserve</td>
</tr>
</tbody>
</table>
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 -r mr phirl/docvec
STORE docvec INTO 'phirl/docvec';

-- naive algorithm: use all terms for finding potential 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 for the soft join 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,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), reducingTo=sumOfP)  
    | ReplaceEach(by=lambda((doc1,doc2),sim):(doc1,doc2,sim))

- this join is where it can get expensive
- if a term appears in N docs in rel1 and M docs in rel2 then you get N*M tuples in the join
- but frequent terms don't count much...
- we should make a **smart** choice about which terms to use
# 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)
Adding heuristics to the soft join - 2

```plaintext
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 accum, (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