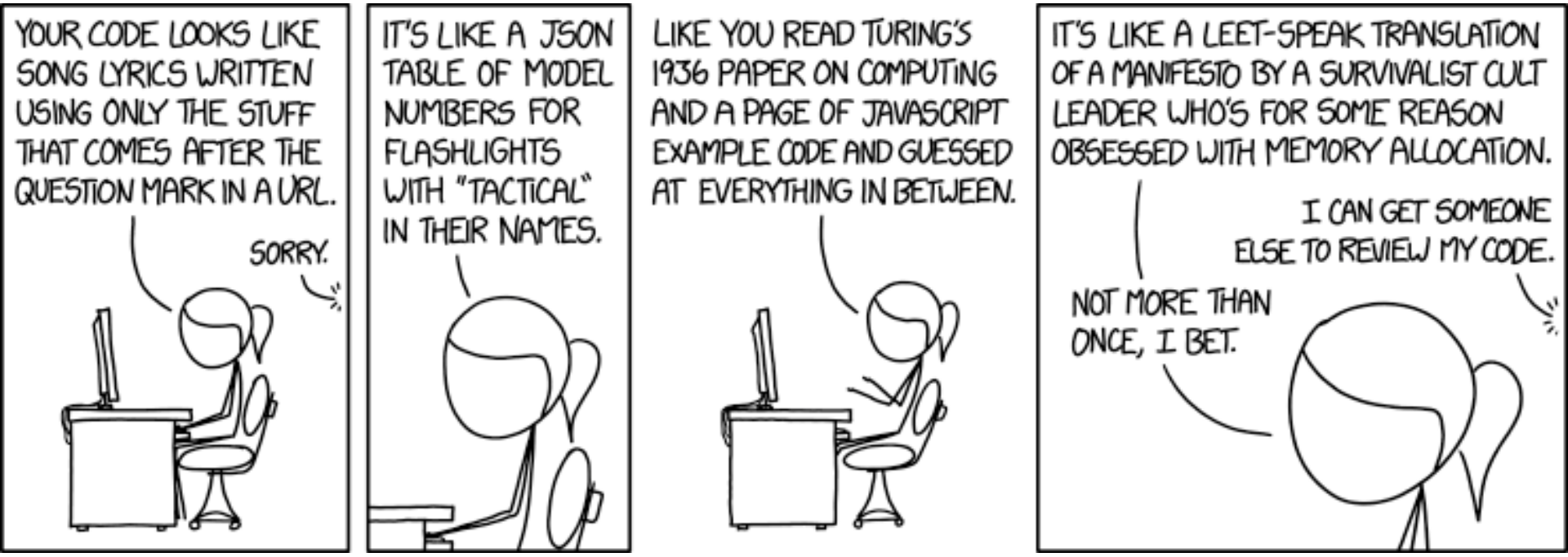


Workflows and Abstractions for Map-Reduce

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:

```
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

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

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

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

#always subclass
class WordCount(

    wc = ReadLines('corpus.txt') | Flatten(by=tokens) | Group(by=lambda x:x, reducing
                                                          h=ReduceToCount())

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

```
ReduceTo(int,by=lambda accum,val:accum+1)
```

```
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")
```

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

```
========================================================================
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                    ort -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
```

Wordcount example ….

- 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'
```

Wordcount example ….

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

```
(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

```
#always subclass Planner
class WordCount(Planner):

    D = GPig.getArgvParams(required=['corpus'])
    wc = ReadLines(D['corpus']) | Flatten(by=tokens) | Group(by=lambda x:x, reducin

# always end like this
if __name__ == "__main__":
    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')
    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))
```

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()
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))
```

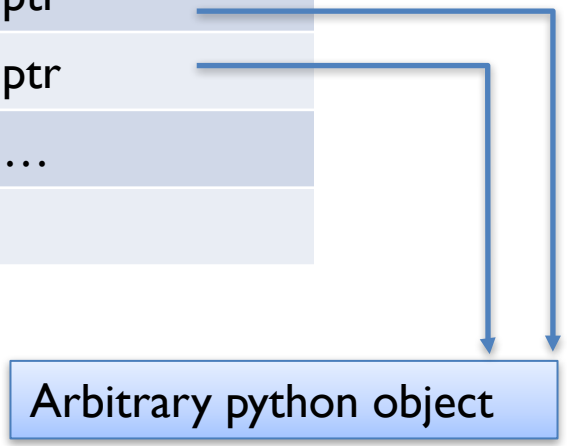
Implementation

```
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*)



```
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)
```

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
```

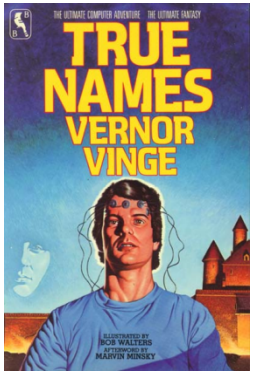
```
#compute unweighted document vectors with a map-side join
udocvec = Augment(data, sideview=inverseDocFreq, loadedBy=loadAsDict) \
    | Map(by=lambda ((docid,term),idfDict):(docid,term,idfDict[term]))

#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])))
```

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

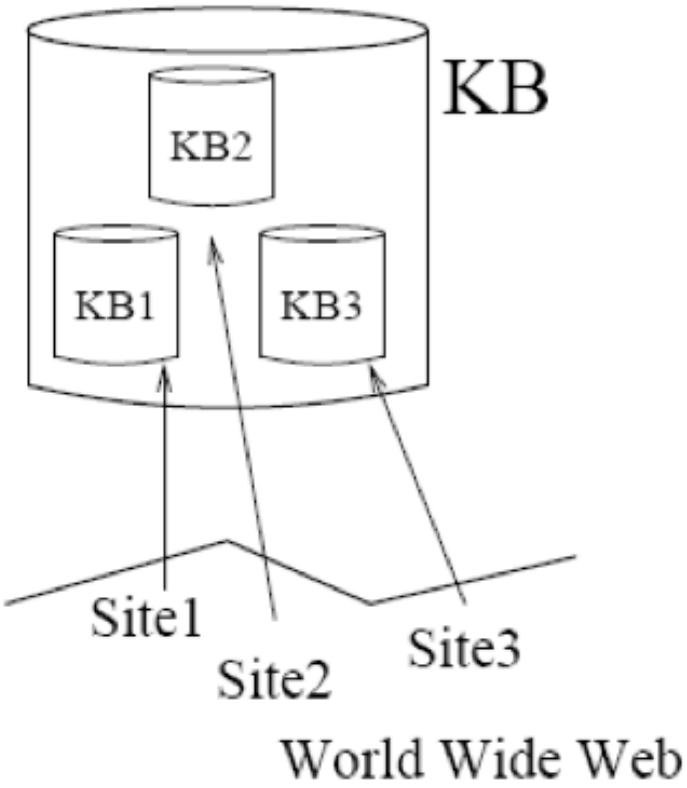
- Why similarity joins are important
- Useful similarity metrics for sets and strings
- Fast methods for K-NN and similarity joins
  - Blocking
  - Indexing
  - Short-cut algorithms
  - Parallel implementation

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….



Humongous	Humongous Entertainment
Headbone	Headbone Interactive
The Lion King: Storybook	Lion King Animated StoryBook
Disney's Activity Center, The Lion King	The Lion King Activity Center

Microsoft	Microsoft Kids
	Microsoft/Scholastic
Kestrel	American Kestrel
	Eurasian Kestrel
Canada Goose	Goose, Aleutian Canada
Mallard	Mallard, Mariana

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.

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

Softjoin Example - I

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

top500:	*hiTech:*
Abbott Laboratories	ACC CORP
Able Telcom Holding Corp.	ADC TELECOMMUNICATION INC
Access Health, Inc.	ADELPHIA COMMUNICATIONS CORP
Acclaim Entertainment, Inc.	ADT LTD
Ace Hardware Corporation	ADTRAN INC
ACS Communications, Inc.	AIRTOUCH COMMUNICATIONS
ACT Manufacturing, Inc.	AMATI COMMUNICATIONS CORP
Active Voice Corporation	AMERITECH CORP
Adams Media Corporation	APERTUS TECHNOLOGIES INC
Adolph Coors Company	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

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

Precision at K: G_K/K

Recall at K: G_K/G

G: # good pairings

G_K: # good pairings in first K

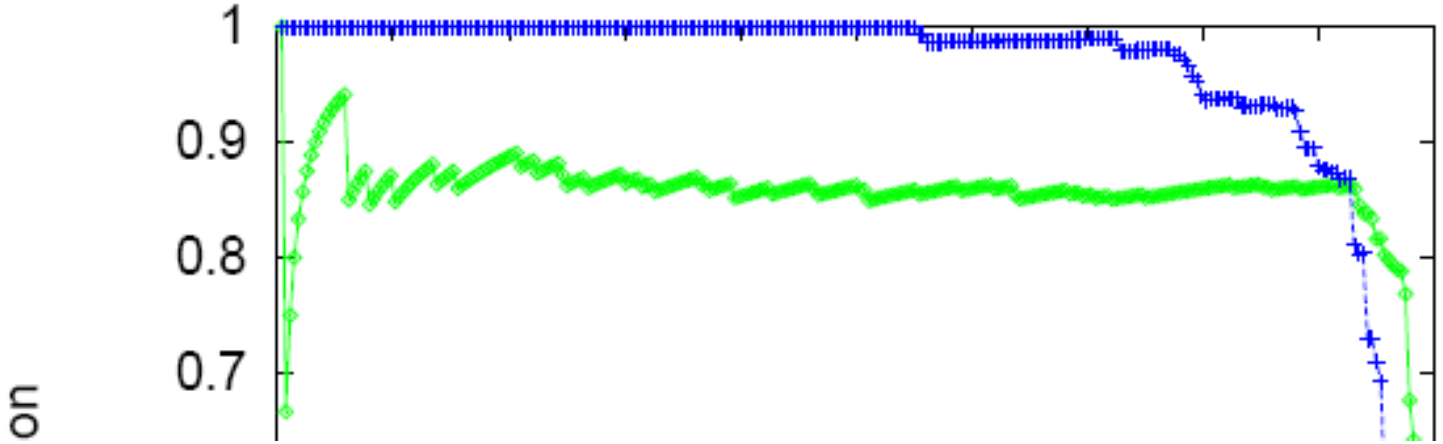


Table VI. Pairs of Names from the Hoovers and Iontech Relations

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

Table V. Average Precision for Similarity Joins

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

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

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

Change photo

William W. Cohen

Edit Follow

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

Title	+ Add	More	1–20	Cited by	Year
Fast Effective Rule Induction WW Cohen Proceedings of the Twelfth International Conference on Machine Learning ...				3367	1995
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				1488	2003
Recommendation as classification: Using social and content-based information in recommendation C Basu, H Hirsh, W Cohen AAAI/IAAI, 714-720				1064	1998

SOFT JOINS WITH TFIDF: HOW?

Rocchio's algorithm

Many variants of these formulae

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

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

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

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

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

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

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

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

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

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

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

TFIDF similarity

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

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

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

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

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

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

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

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 - I

```
# 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(

# build unweighted document vectors
udocvec = Join( Jin(data,by=lambda(rel,docid,term):(rel,term)), Jin(docFreq,by=lambda
    | 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)))

# compute the L2 norm for each document vector - actually this is the square of the L2 norm
sumSquareWeights = ReduceTo(float, lambda accum,(rel,doc,term,weight): accum+weight*weight)
norm = Group( udocvec, by=lambda(rel,doc,term,weight):(rel,doc), reducingTo=sumSquareWeights) \
    | ReplaceEach( by=lambda((rel,doc),z):(rel,doc,z))

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

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

Statistics for computing TFIDF with IDFs local to each relation

Parallel Inverted Index Softjoin - 2

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

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

What's the algorithm?

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

An alternative TFIDF pipeline

```
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])))
```

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

Inverted Index Softjoin – 2/3

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

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

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

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

Inverted Index Softjoin – 3/3

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

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


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

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

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

Results…..

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

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

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

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

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

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

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

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

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

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

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

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

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

Making the algorithm smarter….

Inverted Index Softjoin - 2

```
# naive algorithm 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))
```

- 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

Adding heuristics to the soft join - I

```
# 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

```
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