

# Other Map-Reduce (ish) Frameworks

William Cohen

# Outline

- More concise languages for map-reduce pipelines
- Abstractions built on top of map-reduce
  - General comments
  - Specific systems
    - Cascading, Pipes
    - PIG, Hive
    - Spark, Flink

# **Y:Y=Hadoop+X or Hadoop~=Y**

- What else are people using?
  - instead of Hadoop
    - Not really covered this lecture
  - on top of Hadoop

# Issues with Hadoop

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

# **STREAMING AND MRJOB: MORE CONCISE MAP-REDUCE PIPELINES**

# Hadoop streaming

- start with stream & sort pipeline

```
cat data | mapper.py | sort -k1,1 | reducer.py
```

- run with hadoop streaming instead

```
bin/hadoop jar contrib/streaming/hadoop-*streaming*.jar  
-file mapper.py -file reducer.py  
-mapper mapper.py  
-reducer reducer.py  
-input /hdfs/path/to/inputDir  
-output /hdfs/path/to/outputDir  
-mapred.map.tasks=20  
-mapred.reduce.tasks=20
```

# mrjob word count

- Python level over map-reduce – very concise
- Can run locally in Python
- Allows a single job or a linear chain of steps

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")

class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

    def combiner(self, word, counts):
        yield word, sum(counts)

    def reducer(self, word, counts):
        yield word, sum(counts)

if __name__ == '__main__':
    MRWordFreqCount.run()
```

# mrjob most freq word

```
class MRMostUsedWord(MRJob):

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # optimization: sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

    def steps(self):
        return [
            self.mr(mapper=self.mapper_get_words,
                    combiner=self.combiner_count_words,
                    reducer=self.reducer_count_words),
            self.mr(reducer=self.reducer_find_max_word)
        ]

if __name__ == '__main__':
    MRMostUsedWord.run()
```



# MAP-REDUCE ABSTRACTIONS

# Abstractions On Top Of Hadoop

- MRJob and other tools to make Hadoop pipelines more concise (Dumbo, ...) still keep the same basic language of map-reduce *jobs*
- How else can we express these sorts of computations? Are there some common special cases of map-reduce steps we can parameterize and reuse?

# Abstractions On Top Of Hadoop

- Some obvious streaming processes:
  - for each row in a table
    - Transform it and output the result
  - Decide if you want to keep it with some boolean test, and copy out only the ones that pass the test

**Example:** stem words in a stream of word-count pairs:

("aardvarks",1)  $\rightarrow$  ("aardvark",1)

**Proposed syntax:**  $f(row) \rightarrow row'$

*table2 = MAP table1 TO  $\lambda row: f(row)$*

**Example:** apply stop words

("aardvark",1)  $\rightarrow$  ("aardvark",1)  
("the",1)  $\rightarrow$  *deleted*

**Proposed syntax:**  $f(row) \rightarrow \{true, false\}$

*table2 = FILTER table1 BY  $\lambda row: f(row)$*

# Abstractions On Top Of Hadoop

- A non-obvious? streaming processes:
  - for each row in a table
    - Transform it to a list of items
    - Splice all the lists together to get the output table (**flatten**)

Proposed syntax:

$f(\text{row}) \rightarrow \text{list of rows}$

*table2 = FLATMAP table1 TO  $\lambda \text{ row: } f(\text{row})$*

**Example:** tokenizing a line

"I found an aardvark"  $\rightarrow$  ["i", "found", "an", "aardvark"]

"We love zymurgy"  $\rightarrow$  ["we", "love", "zymurgy"]

..but final table is one word per row

"i"

"found"

"an"

"aardvark"

"we"

"love"

...

# Abstractions On Top Of Hadoop

- Another example from the Naïve Bayes test program...

# NB Test Step (Can we do better?)

How:

- Stream and sort:
  - for each  $C[X=w \wedge Y=y]=n$ 
    - print " $w \ C[Y=y]=n$ "
  - sort and build a *list* of values associated with each key  $w$

*Like an inverted index*

Event counts

$X=w_1 \wedge Y=sports$	5245
$X=w_1 \wedge Y=worldNews$	1054
$X=..$	2120
$X=w_2 \wedge Y=...$	37
$X=...$	3
...	...



w	Counts associated with W
aardvark	$C[w \wedge Y=sports]=2$
agent	$C[w \wedge Y=sports]=1027, C[w \wedge Y=worldNews]=564$
...	...
zynga	$C[w \wedge Y=sports]=21, C[w \wedge Y=worldNews]=4464$

# NB Test Step 1 (Can we do better?)

## The general case:

We're taking rows from a table

- In a particular format (*event,count*)

Applying a function to get a new value

- The *word* for the event

And *grouping* the rows of the table by this new value

## → Grouping operation

*Special case of a map-reduce*

## Event counts

$X=w_1 \wedge Y=sports$	5245
$X=w_1 \wedge Y=worldNews$	1054
$X=..$	2120
$X=w_2 \wedge Y=...$	37
$X=...$	3
...	...



**Proposed syntax:**  $f(row) \rightarrow field$

GROUP *table* BY  $\lambda row: f(row)$

Output: *key, listOfRowsWithkey*

Could define  $f$  via: a function, a field of a defined *record* structure, ...

w	Counts associated with W
aardvark	$C[w \wedge Y=sports]=2$
agent	$C[w \wedge Y=sports]=1027, C[w \wedge Y=worldNews]=564$
...	...
zynga	$C[w \wedge Y=sports]=21, C[w \wedge Y=worldNews]=4464$

# NB Test Step 1 (Can we do better?)

## The general case:

We're taking rows from a table

- In a particular format (*event,count*)

Applying a function to get a new value

- The *word* for the event

And *grouping* the rows of the table by this new value

## → Grouping operation

*Special case of a map-reduce*

Aside: you guys know how to implement this, right?

1. Output pairs  $(f(\text{row}), \text{row})$  with a map/streaming process
2. Sort pairs by key – which is  $f(\text{row})$
3. Reduce and aggregate by *appending together* all the values associated with the same key

**Proposed syntax:**  $f(\text{row}) \rightarrow \text{field}$

GROUP *table* BY  $\lambda \text{row}: f(\text{row})$

Could define  $f$  via: a function, a field of a defined *record* structure, ...



# Abstractions On Top Of Hadoop

- And another example from the Naïve Bayes test program...

# Request-and-answer

Test data

$id_1$	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	....	$w_{1,k1}$
$id_2$	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	....	
$id_3$	$w_{3,1}$	$w_{3,2}$	....		
$id_4$	$w_{4,1}$	$w_{4,2}$	...		
$id_5$	$w_{5,1}$	$w_{5,2}$	....		
..					

Record of all event counts for each word

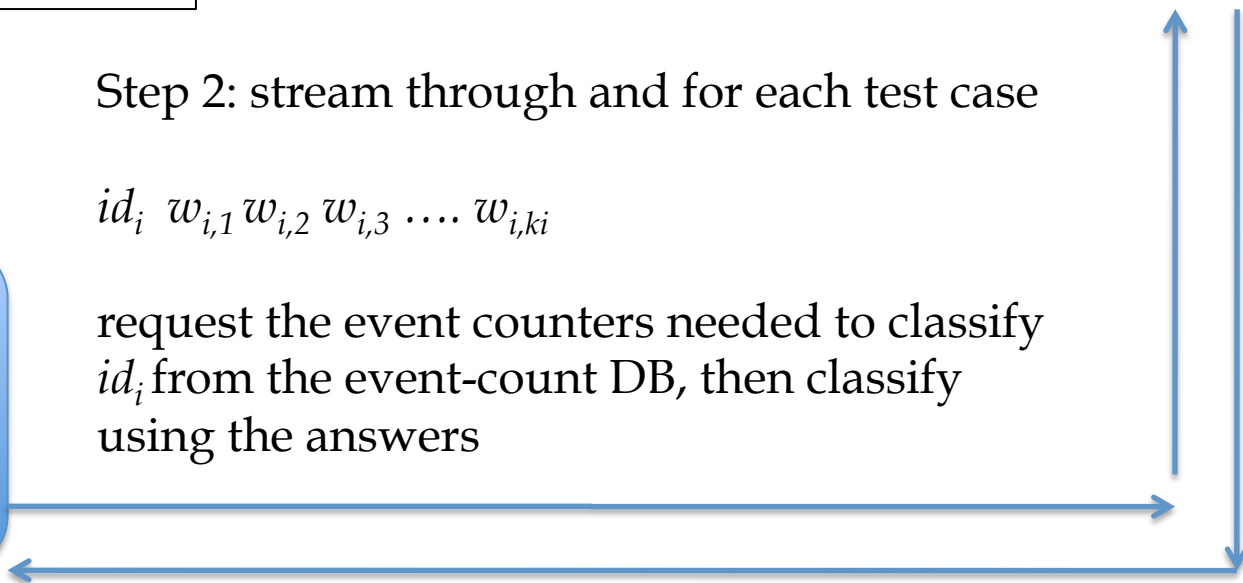
w	Counts associated with W
aardvark	$C[w^Y=sports]=2$
agent	$C[w^Y=sports]=1027, C[w^Y=worldN]$
...	...
zynga	$C[w^Y=sports]=21, C[w^Y=worldN]$



Step 2: stream through and for each test case

$id_i \ w_{i,1} \ w_{i,2} \ w_{i,3} \ \dots \ w_{i,ki}$

request the event counters needed to classify  $id_i$  from the event-count DB, then classify using the answers



# Request-and-answer

- Break down into stages
  - Generate the data being requested (indexed by key, here a word)
    - Eg with group ... by
  - Generate the requests as (key, requestor) pairs
    - Eg with flatmap ... to
  - **Join** these two tables by key
    - Join defined as (1) cross-product and (2) filter out pairs with different values for keys
    - This replaces the step of concatenating two different tables of key-value pairs, and reducing them together
  - Postprocess the joined result

w	Request
found	~ctr to id1
aardvark	~ctr to id1
...	
zynga	~ctr to id1
...	~ctr to id2

w	Counters
aardvark	$C[w^Y = \text{sports}] = 2$
agent	$C[w^Y = \text{sports}] = 1027, C[w^Y = \text{worldNews}] = \dots$
...	...
zynga	$C[w^Y = \text{sports}] = 21, C[w^Y = \text{worldNews}] = \dots$

w	Counters	Requests
aardvark	$C[w^Y = \text{sports}] = 2$	~ctr to id1
agent	$C[w^Y = \text{sports}] = \dots$	~ctr to id345
agent	$C[w^Y = \text{sports}] = \dots$	~ctr to id9854
agent	$C[w^Y = \text{sports}] = \dots$	~ctr to id345
...	$C[w^Y = \text{sports}] = \dots$	~ctr to id34742
zynga	$C[\dots]$	~ctr to id1
zynga	$C[\dots]$	...

w	Request
found	id1
aardvark	id1
...	
zynga	id1
...	id2

w	Counters
aardvark	$C[w^Y = \text{sports}] = 2$
agent	$C[w^Y = \text{sports}] = 1027, C[w^Y = \text{worldNews}] = \dots$
...	...
zynga	$C[w^Y = \text{sports}] = 21, C[w^Y = \text{worldNews}] = \dots$

w	Counters	Requests
aardvark	$C[w^Y = \text{sports}] = 2$	id1
agent	$C[w^Y = \text{sports}] = \dots$	id345
agent	$C[w^Y = \text{sports}] = \dots$	id9854
agent	$C[w^Y = \text{sports}] = \dots$	id345
...	$C[w^Y = \text{sports}] = \dots$	id34742
zynga	$C[\dots]$	id1

### Proposed syntax:

$table3 = \text{JOIN } table1 \text{ BY } \lambda row: f(row), table2 \text{ BY } \lambda row: g(row)$

Could define  $f$  and  $g$  via: a function, a field of a defined *record* structure, ...

# **MAP-REDUCE ABSTRACTIONS: CASCADING, PIPES, SCALDING**

# Y:Y=Hadoop+X

- Cascading
  - Java library for map-reduce workflows
  - Also some library operations for common mappers/reducers

# Cascading WordCount Example

```
Scheme sourceScheme = new TextLine( new Fields( "line" ) );  
Tap source = new Hfs( sourceScheme, inputPath );
```

Input format

Bind to HFS path

```
Scheme sinkScheme = new TextLine( new Fields( "word", "count" ) );  
Tap sink = new Hfs( sinkScheme, outputPath, SinkMode.REPLACE );
```

Output format: pairs

Bind to HFS path

```
Pipe assembly = new Pipe( "wordcount" );
```

A pipeline of map-reduce jobs

```
String regex = "(?>!\pL)(?=\pL)[^ ]*(?<=\pL)(?! \pL)";  
Function function = new RegexGenerator( new Fields( "word" ), regex );  
assembly = new Each( assembly, new Fields( "line" ), function );
```

Replace line with bag of words

Append a step: apply function to the “line” field

```
assembly = new GroupBy( assembly, new Fields( "word" ) );
```

Append step: group a (flattened) stream of “tuples”

```
Aggregator count = new Count( new Fields( "count" ) );  
assembly = new Every( assembly, count );
```

Append step: aggregate grouped values

```
Properties properties = new Properties();  
FlowConnector.setApplicationJarClass( properties, Main.class );
```

```
FlowConnector flowConnector = new FlowConnector( properties );  
Flow flow = flowConnector.connect( "word-count", source, sink, assembly );
```

Run the  
pipeline

```
flow.complete();
```



# Cascading WordCount Example

```
Scheme sourceScheme = new TextLine( new Fields( "line" ) );
```

Tap Many of the Hadoop abstraction levels have a similar flavor:

- Define a pipeline of tasks declaratively
- Optimize it automatically
- Run the final result

```
Sch
```

```
Tap
```

```
Pip
```

The key question: does the system *successfully* hide the details from you?

```
String regex = "(?>!\pL)(?=\pL)[^ ]*(?<=\pL)(?! \pL)";
```

```
Function function = new RegexGenerator( new Fields( "word" ), regex );
```

```
assembly = new Each( assembly, new Fields( "line" ), function );
```

```
assembly = new GroupBy( assembly, new Fields( "word" ) );
```

```
Aggregator count = new Count( new Fields( "count" ) );
```

```
assembly = new Every( assembly, count );
```

```
Properties properties = new Properties();
```

```
Flo
```

We *could* be saved by careful optimization: we know we don't need the GroupBy intermediate result when we run the assembly....

```
Flo
```

```
Flow flow = flowConnector.connect( "word-count", source, sink, assembly );
```

```
flow.complete();
```

Is this inefficient? We *explicitly* form a group for each word, and then count the elements...?

# Cascading WordCount Example

Many of the Hadoop abstraction levels have a similar flavor:

- Define a pipeline of tasks declaratively
- Optimize it automatically
- Run the final result

The key question: does the system *successfully* hide the details from you?

Another pipeline:

```
words = FLATMAP docs BY  $\lambda d$ : tokenize( d)
contentWords = FILTER words BY  $\lambda w$ : !contains(stopwords,w)
stems = MAP contentWords BY  $\lambda w$ : stem(w)
stemGroups = GROUP stems BY  $\lambda s$ : s
stemCounts = MAP stemGroups BY
     $\lambda \text{stem}, \text{listOfStems}$ : (stem, listOfStems.length())
```



Optimize to  
one reduce

...

How many passes do we need to make over the data?

# Y:Y=Hadoop+X

- Cascading
  - Java library for map-reduce workflows
    - expressed as “Pipe”s, to which you add Each, Every, GroupBy, ...
  - Also some library operations for common mappers/reducers
    - e.g. RegexGenerator
  - Turing-complete since it’s an API for Java
- Pipes
  - C++ library for map-reduce workflows on Hadoop
- Scalding
  - More concise Scala library based on Cascading

# **MORE DECLARATIVE LANGUAGES**

# Hive and PIG: word count

- Declarative ..... Fairly stable

```
FROM
(MAP docs.contents USING 'tokenizer_script' AS word, cnt
FROM docs
CLUSTER BY word) map_output

REDUCE map_output.word, map_output.cnt USING 'count_script' AS word, cnt;
```

```
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';
```

PIG program is a bunch of **assignments** where every LHS is a **relation**.  
No loops, conditionals, etc allowed.

# More on Pig

- Pig Latin
  - atomic types + compound types like tuple, bag, map
  - execute locally/interactively or on hadoop
- can embed Pig in Java (and Python and ...)
- can call out to Java from Pig
- Similar (ish) system from Microsoft: DryadLinq

```
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';
```

Tokenize – built-in function

Flatten – special keyword, which applies to the next step in the process – ie foreach is transformed from a MAP to a FLATMAP

PIG parses and **optimizes** a sequence of commands before it executes them  
It's smart enough to turn GROUP ... FOREACH... SUM ... into a map-reduce

- LOAD '*hdfs-path*' AS (*schema*)
  - *schemas can include int, double, bag, map, tuple, ...*
- FOREACH *alias* GENERATE ... AS ..., ...
  - *transforms each row of a relation*
- DESCRIBE *alias*/ILLUSTRATE *alias* -- *debugging*
- GROUP *alias* BY ...
- FOREACH *alias* GENERATE *group*, SUM(...)
  - *GROUP/GENERATE ... aggregate op together act like a map-reduce*
- JOIN *r* BY *field*, *s* BY *field*, ...
  - *inner join to produce rows: r::*f1*, r::*f2*, ... s::*f1*, s::*f2*, ...*
- CROSS *r*, *s*, ...
  - *use with care unless all but one of the relations are singleton*
- User defined functions as operators
  - *also for loading, aggregates, ...*



# **ANOTHER EXAMPLE: COMPUTING TFIDF IN PIG LATIN**

```

1 DEFINE tf_idf(in_relation, id_field, text_field) RETURNS out_relation {
2   token_records = foreach $in_relation generate $id_field, FLATTEN(TOKENIZE($text_field)) as tokens;
3
4   /* Calculate the term count per document */
5   doc_word_totals = foreach (group token_records by ($id_field, tokens)) generate
6     FLATTEN(group) as ($id_field, token),
7     COUNT_STAR(token_records) as doc_total;

```

(docid,token) → (docid,token,tf(token in doc))

```

8
9   /* Calculate the document size */
10  pre_term_counts = foreach (group doc_word_totals by $id_field) generate
11    group AS $id_field,
12    FLATTEN(doc_word_totals.(token, doc_total)) as (token, doc_total),
13    SUM(doc_word_totals.doc_total) as doc_size;

```

(docid,token,tf) → (docid,token,tf,length(doc))

```

14
15  /* Calculate the TF */
16  term_freqs = foreach pre_term_counts generate $id_field as $id_field,
17    token as token,
18    ((double)doc_total / (double)doc_size) AS term_freq;

```

(docid,token,tf,n) → (... ,tf/n)

```

19
20  /* Get count of documents using each token, for idf */
21  token_usages = foreach (group term_freqs by token) generate
22    FLATTEN(term_freqs) as ($id_field, token, term_freq),
23    COUNT_STAR(term_freqs) as num_docs_with_token;

```

(docid,token,tf,n,tf/n) → (... ,df)

```

24
25  /* Get document count */
26  just_ids = foreach $in_relation generate $id_field;
27  ndocs = foreach (group just_ids all) generate COUNT_STAR(just_ids) as total_docs;

```

```

28
29  /* Note the use of Pig Scalars to calculate idf */
30  $out_relation = foreach token_usages {
31    idf      = LOG((double)ndocs.total_docs/(double)num_docs_with_token);
32    tf_idf = (double)term_freq * idf;
33    generate $id_field as $id_field,
34      token as score,
35      (chararray)tf_idf as value:chararray;
36  };
37};

```

ndocs.total\_docs

relation-to-scalar casting

(docid,token,tf,n,tf/n) → (docid,token,tf/n \* id)

# Other PIG features

- ...
- Macros, nested queries,
- FLATTEN “operation”
  - transforms a bag or a tuple into its individual elements
  - this transform affects the *next level* of the aggregate
- STREAM and DEFINE ... SHIP

```
DEFINE myfunc `python myfun.py` SHIP('myfun.py')  
...  
r = STREAM s THROUGH myfunc AS (...);
```

```
DEFINE tokenize_docs `ruby tokenize_documents.rb --id_field=0 --text_field=1 --map` SHIP('tokenize_documents.rb');
```

```
raw_documents = LOAD '$DOCS' AS (doc_id:chararray, text:chararray);
tokenized     = STREAM raw_documents THROUGH tokenize_docs AS (doc_id:chararray, token:chararray);
```

# TF-IDF in PIG - another version

```
DEFINE tokenize_docs `ruby tokenize_documents.rb --id_field=0 --text_field=1 --map` SHIP('tokenize_documents.rb');
```

```
raw_documents = LOAD '$DOCS' AS (doc_id:chararray, text:chararray);
tokenized     = STREAM raw_documents THROUGH tokenize_docs AS (doc_id:chararray, token:chararray);
```

```
doc_tokens      = GROUP tokenized BY (doc_id, token);
doc_token_counts = FOREACH doc_tokens GENERATE FLATTEN(group) AS (doc_id, token), COUNT(tokenized) AS num_doc_tok_usages;
```

```
doc_usage_bag    = GROUP doc_token_counts BY doc_id;
doc_usage_bag_fg = FOREACH doc_usage_bag GENERATE
    group                                AS doc_id,
    FLATTEN(doc_token_counts.(token, num_doc_tok_usages)) AS (token, num_doc_tok_usages),
    SUM(doc_token_counts.num_doc_tok_usages)              AS doc_size
    ;
```

```
term_freqs = FOREACH doc_usage_bag_fg GENERATE
    doc_id AS doc_id,
    token   AS token,
    ((double)num_doc_tok_usages / (double)doc_size) AS term_freq;
    ;
```

```
term_usage_bag = GROUP term_freqs BY token;
token_usages   = FOREACH term_usage_bag GENERATE
    FLATTEN(term_freqs) AS (doc_id, token, term_freq),
    COUNT(term_freqs)   AS num_docs_with_token
    ;
```

```
tfidf_all = FOREACH token_usages {
    idf      = LOG((double)$NDOCS / (double)num_docs_with_token);
    tf_idf   = (double)term_freq * idf;
    GENERATE
        doc_id AS doc_id,
        token   AS token,
        tf_idf  AS tf_idf
    ;
};
```

```
STORE tfidf_all INTO '$OUT';
```

# Issues with Hadoop

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

First: an iterative algorithm in Pig Latin

How to use loops,  
conditionals, etc?

Embed PIG in a  
real programming  
language.

Julien Le Dem -  
Yahoo

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

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

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

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

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

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

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

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



```
#!/usr/bin/python
```

```
from org.apache.pig.scripting import *
```

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

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

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

Iterate 10 times

Pass parameters as a dictionary

Just run P, that was declared above

The output becomes the new input

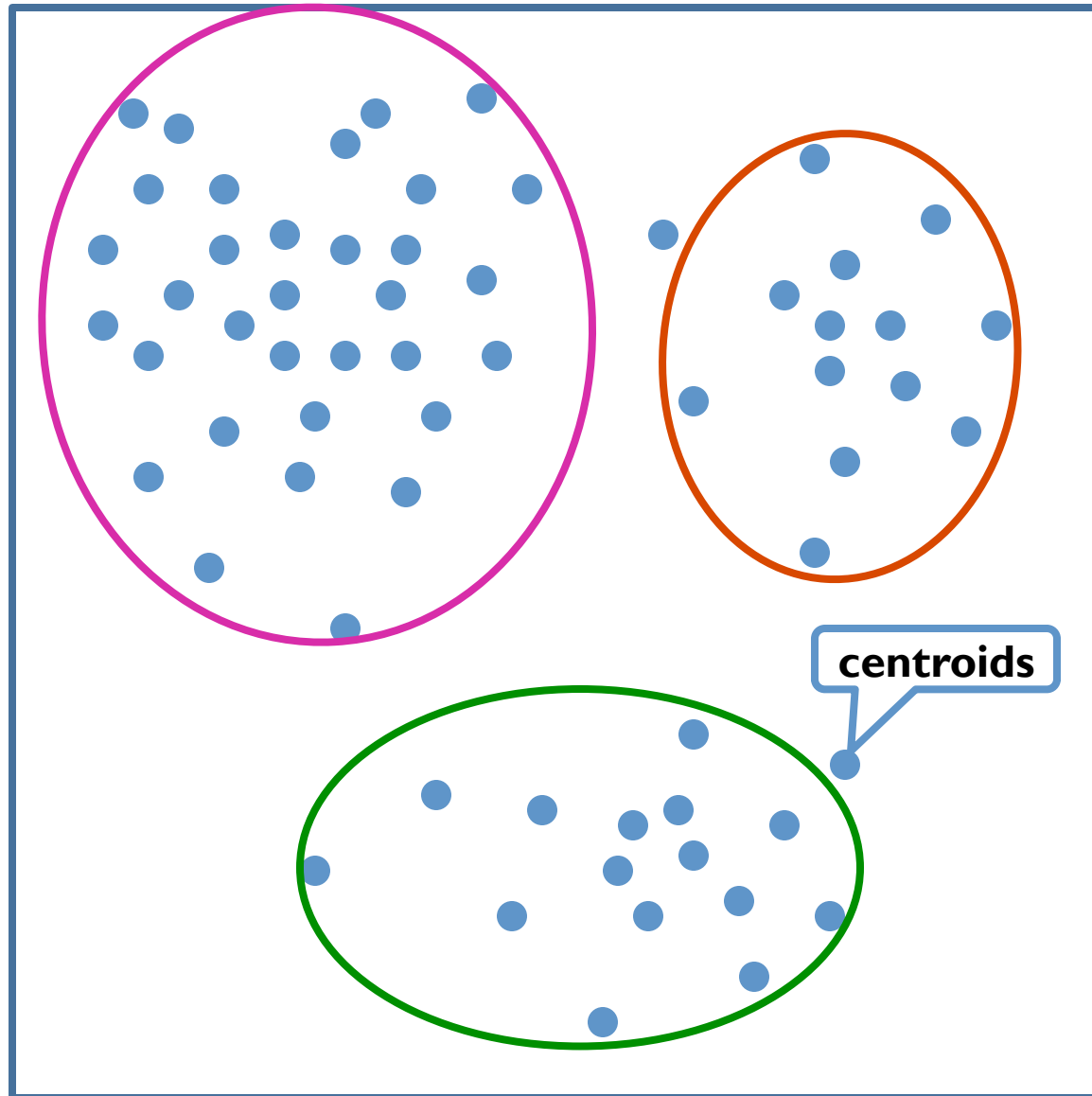
# **An example from Ron Bekkerman**



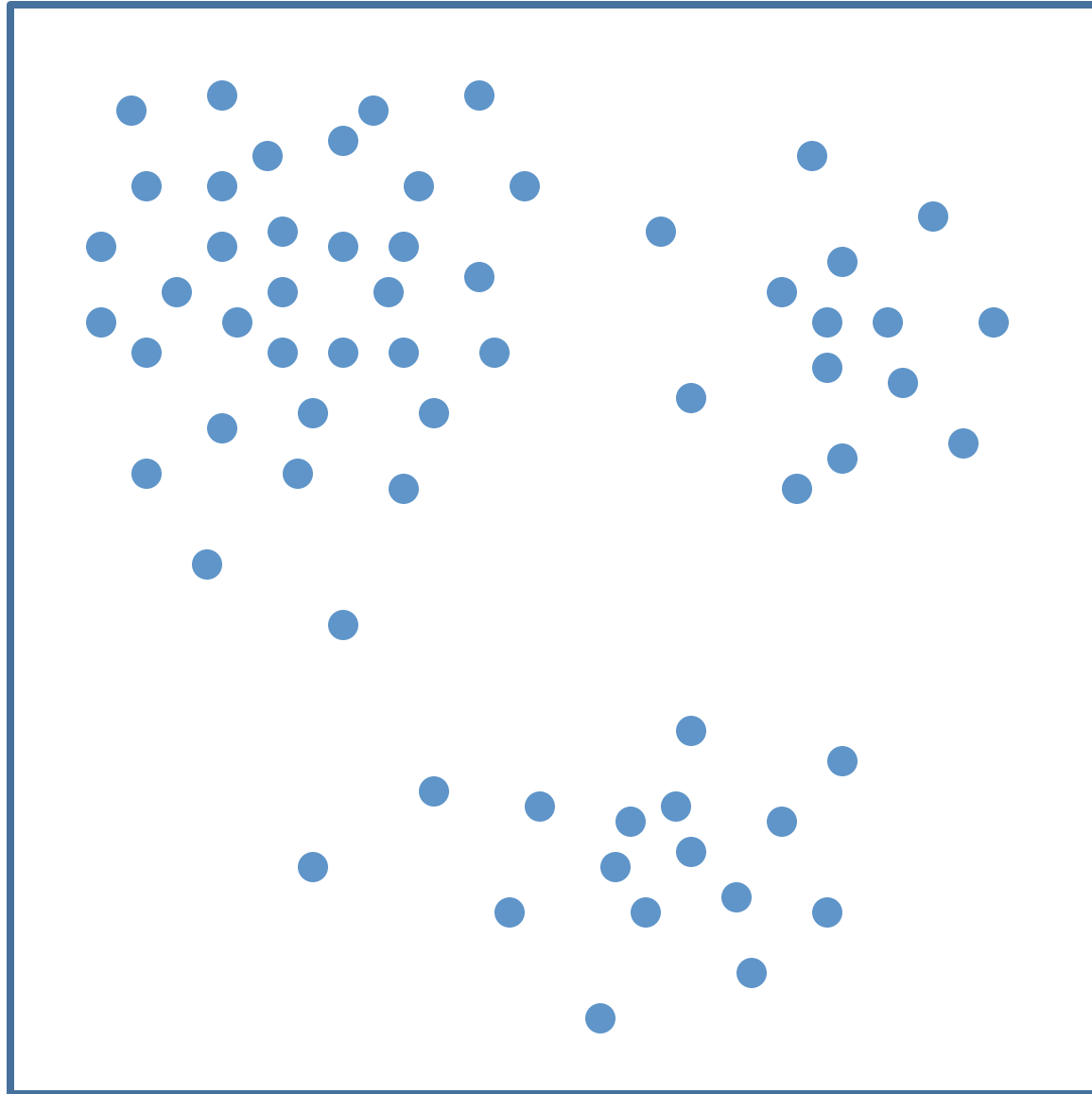
# Example: *k*-means clustering

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

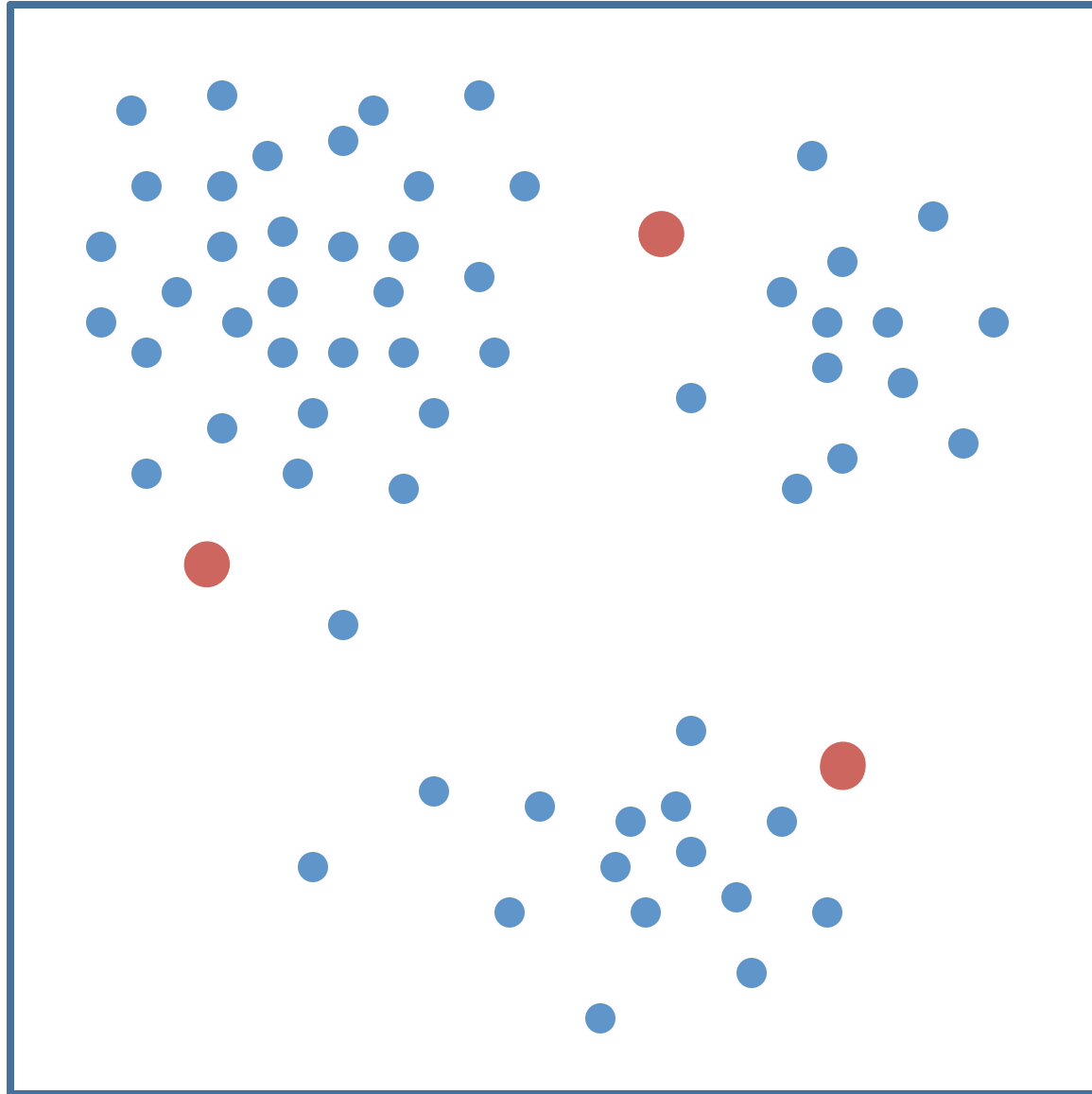
# *k*-means Clustering



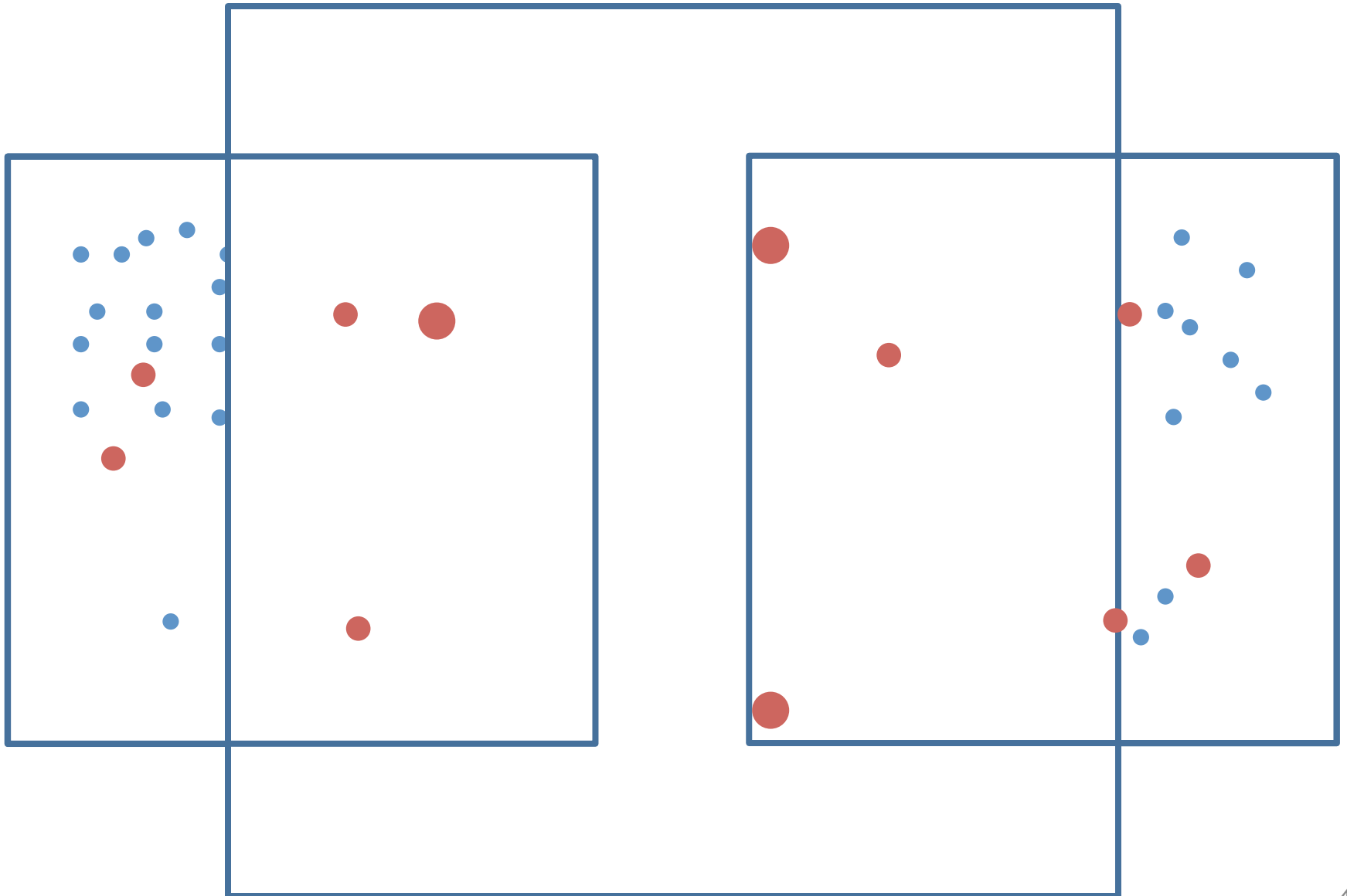
# Parallelizing *k*-means



# Parallelizing *k*-means

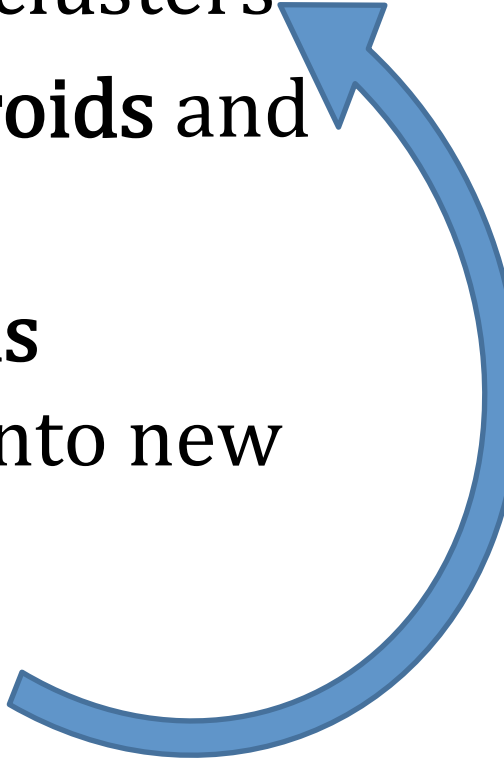


# Parallelizing $k$ -means



# **k-means on MapReduce**

*Panda et al, Chapter 2*

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

# *k*-means in Apache Pig: input data

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

Document	Word	Count
doc1	Carnegie	2
doc1	Mellon	2

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

# k-means in Apache Pig: E-step

$D\_C = \text{JOIN } C \text{ BY } w, D \text{ BY } w;$

$PROD = \text{FOREACH } D\_C \text{ GENERATE } d, c, i_d * i_c \text{ AS } i_d i_c;$

$PROD_g = \text{GROUP } PROD \text{ BY } (d, c);$

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

$SIM = \text{FOREACH } DOT\_LEN \text{ GENERATE } d, c, dXc / len_c;$

$SIM_g = \text{GROUP } SIM \text{ BY } d;$

$CLUSTERS = \text{FOREACH } SIM_g \text{ GENERATE TOP}(1, 2, SIM);$



# k-means in Apache Pig: E-step

$D\_C = \text{JOIN } C \text{ BY } w, D \text{ BY } w;$

$PROD = \text{FOREACH } D\_C \text{ GENERATE } d, c, i_d * i_c \text{ AS } i_d i_c;$

$PROD_g = \text{GROUP } PROD \text{ BY } (d, c);$

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

$SIM = \text{FOREACH } DOT\_LEN \text{ GENERATE } d, c, dXc / len_c;$

$SIM_g = \text{GROUP } SIM \text{ BY } d;$

$CLUSTERS = \text{FOREACH } SIM_g \text{ GENERATE TOP}(1, 2, SIM);$

# k-means in Apache Pig: E-step

$D\_C = \text{JOIN } C \text{ BY } w, D \text{ BY } w;$

$PROD = \text{FOREACH } D\_C \text{ GENERATE } d, c, i_d * i_c \text{ AS } i_d i_c;$

$PROD_g = \text{GROUP } PROD \text{ BY } (d, c);$

$DOT\_LEN\_SQR_g = \text{FOREACH } PROD_g \text{ GENERATE } \sum_{w \in d} i_d^w \cdot i_c^w \text{ AS } dXc;$   
 $c_d = \arg \max_c \frac{\sum_{w \in d} i_d^w \cdot i_c^w}{\sqrt{\sum_{w \in c} (i_c^w)^2}}$

$SIM = \text{FOREACH } DOT\_LEN\_SQR_g \text{ GENERATE } d, c, dXc / len_c;$

$SIM_g = \text{GROUP } SIM \text{ BY } d;$

$CLUSTERS = \text{FOREACH } SIM_g \text{ GENERATE TOP}(1, 2, SIM);$

# k-means in Apache Pig: E-step

$D\_C = \text{JOIN } C \text{ BY } w, D \text{ BY } w;$

$PROD = \text{FOREACH } D\_C \text{ GENERATE } d, c, i_d * i_c \text{ AS } i_d i_c;$

$PROD_g = \text{GROUP } PROD \text{ BY } (d, c);$

$DOT\_LEN\_g$   
 $SQR\_g$   
 $SQR\_g$   
 $LEN\_g$   
 $DOT\_g$

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

$dXc;$   
 $n_c;$

$SIM = \text{FOREACH } DOT\_LEN\_g \text{ GENERATE } d, c, dXc / len_c;$

$SIM_g = \text{GROUP } SIM \text{ BY } d;$

$CLUSTERS = \text{FOREACH } SIM_g \text{ GENERATE TOP}(1, 2, SIM);$

# k-means in Apache Pig: E-step

$D\_C = \text{JOIN } C \text{ BY } w, D \text{ BY } w;$

$PROD = \text{FOREACH } D\_C \text{ GENERATE } d, c, i_d * i_c \text{ AS } i_d i_c;$

$PROD_g = \text{GROUP } PROD \text{ BY } (d, c);$

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

$SIM = \text{FOREACH } DOT\_LEN \text{ GENERATE } d, c, dXc / len_c;$

$SIM_g = \text{GROUP } SIM \text{ BY } d;$

$CLUSTERS = \text{FOREACH } SIM_g \text{ GENERATE TOP}(1, 2, SIM);$

# k-means in Apache Pig: E-step

$D\_C = \text{JOIN } C \text{ BY } w, D \text{ BY } w;$

$PROD = \text{FOREACH } D\_C \text{ GENERATE } d, c, i_d * i_c \text{ AS } i_d i_c;$

$PROD_g = \text{GROUP } PROD \text{ BY } (d, c);$

$DOT\_PROD = \text{FOREACH } PROD_g \text{ GENERATE } d, c, \text{SUM}(i_d i_c) \text{ AS } dXc;$

$SQR = \text{FOREACH } C \text{ GENERATE } c, i_c * i_c \text{ AS } i_c^2;$

$SQR_g = \text{GROUP } SQR \text{ BY } c;$

$LEN\_C = \text{FOREACH } SQR_g \text{ GENERATE } c, \text{SQRT}(\text{SUM}(i_c^2)) \text{ AS } len_c;$

$DOT\_LEN = \text{JOIN } LEN\_C \text{ BY } c, DOT\_PROD \text{ BY } c;$

$SIM = \text{FOREACH } DOT\_LEN \text{ GENERATE } d, c, dXc / len_c;$

$SIM_g = \text{GROUP } SIM \text{ BY } d;$

$CLUSTERS = \text{FOREACH } SIM_g \text{ GENERATE TOP}(1, 2, SIM);$

# k-means in Apache Pig: M-step

$D\_C\_W$  = **JOIN** CLUSTERS BY  $d$ ,  $D$  BY  $d$ ;

$D\_C\_W_g$  = **GROUP**  $D\_C\_W$  BY  $(c, w)$ ;

$SUMS$  = **FOREACH**  $D\_C\_W_g$  **GENERATE**  $c, w, SUM(i_d)$  **AS**  $sum$ ;

$D\_C\_W_{gg}$  = **GROUP**  $D\_C\_W$  BY  $c$ ;

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

$SUMS\_SIZES$  = **JOIN**  $SIZES$  BY  $c$ ,  $SUMS$  BY  $c$ ;

$C$  = **FOREACH**  $SUMS\_SIZES$  **GENERATE**  $c, w, sum / size$  **AS**  $i_c$ ;

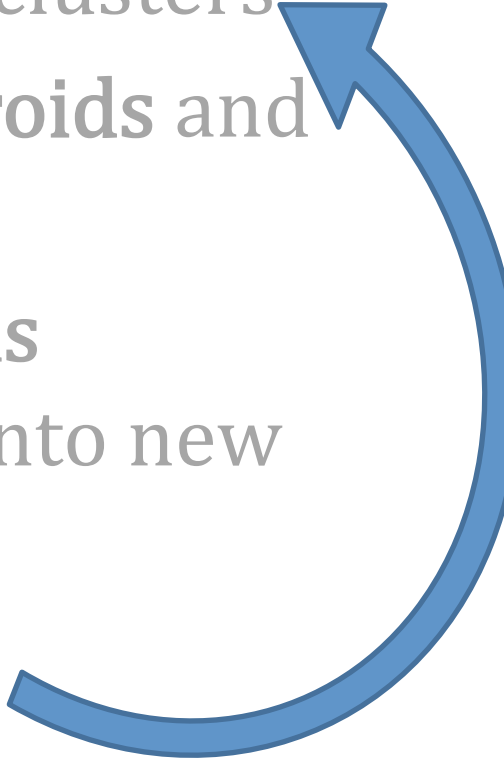
Finally - embed in Java (or Python or ....) to do the looping

# The problem with k-means in Hadoop

I/O costs

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

*Panda et al, Chapter 2*

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



# **SCHEMES DESIGNED FOR ITERATIVE HADOOP PROGRAMS: SPARK AND FLINK**

# Spark word count example

- Research project, based on Scala and Hadoop
- Now APIs in Java and Python as well

---

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                  .map(word => (word, 1))
                  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

---

- Familiar-looking API for abstract operations (map, flatMap, reduceByKey, ...)
- Most API calls are “lazy” – ie, *counts* is a data structure defining a pipeline, not a materialized table.
- Includes ability to store a sharded dataset in cluster memory as an RDD (resilient distributed database)

# Spark logistic regression example

---

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating plane: " + w)
```

---

Note that `w` gets shipped automatically to the cluster with every `map` call.

# Spark logistic regression example

- Allows caching data in memory

---

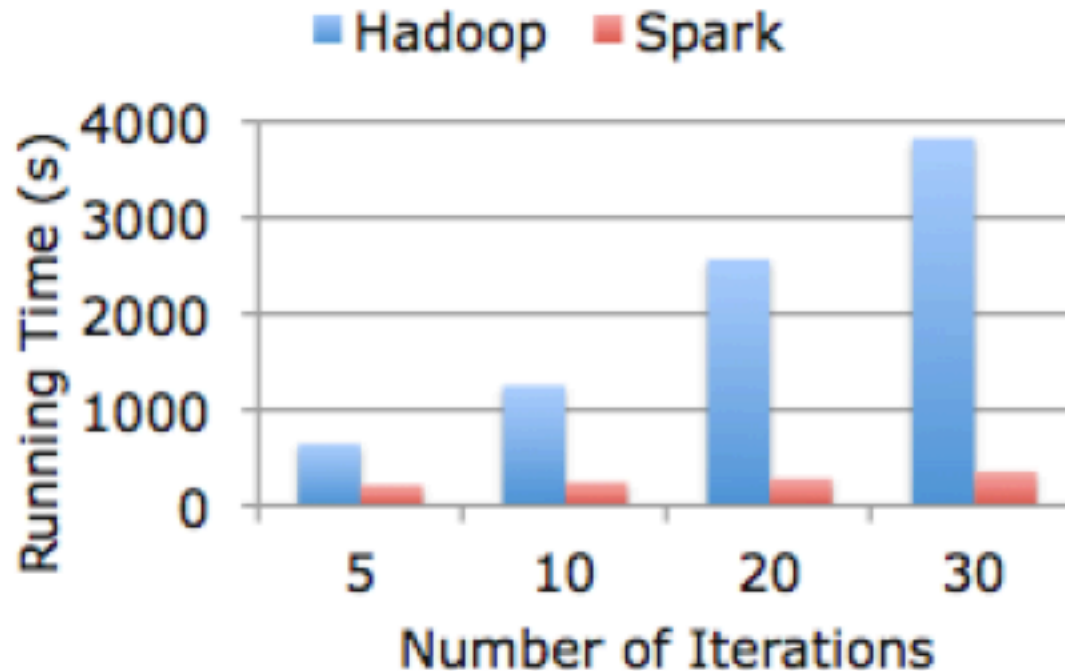
```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating plane: " + w)
```

---

Note that `w` gets shipped automatically to the cluster with every `map` call.

# Spark logistic regression example

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



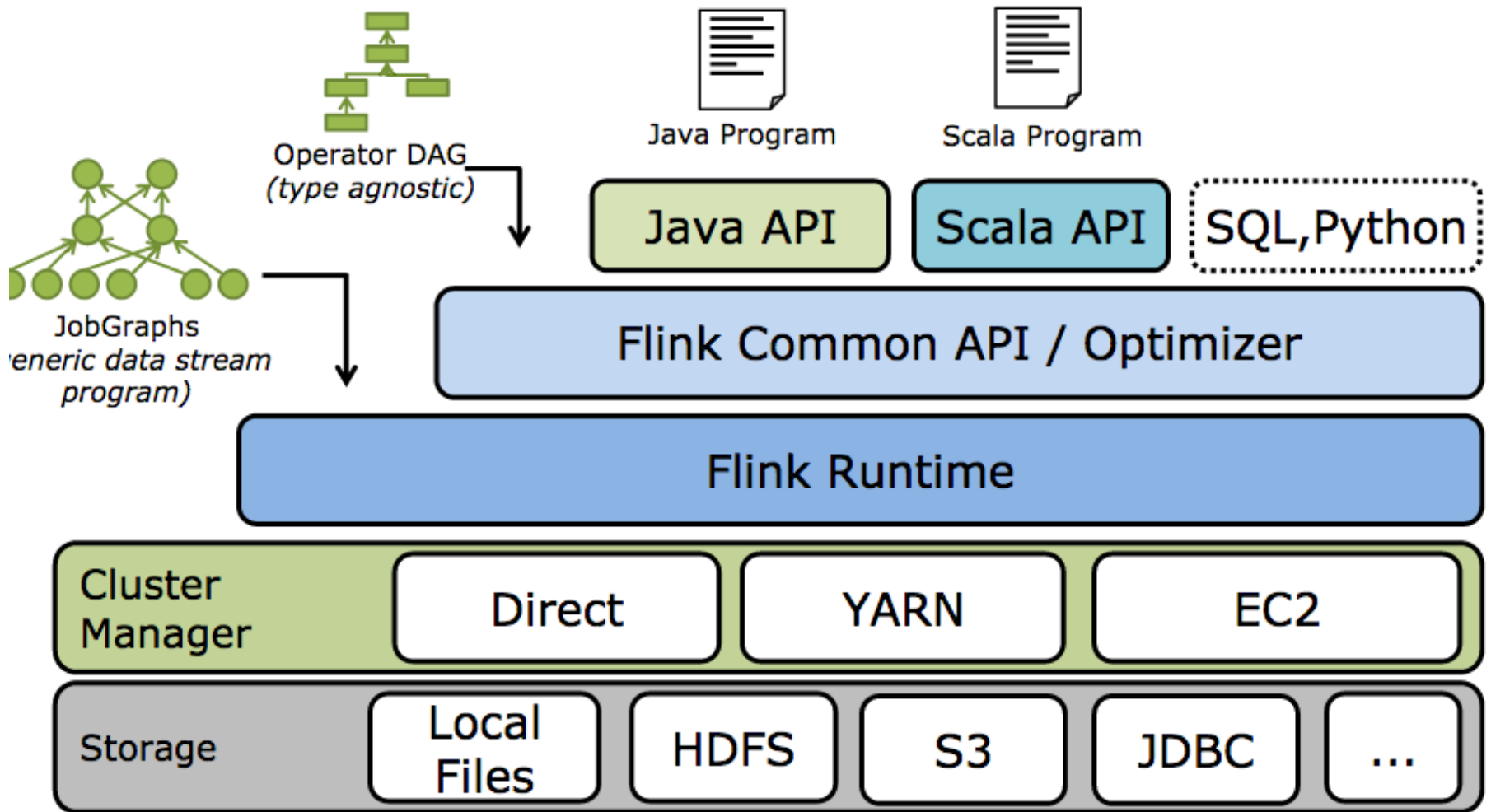
# FLINK

- Recent Apache Project – just moved to top-level at 0.8 – formerly Stratosphere

```
object WordCountJob {  
  def main(args: Array[String]) {  
  
    // set up the execution environment  
    val env = ExecutionEnvironment.getExecutionEnvironment  
  
    // get input data  
    val text = env.fromElements("To be, or not to be,--that is the question:--",  
                                "Whether 'tis nobler in the mind to suffer", "The slings and arrows of outrageous fortune",  
                                "Or to take arms against a sea of troubles,")  
  
    val counts = text.flatMap { _.toLowerCase.split("\\W+") }  
                      .map { (_, 1) }  
                      .groupBy(0)  
                      .sum(1)  
  
    // emit result  
    counts.print()  
  
    // execute program  
    env.execute("WordCount Example")  
  }  
}
```

```
public class WordCount {  
  
    public static void main(String[] args) throws Exception {  
  
        // set up the execution environment  
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();  
  
        // get input data  
        DataSet<String> text = env.fromElements(  
  
            ....  
  
        DataSet<Tuple2<String, Integer>> counts =  
            // split up the lines in pairs (2-tuples) containing: (word,1)  
            text.flatMap(new LineSplitter())  
            // group by the tuple field "0" and sum up tuple field "1"  
            .groupBy(0)  
            .aggregate(Aggregations.SUM, 1);  
  
        // emit result  
        counts.print();  
  
        // execute program  
        env.execute("WordCount Example");  
    }  
}
```

# FLINK





# FLINK

- Like Spark, in-memory or on disk
- Everything is a Java object
- Unlike Spark, contains operations for iteration
  - Allowing query optimization
- Very easy to use and install in local model
  - Very modular
  - Only needs Java

**MORE EXAMPLES IN PIG**

# Phrase Finding in PLG

# Phrase Finding I - loading the input

```

wcohen@shell12:~/pigtrial$ pig
2014-04-01 16:38:07,694 [main] INFO org.apache.pig.Main - Apache Pig version 0.11.1.1.3.0.0-107 (rexported) compiled
2014-04-01 16:38:07,695 [main] INFO org.apache.pig.Main - Logging error messages to: /h/wcohen/pigtrial/pig_13963846
2014-04-01 16:38:07,826 [main] INFO org.apache.pig.impl.util.Utils - Default bootup file /h/wcohen/.pigbootup not fo
2014-04-01 16:38:08,133 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to h
2014-04-01 16:38:08,379 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to m
grunt> SET default_parallel 10;
SET default_parallel 10;
grunt> fs -ls phrases/data/dkos-phraseFreq-5/
fs -ls phrases/data/dkos-phraseFreq-5/
Found 5 items
-rwxr-xr-x 3 wcohen supergroup 28857 2014-03-14 14:00 /user/wcohen/phrases/data/dkos-phraseFreq-5/part-00000
-rwxr-xr-x 3 wcohen supergroup 28210 2014-03-14 14:00 /user/wcohen/phrases/data/dkos-phraseFreq-5/part-00001
-rwxr-xr-x 3 wcohen supergroup 29731 2014-03-14 14:00 /user/wcohen/phrases/data/dkos-phraseFreq-5/part-00002
-rwxr-xr-x 3 wcohen supergroup 27422 2014-03-14 14:00 /user/wcohen/phrases/data/dkos-phraseFreq-5/part-00003
-rwxr-xr-x 3 wcohen supergroup 29198 2014-03-14 14:00 /user/wcohen/phrases/data/dkos-phraseFreq-5/part-00004
grunt> fs -tail phrases/data/dkos-phraseFreq-5/part-00003
fs -tail phrases/data/dkos-phraseFreq-5/part-00003
oluntary code 1.0
volvodrivingliberal sun 1.0
voredy thu 1.0
voter registrations 2.0
voter suppression 3.0
wackyguy thu 1.0
waitingtoderaill tue 1.0
walt starr 1.0
walter reed 1.0
wanna run 1.0
war plans 1.0
war question 1.0
war veterans 1.0
...
years taken 1.0
yes men 1.0
yesterday got 1.0
yesterday senator 1.0
yesterdays diary 1.0
york political 1.0
young people 1.0
zogby poll 1.0
zombiexx thu 1.0
years taken 1.0
yes men 1.0
yesterday got 1.0
yesterday senator 1.0
yesterdays diary 1.0
york political 1.0
young people 1.0
zogby poll 1.0
zombiexx thu 1.0

```

# PIG Features

- comments -- *like this /\* or like this \*/*
- ‘shell-like’ commands:
  - fs -ls ... -- *any hadoop fs ... command*
  - some shorter cuts: *ls, cp, ...*
  - sh ls -al -- *escape to shell*

```

grunt> fgPhrases1 = LOAD 'phrases/data/dkos-pharseFreq-5/' AS (xy,c:int);
fgPhrases1 = LOAD 'phrases/data/dkos-pharseFreq-5/' AS (xy,c:int);
grunt> fgPhrases = FOREACH fgPhrases1 GENERATE STRSPLIT(xy, ' ') AS xy:(x,y);
fgPhrases = FOREACH fgPhrases1 GENERATE STRSPLIT(xy, ' ') AS xy:(x,y), c AS c;
2014-04-01 16:42:44,881 [main] WARN org.apache.pig.PigServer - Encountered
2014-04-01 16:42:44,881 [main] WARN org.apache.pig.PigServer - Encountered
grunt> DESCRIBE fgPhrases;
DESCRIBE fgPhrases;
2014-04-01 16:43:06,631 [main] WARN org.apache.pig.PigServer - Encountered
2014-04-01 16:43:06,631 [main] WARN org.apache.pig.PigServer - Encountered
fgPhrases: {xy: (x: bytearray,y: bytearray),c: int}
grunt> ILLUSTRATE fgPhrases;

```

...

fgPhrases1	xy:bytearray	c:int
	patachon mon	1

fgPhrases	xy:tuple(x:bytearray,y:bytearray)	c:int
	(patachon, mon)	1

# PIG Features

- comments *-- like this /\* or like this \*/*
- 'shell-like' commands:
  - *fs -ls ... -- any hadoop fs ... command*
  - *some shorter cuts: ls, cp, ...*
  - *sh ls -al -- escape to shell*
- *LOAD 'hdfs-path' AS (schema)*
  - *schemas can include int, double, ...*
  - *schemas can include complex types: bag, map, tuple, ...*
- *FOREACH alias GENERATE ... AS ..., ...*
  - *transforms each row of a relation*
  - *operators include +, -, and, or, ...*
  - *can extend this set easily (more later)*
- *DESCRIBE alias -- shows the schema*
- *ILLUSTRATE alias -- derives a sample tuple*



# Phrase Finding I - word counts

```

grunt> bgPhrases1 = LOAD 'phrases/data/brown-phraseFreq-5/' AS (xy,c:int);
bgPhrases1 = LOAD 'phrases/data/brown-phraseFreq-5/' AS (xy,c:int);
2014-04-01 16:46:52,014 [main] WARN org.apache.pig.PigServer - Encountered Warning IMPLICIT_CAST_TO_CHARARRAY 1 time(s).
2014-04-01 16:46:52,014 [main] WARN org.apache.pig.PigServer - Encountered Warning USING_OVERLOADED_FUNCTION 1 time(s).
grunt> bgPhrases = FOREACH bgPhrases1 GENERATE STRSPLIT(xy,' ') AS xy:(x,y), c AS c;
bgPhrases = FOREACH bgPhrases1 GENERATE STRSPLIT(xy,' ') AS xy:(x,y), c AS c;
2014-04-01 16:46:54,750 [main] WARN org.apache.pig.PigServer - Encountered Warning IMPLICIT_CAST_TO_CHARARRAY 2 time(s).
2014-04-01 16:46:54,750 [main] WARN org.apache.pig.PigServer - Encountered Warning USING_OVERLOADED_FUNCTION 2 time(s).
grunt> fgWordFreq1 = GROUP fgPhrases BY xy.x;
fgWordFreq1 = GROUP fgPhrases BY xy.x;

```

-- compute word frequencies

```

fgWordFreq1 = GROUP fgPhrases BY xy.x;
fgWordFreq = FOREACH fgWordFreq1 GENERATE group as w,SUM(fgPhrases.c) as c;

```

fgPhrases1	xy:bytearray	c:int
	expressly gave	1
	expressly reasserted	1

fgPhrases	xy:tuple(x:bytearray,y:bytearray)	c:int
	(expressly, gave)	1
	(expressly, reasserted)	1

fgWordFreq1	group:bytearray	fgPhrases:bag{:tuple(xy:tuple(x:bytearray,y:bytearray),c:int)}
	expressly	{((expressly, gave), 1), ((expressly, reasserted), 1)}

fgWordFreq	w:bytearray	c:long
	expressly	2

# PIG Features

- LOAD '*hdfs-path*' AS (*schema*)
  - *schemas can include int, double, bag, map, tuple, ...*
- FOREACH *alias* GENERATE ... AS ..., ...
  - *transforms each row of a relation*
- DESCRIBE *alias*/ILLUSTRATE *alias* -- *debugging*
- GROUP *r* BY *x*
  - *like a shuffle-sort: produces relation with fields group and r, where r is a bag*

fgWordFreq1	group:bytearray	fgPhrases:bag{:tuple(xy:tuple(x:bytearray,y:bytearray),c:int)}
	expressly	{((expressly, gave), 1), ((expressly, reasserted), 1)}

PIG parses and **optimizes** a sequence of commands before it executes them  
 It's smart enough to turn GROUP ... FOREACH... SUM ... into a map-reduce

-- compute word frequencies

```
fgWordFreq1 = GROUP fgPhrases BY xy.x;
fgWordFreq = FOREACH fgWordFreq1 GENERATE group as w, SUM(fgPhrases.c) as c;
```

fgPhrases1	xy:bytearray	c:int
	expressly gave	1
	expressly reasserted	1

fgPhrases	xy:tuple(x:bytearray,y:bytearray)	c:int
	(expressly, gave)	1
	(expressly, reasserted)	1

fgWordFreq1	group:bytearray	fgPhrases:bag{:tuple(xy:tuple(x:bytearray,y:bytearray),c:int)}
	expressly	{((expressly, gave), 1), ((expressly, reasserted), 1)}

fgWordFreq	w:bytearray	c:long
	expressly	2

# PIG Features

- LOAD '*hdfs-path*' AS (*schema*)
  - *schemas can include int, double, bag, map, tuple, ...*
- FOREACH *alias* GENERATE ... AS ..., ...
  - *transforms each row of a relation*
- DESCRIBE *alias*/ILLUSTRATE *alias* -- *debugging*
- GROUP *alias* BY ...
- FOREACH *alias* GENERATE *group*, SUM(...)
  - *GROUP/GENERATE ... aggregate op together act like a map-reduce*
  - *aggregates: COUNT, SUM, AVERAGE, MAX, MIN, ...*
  - *you can write your own*

PIG parses and **optimizes** a sequence of commands before it executes them  
It's smart enough to turn GROUP ... FOREACH... SUM ... into a map-reduce

-- compute word frequencies

```
fgWordFreq1 = GROUP fgPhrases BY xy.x;
fgWordFreq = FOREACH fgWordFreq1 GENERATE group as w, SUM(fgPhrases.c) as c;

bgWordFreq1 = GROUP bgPhrases BY xy.x;
bgWordFreq = FOREACH bgWordFreq1 GENERATE group as w, SUM(bgPhrases.c) as c;
-- STORE bgWordFreq INTO 'phrases/data/bgWordFreq';
```

# **Phrase Finding 3 - assembling phrase- and word-level statistics**

```

-- join in phrase stats, and then clean up
phraseStats1 = JOIN fgPhrases BY xy, bgPhrases BY xy;
phraseStats2 = FOREACH phraseStats1
    GENERATE fgPhrases::xy AS xy, fgPhrases::c AS fC, bgPhrases::c AS bC;

-- join in word freqs for x and clean up
phraseStats3 = JOIN fgWordFreq BY w, bgWordFreq BY w, phraseStats2 by xy.x;
phraseStats4 = FOREACH phraseStats3
    GENERATE xy,fC,bC,fgWordFreq::c as fxC,bgWordFreq::c as bxC;

-- join in word freqs for y and clean up
phraseStats5 = JOIN fgWordFreq BY w, bgWordFreq BY w, phraseStats4 by xy.y;
phraseStats6 = FOREACH phraseStats5
    GENERATE xy,fC,bC,fxC,bxC,fgWordFreq::c as fyC,bgWordFreq::c as byC;

phraseStats1: {fgPhrases::xy: (x: bytearray,y: bytearray),fgPhrases::c: int,
              bgPhrases::xy: (x: bytearray,y: bytearray),bgPhrases::c: int}

```



bgWordFreq1	group: bytearray	bgPhrases: bag{tuple(xy: tuple(x: bytearray, y: bytearray), c: int)}	
	friday afternoon	{{(friday, afternoon), 1}} {{(afternoon, service), 1}, ((afternoon, mando), 1)}	

bgWordFreq	w: bytearray	c: long	
	friday afternoon	1 2	

phraseStats1	fgPhrases::xy: tuple(x: bytearray, y: bytearray)	fgPhrases::c: int	bgPhrases::xy: tuple(x: bytearray, y: bytearray)	bgPhrases::c: int	
	(friday, afternoon)	1	(friday, afternoon)	1	

phraseStats2	xy: tuple(x: bytearray, y: bytearray)	fc: int	bc: int	
	(friday, afternoon)	1	1	

phraseStats3	fgWordFreq::w: bytearray	fgWordFreq::c: long	bgWordFreq::w: bytearray	bgWordFreq::c: long	phraseStats2::xy: tuple(x: bytearray, y: bytearray)	phraseStats2::fc: int	phraseStats2::bc: int	
	friday	2	friday	1	(friday, afternoon)	1	1	

phraseStats4	phraseStats2::xy: tuple(x: bytearray, y: bytearray)	phraseStats2::fc: int	phraseStats2::bc: int	fxC: long	bxC: long	
	(friday, afternoon)	1	1	2	1	

....

phraseStats6	phraseStats4::phraseStats2::xy: tuple(x: bytearray, y: bytearray)	phraseStats4::phraseStats2::fc: int	phraseStats4::phraseStats2::bc: int	phraseStats4::fxC: long	phraseStats4::bxC: long	fyC: long	byC: long	
	(friday, afternoon)	1	1	2	1	1	2	

bgWordFreq1	group: bytearray	bgPhrases: bag{tuple(xy: tuple(x: bytearray, y: bytearray), c: int)}	
	friday afternoon	{{(friday, afternoon), 1}} {{(afternoon, service), 1}, ((afternoon, mando), 1)}	

bgWordFreq	w: bytearray	c: long	
	friday afternoon	1 2	

phraseStats1	fgPhrases::xy: tuple(x: bytearray, y: bytearray)	fgPhrases::c: int	bgPhrases::xy: tuple(x: bytearray, y: bytearray)	bgPhrases::c: int	
	(friday, afternoon)	1	(friday, afternoon)	1	

phraseStats2	xy: tuple(x: bytearray, y: bytearray)	fc: int	bc: int	
	(friday, afternoon)	1	1	

phraseStats3	fgWordFreq::w: bytearray	fgWordFreq::c: long	bgWordFreq::w: bytearray	bgWordFreq::c: long	phraseStats2::xy: tuple(x: bytearray, y: bytearray)	phraseStats2::fc: int	phraseStats2::bc: int	fxC: long	bxC: long	
	friday	2	friday	1	(friday, afternoon)	1	1	2	1	

phraseStats4	phraseStats2::xy: tuple(x: bytearray, y: bytearray)	phraseStats2::fc: int	phraseStats2::bc: int	fxC: long	bxC: long	
	(friday, afternoon)	1	1	2	1	8

# PIG Features

- LOAD *'hdfs-path'* AS (*schema*)
  - *schemas can include int, double, bag, map, tuple, ...*
- FOREACH *alias* GENERATE ... AS ..., ...
  - *transforms each row of a relation*
- DESCRIBE *alias*/ILLUSTRATE *alias* -- *debugging*
- GROUP *alias* BY ...
- FOREACH *alias* GENERATE *group*, SUM(...)
  - *GROUP/GENERATE ... aggregate op together act like a map-reduce*
- JOIN *r* BY *field*, *s* BY *field*, ...
  - *inner join to produce rows: r::f1, r::f2, ... s::f1, s::f2, ...*

```
phraseStats1: {fgPhrases::xy: (x: bytearray,y: bytearray),fgPhrases::c: int,  
                bgPhrases::xy: (x: bytearray,y: bytearray),bgPhrases::c: int}
```

# Phrase Finding 4 - adding total frequencies

```

grunt> fgPhraseCount1 = group fgPhrases1 ALL;
fgPhraseCount1 = group fgPhrases1 ALL;
2014-04-01 16:57:31,934 [main] WARN org.apache.pig.PigServer - Encountered
2014-04-01 16:57:31,934 [main] WARN org.apache.pig.PigServer - Encountered
grunt> fgPhraseCount = FOREACH fgPhraseCount1 GENERATE SUM(fgPhrases1.c);
fgPhraseCount = FOREACH fgPhraseCount1 GENERATE SUM(fgPhrases1.c);
2014-04-01 16:57:34,607 [main] WARN org.apache.pig.PigServer - Encountered
2014-04-01 16:57:34,607 [main] WARN org.apache.pig.PigServer - Encountered
grunt> bgPhraseCount1 = group bgPhrases1 ALL;
bgPhraseCount1 = group bgPhrases1 ALL;
2014-04-01 16:57:38,271 [main] WARN org.apache.pig.PigServer - Encountered
2014-04-01 16:57:38,271 [main] WARN org.apache.pig.PigServer - Encountered
grunt> bgPhraseCount = FOREACH bgPhraseCount1 GENERATE SUM(bgPhrases1.c);
bgPhraseCount = FOREACH bgPhraseCount1 GENERATE SUM(bgPhrases1.c);
2014-04-01 16:57:40,577 [main] WARN org.apache.pig.PigServer - Encountered
2014-04-01 16:57:40,577 [main] WARN org.apache.pig.PigServer - Encountered

```

bgPhrases1	xy:bytearray	c:int
	continuing series	1
	neighboring lower	1

bgPhraseCount1	group:chararray	bgPhrases1:bag{:tuple(xy:bytearray,c:int)}
	all	{(continuing series, 1), (neighboring lower, 1)}

bgPhraseCount	:long
	2

How do we add the totals to the phraseStats relation?

```
grunt> counts1 = CROSS fgPhraseCount,bgPhraseCount;  
counts1 = CROSS fgPhraseCount,bgPhraseCount;  
2014-04-01 16:59:38,370 [main] WARN org.apache.pig.PigServer - I  
2014-04-01 16:59:38,370 [main] WARN org.apache.pig.PigServer - I  
grunt> counts = FOREACH counts1 GENERATE $0 AS fTot,$1 as bTot;  
counts = FOREACH counts1 GENERATE $0 AS fTot,$1 as bTot;  
2014-04-01 16:59:42,024 [main] WARN org.apache.pig.PigServer - I  
2014-04-01 16:59:42,024 [main] WARN org.apache.pig.PigServer - I  
grunt> phraseStats = CROSS phraseStats6,counts;  
phraseStats = CROSS phraseStats6,counts;  
2014-04-01 16:59:45,083 [main] WARN org.apache.pig.PigServer - I  
2014-04-01 16:59:45,083 [main] WARN org.apache.pig.PigServer - I  
grunt> STORE phraseStats INTO 'phrases/data/phraseStats';
```

**STORE** triggers execution of the query plan....

it also limits optimization



fs -tail phrases/data/phraseStats/part-r-00001

(preliminary,data)	1	1	2	16	4	39	9194	99888
(best,way)	1	5	15	164	3	53	9194	99888
(tour,reached)	1	1	1	3	1	25	9194	99888
(right,way)	1	1	25	85	3	53	9194	99888
(cold,war)	1	19	1	60	16	53	9194	99888
(long,way)	1	10	9	291	3	53	9194	99888
(best,book)	1	1	15	164	3	31	9194	99888
(receive,new)	1	1	4	20	81	1083	9194	99888
(just,got)	6	2	68	258	14	68	9194	99888
(really,got)	1	1	19	148	14	68	9194	99888
(phone,calls)	1	1	7	9	1	11	9194	99888
(congressional,offices)	1	1	7	12	1	4	9194	99888
(second,major)	1	3	11	193	20	182	9194	99888
(special,events)	1	2	6	163	1	12	9194	99888
(civil,rights)	2	5	11	59	6	6	9194	99888
(managing,editor)	1	1	1	2	1	8	9194	99888
(national,press)	1	1	41	255	23	41	9194	99888
(associated,press)	3	1	3	9	23	41	9194	99888
(senate,foreign)	3	2	18	26	7	98	9194	99888
(law,clerk)	1	1	5	47	1	5	9194	99888
(making,clear)	1	1	7	75	7	30	9194	99888
(mutual,fund)	1	1	1	23	2	14	9194	99888
(court,justices)	1	1	21	74	2	2	9194	99888
(sharp,contrast)	1	2	1	41	1	4	9194	99888
(foreign,policy)	1	18	7	98	3	31	9194	99888

Comment: schema is lost when you store....

# PIG Features

- LOAD *'hdfs-path'* AS (*schema*)
  - *schemas can include int, double, bag, map, tuple, ...*
- FOREACH *alias* GENERATE ... AS ..., ...
  - *transforms each row of a relation*
- DESCRIBE *alias*/ILLUSTRATE *alias* -- *debugging*
- GROUP *alias* BY ...
- FOREACH *alias* GENERATE *group*, SUM(...)
  - *GROUP/GENERATE ... aggregate op together act like a map-reduce*
- JOIN *r* BY *field*, *s* BY *field*, ...
  - *inner join to produce rows: r::f1, r::f2, ... s::f1, s::f2, ...*
- CROSS *r*, *s*, ...
  - *use with care unless all but one of the relations are singleton*
  - *newer pigs allow singleton relation to be **cast to a scalar***

# **Phrase Finding 5 - phrasiness and informativeness**



```
package com.wcohen;
```

```
import java.io.*;  
import java.util.*;
```

```
import org.apache.pig.*;  
import org.apache.pig.data.*;  
import org.apache.pig.impl.util.WrappedIOException;
```

```
public class SmoothedPKL extends EvalFunc<Double>  
{  
    public static double smoothPKL(double k1,double n1,double k2,double n2,double p0,double m) {  
        return PKL(k1 + p0*m, n1+m, k2+p0*m, n2+m);  
    }  
    public static double PKL(double k1,double n1,double k2,double n2) {  
        double p1 = k1/n1;  
        double p2 = k2/n2;  
        return p1 * Math.log(p1/p2);  
    }  
  
    @Override  
    public Double exec(Tuple input) throws IOException {  
        if (input==null || input.size()!=6) { return null; }  
        double k1,n1,k2,n2,p0,m;  
        try {  
            k1 = DataType.toDouble(input.get(0));  
            n1 = DataType.toDouble(input.get(1));  
            k2 = DataType.toDouble(input.get(2));  
            n2 = DataType.toDouble(input.get(3));  
            p0 = DataType.toDouble(input.get(4));  
            m = DataType.toDouble(input.get(5));  
        } catch (Exception e) {  
            throw WrappedIOException.wrap("Error in Phrases processing row ",e);  
        }  
        return smoothPKL(k1,n1,k2,n2,p0,m);  
    }  
}
```

How do we compute some complicated function?

With a “UDF”

```

phraseStats = LOAD 'phrases/data/phraseStats' AS (xy:(x,y), fC, bC, fxC, bxC, fyC, byC, fTot, bTot);

-- final compute phraseness, etc

REGISTER ./pkl.jar;

phraseResult = FOREACH phraseStats GENERATE *,
    com.wcohen.SmoothedPKL(fC, fTot, bC, bTot, 1.0/bTot, 1.0) as infoness,
    com.wcohen.SmoothedPKL(fC, fTot, fxC*fyC, fTot*fTot, 1.0/fxC, 1.0) as phraseness;

STORE phraseResult INTO 'phrases/data/phraseResult';

```

# PIG Features

- LOAD *'hdfs-path'* AS (*schema*)
  - *schemas can include int, double, bag, map, tuple, ...*
- FOREACH *alias* GENERATE ... AS ..., ...
  - *transforms each row of a relation*
- DESCRIBE *alias*/ILLUSTRATE *alias* -- *debugging*
- GROUP *alias* BY ...
- FOREACH *alias* GENERATE *group*, SUM(...)
  - *GROUP/GENERATE ... aggregate op together act like a map-reduce*
- JOIN *r* BY *field*, *s* BY *field*, ...
  - *inner join to produce rows: r::f1, r::f2, ... s::f1, s::f2, ...*
- CROSS *r*, *s*, ...
  - *use with care unless all but one of the relations are singleton*
- User defined functions as operators
  - *also for loading, aggregates, ...*

# The full phrase-finding pipeline

```

-- load data
fgPhrases1 = LOAD 'phrases/data/dkos-phraseFreq-5/' AS (xy,c:int);
fgPhrases = FOREACH fgPhrases1 GENERATE STRSPLIT(xy,' ') AS xy:(x,y), c AS c;
bgPhrases1 = LOAD 'phrases/data/brown-phraseFreq-5/' AS (xy,c:int);
bgPhrases = FOREACH bgPhrases1 GENERATE STRSPLIT(xy,' ') AS xy:(x,y), c AS c;

-- compute word frequencies
fgWordFreq1 = GROUP fgPhrases BY xy.x;
fgWordFreq = FOREACH fgWordFreq1 GENERATE group as w,SUM(fgPhrases.c) as c;
bgWordFreq1 = GROUP bgPhrases BY xy.x;
bgWordFreq = FOREACH bgWordFreq1 GENERATE group as w,SUM(bgPhrases.c) as c;

-- join in phrase stats, and then clean up schema
phraseStats1 = JOIN fgPhrases BY xy, bgPhrases BY xy;
STORE phraseStats1 INTO 'phrases/data/phraseStats1';
phraseStats2 = FOREACH phraseStats1 GENERATE fgPhrases::xy AS xy, fgPhrases::c AS fC, bgPhrases::c AS bC;
-- join in word freqs for x and clean up
phraseStats3 = JOIN fgWordFreq BY w, bgWordFreq BY w, phraseStats2 by xy.x;
phraseStats4 = FOREACH phraseStats3 GENERATE xy,fC,bC,fgWordFreq::c as fxC,bgWordFreq::c as bxC;
-- join in word freqs for y and clean up
phraseStats5 = JOIN fgWordFreq BY w, bgWordFreq BY w, phraseStats4 by xy.y;
phraseStats6 = FOREACH phraseStats5 GENERATE xy,fC,bC,fxC,bxC,fgWordFreq::c as fyC,bgWordFreq::c as byC;

-- compute totals
fgPhraseCount1 = group fgPhrases1 ALL;
fgPhraseCount = FOREACH fgPhraseCount1 GENERATE SUM(fgPhrases1.c);
bgPhraseCount1 = group bgPhrases1 ALL;
bgPhraseCount = FOREACH bgPhraseCount1 GENERATE SUM(bgPhrases1.c);

-- join in totals - ok to use cross-product here since all but one table are just singletons
counts1 = CROSS fgPhraseCount,bgPhraseCount;
counts = FOREACH counts1 GENERATE $0 AS fTot,$1 as bTot;
phraseStats = CROSS phraseStats6,counts;

-- finally compute phraseness, etc

REGISTER ./pkl.jar;
phraseResult = FOREACH phraseStats GENERATE *,
    com.wcohen.SmootherPKL(fC, fTot, bC, bTot, 1.0/bTot, 1.0) as infoness,
    com.wcohen.SmootherPKL(fC, fTot, fxC*fyC, fTot*fTot, 1.0/fxC, 1.0) as phraseness;
STORE phraseResult INTO 'phrases/data/phraseResult';

```