Spark vs Hadoop

Spark

Set of concise dataflow operations ("transformation")

Dataflow operations are embedded in an API together with "actions"

- 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

Sharded files are replaced by "RDDs" – resiliant distributed datasets

RDDs can be cached in *cluster* memory and recreated to recover from error

Spark examples

```
errors.cache()
```

spark is a *spark context* object

```
text_file = spark.textFile("hdfs://...")
errors = text_file.filter(lambda line: "ERROR" in line)
# Count all the errors
errors.count()
# Count errors mentioning MySQL
errors.filter(lambda line: "MySQL" in line).count()
# Fetch the MySQL errors as an array of strings
errors.filter(lambda line: "MySQL" in line).collect()
```

Spark examples

```
errors.cache()
```

errors is a *transformation*, and
thus a
that exp
do

```
text_file = spark.textFile("hd
errors = text_file.filter(
# Count all the errors
errors.count()
# Count errors mentioning MySQL
errors.filter(lambda line: "MySQL" in line).count()
# Fetch the MySQL errors as an array of strings
errors.filter(lambda line: "MySQL" in line).collect()
```

count() is an *action*: it will actually execute the plan for **errors** and return a value.

everything is **sharded**, like in Hadoop and GuineaPig

errors.filter() is a *transformation*

collect() is an *action*

Spark examples

everything is **sharded** … and the shards are stored in *memory* of worker machines not local *disk* (if possible)

```
text_file = spark.textFile("hdfs://...")
errors = text_file.filter(lambda line: "ERROR" in line)
errors.cache()
errors.count()
# Count errors mentioning MySQL
errors.filter(lambda line: "MySQL" in line).count()
# Fetch the MySQL errors as an array of strings
errors.filter(lambda line: "MySQL
```

\# modify **errors** to be stored in *cluster memory*

subsequent actions will be much faster

You can also **persist()** an RDD on disk, which is like marking it as opts(stored=True) in GuineaPig. Spark's *not* smart about persisting data.

Spark examples: wordcount

```
text_file = spark.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

the action

transformation on (key,value) pairs , which are special

Spark examples: batch logistic regression

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

reduce is an action – it produces a numpy vector

p.x and **w** are *vectors*, from the numpy package. Python overloads operations like * and + for vectors.

Spark examples: batch logistic regression

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

Important note: numpy vectors/matrices are not just "syntactic sugar".

- They are *much more compact* than something like a list of python floats.
- numpy operations like **dot, *, +** are calls to *optimized C code*
- a little python logic around a lot of numpy calls is pretty efficient

Spark examples: batch logistic regression

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s"   w
```

So: python builds a *closure* – code including the *current value* of **w** – and Spark ships it off to each worker. So **w** is *copied*, and must be *read-only*.

w is defined *outside* the lambda function, but used *inside* it

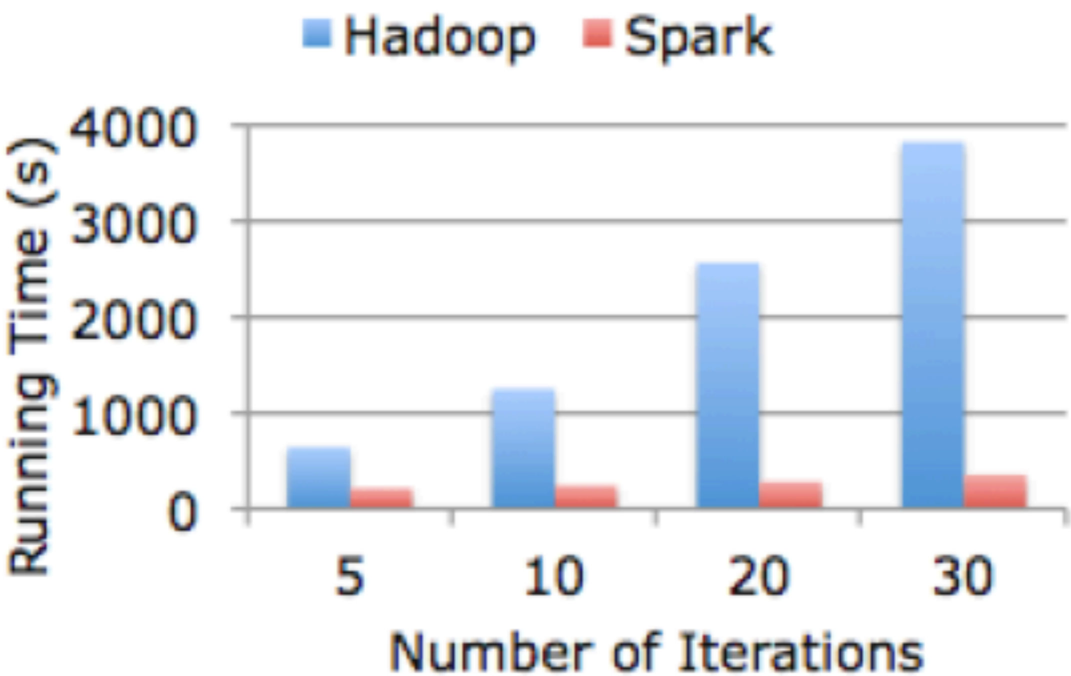
Spark examples: batch logistic regression

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

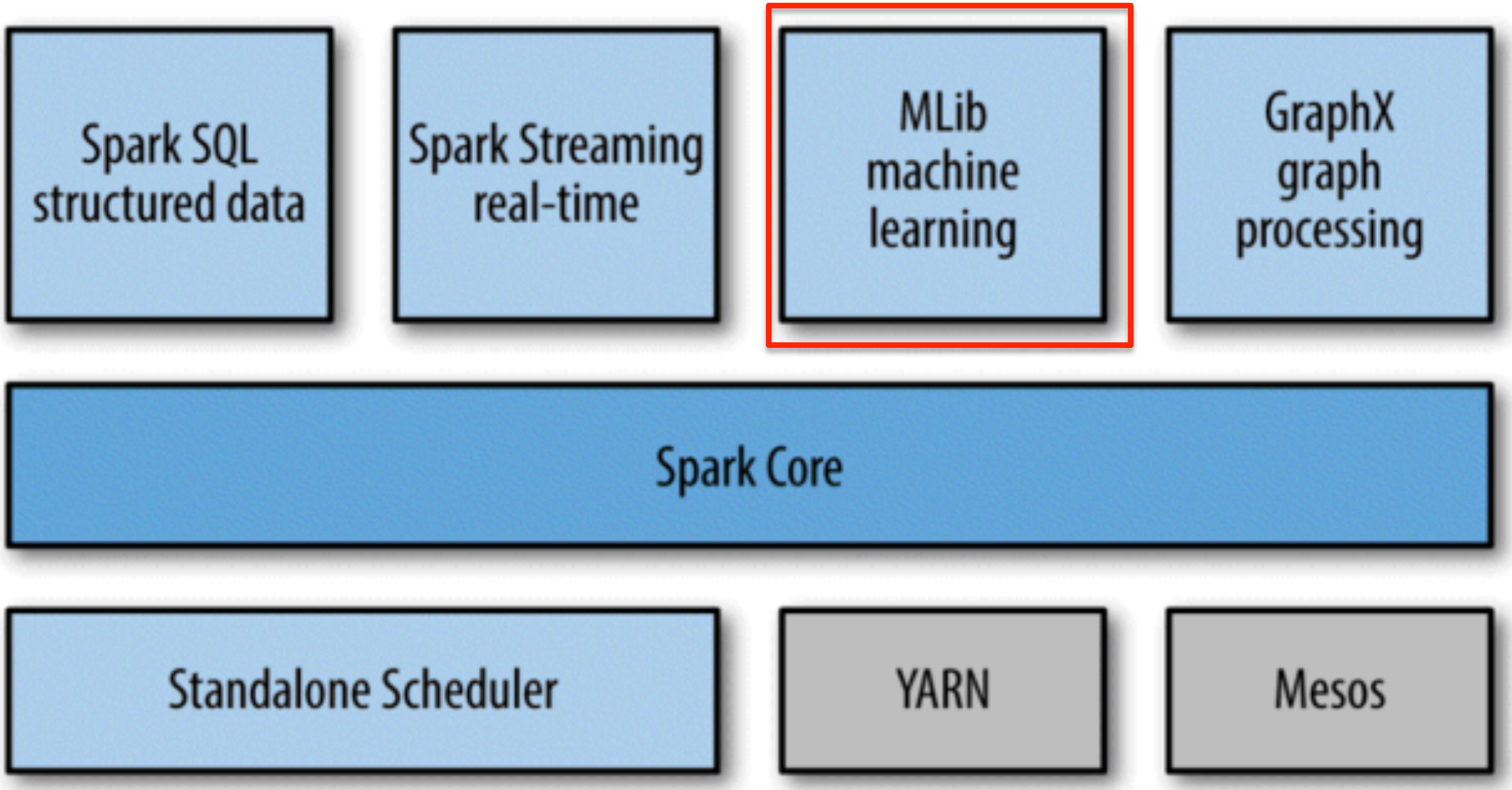
dataset of points is cached in cluster memory to reduce i/o

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:



Spark


Spark SQL
structured data
Spark Streaming
real-time
MLib
machine
learning
GraphX
graph
processing
Spark Core
Standalone Scheduler
YARN
Mesos

Spark details: broadcast

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

So: python builds a *closure* – code including the *current value* of **w** – and Spark ships it off to each worker. So **w** is *copied*, and must be *read-only*.

Spark details: broadcast

little penalty for distributing something that's not used by all workers

```
points = spark.textFile(...).map(parsePoint).cac
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %
```

alternative: create a *broadcast variable*, e.g.,

- w_broad = spark.broadcast(w)

which is accessed by the worker via

- w_broad.value()

what's sent is a *small pointer* to **w** (e.g., the name of a file containing a serialized version of **w**) and when **value** is called, some clever all-reduce like machinery is used to reduce network load.

Spark details: mapPartitions

```
class WordProb(Planner):

    wc = ReadLines('corpus.txt') | Flatten(by=tokens) \
         | Group(by=lambda x:x, reducingTo=ReduceToCount())
    total = ...
    wcWithTotal = Augment(wc, sideview=total,loadedBy=lambda v:GPig.onlyRowOf(v))
    prob = ReplaceEach(wcWithTotal, by=lambda ((word,count),n): (word,count,n,float(count)/n))
```

Common issue:

- map task requires loading in some small shared value
- more generally, map task requires some sort of *initialization* before processing a shard
- GuineaPig:
  - special *Augment* … *sideview* … pattern for shared values
  - can kludge up any initializer using Augment
- Raw Hadoop: **mapper.configure()** and **mapper.close()** methods

Spark details: mapPartitions

```
class WordProb(Planner):

    wc = ReadLines('corpus.txt') | Flatten(by=tokens) \
         | Group(by=lambda x:x, reducingTo=ReduceToCount())
    total = ...
    wcWithTotal = Augment(wc, sideview=total,loadedBy=lambda v:GPig.onlyRowOf(v))
    prob = ReplaceEach(wcWithTotal, by=lambda ((word,count),n): (word,count,n,float(count)/n))
```

Spark:

- **rdd.mapPartitions(f)**: will call **f(iteratorOverShard)** once per shard, and return an iterator over the mapped values.

- **f()** can do any setup/close steps it needs

Also:

- there are transformations to partition an RDD with a user-selected function, like in Hadoop. Usually you partition and persist/cache.

Other Map-Reduce (ish) Frameworks

William Cohen

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 );

flow.complete();
```

Run the pipeline

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

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

```
Sch
Tap

Pip

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

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…?

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

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.

FLINK

- Recent Apache Project – 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")
  }
}
```

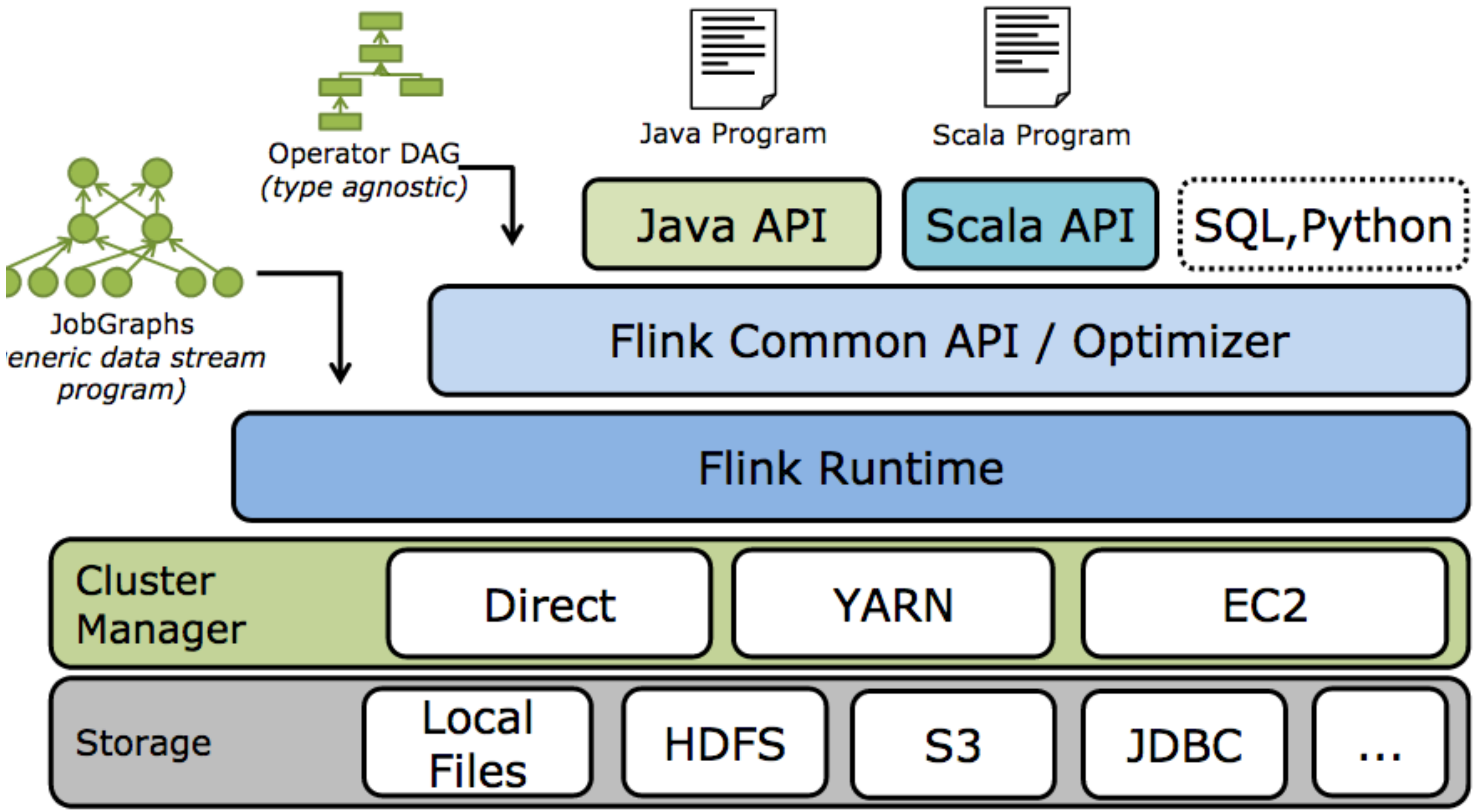
Java API

```
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



Operator DAG
(type agnostic)
Java Program
Scala Program
JobGraphs
eneric data stream
program)
Java API
Scala API
SQL,Python
Flink Common API / Optimizer
Flink Runtime
Cluster Manager
Direct
YARN
EC2
Storage
Local Files
HDFS
S3
JDBC
...

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

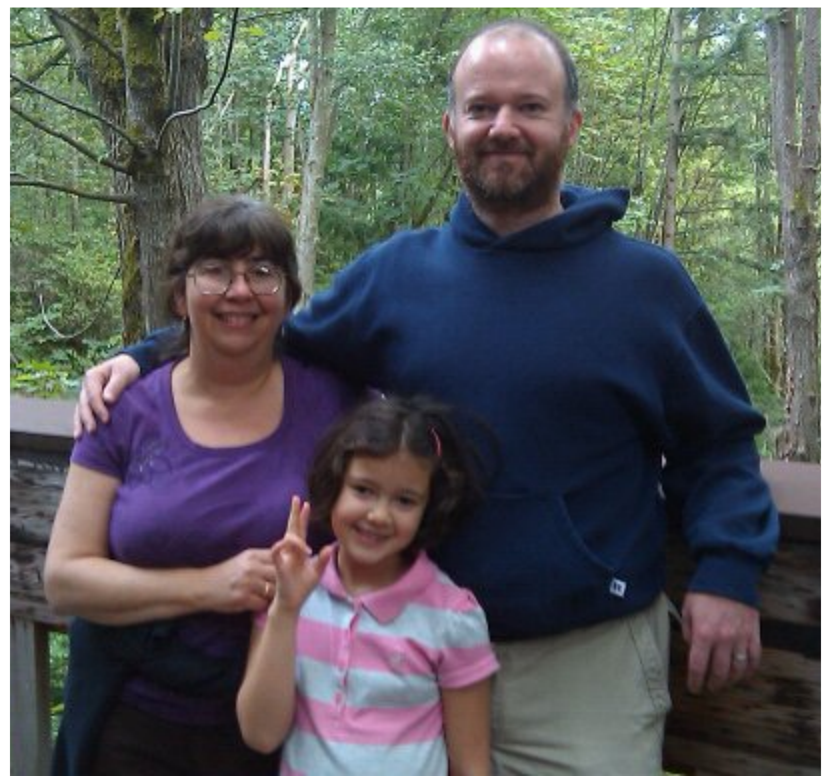
One more algorithm to discuss as a Map-reduce implementation….

A Language Model Approach to Keyphrase Extraction

Takashi Tomokiyo and Matthew Hurst
Applied Research Center
Intelliseek, Inc.
Pittsburgh, PA 15213
{ttomokiyo,mhurst}@intelliseek.com

A Language Model Approach to Keyphrase Extraction

Takashi Tomokiyo and Matthew Hurst
Applied Research Center
Intelliseek, Inc.
Pittsburgh, PA 15213
{ttomokiyo,mhurst}@intelliseek.com



ACL Workshop 2003

1	civic hybrid	21	mustang gt
2	honda civic hybrid	22	ford escape
3	toyota prius	23	steering wheel
4	electric motor	24	toyota prius today
5	honda civic	25	electric motors
6	fuel cell	26	gasoline engine
7	hybrid cars	27	internal combustion engine
8	honda insight	28	gas engine
9	battery pack	29	front wheels
10	sports car	30	key sense wire
11	civic si	31	civic type r
12	hybrid car	32	test drive
13	civic lx	33	street race
14	focus fcv	34	united states
15	fuel cells	35	hybrid powertrain
16	hybrid vehicles	36	rear bumper
17	tour de sol	37	ford focus
18	years ago	38	detroit auto show
19	daily driver	39	parking lot
20	jetta tdi	40	rear wheels

Figure 1: Top 40 keyphrases automatically extracted from messages relevant to "*civic hybrid*" using our system

Why phrase-finding?

- There are lots of phrases
- There's not supervised data
- It's hard to articulate
  - What makes a phrase a phrase, *vs* just an n-gram?
    - a phrase is independently meaningful ("test drive", "red meat") or not ("are interesting", "are lots")
  - What makes a phrase interesting?

The breakdown: what makes a good phrase

- Two properties:
  - Phraseness: "the degree to which a given word sequence is considered to be a phrase"
    - Statistics: how often words co-occur together vs separately
  - Informativeness: "how well a phrase captures or illustrates the key ideas in a set of documents" – something novel and important **relative to a domain**
    - Background corpus and foreground corpus; how often phrases occur in each

“Phraseness”$_{\text{I}}$ – based on BLRT

- Binomial Ratio Likelihood Test (BLRT):
  - Draw samples:
    - n_1 draws, k_1 successes
    - n_2 draws, k_2 successes
    - Are they from one binominal (i.e., k_1/n_1 and k_2/n_2 were different due to chance) or from two distinct binomials?
  - Define
    - $p_1=k_1 / n_1, p_2=k_2 / n_2, p=(k_1+k_2)/(n_1+n_2),$
    - $L(p,k,n) = p^k(1-p)^{n-k}$

$$BLRT(n_1,k_1,n_2,k_2) = \frac{L(p_1,k_1,n_1)L(p_2,k_2,n_2)}{L(p,k_1,n_1)L(p,k_2,n_2)}$$

"Phraseness"$_1$ – based on BLRT

- Binomial Ratio Likelihood Test (BLRT):
  - Draw samples:
    - n_1 draws, k_1 successes
    - n_2 draws, k_2 successes
    - Are they from one binominal (i.e., k_1/n_1 and k_2/n_2 were different due to chance) or from two distinct binomials?
  - Define
    - $p_i = k_i / n_i$, $p = (k_1 + k_2)/(n_1 + n_2)$,
    - $L(p,k,n) = p^k(1-p)^{n-k}$

$$BLRT(n_1, k_1, n_2, k_2) = 2 \log \frac{L(p_1, k_1, n_1) L(p_2, k_2, n_2)}{L(p, k_1, n_1) L(p, k_2, n_2)}$$

"Informativeness"$_I$ – based on BLRT

Phrase *x y*: $W_1=x \wedge W_2=y$ and two corpora, C and B

– Define

- $p_i=k_i/n_i$, $p=(k_1+k_2)/(n_1+n_2)$,
- $L(p,k,n) = p^k(1-p)^{n-k}$

$$\varphi_i(n_1,k_1,n_2,k_2) = 2\log\frac{L(p_1,k_1,n_1)L(p_2,k_2,n_2)}{L(p,k_1,n_1)L(p,k_2,n_2)}$$

		comment
k_1	$C(W_1=x \wedge W_2=y)$	how often bigram *x y* occurs in corpus C
n_1	$C(W_1=* \wedge W_2=*)$	how many bigrams in corpus C
k_2	$B(W_1=x \wedge W_2=y)$	how often *x y* occurs in **background corpus**
n_2	$B(W_1=* \wedge W_2=*)$	how many bigrams in background corpus

Does x *y* occur at the same frequency in both corpora?

"Phraseness"$_1$ – based on BLRT

Phrase x y: $W_1=x \wedge W_2=y$

– Define

- $p_i=k_i/n_i$, $p=(k_1+k_2)/(n_1+n_2)$,
- $L(p,k,n) = p^k(1-p)^{n-k}$

$$\varphi_p(n_1,k_1,n_2,k_2) = 2\log\frac{L(p_1,k_1,n_1)L(p_2,k_2,n_2)}{L(p,k_1,n_1)L(p,k_2,n_2)}$$

		comment
k_1	$C(W_1=x \wedge W_2=y)$	how often bigram *x y* occurs in corpus C
n_1	$C(W_1=x)$	how often word *x* occurs in corpus C
k_2	$C(W_1\neq x \wedge W_2=y)$	how often *y* occurs in C after a non-*x*
n_2	$C(W_1\neq x)$	how often a non-*x* occurs in C

Does *y* occur at the same frequency after *x* as in other positions?

The breakdown: what makes a good phrase

- "Phraseness" and "informativeness" are then combined with a tiny classifier, tuned on labeled data.

$$\varphi = \frac{1}{1 + \exp(-a\varphi_p - b\varphi_i + c)}$$

$$\left(\log \frac{p}{1-p} = s\right) \Leftrightarrow \left(p = \frac{1}{1+e^s}\right)$$

  - Background corpus: 20 newsgroups dataset (20k messages, 7.4M words)
  - Foreground corpus: rec.arts.movies.current-films June-Sep 2002 (4M words)
  - Results?

1	message news	16	sixth sense
2	minority report	17	hey kids
3	star wars	18	gaza man
4	john harkness	19	lee harrison
5	derek janssen	20	years ago
6	robert frenchu	21	julia roberts
7	sean o'hara	22	national guard
8	box office	23	bourne identity
9	dawn taylor	24	metrotoday www.zap2it.com
10	anthony gaza	25	starweek magazine
11	star trek	26	eric chomko
12	ancient race	27	wilner starweek
13	scooby doo	28	tim gueguen
14	austin powers	29	jodie foster
15	home.attbi.com hey	30	johnnie kendricks

The breakdown: what makes a good phrase

- Two properties:
  - Phraseness: “the degree to which a given word sequence is considered to be a phrase”
    - Statistics: how often words co-occur together vs separately
  - Informativeness: “how well a phrase captures or illustrates the key ideas in a set of documents” – something novel and important relative to a domain
    - Background corpus and foreground corpus; how often phrases occur in each
  - Another intuition: our goal is to compare distributions and see how **different** they are:
    - Phraseness: estimate *x y* with bigram model or unigram model
    - Informativeness: estimate with foreground vs background corpus

The breakdown: what makes a good phrase

- Another intuition: our goal is to compare distributions and see how **different** they are:
  - Phraseness: estimate *x y* with bigram model or unigram model
  - Informativeness: estimate with foreground vs background corpus
- To compare distributions, use KL-divergence

$$D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

"Pointwise KL divergence"

$$\delta_{\mathbf{w}}(p \parallel q) \stackrel{\text{def}}{=} p(\mathbf{w}) \log \frac{p(\mathbf{w})}{q(\mathbf{w})}$$

The breakdown: what makes a good phrase

- To compare distributions, use KL-divergence

$$D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

"Pointwise KL divergence"

$$\delta_{\mathbf{w}}(p \parallel q) \stackrel{\text{def}}{=} p(\mathbf{w}) \log \frac{p(\mathbf{w})}{q(\mathbf{w})}$$

Phraseness: difference between bigram and unigram language model in foreground

$$\delta_{\mathbf{w}}(LM_{\text{fg}}^{N} \parallel LM_{\text{fg}}^{1})$$

Bigram model: P(x y)=P(x)P(y|x)

Unigram model: P(x y)=P(x)P(y)

The breakdown: what makes a good phrase

– To compare distributions, use KL-divergence

$$D(p \,||\, q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

“Pointwise KL divergence”

$$\delta_{\mathbf{w}}(p \,||\, q) \stackrel{\text{def}}{=} p(\mathbf{w}) \log \frac{p(\mathbf{w})}{q(\mathbf{w})}$$

Bigram model: P(x y)=P(x)P(y|x)

Unigram model: P(x y)=P(x)P(y)

Informativeness: difference between foreground and background models

$$\delta_{\mathbf{w}}(LM_{\text{fg}}^{N} \,||\, LM_{\text{bg}}^{N}), \text{ or}$$
$$\delta_{\mathbf{w}}(LM_{\text{fg}}^{1} \,||\, LM_{\text{bg}}^{1})$$

$$\delta_{\mathbf{w}}(LM_{\text{fg}}^{N} \,||\, LM_{\text{bg}}^{1})$$

The breakdown: what makes a good phrase

– To compare distributions, use KL-divergence

$$D(p \,\|\, q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

"Pointwise KL divergence"

$$\delta_{\mathbf{w}}(p \,\|\, q) \stackrel{\text{def}}{=} p(\mathbf{w}) \log \frac{p(\mathbf{w})}{q(\mathbf{w})}$$

Bigram model: P(x y)=P(x)P(y|x)

Unigram model: P(x y)=P(x)P(y)

Combined: difference between foreground bigram model and background unigram model

$$\delta_{\mathbf{w}}(LM^N_{\text{fg}} \,\|\, LM^1_{\text{bg}})$$

$$\begin{tikzcd}
LM^N_{\text{fg}} \arrow[r, dash, "\longleftarrow\ \mathit{informativeness}\ \longrightarrow"] \arrow[d, dash, "\updownarrow\ \mathit{phraseness}"'] & LM^N_{\text{bg}} \arrow[d, dash] \\
LM^1_{\text{fg}} \arrow[r, dash] & LM^1_{\text{bg}}
\end{tikzcd}$$

The breakdown: what makes a good phrase

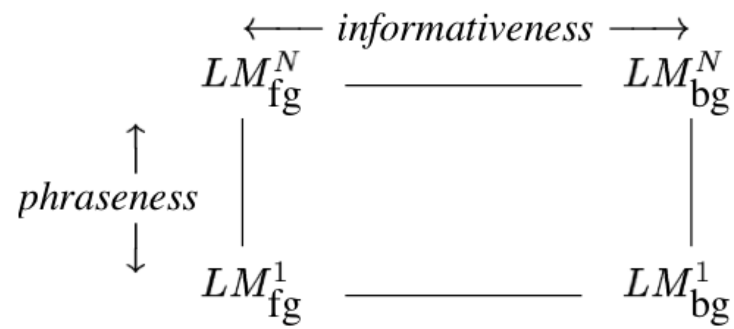
– To compare distributions, use KL-divergence

Subtle advantages:

- BLRT scores “more frequent in foreground” and “more frequent in background” symmetrically, pointwise KL does not.
- Phrasiness and informativeness scores are more comparable – straightforward combination w/o a classifier is reasonable.
- Language modeling is well-studied:
  - extensions to n-grams, smoothing methods, …
  - we can build on this work in a modular way

Combined: difference between foreground bigram model and background unigram model

$$\delta_{\mathbf{w}}(LM_{\text{fg}}^{N} \parallel LM_{\text{bg}}^{1})$$



Pointwise KL, combined

1	message news	16	hey kids
2	minority report	17	years ago
3	star wars	18	gaza man
4	john harkness	19	sixth sense
5	robert frenchu	20	lee harrison
6	derek janssen	21	julia roberts
7	box office	22	national guard
8	sean o'hara	23	bourne identity
9	dawn taylor	24	metrotoday www.zap2it.com
10	anthony gaza	25	starweek magazine
11	star trek	26	eric chomko
12	ancient race	27	wilner starweek
13	home.attbi.com hey	28	tim gueguen
14	scooby doo	29	jodie foster
15	austin powers	30	kevin filmnutboy

Why phrase-finding?

- Phrases are where the standard supervised "bag of words" representation starts to break.
- There's not supervised data, so it's hard to see what's "right" and why
- It's a nice example of using unsupervised signals to solve a task that could be formulated as supervised learning
- It's a nice level of complexity, if you want to do it in a scalable way.

Phrase Finding in Guinea Pig

Phrase Finding 1 – counting words

```
# supporting routines can go here
def tokens(line):
    for tok in line.split():
        yield tok.lower()

class Phrases(Planner):

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

    bgWordCount = wcPipe('browncorpus.txt')
    fgWordCount = wcPipe('dkos-entries.txt')
```

```
('zogby,', 1)
('zombiexx', 1)
('zone', 2)
('zone,', 1)
('zone.', 2)
('zones.', 1)
('zonkette.', 1)
('zope.', 1)
('zuniga', 2)
```

background corpus

Phrase Finding 2 – counting phrases

```
(('Democratic', 'Party,"'), 1)
(('Democratic', 'Party,'), 4)
(('Democratic', 'Party.'), 1)
(('Democratic', 'Senators'), 1)
(('Democratic', 'State'), 1)
(('Democratic', 'U.S.'), 1)
(('Democratic', 'Underground'), 1)
(('Democratic', 'Underground.'), 1)
(('Democratic', 'Veteran'), 1)
(('Democratic', 'accusations'), 1)
```

```
def bigrams(line):
    tokens = line.split()
    for i in range(len(tokens)-1):
        (x,y) = (tokens[i],tokens[
        if (not x in STOPWORDS) ar
            yield (x,y)

def pcPipe(corpus):
    return ReadLines(corpus) \
        | Flatten(by=bigrams) \
        | Group(by=lambda x:x, reducingTo=ReduceToCount())

bgPhraseCount = pcPipe('browncorpus.txt')
fgPhraseCount = pcPipe('dkos-entries.txt')
```

Phrase Finding 3 – collecting info

dictionary: {'statistic name':value}

```
def extendStats(stats,key,val):
    return dict(stats.items() + [(key,val)])
```

returns copy with a new key,value pair

Phrase Finding 3 – collecting info

```
def extendStats(stats,key,val):
    return dict(stats.items() + [(key,val)])


phraseCount = \
    Join( Jin(fgPhraseCount, by=lambda(phrase,fC):phrase),
          Jin(bgPhraseCount, by=lambda(phrase,bC):phrase)) \
    | Map( by=lambda ((phrase1,fC),(phrase2,bC)):(phrase1,{'fC':fC,'bC':bC}))

phraseStats1 = \
    Join( Jin(phraseCount, by=lambda((x,y),stats):x),
          Jin(fgWordCount, by=lambda(w,c):w)) \
    | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendStats(stats, 'fxC', c))) \
    | JoinTo( Jin(bgWordCount, by=lambda(w,c):w), by=lambda((x,y),stats):x) \
    | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendStats(stats, 'bxC', c)))
```

join fg and bg phrase counts and output a dict

join fg and bg count for first word “x” in “x y”

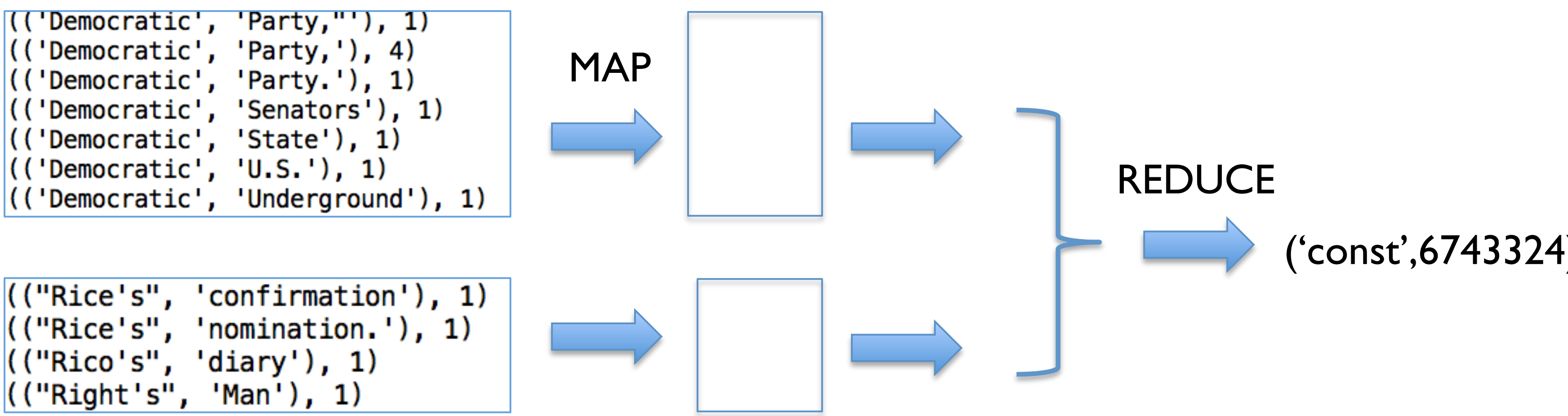
Phrase Finding 3 – collecting info

```
def extendStats(stats,key,val):
    return dict(stats.items() + [(key,val)])


phraseCount = \
    Join( Jin(fgPhraseCount, by=lambda(phrase,fC):phrase),
          Jin(bgPhraseCount, by=lambda(phrase,bC):phrase)) \
    | Map( by=lambda ((phrase1,fC),(phrase2,bC)):(phrase1,{'fC':fC,'bC':bC}))

phraseStats1 = \
    Join( Jin(phraseCount, by=lambda((x,y),stats):x),
          Jin(fgWordCount, by=lambda(w,c):w)) \
    | Map( by=lambda((phrase,stats),(w,c)): (phrase,extend
    | JoinTo( Jin(bgWordCount, by=lambda(w,c):w), by=lambda((     stats):x) \
    | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendSt     stats, 'bxC', c)))


phraseStats2 = \
    Join( Jin(phraseStats1, by=lambda((x,y),stats):y),
          Jin(fgWordCount, by=lambda(w,c):w)) \
    | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendStats(stats, 'fyC', c))) \
    | JoinTo( Jin(bgWordCount, by=lambda(w,c):w), by=lambda((x,y),stats):y) \
    | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendStats(stats, 'byC', c)))
```

join fg and bg count for word “y” in “x y”

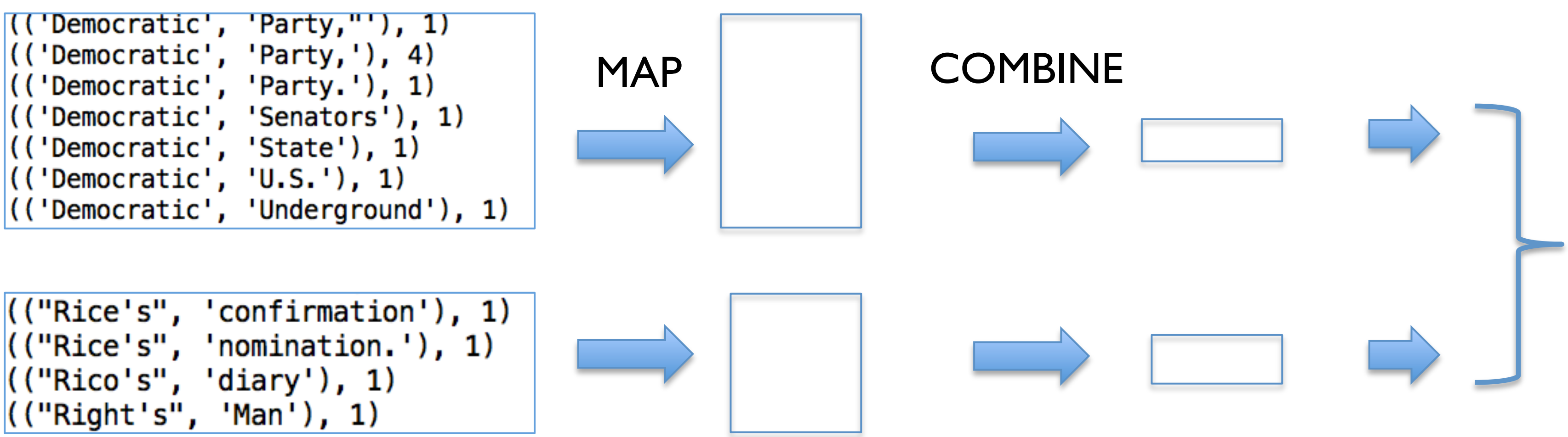
Phrase Finding 4 – totals

```
# compute totals
numBGPhrases = Group(bgPhraseCount,
                     by=lambda(phrase,c):'const',
                     retaining=lambda(phrase,c):c,
                     reducingTo=ReduceToSum(),
                     combiningTo=ReduceToSum())
```



Phrase Finding 4 – totals

```
# compute totals
numBGPhrases = Group(bgPhraseCount,
                     by=lambda(phrase,c):'const',
                     retaining=lambda(phrase,c):c,
                     reducingTo=ReduceToSum(),
                     combiningTo=ReduceToSum())
```



Phrase Finding 4 – totals

```
# compute totals
numBGPhrases = Group(bgPhraseCount,
                     by=lambda(phrase,c):'const',
                     retaining=lambda(phrase,c):c,
                     reducingTo=ReduceToSum(),
                     combiningTo=ReduceToSum())
numFGPhrases = Group(fgPhraseCount,
                     by=lambda(phrase,c):'const',
                     retaining=lambda(phrase,c):c,
                     reducingTo=ReduceToSum(),
                     combiningTo=ReduceToSum())
```

Phrase Finding 4 – totals (map-side)

```
# compute totals
numBGPhrases = Group(bgPhraseCount,
                     by=lambda(phrase,c):'const',
                     retaining=lambda(phrase,c):c,
                     reducingTo=ReduceToSum(),
                     combiningTo=ReduceToSum())
numFGPhrases = Group(fgPhraseCount,
                     by=lambda(phrase,c):'const',
                     retaining=lambda(phrase,c):c,
                     reducingTo=ReduceToSum(),
                     combiningTo=ReduceToSum())
```

```
phraseStats = \
    Augment(phraseStats2,
            sideviews = [numFGPhrases,numFGPhrases],
            loadedBy = lambda nfg,nbg: (GPig.onlyRowOf(nfg),GPig.onlyRowOf(nbg))) \
    | Map(by=lambda ((phrase,stats),((dummy1,fTot),(dummy2,bTot))): \
              (phrase, extendStats(extendStats(stats,'fTot',fTot),'bTot',bTot)))
```

Phrase Finding 5 – collect totals

```
phraseStats = \
    Join( Jin(numFGPhrases,by=lambda(dummy,fTot):'const'),
          Jin(phraseStats2,by=lambda(phrase,stats):'const')) \
    | Map( by=lambda((dummy,fTot),(phrase,stats)):
               (phrase,extendStats(stats,'fTot',fTot))) \
    | JoinTo( Jin(numBGPhrases,by=lambda(dummy,bTot):'const'),
              by=lambda(phrase,stats):'const') \
    | Map( by=lambda((phrase,stats),(dummy,bTot)):
               (phrase,extendStats(stats,'bTot',bTot)))
```

Phrase Finding 6 – compute

```
def PKL(k1,n1,k2,n2):
    p1 = k1/n1
    p2 = k2/n2
    return p1 * math.log(p1/p2)

def smoothPKL(k1,n1,k2,n2,p0,m):
    return PKL(k1 + p0*m, n1+m, k2+p0*m, n2+m)

def infoness(d):
    fC = d['fC']; fTot = d['fTot']; bC = d['bC']; bTot = d['bTot']
    return smoothPKL( fC, fTot, bC, bTot, 1.0/bTot, 1.0)

def phraseness(d):
    fC = d['fC']; fTot = d['fTot']; fxC = d['fxC']; fyC = d['fyC']
    return smoothPKL( fC, fTot, fxC*fyC, fTot*fTot, 1.0/fxC, 1.0)
```

....

```
phraseResult = Map(phraseStats,
                   by=lambda(phrase,stats):
                       (phrase,infoness(stats),phraseness(stats)))
```

Phrase Finding results

Overall

```
('right', 'wing')
('vast', 'majority')
('just', 'got')
("we've", 'got')
("don't", 'think')
('press', 'release')
('voting', 'machines')
('school', 'districts')
('national', 'security')
('people,', 'including')
('immediate', 'threat')
('civil', 'liberties')
```

Phrasiness Only

```
('right', 'wing')
('vast', 'majority')
("don't", 'think')
('school', 'districts')
("we've", 'got')
("don't", 'know')
('voting', 'machines')
('press', 'release')
('years', 'ago')
('national', 'security')
('civil', 'liberties')
('soap', 'opera')
('hospital', 'facilities')
('cocktail', 'circuit')
('aircraft', 'carrier')
('loved', 'ones.')
```

Top 100 phraseiness, lo informativeness

```
('years', 'ago')
('make', 'sure')
('years', 'ago.')
('great', 'deal')
('human', 'beings')
('real', 'estate')
('years', 'old.')
('years', 'old,')
('young', 'men')
("you've", 'got')
```

Phrase Finding results

Overall

```
('right', 'wing')
('vast', 'majority')
('just', 'got')
("we've", 'got')
("don't", 'think')
('press', 'release')
('voting', 'machines')
('school', 'districts')
('national', 'security')
('people,', 'including')
('immediate', 'threat')
('civil', 'liberties')
```

Top 100 informativeness, lo phraseiness

```
('results', '--')
('big', 'story')
('time,', 'said')
('doing', 'it.')
("didn't", 'believe')
('security', 'does')
('way', 'different')
('new', 'legislation')
('said', 'today')
('church', 'like')
```

The full phrase-finding pipeline

```
# supporting routines can go here
def tokens(line):
    for tok in line.split():
        yield tok.lower()

def bigrams(line):
    tokens = line.split()
    for i in range(len(tokens)-1):
        (x,y) = (tokens[i],tokens[i+1])
        if (not x in STOPWORDS) and (not y in STOPWORDS):
            yield (x,y)

def extendStats(stats,key,val):
    return dict(stats.items() + [(key,val)])

def PKL(k1,n1,k2,n2):
    p1 = k1/n1
    p2 = k2/n2
    return p1 * math.log(p1/p2)

def smoothPKL(k1,n1,k2,n2,p0,m):
    return PKL(k1 + p0*m, n1+m, k2+p0*m, n2+m)

def infoness(d):
    fC = d['fC']; fTot = d['fTot']; bC = d['bC']; bTot = d['bTot']
    return smoothPKL( fC, fTot, bC, bTot, 1.0/bTot, 1.0)

def phraseness(d):
    fC = d['fC']; fTot = d['fTot']; fxC = d['fxC']; fyC = d['fyC']
    return smoothPKL( fC, fTot, fxC*fyC, fTot*fTot, 1.0/fxC, 1.0)
```

The full phrase-finding pipeline

```
class Phrases(Planner):

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

    bgWordCount = wcPipe('browncorpus.txt')
    fgWordCount = wcPipe('dkos-entries.txt')

    def pcPipe(corpus):
        return ReadLines(corpus) \
            | Flatten(by=bigrams) \
            | Group(by=lambda x:x, reducingTo=ReduceToCount())

    bgPhraseCount = pcPipe('browncorpus.txt')
    fgPhraseCount = pcPipe('dkos-entries.txt')

    # collect data on each phrase

    phraseCount = \
        Join( Jin(fgPhraseCount, by=lambda(phrase,fC):phrase),
              Jin(bgPhraseCount, by=lambda(phrase,bC):phrase)) \
        | Map( by=lambda ((phrase1,fC),(phrase2,bC)):(phrase1,{'fC':fC,'bC':bC}))

    phraseStats1 = \
        Join( Jin(phraseCount, by=lambda((x,y),stats):x),
              Jin(fgWordCount, by=lambda(w,c):w)) \
        | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendStats(stats, 'fxC', c))) \
        | JoinTo( Jin(bgWordCount, by=lambda(w,c):w), by=lambda((x,y),stats):x) \
        | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendStats(stats, 'bxC', c)))
```

The full phrase-finding pipeline

```
phraseStats2 = \
    Join( Jin(phraseStats1, by=lambda((x,y),stats):y),
          Jin(fgWordCount, by=lambda(w,c):w)) \
    | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendStats(stats, 'fyC', c))) \
    | JoinTo( Jin(bgWordCount, by=lambda(w,c):w), by=lambda((x,y),stats):y) \
    | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendStats(stats, 'byC', c)))


# compute totals
numBGPhrases = Group(bgPhraseCount,
                     by=lambda(phrase,c):'const',
                     retaining=lambda(phrase,c):c,
                     reducingTo=ReduceToSum(),
                     combiningTo=ReduceToSum())
numFGPhrases = Group(fgPhraseCount,
                     by=lambda(phrase,c):'const',
                     retaining=lambda(phrase,c):c,
                     reducingTo=ReduceToSum(),
                     combiningTo=ReduceToSum())

phraseStats = \
    Join( Jin(numFGPhrases,by=lambda(dummy,fTot):'const'),
          Jin(phraseStats2,by=lambda(phrase,stats):'const')) \
    | Map( by=lambda((dummy,fTot),(phrase,stats)):
                (phrase,extendStats(stats,'fTot',fTot))) \
    | JoinTo( Jin(numBGPhrases,by=lambda(dummy,bTot):'const'),
              by=lambda(phrase,stats):'const') \
    | Map( by=lambda((phrase,stats),(dummy,bTot)):
                (phrase,extendStats(stats,'bTot',bTot)))

phraseResult = Map(phraseStats,
                   by=lambda(phrase,stats):
                       (phrase,infoness(stats),phraseness(stats)))

phraseScore = Format(phraseResult,
                     by=lambda(phrase,infoscore,phrasescore):
                         '%g\t%g\t%g\t%s' % (infoscore+phrasescore,infoscore,phrasescore,phrase))
```

Phrase Finding in PIG

Phrase Finding 1 - loading the input

```
wcohen@shell2:~/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
voreddy thu	1.0
voter registrations	2.0
voter suppression	3.0
wackyguy thu	1.0
waitingtoderail 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-phraseFreq-5/' AS (xy,c:int);
fgPhrases1 = LOAD 'phrases/data/dkos-phraseFreq-5/' AS (xy,c:int);
grunt> fgPhrases = FOREACH fgPhrases1 GENERATE STRSPLIT(xy,' ') AS xy:(x,y), c AS c;
grunt> fgPhrases1 = LOAD 'phrases/data/dkos-phraseFreq-5/' AS (xy,c:int);
fgPhrases1 = LOAD 'phrases/data/dkos-phraseFreq-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
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	{((friday, afternoon), 1)}
	afternoon	{((afternoon, service), 1), ((afternoon, mando), 1)}

bgWordFreq	w:bytearray	c:long
	friday	1
	afternoon	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	{((friday, afternoon), 1)}
	afternoon	{((afternoon, service), 1), ((afternoon, mando), 1)}

bgWordFreq	w:bytearray	c:long
	friday	1
	afternoon	2

phraseStats1	fgPhrases::xy:tuple(x:bytearray,y:bytearray)	fgPhrases::c:int	bgPhrases::xy:tuple(x:bytearray,y:bytearray)	bgPhras
	(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:byte
	friday	2	friday	1	(friday, afternoon)

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

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 - 
2014-04-01 16:59:38,370 [main] WARN  org.apache.pig.PigServer - 
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 - 
2014-04-01 16:59:42,024 [main] WARN  org.apache.pig.PigServer - 
grunt> phraseStats = CROSS phraseStats6,counts;
phraseStats = CROSS phraseStats6,counts;
2014-04-01 16:59:45,083 [main] WARN  org.apache.pig.PigServer - 
2014-04-01 16:59:45,083 [main] WARN  org.apache.pig.PigServer - 
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

How do we compute some complicated function?

With a “UDF”

```java
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);
    }

}
```

```
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 in PIG

```
-- 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.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';
```