Spark vs Hadoop
Spark

- 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

Set of concise dataflow operations ("transformation")

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

Sharded files are replaced by “RDDs” – resilient distributed datasets

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

```python
errors.cache()

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* is a *spark context* object
Spark examples

```scala
text_file = spark.textFile("hdfs://example.com/errors")
errors = text_file.filter(lamda line: "ERROR" in line)
# Count all the errors
errors.count()
# Count errors mentioning MySQL
errors.filter(lamda line: "MySQL" in line).count()
# Fetch the MySQL errors as an array of strings
errors.filter(lamda line: "MySQL" in line).collect()
```
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()**  # modify **errors** to be stored in *cluster memory*
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)

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.

subsequent actions will be much faster
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://...")
Spark examples: batch logistic regression

points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.randn(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

```python
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.randn(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.random(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

```python
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.randn(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
```
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.randn(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: w = \n"
Spark details: broadcast

points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.randn(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()

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

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

```python
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

```python
class WordProb(Planner):
    wc = ReadLines('corpus.txt') | Flatten(by=tokens) \n        | 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
• Cascading
  – Java library for map-reduce workflows
  – Also some library operations for common mappers/reducers
Scheme sourceScheme = new TextLine( new Fields( "line" ) );
Tap source = new Hfs( sourceScheme, inputPath );

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

Pipe assembly = new Pipe( "wordcount" );

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();
FlowConnector.setApplicationJarClass( properties, Main.class );

FlowConnector flowConnector = new FlowConnector( properties );
Flow flow = flowConnector.connect( "word-count", source, sink, assembly );
flow.complete();
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 \textit{successfully} hide the details from you?

Is this inefficient? We \textit{explicitly} form a group for each word, and then count the elements…?

We \textit{could} be saved by careful optimization: we know we don’t need the GroupBy intermediate result when we run the assembly….
• 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

```scala
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(

            ....

        )

        .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

Java API
Scala API

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

**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” – 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” – 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” – based on BLRT

- Define
  
  \[ p_i = \frac{k_i}{n_i}, \quad p = \frac{(k_1 + k_2)}{(n_1 + n_2)}, \]
  
  \[ L(p, k, n) = p^k(1-p)^{n-k} \]

\[ \phi_i(n_1, k_1, n_2, k_2) = 2 \log \frac{L(p_1, k_1, n_1)L(p_2, k_2, n_2)}{L(p, k_1, n_1)L(p, k_2, n_2)} \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k_1)</td>
<td>C((W_1=x \wedge W_2=y))</td>
<td>how often bigram (x y) occurs in corpus C</td>
</tr>
<tr>
<td>(n_1)</td>
<td>C((W_1=* \wedge W_2=*))</td>
<td>how many bigrams in corpus C</td>
</tr>
<tr>
<td>(k_2)</td>
<td>B((W_1=x^\wedge W_2=y))</td>
<td>how often (x y) occurs in <strong>background corpus</strong></td>
</tr>
<tr>
<td>(n_2)</td>
<td>B((W_1=<em>^\wedge W_2=</em>))</td>
<td>how many bigrams in background corpus</td>
</tr>
</tbody>
</table>

Does \(x y\) occur at the same frequency in both corpora?
“Phraseness” – based on BLRT

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

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

| \( k_1 \) | \( C(W_1=x, W_2=y) \) | how often bigram \( x y \) occurs in corpus \( C \) |
| \( n_1 \) | \( C(W_1=x) \) | how often word \( x \) occurs in corpus \( C \) |
| \( k_2 \) | \( C(W_1 \neq x, W_2=y) \) | 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.

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

\[ \begin{align*}
\log \left( \frac{p}{1 - p} \right) &= s \\
\iff p &= \frac{1}{1 + e^s}
\end{align*} \]

• Background corpus: 20 newsgroups dataset (20k messages, 7.4M words)
• Foreground corpus: rec.arts.movies.current-films June-Sep 2002 (4M words)
• Results?
message news
minority report
star wars
john harkness
derek janssen
robert frenchu
sean o’hara
box office
dawn taylor
anthony gaza
star trek
ancient race
scooby doo
austin powers
home.attbi.com hey
sixth sense
hey kids
gaza man
lee harrison
years ago
julia roberts
national guard
bourne identity
metro today www.zap2it.com
starweek magazine
eric chomko
wilner starweek
tim gueguen
jodie foster
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 $xy$ 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 $xy$ 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”

$$\delta_w(p \parallel q) \overset{\text{def}}{=} p(w) \log \frac{p(w)}{q(w)}$$
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”

\[ \delta_w(p \parallel q) \overset{\text{def}}{=} p(w) \log \frac{p(w)}{q(w)} \]

Phraseness: difference between bigram and unigram language model in foreground

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

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

“Pointwise KL divergence”

\[ \delta_w(p \parallel q) := p(w) \log \frac{p(w)}{q(w)} \]

Informativeness: difference between foreground and background models

\[ \delta_w(LM^N_{fg} \parallel LM^N_{bg}), \text{ or} \]
\[ \delta_w(LM^1_{fg} \parallel LM^1_{bg}) \]

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

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

“Pointwise KL divergence”

\[ \delta_w(p \parallel q) \overset{\text{def}}{=} p(w) \log \frac{p(w)}{q(w)} \]

Combined: difference between foreground bigram model and background unigram model

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

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

\[ \delta_w(LM_{fg}^N \parallel LM_{bg}^1) \]
# Pointwise KL, combined

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

```python
# 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')
```

background corpus
**Phrase Finding 2 – counting phrases**

```python
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 or y in STOPWORDS:
            yield (x,y)

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

bgPhraseCount = pcPipe('brown_corpus.txt')
fgPhraseCount = pcPipe('dkos-entries.txt')
```
def extendStats(stats, key, val):
    return dict(stats.items() + [(key, val)])

dictionary: {'statistic name': value}

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)) \n    | 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)) \n    | Map( by=lambda((phrase, stats), (w, c)): (phrase, extendStats(stats, 'fxC', c))) \n    | JoinTo( Jin(bgWordCount, by=lambda(w, c): w), by=lambda((x, y), stats): x) \n    | 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”
def extendStats(stats, key, val):
    return dict(stats.items() + [(key, val)])

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

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

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

Result:

- Democratic, 'Party', 1
- Democratic, 'Party.', 4
- Democratic, 'Party.', 1
- Democratic, 'Senators', 1
- Democratic, 'State', 1
- Democratic, 'U.S.', 1
- Democratic, 'Underground', 1
- Rice's, 'confirmation', 1
- Rice's, 'nomination.', 1
- Rico's, 'diary', 1
- Right's, 'Man', 1

('-const', 6743324)
Phrase Finding 4 – totals

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

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

(("Rice's", 'confirmation'), 1)
(("Rice's", 'nomination.'), 1)
(("Rico's", 'diary'), 1)
(("Right's", 'Man'), 1)
```
# 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())
# 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 = \n  Augment(phraseStats2,
            sideviews = [numFGPhrases, numFGPhrases],
            loadedBy = lambda nfg, nbg: (GPig.onlyRowOf(nfg), GPig.onlyRowOf(nbg))) \n      | 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)))
```
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, low 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,
ilo 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

```python
# 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*fxTot, 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( Join( 
        Join( fgPhraseCount, by=lambda(phrase,fC):phrase), 
        Join(bgPhraseCount, by=lambda(phrase,bC):phrase) ) \ 
        | Map( by=lambda ((phrase1,fC),(phrase2,bC)): (phrase1,{'fC':fC,'bC':bC}) ) )

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

```haskell
phraseStats2 = \
  Join( Join(phraseStats1, by=lambda((x,y),stats):y), \
        Join(fgWordCount, by=lambda((w,c),w)) \n        | Map( by=lambda((phrase,stats),(w,c)): (phrase,extendStats(stats, 'fyC', c))) \n        | JoinTo( Join(bgWordCount, by=lambda((w,c),w), by=lambda((x,y),stats):y) \n        | 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( Join(numFGPhrases, by=lambda((dummy, fTot),): 'const'), 
        Join(phraseStats2, by=lambda((phrase, stats),): 'const')) \n  | Map( by=lambda((dummy, fTot), (phrase, stats)): 
        (phrase,extendStats(stats, 'fTot', fTot))) \n  | JoinTo( Join(numBGPhrases, by=lambda((dummy, bTot),): 'const'), 
            by=lambda((phrase, stats),): 'const') \n  | Map( by=lambda((phrase, stats), (dummy, bTot)): 
        (phrase,extendStats(stats, 'bTot', bTot)))

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

phraseScore = Format(phraseResult, 
  by=lambda((phrase, infoscore, phrescore),): 
  ' %g\t%g\t%g\t%g\t\s' % (infoscore+phrescore,infoscore,phrescore,phrase))
```
Phrase Finding in PIG
Phrase Finding I - loading the input

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  28731 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
wackgyu 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
...
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);
grunt> 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> DESCRIBE fgPhrases;
grunt> ILLUSTRATE fgPhrases;
```

```
<table>
<thead>
<tr>
<th>fgPhrases1</th>
<th>xy:bytearray</th>
<th>c:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>patachon mon</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fgPhrases</th>
<th>xy:tuple(x:bytearray,y:bytearray)</th>
<th>c:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(patachon, mon)</td>
<td>1</td>
</tr>
</tbody>
</table>
```
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
-- compute word frequencies

```plaintext
fgWordFreq1 = GROUP fgPhrases1 BY xy.x;
fgWordFreq = FOREACH fgWordFreq1 GENERATE group as w, SUM(fgPhrases.c) as c;
```

<table>
<thead>
<tr>
<th>fgPhrases1</th>
<th>xy:bytearray</th>
<th>c:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>expressly gave</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>expressly reasserted</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fgPhrases</th>
<th>xy:tuple(x:bytearray,y:bytearray)</th>
<th>c:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(expressly, gave)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(expressly, reasserted)</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fgWordFreq1</th>
<th>group:bytearray</th>
<th>fgPhrases:bag{:tuple(xy:tuple(x:bytearray,y:bytearray),c:int)}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>expressly</td>
<td>{((expressly, gave), 1), ((expressly, reasserted), 1)}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fgWordFreq</th>
<th>w:bytearray</th>
<th>c:long</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>expressly</td>
<td>2</td>
</tr>
</tbody>
</table>
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
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;
```

```
<table>
<thead>
<tr>
<th>fgPhrases1</th>
<th>xy:bytearray</th>
<th>c:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>expressly gave</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>expressly reasserted</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>fgPhrases</th>
<th>xy:tuple(x:bytearray,y:bytearray)</th>
<th>c:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(expressly, gave)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(expressly, reasserted)</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>fgWordFreq1</th>
<th>group:bytearray</th>
<th>fgPhrases:bag{::tuple(xy:tuple(x:bytearray,y:bytearray),c:int)}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>expressly</td>
<td>{((expressly, gave), 1), ((expressly, reasserted), 1)}</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>fgWordFreq</th>
<th>w:bytearray</th>
<th>c:long</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>expressly</td>
<td>2</td>
</tr>
</tbody>
</table>
```
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;
fqWordFreq = 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 bxC;

phraseStats1: {fgPhrases::xy: (x: bytarray,y: bytarray),fgPhrases::c: int,
   bgPhrases::xy: (x: bytarray,y: bytarray),bgPhrases::c: int}
<table>
<thead>
<tr>
<th>bgWordFreq</th>
<th>group:bytearray</th>
<th>bgPhrases::beg::{tuple(xy:bytearray,y:bytearray),c:int}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>friday</td>
<td>(((friday, afternoon), 1))</td>
</tr>
<tr>
<td></td>
<td>afternoon</td>
<td>(((afternoon, service), 1), (afternoon, mando), 1))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bgWordFreq</th>
<th>w:bytearray</th>
<th>c:long</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>friday</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>afternoon</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phraseStats1</th>
<th>fgPhrases::xy:tuple(x:bytearray,y:bytearray)</th>
<th>fgPhrases::c:int</th>
<th>bgPhrases::xy:tuple(x:bytearray,y:bytearray)</th>
<th>bgPhrases::c:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(friday, afternoon)</td>
<td>1</td>
<td>(friday, afternoon)</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phraseStats2</th>
<th>xy:tuple(x:bytearray,y:bytearray)</th>
<th>fc:int</th>
<th>bc:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(friday, afternoon)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phraseStats3</th>
<th>fgWordFreq::w:bytearray</th>
<th>bgWordFreq::c:long</th>
<th>bgWordFreq::w:bytearray</th>
<th>bgWordFreq::c:long</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>friday</td>
<td>2</td>
<td>friday</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(friday, afternoon)</td>
<td>1</td>
<td>(friday, afternoon)</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phraseStats4</th>
<th>phraseStats2::xy:tuple(x:bytearray,y:bytearray)</th>
<th>phraseStats2::fc:int</th>
<th>phraseStats2::bc:int</th>
<th>fxC:long</th>
<th>bcX:long</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(friday, afternoon)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>phraseStats5</th>
<th>phraseStats4::phraseStats2::xy:tuple(x:bytearray,y:bytearray)</th>
<th>phraseStats4::phraseStats2::fc:int</th>
<th>phraseStats4::phraseStats2::bc:int</th>
<th>phraseStats4::fxC:long</th>
<th>phraseStats4::bcX:long</th>
<th>fxC:long</th>
<th>bcX:long</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(friday, afternoon)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bgWordFreq1</th>
<th>group:bytearray</th>
<th>bgPhrases::beg::{tuple(xy:tuple(x:bytearray,y:bytearray),c:int)}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>friday</td>
<td>(((friday, afternoon), 1))</td>
</tr>
<tr>
<td></td>
<td>afternoon</td>
<td>(((afternoon, service), 1), (afternoon, mando), 1))</td>
</tr>
</tbody>
</table>
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, ...
Phrase Finding 4 - adding total frequencies
83
How do we add the totals to the phraseStats relation?

```pig
grunt> counts1 = CROSS fgPhraseCount, bgPhraseCount;
grunt> 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;
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
```
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”
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.x, fgPhrases.c AS xc, 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,x,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,x,fC,bC,fxC,bxC,fgWordFreq.c as fxC,bgWordFreq.c as bxC;

-- 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,counds;

-- finally compute phraseness, etc

REGISTER ./pk1.jar;
phraseResult = FOREACH phraseStats GENERATE *,
    com.wcohen.SmoothedPKL(fC, fTot, bC, bTot, 1.0/bTot, 1.0) as infofness,
    com.wcohen.SmoothedPKL(fC, fTot, fxC*fxC, fTot*fTot, 1.0/fxC, 1.0) as phraseness;
STORE phraseResult INTO 'phrases/data/phraseResult';