Workflows 3: Graphs, PageRank, Loops, Spark

Always start off with a joke, to lighten the mood
The course so far

• Cost of operations
• Streaming learning algorithms
  – Parallel streaming with map-reduce
  – Building complex parallel algorithms with dataflow languages (Pig, GuineaPig, ...)
    • Mostly for text
• Another common large data structure: graphs
  • Examples: web, Facebook, Twitter, citation data, Freebase, Wikipedia, biological networks, ...
    – PageRank in dataflow languages
    – Iteration in dataflow
    – Spark
• Catchup: similarity joins
PageRank
Inlinks are “good” (recommendations)

Inlinks from a “good” site are better than inlinks from a “bad” site

but inlinks from sites with many outlinks are not a “good”...

“Good “ and “bad” are relative.
Imagine a “pagehopper” that always either

• follows a random link, or

• jumps to random page
Google’s PageRank

Imagine a “pagehopper” that always either

• follows a random link, or
• jumps to random page

PageRank ranks pages by the amount of time the pagehopper spends on a page:

• or, if there were many pagehoppers, PageRank is the expected “crowd size”
PageRank in Memory

- Let \( u = (1/N, ..., 1/N) \)
  - dimension = #nodes N
- Let \( A = \text{adjacency matrix: } [a_{ij} = 1 \iff i \text{ links to } j] \)
- Let \( W = [w_{ij} = a_{ij} / \text{outdegree}(i)] \)
  - \( w_{ij} \) is probability of jump from \( i \) to \( j \)
- Let \( v^0 = (1,1,\ldots,1) \)
  - or anything else you want
- Repeat until converged:
  - Let \( v^{t+1} = cu + (1-c)v^tW \)
    - \( c \) is probability of jumping “anywhere randomly”
Streaming PageRank
Streaming PageRank

• Assume we can store \( v \) but not \( W \) in memory

• Repeat until converged:
  
  \[
  \text{Let } v^{t+1} = cu + (1-c) v^t W
  \]

• \( W \) is a \textit{sparse row matrix}: each line is
  
  \( i \ j_{i,1},...,j_{i,d} \) [the neighbors of \( i \)]

• Store \( v' \) and \( v \) in memory: \( v' \) starts out as \( cu \)

• For each line “\( i \ j_{i,1},...,j_{i,d} \)“
  
  – For each \( j \) in \( j_{i,1},...,j_{i,d} \)
    
    \( v'[j] += (1-c) v[i]/d \)


Each iteration: from \( v \), compute \( v' \), then let \( v = v' \)

Everything needed for update is right there in row….
Streaming PageRank

• Assume we can store one row of $W$ in memory at a time, but not $\mathbf{v}$ – i.e., links on a web page fit in memory, not pageRank values for the whole web.

• Repeat until converged:
  – Let $\mathbf{v}^{t+1} = c \mathbf{u} + (1-c) \mathbf{v}^t \mathbf{W}$

• Store $\mathbf{W}$ as a row matrix: each line is
  – $i \ j_{i,1},...,j_{i,d}$ [the neighbors of $i$]

• $\mathbf{v}'$ starts out as $c \mathbf{u}$

• For each line “$i \ j_{i,1},...,j_{i,d}$”
  – For each $j$ in $j_{i,1},...,j_{i,d}$
    • $\mathbf{v}'[j] += (1-c) \mathbf{v}[i]/d$

Like in naïve Bayes: a document fits, but the model doesn’t

We need to convert these counter updates to messages, like we did for naïve Bayes
Streaming PageRank

• Assume we can store one row of $W$ in memory at a time, but not $v$ – i.e., links on a web page fit in memory, not pageRank for the whole web.
• Repeat until converged:
  – Let $v^{t+1} = cu + (1-c)v^tW$

• Store $W$ as a row matrix: each line is
  – $i \ j_{i,1},...,j_{i,d}$ [the neighbors of $i$]
• $v'$ starts out as $cu$
• For each line “$i \ j_{i,1},...,j_{i,d}$ “
  – For each $j$ in $j_{i,1},...,j_{i,d}$
    • $v'[j] += (1-c)v[i]/d$

Streaming: if we know $c$, $v[i]$ and have the linked-to $j$’s then
• we can compute $d$
• we can produce messages saying how much to increment each $v[j]$

Then we sort the messages by $v[j]$ and add up all the increments
• and somehow also add in $c/N$
Streaming PageRank

- Assume we can store one row of $W$ in memory at a time, but not a web page fit in memory, not whole web.

- Repeat until converged:
  - Let $v_{t+1} = cu + (1-c)v_tW$
  - Store $W$ as a row matrix: each line is
    - $i \quad j_{i,1}, \ldots, j_{i,d}$ [the neighbors of $i$]
  - $v'$ starts out as $cu$
  - For each line “$i \quad j_{i,1}, \ldots, j_{i,d}$“
    - For each $j$ in $j_{i,1}, \ldots, j_{i,d}$
      - $v'[j] += (1-c)v[i]/d$

Streaming: if we know $c$, $v[i]$ and have the linked-to $j$’s then
- we can compute $d$
- we can produce messages saying how much to increment each $v[j]$

Then we sort the messages by $v[j]$ and add up all the increments
- and somehow also add in $c/N$
PageRank in Dataflow Languages
Iteration in Dataflow

• PIG and Guinea Pig are *pure* data flow languages
  – no conditionals
  – no iteration

• To loop you need to embed a dataflow program into a ‘real’ program
```python
#!/usr/bin/python
from org.apache.pig.scripting import *

P = Pig.compile(""
""
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
" params: d, docs_in, docs_out 

pig script: PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn)) 

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

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

Iterate 10 times 

Pass parameters as a dictionary

Just run P, that was declared above

The output becomes the new input

15
```
Recap: a Gpig program....

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

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

# always subclass Planner
class WordCount(Planner):

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

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

There's no "program" object, and no obvious way to write a main that will call a program
Calling `GuineaPig` programmatically

Somewhere in the execution of the plan we will call this script with special arguments that tell it to do a substep of the plan.

So we need a Python `main` that will do that right.

But I also want to write an actual main program so…

Creatiing a Planner object I can call (not a subclass) – remember to call `setup()`

```python
def pageRankPlanner():
    p = Planner()
    p.x = ReadLines(...) | ...
    p.y = Map(p.x, by=....)
    ...
    p.setup()
    return p

if __name__ == "__main__":
    # run subplan step, if called recursively
    if Planner.partOfPlan(sys.argv):
        pageRankPlanner().main(sys.argv)
    else:
        # run your actual main here
        # for instance:
        planner = pageRankPlanner()
        v = planner.getView('wordcount')
        v.storagePlan().execute(planner)
```
Calling GuineaPig programmatically

```python
if __name__ == "__main__":
    #run subplan step, if called recursively
    if Planner.partOfPlan(sys.argv):
        pagerankPlanner().main(sys.argv)
    else:

        #create the initial graph w pagerank
        planner = pagerankPlanner()
        v0 = planner.getView('serializedInitRankedGraph')
        v0.storagePlan().execute(planner)
        print '>>> moving initial pageranks to',TMPFILE
        os.rename(v0.storedFile(), TMPFILE)

        for i in range(N):
            print '>>> pagerank iteration',i
            vi = planner.getView('serializedRankedGraph')
            vi.storagePlan().execute(planner)
            print '>>> moving round',i,'pageranks to',TMPFILE
            os.rename(vi.storedFile(), TMPFILE)
```
def pageRankPlanner():
    p = Planner()

    def serialize(graphView):
        return \\
        Format(graphView,
            by=lambda(url, pagerank, outlinks): '	'.join([url, '%g' % pagerank] + outlinks))

    # read in and create initial ranked graph
    p.edges = ReadLines(EDGEFILE) | Map(by=lambda line: line.strip().split(' '))
    p.initGraph = Group(p.edges, by=lambda(src, dst): src, retaining=lambda(src, dst): dst)
    p.initRankedGraph = Map(p.initGraph, by=lambda(url, outlinks): (url, 1.0, outlinks))
    p.serializedInitRankedGraph = serialize(p.initRankedGraph)

    # one step of the update, reading the last iteration from a temp file
    p.prevGraph = \\
        ReadLines(TMPPFILE) \\
        | Map(by=lambda line: line.strip().split('"t"')) \\
        | Map(by=lambda parts: (parts[0], float(parts[1]), parts[2:]))
    p.outboundPageRankMessages = \\
        FlatMap(p.prevGraph,
            by=lambda(url, pagerank, outlinks):
                map(lambda dst: (dst, pagerank / len(outlinks)), outlinks))
    p.newPageRank = \\
        Group(p.outboundPageRankMessages,
            by=lambda (dst, deltaPageRank): dst,
            retaining=lambda (dst, deltaPageRank): deltaPageRank,
            reducingTo=ReduceTo(lambda: (RESET, lambda accum, delta: accum + (1 - RESET) * delta))
    p.newRankedGraph = \\
        Join( Jin(p.prevGraph, by=lambda (url, pagerank, outlinks): url),
            Jin(p.newPageRank, by=lambda (dst, newPageRank): dst) \\
            | Map(by=lambda((url, oldPageRank, outlinks), (url, newPageRank)): (url, newPageRank, outlinks))
        p.serializedRankedGraph = serialize(p.newRankedGraph)

    p.setup()
    return p

lots and lots of i/o happening here...
```python
def pageRankPlanner():
    p = Planner()

def serialize(graphView):
    return \
    Format(graphView, \
        by=lambda(url, pagerank):
            pagerank)

# create the initial graph w pagerank
planner = pageRankPlanner()
v0 = planner.getView('serializedInitRankedGraph')
v0.storagePlan().execute(planner)
print '>>> moving initial pageranks to', TMPFILE
os.rename(v0.storedFile(), TMPFILE)

for i in range(N):
    print '>>> pagerank iteration', i
    vi = planner.getView('serializedRankedGraph')
    vi.storagePlan().execute(planner)
    print '>>> moving round', i, 'pageranks to', TMPFILE
    os.rename(vi.storedFile(), TMPFILE)

# one step of the update, reading the last iteration from a temp file
p.prevGraph = \[
    ReadLines(TMPFILE) \ 
    | Map(by=lambda line: line.split('t')) \ 
    | Map(by=lambda parts: (parts[0], float(parts[1]), parts[2:]))
]

p.outboundPageRankMessages = \ 
    FlatMap(p.prevGraph, \
        by=lambda (url, pagerank, outlinks):
            map(lambda dst: (dst, pagerank / len(outlinks)), outlinks))

p.newPageRank = \ 
    Group(p.outboundPageRankMessages, \
        by=lambda (dst, deltaPageRank): dst,
        retaining=lambda (dst, deltaPageRank): deltaPageRank,
        reducingTo=ReduceTo(lambda: (RESET), lambda accum, delta: accum + (1-RESET)*delta))

p.newRankedGraph = \ 
    Join( p.prevGraph, by=lambda (url, pagerank, outlinks): url),
        p.newPageRank, by=lambda (dst, newPageRank): dst) \ 
    | Map(by=lambda ((url, oldPageRank, outlinks), (url, newPageRank)): (url, newPageRank, outlinks))

p.serializedRankedGraph = serialize(p.newRankedGraph)

p.setup()
return p
```

Note: there is lots of I/O happening here...
previous_pagerank =
LOAD '$docs_in'
USING PigStorage('	')
AS ( url: chararray, pagerank: float, links:{ link: ( url: chararray ) } );

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

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

STORE new_pagerank
    INTO '$docs_out'
    USING PigStorage('	');

lots of i/o happening here…
An example from Ron Bekkerman of an iterative dataflow computation
Example: *k*-means clustering

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

![Clustered Data Points with Centroids](image)
Parallelizing $k$-means
Parallelizing $k$-means
Parallelizing $k$-means
**k-means on MapReduce**

Panda et al, Chapter 2

- Mappers read data portions and centroids
- Mappers **assign data instances** to clusters
- Mappers **compute new local centroids** and local cluster sizes
- Reducers **aggregate local centroids** (weighted by local cluster sizes) into new global centroids
- Reducers **write the new centroids**
**k-means in Apache Pig: input data**

• Assume we need to cluster documents
  – Stored in a 3-column table $D$:

<table>
<thead>
<tr>
<th>Document</th>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc1</td>
<td>Carnegie</td>
<td>2</td>
</tr>
<tr>
<td>doc1</td>
<td>Mellon</td>
<td>2</td>
</tr>
</tbody>
</table>

• Initial centroids are $k$ randomly chosen docs
  – Stored in table $C$ in the same format as above
**k-means in Apache Pig: E-step**

\[ D_C = \text{JOIN} \ C \text{ BY } w, D \text{ BY } w; \]

\[ \text{PROD} = \text{FOREACH} \ D_C \ \text{GENERATE} \ d, c, i_d \ast i_c \ \text{AS} \ i_di_c; \]

\[ \text{PROD}_g = \text{GROUP} \ \text{PROD} \ \text{BY} \ (d, c); \]

\[ c_d = \arg \max_c \ \sum_{w \in d} \left( i_d^w \cdot i_c^w \right) / \sqrt{\sum_{w \in c} \left( i_c^w \right)^2} \]

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

\[ \text{SQR}_g = \text{GROUP} \ \text{SQR} \ \text{BY} \ c; \]

\[ \text{DOT} = \text{FOREACH} \ \text{SQR}_g \ \text{GENERATE} \ c, \ \sqrt{\sum_{w \in c} \left( i_c^w \right)^2} \ \text{AS} \ \text{len}_c; \]

\[ \text{DOT}_g = \text{GROUP} \ \text{DOT} \ \text{BY} \ d, \ \text{PROD}_g \ \text{BY} \ d; \]

\[ \text{SIM} = \text{FOREACH} \ \text{DOT}_g \ \text{LEN} \ \text{GENERATE} \ d, c, dXc / \text{len}_c; \]

\[ \text{SIM}_g = \text{GROUP} \ \text{SIM} \ \text{BY} \ d; \]

\[ \text{CLUSTERS} = \text{FOREACH} \ \text{SIM}_g \ \text{GENERATE} \ \text{TOP}(1, 2, \text{SIM}); \]
**k-means in Apache Pig: E-step**

\[
D_C = \text{JOIN } C \text{ BY } w, \ D \text{ BY } w;
\]

\[
\text{PROD} = \text{FOREACH } D_C \text{ GENERATE } d, c, \ i_d \ast i_c \ \text{AS } i_di_c;
\]

\[
\text{PROD}_g = \text{GROUP PROD BY } (d, c);
\]

\[
\sum_{w \in d} i_w^d \cdot i_w^c
\]

\[
c_d = \arg \max_c \sqrt{\sum_{w \in C} (i_w^c)^2}
\]

\[
\text{DOT}_g = \text{FOREACH } \text{DOT}_\text{LEN} \text{ GENERATE } d, c, dXc / \text{len}_c;
\]

\[
\text{SIM} = \text{GROUP } \text{SIM BY } d;
\]

\[
\text{CLUSTERS} = \text{FOREACH } \text{SIM}_g \text{ GENERATE } \text{TOP}(1, 2, \text{SIM});
\]
**k-means in Apache Pig: E-step**

\[
D_C = \text{JOIN} \ C \ \text{BY} \ w, \ D \ \text{BY} \ w; \\
\text{PROD} = \text{FOREACH} \ D_C \ \text{GENERATE} \ d, \ c, \ i_d * i_c \ \text{AS} \ i_d i_c; \\
\text{PROD}_q = \text{GROUP} \ \text{PROD} \ \text{BY} \ (d, \ c); \\
\text{DOT}_q = \text{FOREACH} \ \text{PROD}_q \ \text{GENERATE} \ d, \ c, \ \sqrt{\sum_{w \in C} (i_c^w)^2} \ \text{AS} \ dXc; \\
\text{SQR} = \text{FOREACH} \ C \ \text{GENERATE} \ c, \ i_c * i_c \ \text{AS} \ i_c^2; \\
\text{SQR}_g = \text{GROUP} \ \text{SQR} \ \text{BY} \ c; \\
\text{LEN}_C = \text{FOREACH} \ \text{SQR}_g \ \text{GENERATE} \ c, \ \sqrt{\sum_{w \in C} (i_c^w)^2} \ \text{AS} \ len_c; \\
\text{DOT}_LLEN = \text{JOIN} \ \text{LEN}_C \ \text{BY} \ c, \ \text{DOT}_q \ \text{BY} \ c; \\
\text{SIM} = \text{FOREACH} \ \text{DOT}_LLEN \ \text{GENERATE} \ d, \ c, \ dXc / len_c; \\
\text{SIM}_g = \text{GROUP} \ \text{SIM} \ \text{BY} \ d; \\
\text{CLUSTERS} = \text{FOREACH} \ \text{SIM}_g \ \text{GENERATE} \ \text{TOP}(1, \ 2, \ \text{SIM});
\]
**k-means in Apache Pig: E-step**

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

\[ \text{PROD} = \text{FOREACH } D\_C \text{ GENERATE } d, c, i_d \ast i_c \text{ AS } i_di_c; \]

\[ \text{PROD}_g = \text{GROUP } \text{PROD} \text{ BY } (d, c); \]

\[ \text{DOT}_d = \text{arg max}_c \frac{\sum_{w \in d} i_d^w \cdot i_c^w}{\sqrt{\sum_{w \in c} (i_c^w)^2}}, \text{ SQR}_c; \]

\[ \text{SQR}_g = \text{GROUP } \text{SQR} \text{ BY } c; \]

\[ \text{DOT}_d \text{ = FOREACH } \text{DOT}_d \text{ GENERATE } d, c, dXc \text{ / len}_c; \]

\[ \text{SIM} = \text{FOREACH } \text{DOT}_d \text{ GENERATE } d, c, dXc \text{ / len}_c; \]

\[ \text{SIM}_g = \text{GROUP } \text{SIM} \text{ BY } d; \]

\[ \text{CLUSTERS} = \text{FOREACH } \text{SIM}_g \text{ GENERATE } \text{TOP}(1, 2, \text{SIM}); \]
**k-means in Apache Pig: E-step**

\[ D_C = \text{JOIN} C \text{ BY } w, D \text{ BY } w; \]
\[ \text{PROD} = \text{FOREACH } D_C \text{ GENERATE } d, c, i_d * i_c \text{ AS } i_d i_c; \]
\[ \text{PROD}_q = \text{GROUP} \text{ PROD} \text{ BY } (d, c); \]
\[ \text{DOT}_\text{PROD}_q = \text{FOREACH} \text{ PROD}_q \text{ GENERATE } d, c, \sum_{w \in d} i_d^w \cdot i_c^w \text{ AS } d xc; \]
\[ \text{SQR}_C = \text{FOREACH} C \text{ GENERATE } c, i_c^2 \text{ AS } i_c^2; \]
\[ \text{SQR}_g = \text{GROUP} \text{ SQR} \text{ BY } c; \]
\[ \text{LEN}_C = \text{FOREACH} \text{ SQR}_g \text{ GENERATE } c, \sqrt{\sum_{w \in c} (i_c^w)^2} \text{ AS } \text{len}_c; \]
\[ \text{DOT}_\text{LEN} = \text{JOIN} \text{ LEN}_C \text{ BY } c, \text{DOT}_\text{PROD}_q \text{ BY } c; \]
\[ \text{SIM} = \text{FOREACH} \text{ DOT}_\text{LEN} \text{ GENERATE } d, c, \frac{d xc}{\text{len}_c}; \]
\[ \text{SIM}_g = \text{GROUP} \text{ SIM} \text{ BY } d; \]
\[ \text{CLUSTERS} = \text{FOREACH} \text{ SIM}_g \text{ GENERATE } \text{TOP}(1, 2, \text{SIM}); \]
**k-means in Apache Pig: E-step**

\[ D_{\cdot C} = \text{JOIN } D \text{ BY } w, D \text{ BY } w; \]

\[ \text{PROD} = \text{FOREACH } D_{\cdot C} \text{ GENERATE } d, c, i_d * i_c \text{ AS } i_di_c; \]

\[ \text{PROD}_g = \text{GROUP } \text{PROD} \text{ BY } (d, c); \]

\[ \text{DOT}_\text{PROD} = \text{FOREACH } \text{PROD}_g \text{ GENERATE } d, c, \text{SUM}(i_d i_c) \text{ AS } dXc; \]

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

\[ \text{SQR}_g = \text{GROUP } \text{SQR} \text{ BY } c; \]

\[ \text{LEN}_\text{C} = \text{FOREACH } \text{SQR}_g \text{ GENERATE } c, \text{SQRT(}\text{SUM}(i_c^2)) \text{ AS } len_c; \]

\[ \text{DOT}_\text{LEN} = \text{JOIN } \text{LEN}_\text{C} \text{ BY } c, \text{DOT}_\text{PROD} \text{ BY } c; \]

\[ \text{SIM} = \text{FOREACH } \text{DOT}_\text{LEN} \text{ GENERATE } d, c, dXc / len_c; \]

\[ \text{SIM}_g = \text{GROUP } \text{SIM} \text{ BY } d; \]

\[ \text{CLUSTERS} = \text{FOREACH } \text{SIM}_g \text{ GENERATE } \text{TOP}(1, 2, \text{SIM}); \]

# documentation d, cluster c similarities

# cluster assignments: d \( \rightarrow \) closest c
**k-means in Apache Pig: M-step**

\[
D_C_W = \text{JOIN} CLUSTERS \text{ BY } d, D \text{ BY } d;
\]

\[
D_C_W_g = \text{GROUP} D_C_W \text{ BY } (c, w);
\]

\[
SUMS = \text{FOREACH } D_C_W_g \text{ GENERATE } c, w, \text{SUM}(i_d) \text{ AS } sum;
\quad \# \text{ add up documents in each cluster}
\]

\[
D_C_W_{gg} = \text{GROUP} D_C_W \text{ BY } c;
\]

\[
SIZES = \text{FOREACH } D_C_W_{gg} \text{ GENERATE } c, \text{COUNT}(D_C_W) \text{ AS } size;
\quad \# \text{ figure out cluster size}
\]

\[
SUMS_SIZES = \text{JOIN} SIZES \text{ BY } c, SUMS \text{ BY } c;
\]

\[
C = \text{FOREACH } SUMS_SIZES \text{ GENERATE } c, w, \text{sum} / \text{size} \text{ AS } i_c ;
\quad \# \text{ figure out weighted average of docs in each cluster}
\]

Finally - embed in Java (or Python or ….) to do the looping
Data is read, and model is written, with every iteration

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

Panda et al, Chapter 2
Spark: Another Dataflow Language
Spark

- Hadoop: Too much typing
  - programs are not concise
- Hadoop: Too low level
  - missing abstractions
  - hard to specify a workflow
- Pig, Guinea Pig, ... address these problems: also Spark
- Not well suited to iterative operations
  - E.g., PageRank, E/M, k-means clustering, ...
  - Spark lowers cost of repeated reads

Spark: Sharded files are replaced by “RDDs” – resilient distributed datasets.
RDDs can be cached in cluster memory and recreated to recover from error
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()
`errors` is a transformation, and thus a data structure that explains how to do something.

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

Everything is sharded, like in Hadoop and GuineaPig.

```python
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

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
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.rand(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.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.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
```

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.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

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 SQL structured data
Spark Streaming real-time
MLib machine learning
GraphX graph processing

Spark Core

Standalone Scheduler
YARN
Mesos