Other Map-Reduce (ish) Frameworks

William Cohen
Outline

• More concise languages for map-reduce pipelines
• Abstractions built on top of map-reduce
  – General comments
  – Specific systems
    • Cascading, Pipes
    • PIG, Hive
    • Spark, Flink
Y:Y=Hadoop+X or Hadoop~=Y

• What else are people using?
  – instead of Hadoop
  – on top of Hadoop
Issues with Hadoop

• Too much typing
  – programs are not concise
• Too low level
  – missing abstractions
  – hard to specify a workflow
• Not well suited to iterative operations
  – E.g., E/M, k-means clustering, ...
  – Workflow and memory-loading issues
STREAMING AND MRJOB: MORE CONCISE MAP-REDUCE PIPELINES
Hadoop streaming

• start with stream & sort pipeline

\texttt{cat data | mapper.py | sort \_\_k1,1 | reducer.py}

• run with hadoop streaming instead

\texttt{bin/hadoop jar contrib/streaming/hadoop-\*streaming\*.jar}
  \texttt{-file mapper.py \_\_file reducer.py}
  \texttt{-mapper mapper.py}
  \texttt{-reducer reducer.py}
  \texttt{-input /hdfs/path/to/inputDir}
  \texttt{-output /hdfs/path/to/outputDir}
  \texttt{-mapred.map.tasks=20}
  \texttt{-mapred.reduce.tasks=20}
mrjob word count

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

```python
from mrjob.job import MRJob
import re

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

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

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

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

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

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

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

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

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

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

if __name__ == '__main__':
    MRMostUsedWord.run()
MAP-REDUCE ABSTRACTIONS: CASCADING, PIPES, SCALDING
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 );

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 successfully hide the details from you?

```java
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();
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.
More on Pig

• Pig Latin
  – atomic types + compound types like tuple, bag, map
  – execute locally/interactively or on hadoop
• can embed Pig in Java (and Python and …)
• can call out to Java from Pig
• Similar (ish) system from Microsoft: DryadLinq
A = load '/tmp/bible+shakes.nopunc';
B = foreach A generate flatten(TOKENIZE((chararray)$0)) as word;
C = filter B by word matches '\w+';
D = group C by word;
E = foreach D generate COUNT(C) as count, group as word;
F = order E by count desc;
store F into '/tmp/wc';

- **Tokenize** – built-in function
- **Flatten** – special keyword, which applies to the next step in the process – ie foreach is transformed from a MAP to a FLATMAP
PIG parses and **optimizes** a sequence of commands before it executes them. It's smart enough to turn GROUP … FOREACH… SUM … into a map-reduce.

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

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

First: an iterative algorithm in Pig Latin
How to use loops, conditionals, etc?

Embed PIG in a real programming language.

Julien Le Dem - Yahoo
#!/usr/bin/python
from org.apache.pig.scripting import *

P = Pig.compile(""
    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("rmr " + out)
    stats = P.bind(params).runSingle()
    if not stats.isSuccessful():
        raise 'failed'
    params["docs_in"] = out
An example from Ron Bekkerman
**Example: k-means clustering**

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

![Diagram of k-means clustering with centroids marked]
Parallelizing $k$-means
Parallelizing \( k \)-means
Parallelizing $k$-means
**k-means on MapReduce**

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

- Assume we need to cluster documents
  - Stored in a 3-column table $D$:
    - Initial centroids are $k$ randomly chosen docs
      - Stored in table $C$ in the same format as above

<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>
**k-means in Apache Pig: E-step**

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

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

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

\[ \text{DOT} = \text{FOREACH} \ \text{PROD} \_a \text{ GENERATE} \ d, \ c, \ \sum_{w \in a} i_d^w \cdot i_c^w \text{ AS } dXc; \]

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

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

\[ \text{LEN} = \text{FOREACH} \ \text{SQR} \_g \text{ GENERATE} \ c, \ \sqrt{\sum_{w \in C} (i_c^w)^2} \text{ AS } len_c; \]

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

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

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

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

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

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

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

\[ \text{DOT} = \text{FOREACH} \text{ PROD}_g \text{ GENERATE } d, c, \sum \sum_{w} i^w_d i^w_c \]

\[ c_d = \arg \max_c \frac{\sum_{w} i^w_d \cdot i^w_c}{\sqrt{\sum_{w} (i^w_c)^2}} \]

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

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

\[ \text{LEN}_C = \text{FOREACH} \text{ SQR}_g \text{ GENERATE } c, \sqrt{\sum_{w} (i^w_c)^2} \text{ AS } \text{len}_c; \]

\[ \text{DOT}_C = \text{JOIN} \text{ LEN}_C \text{ BY } c, \text{DOT}_C \text{ BY } c; \]

\[ \text{SIM} = \text{FOREACH} \text{ DOT}_C \text{ GENERATE } d, c, \frac{d \cdot c}{\text{len}_c}; \]

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

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

\[ D_\_C = \text{JOIN } C \text{ BY } w, \text{ D BY } w; \]
\[ \text{PROD} = \text{FOREACH } D_\_C \text{ GENERATE } d, c, i_d * i_c \text{ AS } i_d i_c; \]

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

\[ \text{DOT} = \text{FOREACH } \text{PROD}_a \text{ GENERATE } d, c, \sum i_d i_c; \]

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

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

\[ \text{LEN} = \text{FOREACH } \text{SQR}_g \text{ GENERATE } c, \sqrt{\sum (i_c^2)} \text{ AS } \text{len}_c; \]

\[ \text{DOT}_c = \text{JOIN } \text{LEN} \text{ BY } c, \text{PROD}_a \text{ BY } c; \]

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

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

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

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

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

\[
C_d = \text{arg max}_c \left( \frac{\sum_{w \in d} \sum_{w \in c} i_d^w \cdot i_c^w}{\sqrt{\sum_{w \in c} (i_c^w)^2}} \right)
\]

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

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

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

\[
\text{DOT}_c = \text{FOREACH } \text{LEN}_c \text{ GENERATE } c, \text{len}_c;
\]

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

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

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

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

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

\[
\text{DOT}\_\text{PROD} = \text{FOREACH} \ \text{PROD}_g \text{ GENERATE} \ d, \ c, \ \sum_{w \in d} i_d^w \cdot i_c^w \text{ AS} \ dXc;
\]

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

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

\[
\text{LEN}_C = \text{FOREACH} \ \text{SQR}_g \text{ GENERATE} \ c, \ \sqrt{\sum_{w \in c} (i_c^w)^2} \text{ AS} \ len_c;
\]

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

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

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

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

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

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

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

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

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

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

\[
\text{LEN}_C = \text{FOREACH } \text{SQR}_g \text{ GENERATE } c, \text{SQRT}(\text{SUM}(i_{c}^2)) \text{ AS } \text{len}_c;
\]

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

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

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

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

```
D_C_W = JOIN CLUSTERS BY d, D BY d;

D_C_W_g = GROUP D_C_W BY (c, w);
SUMS = FOREACH D_C_W_g GENERATE c, w, SUM(i_d) AS sum;

D_C_W_gg = GROUP D_C_W BY c;
SIZES = FOREACH D_C_W_gg GENERATE c, COUNT(D_C_W) AS size;

SUMS_SIZES = JOIN SIZES BY c, SUMS BY c;
C = FOREACH SUMS_SIZES GENERATE c, w, sum / size AS i_c;
```

Finally - embed in Java (or Python or ....) to do the looping
The problem with k-means in Hadoop

I/O costs
Data is read, and model is written, with every iteration

- 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
SCHEMES DESIGNED FOR ITERATIVE HADOOP PROGRAMS: SPARK AND FLINK
Spark word count example

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

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

- Familiar-looking API for abstract operations (map, flatMap, reduceByKey, …)
- Most API calls are “lazy” – ie, `counts` is a data structure defining a pipeline, not a materialized table.
- Includes ability to store a sharded dataset in cluster memory as an RDD (resilient distributed database)
```scala
val points = spark.textFile(...).map(parsePoint).cache()
val w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
    val gradient = points.map(p =>
        (1 / (1 + exp(-p.y*(w dot p.x)))) - 1) * p.y * p.x
    .reduce(_ + _)
    w -= gradient
}
println("Final separating plane: " + w)
```

Note that w gets shipped automatically to the cluster with every map call.
Spark logistic regression example

• Allows caching data in memory

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

Note that w gets shipped automatically to the cluster with every map call.
Spark logistic regression example

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

- Recent Apache Project – 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

• Like Spark, in-memory or on disk
• Everything is a Java object
• Unlike Spark, contains operations for iteration
  – Allowing query optimization
• Very easy to use and install in local model
  – Very modular
  – Only needs Java
MORE EXAMPLES IN PIG
Phrase Finding in 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
-rwrxr-xr-x 3 wcohen supergroup 28857 2014-03-14 14:00 /user/wcohen/phrases/data/dkos-phraseFreq-5/part-00000
-rwrxr-xr-x 3 wcohen supergroup 28210 2014-03-14 14:00 /user/wcohen/phrases/data/dkos-phraseFreq-5/part-00001
-rwrxr-xr-x 3 wcohen supergroup 29731 2014-03-14 14:00 /user/wcohen/phrases/data/dkos-phraseFreq-5/part-00002
-rwrxr-xr-x 3 wcohen supergroup 27422 2014-03-14 14:00 /user/wcohen/phrases/data/dkos-phraseFreq-5/part-00003
-rwrxr-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
volvoldrivingliberal 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

...
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> 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;

DESCRIBE fgPhrases;
fgPhrases: {xy: (x: bytearray,y: bytearray),c: int}

 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 1 - word counts
```sql
-- compute word frequencies

fgWordFreq1 = GROUP fgPhrases BY xy.x;
fgWordFreq1 = GROUP fgPhrases BY xy.x;

<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(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.

```plaintext
-- compute word frequencies

fgWordFreq1 = GROUP fgPhrases BY xy.x;
fqWordFreq = 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.

```scala
-- 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 fxC,bgWordFreq::c as bxC;

phraseStats1: {fgPhrases::xy: (x: bytearray,y: bytearray),fgPhrases::c: int,
    bgPhrases::xy: (x: bytearray,y: bytearray),bgPhrases::c: int}
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
fgPhraseCount = group fgPhrases1 ALL;
fgPhraseCount = group fgPhrases1 ALL;

bgPhraseCount1 = group bgPhrases1 ALL;
bgPhraseCount1 = group bgPhrases1 ALL;

fgPhraseCount = FOREACH fgPhraseCount1 GENERATE SUM(fgPhrases1.c);
fgPhraseCount = FOREACH fgPhraseCount1 GENERATE SUM(fgPhrases1.c);

bgPhraseCount = FOREACH bgPhraseCount1 GENERATE SUM(bgPhrases1.c);
bgPhraseCount = FOREACH bgPhraseCount1 GENERATE SUM(bgPhrases1.c);

<table>
<thead>
<tr>
<th>bgPhrases1</th>
<th>xy:bytearray</th>
<th>c:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>continuing series</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>neighboring lower</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bgPhraseCount1</th>
<th>group:chararray</th>
<th>bgPhrases1:bag{tuple(xy:bytearray,c:int)}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>all</td>
<td>{{(continuing series, 1), (neighboring lower, 1)}}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bgPhraseCount</th>
<th>:long</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
How do we add the totals to the phraseStats relation?

grunt> counts1 = CROSS fgPhraseCount,bgPhraseCount;
counts1 = CROSS fgPhraseCount,bgPhraseCount;
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
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,fxC,byC,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*fxC, fTot*fTot, 1.0/fxC, 1.0) as phraseness;

STORE phraseResult INTO 'phrases/data/phraseResult';
**PIG Features**

- **LOAD** `‘hdfs-path’ AS (schema)`
  - `schemas can include int, double, bag, map, tuple, ...`
- **FOREACH** `alias GENERATE ... AS ..., ...`
  - `transforms each row of a relation`
- **DESCRIBE** `alias/ILLUSTRATE alias -- debugging`
- **GROUP** `alias BY ...`
- **FOREACH** `alias GENERATE group, SUM(....)`
  - `GROUP/GENERATE ... aggregate op together act like a map-reduce`
- **JOIN** `r BY field, s BY field, ...`
  - `inner join to produce rows: r::f1, r::f2, ... s::f1, s::f2, ...`
- **CROSS** `r, s, ...`
  - `use with care unless all but one of the relations are singleton`
- User defined functions as operators
  - `also for loading, aggregates, ...`
The full phrase-finding pipeline
-- load data
fgPhrases1 = LOAD 'phrases/data/dkos-phraseFreq-5/' AS (xy,c:int);
fgPhrases = FOREACH fgPhrases1 GENERATE STRSPLIT(xy,' ') AS xy:(x,y), c AS c;
bgPhrases1 = LOAD 'phrases/data/brown-phraseFreq-5/' AS (xy,c:int);
bgPhrases = FOREACH bgPhrases1 GENERATE STRSPLIT(xy,' ') AS xy:(x,y), c AS c;

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

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

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

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

-- finally compute phraseness, etc
REGISTER ./pkl.jar;
phraseResult = FOREACH phraseStats GENERATE *,
             com.wcohen.SmoothedPKL(fC, fTot, bC, bTot, 1.0/bTot, 1.0) as infoness,
             com.wcohen.SmoothedPKL(fC, fTot, fxC*fxC, fTot*fTot, 1.0/fxC, 1.0) as phraseness;
STORE phraseResult INTO 'phrases/data/phraseResult';
GUINEA PIG
GuineaPig: PIG in Python

• Pure Python (< 1500 lines)
• Streams Python data structures
  – strings, numbers, tuples (a,b), lists [a,b,c]
  – No records: operations defined functionally
• Compiles to Hadoop streaming pipeline
  – Optimizes sequences of MAPs
• Runs locally without Hadoop
  – compiles to stream-and-sort pipeline
  – intermediate results can be viewed
• Can easily run parts of a pipeline
• http://curtis.ml.cmu.edu/w/courses/index.php/Guinea_Pig
GuineaPig: PIG in Python

- Pure Python, streams Python data structures
  - not too much new to learn (eg field/record notation, special string operations, UDFs, ...)
  - codebase is small and readable
- Compiles to Hadoop or stream-and-sort, can easily run parts of a pipeline
  - intermediate results often are (and always can be) stored and inspected
  - plan is fairly visible
- Syntax includes high-level operations but also fairly detailed description of an optimized map-reduce step
  - Flatten | Group(by=..., retaining=..., reducingTo=...)
A wordcount example

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

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

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

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

wordCount = Group(words, by=lambda x: x, reducingTo=ReduceToCount())

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

class variables in the planner are data structures
Wordcount example ....

- Data structure can be converted to a series of "abstract map-reduce tasks"

```python
python longer-wordcount.py --view=wordCount --do=doGroupMap < corpus.txt \  | LC_COLLATE=C sort -k1 \  | python longer-wordcount.py --view=wordCount --do=doStoreRows \ > gpiig_views/wordCount.gp
```
More examples of GuineaPig

Join syntax, macros, Format command

class WordCmp(Planner):
    def wcPipe(fileName):
        return ReadLines(fileName) | Flatten(by=tokens) | Group(by=lambda x:x, reducingTo=

wc1 = wcPipe('bluecorpus.txt')
wc2 = wcPipe('redcorpus.txt')

cmp = Join( Jin(wc1, by=lambda(word,n):word), Jin(wc2, by=lambda(word,n):word) ) \ 
    | ReplaceEach(by=lambda((word1,n1),(word2,n2)): (word1, score(n1,n2)))
result = Format(cmp, by=lambda(word,blueScore):'%.4f %s' % (blueScore,word))

Incremental debugging, when intermediate views are stored:

  % python wrdcmp.py –store result
  ...
  % python wrdcmp.py –store result –reuse cmp
More examples of GuineaPig

**Full Syntax for Group**

\[
\text{Group}(wc, \ \text{by} = \lambda (word, count): word[:k], \\
\text{retaining} = \lambda (word, count): count, \\
\text{reducingTo} = \text{ReduceToSum()})
\]

\text{equiv to:}

\[
\text{Group}(wc, \ \text{by} = \lambda (word, count): word[:k], \\
\text{reducingTo} = \\
\ \ \text{ReduceTo}(\text{int}, \\
\ \ \ \ \lambda \ accum, word, count): accum + count))
\]