Scaling Up

Pig

Duen Horng (Polo) Chau
Assistant Professor
Associate Director, MS Analytics
Georgia Tech

Partly based on materials by
Professors Guy Lebanon, Jeffrey Heer, John Stasko, Christos Faloutsos, Parishit Ram (GT PhD alum; SkyTree), Alex Gray
Pig

High-level language

• instead of writing low-level map and reduce functions

Easy to program, understand and maintain

Created at Yahoo!

Produces sequences of Map-Reduce programs

(Lets you do “joins” much more easily)

http://pig.apache.org
Your data analysis task -> a **data flow** sequence

- **Data flow** sequence  
  = sequence of **data transformations**

Input -> **data flow** -> output

You specify the **data flow** in **Pig Latin** (Pig’s language)

- Pig turns the data flow into a sequence of MapReduce jobs automatically!
Pig: 1st Benefit

Write only a few lines of Pig Latin

Typically, MapReduce development cycle is long

- Write mappers and reducers
- Compile code
- Submit jobs
- ...
Pig: 2nd Benefit

Pig can perform a **sample run** on representative subset of your input data automatically!

Helps debug your code in smaller scale (**much faster!**), before applying on full data
What Pig is good for?

Batch processing, since it’s built on top of MapReduce

- Not for random query/read/write

May be slower than MapReduce programs coded from scratch

- You trade ease of use + coding time for some execution speed
How to run Pig

Pig is a client-side application
(run on your computer)

Nothing to install on Hadoop cluster
How to run Pig: 2 modes

Local Mode

• Run on your computer
• Great for trying out Pig on small datasets

MapReduce Mode

• Pig translates your commands into MapReduce jobs
• Remember you can have a single-machine cluster set up on your computer
Pig program: 3 ways to write

Script

**Grunt** (interactive shell)

- Great for debugging

Embedded (into Java program)

- Use PigServer class (like JDBC for SQL)
- Use PigRunner to access Grunt
Grunt (interactive shell)

Provides **code completion**

Press **Tab** key to complete Pig Latin keywords and functions

Let’s see an example Pig program run with Grunt

- Find highest temperature by year
Example Pig program

Find highest temperature by year

records = LOAD 'input/ncdc/micro-tab/sample.txt' AS (year:chararray, temperature:int, quality:int);

filtered_records =
  FILTER records BY temperature != 9999
  AND (quality == 0 OR quality == 1 OR
    quality == 4 OR quality == 5 OR
    quality == 9);

grouped_records = GROUP filtered_records BY year;

max_temp = FOREACH grouped_records GENERATE group, MAX(filtered_records.temperature);

DUMP max_temp;
Example Pig program

Find highest temperature by year

grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
    AS (year: chararray, temperature: int, quality: int);

grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
called a “tuple”

grunt> DESCRIBE records;
records: {year: chararray, temperature: int, quality: int}
Example Pig program

Find highest temperature by year

grunt>
filtered_records = 
   \texttt{FILTER} \texttt{records BY} temperature \neq 9999 \\
   \texttt{AND} (quality == 0 \texttt{OR} quality == 1 \texttt{OR} \\
   quality == 4 \texttt{OR} quality == 5 \texttt{OR} \\
   quality == 9);

grunt> \texttt{DUMP} filtered_records;

(1950,0,1) \\
(1950,22,1) \\
(1950,-11,1) \\
(1949,111,1) \\
(1949,78,1) \\

\textbf{In this example, no tuple is filtered out}
Example Pig program

Find highest temperature by year

grunt> grouped_records = GROUP filtered_records BY year;

grunt> DUMP grouped_records;

(1949, {(1949, 111, 1), (1949, 78, 1)})
(1950, {(1950, 0, 1), (1950, 22, 1), (1950, -11, 1)})

called a “bag”
= unordered collection of tuples

grunt> DESCRIBE grouped_records;

grouped_records: {group: chararray, filtered_records: {year: chararray, temperature: int, quality: int}}

alias that Pig created
Example Pig program

Find highest temperature by year

```pig
grunt> max_temp = FOREACH grouped_records GENERATE group, MAX(filtered_records.temperature);

grunt> DUMP max_temp;

(1949,111)
(1950,22)
```

<table>
<thead>
<tr>
<th>grouped_records: {group: chararray, filtered_records: {year: chararray, temperature: int, quality: int}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1949,{(1949,111,1), (1949,78,1)})</td>
</tr>
<tr>
<td>(1950,{(1950,0,1),(1950,22,1),(1950,-11,1)})</td>
</tr>
</tbody>
</table>
Run Pig program on a subset of your data

You saw an example run on a tiny dataset

How to do that for a larger dataset?

• Use the **ILLUSTRATE** command to generate sample dataset
Run Pig program on a subset of your data

```
grunt> ILLUSTRATE max_temp;
```

<table>
<thead>
<tr>
<th>records</th>
<th>year:chararray</th>
<th>temperature:int</th>
<th>quality:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1949</td>
<td>78</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1949</td>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1949</td>
<td>9999</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>filtered_records</th>
<th>year:chararray</th>
<th>temperature:int</th>
<th>quality:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1949</td>
<td>78</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1949</td>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>

| grouped_records | group:chararray | filtered_records:bag{tuple(year:chararray, |
|-----------------|-----------------|-----------------|------------|
|                 | 1949            | {(1949, 78, 1), (1949, 111, 1)} |

<table>
<thead>
<tr>
<th>max_temp</th>
<th>group:chararray</th>
<th>:int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1949</td>
<td>111</td>
</tr>
</tbody>
</table>
How does Pig compare to SQL?

SQL: “fixed” schema

PIG: loosely defined schema, as in

```java
records = LOAD 'input/ncdc/micro-tab/sample.txt'
    AS (year:chararray, temperature:int, quality:int);
```
How does Pig compare to SQL?

**SQL**: supports fast, random access (e.g., <10ms)

**PIG**: batch processing
Much more to learn about Pig

Relational Operators, Diagnostic Operators (e.g., describe, explain, illustrate), utility commands (cat, cd, kill, exec), etc.

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loading and storing</td>
<td>LOAD</td>
<td>Loads data from the filesystem or other storage into a relation</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td>Saves a relation to the filesystem or other storage</td>
</tr>
<tr>
<td></td>
<td>DUMP</td>
<td>Prints a relation to the console</td>
</tr>
<tr>
<td>Filtering</td>
<td>FILTER</td>
<td>Removes unwanted rows from a relation</td>
</tr>
<tr>
<td></td>
<td>DISTINCT</td>
<td>Removes duplicate rows from a relation</td>
</tr>
<tr>
<td></td>
<td>FOREACH...GENERATE</td>
<td>Adds or removes fields from a relation</td>
</tr>
<tr>
<td></td>
<td>MAPREDUCE</td>
<td>Runs a MapReduce job using a relation as input</td>
</tr>
<tr>
<td></td>
<td>STREAM</td>
<td>Transforms a relation using an external program</td>
</tr>
<tr>
<td></td>
<td>SAMPLE</td>
<td>Selects a random sample of a relation</td>
</tr>
<tr>
<td>Grouping and joining</td>
<td>JOIN</td>
<td>Joins two or more relations</td>
</tr>
<tr>
<td></td>
<td>COGROUP</td>
<td>Groups the data in two or more relations</td>
</tr>
<tr>
<td></td>
<td>GROUP</td>
<td>Groups the data in a single relation</td>
</tr>
<tr>
<td></td>
<td>CROSS</td>
<td>Creates the cross-product of two or more relations</td>
</tr>
<tr>
<td>Sorting</td>
<td>ORDER</td>
<td>Sorts a relation by one or more fields</td>
</tr>
<tr>
<td></td>
<td>LIMIT</td>
<td>Limits the size of a relation to a maximum number of tuples</td>
</tr>
<tr>
<td>Combining and splitting</td>
<td>UNION</td>
<td>Combines two or more relations into one</td>
</tr>
<tr>
<td></td>
<td>SPLIT</td>
<td>Splits a relation into two or more relations</td>
</tr>
</tbody>
</table>