Lecture 26b: Supplementary slides for Pig Latin

Friday, Dec 3, 2010
Outline

Based entirely on *Pig Latin: A not-so-foreign language for data processing*, by Olston, Reed, Srivastava, Kumar, and Tomkins, 2008

Quiz section tomorrow: in CSE 403 (this is CSE, don’t go to EE1)
Why?

- Map-reduce is a low-level programming environment
- In most applications need more complex queries
- Pig-latin accepts higher level queries, translates them to sequences of map-reduce
Pig-Latin Overview

• Data model = loosely typed *nested relations*
• Query model = a sql-like, dataflow language

• Execution model:
  – Option 1: run locally on your machine
  – Option 2: compile into sequence of map/reduce, run on a cluster supporting Hadoop

• Main idea: use Opt1 to debug, Opt2 to execute
Example

• Input: a table of urls:
  (url, category, pagerank)

• Compute the average pagerank of all sufficiently high pageranks, for each category

• Return the answers only for categories with sufficiently many such pages
First in SQL…

```
SELECT category, AVG(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP By category
HAVING COUNT(*) > 10^6
```
...then in Pig-Latin

good_urls = FILTER urls BY pagerank > 0.2

groups = GROUP good_urls BY category

big_groups = FILTER groups

BY COUNT(good_urls) > 10^6

output = FOREACH big_groups GENERATE

category, AVG(good_urls.pagerank)
Types in Pig-Latin

• Atomic: string or number, e.g. ‘Alice’ or 55

• Tuple: (‘Alice’, 55, ‘salesperson’)

• Bag: {('Alice', 55, 'salesperson'),
     ('Betty', 44, 'manager'), ...}

• Maps: we will try not to use these
Types in Pig-Latin

Bags can be nested!

- \{('a', \{1,4,3\}), ('c', \{\}), ('d', \{2,2,5,3,2\})\}

Tuple components can be referenced by number

- $0, $1, $2, …
\[ t = (\text{\textquotesingle}alice\textquotesingle}, \{ (\text{\textquotesingle}lakers\textquotesingle}, 1) \}, \{ (\text{\textquotesingle}iPod\textquotesingle}, 2) \}, \{ \text{\textquotesingle}age\textquotesingle \to 20 \} ) \]

Let fields of tuple \( t \) be called \( f_1, f_2, f_3 \)

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Example</th>
<th>Value for ( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>\text{\textquotesingle}bob\textquotesingle</td>
<td>Independent of ( t )</td>
</tr>
<tr>
<td>Field by position</td>
<td>$0</td>
<td>\text{\textquotesingle}alice\textquotesingle</td>
</tr>
<tr>
<td>Field by name</td>
<td>( f_3 )</td>
<td>{ \text{\textquotesingle}age\textquotesingle \to 20 }</td>
</tr>
<tr>
<td>Projection</td>
<td>( f_2.$0 )</td>
<td>{ (\text{\textquotesingle}lakers\textquotesingle) } { (\text{\textquotesingle}iPod\textquotesingle) }</td>
</tr>
<tr>
<td>Map Lookup</td>
<td>( f_3#\text{\textquotesingle}age\textquotesingle )</td>
<td>20</td>
</tr>
<tr>
<td>Function Evaluation</td>
<td>( \text{SUM}(f_2.$1) )</td>
<td>1 + 2 = 3</td>
</tr>
<tr>
<td>Conditional Expression</td>
<td>( f_3#\text{\textquotesingle}age\textquotesingle\to18?\text{\textquotesingle}adult\textquotesingle:\text{\textquotesingle}minor\textquotesingle )</td>
<td>\text{\textquotesingle}adult\textquotesingle</td>
</tr>
<tr>
<td>Flattening</td>
<td>\text{FLATTEN}(f_2)</td>
<td>{ (\text{\textquotesingle}lakers\textquotesingle}, 1 } { (\text{\textquotesingle}iPod\textquotesingle}, 2 }</td>
</tr>
</tbody>
</table>
Loading data

• Input data = FILES!
  – Heard that before?

• The LOAD command parses an input file into a bag of records

• Both parser (=“deserializer”) and output type are provided by user
Loading data

queries = LOAD 'query_log.txt'
    USING myLoad()
    AS (userID, queryString, timeStamp)
Loading data

• USING userfuction() -- is optional
  – Default deserializer expects tab-delimited file

• AS type – is optional
  – Default is a record with unnamed fields; refer to them as $0, $1, …

• The return value of LOAD is just a handle to a bag
  – The actual reading is done in pull mode, or parallelized
FOREACH

expanded_queries =
  FOREACH queries
  GENERATE userId, expandQuery(queryString)

expandQuery( ) is a UDF that produces likely expansions
Note: it returns a bag, hence expanded_queries is a nested bag
expanded_queries =
    FOREACH queries
    GENERATE userId,
        flatten(expandQuery(queryString))

Now we get a flat collection
queries:
(userId, queryString, timestamp)

(alice, lakers, 1)
(bob, iPod, 3)

FOREACH queries GENERATE
expandQuery(queryString)
(without flattening)

(alice, {lakers rumors})
(bob, {iPod nano})

(alice, {lakers news})
(bob, {iPod shuffle})
(with flattening)

(alice, lakers rumors)
(alice, lakers news)
(bob, iPod nano)
(bob, iPod shuffle)
FLATTEN

Note that it is NOT a first class function!
(that’s one thing I don’t like about Pig-latin)

• First class FLATTEN:
  – FLATTEN(\{\{2,3\},\{5\},\{\},\{4,5,6\}\}) = \{2,3,5,4,5,6\}
  – Type: \{\{T\}\} \rightarrow \{T\}

• Pig-latin FLATTEN
  – FLATTEN(\{4,5,6\}) = 4, 5, 6
  – Type: \{T\} \rightarrow T, T, T, …, T     ?????
FILTER

Remove all queries from Web bots:

real_queries = FILTER queries BY userId neq 'bot'

Better: use a complex UDF to detect Web bots:

real_queries = FILTER queries BY NOT isBot(userId)
JOIN

results:  { (queryString, url, position) }
revenue:  { (queryString, adSlot, amount) }

join_result = JOIN results BY queryString
             revenue BY queryString

join_result : { (queryString, url, position, adSlot, amount) }
results:
(queryString, url, rank)
(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)

revenue:
(queryString, adSlot, amount)
(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)

JOIN

(lakers, nba.com, 1, top, 50)
(lakers, nba.com, 1, side, 20)
(lakers, espn.com, 2, top, 50)
(lakers, espn.com, 2, side, 20)
...
GROUP BY

revenue: {(queryString, adSlot, amount)}

grouped_revenue = GROUP revenue BY queryString

query_revenues =
  FOREACH grouped_revenue
  GENERATE queryString,
    SUM(revenue.amount) AS totalRevenue

grouped_revenue: {(queryString, {(adSlot, amount)})}
query_revenues: {(queryString, totalRevenue)}
Simple Map-Reduce

\[
\text{input} : \{(\text{field1, field2, field3, \ldots})\}
\]

\[
\text{map\_result} = \text{FOREACH input} \quad \text{GENERATE FLATTEN(map(\ast))}
\]

\[
\text{key\_groups} = \text{GROUP map\_result BY $0$
\]

\[
\text{output} = \text{FOREACH key\_groups} \quad \text{GENERATE reduce($1$)}
\]

\[
\text{map\_result} : \{(a1, a2, a3, \ldots)\}
\]

\[
\text{key\_groups} : \{(a1, \{(a2, a3, \ldots)\})\}
\]
Co-Group

results: {(queryString, url, position)}
revenue: {(queryString, adSlot, amount)}

grouped_data =
  COGROUP results BY queryString,
  revenue BY queryString;

grouped_data: {(queryString, results:{(url, position)},
  revenue:{(adSlot, amount)})}

What is the output type in general?
Co-Group

Is this an inner join, or an outer join?
Co-Group

grouped_data: {{queryString, results:{{url, position}},
    revenue:{{adSlot, amount}}}}

url_revenues = FOREACH grouped_data
    GENERATE
    FLATTEN(distributeRevenue(results, revenue));

distributeRevenue is a UDF that accepts search results and revenue information for a query string at a time, and outputs a bag of urls and the revenue attributed to them.
Co-Group v.s. Join

grouped_data: { (queryString, results: { (url, position) },
    revenue: { (adSlot, amount) }) }

grouped_data = COGROUP results BY queryString,
    revenue BY queryString;
join_result = FOREACH grouped_data
    GENERATE FLATTEN(results),
    FLATTEN(revenue);

Result is the same as JOIN
Asking for Output: STORE

STORE query_revenues INTO `myoutput'
    USING myStore();

Meaning: write query_revenues to the file `myoutput'
Implementation

- Over Hadoop!
- Parse query:
  - Everything between LOAD and STORE $\rightarrow$ one logical plan
- Logical plan $\rightarrow$ sequence of Map/Reduce ops
- All statements between two (CO)GROUPs $\rightarrow$ one Map/Reduce op
Implementation