Introduction to Database Systems
CSE 444

Lecture 22: Pig Latin
Outline

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

• Map-reduce is a low-level programming environment

• In most applications need more complex queries

• Pig accepts higher level queries written in Pig Latin, translates them into ensembles of MapReduce jobs
  – Pig is the system
  – Pig Latin is the language
Pig Engine 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
Pig Engine Overview

Pig Latin program

Parsed program

Execution plan

MR jobs

Hadoop

Disk

Disk

Disk

Disk

Disk
Pig-latin will NOT be on the Final

• Pig-latin is a new, experimental language
  – (imperfect design, in my opinion)

• Why do we discuss this in class?
  – Because we want to learn massively parallel queries → Project4
  – And because MapReduce is too difficult to use
  – And because no other free language is available
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...

```sql
SELECT category, AVG(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP BY category
HAVING COUNT(*) > 10^6
```
Pig Latin combines
• high-level declarative querying in the spirit of SQL, and
• low-level, procedural programming a la map-reduce.
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, …
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>'bob'</td>
<td>Independent of ( t )</td>
</tr>
<tr>
<td>Field by position</td>
<td>0</td>
<td>'alice'</td>
</tr>
<tr>
<td>Field by name</td>
<td>( f_3 )</td>
<td></td>
</tr>
<tr>
<td>Projection</td>
<td>( f_2.0 )</td>
<td>{ ('lakers') }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ ('iPod') }</td>
</tr>
<tr>
<td>Map Lookup</td>
<td>( f_3#'age' )</td>
<td>20</td>
</tr>
<tr>
<td>Function Evaluation</td>
<td>( \text{SUM}(f_2.0) )</td>
<td>1 + 2 = 3</td>
</tr>
<tr>
<td>Conditional Expression</td>
<td>( f_3#'age'&gt;18? )\ 'adult': 'minor'</td>
<td>'adult'</td>
</tr>
<tr>
<td>Flattening</td>
<td>( \text{FLATTEN}(f_2) )</td>
<td>{ ('lakers', 1) } { ('iPod', 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
FOREACH

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, iPod shuffle})

(with flattening)

(alice, lakers rumors)
(alice, lakers news)
(bob, iPod nano)
(bob, iPod shuffle)
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}} → {T}

• Pig-latin FLATTEN
  – FLATTEN({4,5,6}) = 4, 5, 6
  – Type: {T} → T, T, T, ..., T
FILTER

Remove all queries from Web bots:

\[
\text{real}_{\text{queries}} = \text{FILTER queries BY userId neq 'bot'}
\]

Better: use a complex UDF to detect Web bots:

\[
\text{real}_{\text{queries}} = \text{FILTER queries BY NOT isBot(userId)}
\]
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)

(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

input : {((field1, field2, field3, . . . .))}

map_result = FOREACH input
    GENERATE FLATTEN(map(*))

key_groups = GROUP map_result BY $0

output = FOREACH key_groups
    GENERATE reduce($1)

map_result : {((a1, a2, a3, . . .))}

key_groups : {((a1, {(a2, a3, . . .))})}
Final Comment

• More about Pig and Pig Latin next week

• Project 4: start by downloading pig, run the tutorial on your local machine