Lecture 26b: Supplementary slides for Pig Latin

Friday, Dec 3, 2010

1

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)

2

Why ?

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

3

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

4

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

5

First in SQL…

```
SELECT category, AVG(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP By category
HAVING COUNT(*) > 10^6
```

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)
```

7

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

8

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, …

9

$$t = \left(\text{'alice'}, \left\{\begin{matrix}(\text{'lakers'}, 1)\\(\text{'iPod'}, 2)\end{matrix}\right\}, [\text{'age'} \rightarrow 20]\right)$$

Let fields of tuple t be called f1, f2, f3

Expression Type	Example	Value for t
Constant	'bob'	Independent of t
Field by position	$0	'alice'
Field by name	f3	['age' → 20]
Projection	f2.$0	{('lakers'), ('iPod')}
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2

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

11

Loading data

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

12

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

13

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

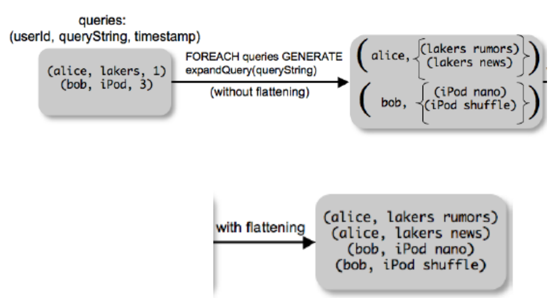
14

FOREACH

```
expanded_queries =
  FOREACH queries
  GENERATE userId,
              flatten(expandQuery(queryString))
```

Now we get a flat collection

15



16

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}} → {T}
- Pig-latin FLATTEN
  - FLATTEN({4,5,6}) = 4, 5, 6
  - Type: {T} → T, T, T, …, T   ?????

17

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)
```

18

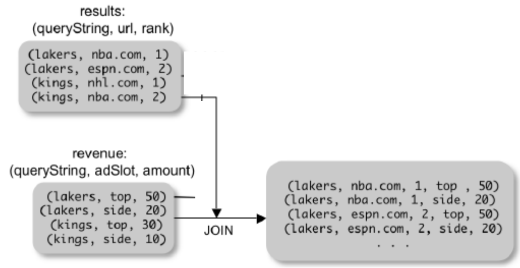
JOIN

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

```
join_result = JOIN results BY queryString
                   revenue BY queryString
```

join_result : {(queryString, url, position, adSlot, amount)}

19



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)}

21

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, . . .)})}

22

Co-Group

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

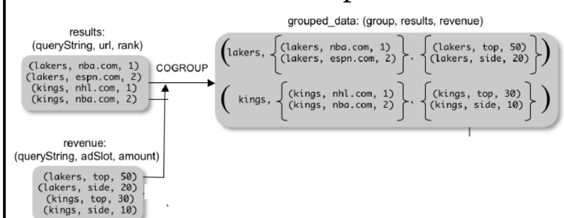
```
grouped_data =
    COGROUP results BY queryString,
            revenue BY queryString;
```

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

What is the output type in general ?

23

Co-Group



Is this an inner join, or an outer join ?

24

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 re-
sults and revenue information for a query string at a time,
and outputs a bag of urls and the revenue attributed to them.

25

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

26

Asking for Output: STORE

```
STORE query_revenues INTO `myoutput'
    USING myStore();
```

Meaning: write query_revenues to the file 'myoutput'

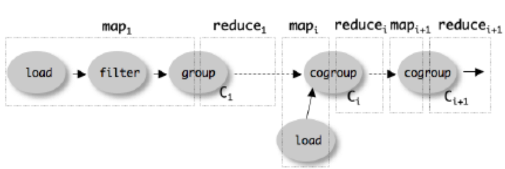
27

Implementation

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

28

Implementation



29