Introduction to Data Management CSE 344

Lecture 23: Pig Latin

Some slides are courtesy of
Alan Gates, Yahoo!Research

Outline

- Review of MapReduce
- Introduction to Pig System
- Pig Latin tutorial

MapReduce

- Google: paper published 2004
- Open source implementation: Hadoop
- MapReduce = high-level programming model and implementation for large-scale parallel data processing
- Main competing alternative Dryad from Microsoft

MapReduce Data Model

- Files !
- A file = a bag of (key, value) pairs
- A MapReduce program:
  - Input: a bag of (input key, value) pairs
  - Output: a bag of (output key, value) pairs

Step 1: the MAP Phase

- User provides the MAP function:
  - Input: one (input key, value)
  - Ouput: a bag of (intermediate key, value) pairs
- System applies map function in parallel to all (input key, value) pairs in the input file

Step 2: the REDUCE Phase

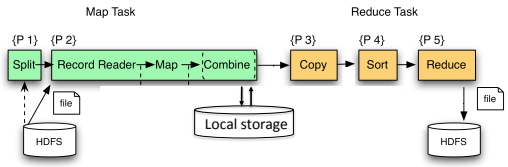
- User provides the REDUCE function:
  - Input: intermediate key, and bag of values
  - Output: bag of output values
- System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

Parallel Execution in Cluster

See illustration on white board:

- Data is typically a file in the Google File System
  - HDFS for Hadoop
  - File system partitions file into chunks
  - Each chunk is replicated on k (typically 3) machines
- Each machine can run a few map and reduce tasks simultaneously
- Each map task consumes one chunk
  - Can adjust how much data goes into each map task using "splits"
  - Scheduler tries to schedule map task where its input data is located
- Map output is partitioned across reducers
- Map output is also written locally to disk
- Number of reduce tasks is configurable
- System shuffles data between map and reduce tasks
- Reducers sort-merge data before consuming it

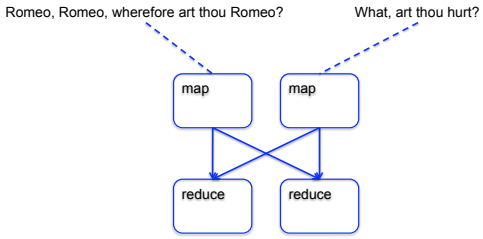
MapReduce Phases



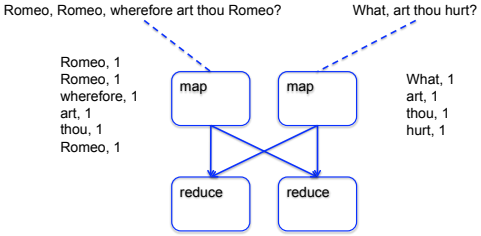
MapReduce

- Computation is moved to the data
- A simple yet powerful *programming* model
  - Map: every record handled individually
  - Shuffle: records collected by key
  - Reduce: key and iterator of all associated values
- User provides:
  - input and output (usually files)
  - map Java function
  - key to aggregate on
  - reduce Java function
- Opportunities for more control: partitioning, sorting, partial aggregations, etc.

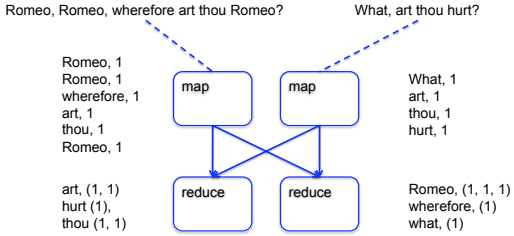
MapReduce Illustrated



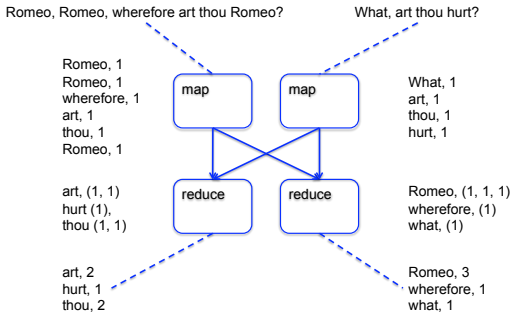
MapReduce Illustrated



MapReduce Illustrated



MapReduce Illustrated



Making Parallelism Simple

- Sequential reads = good read speeds
- In large cluster failures are guaranteed; MapReduce handles retries
- Good fit for batch processing applications that need to touch all your data:
  - data mining
  - model tuning
- Bad fit for applications that need to find one particular record
- Bad fit for applications that need to communicate between processes; oriented around independent units of work

MapReduce (MR) tools

Open source MapReduce implementation:

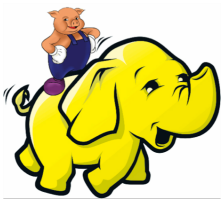
One MR query language: Pig Latin

Query engine:

Graphics taken from: hadoop.apache.org and research.yahoo.com/node/90

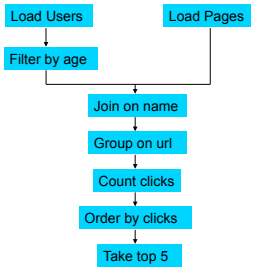
What is Pig?

- An engine for executing programs on top of Hadoop
- It provides a language, Pig Latin, to specify these programs
- An Apache open source project
  http://hadoop.apache.org/pig/



Why use Pig?

Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited sites by users aged 18 - 25.



In MapReduce

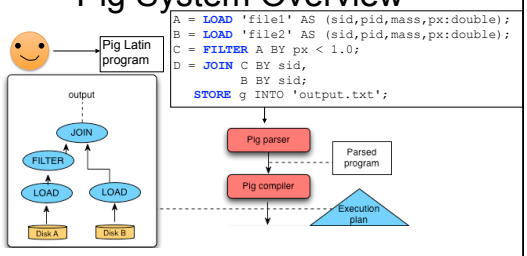
170 lines of code, 4 hours to write

In Pig Latin

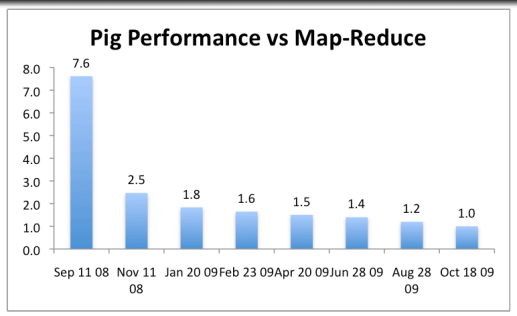
```
Users = load 'users' as (name, age);
Fltrd = filter Users by
        age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
       COUNT(Jnd) as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5;
store Top5 into 'top5sites';
```

9 lines of code, 15 minutes to write

Pig System Overview



But can it fly?



Pig Latin Mini-Tutorial

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

Pig-Latin Overview

- Data model = loosely typed *nested relations*
- Query model = Relational Algebra (less SQL)
- Execution model:
  - Option 1: run locally (pig –x local)
  - Option 2: run in a Hadoop cluster
- Option 1 is good for debugging

Pages(url, category, pagerank)

Example: In SQL…

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

Pages(url, category, pagerank)

Example: …in Pig-Latin

```
pages = LOAD 'pages-file.txt' as (url, category, pagerank);
good_pages = FILTER pages BY pagerank > 0.2;
groups = GROUP good_pages BY category;
big_groups = FILTER groups
                BY COUNT(good_pages) > 10^6;
result = FOREACH big_groups GENERATE
         group, AVG(good_pages.pagerank);
```

Use "describe" to check schema

For the last statement, can also use:

```
result = FOREACH big_groups GENERATE $0, AVG($1.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 not 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 name or by number

- $0, $1, $2, …
- url, category, pagerank,…

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

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 !
- LOAD command parses an input file
- Parser provided by user
  - In HW6: myudfs.jar, function RDFSplit3
  - Example.pig shows you how to use it

Loading data

```
pages = LOAD 'pages-file.txt'
        USING myLoad( )
          AS (url, category, pagerank)
```

```
pages = LOAD 'pages-file.txt'
        USING PigStorage(',')
          AS (url, category, pagerank)
```

FOREACH

```
pages_projected =
   FOREACH pages
   GENERATE url, pageRank
```

Note: you cannot write joins in FOREACH
FOREACH pages, users = error

Loading Data in HW6

There is no parser for RDF data, hence:

```
-- example.pig = in your starter code
register s3n://uw-cse344-code/myudfs.jar

raw = LOAD 's3n://uw-cse344-test/cse344-test-file'
        USING TextLoader as (line:chararray);

ntriples = FOREACH raw
           GENERATE FLATTEN(myudfs.RDFSplit3(line))
           AS (subject:chararray,
               predicate:chararray,
               object:chararray);
```

FILTER

Remove all queries from Web bots:

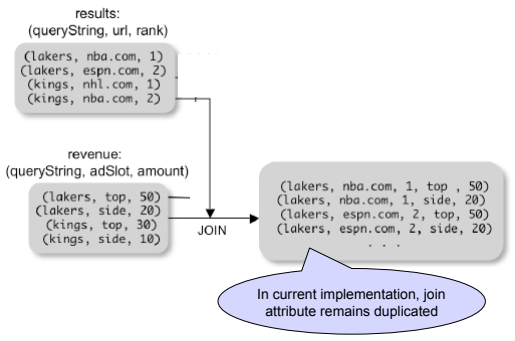
```
good_pages =  FILTER pages BY pageRank > 0.8;
```


JOIN

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

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

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



GROUP BY

revenue: {(queryString, adSlot, amount)}

```
grp_revenue = GROUP revenue BY queryString
query_revenues =
    FOREACH grp_revenue
    GENERATE group as queryString,
             SUM(revenue.amount) AS totalRevenue
```

grp_revenue: {(group, {(queryString, adSlot, amount)})}
query_revenues: {(queryString, totalRevenue)}

Simple MapReduce

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

```
map_result = FOREACH input
                GENERATE FLATTEN(map(*))
key_groups = GROUP map_result BY $0
output = FOREACH key_groups
          GENERATE reduce(*)
```

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

Co-Group

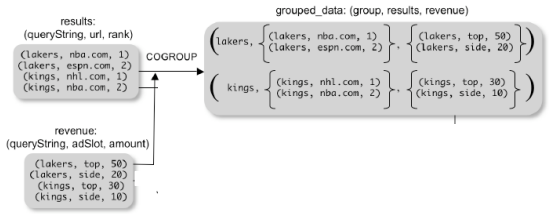
results: {(queryString, url, position)}
revenue: {(queryString, adSlot, amount)}

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

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

What is the output type in general ?

Co-Group



Is this an inner join or an outer join ?

Co-Group

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

In example.pig

```
STORE count_by_object_ordered
IN '/user/hadoop/example-results'
USING PigStorage();
```

Implementation

- Over Hadoop !
- Parse query:
  - Everything between LOAD and STORE → one logical plan
- Logical plan → sequence of MapReduce jobs
- All statements between two (CO)GROUPs → one MapReduce job

Implementation