ADVANCED PIG

CKME 134 – BIG DATA ANALYTICS TOOLS
RYERSON UNIVERSITY
SPRING 2015

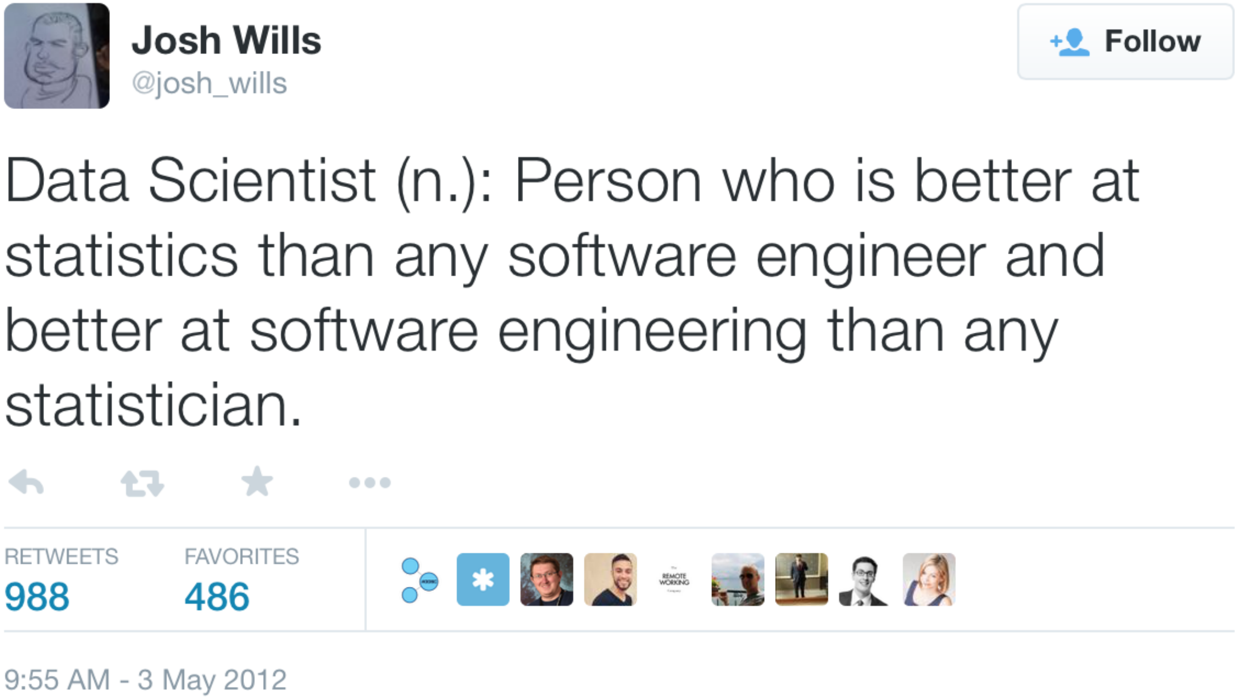
Instructor: Shaohua Zhang

Course Outline

1. Intro to Big Data
2. Distributed Computing and MapReduce
3. Hadoop Ecosystem
4. Programming Hive
5. Advanced Hive
6. Mid-Term Review
7. Programming Pig
8. Advanced Pig

9. Hadoop Performance Optimization
10. Hadoop In Action: Building Data Pipelines
9. Location Analytics and Recommender Systems
11. Beyond Hadoop: Spark
12. Beyond Hadoop: Graph Analytics

Data Scientist?



Josh Wills
@josh_wills

Data Scientist (n.): Person who is better at statistics than any software engineer and better at software engineering than any statistician.

RETWEETS 988
FAVORITES 486

9:55 AM - 3 May 2012

(video) Airbnb Tech Talk: Josh Wills - The Life of a Data Scientist
https://www.youtube.com/watch?v=h9vQIPfe2uU

Recap

Type	Description	Example
int, long, float, duble, chararray, bytearray, boolean, datetime	• primitive data types in pig	
tuple	• Fixed length, **ordered** set of fields, like a row with multiple columns in SQL • Allows random access • Must fit in memory	(toronto, 3) (‘bob’, 53, ‘toronto’, ‘male’)

bag	• An **unordered** collection of tuples
	• Can have tuples with differing numbers of fields
	• Can spill to disk (doesn’t have to fit in memory
		{(toronto, 3),(chicago, 5)}
		{(‘bob’,53,’toronto’,’male’), (‘sally’, 23), ‘george’, ‘montreal’)}

map	• A set of key/value pairs • Key/values in a relation must be unique • Key must be chararray, data element can be any type	[city#toronto] [name#bob,age#53,city#toronto,gender#male]

Tuple

- □ Tuple

twitter:tuple(id:chararray, lat:double, lon:double, tweet:chararray)

twitter.id → USER_12345678

twitter.$3 → #Hadoop, big data is the new oil

twitter.($1,$2) → (40.48956, -156.22234)

twitter.(lat,lon) → (40.48956, -156.22234)

Bag

id	field1	field2	field3
user1	a	b	c
user2	a	b	
user3	a		

```
grunt> data = load '/user/lab/pig/data_test_bag' using PigStorage('\t') as
(id:chararray, f1:chararray, f2:chararray, f3:chararray);
grunt> grpd = group data ALL;

grunt> describe grpd;
grpd: {group: chararray,data: {(id: chararray,f1: chararray,f2: chararray,f3: chararray)}}

grunt> dump grpd;
(all,{(user3, a, , ), (user2, a, b, ), (user1, a, b, c)})

cnt_id = foreach grpd generate COUNT(data.id) as cnt;     ← 3
cnt_f1 = foreach grpd generate COUNT(data.$1) as cnt;     ← 3
cnt_f2 = foreach grpd generate COUNT(data.f2) as cnt;     ← 2
cnt_f3 = foreach grpd generate COUNT(data.$3) as cnt;     ← 1
```

Complex Type

- Type Construction Operators/Functions
  - Tuple construction (a, b) → Same as TOTUPLE()
  - Bag construction {a, b} → Same as TOBAG()
  - Map construction [a, b] → Same as TOMAP()

Bag Construction

```
A = load 'students' as (name:chararray, age:int, gpa:float);
B = foreach A generate {(name, age)}, {name, age};
store B into ‘results’;
```

Input (students):

```
joe smith    20    3.5
amy chen     22    3.2
leo allen    18    2.1
```

Output (results):

```
{(joe smith,20)} {(joe smith), (20)}
{(amy chen,22)} {(amy chen), (22)}
{(leo allen,18)} {(leo allen), (18)}
```

Map Construction

```
A = load 'students' as (name:chararray, age:int, gpa:float);
B = foreach A generate [name, gpa];
store B into ‘results’;
```

Input (students):

```
joe smith 20 3.5
amy chen 22 3.2
leo allen 18 2.1
```

Output (results):

```
[joe smith#3.5]
[amy chen#3.2]
[leo allen#2.1]
```

Comparison Operator

```
data = load '/user/lab/pig/full_text.txt' AS (id:chararray,
ts:chararray, location:chararray, lat:float, lon:float,
tweet:chararray);
filtr = FILTER data BY tweet MATCHES '.*@USER_\\S{8}.*';
limt = limit filtr 20;
dump limt;
```

Scalar Projections

- Pig allows you to cast the elements of a single-tuple relation into a scalar value
- The primary use case for casting relations to scalars is the ability to use the values of global aggregates in follow up computations

Scalar Projection Demo

Normalize average number of tweets per user

```
a = load '/user/lab/pig/full_text.txt' AS (id:chararray, ts:chararray,
location:chararray, lat:float, lon:float, tweet:chararray);

b = group a by id;

c = foreach b generate group as id, COUNT(a) as user_cnt;
```

id	*cnt*
user1	*45*
user2	*22*
user3	*67*

```
d = group c ALL;

e = foreach d generate AVG(c.user_cnt) as global_avg;
```

	global_avg
ALL	*44.67*

```
f = foreach c generate id, user_cnt/(float)e.global_avg as index;
```

id	*cnt*	*index*
user1	*45*	***1.00***
user2	*22*	***0.49***
user3	*67*	***1.50***

```
store f into '/user/lab/pig/tweet_count_index';
```

COGROUP

- COGROUP is a generalization of GROUP that works with more relations
- You can COGROUP up to but no more than 127 relations at a time
- The result is a record with a key and one bag for each input. Each bag contains all records from that input that have the given value for the key
- Another way to think of COGROUP is as the first half of a join. The keys are collected together, but the cross product is not done
- COGROUP is useful when you want to do join-like things but not a full join
- COGROUP + FLATTEN = JOIN

COGROUP Demo

uid	age	gender	region
u1	14	M	US
u2	32	F	UK
u3	22	M	US

uid	region
u1	US
u1	UK
u1	CA
u2	US

```
grunt> user = load '/user/lab/pig/user.txt' using PigStorage(',') as (uid:chararray,
age:int, gender:chararray, region:chararray);
grunt> session = load '/user/lab/pig/session.txt' using PigStorage(',') as
(uid:chararray, region:chararray);

grunt> cogrp = COGROUP user BY uid, session BY uid;
grunt> describe cogrp;
cogrp: {group: chararray,user: {(uid: chararray,age: int,gender: chararray,region:
chararray)},session: {(uid: chararray,region: chararray)}}

grunt> dump cogrp;
(u1,{(u1,14, M, US)}, {(u1, CA),(u1, UK),(u1, US)})
(u2,{(u2,32, F, UK)}, {(u2, US)})
(u3,{(u3,22, M, US)}, {})

grunt> cogrp_nest = foreach cogrp {
    crossed = cross user, session;
    generate crossed;
}
grunt> dump cogrp_nest;
({(u1,14, M, US, u1, CA), (u1,14, M, US, u1, UK), (u1,14, M, US, u1, US)})
({(u2,32, F, UK,u2, US)})
```

Set Intersection

with COGROUP

name	*hits*
John	*3*
Harry	*4*
George	*2*

name	*errors*
John	*2*
John	*3*
George	*0*
Sue	*1*

```
grunt> s1 = load ‘hits-data’ as (name:chararray, hits:int);
grunt> s2 = load ‘errors-data’ as (name:chararray, errors:int);

grunt> grps = COGROUP s1 BY name, s2 BY name;

(John, {(John, 3)}, {(John, 2), (John, 3)})
(Harry, {(Harry, 4)}
(George, {(George, 2), (George, 2)})
(Sue, { }, {(Sue, 1)})

-- Note:
-- Something is in the intersection of s1 and s2 if there are no {}’s in the
cogroup.

grunt> grps2 = FILTER grps by NOT(IsEmpty(s1)) AND NOT(IsEmpty(s2));
grunt> intersection = FOREACH grp2 GENERATE group as grp, s1, s2 ;
grunt> dump intersection;

(John,{(John,3)},{(John,3),(John,2)})
(George,{(George,2)},{(George,0)})
```

Set Difference

with COGROUP

name	hits
John	3
Harry	4
George	2

name	errors
John	2
John	3
George	0
Sue	1

```
grunt> s1 = load ‘hits-data’ as (name:chararray, hits:int);
grunt> s2 = load ‘errors-data’ as (name:chararray, errors:int);

grunt> grps = COGROUP s1 BY name, s2 BY name;
```

```
(John, {(John, 3)}, {(John, 2), (John, 3)})
(Harry, {(Harry, 4)}
(George, {(George, 2), (George, 2)})
(Sue, { }, {(Sue, 1)})
```

```
-- Note:
-- Something is in the difference between s1 and s2 if there are non-empty
second sets.

grunt> grps2 = FILTER grps by IsEmpty(s2);
grunt> set_diff = FOREACH grps2 GENERATE group as grp, s1, s2 ;
grunt> dump set_diff;
```

```
(Harry,{(Harry,4)},{})
```

Advanced Joins

- replicated join
- skewed join
- semi-join (co-group)
- non-equi join (cross)

Replicated Join

- Joins small data to large data
- Often used when a lookup file is in a smaller file
  - Small enough to fit in memory of your computer node
- Using **distributed cache**
- Replicated join applies map-only join
  - No reduce phase – Yeaaah!!
- Supports only inner and left outer join
- Can be used with more than two tables.
- In this case, all but the first (left-most) table are read into memory
  - The lookup file should always be on the right side

Replicated JOIN Demo

Finding Weekend Tweets

```
a = load '/user/lab/pig/full_text.txt' using PigStorage('\t') AS (id:chararray,
ts:chararray, location:chararray, lat:float, lon:float, tweet:chararray);
a1 = foreach a generate id, ts, SUBSTRING(ts,0,10) as date;

b = load '/user/lab/pig/dayofweek.txt' using PigStorage('\t') as
(date:chararray, dow:chararray);
b1 = filter b by dow=='Saturday' or dow=='Sunday';

c = join a1 by date, b1 by date using 'replicated';
d = foreach c generate a1::id .. a1::date, b1::dow as dow;
e = limit d 5;
dump e;
```

Skewed Join c = join a by key, b by key using ‘skewed’;

- □ Skewed join is used when the data has significant skew in the number of records per key. This results in one or two reducers that will take much longer than the rest → Straggler problem
- □ Adds 5% overhead to the performance due to the sampling approach at the beginning
- □ Skew join works by first sampling one input for the join. In that input it identifies any keys that have so many records that skew join estimates it will not be able to fit them all into memory.
- □ Those keys identified as too large are treated differently. Based on how many records were seen for a given key, those records are split across the appropriate number of reducers. The number of reducers is chosen based on Pig’s estimate of how wide the data must be split such that each reducer can fit its split into memory. For the input to the join that is not split, those keys that were split are then replicated to each reducer that contains that key.

Non-Equi Join (CROSS)

find user with more than 2 friends

p_name	value
nfriends	2
ndays	100
nvists	13

name	friend
Amy	George
George	Fred
Fred	Anne
George	Joe
George	Harry

```
grunt> params = load '/user/lab/pig/params.txt' using PigStorage(',') as (p_name:chararray, value:int);
grunt> friend = load '/user/lab/pig/friend.txt' using PigStorage(',') as (name:chararray, friend:chararray);
grunt> friend_grp = group friend by name;
grunt> friend_cnt = foreach friend_grp generate group as name, COUNT(friend.friend) as cnt;
```

name	friend
Amy	1
George	3
Fred	1

```
grunt> friend_param = filter params by p_name=='nfriends';
grunt> friend_param_p = foreach friend_param generate value;
```

2

```
grunt> friend_cross = CROSS friend_cnt, friend_param_p;
```

name	friend	fr_param_p
Amy	1	2
George	3	2
Fred	1	2

```
grunt> friend_cross_1 = filter friend_cross by friend_cnt::cnt >= friend_param_p::value;
grunt> dump friend_cross_1;
```

George	3	2

Sampling

```
-- approach 1
grunt> data = load 'file';
grunt> sample1= SAMPLE data 0.1;

-- approach 2
grunt> data = load 'file';
grunt> sample2 = FILTER A by random() <= 0.1;

-- approach 3
grunt> data = load 'file';
grunt> grpd = group a all;
grunt> grpd_cnt = foreach b generate COUNT(data) as num_rows;
grunt> sample3 = SAMPLE a 10000/grpd_cnt .num_rows;
```

UDF – User Defined Function

- Java
  - Java has the most extensive support. You can customize all parts of the processing including:
- Jython/Python/Javascript/Ruby/Groovy
  - Limited support is provided for Python, Jython etc.
- Where to get UDFs?
  - Built-In Functions
  - Piggybank/DataFu/Pigeon/etc.
    - Download from project site (maven central)
    - Download source and compile

Git & Github

- Git Installation and GitHub Setup
  - Refer to “Git & GitHub” section in Module 8 BlackBoard

UDF Usage

- Find the JAR online via maven central
  - OR download source and build your own in Eclipse
- Upload the JAR to Hadoop Sandbox
- In Pig
  - Register the JAR first
  - Define the functions to use
  - Invoke the function

Piggybank

http://search.maven.org

```
------------------------

-- piggybank UDFs

------------------------

REGISTER '/home/lab/piggybank-0.14.0.jar';

DEFINE isotounix org.apache.pig.piggybank.evaluation.datetime.convert.ISOToUnix();


a = load '/user/lab/pig/full_text.txt' AS (id:chararray, ts:chararray, location:chararray,
lat:float, lon:float, tweet:chararray);

b = foreach a generate id, ts, isotounix(ts) as ts_unix;

c = limit b 3;

dump c;
```

DataFu – Random Sampling

```
---------------------------------------------------------
-- Simple Random Sampling (SRS)
-- Take a 1% random sample from the dataset
---------------------------------------------------------
-- Register UDFs and define functions first
register /home/lab/datafu-1.2.0.jar
DEFINE SRS datafu.pig.sampling.SimpleRandomSample('0.01');

-- Simple Random Sampling (SRS)
data = load '/user/lab/pig/full_text.txt' AS (id:chararray, ts:chararray, location:chararray,
lat:float, lon:float, tweet:chararray);
sampled = foreach (group data all) generate flatten(SRS(data));
store sampled into '/user/lab/pig/full_text_src';

-- Stratified Sampling (by date)
-- Take a 1% random sample from each date using SRS and group by date
data = load '/user/lab/pig/full_text.txt' AS (id:chararray, ts:chararray, location:chararray,
lat:float, lon:float, tweet:chararray);
data1 = foreach data generate id, SUBSTRING(ts, 0, 10) as date, lat, lon, tweet;
grouped = GROUP data1 BY date;
sampled = foreach grouped generate flatten(SRS(data1));
store sampled into '/user/lab/pig/full_text_stratified';
```

Pigeon – UDF for GeoSpatial

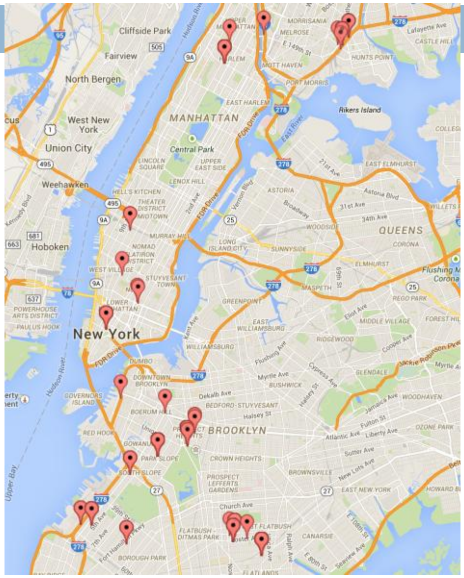
- Download and build pigeon jars
  - git clone https://github.com/aseldawy/pigeon.git
  - import project as maven in eclipse
  - build jar
  - under "target" sub-folder, find jar file
  - Upload to sandbox via ftp
- Demo

Pigeon UDF Demo

Find tweets tweeted from NYC

```
-- register UDFs
REGISTER /home/lab/pigeon-1.0-SNAPSHOT.jar;
REGISTER /home/lab/esri-geometry-api-1.2.jar;

-- define functions
DEFINE ST_MakeBox edu.umn.cs.pigeon.MakeBox;
DEFINE ST_Contains edu.umn.cs.pigeon.Contains;
DEFINE ST_MakePoint edu.umn.cs.pigeon.MakePoint;
```



```
data = load '/user/lab/pig/full_text.txt' AS (id:chararray, ts:chararray,
location:chararray, lat:double, lon:double, tweet:chararray);
data1 = FOREACH data GENERATE id, ts, lat, lon, ST_MakePoint(lat, lon)
AS geom_point, tweet;
data2 = FILTER data1 BY ST_Contains(ST_MakeBox(40.4774, -74.2589,
40.9176, -73.7004), geom_point);
data3 = limit data2 200;
data4 = foreach data3 generate lat, lon;
dump data4;
```

http://www.darrinward.com/lat-long/?id=490564