So far,

Wednesday, February 23, 2011
7:06 PM

So far,

We understand what Pig is for.
We have a basic knowledge of the primitives.
Now it's time to do something non-trivial with it.

Pig actions as transformations

Thursday, February 24, 2011
1:33 PM

Pig actions can either

filter and order records (without changing the schema):

DISTINCT, FILTER, LIMIT, ORDER...

or transform data schemas:

FOREACH-GENERATE, GROUP-BY, JOIN-BY …

Reading a schema: remember that

`(…)` is a tuple

`{…}` is a bag of tuples

`a:int` is an element named a that is an int.

So the schema

```
foo: {a:int, b:chararray}
```

refers to an alias foo that generates a tuple space (outer bag) of tuples where the first term is an int and the second term is a chararray (string). while the schema

```
bar: {a:int, b:{c:int, d:chararray}}
```

refers to an alias bar that generates a tuple space (outer bag) of tuples, where each tuple has a first term that is an int and a second term that is a bag of two-tuples.

You can print schemas with the DESCRIBE command.

In the following,

Thursday, February 24, 2011
1:36 PM

In the following,

I am going to make significant use of schema transformation.
Figuring out what I am doing is going to depend upon
tracking how schemas change and
what the changes mean.

Let's review the basic schema transformations:
If I have a schema, e.g.,

```
input: {a:int, b:chararray}
```

then

```
output = FOREACH input GENERATE a AS c, a+1 as d;
```

produces the schema

```
output: {c:int, d:int}
```

And the statement

```
g = GROUP input by a;
```

produces the schema

original schema

```
g: {group:int, input:{a:int, b:chararray}}
```

If I want, I can remove a part of the inner bag tuples by subranging:

```
g2 = FOREACH g GENERATE group as a, input.b as bs;
```

which will give the schema

```
g2: {a:int, bs:{b: chararray}}
```

smallt bag

If I have two schemas:

```
in1: {a:int, b:chararray}
in2: {c:int, d:chararray}
```

then the inner join

```
j = JOIN in1 BY a, in2 BY c;
```

produces the schema

```
j: {in1::a :int, in1::b :chararray, in2::c :int,
in2::d :chararray};
```

Note: {} implies {()} because a bag can only contain tuples. Projecting those to scalars creates scalar tuples, e.g., {(a, {(b), (d)})}

Don't pay attention…

Thursday, February 24, 2011
10:17 AM

Don't pay attention…

...to that man behind the curtain.

Much of the power of Pig comes from **user-defined functions** (UDFs).

In fact, as a user of much more powerful functional programming systems, I think Pig has remarkably few builtin functions (and I miss some of the builtins in FP)

For this lecture, I will concentrate on the power of Pig itself,

So I will start with data that has already been massaged by UDFs into an appropriate form.

A general caveat

Thursday, February 24, 2011
10:25 AM

A general caveat

In Perl, we say

"There is always another way to do it."

In Pig, I have learned

"There is always an easier way to do it."

My first conception of how to solve each of the following problems was **extremely inefficient**, and only by **changing my entire approach** could I unleash the power of Pig.

… even though I am expert in traditional dataflow and functional programming…

So,

Pig programs are **really short,** but
if you get on the **wrong track**,
expect to **throw the entire program away**.
You have been warned!

What's a good Pig problem?

Thursday, February 24, 2011
10:28 AM

What's a good Pig problem?

- Huge dataset.
- Tiny results (needle-in-haystack search).
- Combinatorial search (e.g., checking through lots of options).

A few typical Pig applications

Thursday, February 24, 2011
10:27 AM

A few typical Pig applications

Social flow analysis: co-location/correlation of blog posts/tweets/facebook posts about specific products or services.

Incident detection: reporting security incidents through correlation between security log entries.

Detecting social buzz

Thursday, February 24, 2011
12:15 PM

Detecting social buzz

Input: a set of post events in social networks.

Output: a list of times at which there are flurries of activity around specific topics.

The man behind the curtain

Thursday, February 24, 2011
12:18 PM

Input data for our Pig program

time	user	topic	opinion
2839203081L	couch	Honda	positive

where

time is the time of the post in seconds since Jan 1 1970 00:00:00 GMT (a.k.a. "the epoch")
user is the identity of the poster.
topic is what they posted about.
opinion is 'positive', 'neutral', or 'negative', as estimated by a natural-language parser UDF.

Our problem: detect negative buzz about Hondas.

Initial filtering

Thursday, February 24, 2011
12:24 PM

Our schema is now

```
posts: {time:long, user:chararray, topic:chararray,
opinion:chararray}
```

let's concentrate on the negative posts:

```
hates = FILTER posts BY topic=='Honda' and
opinion=='negative';
```

and let's remove those fields, because we don't need them now:

```
hate2 = FOREACH hates GENERATE time, user;
```

Our schema is now

```
hate2: {time:long, user:chararray}
```

Binning time

Thursday, February 24, 2011
12:30 PM

Our schema is now

```
hate2: {time:long, user:chararray}
```

We are interested in "buzz" events, which are negative posts by multiple users at roughly the same time.

First, we select a time interval (e.g., one hour=3600 seconds) and create timestamps every hour:

```
clusters = FOREACH posts GENERATE time-time%3600 as
time, user;
```

We are only interested in how many distinct users are involved, so

```
cluster2 = DISTINCT clusters;
```

So far, our schema hasn't changed:

```
cluster2: {time:long, user:chararray};
```

Grouping by time

Thursday, February 24, 2011
12:40 PM

Now we group behavior by time:

```
buzzes = GROUP cluster2 BY time;
```

Now our schema is

```
buzzes: {group:long,
cluster2:{time:long, user:chararray}}
```

Let's count the number of distinct users:

```
buzz2 = FOREACH buzzes GENERATE group
as time, COUNT(cluster2) as hits;
```

Now our schema is:

```
buzz2: {time:long, hits:int};
```

Finally, we filter for lotsa hits:

```
buzz3 = FILTER buzz2 BY hits>1000;
```

And report those times:

```
DUMP buzz3;
```

Incident detection

Thursday, February 24, 2011
10:43 AM

Security incident detection:

- Input: log entries from a set of machines.
- Output: suspicious behaviors detected.
- UDF: Parse log entries, select failed password attempts.
- Suspicious pattern to detect:
  - bot password cracking attack,
  - many source locations,
  - one user,
  - many login failures,
  - distributed in time.
- Very difficult to detect by standard log scanning methods:
  - Distributed cracking attempts,
  - long time-scale,
  - huge data scale.

The man behind the curtain…

Thursday, February 24, 2011
10:50 AM

Input data for our Pig program

time	user	action	status	subnet
2839203081L	couch	login	success	130.64.23/24

where

time is the time of the incident in seconds since Jan 1 1970 00:00:00 GMT (a.k.a. "the epoch")
user is the account in question.
action is what they did.
status is success or failure.
subnet is the address block on the internet from which they tried (grouping related addresses together).
(In practice, the log would be **collected** using HDFS!)

We start with something like:

```
log = LOAD 'log' USING MyLogParser()
  AS (time:long, user:chararray,
  action:chararray, status:chararray,
  subnet:chararray);
```

Starting our program

Thursday, February 24, 2011
10:56 AM

Now we have the schema

```
DESCRIBE log;
log: {time :long, user :chararray, action :chararray,
status :chararray, subnet :chararray}
```

Next, filter out things we aren't interested in

```
failures = FILTER log BY action=='login'
  and status=='failure';
```

Then, omit the action and status fields (by **projection**), because we don't need them anymore.

```
failure2 = FOREACH failures GENERATE user, subnet;
```

Now our schema is

```
DESCRIBE failure2;
failure2: {user :chararray, subnet :chararray}
```

General principle: filter as small as possible, as early as possible.

Grouping

Thursday, February 24, 2011
11:50 AM

Now we have the schema:

```
failure2: {user:chararray, subnet:chararray}
```

But, an incident is specific to one login name, so we group by that:

```
users = GROUP failure2 by user;
```

Now our schema is

```
users: {group: chararray, failure2: {user:chararray,
subnet:chararray}}
```

Now we don't need the user in the subsets, so let's project that out:

```
user2 = FOREACH users GENERATE group as user,
failure2.subnet as locations;
```

Now our schema is:

```
user2: {user:chararray, locations:{subnet:chararray}}
```

Final filtering

Thursday, February 24, 2011
12:13 PM

Now our schema is:

```
user2: {user: chararray, locations:
{subnet: chararray}}
```

A bot attack is a lot of failed attempts from many different locations. First, quantify "a lot".

```
lots = FILTER user2 by
COUNT(locations)>1000;
```

Then quantify how many locations:

```
user3 = FOREACH user2 {
  places = DISTINCT locations;
  GENERATE user, COUNT(places) as
  places;
}
user4 = FILTER user3 by places>100;
```

Tell the administrator that these users are under attack by bots:

```
DUMP user4;
```

Did they get in?

Thursday, February 24, 2011
1:11 PM

Second question: did the attackers get in?
A signature of a successful distributed attack is a successful login from somewhere the user doesn't frequent, **after** several unsuccessful login attempts from the same place.
Now we really have to work!

Step 1: start over!

```
log = LOAD 'log' USING MyLogParser()
  AS (time:long, user:chararray,
  action:chararray, status:chararray,
  subnet:chararray);
```

We do not have to retype this. It is an alias, not a command!

Select the users we know are under attack: Remember that the schema for that list is

```
user4: {user:chararray, places:int}
```

First we make a list of users under attack:

```
attacked = FOREACH user4 GENERATE
user;
```

And then we select all records for those users via an inner join:

```
vulnerable = JOIN log BY user,
attacked BY user;
```

And we are interested only in login attempts:

```
vulnerable2 = FILTER vulnerable by
```

```
log::action=='login';
```

And then strip the joined field off (it's a duplicate), along with the action field (it's redundant now).

```
vulnerable3 = FOREACH vulnerable2
GENERATE log::time as time, log::user
as user, log::status as status,
log::subnet as subnet;
```

Now our schema is

```
vulnerable3: {time:long,
user:chararray, status:chararray,
subnet:chararray}
```

We used the inner join to delete data.

Determining valid login sites

Thursday, February 24, 2011
3:57 PM

Now our schema is

```
vulnerable3: {time:long, user:chararray,
status:chararray, subnet:chararray}
```

Let's filter out subnets from which the user regularly logs in correctly. Let's make a whitelist.

```
correct = FILTER vulnerable3 BY status=='success';
correct2 = FOREACH correct GENERATE user, subnet;
good = GROUP correct2 BY (user,subnet);
```

Now schema is

```
good: {group: (user:chararray, subnet:chararray),
correct2: {user:chararray, subnet:chararray}}
```

Let's count the correct logins:

```
whitelist = FOREACH good GENERATE
group.user,group.subnet,COUNT(correct2) as count;
whitelist2 = FILTER whitelist by counter>100;
```

And throw them out of the log!

Pattern of the whole program

Thursday, February 24, 2011
5:23 PM

Pattern of the whole program

Determine vulnerable users, delete other records.
Compute whitelist of sites, delete from user records.
For each site, compare the earliest successful login with the number of unsuccessful logins before it.
If this is >0, it is likely that the account is cracked.

Pig cliches

Wednesday, February 23, 2011
7:07 PM

Pig Cliches

Some operations in Pig are so common that they have become cliches.

Flattening collapse: in flattening nested structures, empty tuples and bags disappear.

Implied cross-products: flattening two bags in one tuple forms the cross-product between them: all possible permutations taken two at a time.

Generate-and-test: generate lots of options; then filter for the ones that you want.

Inner bag manipulation: can filter and otherwise manipulate bags contained within others.

I'll demonstrate some of these through actual problem solving.