Two kinds of Map/Reduce programming
   In Java/Python
   In Pig+Java
Today, we'll start with Pig
Recall from last time

We have class tomorrow (Monday's schedule)!

Recall from last time

Map/Reduce consists of

A Map phase in which we do something to each record.

A Reduce phase where we combine results.

We only write what to do to each element, and how to combine a list of things.

The Map/Reduce framework does the rest.
Understanding gmail

Your mail messages are stored "in the cloud". Each message has a unique ID that serves as its key.
There is no concept of message locality; two "adjacent" messages in your mailbox may be stored across the world from one another.
The convenient human concept of locality (i.e., that messages are a stream) is constructed by MapReduce.
Map selects messages of interest
Reduce concatenates them into a stream.
Common Map/Reduce algorithms

Filter: map selects, reduce combines.
Sort: map selects, reduce merge sorts.
Construct: map creates new data, reduce reports what was done.
Understanding limits of AppEngine MapReduce

You might have noticed that there are severe limits on what you can do with MapReduce directly inside AppEngine:

- No choice on reducer (combine and eliminate duplicates).
- Very limited map capabilities.

Why?
- If you write a mapper, it has to be propagated to each map node.
- This is moving code, not data.
- This is relatively expensive and somewhat dangerous.
- Otherwise, you cannot inject code into the cloud.
- This has security and other implications.
There are two distinct reasons for Map/Reduce

**Retrieval**: retrieving data that is already stored.

**Transformation**: summarizing data in a new and more useful form.

Actually, the two prevalent forms of Map/Reduce have different strengths:

- The (implicit) MR in Google AppEngine is intended for data **retrieval**.
- The (explicit) MR in Hadoop is intended mainly for data **transformation**.

(Hadoop presumes that you will eventually retrieve data by another mechanism!)
Myths and Realities of MR

Google MR is fast because
  Map and Reduce are pre-coded.
  There is no need to propagate code.
Hadoop MR has a very slow startup time because
  Must propagate jarfile for Map, Combine, Reduce classes to each Hadoop node.
  Up to 30 seconds for a large array.
  So, basically no one uses it for real time queries.
So, what is Hadoop MR good for?

Often, can compute and pre-store results of commonly needed queries. These run much faster on Hadoop than serially. Then we query the results normally.
Common uses for Hadoop Map/Reduce

Scanning system logfiles for patterns that indicate security problems.
Scanning social networks for mentions of products, brands, or services.
Scanning data warehouses for overall economic trends or patterns.

In all three cases:
Data is huge.
Processing time is prohibitive.
Results are small.
A data restructuring subsystem of Hadoop
With its own language Pig Latin with SQL-like syntax.

But

SQL is intended for queries.
Input is a database.
Output is a table.

Pig Latin is intended for restructuring data.
Input is a set of distributed files.
Output is another distributed file.
Pig runs in two modes:
   In local mode, filenames refer to local files.
   In MapReduce mode, filenames refer to distributed (HDFS) files.
We'll run in local mode for now, but
We'll imagine running in MapReduce mode, and
Our programs will be identical in either case.
Good news and bad news

Tuesday, February 22, 2011
12:33 PM

Good news:
  Pig handles the nuisance programming required for data restructuring,
  Pig Latin programs are much shorter than Java programs
  Map/Reduce is implicit in Pig Latin programs.

Bad news:
  You have to learn a whole new language with a rather non-intuitive syntax.
• Register the tutorial JAR file so that the included user-defined functions can be called in the script.
REGISTER ./tutorial.jar;
• Use the PigStorage function to load the excite log file (excite.log or excite-small.log) into the “raw” bag as an array of records with the fields user, time, and query. raw = LOAD 'excite.log' USING PigStorage('\t') AS (user, time, query);
• Use the FILTER-BY construction to call the NonURLDetector user-defined function (Java class in tutorial.jar) to remove records if the query field is empty or a URL.
clean1 = FILTER raw BY org.apache.pig.tutorial.NonURLDetector(query);
• Call the ToLower user-defined function to change the query field to lowercase.
clean2 = FOREACH clean1 GENERATE user, time, org.apache.pig.tutorial.ToLower(query) as query;
• The excite query log timestamp format is YYMMDDHHMMSS. Call the ExtractHour user-defined function to extract the hour (HH) from the time field.
houred = FOREACH clean2 GENERATE user, org.apache.pig.tutorial.ExtractHour(time) as hour, query;
• Call the NGramGenerator user-defined function to compose the n-grams of the query.
ngramed1 = FOREACH houred GENERATE user, hour, flatten(org.apache.pig.tutorial.NGramGenerator(query)) as ngram;
• Use the DISTINCT command to get the unique n-grams for all records.
ngramed2 = DISTINCT ngramed1;
• Use the GROUP command to group records by n-gram and hour.
hour_frequency1 = GROUP ngramed2 BY (ngram, hour);
• Use the COUNT function to get the count (occurrences) of each n-gram.
hour_frequency2 = FOREACH hour_frequency1 GENERATE flatten($0), COUNT($1) as count;
• Use the GROUP command to group records by n-gram only. Each group now corresponds to a distinct n-gram and has the count for each hour.
uniq_frequency1 = GROUP hour_frequency2 BY group::ngram;
• For each group, identify the hour in which this n-gram is used with a particularly high frequency. Call the ScoreGenerator user-defined function to calculate a "popularity" score for the n-gram.
uniq_frequency2 = FOREACH uniq_frequency1 GENERATE flatten($0),
Use the FOREACH-GENERATE command to assign names to the fields.

\[
\text{uniq\_frequency3} = \text{FOREACH uniq\_frequency2 GENERATE $1$ as hour, $0$ as ngram, $2$ as score, $3$ as count, $4$ as mean;}
\]

Use the FILTER-BY command to remove all records with a score less than or equal to 2.0.

\[
\text{filtered\_uniq\_frequency} = \text{FILTER uniq\_frequency3 BY score > 2.0;}
\]

Use the ORDER-BY command to sort the remaining records by hour and score.

\[
\text{ordered\_uniq\_frequency} = \text{ORDER filtered\_uniq\_frequency BY (hour, score);}
\]

Use the PigStorage function to store the results. The output file contains a list of n-grams with the following fields: **hour, ngram, score, count, mean**.

\[
\text{STORE ordered\_uniq\_frequency INTO '/tmp/tutorial-results'} \text{USING PigStorage();}
\]

Pasted from `<http://wiki.apache.org/pig/PigTutorial>`
Invoking Pig

Download the cloudera training VM (stable release 0.3.3)
Unpack into a local directory, e.g., cloudera-training-0.3.3
Download VMWare player (or equivalent).
Double-click on the file cloudera-training-0.3.3.vmx (which will invoke VMWare player)
Wait for it to boot linux.
   Login: hadoop
   Password: hadoop
Select Applications/Terminal from top menu
Make yourself a working directory:
   $ mkdir 150CPA
   $ cd 150CPA
Make yourself a file to read into pig
   $ vi junk.dat
   or
   $ emacs junk.dat
Run pig in local interactive mode
   $ pig -x local
grunt>
The REGISTER command

REGISTER ./something.jar
Make every class in the jarfile available on the command line.
(Note: only static functions are available, not methods.)
(Try doing this in SQL!)
Data types in Pig:

A **field** is a piece of data.

A **tuple** is a sequence of 1 or more fields in parentheses,

e.g. (f1,f2,f3).

A **bag** is a set of tuples, e.g., {(1,2),(4,5,6)}

A **relation** is an **outer bag**.

An **outer bag** can be referred to by a top-level alias.

An **inner bag** doesn't have an alias.
To make things as efficient as possible, Pig computes the results of an expression as **late as possible**. This is called **lazy evaluation**.

When you write a statement in Pig, it is not executed immediately, but instead, recorded as something to be executed later.

Thus, all variables in Pig are actually **aliases for instructions to be executed later**.
What Pig does "could be" done by SQL. But it would be incredibly complex SQL. Aliases allow us to write a Pig command in pieces. The result is still one huge (unreadable) command that gets executed, in the end.
Debugging in pig

DUMP <alias>;
prints a relation's contents as text, after computing it.

DESCRIBE <alias>;
prints its schema (metadata).

EXPLAIN <alias>;
prints an explanation of what Pig will do to get it.

ILLUSTRATE <alias>;
prints a sample run with a subset of data.
The LOAD command:
```pigscript
<alias> = LOAD '<filename>' USING PigStorage('\t') AS (<fieldname>:<type>, <fieldname>,...);
```
Read a text file, parse lines into fields at '\t', assign names to fields.
Example:
```pigscript
raw = LOAD 'excite.log' USING PigStorage('\t') AS (user, time, query);
```
The STORE command:
```pigscript
STORE <alias> INTO '<filename>' USING PigStorage();
```
Write contents of a relation into a file.
Example:
```pigscript
STORE same1 INTO '/tmp/tutorial-join-results' USING PigStorage();
```
Notes:
PigStorage is a class that reads and writes text files.
You can REGISTER your own class to do this.
The argument is the argument to the constructor of an instance.

Pasted from <http://wiki.apache.org/pig/PigTutorial>
Suppose that test.dat contains

Alva 20
George 30
Joe 10

And that we execute

$ pig -x local

grunt> test = LOAD 'test.dat' USING PigStorage(' ') AS (name, toys);
grunt> DUMP test;
... some diagnostic output removed...
(Alva,20)
(George,30)
(Joe,10)
grunt> DESCRIBE test;
test: {name: bytearray, toys: bytearray}
grunt> STORE test INTO 'out.dat' USING PigStorage();
grunt>

Note that:

Without further information, everything is a **bytearray**.
STORE creates a duplicate of the input file 'test.dat', space-delimited.
Aliases in Pig Latin

Each symbol is an alias that represents a relation to be computed.

A relation has **rows(records/tuples)** and **columns/fields**.

Columns/fields may be **named** or **unnamed**.

If they are unnamed, one may refer to them as $0, $1, $2....

They are always named when the relation is being constructed; you cannot go back and name them "later".

Relations cannot be changed after they are constructed; new relations can be created.
A statement in Pig represents a potential computation. Statements are used only when needed. Only a few statements actually perform computation:

- **STORE**: compute something and store it.
- **DUMP**: compute and print something for debugging purposes.

The rest store descriptions of computations to perform later!
Simple data processing statements
  FILTER-BY: keep only tuples satisfying some criteria.
  FOREACH-GENERATE: do something with each tuple.
  DISTINCT: eliminate duplicate tuples.
  ORDER: sort things into an order.
  LIMIT: only return n things where n is small.
  SAMPLE: randomly sample a subset!
  SPLIT: put tuples into different datasets.
  UNION: combine tuples from different datasets.
FILTER-BY: omit rows (records) not matching an expression

<alias> = FILTER <alias2> BY <expression>;
makes <alias> the result of omitting rows from
<alias2> not satisfying <expression>.
Expression can be any boolean expression of the columns or, optionally, a user-defined Java function with a boolean value.

Example:
rich = FILTER people BY salary>0;
Let the alias 'rich' to refer to the records/tuples in relation 'people' where the column salary>0; ignore all other records.

Note: destination must be different than source, because this is an implied Map/Reduce; symbols are mapped to HDFS filenames.
grunt> test = LOAD 'test.dat' USING PigStorage(' ') AS (name:chararray, toys:int);
grunt> DUMP test;
... Success!!
(Alva,20)
(George,30)
(Joe,10)
grunt> jest = FILTER test BY toys>10;
grunt> DUMP jest;
... Success!!
(Alva,20)
(George,30)
grunt>
FOREACH-GENERATE

<alias> = FOREACH <alias2> GENERATE <field expression> [AS <field name>], ...;

   Makes <alias> correspond to the output of the specified mapping operation
   upon <alias2>.

   Mappings can be user-defined Java functions.

Example:

   clean2 = FOREACH clean1 GENERATE user, time,
            org.apache.pig.tutorial.ToLower(query) as query;

   For each record (user, time, query), replace it with a record containing the
   user, the time, and the lowercase version of the query.
grunt> DUMP test;
...Success!!
(Alva,20)
(George,30)
(Joe,10)
grunt> pest = FOREACH test GENERATE name, toys+1 as toys;
grunt> DUMP pest;
...Success!!
(Alva,21)
(George,31)
(Joe,11)
grunt>
Splitting a dataset into subsets

\texttt{SPLIT <relation> into <relation2> if <expression2>, <relation3> if <expression3>, ...;}

Equivalent with multiple FILTER-BY statements.

Example

\texttt{SPLIT employees into salaried if salary>0, unsalaried if salary==0;}

Putting together split relations

\texttt{<relation> = UNION <relation2>, <relation3>, ...;}

Example:

\texttt{stuff = UNION salaried, unsalaried;}
grunt> SPLIT test INTO greater if toys>15, lesser if toys<=15;
grunt> DUMP test;
... Success!!
(Alva,20)
(George,30)
(Joe,10)
grunt> DUMP greater;
... Success!!
(Alva,20)
(George,30)
grunt> DUMP lesser;
... Success!!
(Joe,10)
grunt>
But...!

A field can be anything, including

A **primitive data type** (e.g., int, float, chararray).

**Another tuple**, e.g., (George, 5)

An **inner bag** of tuples, e.g.,

{ (George, 5), (Beth, 10) }

And types can be mixed within one relation!

So, it is perfectly reasonable to have a relation

DUMP R; -- how you debug in pig!

(George, 5)

(Beth, {2,3,4})

({Beth,George},17)

which contains

- a tuple of a chararray and an int
- a tuple of a chararray and a bag of ints.
- a tuple of a bag of chararrays and an int
Some very important caveats

Bag processing is done in random order. One cannot predict it or even serialize it. It can be done in parallel!
Structural transformations

Several operations change the structure of data:

Functions:
- **COUNT**(bag or tuple) returns the number of tuples in a bag, or the number of fields in a tuple.
- **FLATTEN**(bag or tuple) removes hierarchy from a bag or tuple.

Statements:
- **GROUP-BY**: put similar things into bags.
- **JOIN-BY**: combine two relations via a common key.
GROUP-BY

Input: a bag and a sequence of fields
Output: a new bag in which
The selected sequence of fields remains.
Remaining fields are grouped into a bag.

E.g.,

DUMP X;
(Alex,1)
(Alex,2)
(George,5)
Y = GROUP X by $0;
DUMP Y;
(Alex, {(Alex,1),(Alex,2)})
(George,{(George,5)})
grunt> junk = LOAD 'junk.dat' USING PigStorage('\t') AS (name, toys);
grunt> DUMP junk;
(Alva,3)
(George,5)
(Fred,7)
(Alva,10)
(Alva,12)
grunt> stuff = GROUP junk BY name;
grunt> DUMP stuff;
(Alva,{{Alva,3},{Alva,10},{Alva,12}})
(Fred,{{Fred,7}})
(George,{{George,5}})
grunt> counts = FOREACH stuff GENERATE group, COUNT(junk);
grunt> DUMP counts;
... Success!!
(Alva,3L)
(Fred,1L)
(George,1L)
grunt> sums = FOREACH stuff GENERATE group, SUM(junk.toys);
grunt> DUMP sums;
... Success!!
(Alva,25.0)
(Fred,7.0)
(George,5.0)
JOIN-BY: create a new relation via common keys

<alias> = JOIN <alias2> BY <fieldname2>, <alias3> BY <fieldname3>;

Creates an **inner join**; data that doesn't match is omitted.
grunt> names = LOAD 'names.dat' USING PigStorage(' ') as (name, species);
grunt> actions = LOAD 'actions.dat' USING PigStorage(' ') as (species, action);
grunt> DUMP names;
... Success!!
(Bill,Cat,)
(Morris,Cat)
(Ed,Horse)
(Tux, Penguin)
grunt> DUMP actions;
... Success!!
(Horse,neighs)
(Cat,purrs)
(Cat,scratches)
(Dog,barks)
grunt> joined = JOIN names BY species, actions BY species;
grunt> DUMP joined;
... Success!!
(Bill,Cat,Cat,purrs)
(Bill,Cat,Cat,scratches)
(Morris,Cat,Cat,purrs)
(Morris,Cat,Cat,scratches)
(Ed,Horse,Horse,neighs)
grunt> acts = FOREACH joined GENERATE $0 as name, $3 as action;
grunt> DUMP acts;
... Success!!
(Bill,purrs)
(Bill,scratches)
(Morris,purrs)
(Morris,scratches)
(Ed,neighs)
An inner join omits records for which there is no match. There are three other kinds of (binary) joins:

- **LEFT OUTER**: left record included even if no match.
- **RIGHT OUTER**: right record included even if no match.
- **FULL OUTER**: both sides: record included even if no match
Detailed example of outer joins

Wednesday, February 23, 2011
2:41 PM

grunt> out = JOIN names by species LEFT, actions by species;
grunt> DUMP out;
... Success!!
(Bill,Cat,Cat,purrs)
(Bill,Cat,Cat,scratches)
(Morris,Cat,Cat,purrs)
(Morris,Cat,Cat,scratches)
(Ed,Horse,Horse,neighs)
(Tux,Penguin,,)
grunt> out = JOIN names by species RIGHT, actions by species;
grunt> DUMP out;
... Success!!
(Bill,Cat,Cat,purrs)
(Bill,Cat,Cat,scratches)
(Morris,Cat,Cat,purrs)
(Morris,Cat,Cat,scratches)
(,,Dog,barks)
(Ed,Horse,Horse,neighs)
grunt> out = JOIN names by species FULL, actions by species;
grunt> DUMP out;
... Success!!
(Bill,Cat,Cat,purrs)
(Bill,Cat,Cat,scratches)
(Morris,Cat,Cat,purrs)
(Morris,Cat,Cat,scratches)
(,,Dog,barks)
(Ed,Horse,Horse,neighs)
(Tux,Penguin,,)
grunt>
Flattening is a strange operation
   Different for tuples than for bags.
   For a tuple, removes all parenthesis.
   For a bag, expands inline bags to groups of rows.
Detailed example of flattening

Wednesday, February 23, 2011
3:08 PM

grunt> DUMP x;
(1,(2,(3)))
(4,(5,(6)))
grunt> y = FOREACH x GENERATE $0,FLATTEN($1);
grunt> DUMP y;
(1,2,3)
(4,5,6)
grunt> DUMP z;
(1,{(2),(3)})
(4,{(5)})
(6,{(7)})
grunt> w = FOREACH z GENERATE $0,FLATTEN($1);
grunt> DUMP w;
(1,2)
(1,3)
(4,5)
(6,7)