Apache Pig

Jonathan Coveney, @jco
Data Systems Engineer, Twitter

Why do we need Pig?

- Writing native Map/Reduce is hard
  - Difficult to make abstractions
  - Extremely verbose
    - 400 lines of Java becomes < 30 lines of Pig
  - Joins are very difficult
    - A big motivator for Pig
  - Chaining together M/R jobs is tedious
  - Decouples logic from optimization
    - Optimize Pig and everyone benefits

Basically, everything about Java M/R is painful

Understanding a basic Pig script

This is a relation (NOT a variable)

Loads a file into a relation, with a defined schema

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
```

Filters out all rows that don't fulfill the predicate

```
students_filtered = FILTER students BY age >= 20;
```

Loads another file

```
dept_info = LOAD ‘dept_info.txt’ as (dept:chararray, credits:int);
```

Equi-joins two relations on the dept column

```
students_dept = JOIN students BY dept, dept_info BY dept;
```

```
students_proj = FOREACH students_dept GENERATE
    students::first as first, students::last as last,
    students::dept as dept, dept_info::credits as credits;
```

Chooses what columns we care about and renames them

These are all relations! They represent rows of data, each row organized into columns

Basic Pig script (cont)

This makes a jar of User Defined Functions (UDFs) available for use

```
register ClassesUdfs.jar;
```

Adds a column the the relation, the result of using a UDF

```
students_hours = FOREACH students_proj GENERATE *,
                    CalculateSemestersFromCredits(credits) as semesters;
```

Globally orders relation by the semesters column, in descending order

```
students_ordered = ORDER students_hours BY semesters DESC;
```

Stores the relation using a custom store function (could even store to a database)

```
STORE students_ordered INTO ‘students_hours’ USING CustomStudentsStoreFunc();
```

An even simpler example: Word count

- Can’t have a tutorial on anything Hadoop related without word count

```
text = LOAD 'text' USING TextLoader();
tokens = FOREACH text GENERATE FLATTEN(TOKENIZE($0)) as word;
wordcount = FOREACH (GROUP tokens BY word) GENERATE
    group as word,
    COUNT_STAR($1) as ct;
```

← Loads each line as one column

What Pig can do for you

What is Pig?

- Pig is a **scripting language**
  - No compiler
  - Rapid prototyping
  - Command line prompt (grunt shell)
- Pig is a **domain specific language**
  - No control flow (no if/then/else)
  - Specific to data flows
    - Not for writing ray tracers
    - For the distribution of a pre-existing ray tracer

What ISN'T Pig?

- A general framework for all distributed computation
  - Pig is MapReduce! Just easier
- A general purpose language
  - No scope
  - Minimal "variable" support
  - No control flow

What CAN Pig do?

- Ad hoc analysis
- Move and transform data (ETL)
  – Good integration with many systems
- Build regular reports
- Machine learning
- …all on Terabytes and Petabytes of data!

What CAN'T Pig do?

- Iterative ML algorithms
  - Graph algorithms are generally slow
- Transcend Hadoop
  - Pig is still Hadoop
- Algorithms that converge
  - Without control flow can't check convergence
  - PigServer (Jython/Java) gives some control flow

What are my other options?

- Hive (SQL)
  - Pros
    - Leverages well-known syntax
    - JDBC means integration is easy
  - Cons
    - Complex transformations can be unwieldy
    - Less fine grained control

What are my other options? (cont)

- Cascading (Java API)
  - Various DSLs
    - Scalding (Scala API)
    - Cascalog (Clojure API)
  - Pros
    - All one language
    - The DSLs make Cascading simpler and more powerful
    - Easier to make converging algorithms
  - Cons
    - No optimization
    - A lower-level API

The Pig Object Model

Relations

- Fundamental building block
- Analogous to a **table**, not a variable

```
students = load 'students.txt' as (first:chararray, last:chararray, age:int,
dept:chararray);
```

Defines a relation

A relation, as it appears in Excel

first	last	age	major
Willia	Cracknell	18	CS
Francesco	Corraro	21	English
Lino	Feddes	22	History
Wes	Knill	23	EE
Ellyn	Meyerhoefer	18	English
Warner	Caminita	24	Psych
Lucius	Orlosky	20	History
Del	Graefe	20	CS
Douglass	Adelizzi	23	CS
Lesley	Kellywood	20	Biology

Loading a file

- Save file as tab delimited text without headings
- Boot up Pig

```
$ bin/pig –x local
2012-08-15 14:29:04,968 [main] INFO  org.apache.pig.Main - Apache Pig version 0.11.0-SNAPSHOT (r1373633) compiled Aug 15 2012,
14:20:59
2012-08-15 14:29:04,968 [main] INFO  org.apache.pig.Main - Logging error messages to: /Users/jcoveney/workspace/berkeley_pig/
pig_1345066144966.log
2012-08-15 14:29:04,983 [main] INFO  org.apache.pig.impl.util.Utils - Default bootup file /Users/jcoveney/.pigbootup not found
2012-08-15 14:29:05,138 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file
system at: file:///
grunt> students = LOAD 'students.txt’ USING PigStorage(‘\t’);
grunt> DUMP students;
```

A detour: firing up Pig

Signifies local mode

The version of Pig currently running

```
$ bin/pig –x local
2012-08-15 14:29:04,968 [main] INFO  org.apache.pig.Main –
Apache Pig version 0.11.0-SNAPSHOT (r1373633) compiled Aug 15 2012, 14:20:59
2012-08-15 14:29:04,968 [main] INFO  org.apache.pig.Main - Logging error messages to:
/Users/jcoveney/workspace/berkeley_pig/pig_1345066144966.log
2012-08-15 14:29:04,983 [main] INFO  org.apache.pig.impl.util.Utils - Default bootup file /
Users/jcoveney/.pigbootup not found
2012-08-15 14:29:05,138 [main] INFO
org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop
file system at: file:///
```

Errors will be logged here

Means we're working on local files

- Local mode is great for iterating quickly
  - Much faster than a cluster

Loading data

Declares the relation for the data

USING specifies the function that will load the data

```
grunt> students = LOAD 'students.txt' USING PigStorage('\t');
grunt> DUMP students;
```

Refers to a Java function which knows how to load rows from text

Sets delimiter

DUMP keyword streams the relation to the console

Loads the data in this file

Output:

```
(Willia,Cracknell,18,CS)
(Francesco,Corraro,21,English)
(Lino,Feddes,22,History)
(Wes,Knill,23,EE)
(Ellyn,Meyerhoefer,18,English)
(Warner,Caminita,24,Psych)
(Lucius,Orlosky,20,History)
(Del,Graefe,20,CS)
(Douglass,Adelizzi,23,CS)
(Lesley,Kellywood,20,Biology)
```

Each row is a Tuple

It's the same as the excel file!

first	**last**	**age**	**major**
Willia	Cracknell	18	CS
Francesco	Corraro	21	English
Lino	Feddes	22	History
Wes	Knill	23	EE
Ellyn	Meyerhoefer	18	English
Warner	Caminita	24	Psych
Lucius	Orlosky	20	History
Del	Graefe	20	CS
Douglass	Adelizzi	23	CS
Lesley	Kellywood	20	Biology

Projection (aka FOREACH)

```
students = LOAD ‘students.txt’;
pruned = FOREACH students GENERATE $0, $2;
dump pruned;
```

Here we define what each row in the new relation will be

Signifies the relation to transform

We can refer to columns positionally

- Foreach means “do something on every row in a relation”
- Creates a new relation
- In this example, pruned is a new relation whose columns will be first name and age
- With schemas, can also use column aliases

Schemas and types

- Schema-less analysis is useful
  - Many data sources don't have clear types
- But schemas are also very useful
  - Ensuring correctness
  - Aiding optimization

Second half defines expected schema

```
students = LOAD 'students.txt'
          as (first:chararray, last:chararray, age:int, dept:chararray);
```

The first half stays the same

- Schema gives an alias and type to the columns
  - Absent columns will be made null
  - Extra columns will be thrown out

We have the types, now use them!

- This does the same as the previous example:

```
students = LOAD ‘students.txt’ as (first:chararray, last:chararray, age:int, dept:chararray);
pruned = FOREACH students GENERATE first, age;
DUMP pruned;
```

- *DESCRIBE* prints the schema of a relation

```
grunt> describe pruned;
pruned: {first: chararray,last: chararray}
```

The relation being described

The schema of the relation

Types and Schemas

Schemas vs. Types

- Schemas
  - A description of the types present
  - Used to help maintain correctness
  - Generally not enforced once script is run
- Types
  - Describes the data present in a column
  - Generally parallel Java types

Type Overview

- Pig has a nested object model
  - Nested types
  - Complex objects can contain other object
- Pig primitives mirror Java primitives
  - String, Int, Long, Float, Double
  - DataByteArray wraps a byte[]
  - Working to add native DateTime support, and more!

Complex types: Tuples

- Every row in a relation is a Tuple
- Allows random access
- Wraps ArrayList<Object>
- Must fit in memory
- Can have a Schema, but is not enforced
  - Potential for optimization

Complex types: Bags

- Pig’s only spillable data structure
  - Full structure does not have to fit in memory
- Two key operations
  - Add a Tuple
  - Iterate over Tuples
- No random access!
- No order guarantees
- The object version of a relation
  - Every row is a Tuple

Complex types: Maps

- Wraps HashMap<String, Object>
  - Keys must be Strings
- Must fit in memory
- Can be cumbersome to use
  - Value type poorly understood in script

More operators!

Filter

- A predicate is evaluated for each row
  - If false, the row is thrown out

```
students = LOAD ‘students.txt’ as (first:chararray, last:chararray, age:int, dept:chararray);
under_20  = FILTER students BY age < 20;

(Willia,Cracknell,18,CS)
(Ellyn,Meyerhoefer,18,English)
```

- Supports complicated predicates
  - Boolean logic
  - Regular expressions
  - See Pig documentation for more

Grouping

- Abstractly, GROUPing creates a relation with unique keys, and the associated rows
- Example: how many people in our data set are in each department?

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
students_grouped = GROUP students BY dept;
DESCRIBE students_grouped;

students_grouped: {group: chararray,
                   students: {(first: chararray,last: chararray,age: int,dept: chararray)}}
```

Specifies the key to group on

First column in post-group relation is always named "group," and is identical to the group key. Relation has one row per unique group key value

Second column is always named after grouped relation, and is a Bag

Grouping gives the rows corresponding to the key, so the schema of this Bag is the same as the relation we grouped (ie students)

Visualizing the group

```
(CS,{(Willia,Cracknell,18,CS),(Del,Graefe,20,CS),(Douglass,Adelizzi,23,CS)})
(EE,{(Wes,Knill,23,EE)})
(Psych,{(Warner,Caminita,24,Psych)})
(Biology,{(Lesley,Kellywood,20,Biology)})
(English,{(Francesco,Corraro,21,English),(Ellyn,Meyerhoefer,18,English)})
(History,{(Lino,Feddes,22,History),(Lucius,Orlosky,20,History)})
```

One row per unique key →

↑ Every row associated with that unique key

An alternate visualization

- Bags contain rows, so let's print it as such

students_grouped:
```
({(Willia,Cracknell,18,CS),
  (Del,Graefe,20,CS),
  (Douglass,Adelizzi,23,CS)})
({(Wes,Knill,23,EE)})
({(Warner,Caminita,24,Psych)})
({(Lesley,Kellywood,20,Biology)})
({(Francesco,Corraro,21,English),
  (Ellyn,Meyerhoefer,18,English)})
({(Lino,Feddes,22,History),
  (Lucius,Orlosky,20,History)})
```

students:
```
(Willia,Cracknell,18,CS)
(Del,Graefe,20,CS)
(Douglass,Adelizzi,23,CS)
(Wes,Knill,23,EE)
(Warner,Caminita,24,Psych)
(Lesley,Kellywood,20,Biology)
(Francesco,Corraro,21,English)
(Ellyn,Meyerhoefer,18,English)
(Lino,Feddes,22,History)
(Lucius,Orlosky,20,History)
```

They are the same!

Using GROUP: an example

- Goal: what % does each age group make up of the total?

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
students_grp = GROUP students BY age;
students_ct = FOREACH students_grp GENERATE group as age, COUNT_STAR(students) as ct;
students_total = FOREACH (GROUP students_ct ALL) generate SUM(students_ct.ct) as total;
students_proj = FOREACH students_join GENERATE
   students_ct::age as age,
   (double)students_ct::ct / (long)students_total.total as pct;
```

```
(18,0.2)
(20,0.3)
(21,0.1)   ←   It works! But how?
(22,0.1)
(23,0.2)
(24,0.1)
```

Understanding GROUP

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);

students_grp = GROUP students BY age;
```

Column "group" is the key we group on

```
students_ct = FOREACH students_grp GENERATE group as age, COUNT_STAR(students) as ct;
```

COUNT_STAR gives us the number of elements in the Bag "students," which is the number of rows with the same key

Transformations can be nested

```
students_total = FOREACH (GROUP students_ct ALL)
    GENERATE SUM(students_ct.ct) as total;
```

SUM UDF takes a Bag and returns the sum of the contents

This syntax returns a Bag with just the specified column

ALL key means resulting relation will be one row, the group column will be "all," and the second column will be every row in original relation

```
students_proj = FOREACH students_join GENERATE
    students_ct::age as age,
   (double)students_ct::ct / (long)students_total.total as pct;
```

Pig follows Java's math, so we cast for a decimal percent

This is a "scalar projection" of a relation. If a relation has one row with one value, we can cast it to that value.

Groups: a retrospective

- Grouping does not change the data
  - Reorganizes it based on the given key
  - Can group on multiple keys
- First column is always called group
  - A compound group key will be a Tuple (“group”)whose elements are the keys
- Second column is a Bag
  - Name is the grouped relation
  - Contains every row associated with key

FLATTENing

- Flatten is the opposite of group
- Turns Tuples into columns
- Turns Bags into rows

Flattening Tuples

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
students_grouped = GROUP students BY (age, dept);
students_ct = FOREACH students_grouped GENERATE group, COUNT(students) as ct;

DESCRIBE students_ct;
students_ct: {group: (age: int,dept: chararray),ct:long}

dump students_ct;
((18,CS),1)
((18,English),1)
((20,CS),1)
((20,Biology),1)
((20,History),1)
((21,English),1)
((22,History),1)
((23,CS),1)
((23,EE),1)
((24,Psych),1)
```

We are grouping on multiple keys

Irrespective of the number of group keys, there is always one first column, and it is called "group"

This is the schema of a Tuple. students_ct has TWO columns, one of which is a Tuple

This is how a Tuple looks. Flatten let's us un-nest the columns it contains

Flattening Tuples (cont)

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
students_grouped = GROUP students BY (age, dept);
students_ct = FOREACH students_grouped GENERATE FLATTEN(group) as (age, dept),
                                                COUNT(students) as ct;

DESCRIBE students_ct;
students_ct: {age: int,dept: chararray, ct:long}

dump students_ct;
(18,CS,1)
(18,English,1)
(20,CS,1)
(20,Biology,1)
(20,History,1)
(21,English,1)
(22,History,1)
(23,CS,1)
(23,EE,1)
(24,Psych,1)
```

Flatten un-nests the compound key

The same re-aliasing statement as before works when working with Tuples

The columns have been brought down a level. students_ct now has three columns

The values are the same, but now there are two columns instead of a Tuple

Flattening Bags

- Syntax is the same as Flattening Tuples, but the idea is different
- Tuples contain columns, thus flattening a Tuple turns one column into many columns
- Bags contain rows, so flattening a Bag turns one row into many rows

Returning to an earlier example

```
students = LOAD ‘students.txt’ as (first:chararray, last:chararray, age:int,
dept:chararray);
students_grouped = GROUP students BY dept;
students_proj = FOREACH students_grouped GENERATE students;
```

students_grouped:

```
({(Willia,Cracknell,18,CS),
  (Del,Graefe,20,CS),
  (Douglass,Adelizzi,23,CS)})
({(Wes,Knill,23,EE)})
({(Warner,Caminita,24,Psych)})
({(Lesley,Kellywood,20,Biology)})
({(Francesco,Corraro,21,English),
  (Ellyn,Meyerhoefer,18,English)})
({(Lino,Feddes,22,History),
  (Lucius,Orlosky,20,History)})
```

students:

```
(Willia,Cracknell,18,CS)
(Del,Graefe,20,CS)
(Douglass,Adelizzi,23,CS)
(Wes,Knill,23,EE)
(Warner,Caminita,24,Psych)
(Lesley,Kellywood,20,Biology)
(Francesco,Corraro,21,English)
(Ellyn,Meyerhoefer,18,English)
(Lino,Feddes,22,History)
(Lucius,Orlosky,20,History)
```

Data is the same, just with different nesting. On the left, the rows are divided into different Bags.

Flattening Bags (cont)

- The schema indicates what is going on

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
DESCRIBE students;
students: {first: chararray,last: chararray,age: int,dept: chararray}
students_grouped = GROUP students BY dept;
students_proj  = FOREACH students_grouped GENERATE students;
DESCRIBE students_proj;
students_proj: {students: {(first: chararray,last: chararray,age: int,dept: chararray)}}

students_flatten = FOREACH students_proj GENERATE FLATTEN(students);
DESCRIBE students_flatten;
students_flatten: {students::first: chararray,students::last: chararray,students::age: int,students::dept: chararray}
```

Post flatten, the original Schema matches up

Notice that the two schemas are basically the same, except in the second case the rows are contained inside of a Bag

- Group goes from a flat structure to a nested one
- Flatten goes from a nested structure to a flat one

So let’s flatten the Bag already

- Now that we’ve seen grouping, there’s a useful operation

```
students = LOAD ‘students.txt’ as (first:chararray, last:chararray, age:int,
dept:chararray);
students_grouped = GROUP students BY dept;
students_proj  = FOREACH students_grouped GENERATE
FLATTEN(students) as (first, last, age, dept);
DUMP students_proj;
(Willia,Cracknell,18,CS)
(Del,Graefe,20,CS)
(Douglass,Adelizzi,23,CS)
(Wes,Knill,23,EE)
(Warner,Caminita,24,Psych)
(Lesley,Kellywood,20,Biology)
(Francesco,Corraro,21,English)
(Ellyn,Meyerhoefer,18,English)
(Lino,Feddes,22,History)
(Lucius,Orlosky,20,History)
```

Same re-aliasing statement as before works with the rows resulting from flattened Bags

It worked! We have the original rows

Fun with flattens

- The GROUP example can be done with flattens

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int,
dept:chararray);

students_grp = GROUP students BY age;

students_ct = FOREACH students_grp GENERATE
   group as age,
   COUNT_STAR(students) as ct;

students_total = FOREACH (GROUP students_ct ALL) GENERATE
   FLATTEN(students_ct.(age, ct)),
   SUM(students_ct.ct) as total;

students_proj = FOREACH students_total GENERATE
   age,
   (double) ct / total as pct;
```

Flattens out the original rows

Aggregates the ct values

No need for an awkward scalar projection

Joins

- A big motivator for Pig easier joins
- Compares relations using a given key
- Output all combinations of rows with equal keys
- See appendix for more variations

```
dept_info = LOAD ‘dept_info.txt’ as (dept:chararray, credits:int);
DUMP dept_info;

(CS,1000)
(EE,1000)
(Psych,10)
(Biology,20)
(English,1)
(History,3)
```

Let’s introduce a new data source related to our previous one. This one connects a department to its required credits

Joins (cont)

- How many credits does each student need?

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
dept_info = LOAD ‘dept_info.txt’ as (dept:chararray, credits:int);
students_dept = JOIN students BY dept, dept_info BY dept;

describe students_dept;
students_dept: {students::first: chararray,students::last: chararray,students::age:
int,students::dept: chararray,dept_info::dept: chararray,dept_info::credits: int}
```

This signifies the join key (the keys whose equality will be tested)

- Joined schema is concatenation of joined relations’ schemas
- Relation name appended to aliases in case of ambiguity
  - In this case, there are two “dept” aliases

Order by

- Order by globally sorts a relation on a key (or set of keys)
- Global sort not guaranteed to be preserved through other transformations
- A store after a global sort will result in one or more globally sorted part files

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
students_by_age = ORDER students BY age ASC;
store students_by_age into 'students_by_age';
```

This output file will be globally sorted by age in ascending order

Extending Pig: UDFs

Extending Pig

- UDF's, coupled with Pig's object model, allow for extensive transformation and analysis

UDF must be compiled
and packaged into a jar,
then registered.

```
register JarContainingUdf.jar
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
students_names = FOREACH students GENERATE myudfs.ToUpperCase(first);
```

UDF is invoked by name, including package (case-sensitive!)

What's a UDF?

- A User Defined Function (UDF) is a java function implementing EvalFunc<T>, and can be used in a Pig script
  - Additional support for functions in Jython, JRuby, Groovy, and Rhino (experimental)
- Much of the core Pig functionality is actually implemented in UDFs
  - COUNT in the previous example
  - Useful for learning how to implement your own
    - src/org/apache/pig/builtin has many examples

Types of UDFs

- EvalFunc<T>
  - Simple, one to one functions
- Accumulator<T>
  - Many to one
  - Left associative, NOT commutative
- Algebraic<T>
  - Many to one
  - Associative, commutative
  - Makes use of combiners
- All UDFs must returns Pig types
  - Even intermediate stages

EvalFunc<T>

- Simplest kind of UDF
- Only need to implement an “exec” function
- Not ideal for “many to one” functions that vastly reduce amount of data (such as SUM or COUNT)
  - In these cases, Algebraics are superior
- src/org/apache/pig/builtin/TOKENIZE.java is a nontrivial example

A basic UDF

```
package myudfs;
import org.apache.EvalFunc;
public class ToUpperCase extends EvalFunc<String> {
        public String exec(Tuple input) {
                String inp = (String) input.get(0);
                return inp.toUpperCase();
        }
}
```

Type being returned (must be a Pig type)

Input is always a Tuple (ie a row). Thus, UDF input is also untyped

Input Tuple is untyped, so we must cast entries ourselves.

What happens when you run a script?

The DAG

- Pig script results in 1+ MapReduce jobs
- Graph of dependencies between these jobs is a directed acyclic graph (DAG)
  - http://www.github.com/twitter/ambrose is a great tool for visualizing a Pig script's DAG
- DAG is a result of Map/Reduce barriers

What is a Map/Reduce barrier

- A Map/Reduce barrier is a part of a script that forces a reduce stage
  - Some scripts can be done with just mappers

```
students = LOAD ‘students.txt’ as (first:chararray, last:chararray, age:int,
dept:chararray);
students_filtered = FILTER students BY age >= 20;
students_proj = FOREACH students_filtered GENERATE last, dept;
```

  - But most will need the full MapReduce cycle

```
students = LOAD ‘students.txt’ as (first:chararray, last:chararray, age:int,
dept:chararray);
students_grouped = GROUP students BY dept;
students_proj  = FOREACH students_grouped GENERATE group, COUNT(students);
```

  - The group is the difference, a “map reduce barrier” which requires a reduce step

M/R implications of operators

- What will cause a map/reduce job?
  - GROUP and COGROUP
  - JOIN
    - Excluding replicated join (see Appendix)
  - CROSS
    - To be avoided unless you are absolutely certain
    - Potential for huge explosion in data
  - ORDER
  - DISTINCT
- What will cause multiple map reduce jobs?
  - Multiple uses of the above operations
  - Forking code paths

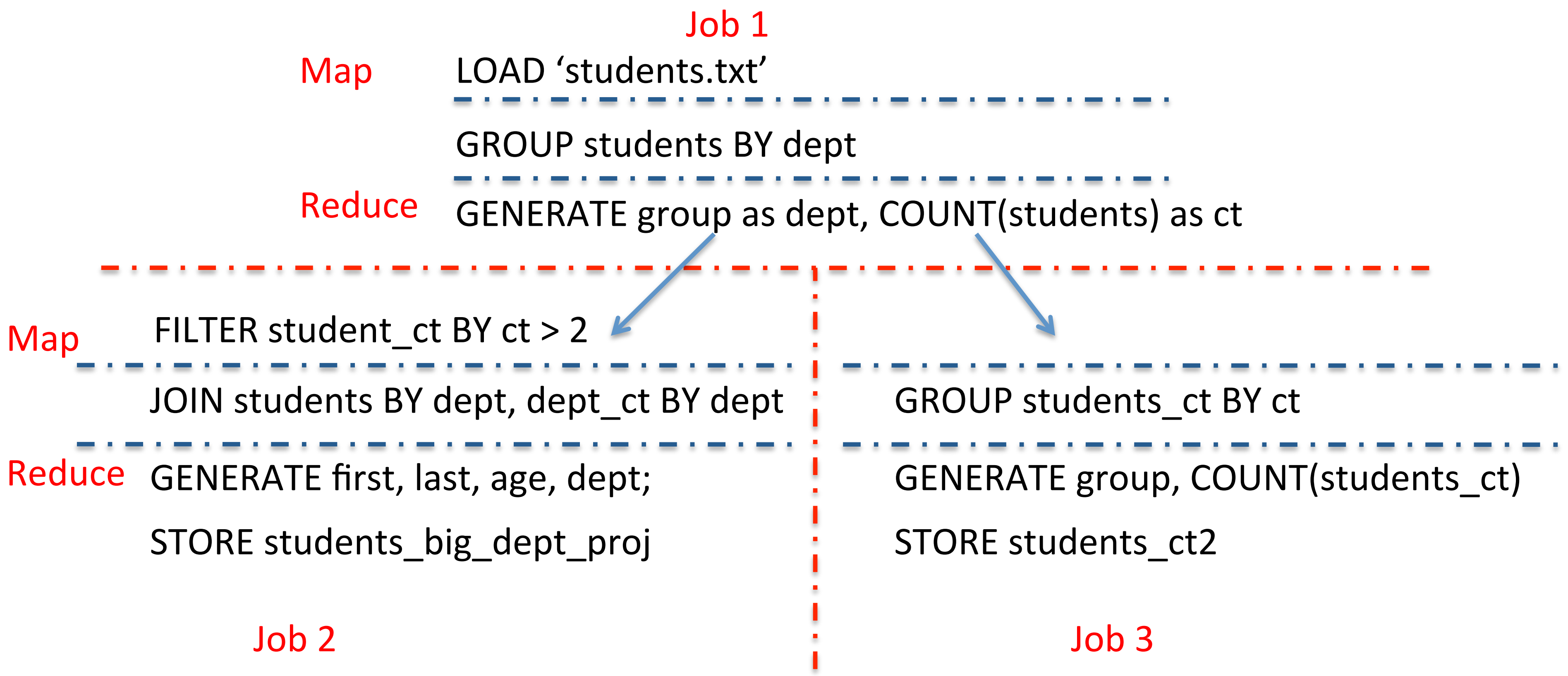
Making it more concrete

- First step is identifying the M/R barriers

Job #

```
  students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int, dept:chararray);
1 students_grp = GROUP students BY dept;
  students_ct = FOREACH students_grp GENERATE group as dept, COUNT(students) as ct;
- . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . -
  students_grp2 = GROUP students_ct BY ct;
2 students_ct2 = FOREACH students_grp2 GENERATE group, COUNT(students_ct);
  STORE students_ct2 INTO ‘histogram’;
- . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . - . -
  dept_ct = FILTER student_ct BY ct > 2;
  students_big_dept = JOIN students BY dept, dept_ct BY dept;
3 students_big_dept_proj = FOREACH students_big_dept GENERATE
     students::first as  first, students::last as last, students::age as age, students::dept as dept;
  STORE students_big_dept_proj INTO ‘students_in_big_departments’;
```

A DAG example



Job 1

```
LOAD 'students.txt'
GROUP students BY dept
GENERATE group as dept, COUNT(students) as ct
```

Job 2

```
FILTER student_ct BY ct > 2
JOIN students BY dept, dept_ct BY dept
GENERATE first, last, age, dept;
STORE students_big_dept_proj
```

Job 3

```
GROUP students_ct BY ct
GENERATE group, COUNT(students_ct)
STORE students_ct2
```

Life lessons

How to make good Pig scripts

- Project early, project often
  - Always better to be more explicit
  - Reduces the amount of information being shuffled
- Explicitly name things
  - Pig is smart about it’s schemas, but making things explicit will make scripts much more readable

How to make good Pig scripts (cont)

- Don't reuse relation names
  - Makes troubleshooting failing M/R jobs harder
- For UDFs, implement Accumulator and Algebraic if possible
  - Unless it is just a simple one to one EvalFunc
- General life lesson: nobody likes unreadable, terse code, no matter how powerful the language. Be explicit!

Getting help from the Pig community

- Read the docs! http://pig.apache.org has more formal information on all of this
- There are a bunch of Pig experts that want to help you: user@pig.apache.org
- When having issues, please include the script
  - Ideally, a smaller script that isolates your error
  - Example data even more ideal
- If you found a bug, file a bug report!
  - https://issues.apache.org/jira/browse/PIG

Appendix

More on projections

- Projection reduces the amount of data being processed
  - Especially important between map and reduce stages when data goes over the network

```
rel1 = LOAD ‘data1.txt’ as (a, b, c, … lots of columns ..);
rel2 = LOAD ‘data2.txt’ as (a, … lots of columns ..);
rel3 = JOIN rel1 BY a, rel2 BY a;
rel4 = FOREACH rel3 GENERATE rel1::a, rel1::b, rel1::c;
```

In this case, all of the columns in relation1 and relation2 would be se across the network! Pig tries to optimize this, but sometimes fails. Remember: be explicit!

```
rel1 = LOAD ‘data1.txt’ as (a, b, c, … lots of columns ..);
rel1_proj = FOREACH rel1 GENERATE a, b, c;
rel2 = LOAD ‘data2.txt’ as (a, … lots of columns ..);
rel2_proj = FOREACH rel2 GENERATE a;
rel3 = JOIN rel1_proj BY a, rel2_proj BY a;
rel4 = FOREACH rel3 GENERATE rel1_proj::a, rel1_proj::b, rel1_proj::c;
```

This ensures extra data won’t be shuffled, and makes your code more explicit

Scalar projection

- All interaction and transformation is in Pig is done on relations
- Sometimes, we want access to an aggregate
  - Scalar projection allow us to use intermediate aggregate results in a script

```
students = LOAD 'students.txt';
count = FOREACH (GROUP students ALL) GENERATE COUNT_STAR($1) as ct;
```

(Incidentally, this is the pattern for counting a relation)

This will make the column ct in count available as a long

```
proj = foreach students generate *, (long) count.ct;
```

If the relation has more than one row or the specified column isn't of the right type, it will error out

More on SUM, COUNT, COUNT_STAR

- In general, SUM, COUNT, and other aggregates implicitly work on the first column

```
rel1 = LOAD ‘data1.txt’ as (x:int, y:int, z:int);
rel2 = GROUP rel1 ALL;
rel3 = FOREACH rel2 GENERATE SUM(rel1);
rel4 = FOREACH rel2 GENERATE SUM(rel1.x);
rel5 = FOREACH rel2 GENERATE SUM(rel1.y);
```

These will be the same

These won’t

- COUNT counts only non-null fields
- COUNT_STAR counts all fields

```
rel6 = FOREACH rel2 GENERATE COUNT(rel1);
rel7 = FOREACH rel2 GENERATE COUNT_STAR(rel1);
```

This counts only non-null values of x

Whereas this counts the number of values in the relation

More on sorting

- Sorting is a global operation, but can be distributed
- Must approximate distribution of the sort key
- Imagine evenly distributed data between 1 and 100. With 10 reducers, can send 1-10 to computer 1, 11-20 to computer 2, and so on. In this way, the computation is distributed but the sort is global
- Pig inserts a sorting job before an order by to estimate the key distribution

More on spilling

- Spilling means that, at any time, a data structure can be asked to write itself to disk
- In Pig, there is a memory usage threshold
- This is why you can only add to Bags, or iterate on them
  - Adding could force a spill to disk
  - Iterating can mean having to go to disk for the contents

Flattening multiple Bags

- The result from multiple flatten statements will be **crossed**
  – To only select a few columns in a Bag, syntax is bag_alias.(col1, col2)

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int,
dept:chararray);
students_grouped = GROUP students BY dept;
students_proj  = FOREACH students_grouped GENERATE FLATTEN(students.first),
                                                  FLATTEN(students.age);
```

These two are not the same!! I highly recommend running them to understand what's going on

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int,
dept:chararray);
students_grouped = GROUP students BY dept;
students_proj  = FOREACH students_grouped GENERATE FLATTEN(students.(first, age));
```

This is the proper way to choose a specific set of columns from a bag

Nested foreach

- An advanced, but extremely powerful use of FOREACH let's a script do more analysis on the reducers
- Imagine we wanted the distinct number of ages per department

```
students = LOAD 'students.txt' as (first:chararray, last:chararray, age:int,
dept:chararray);
students_grouped = GROUP students BY dept;
unique_ages = FOREACH students_grouped {
   dst = DISTINCT students.age;
   GENERATE group as dept, FLATTEN(dst) as age;
}
```

Curly braces denote nested block

Within the nested block can use a subset of Pig commands (FILTER, DISTINCT, ORDER BY) to manipulate the bag of rows associated with the group key

This creates a Bag, dst, which is the distinct of the Bag students.age

Nested foreach: be careful!

- Is very useful, but since computation is done in memory, can blow up

```
students = LOAD ‘students.txt’ as (first:chararray, last:chararray, age:int,
dept:chararray);
students_grouped = GROUP students BY dept;
unique_ages = FOREACH students_grouped {
   dst = DISTINCT students.age;
   GENERATE group as dept, FLATTEN(dst) as age;
}
```

Is the same as…

```
students = LOAD ‘students.txt’ as (first:chararray, last:chararray, age:int,
dept:chararray);
students_dst = FOREACH (GROUP students BY (dept, age))
   GENERATE FLATTEN(group) as (dept, age);
unique_ages = FOREACH (GROUP students_dst BY dept)
   GENERATE group as dept, FLATTEN($1.age) as age;
```

The latter scales better but will be slower, requiring 2 MR jobs

Join optimizations

- Pig has three join optimizations. Using them can potentially make jobs run MUCH faster
- Replicated join
  - a = join rel1 by x, rel2 by x using ‘**replicated**’;
- Skewed join
  - a = join rel1 by x, rel2 by x using ‘**skewed**’;
- Merge join
  - a = join rel1 by x, rel2 by x using ‘**merge**’;

Replicated join

- Can be used when:
  - Every relation besides the left-most relation can fit in memory
- Will invoke a map-side join
  - Will load all other relations into memory in the mapper and do the join in place
- Where applicable, massive resources savings

Skewed join

- Useful when **one** of the relations being joined has a key which dominates
  - Web logs, for example, often have a logged out user id which can be a large % of the keys
- The algorithm first samples the key distribution, and then replicate the most popular keys
  - Some overhead, but worth it in cases of bad skew
- Only works if there is skew in one relation
  - If both relations have skew, the join degenerates to a cross, which is unavoidable

Merge join

- This is useful when you have two relations that are already ordered
  - Cutting edge let's you put an "order by" before the merge join
- Will index the blocks that correspond to the relations, then will do a traditional merge algorithm
- Huge saving when applicable

Tying it to Hadoop

So what’s actually going on?

- So now we can write a script, but we haven’t really talked about how Pig executes a script “under the hood”
- Pig puts together a Map/Reduce job based on the script that you give it

What happens when run a script?

- First, Pig parses your script using ANTLR
- The parser creates an intermediate representation (AST)
- The AST is converted to a Logical Plan
- The Logical Plan is optimized, then convert to a Physical Plan
- The Physical Plan is optimized, then converted to a series of Map/Reduce jobs

Wait…what?

- Layers of abstractions are very useful for performing optimizations at various levels
- Logical Plan
  - High level description of the computation
- Physical Plan
  - Pipeline that performs the computation
- Map/Reduce Job
  - Graph of jobs that actually run on the cluster

Logical Plan

- High level description of the data flow
- Describes the computation that will be done, **without** implementation
- Can be optimized
  - Column pruning
    - Throws out any unused columns
  - Filter push-down
    - Push filters as high as possible in order to reduce data being processed and shuffled

Physical plan

- Physical description of the computation
- Creates a usable pipeline
  - Pipeline usable independent of M/R
  - Could use this pipeline and target other processing frameworks
- This stage can also be optimized
  - In memory aggregation instead of combiners

MapReduce plan

- The logical and physical plans are ideally divorced from the details of running on Hadoop
  - This is not always the case, but it's close
- MRCompiler breaks the physical plan into a DAG of M/R jobs
- Can be optimized as well
  - Combining multiple M/R jobs into one

How do I see the plans?

- Pigs “explain <relation>” will print the three plans that Pig generates
  - Extremely useful for debugging
  - Can be a bit advanced, beyond scope of this presentation
  - Pig listserv can be very helpful

Advanced UDFs

Accumulator<T>

- Used when the input is a large bag, but order matters
  - Allows you to work on the bag incrementally, can be much more memory efficient
- Difference between Algebraic UDFs is generally that you need to work on the data in a given order
  - Used for session analysis when you need to analyze events in the order they actually occurred
- src/org/apache/pig/builtin/COUNT.java is an example
  - Also implements Algebraic (most Algebraic functions are also Accumulative)

Algebraic

- Commutative, algebraic functions
  - You can apply the function to any subset of the data (even partial results) in any order
- The most efficient
  - Takes advantage of Hadoop combiners
  - Also the most complicated ☹
- src/org/apache/pig/builtin/COUNT.java is an example

Algebraic: initial function

- Accepts a Tuple with a Bag with one row
  - Complicated, but that's how it is
- Makes no assumptions on order of execution
  - Instantiation may be executed on 0 or more rows
- Must return a Tuple containing valid Pig data type
  - EvalFunc<Tuple>

Algebraic: intermed function

- Accepts Tuple which contains a Bag of results from the initial **or intermediate** function
  – Must be able to accept it's own output
- Makes no assumptions on how many elements that Bag will contain
- Might not be called at all
- Must return a Tuple containing valid Pig data types
  - EvalFunc<Tuple>

Algebraic: final function

- Responsible for returning the final aggregate
- Accepts Tuple which contains a Bag of results from initial or intermediate function
- Only invoked once
- Must return a valid Pig data type
  - EvalFunc<T>

Algebraic: Sum example

This abstract class lets us just implement Algebraic

```
public SimplerSum extends AlgebraicEvalFunc<Long> {
  private static final TupleFactory mTupleFactory = TupleFactory.getInstance();

  public String getInitial() { return Initial.class.getName(); }
  public String getIntermed() { return Initial.class.getName(); }
  public String getFinal() { return Final.class.getName(); }

  private static long sumBag(DataBag inp) {
    long sum = 0;
    for (Tuple t : inp) { sum += ((Number)t.get(0)).longValue(); }
    return sum;
  }

  // implementation continued on next slide
}
```

This shared class makes Tuples

These functions return the classes that will do the processing

In this case, the initial and intermediate functions are the same

This helper function actually does the summing

Algebraics: Sum example (cont)

```
public SimplerSum implements Algebraic {
  // assumes previous slide

  static class Initial extends EvalFunc<Tuple> {
    public Tuple exec(Tuple input) throws IOException {
      Tuple out = mTupleFactory.newTuple(1);
      out.set(0, sumBag((DataBag)input.get(0)));
      return out;
    }
  }

  static class Final extends EvalFunc<Long> {
    public Long exec(Tuple input) throws IOException {
      return sumBag((DataBag)input.get(0));
    }
  }
}
```

Both classes extend the EvalFunc class

In this case, the Initial function can also work on collections of it's own output

Only difference is that it returns the actual Long instead of wrapping it

Other extension interfaces

- Pig has other interfaces used in advanced cases
- AlgebraicEvalFunc<K>
  - Implement Algebraic, get Accumulator for free
- AccumulatorEvalFunc<K>
  - Implement Accumulator, get EvalFunc free
- TerminatingAccumulator<K>
  - Accumulator which can terminate early
    - Can save a significant amount of processing
- IteratingAccumulatorEvalFunc<K>
  - Provides a more human accumulator interface (at the cost of some potential overhead)

See the documentation!

- The documentation has information on all of these topics and more, including
  - Outer joins
  - Cogroups
  - Macros
  - Parameters
  - More in-depth Accumulator and Algebraic
- Also be sure to look at the source of the built-in functions (as well as those in Piggybank)
  - This is a huge benefit of open source!