PART 2.
DATA STORAGE AND FLOW MANAGEMENT

Instructor: Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

Today’s topics
- FAQs
- Pig-continued

FAQs
- Term project
  - Please sign up for the Final presentation/Final demonstration
  - Per team
- Term project final presentation/final report
  - Preparation guideline is available at the course web page
- Final presentation
  - 3 days (Tuesday/Thursday/Friday): 12:30PM – 1:50PM
  - You are required to attend at least 2 of above sessions
  - Participation score (4)
    - Your attendance and questions/feedbacks during the presentation

Apache Pig
Relational Operator: join
[Using 'replicated' | 'skewed' | 'merge']

Apache Pig
Arithmetic Operator: flatten
**flatten** for Tuples

- *flatten* substitutes the fields of a tuple in place of the tuple
  - (a, (b, c))
  - GENERATE $0$, flatten($1$)
  - Tuple will become (a,b,c)

- Does not produce cross product
  - Elevates each field in the tuple to a top-level field
  - Empty tuple will be removed the entire record.

**flatten** for Bags

- *Flatten* un-nests a bag to create new tuples
  - $\{((b,c),(d,e))\}$
  - GENERATE flatten($0$)
  - Results will be (b,c) and (d,e)

- *Flatten* cross products tuples to remove a level of nesting in a bag
  - (a, ((b,c),(d,e)))
  - GENERATE $0$, flatten($1$)
  - Results will be
    - (a, b, c) and (a, d, e)

**flatten** with multiple bags and nulls

- If there is more than one bag and both are flattened
  - $n \times m$ rows ($n$ and $m$ are the numbers of records in bags)

- What if the bag is empty?
  - No records are produced
  - Pig may or may not have the schema of the data in the bag
  - Does not know how to fill in nulls for the missing values
  - Mathematically correct
  - Crossing a set $S$ with the empty set results in the empty set

**flatten** with nulls

- Replace empty bags with a constant bag

```
players = load 'baseball' as (name:chararray, 
  team:chararray, position:bag{t: (p:chararray)}, 
  bat:map{});

nonempty = foreach players generate name, 
  ((position is null or IsEmpty(position))?
    {'unknown'}:position) 
  {('unknown'):position};

pos = foreach nonempty generate name, 
  flatten(position) as position;

bypos = group pos by position;
```

Apache Pig

Relational Operator: cogroup, cross and union
cogroup
- Generalization of group
- Collects records of \( n \) inputs based on key
- A record with a key and one bag for each input

\[
A = \text{load} \ 'input1' \ \text{as} \ \{(id:\text{int}, \text{val}:\text{float})\};
\]
\[
B = \text{load} \ 'input2' \ \text{as} \ \{(id:\text{int}, \text{val}:\text{float})\};
\]
\[
C = \text{cogroup} \ A \ \text{by} \ id, \ B \ \text{by} \ id;
\]
describe \( C \);

\[
C: \{\text{group}: \text{int}, \ A: \{(id: \text{int}, \text{val}: \text{float})\}, \ B: \{(id: \text{int}, \text{val2}: \text{int})\}\}
\]

cross
- Computes the cross product of two or more relations

\[
A = \text{LOAD} \ 'data1' \ \text{AS} \ \{(a1:\text{int}, a2:\text{int}, a3:\text{int})\};
\]
\[
\text{DUMP} \ A;
\]
\[
(1,2,3)
(4,2,1)
\]
\[
B = \text{LOAD} \ 'data2' \ \text{AS} \ \{(b1:\text{int}, b2:\text{int})\};
\]
\[
\text{DUMP} \ B;
\]
\[
(2,4)
(8,9)
(1,3)
\]
\[
X = \text{CROSS} \ A, \ B;
\]
\[
\text{DUMP} \ X;
\]
\[
(1,2,3,2,4)
(1,2,3,8,9)
(1,2,3,1,3)
(4,2,1,2,4)
(4,2,1,8,9)
(4,2,1,1,3)
\]

union
- Concatenating two datasets
- Not joining

\[
A = \text{LOAD} \ 'data' \ \text{AS} \ \{(a1:\text{int}, a2:\text{int}, a3:\text{int})\};
\]
\[
\text{DUMP} \ A;
\]
\[
(1,2,3)
(4,2,1)
\]
\[
B = \text{LOAD} \ 'data' \ \text{AS} \ \{(b1:\text{int}, b2:\text{int})\};
\]
\[
\text{DUMP} \ B;
\]
\[
(1,2,3)
(1,2,3)
(4,2,1)
(4,2,1)
(2,4)
(8,9)
(1,3)
\]
\[
X = \text{UNION} \ A, \ B;
\]
\[
\text{DUMP} \ X;
\]
\[
(1,2,3)
(1,2,3)
(4,2,1)
(4,2,1)
(2,4)
(8,9)
(1,3)
\]

Apache Pig
Non-linear dataflows
Nonlinear Data Flows

- Linear Data flows
  - One input is loaded, processed, and stored
- Tree Data flows
  - Combine multiple data flows
    - Join, cogroup, union
  - One input results in more than one output
  - Data flow is split and eventually joined back together
  - Diamonds
  - Pig supports these directed acyclic graphs (DAGs).

Splits in your data flow

- Implicit
  - No specific operator or syntax is required in your script
  - User refers to a given relation multiple times
- Explicit
  - By using split operator

Implicit Splits

- No specific operator or syntax required

```
players = load 'baseball' as (name:chararray, team:chararray, position:bag{t: (p:chararray)}, bat:map);
pwithba = foreach players generate name, team, position, bat#'batting_average' as batavg;
byteam = group pwithba by team;
avgbyteam = foreach byteam generate group, AVG(pwithba.batavg);
store avgbyteam into 'by_team';
flattenpos = foreach pwithba generate name, team, flatten(position) as position, batavg;
bypos = group flattenpos by position;
avgbypos = foreach bypos generate group, AVG(flattenpos.batavg);
store avgbypos into 'by_position';
```

Explicit Splits

- Split your data flow in as many ways as you like.

```
wlogs = load 'weblogs' as (pageid, url, timestamp);
split wlogs into apr03 if timestamp < '20110404',
  apr02 if timestamp < '20110403' and timestamp > '20110401',
  apr01 if timestamp>'20110402' and timestamp < '20110331';
store apr03 into '20110403';
store apr02 into '20110402';
store apr01 into '20110401';
```

- Is split like switch or case?
  - Single record can go to multiple paths of the split.

Rewriting Splits

- Pig will internally rewrite the original script in this way.

```
wlogs = load 'weblogs' as (pageid, url, timestamp);
apr03 = filter wlogs by timestamp < '20110404';
apr02 = filter wlogs by timestamp < '20110403' and timestamp > '20110401';
apr01 = filter wlogs by timestamp< '20110402' and timestamp >'20110331';
store apr03 into '20110403';
store apr02 into '20110402';
store apr01 into '20110401';
```

Processing nonlinear data flow

- Multi-query
  - Combines them into single MapReduce jobs
  - Creates separate pipelines inside the map
  - Sends the appropriate records to each pipeline
Combining multiple groups

- Both groups will be executed inside one MapReduce job
- Duplicate records on the map side and annotate each record with its pipeline number

```
pwithba = foreach players generate name, team, position, batting_average as batavg;
byteam = group pwithba by team;
avgbyteam = foreach byteam generate group,
             AVG(pwithba.batavg) as batavg;
store avgbyteam into 'by_team';
flattenpos = foreach pwithba generate name, team, flatten(position) as position,
             batavg;
bypos = group flattenpos by position;
```

Multi-query

```
Pipe:0 team:SF, avg:0.239 shuffle key = SF
Pipe:1 pos:2B, avg:0.239 shuffle key = 2B
```

Pig will NOT combine operators..

- Multiple-input operators
  - join
  - union
  - cross
  - cogroup
- order
- Multiple group statements (with some of them using Hadoop’s combiner).

Limits of multi-query

- Slow down the shuffle phase
  - It requires replicating records in the map
- If the multiquery does not work for your task
  - Combines too many jobs and slows down too much
  - Make scripts for some of the jobs separately
- Or turn off the multiquery
  - -M or --no_multiquery
  - Set the property opt.multiquery to false

Controlling Execution: environment

```
set default.parallel 10;
set job.name my_job;
set opt.multiquery false;
set io.sort.mb 2048;
users = load 'users';
```
Controlling Execution: partitioner

- Pig does not override the default partitioner
  - Except for order and skew join
- If you want to set your own partitioner,
  - Write a partitioner extending org.apache.hadoop.mapreduce.Partitioner<KEY,VALUE>
  - Register it!

```java
register acme.jar;
users = load 'users' as (id, age, zip);
grp = group users by id partition by com.acme.userpartitioner parallel 100;
```

Parameter Substitution

- Parameter substitution

```java
yesterday = filter daily by date == '$DATE';
grpd = group yesterday all;
```

Including other Pig Latin Script

- Use the relative path and full path for the file to be included

```java
import '../examples/ch6/dividend_analysis.pig';
```

Developing tools

- Diagnostic operators
  - describe
    - Shows you the schema of a relation
  - dump
    - Dumps or displays results to screen
  - explain
    - Shows the execution plan
  - illustrate
    - Runs your script on a sample of your data

- Pig Statistics
  - Summary set of statistics
  - Pig produces at the end of every run
  - PigUnit

Example

```java
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);
B = GROUP A BY name;
C = FOREACH B GENERATE COUNT(A.age);
EXPLAIN C;
```
Example

Logical Plan:

Store xxx-Fri Dec 05 19:42:29 UTC 2008-23 Schema: {long} Type: Unknown

---ForEach xxx-Fri Dec 05 19:42:29 UTC 2008-15 Schema: {long} Type: bag
ebar

Physical Plan:


---New For Each(false){bag} - xxx-Fri Dec 05 19:42:29 UTC 2008-39

|POUserFunc(org.apache.pig.builtin.COUNT){long} - xxx-Fri Dec 05 19:42:29 UTC 2008-38

Example

Logical Plan:

Store xxx-Fri Dec 05 19:42:29 UTC 2008-23 Schema: {long} Type: Unknown

---ForEach xxx-Fri Dec 05 19:42:29 UTC 2008-15 Schema: {long} Type: bag
ebar

Physical Plan:


---New For Each(false){bag} - xxx-Fri Dec 05 19:42:29 UTC 2008-39

|POUserFunc(org.apache.pig.builtin.COUNT){long} - xxx-Fri Dec 05 19:42:29 UTC 2008-38

MapReduce node xxx-Fri Dec 05 19:42:29 UTC 2008-41

Map Plan

Map Plan

Local Rearrange[tuple]{bytearray}{false}

Project{bytearray}[0]

---argThis: New For Each{false}{bytearray}[bag]

POUserFunc(org.apache.pig.builtin.COUNT){long}

Pre Compiler Local Rearrange[tuple]{Unknown}

---divs: load(file:///home/gates/gi/programming/pig/datasets/AVG5Dividends.csv)

Combine Plan

Map Plan

Local Rearrange[tuple]{bytearray}{false}

Project{bytearray}[0]

---argThis: New For Each{false}{bytearray}[bag]

POUserFunc(org.apache.pig.builtin.COUNT){long}

Pre Compiler Local Rearrange[tuple]{Unknown}

---divs: load(file:///home/gates/gi/programming/pig/datasets/AVG5Dividends.csv)

Reduce Plan

argThis: Store(file:///home/gates/gi/programming/pig/datasets/AVG5Dividends.csv)

POUserFunc(org.apache.pig.builtin.COUNT){long}

---argThis: New For Each{false}{bytearray}[bag]

Evaluation Function and Filter Function

- Evaluation functions
  - UDFs that operate on single elements of data or collections of data
- Filter functions
  - UDFs that can be used as part of filter statements

Apache Pig

Evaluation function and Filter function
Writing an Evaluation Function in Java

- Java or Python
- Evaluation functions
- Pig constructs a separate instance of your UDF
  - In addition to an instance in each task,
  - Pig constructs an instance of your UDF for the planning stage
- All evaluation functions extend,
  - org.apache.pig.EvalFunc
  - Uses Java generic

```
public class Pow extends EvalFunc<Long> {
    public Long exec(Tuple input) throws IOException {
        try {
            int first = (Integer) input.get(0);
            int second = (Integer) input.get(1);
            long result = 1;
            for (int i = 0; i < second; i++) {
                long prereesult = result;
                result *= first;
                if (prereesult > result) {
                    warn("Overflow!", PigWarning.Too_LARGE_FOR_INT);
                    return null;
                }
            }
            return result;
        } catch (…)
    }
}
```

What does this UDF do?

```
private TupleFactory tupleFactory = TupleFactory.getInstance();
private Object readField(JsonParser p, ResourceFieldSchema field, int fieldnum) throws IOException {
    ResourceSchema s = field.getSchema();
    ResourceFieldSchema[] fas = field.getFields();
    Tuple t = tupleFactory.newTuple(fas.length);
    for (int j = 0; j < fas.length; j++) {
        t.set(j, readField(p, fas[j], j));
    }
    return t;
}
```

Interacting with Pig

- Tuples and bags
  - Use factory classes
- Get an Instance of Tuple: TupleFactory
  - An abstract singleton class for creating tuples
  - Users can choose a TupleFactory for their UDFs
  - TupleFactory.getInstance()
- Create new tuples
  - newTuple(), newTuple(int size)
  - set(int fieldNum, Object val)

Evaluation functions

```
public class Pow extends EvalFunc<Long> {
    public Long exec(Tuple input) throws IOException {
        try {
            int first = (Integer) input.get(0);
            int second = (Integer) input.get(1);
            long result = 1;
            for (int i = 0; i < second; i++) {
                long prereesult = result;
                result *= first;
                if (prereesult > result) {
                    warn("Overflow!", PigWarning.Too_LARGE_FOR_INT);
                    return null;
                }
            }
            return result;
        } catch (…)
    }
}
```

Create a new Tuple

```
private TupleFactory tupleFactory = TupleFactory.getInstance();
private Object readField(JsonParser p, ResourceFieldSchema field, int fieldnum) throws IOException {
    ResourceSchema s = field.getSchema();
    ResourceFieldSchema[] fas = field.getFields();
    Tuple t = tupleFactory.newTuple(fas.length);
    for (int j = 0; j < fas.length; j++) {
        t.set(j, readField(p, fas[j], j));
    }
    return t;
}
```
**BagFactory**

- Construct a Bag
  - BagFactory.getInstance()

- Create a new bag
  - newDefaultBag()

- Add tuple to a bag
  - add(Tuple t);

---

**Type Checking**

- Pig type-checks a script before running it.

- EvalFunc’s getArgToFuncMapping() method will be called during the type-checks of Pig

- Simple output type
  - Pig uses Java reflection

- Input type
  - Pig cannot examine the type of input

---

**Type checking example**

```java
public class Pow extends EvalFunc<Long> {
    public Long exec(Tuple input) throws IOException {
        try {
            int first = (Integer) input.get(0);
            int second = (Integer) input.get(1);
            long result = 1;
            for (int k = 0; i < second; i++) {
                long preresult = result;
                result *= first;
                if (preresult > result) {
                    warn("Overflow!", PigWarning.Too_LARGE_FOR_INT);
                    return null;
                }
            }
            return result;
        } catch (…) {
            ...
        }
    }
}
```

---

**Error Handling : Warning**

- Warn
  - A method of EvalFunc
  - Takes message and a warning code
  - org.apache.pig.PigWarning
  - Reported to the user at the end of the job
  - Allows your job to continue
Error Handling: Error
- Stop all the jobs
- Hadoop will restart your job
- If any particular task fails three times
  - Hadoop will not restart it again
- EvalFunc
  - log()
  - Hadoop will print any log message into logfiles

Accessing data file(s) in HDFS from your UDF

Accessing data file(s) in HDFS from your UDF

Using non-default constructor

Using non-default constructor

Complete exec()

Complete exec()

Interacting with frontend

Interacting with frontend
What if my UDF needs more memory?

- Pig's bag handles spilling data to disk automatically
  - If they pass a certain size threshold or when only a certain amount of heap space remains

- Pig's tuples and maps must fit into memory

Apache Pig

Algebraic Interface: Using Hadoop's Combiner

Combiner

- "mini reducer"
  - Applied during the map phase
  - Before to send the new key-value pairs to reducers

- Implement setCombinerClass()

  It can be applied potentially several times still during the map phase before to send the new set of key/value pairs to the reducer

Hadoop Combiner

- The function you want to apply should be commutative and associative

  - Commutative operations
    - $f(a,b) = f(b,a)$
    - $A+B = B+A$
    - $AxB = BxA$
    - $\text{Max}(A,B) = \text{Max}(B,A)$

  - Associative operations
    - $f(f(a,b),c) = f(a,f(b,c))$
    - $(A+B)+C = A+(B+C)$
    - $A+(BxC) = (AxB)+C$

Algebraic Interface[1/3]

- Best use cases of combiner (1)
  - Distributive functions
    - Same results from
      - Dividing the input set into subsets, applying the functions, and applying the functions to the results from the subsets
      - Applying function to the original sets
    - SUM is an example of this

Algebraic Interface[2/3]

- Best use cases of combiner (2)
  - Aggregate functions
    - Usually applied to grouped data

  ```
  A = LOAD 'student_data' AS (name: chararray, age: int, gpa: float);
  B = GROUP A BY name;
  C = FOREACH B GENERATE group, COUNT(A);
  DUMP C;
  ```

  - e.g. Algebraic functions
Algebraic Interface [3/3]

- Algebraic functions
  - It can be divided into initial, intermediate, and final functions (initial ± final)
  - The initial function is applied to subsets of the input set
  - Invoked once for each input tuple by the map process and produces partial results
  - The intermediate function is applied to the result of the initial functions
  - Invoked once by each combiner invocation (which can happen zero or more times) and also produces partial results
  - The final function is applied to all of the results of the intermediate function
    - Invoked once by the reducer and produces the final result
  - COUNT is an example of an algebraic function
    - COUNT: initial function
    - SUM: intermediate and final function
  - With combiner?
  - Without combiner?

How it works in the pipeline

```java
A = LOAD 'student_data' AS (name: chararray, age: int, grade: float);
B = GROUP A BY name;
C = FOREACH B GENERATE group, COUNT(A);
```

Example: Implementation of COUNT

```java
static public class Initial extends EvalFunc<Tuple> {
    protected final long sum = 0;
    public Tuple exec(Tuple input) throws IOException {
        try {
            String msg = String.valueOf(input.get(0));
            throw new TypeCastException(msg, input);}
        } catch (TypeCastException e) {
            return this.getException(msg, input, new IOException(e));
        }
    }
    static protected long sum(Tuple input) throws IOException,
    NumberFormatException {
        String val = String.valueOf(input.get(0));
        long sum = 0;
        for (int n = val.length(); n > 0; --n) {
            sum += Long.valueOf(val.substring(n - 1, n));
        }
        return sum;
    }
    public String getInitial();
    public String getInermed();
    public String getFinal();
}
```

Example: Implementation of COUNT

```java
static public class Final extends EvalFunc<Tuple> {
    protected final long sum = 0;
    public Tuple exec(Tuple input) throws IOException {
        try {
            String msg = String.valueOf(input.get(0));
            throw new TypeCastException(msg, input);}
        } catch (TypeCastException e) {
            return this.getException(msg, input, new IOException(e));
        }
    }
    static protected long sum(Tuple input) throws IOException,
    NumberFormatException {
        String val = String.valueOf(input.get(0));
        long sum = 0;
        for (int n = val.length(); n > 0; --n) {
            sum += Long.valueOf(val.substring(n - 1, n));
        }
        return sum;
    }
    public String getInitial();
    public String getInermed();
    public String getFinal();
}
```