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

FAQs
• Help session for PA3
  • Link is available

• AWS account?
  • Please create one ASAP!

Apache Pig
Data types

Schemas
• Pig supports schema
  • If a schema for the data is available
    • Up-front error checking
    • Optimization
  • If no schema is available
    • Pig will still process the data

If a schema is available..
• Your program should tell Pig what it is when you load the data:

```pig
Dividends = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray, date:chararray, dividend:float);
```

```pig
Dividends = load 'NYSE_dividends' as (exchange, symbol, date, dividend);
```
Multiple types with schema [1/2]

```
cat student;
John  18  4.0
Mary  19  3.8
Bill  20  3.9
Joe   18  3.8

A = LOAD 'student' AS (name:chararray, age:int, gpa:float);

DESCRIBE A;
A: {name: chararray, age: int, gpa: float}

DUMP A;
(John, 18, 4.0F)
(Mary, 19, 3.8F)
(Bill, 20, 3.9F)
(Joe, 18, 3.8F)
```

Multiple types with schema [2/2]

```
cat student;

A = LOAD 'data' AS (name: chararray, age: int, gpa: bytearray);

DESCRIBE A;
A: {name: chararray, age: int, gpa: bytearray}

DUMP A;
(John, 18, 4.0)
(Mary, 19, 3.8)
(Bill, 20, 3.9)
(Joe, 18, 3.8)
```

If no schema is available

- Use a dollar sign + position
  
  daily = load 'NYSE_daily';
  calcs = foreach daily generate $7/1000, $3 * 100.0, SUBSTRING($0, 0, 1), $6 - $3;

- Pig makes a safe guess
  - $7/1000
  - Guess: $7 (eight'th field) is numeric type
  - $3*100.0
  - Guess: $3 is numeric type
  - Substring()
    - Guess: $0 is chararray
  - $6 - $3
  - Guess: $6 and $3 are numeric type

- $6 - $3
  - Guess: $6 and $3 are numeric type

- $6 - $3
  - Guess: $6 and $3 are numeric type
  - Guess: $6 and $3 are numeric type

Casts

- **Casts to byarray are NOT allowed**
- Casts from byarray to any type are allowed
- Casts to and from complex types are NOT allowed
Internal Casts

- Pig will change \texttt{volume} to \texttt{(float)volume} internally.
  - Without losing precision
  - Lower precision $\rightarrow$ higher precision
- Pig widens types
  - \texttt{int} with \texttt{long} $\rightarrow$ \texttt{(long)int}
  - \texttt{long} with \texttt{float} $\rightarrow$ \texttt{float}
- There is NO internal casts between numeric types and chararrays

How strongly typed is Pig?

- Strongly typed computer language
  - Users must declare up front the type for all variables
    - e.g. Java
- Weakly typed language
  - Variables can take on values of different types
    - Adapt as the occasion demands
    - e.g. Perl

How strongly typed is Pig?-contd

- Pig: “Gently typed”
  - If the schema provided, it follows the schema
  - If no schema provided, it adapts to the actual types at runtime

Relation Names

- Pig Latin is a dataflow language
  - Each processing step results in a new data set or relation.

Apache Pig
Basic Components
**Reusing the relation names**

- It is possible to reuse relation names.
  
  ```
  A = load 'NYSE_dividends' (exchange, symbol, date, dividends);
  A = filter A by dividends > 0;
  A = foreach A generate UPPER(symbol);
  ```

- This example creates a new relation called A repeatedly.
- It loses track of the old relations called A.
- Not recommended

**Field Names**

- Field (or column) in a relation
  
  ```
  A = load 'NYSE_dividends' (exchange, symbol, date, dividends);
  A = filter A by dividends > 0;
  A = foreach A generate UPPER(symbol);
  ```

- dividends, symbol, date ...

- Looks like variables
  - But you cannot assign values to them!

**Preliminary Matters - 1**

- Both relation and field names MUST start with an alphabetic character
  - Then they can have 0 or more alphabetic, numeric, or _ characters.
- All characters in the name must be ASCII
- Keyword in Pig Latin is NOT case-sensitive
- LOAD = load
- Field/relation names ARE case-sensitive
  - A = load 'foo'
  - a = load 'foo'
  - Two different relations

**Preliminary Matters - 2**

- UDF names ARE case-sensitive
  - COUNT() is not the same UDF as count().
- Comments
  - SQL-style
    - --
  - Java-style
    - /* */
  
  ```
  A = load 'foo'; -- this is a single-line comment
  /* This is a multiline comment. */
  B = load '/a comment in the middle '/bar';
  ```

**Input and Output: Load**

- Tab separated file store in HDFS
  
  ```
  dive = load '/data/examples/NYSE_dividends'
  ```

- All of the relative path to the data file is based on your home directory of HDFS
- /users/yourlogin
- Using complete path name

- Data stored in HBase
  
  ```
  dive = load 'NYSE_dividends' using HBaseStorage()
  ```
PigStorage()

- Default data loader for PIG
  - Indicating the separator
    ```
divs = load 'NYSE_dividends' using PigStorage(',');
```
- Specifying the schema
  ```
divs = load 'NYSE_dividends' AS (exchange, symbol, date, dividends);
```
- Loads data from file and directory
- Allows using globs and multiple file loading

Input and Output: Store

- Pig stores your data on HDFS in a tab-delimited file using PigStorage()
  ```
store processed into '/data/examples/processed';

store processed into 'hdfs://nn.acme.com/data/examples/processed';
```
- Pig stores your data to HBase
  ```
store processed into 'processed' using HBaseStorage()
```

Input and Output:

- PigStorage() takes an argument to indicate the separator.
  ```
store processed into 'processed' using PigStorage('');
```
- Print on the screen.
  ```
dump processed;
```

Apache Pig

Relational Operator: foreach

A = load 'input' as (user:chararray, id:long, address:chararray, phone:chararray, preferences:map[]);
B = foreach A generate user, id

Relational Operations: foreach

- Pig's projection operator
- Takes a set of expressions
- Applies them to every record in the data pipeline
- Pass to the next operator

prices = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close, volume, adj_close);
gain = foreach prices generate close – open;
gain2 = foreach prices generate $6-$3;

- Field reference
  - Name of the fields
  - Positional references
    - $ dollar sign + start from 0

- Relation gain and gain2 will have the same values.
- Positional style reference is useful when the schema is unknown.
Relational Operations: `foreach`

- Expressions in `foreach`
  - `prices = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close, volume, adj_close);`
  - `alldata = foreach prices generate *;`
  - `beginning = foreach prices generate ..open;`
  - `middle = foreach prices generate open..close;`
  - `end = foreach prices generate volume..;`

- Arithmetic operators
  - Integers and floating-point numbers
    - `+,-,*,/`
    - Return value of their own type
    - `5+2 = 2`
    - `5.0/2.0 = 2.5`
    - `null + ? = ?`
    - `If either subexpression is null, the resulting expression is null`

- Binary condition operator
  - `2 == 2 ? 1 : 4`
  - `2 == 3 ? 1 : 4`
  - `null == 2 ? 1 : 4`
  - `2 == 2 ? 'fred' : 4`

Using `foreach` for the complex types

- Extract data from complex types
  - Use the projection operators
    - `#` for maps
      - `bball = load 'baseball' as (name:chararray, team:chararray, position:bag{t: (pchararray)}, bat:map[]);`
      - `avgs = foreach bball generate bat#'batting_average';`
  - `for` tuples
    - `A = load 'input' as (t:tuple(x:int, y:int));`
    - `b = foreach A generate t.x, t.y;`

Referencing Fields with that are Tuples

- Extract data from complex types
  - Use the projection operators
    - `#` for maps
      - `cat data;`
      - `((3,8,9), (4,5,6))`
      - `((1,4,7), (3,7,5))`
      - `((2,5,8), (9,5,8));`
      - `A = LOAD 'data' AS (t1:tuple(t1a:int, t1b:int, t1c:int), t2:tuple(t2a:int, t2b:int, t2c:int));`
      - `DUMP A;`
      - `((3,8,9), (4,5,6))`
      - `((1,4,7), (3,7,5))`
      - `((2,5,8), (9,5,8));`
    - `X = FOREACH A GENERATE t1.t1a, t2.t2b;`
    - `DUMP X;`
      - `(3,4)`
      - `(1,3)`
      - `(2,9)`
UDFs in **foreach**

- User Defined Functions (UDFs) can be invoked in **foreach**
  - Evaluation functions
- Takes one record at a time and produces one output
- Input or output can be a bag
  - A bag of records

UDFs in **foreach**

- **UPPER()**, **SUM()**

```sql
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
upped = foreach divs generate UPPER(symbol) as symbol, dividends;
grpd = group upped by symbol;
sums = foreach grpd generate group, SUM(upped.dividends)
```

Output of **foreach**

- Projection
  - **foreach** ... generate

```sql
A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;

(1,2,3)
(4,2,1)
(4,3,2)
(7,2,5)
(8,4,3)

X = FOREACH A GENERATE *

DUMP X;

(1,2,3)
(4,2,1)
(4,3,2)
(7,2,5)
(8,4,3)
```

Naming fields in **foreach**

- Result of each **foreach** statement is a new **tuple**
  - With different schema

```sql
X = FOREACH A GENERATE a1+a2 AS f1:int;

DESCRIBE X;

x: {f1: int}

DUMP X;

(3)
(6)
(11)
(7)
(9)
(12)
```

Naming fields in **foreach**

- Once any expression is applied...
  - Pig does not assign a name to the field
  - Result field will be nameless
  - You can assign a name explicitly
  - You can use a positional parameter

```sql
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray, date:chararray, dividends:float);
in_cents = foreach divs generate dividends * 100.0 as dividend, dividends * 100.0;

describe in_cents;

in_cents:(dividend: double, double)```
Apache Pig
Relational Operator: filter

Using Boolean operators in filter

- Pig allows us to combine multiple predicates into one by using the Boolean operators and and or.
- Reverse the outcome of any predicate by using the Boolean not operator.
- Pig follows java's regular expression format.

Using null in filters

- "X == null"
  - Does not result in false or true
  - Results null
  - Filters will not pass the data through
    - records: 2, null, and 4
      - filter (x == 2)
        - 2
      - filter x != 4
        - 2

- Finding values that are not null?
  - Use, is not null
group

- The **group** statement collects together records with the same key

**group by** in SQL vs. **group** in Pig Latin

- In SQL
  - Creates a group that can feed directly into one or more aggregate functions
  ```sql
  SELECT Customer, SUM(OrderPrice) 
  FROM Orders 
  WHERE Customer='Hansen' OR Customer='Jensen' 
  GROUP BY Customer 
  HAVING SUM(OrderPrice)>1500
  ```

- In Pig Latin
  - No direct connection between group and aggregate functions
  - Just executes "group"

Example

A = load 'student' AS (name:chararray, age:int, gpa:float);

```pig
DESCRIBE A;
```

```plaintext
A: {name:chararray, age:int, gpa:float}
```

```pig
DUMP A;
```

```plaintext
(John,18,4.0) 
(Mary,19,3.8) 
(Bill,20,3.9) 
(Joe,18,3.8)
```

B = GROUP A BY age;

```pig
DESCRIBE B;
```

```plaintext
B: {group:int, A: {name:chararray, age:int, gpa:float}}
```

```pig
ILLUSTRATE B;
```

```plaintext
------------------------------------------------------------
<table>
<thead>
<tr>
<th>B</th>
<th>group: int</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>18</td>
<td>{(John, 18, 4.0), (Joe, 18, 3.8)}</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>{(Mary, 19, 3.8)}</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>{(Bill, 20, 3.9)}</td>
</tr>
</tbody>
</table>
------------------------------------------------------------
```

```pig
DUMP B;
```

```plaintext
(18,{(John,18,4.0),(Joe,18,3.8)}) 
(19,{(Mary,19,3.8)}) 
(20,{(Bill,20,3.9)})
```

Results of **group by**

- The records from the **group by** statement:
  - The key field
  - Named "group"
  - The bag of collected records
  - Can be named for the alias
  - Bag inherits the schema of original relations

Skewed results

- We cannot expect that the number of values per key is distributed evenly over the reducers.
  - E.g. group the index of web pages by the base URL.
  - Certain values such as google.com will appear far more frequently.
  - Some reducers get far more data than others.
  - Your MapReduce job will not finish until all your reducers have finished.

Skew reducing in Pig for **group**

- Using Hadoop’s combiner
  - Pig’s operators can indicate when they can work with the combiner by implementing the Algebraic interface.
Apache Pig

Relational Operator: **order by**

- The `order` statement sorts your data
- Provides total order
- Not only in each partition
- `order` causes your data pipeline to go through a reduce phase

---

Skew reducing for `order`

- Skew of the values in data is very common with `order`
- Pig balances the output across reducers
  - Samples the input of the order statement to get an estimate of the key distribution
  - Builds a partitioner that produces a balanced total order
- Example
  - Sampled data (a,b,e,e,e,e,e,m,q,r,z) and three reducers
  - Partitioner decides
    - a ~ b go to reducer 1 / e goes to reducer 2 / m ~ z go to reducer 3