# CS523 - BDT
# Big Data Technologies

## Advanced Pig
## (Building High-Level Dataflows over Map-Reduce)

# Validation and nulls

- In Pig, if the value cannot be cast to the type declared in the schema, it will substitute a null value.
- As an example consider the following input for the weather data, which has an "e" character in place of an int.

```
1950 0   1
1950 22  1
1950 e   1
1949 111 1
1949 78  1
```

- Pig handles the corrupt line by producing a null for the offending value, which is displayed as the absence of a value when dumped to screen (and also when saved using STORE):

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'
>>   AS (year:chararray, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,,1)
(1949,111,1)
(1949,78,1)
```

# Dealing with Corrupt Data

- For large datasets, it is very common to have corrupt, invalid, or unexpected data, and it is generally infeasible to incrementally fix every unparsable record.
- Instead, we can pull out all of the invalid records in one go so we can take action on them, perhaps by fixing our program or by filtering them out:

```
grunt> corrupt_records = FILTER records BY temperature is null;
grunt> DUMP corrupt_records;
(1950,,1)
```

- We can find the number of corrupt records as:

```
grunt> grouped = GROUP corrupt_records ALL;
grunt> all_grouped = FOREACH grouped GENERATE group, COUNT(corrupt_records);
grunt> DUMP all_grouped;
(all,1)
```

# SPLIT Operator

- Another useful technique is to use the SPLIT operator to partition the data into "good" and "bad" relations, which can then be analyzed separately:

```
grunt> SPLIT records INTO good_records IF temperature is not null,
>>   bad_records OTHERWISE;
grunt> DUMP good_records;
(1950,0,1)
(1950,22,1)
(1949,111,1)
(1949,78,1)
grunt> DUMP bad_records;
(1950,,1)
```

# Dealing with Corrupt Data

- Sometimes corrupt data shows up as smaller tuples because fields are simply missing.

```
grunt> A = LOAD 'input/pig/corrupt/missing_fields';
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3)
(1,Scarf)
```

- You can filter these out by using the SIZE function as:

```
grunt> B = FILTER A BY SIZE(TOTUPLE(*)) > 1;
grunt> DUMP B;
(2,Tie)
(4,Coat)
(1,Scarf)
```

TOTUPLE

# Multiquery Execution

- Because DUMP is a diagnostic tool, it will always trigger execution. However, the STORE command is different.
- In interactive mode, STORE acts like DUMP and will always trigger execution, but in batch mode it will not!
  - The reason for this is efficiency.
  - In batch mode, Pig will parse the whole script to see whether there are any optimizations that could be made to limit the amount of data to be written to or read from disk.

```
A = LOAD 'input/pig/multiquery/A';
B = FILTER A BY $1 == 'banana';
C = FILTER A BY $1 != 'banana';
STORE B INTO 'output/b';
STORE C INTO 'output/c';
```

# COGROUP

- COGROUP is a generalization of GROUP.
- Instead of collecting records of one input based on a key, it collects records of *n* inputs based on a key.
  - Group two datasets together by a common attribute.
- The result is a record with a key and one bag for each input. Each bag contains all records from that input that have the given value for the key:

```
A = load 'input1' as (id:int, val:float);
B = load 'input2' as (id:int, val2:int);
C = cogroup A by id, B by id;
describe C;

C: {group: int,A: {id: int,val: float},B: {id: int,val2: int}}
```

**Use GROUP when only one relation is involved; use COGROUP when multiple relations are involved.**

# COGROUP and JOIN

- JOIN always gives a flat structure: a set of tuples.
- The COGROUP statement is similar to JOIN, but instead it creates a nested set of output tuples.
  - This can be useful if you want to exploit the structure in subsequent statements:

```
grunt> D = COGROUP A BY $0, B BY $1;
grunt> DUMP D;
(0,{},{(Ali,0)})
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Hank,2),(Joe,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

# COGROUP

- If for a particular key a relation has no matching key, the bag for that relation is empty.
- For example, since no one has bought a scarf (with ID 1), the second bag in the tuple for that row is empty.
- This is an example of an outer join, which is the default type for COGROUP.
- It can be made explicit using the OUTER keyword, making this COGROUP statement the same as the previous one:

```
grunt> D = COGROUP A BY $0 OUTER, B BY $1 OUTER;
```

# COGROUP

- You can suppress rows with empty bags by using the INNER keyword, which gives the COGROUP inner join semantics.
- The INNER keyword is applied per relation, so the following suppresses rows only when relation A has no match (dropping the unknown product 0 here):

```
grunt> E = COGROUP A BY $0 INNER, B BY $1;
grunt> DUMP E;
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Hank,2),(Joe,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

# COGROUP

```
grunt> E = COGROUP A BY $0 INNER, B BY $1;
grunt> DUMP E;
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Hank,2),(Joe,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

- We can flatten this structure to discover who bought each of the items in relation A:

```
grunt> F = FOREACH E GENERATE FLATTEN(A), B.$0;
grunt> DUMP F;
(1,Scarf,{})
(2,Tie,{(Hank),(Joe)})
(3,Hat,{(Eve)})
(4,Coat,{(Hank)})
```

# COGROUP

- Using a combination of COGROUP, INNER, and FLATTEN (which removes nesting) it's possible to simulate an (inner) JOIN:

```
grunt> G = COGROUP A BY $0 INNER, B BY $1 INNER;
grunt> H = FOREACH G GENERATE FLATTEN($1), FLATTEN($2);
grunt> DUMP H;
(2,Tie,Hank,2)
(2,Tie,Joe,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

This gives the same result as `JOIN A BY $0, B BY $1`.

# COGROUP

- Another way to think of COGROUP is as the first half of a join. The keys are collected together, but the cross product is not done.
- In fact, COGROUP plus FOREACH, where each bag is flattened, is equivalent to a join—as long as there are no null values in the keys.
- COGROUP handles null values in the keys similarly to GROUP and unlike JOIN. That is, all records with a null value in the key will be collected together.
- COGROUP is useful when you want to do join-like things but not a full join.
- Because cogroup needs to collect records with like keys together, it requires a reduce phase.

# Fragment Replicate Join

- Assume we join two datasets, one of which is considerably smaller than the other
  - For instance, suppose a dataset fits in memory

- **Fragment replicate join**
  - Syntax: append the clause USING "replicated" to a JOIN statement
  - Uses a **distributed cache** available in Hadoop
  - All mappers will have a copy of the small input
  - This is a Map-side join

# User Defined Functions(UDFs)

- One of the strong features of Pig is that the developer / analyst does not have to depend solely on the Pig-supplied operators. It is possible to extend the language by coding one's own user-defined functions (UDFs).
- If the in-built operators do not provide some functions then programmers can implement those functionalities by writing **user defined functions** using other programming languages like Java, Python, Ruby, Groovy, JavaScript etc.
- These *User Defined Functions (UDF's)* can then be embedded into a Pig Latin Script.
- Pig itself comes packaged with some built-in UDFs.
- **When you use a UDF that is not already built into Pig, you have to tell Pig where to look for that UDF. This is done via the *register* command.**

# Eval UDF

- All evaluation functions extend the Java class org.apache.pig.EvalFunc. This class uses Java generics.
- It is parameterized by the return type of your UDF.
- The core method in this class is exec.
  - It takes one record and returns one result, which will be invoked for every record that passes through your execution pipeline.
  - As input it takes a tuple, which contains all of the fields the script passes to your UDF.
  - It returns the type by which you parameterized EvalFunc.
  - For simple UDFs, this is the only method you need to implement.

# Eval UDF Example

- An EvalFunc UDF to trim leading and trailing whitespace from chararray values.

```
public class Trim extends PrimitiveEvalFunc<String, String> {
  @Override
  public String exec(String input) {
    return input.trim();
  }
}
```

- PrimitiveEvalFunc is a specialization of EvalFunc and can be used when the input is a single primitive (atomic) type.
  - For the Trim UDF, the input and output types are both of type String.

# Eval UDF Example

- The Trim UDF returns a string, which Pig translates as a chararray, as can be seen from the following:

```
grunt> DUMP A;
( pomegranate)
(banana )
(apple)
( lychee )
grunt> DESCRIBE A;
A: {fruit: chararray}
grunt> B = FOREACH A GENERATE com.hadoop.pig.Trim(fruit);
grunt> DUMP B;
(pomegranate)
(banana)
(apple)
(lychee)
grunt> DESCRIBE B;
B: {chararray}
```

# Eval UDF Example

- In general, when you write an eval function, you need to consider what the output's schema looks like.
- In the following statement, the schema of B is determined by the function my_udf:

```
B = FOREACH A GENERATE my_udf($0);
```

- If my_udf creates tuples with scalar fields, then Pig can determine B's schema through reflection.
- For complex types such as bags, tuples, or maps, Pig needs more help, and you should implement the outputSchema() method to give Pig the information about the output schema.

# Filter UDF Example

- Let's write a filter function for filtering out weather records that do not have a temperature quality reading of satisfactory (or better).
- The idea is to change this line:

```
filtered_records = FILTER records BY temperature != 9999 AND
  quality IN (0, 1, 4, 5, 9);
```

- to:

```
filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
```

- This achieves two things: it makes the Pig script a little more concise, and it encapsulates the logic in one place so that it can be easily reused in other scripts.
- When you start doing the same kind of processing over and over again then you see opportunities for reusable UDFs.

# Filter UDF Example

- Filter UDFs are all subclasses of FilterFunc, which itself is a subclass of EvalFunc.

```
public abstract class EvalFunc<T> {
  public abstract T exec(Tuple input) throws IOException;
}
```

- EvalFunc's only abstract method, exec(), takes a tuple and returns a single value, the (parameterized) type T.
- The fields in the input Tuple consist of the expressions passed to the function—in this case, a single integer.
- The Tuple class is essentially a list of objects with associated types.
- For FilterFunc, T is Boolean, so the method should return true only for those tuples that should not be filtered out.

# Filter UDF Example

- For the quality filter, we write a class, IsGoodQuality, that extends FilterFunc and implements the exec() method.
- Compile it and package it in a JAR file.

```
public class IsGoodQuality extends FilterFunc {

  @Override
  public Boolean exec(Tuple tuple) throws IOException {
    if (tuple == null || tuple.size() == 0) {
      return false;
    }
    try {
      Object object = tuple.get(0);
      if (object == null) {
        return false;
      }
      int i = (Integer) object;
      return i == 0 || i == 1 || i == 4 || i == 5 || i == 9;
    } catch (ExecException e) {
      throw new IOException(e);
    }
  }
}
```

*A FilterFunc UDF to remove records with unsatisfactory temperature quality readings*

# Filter UDF Example

- This is how we use the just created UDF:

```
REGISTER /home/cloudera/jars/Pig_UDFs.jar;

temp = LOAD 'temp' as (year:chararray,
                       temperature:int, quality:int);

filtered = FILTER temp BY temperature != 9999 AND
                          udf.IsGoodQuality(quality);
```

- Pig resolves function calls by treating the function's name as a Java classname and attempting to load a class of that name.
- When searching for classes, Pig uses a classloader that includes the JAR files that have been registered.
- When running in distributed mode, Pig will ensure that your JAR files get shipped to the cluster.

# DEFINE Operator

- Function names can be shortened by defining an alias, using the **DEFINE** operator:

```
DEFINE isGood com.hadoop.pig.udf.IsGoodQuality();
filtered = FILTER records BY temperature != 9999 AND
                                        isGood(quality);
```

- Defining an alias is a good idea if you want to use the function several times in the same script.
- It's also necessary if you want to pass arguments to the constructor of the UDF's implementation class.

# Where Your UDF Will Run?

- Writing code that will run in a parallel system presents challenges.
- A separate instance of your UDF will be constructed and run in each map or reduce task.
- It is not possible to share state across these instances because they may not all be running at the same time.
- There will be only one instance of your UDF per map or reduce task, so you can share state within that context.
  - Assuming there is one instance of your UDF in the script.
  - Each reference to a UDF in a script becomes a separate instance on the backend, even if they are placed in the same map or reduce task.

# Where Your UDF Will Run?

- When writing code for a parallel system, you must remember the power of parallelism. Operations that are acceptable in serial programs may no longer be advisable.
- Consider a UDF that, when it first starts, connects to a database server to download a translation table.
- In a serial or low-parallelism environment, this is a reasonable approach.
- But if you have 10,000 map tasks in your job and they all connect to your database at once, you will most likely hear from your DBA, and the conversation is unlikely to be pleasant.

# Dynamic Invokers

- Sometimes you want to use a function that is provided by a Java library, but without going to the effort of writing a UDF.
- Dynamic invokers allow you to do this by calling Java methods directly from a Pig script.
- The trade-off is that method calls are made via reflection, which can impose significant overhead when calls are made for every record in a large dataset.
- So for scripts that are run repeatedly, a dedicated UDF is normally preferred.

# Dynamic Invokers Example

- Let's define and use a trim UDF that uses the Apache Commons Lang StringUtils class:

```
grunt> DEFINE trim
InvokeForString('org.apache.commons.lang.String
Utils.trim', 'String');
grunt> B = FOREACH A GENERATE trim(fruit);
grunt> DUMP B;
(pomegranate)
(banana)
(apple)
(lychee)
```

# Dynamic Invokers Example

- The InvokeForString invoker is used because the return type of the method is a String.
- (There are also InvokeForInt, InvokeForLong, InvokeForDouble, and InvokeForFloat invokers.)
- The first argument to the invoker constructor is the fully qualified method to be invoked.
- The second is a space-separated list of the method argument classes.

# Before writing a custom UDF

- Visit…
  - **PiggyBank**
    - Piggybank is a place for Pig users to share their functions
      - LOAD/STORE functions (e.g. from CSV, XML, Hive RCFiles etc.)
      - datetime, text functions, math, stats functions
  - **DataFu** (Linkedin's collection of UDFs)
    - Hadoop library for large scale data processing
      - Statistics functions (quantiles, variance, etc.)
      - Convenient bag functions (intersection, union etc.)
      - Utility functions (assertions, random numbers, MD5, distance between lat/long pair), PageRank, Sessionization, Link analysis etc.

# Piggybank

- *Piggybank* is Pig's repository of user-contributed functions.
- Piggybank functions are distributed as part of the Pig distribution, but they are not built-in.
- Piggybank UDFs are not included in the Pig JAR, and thus you have to register them manually in your script.
  - E.g. let's say you want to use the Reverse UDF provided in Piggybank.

```
register '/usr/lib/pig/piggybank.jar';

users = LOAD 'users' AS (name:chararray, age:int);

backwardsName = FOREACH users GENERATE
   org.apache.pig.piggybank.evaluation.string.Reverse(name);
```

# Parallelism

- When running in MapReduce mode, it's important that the degree of parallelism matches the size of the dataset.
- By default, Pig sets the number of reducers by looking at the size of the input and using one reducer per 1 GB of input, up to a maximum of 999 reducers.
- You can override these parameters by setting **pig.exec.reducers.bytes.per.reducer** (the default is 1,000,000,000 bytes) and **pig.exec.reducers.max** (the default is 999)

# PARALLEL Clause

- To explicitly set the number of reducers you want for each job, you can use a PARALLEL clause for operators that run in the reduce phase.
- These include all the grouping and joining operators (GROUP, COGROUP, JOIN, CROSS), as well as DISTINCT and ORDER.
- The following line sets the number of reducers to 30 for the GROUP:

```
gpd_Rec = GROUP records BY year PARALLEL 30;
```

- Alternatively, you can set the default_parallel option, and it will take effect for all subsequent jobs:

```
grunt> set default_parallel 30
```

# Setting the Partitioner

- Pig allows you to set the partitioner. To do this, you need to tell Pig which Java class to use to partition your data.
- This class must extend Hadoop's org.apache.hadoop.mapreduce.Partitioner<KEY,VALUE>

- For Example:

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

- Operators that reduce data can take the partition clause.
- These operators are cogroup, cross, distinct, group, and join.

# Pig Use Cases

- **Extract Transform Load (ETL)**
  - Ex: Processing large amounts of log data
  - clean bad entries, join with other data-sets
- **Research of "raw" information**
  - Ex. User Audit Logs
  - Schema maybe unknown or inconsistent
  - Data Scientists and Analysts may like Pig's data transformation paradigm

- Pig has various user groups, for instance, Yahoo, Twitter, AOL , Sales force, LinkedIn and Nokia.
- Apache Pig is an integral part of the "People You May Know" data product at LinkedIn.
- PayPal is a major contributor to the Pig -Eclipse project and uses Apache Pig to analyze transactional data and prevent fraud.

# Pig – Final Words

- Pig's sweet spot is its ability to process terabytes of data in response to a half-dozen lines of Pig Latin issued from the console.
- Indeed, it was created at Yahoo! to make it easier for researchers and engineers to mine the huge datasets there.
- Pig is very supportive of a programmer writing a query, since it provides several commands for introspecting the data structures in your program as it is written.
- In some cases, Pig doesn't perform as well as programs written in MapReduce. However, the gap is narrowing with each release, as the Pig team implements sophisticated algorithms for applying Pig's relational operators.
- It's fair to say that unless you are willing to invest a lot of effort optimizing Java MapReduce code, writing queries in Pig Latin will save you time.