# BIG DATA ANALYTICS

## Apache Pig

## Data Processing Operators

*https://www.youtube.com/c/RASINENIMADANAMOHANA*

# Data Processing Operators: *Outline*

- Loading and Storing Data 
- Filtering Data
- Grouping and Joining Data
- Sorting Data
- Combining and Splitting Data

# Data Processing Operators: Loading and Storing Data

- Throughout the examples discussed so far, we have seen how to load data from external storage for processing in Pig.
- Storing the results is straightforward, too.
- The following is an example of using **PigStorage** to store tuples as plain-text values separated by a colon character:

```
grunt> STORE A INTO 'out' USING PigStorage(':');

grunt> cat out

Joe:cherry:2

Ali:apple:3

Joe:banana:2

Eve:apple:7
```

# Data Processing Operators: *Outline*

- Loading and Storing Data
- Filtering Data 
- Grouping and Joining Data
- Sorting Data
- Combining and Splitting Data

# Data Processing Operators: Filtering Data

- Once you have some data loaded into a relation, often the next step is to filter it to remove the data that we are not interested in.
- By filtering early in the processing pipeline, we minimize the amount of data flowing through the system, which can improve efficiency.

# Data Processing Operators: Filtering Data

## FOREACH...GENERATE

- We have already seen how to remove rows from a relation using the **`FILTER`** operator with simple expressions and a **UDF**.
- The **`FOREACH...GENERATE`** operator is used to act on every row in a relation.
- It can be used to remove fields or to generate new ones.
- In the following example, we do both:

# Data Processing Operators: Filtering Data

## FOREACH...GENERATE

Example:

```
grunt> DUMP A;
(Joe,cherry,2)
(Ali,apple,3)
(Joe,banana,2)
(Eve,apple,7)
grunt> B = FOREACH A GENERATE $0, $2+1, 'Constant';
grunt> DUMP B;
(Joe,3,Constant)
(Ali,4,Constant)
(Joe,3,Constant)
(Eve,8,Constant)
```

# Data Processing Operators: Filtering Data

## FOREACH…GENERATE

- Here we have created a new relation, **B**, with three fields. Its first field is a projection of the first field (`$0`) of **A**. **B**'s second field is the third field of **A** (**$2**) with **1** added to it. `B`'s third field is a constant field (every row in **B** has the same third field) with the `chararray` value `Constant`.

# Data Processing Operators: Filtering Data

## FOREACH...GENERATE

- The **FOREACH...GENERATE** operator has a nested form to support more complex processing.
- In the following example, we compute various statistics for the weather dataset:

```
-- year_stats.pig
REGISTER pig-examples.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
records = LOAD 'input/ncdc/all/19{1,2,3,4,5}0*'
  USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,16-19,88-92,93-93')
  AS (usaf:chararray, wban:chararray, year:int, temperature:int, quality:int);

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

# Data Processing Operators: Filtering Data

## FOREACH...GENERATE

- In the following example, we compute various statistics for the weather dataset: cont'd.

```
year_stats = FOREACH grouped_records {
  uniq_stations = DISTINCT records.usaf;
  good_records = FILTER records BY isGood(quality);
  GENERATE FLATTEN(group), COUNT(uniq_stations) AS station_count,
    COUNT(good_records) AS good_record_count, COUNT(records) AS record_count;
}

DUMP year_stats;
```

# Data Processing Operators: Filtering Data

## FOREACH...GENERATE

Running it on a few years' worth of data, we get the following:

```
(1920,8L,8595L,8595L)

(1950,1988L,8635452L,8641353L)

(1930,121L,89245L,89262L)

(1910,7L,7650L,7650L)

(1940,732L,1052333L,1052976L)
```

The fields are year, number of unique stations, total number of good readings, and total number of readings. We can see how the number of weather stations and readings grew over time.

# Data Processing Operators: Filtering Data

## STREAM

The `STREAM` operator allows us to transform data in a relation using an external program or script.

It is named by similarity with `Hadoop Streaming`, which provides a similar capability for MapReduce.

# Data Processing Operators: Filtering Data

## STREAM

**STREAM** can use built-in commands with arguments.

Here is an example that uses the **Unix cut** command to extract the second field of each tuple in **A**. Note that the command and its arguments are enclosed in backticks:

```
grunt> C = STREAM A THROUGH `cut -f 2`;

grunt> DUMP C;

(cherry)

(apple)

(banana)

(apple)
```

# Data Processing Operators: Filtering Data

## STREAM

- The `STREAM` operator uses `PigStorage` to serialize and deserialize relations to and from the program's standard input and output streams.
- Tuples in `A` are converted to tab delimited lines that are passed to the script.
- The output of the script is read one line at a time and split on tabs to create new tuples for the output relation `C`.
- We can provide a custom serializer and deserializer by subclassing `PigStreamingBase` (in the `org.apache.pig` package), then using the `DEFINE` operator.

# Data Processing Operators: Filtering Data

## STREAM

- Pig streaming is most powerful when we write custom processing scripts.
- The following Python script filters out bad weather records:

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  (year, temp, q) = line.strip().split()
  if (temp != "9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

# Data Processing Operators: Filtering Data

## STREAM

- To use the script, we need to ship it to the cluster. This is achieved via a **DEFINE** clause, which also creates an alias for the **STREAM** command.
- The **STREAM** statement can then refer to the alias, as the following `Pig script` shows:

```
-- max_temp_filter_stream.pig
DEFINE is_good_quality `is_good_quality.py`
  SHIP ('ch16-pig/src/main/python/is_good_quality.py');
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = STREAM records THROUGH is_good_quality
  AS (year:chararray, temperature:int);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

# Data Processing Operators: *Outline*

- Loading and Storing Data
- Filtering Data
- Grouping and Joining Data 
- Sorting Data
- Combining and Splitting Data

# Data Processing Operators: Grouping and Joining Data

- Joining datasets in **MapReduce** takes some work on the part of the programmer, whereas **Pig** has very good built-in support for join operations, making it much more approachable. Since the large datasets that are suitable for analysis by **Pig** (and **MapReduce** in general) are not normalized, however, joins are used more infrequently in **Pig** than they are in **SQL**.

# Data Processing Operators: Grouping and Joining Data

## JOIN

Example of an inner join:

Consider the relations **A** and **B**:

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)
grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

# Data Processing Operators: Grouping and Joining Data

## JOIN

Example of an inner join:

We can join the two relations on the numerical (identity) field in each:

```
grunt> C = JOIN A BY $0, B BY $1;
grunt> DUMP C;
(2,Tie,Hank,2)
(2,Tie,Joe,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

This is a classic inner join, where each match between the two relations corresponds to a row in the result. (It's actually an equijoin because the join predicate is equality.) The result's fields are made up of all the fields of all the input relations.

# Data Processing Operators: Grouping and Joining Data

## JOIN

Example of an inner join:

- WE should use the general join operator when all the relations being joined are too large to fit in memory. If one of the relations is small enough to fit in memory, we can use a special type of join called a ***fragment replicate join***, which is implemented by distributing the small input to all the mappers and performing a map-side join using an in-memory lookup table against the (fragmented) larger relation.
- There is a special syntax for telling Pig to use a fragment replicate join:

```
grunt> C = JOIN A BY $0, B BY $1 USING 'replicated';
```

The first relation must be the large one, followed by one or more small ones (all of which must fit in memory).

# Data Processing Operators: Grouping and Joining Data

## JOIN

- Pig also supports outer joins using a syntax that is similar to SQL's.

For example:

```
grunt> C = JOIN A BY $0 LEFT OUTER, B BY $1;
grunt> DUMP C;
(1,Scarf,,)
(2,Tie,Hank,2)
(2,Tie,Joe,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

# Data Processing Operators: Grouping and Joining Data

## COGROUP

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

For example:

```
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)})
```

# Data Processing Operators: Grouping and Joining Data

## COGROUP

- `COGROUP` generates a tuple for each unique grouping key.
- The first field of each tuple is the key, and the remaining fields are bags of tuples from the relations with a matching key.
- The first bag contains the matching tuples from relation `A` with the same key.
- Similarly, the second bag contains the matching tuples from relation `B` with the same key.

# Data Processing Operators: Grouping and Joining Data

## 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:

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

# Data Processing Operators: Grouping and Joining Data

## COGROUP

- We 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)})
```

# Data Processing Operators: Grouping and Joining Data

## COGROUP

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)})
```

# Data Processing Operators: Grouping and Joining Data

## 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**.

# Data Processing Operators: Grouping and Joining Data

## COGROUP

Example of a join in Pig using the cut UDF, in a script for calculating the maximum temperature for every station over a time period controlled by the input:

```
-- max_temp_station_name.pig
REGISTER pig-examples.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();

stations = LOAD 'input/ncdc/metadata/stations-fixed-width.txt'
  USING com.hadoopbook.pig.CutLoadFunc('1-6,8-12,14-42')
  AS (usaf:chararray, wban:chararray, name:chararray);

trimmed_stations = FOREACH stations GENERATE usaf, wban, TRIM(name);

records = LOAD 'input/ncdc/all/191*'
  USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,88-92,93-93')
  AS (usaf:chararray, wban:chararray, temperature:int, quality:int);

filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
grouped_records = GROUP filtered_records BY (usaf, wban) PARALLEL 30;
max_temp = FOREACH grouped_records GENERATE FLATTEN(group),
  MAX(filtered_records.temperature);
max_temp_named = JOIN max_temp BY (usaf, wban), trimmed_stations BY (usaf, wban)
  PARALLEL 30;
max_temp_result = FOREACH max_temp_named GENERATE $0, $1, $5, $2;

STORE max_temp_result INTO 'max_temp_by_station';
```

# Data Processing Operators: Grouping and Joining Data

## COGROUP

Here are a few results for the 1910s:

```
228020 99999 SORTAVALA 322

029110 99999 VAASA AIRPORT 300

040650 99999 GRIMSEY 378
```

This query could be made more efficient by using a fragment replicate join, as the station metadata is small.

# Data Processing Operators: Grouping and Joining Data

## CROSS

**Pig Latin** includes the cross-product operator (also known as the Cartesian product), `CROSS`, which joins every tuple in a relation with every tuple in a second relation (and with every tuple in further relations, if supplied).

# Data Processing Operators: Grouping and Joining Data

## CROSS

The size of the output is the product of the size of the inputs, potentially making the output very large:

```
grunt> I = CROSS A, B;

grunt> DUMP I;
```

```
(2,Tie,Joe,2)
(2,Tie,Hank,4)
(2,Tie,Ali,0)
(2,Tie,Eve,3)
(2,Tie,Hank,2)
(4,Coat,Joe,2)
(4,Coat,Hank,4)
(4,Coat,Ali,0)
(4,Coat,Eve,3)
(4,Coat,Hank,2)
(3,Hat,Joe,2)
(3,Hat,Hank,4)
(3,Hat,Ali,0)
(3,Hat,Eve,3)
(3,Hat,Hank,2)
(1,Scarf,Joe,2)
(1,Scarf,Hank,4)
(1,Scarf,Ali,0)
(1,Scarf,Eve,3)
(1,Scarf,Hank,2)
```

# Data Processing Operators: Grouping and Joining Data

## CROSS

- When dealing with large datasets, we should try to avoid operations that generate intermediate representations that are quadratic (or worse) in size.
- Computing the cross product of the whole input dataset is rarely needed, if ever.

# Data Processing Operators: Grouping and Joining Data

## GROUP

- Where `COGROUP` groups the data in two or more relations, the `GROUP` statement groups the data in a single relation.
- `GROUP` supports grouping by more than equality of keys: we can use an expression or user-defined function as the group key.

# Data Processing Operators: Grouping and Joining Data

## GROUP

For example, consider the following relation **A**:

```
grunt> DUMP A;
(Joe,cherry)
(Ali,apple)
(Joe,banana)
(Eve,apple)
```

Let's group by the number of characters in the second field:

```
grunt> B = GROUP A BY SIZE($1);
grunt> DUMP B;
(5,{(Eve,apple),(Ali,apple)})
(6,{(Joe,banana),(Joe,cherry)})
```

# Data Processing Operators: Grouping and Joining Data

## GROUP

- **GROUP** creates a relation whose first field is the grouping field, which is given the alias group.
- The second field is a bag containing the grouped fields with the same schema as the original relation (in this case, **A**).
- There are also two special grouping operations: **ALL** and **ANY**.
- **ALL** groups all the tuples in a relation in a single group, as if the **GROUP** function were a constant:

```
grunt> C = GROUP A ALL;

grunt> DUMP C;

(all,{(Eve,apple),(Joe,banana),(Ali,apple),(Joe,cherry)})
```

# Data Processing Operators: Grouping and Joining Data

## GROUP

- Note that there is no **BY** in this form of the **GROUP** statement.
- The **ALL** grouping is commonly used to count the number of tuples in a relation.
- The **ANY** keyword is used to group the tuples in a relation randomly, which can be useful for sampling.

# Data Processing Operators: *Outline*

- Loading and Storing Data
- Filtering Data
- Grouping and Joining Data
- Sorting Data 
- Combining and Splitting Data

# Data Processing Operators: Sorting Data

- Relations are unordered in Pig.
- Consider a relation **A**:

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
```

- There is no guarantee which order the rows will be processed in. In particular, when retrieving the contents of **A** using **DUMP** or **STORE**, the rows may be written in any order.
- If we want to impose an order on the output, you can use the **ORDER** operator to sort a relation by one or more fields. The default sort order compares fields of the same type using the natural ordering, and different types are given an arbitrary, but deterministic, ordering (a tuple is always "less than" a bag, for example).

# Data Processing Operators: Sorting Data

The following example sorts **A** by the first field in ascending order and by the second field in descending order:

```
grunt> B = ORDER A BY $0, $1 DESC;
grunt> DUMP B;
(1,2)
(2,4)
(2,3)
```

Any further processing on a sorted relation is not guaranteed to retain its order. For example:

```
grunt> C = FOREACH B GENERATE *;
```

Even though relation **C** has the same contents as relation **B**, its tuples may be emitted in any order by a **DUMP** or a **STORE**. It is for this reason that it is usual to perform the **ORDER** operation just before retrieving the output.

# Data Processing Operators: Sorting Data

- The **LIMIT** statement is useful for limiting the number of results as a quick-and-dirty way to get a sample of a relation. (Although random sampling using the **SAMPLE** operator, or prototyping with the **ILLUSTRATE** command, should be preferred for generating more representative samples of the data.)
- It can be used immediately after the **ORDER** statement to retrieve the first ***n*** tuples. Usually, **LIMIT** will select any ***n*** tuples from a relation, but when used immediately after an **ORDER** statement, the order is retained (in an exception to the rule that processing a relation does not retain its order):

```
grunt> D = LIMIT B 2;
grunt> DUMP D;
(1,2)
(2,4)
```

If the limit is greater than the number of tuples in the relation, all tuples are returned (so **LIMIT** has no effect).

# Data Processing Operators: *Outline*

- Loading and Storing Data
- Filtering Data
- Grouping and Joining Data
- Sorting Data
- Combining and Splitting Data 

# Data Processing Operators: Combining and Splitting Data

- Sometimes we have several relations that you would like to combine into one.
- For this, the **UNION** statement is used.
- Example:

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
grunt> DUMP B;
(z,x,8)
(w,y,1)
grunt> C = UNION A, B;
grunt> DUMP C;
(2,3)
(z,x,8)
(1,2)
(w,y,1)
(2,4)
```

# Data Processing Operators: Combining and Splitting Data

- `C` is the union of relations `A` and `B`, and because relations are unordered, the order of the tuples in `C` is undefined.
- Also, it's possible to form the union of two relations with different schemas or with different numbers of fields, as we have done here.
- Pig attempts to merge the schemas from the relations that `UNION` is operating on. In this case, they are incompatible, so `C` has no schema:

```
grunt> DESCRIBE A;
A: {f0: int,f1: int}
grunt> DESCRIBE B;
B: {f0: chararray,f1: chararray,f2: int}
grunt> DESCRIBE C;
Schema for C unknown.
```

# Data Processing Operators: Combining and Splitting Data

- If the output relation has no schema, our script needs to be able to handle tuples that vary in the number of fields and/or types.
- The **SPLIT** operator is the opposite of **UNION**: it partitions a relation into two or more relations.

Example:

```
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)
```

# Data Processing Operators: Combining and Splitting Data

Example: cont'd.

```
grunt> corrupt_records = FILTER records BY temperature is null;
grunt> DUMP corrupt_records;
(1950,,1)
grunt> grouped = GROUP corrupt_records ALL;
grunt> all_grouped = FOREACH grouped GENERATE group,
COUNT(corrupt_records);
grunt> DUMP all_grouped;
(all,1)
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)
```