Chapter 11

Pig

Learning Objectives

- Defining and Running Pig
- Pig Latin
- User-Defined Functions
- Data Processing Operators
- Pig in Practice

What Is Pig – 1

- Pig is a scripting language for exploring large datasets.
- Pig is made up of two pieces:
  - The language used to express data flows, called Pig Latin.
  - The execution environment to run Pig Latin programs.
- There are currently two environments:
  - Local execution in a single JVM
  - Distributed execution on a Hadoop cluster.

What Is Pig – 2

- Pig can process terabytes of data simply by issuing a half-dozen lines of Pig Latin from the console.
- Pig is very supportive of a programmer writing a query.
- Pig is designed to be extensible.
- Pig is designed to process large portions of a large dataset.
- The execution environment to run Pig Latin programs.

Run Pig

Script

Pig can run a script file that contains Pig commands. For example, `pig script.pig` runs the commands in the local file *script.pig*. Alternatively, for very short scripts, you can use the `-e` option to run a script specified as a string on the command line.

Grunt

Grunt is an interactive shell for running Pig commands. Grunt is started when no file is specified for Pig to run and the `-e` option is not used. It is also possible to run Pig scripts from within Grunt using `run` and `exec`.

Embedded

You can run Pig programs from Java using the `PigServer` class, much like you can use JDBC to run SQL programs from Java. For programmatic access to Grunt, use `PigRunner`.

A Pig Example – p. 371

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

Illustrate

```
grunt> ILLUSTRATE max_temp;
-------------------------------------------------------------------------------
| records     | year:chararray      | temperature:int      | quality:int      |
-------------------------------------------------------------------------------
|             | 1949                | 78                   | 1                |
|             | 1949                | 111                  | 1                |
|             | 1949                | 9999                 | 1                |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| filtered_records   | year:chararray    | temperature:int    | quality:int     |
-------------------------------------------------------------------------------
|                    | 1949              | 78                 | 1               |
|                    | 1949              | 111                | 1               |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| grouped_records  | group:chararray   | filtered_records:bag{:tuple(year:chararray, |
                                              temperature:int,quality:int)} |
-------------------------------------------------------------------------------
|                  | 1949              | {(1949, 78, 1), (1949, 111, 1)}           |
-------------------------------------------------------------------------------
-------------------------------------------------
| max_temp     | group:chararray      | :int      |
-------------------------------------------------
|              | 1949                 | 111       |
-------------------------------------------------
```

Pig/Pig Latin vs. Relational/SQL

Difference between Pig Latin and SQL:

- Pig Latin is a data flow programming language, whereas SQL is a declarative programming language.
- RDBMSs store data in tables, with tightly predefined schemas; Pig is more relaxed about the data that it processes: you can define a schema at runtime.
- Pig's support for complex, nested data structures differentiates it from SQL, which operates on flatter data structures.
- RDBMSs have several features to support online, low-latency queries, such as transactions and indexes, that are absent in Pig.

Pig Latin – Structure

Statements that have to be terminated with a semicolon can be split across multiple lines for readability:

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
```

Pig Latin has two forms of comments. Double hyphens are single-line comments. Everything from the first hyphen to the end of the line is ignored by the Pig Latin interpreter:

```
-- My program
DUMP A; -- What's in A?
```

C-style comments are more flexible since they delimit the beginning and end of the comment block with /* and */ markers. They can span lines or be embedded in a single line:

```
/*
 * Description of my program spanning
 * multiple lines.
 */
A = LOAD 'input/pig/join/A';
B = LOAD 'input/pig/join/B';
C = JOIN A BY $0, /* ignored */ B BY $1;
DUMP C;
```

Pig Latin Commands

Table 11-4. Pig Latin commands

Category	Command	Description
Hadoop Filesystem	`cat`	Prints the contents of one or more files
	`cd`	Changes the current directory
	`copyFromLocal`	Copies a local file or directory to a Hadoop filesystem
	`copyToLocal`	Copies a file or directory on a Hadoop filesystem to the local filesystem
	`cp`	Copies a file or directory to another directory
	`fs`	Accesses Hadoop's filesystem shell
	`ls`	Lists files
	`mkdir`	Creates a new directory
	`mv`	Moves a file or directory to another directory
	`pwd`	Prints the path of the current working directory
	`rm`	Deletes a file or directory
	`rmf`	Forcibly deletes a file or directory (does not fail if the file or directory does not exist)
Hadoop MapReduce	`kill`	Kills a MapReduce job
Utility	`exec`	Runs a script in a new Grunt shell in batch mode
	`help`	Shows the available commands and options
	`quit`	Exits the interpreter
	`run`	Runs a script within the existing Grunt shell
	`set`	Sets Pig options and MapReduce job properties
	`sh`	Run a shell command from within Grunt

Pig Latin Schemas – 1

A relation in Pig may have an associated schema, which gives the fields in the relation names and types. We've seen how an AS clause in a LOAD statement is used to attach a schema to a relation:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year:int, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: int,temperature: int,quality: int}
```

It's possible to omit type declarations completely, too:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year, temperature, quality);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: bytearray,quality: bytearray}
```

Pig Latin Schemas – 2

You don't need to specify types for every field; you can leave some to default to `bytearray`, as we have done for `year` in this declaration:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: int,quality: int}
```

hand, the schema is entirely optional and can be omitted by not specifying an AS clause:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt';
grunt> DESCRIBE records;
Schema for records unknown.
```

Fields in a relation with no schema can be referenced using only positional notation: `$0` refers to the first field in a relation, `$1` to the second, and so on. Their types default to `bytearray`:

```
grunt> projected_records = FOREACH records GENERATE $0, $1, $2;
grunt> DUMP projected_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
grunt> DESCRIBE projected_records;
projected_records: {bytearray,bytearray,bytearray}
```

Validation and Nulls – 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)
```

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 (because they indicate that we have made a mistake) or by filtering them out (because the data is genuinely unusable):

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

Validation and Nulls – 2

We can find the number of corrupt records using the following idiom for counting the number of rows in a relation:

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

("GROUP" on page 406 explains grouping and the ALL operation in more detail.)

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 IF temperature is null;
grunt> DUMP good_records;
(1950,0,1)
(1950,22,1)
(1949,111,1)
(1949,78,1)
grunt> DUMP bad_records;
(1950,,1)
```

Validation and Nulls – 3

Going back to the case in which temperature's type was left undeclared, the corrupt data cannot be detected easily, since it doesn't surface as a null:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'
>>   AS (year:chararray, temperature, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,e,1)
(1949,111,1)
(1949,78,1)
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>>   (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grunt> grouped_records = GROUP filtered_records BY year;
grunt> max_temp = FOREACH grouped_records GENERATE group,
>>   MAX(filtered_records.temperature);
grunt> DUMP max_temp;
(1949,111.0)
(1950,22.0)
```

What happens in this case is that the temperature field is interpreted as a bytearray, so the corrupt field is not detected when the input is loaded. When passed to the MAX function, the temperature field is cast to a double, since MAX works only with numeric types. The corrupt field cannot be represented as a double, so it becomes a null, which MAX silently ignores. The best approach is generally to declare types for your data on

Validation and Nulls – 4

Sometimes corrupt data shows up as smaller tuples because fields are simply missing. You can filter these out by using the SIZE function as follows:

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

Macros – 1

Macros provide a way to package reusable pieces of Pig Latin code from within Pig Latin itself. For example, we can extract the part of our Pig Latin program that performs grouping on a relation and then finds the maximum value in each group by defining a macro as follows:

```
DEFINE max_by_group(X, group_key, max_field) RETURNS Y {
  A = GROUP $X by $group_key;
  $Y = FOREACH A GENERATE group, MAX($X.$max_field);
};
```

The macro, called `max_by_group`, takes three parameters: a relation, `X`, and two field names, `group_key` and `max_field`. It returns a single relation, `Y`. Within the macro body, parameters and return aliases are referenced with a `$` prefix, such as `$X`.

Macros – 2

The macro is used as follows:

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
max_temp = max_by_group(filtered_records, year, temperature);
DUMP max_temp
```

At runtime, Pig will expand the macro using the macro definition. After expansion, the program looks like the following, with the expanded section in bold.

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
macro_max_by_group_A_0 = GROUP filtered_records by (year);
max_temp = FOREACH macro_max_by_group_A_0 GENERATE group,
  MAX(filtered_records.(temperature));
DUMP max_temp
```

A User-Defined Filter – 1

Let's demonstrate by writing 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 == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
```

to:

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

This achieves two things: it makes the Pig script more concise, and it encapsulates the logic in one place so that it can be easily reused in other scripts. If we were just writing an ad hoc query, we probably wouldn't bother to write a UDF. It's when you start doing the same kind of processing over and over again that you see opportunities for reusable UDFs.

A User-Defined Filter – 2

- P. 392: Example 11-1

A User-Defined Filter – 3

Then we tell Pig about the JAR file with the REGISTER operator, which is given the local path to the filename (and is *not* enclosed in quotes):

```
grunt> REGISTER pig-examples.jar;
```

Finally, we can invoke the function:

```
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>>   com.hadoopbook.pig.IsGoodQuality(quality);
```

We can add our package name to the search path by invoking Grunt with this command-line argument: `-Dudf.import.list=com.hadoopbook.pig`. Alternatively, we can shorten the function name by defining an alias, using the DEFINE operator:

```
grunt> DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
grunt> filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
```

Three User-Defined Functions

- P. 394: An Eval UDF
- P. 396: A Load UDF
- P. 400: Filtering Data

Throughout this chapter, we have seen how to load data from external storage for processing in Pig. Storing the results is straightforward, too. Here's 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
```

Other built-in storage functions were described in Table 11-7.

JOIN – 1

JOIN

Let's look at an 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)
```

JOIN – 2

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,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

large to fit in memory. If one of the relations is small enough to fit in memory, there is 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:[8]

```
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).

JOIN – 3

Pig also supports outer joins using a syntax that is similar to SQL's (this is covered for Hive in "Outer joins" on page 447). For example:

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

More Features

- P. 404: More Examples
- P. 405: Cross
- P. 406: Group
- P. 407: Sorting Data
- P. 408: Combining and Splitting data

Parallelism

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:

```
grouped_records = 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
```

A good setting for the number of reduce tasks is slightly fewer than the number of reduce slots in the cluster. See "Choosing the Number of Reducers" on page 231 for further discussion.

Parameter Substitution

daily may use the date to determine which input files it runs over. Pig supports *parameter substitution*, where parameters in the script are substituted with values supplied at runtime. Parameters are denoted by identifiers prefixed with a `$` character; for example, `$input` and `$output` are used in the following script to specify the input and output paths:

```
-- max_temp_param.pig
records = LOAD '$input' AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
STORE max_temp into '$output';
```

Parameters can be specified when launching Pig, using the `-param` option, one for each parameter:

```
% pig -param input=/user/tom/input/ncdc/micro-tab/sample.txt \
>     -param output=/tmp/out \
>     ch11/src/main/pig/max_temp_param.pig
```