CHAPTER 16
Pig

Apache Pig raises the level of abstraction for processing large datasets. MapReduce allows you, as the programmer, to specify a map function followed by a reduce function, but working out how to fit your data processing into this pattern, which often requires multiple MapReduce stages, can be a challenge. With Pig, the data structures are much richer, typically being multivalued and nested, and the transformations you can apply to the data are much more powerful. They include joins, for example, which are not for the faint of heart in MapReduce.

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 and distributed execution on a Hadoop cluster.

A Pig Latin program is made up of a series of operations, or transformations, that are applied to the input data to produce output. Taken as a whole, the operations describe a data flow, which the Pig execution environment translates into an executable representation and then runs. Under the covers, Pig turns the transformations into a series of MapReduce jobs, but as a programmer you are mostly unaware of this, which allows you to focus on the data rather than the nature of the execution.

Pig is a scripting language for exploring large datasets. One criticism of MapReduce is that the development cycle is very long. Writing the mappers and reducers, compiling and packaging the code, submitting the job(s), and retrieving the results is a time-consuming business, and even with Streaming, which removes the compile and package step, the experience is still involved. 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. Even more useful, it can perform a sample run on a representative subset of your input data, so you can see whether there are errors in the processing before unleashing it on the full dataset.

Pig was designed to be extensible. Virtually all parts of the processing path are customizable: loading, storing, filtering, grouping, and joining can all be altered by user-defined functions (UDFs). These functions operate on Pig's nested data model, so they can integrate very deeply with Pig's operators. As another benefit, UDFs tend to be more reusable than the libraries developed for writing MapReduce programs.

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.

Installing and Running Pig

Pig runs as a client-side application. Even if you want to run Pig on a Hadoop cluster, there is nothing extra to install on the cluster: Pig launches jobs and interacts with HDFS (or other Hadoop filesystems) from your workstation.

Installation is straightforward. Download a stable release from *http://pig.apache.org/releases.html*, and unpack the tarball in a suitable place on your workstation:

```
% tar xzf pig-x.y.z.tar.gz
```

It's convenient to add Pig's binary directory to your command-line path. For example:

```
% export PIG_HOME=~/sw/pig-x.y.z
% export PATH=$PATH:$PIG_HOME/bin
```

You also need to set the `JAVA_HOME` environment variable to point to a suitable Java installation.

Try typing `pig -help` to get usage instructions.

Execution Types

Pig has two execution types or modes: local mode and MapReduce mode. Execution modes for Apache Tez and Spark (see Chapter 19) were both under development at the time of writing. Both promise significant performance gains over MapReduce mode, so try them if they are available in the version of Pig you are using.

Local mode

In local mode, Pig runs in a single JVM and accesses the local filesystem. This mode is suitable only for small datasets and when trying out Pig.

The execution type is set using the `-x` or `-exectype` option. To run in local mode, set the option to `local`:

```
% pig -x local
grunt>
```

This starts Grunt, the Pig interactive shell, which is discussed in more detail shortly.

MapReduce mode

In MapReduce mode, Pig translates queries into MapReduce jobs and runs them on a Hadoop cluster. The cluster may be a pseudo- or fully distributed cluster. MapReduce mode (with a fully distributed cluster) is what you use when you want to run Pig on large datasets.

To use MapReduce mode, you first need to check that the version of Pig you downloaded is compatible with the version of Hadoop you are using. Pig releases will only work against particular versions of Hadoop; this is documented in the release notes.

Pig honors the `HADOOP_HOME` environment variable for finding which Hadoop client to run. However, if it is not set, Pig will use a bundled copy of the Hadoop libraries. Note that these may not match the version of Hadoop running on your cluster, so it is best to explicitly set `HADOOP_HOME`.

Next, you need to point Pig at the cluster's namenode and resource manager. If the installation of Hadoop at `HADOOP_HOME` is already configured for this, then there is nothing more to do. Otherwise, you can set `HADOOP_CONF_DIR` to a directory containing the Hadoop site file (or files) that define `fs.defaultFS`, `yarn.resourcemanager.address`, and `mapreduce.framework.name` (the latter should be set to `yarn`).

Alternatively, you can set these properties in the *pig.properties* file in Pig's *conf* directory (or the directory specified by `PIG_CONF_DIR`). Here's an example for a pseudo-distributed setup:

```
fs.defaultFS=hdfs://localhost/
mapreduce.framework.name=yarn
yarn.resourcemanager.address=localhost:8032
```

Once you have configured Pig to connect to a Hadoop cluster, you can launch Pig, setting the `-x` option to `mapreduce` or omitting it entirely, as MapReduce mode is the default. We've used the `-brief` option to stop timestamps from being logged:

```
% pig -brief
Logging error messages to: /Users/tom/pig_1414246949680.log
Default bootup file /Users/tom/.pigbootup not found
```

```
Connecting to hadoop file system at: hdfs://localhost/
grunt>
```

As you can see from the output, Pig reports the filesystem (but not the YARN resource manager) that it has connected to.

In MapReduce mode, you can optionally enable *auto-local mode* (by setting `pig.auto.local.enabled` to `true`), which is an optimization that runs small jobs locally if the input is less than 100 MB (set by `pig.auto.local.input.maxbytes`, default 100,000,000) and no more than one reducer is being used.

Running Pig Programs

There are three ways of executing Pig programs, all of which work in both local and MapReduce mode:

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`.

Grunt

Grunt has line-editing facilities like those found in GNU Readline (used in the bash shell and many other command-line applications). For instance, the Ctrl-E key combination will move the cursor to the end of the line. Grunt remembers command history, too,[1] and you can recall lines in the history buffer using Ctrl-P or Ctrl-N (for previous and next), or equivalently, the up or down cursor keys.

Another handy feature is Grunt's completion mechanism, which will try to complete Pig Latin keywords and functions when you press the Tab key. For example, consider the following incomplete line:

```
grunt> a = foreach b ge
```

1. History is stored in a file called *.pig_history* in your home directory.

If you press the Tab key at this point, `ge` will expand to `generate`, a Pig Latin keyword:

```
grunt> a = foreach b generate
```

You can customize the completion tokens by creating a file named *autocomplete* and placing it on Pig's classpath (such as in the *conf* directory in Pig's *install* directory) or in the directory you invoked Grunt from. The file should have one token per line, and tokens must not contain any whitespace. Matching is case sensitive. It can be very handy to add commonly used file paths (especially because Pig does not perform filename completion) or the names of any user-defined functions you have created.

You can get a list of commands using the `help` command. When you've finished your Grunt session, you can exit with the `quit` command, or the equivalent shortcut `\q`.

Pig Latin Editors

There are Pig Latin syntax highlighters available for a variety of editors, including Eclipse, IntelliJ IDEA, Vim, Emacs, and TextMate. Details are available on the Pig wiki.

Many Hadoop distributions come with the Hue web interface, which has a Pig script editor and launcher.

An Example

Let's look at a simple example by writing the program to calculate the maximum recorded temperature by year for the weather dataset in Pig Latin (just like we did using MapReduce in Chapter 2). The complete program is only a few lines long:

```
-- 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 IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

To explore what's going on, we'll use Pig's Grunt interpreter, which allows us to enter lines and interact with the program to understand what it's doing. Start up Grunt in local mode, and then enter the first line of the Pig script:

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

For simplicity, the program assumes that the input is tab-delimited text, with each line having just year, temperature, and quality fields. (Pig actually has more flexibility than this with regard to the input formats it accepts, as we'll see later.) This line describes the input data we want to process. The `year:chararray` notation describes the field's name

and type; `chararray` is like a Java `String`, and an `int` is like a Java `int`. The `LOAD` operator takes a URI argument; here we are just using a local file, but we could refer to an HDFS URI. The `AS` clause (which is optional) gives the fields names to make it convenient to refer to them in subsequent statements.

The result of the `LOAD` operator, and indeed any operator in Pig Latin, is a *relation*, which is just a set of tuples. A *tuple* is just like a row of data in a database table, with multiple fields in a particular order. In this example, the `LOAD` function produces a set of (year, temperature, quality) tuples that are present in the input file. We write a relation with one tuple per line, where tuples are represented as comma-separated items in parentheses:

```
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
```

Relations are given names, or *aliases*, so they can be referred to. This relation is given the `records` alias. We can examine the contents of an alias using the `DUMP` operator:

```
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

We can also see the structure of a relation—the relation's *schema*—using the `DESCRIBE` operator on the relation's alias:

```
grunt> DESCRIBE records;
records: {year: chararray,temperature: int,quality: int}
```

This tells us that `records` has three fields, with aliases `year`, `temperature`, and `quality`, which are the names we gave them in the `AS` clause. The fields have the types given to them in the `AS` clause, too. We examine types in Pig in more detail later.

The second statement removes records that have a missing temperature (indicated by a value of 9999) or an unsatisfactory quality reading. For this small dataset, no records are filtered out:

```
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>>   quality IN (0, 1, 4, 5, 9);
grunt> DUMP filtered_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

The third statement uses the GROUP function to group the records relation by the year field. Let's use DUMP to see what it produces:

```
grunt> grouped_records = GROUP filtered_records BY year;
grunt> DUMP grouped_records;
(1949,{(1949,78,1),(1949,111,1)})
(1950,{(1950,-11,1),(1950,22,1),(1950,0,1)})
```

We now have two rows, or tuples: one for each year in the input data. The first field in each tuple is the field being grouped by (the year), and the second field has a bag of tuples for that year. A *bag* is just an unordered collection of tuples, which in Pig Latin is represented using curly braces.

By grouping the data in this way, we have created a row per year, so now all that remains is to find the maximum temperature for the tuples in each bag. Before we do this, let's understand the structure of the grouped_records relation:

```
grunt> DESCRIBE grouped_records;
grouped_records: {group: chararray,filtered_records: {year: chararray,
temperature: int,quality: int}}
```

This tells us that the grouping field is given the alias group by Pig, and the second field is the same structure as the filtered_records relation that was being grouped. With this information, we can try the fourth transformation:

```
grunt> max_temp = FOREACH grouped_records GENERATE group,
>>   MAX(filtered_records.temperature);
```

FOREACH processes every row to generate a derived set of rows, using a GENERATE clause to define the fields in each derived row. In this example, the first field is group, which is just the year. The second field is a little more complex. The filtered_records.temperature reference is to the temperature field of the filtered_records bag in the grouped_records relation. MAX is a built-in function for calculating the maximum value of fields in a bag. In this case, it calculates the maximum temperature for the fields in each filtered_records bag. Let's check the result:

```
grunt> DUMP max_temp;
(1949,111)
(1950,22)
```

We've successfully calculated the maximum temperature for each year.

Generating Examples

In this example, we've used a small sample dataset with just a handful of rows to make it easier to follow the data flow and aid debugging. Creating a cut-down dataset is an art, as ideally it should be rich enough to cover all the cases to exercise your queries (the *completeness* property), yet small enough to make sense to the programmer (the *conciseness* property). Using a random sample doesn't work well in general because join

and filter operations tend to remove all random data, leaving an empty result, which is not illustrative of the general data flow.

With the `ILLUSTRATE` operator, Pig provides a tool for generating a reasonably complete and concise sample dataset. Here is the output from running `ILLUSTRATE` on our dataset (slightly reformatted to fit the page):

```
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       |
-------------------------------------------------
```

Notice that Pig used some of the original data (this is important to keep the generated dataset realistic), as well as creating some new data. It noticed the special value 9999 in the query and created a tuple containing this value to exercise the `FILTER` statement.

In summary, the output of `ILLUSTRATE` is easy to follow and can help you understand what your query is doing.

Comparison with Databases

Having seen Pig in action, it might seem that Pig Latin is similar to SQL. The presence of such operators as `GROUP BY` and `DESCRIBE` reinforces this impression. However, there are several differences between the two languages, and between Pig and relational database management systems (RDBMSs) in general.

The most significant difference is that Pig Latin is a data flow programming language, whereas SQL is a declarative programming language. In other words, a Pig Latin pro-

gram is a step-by-step set of operations on an input relation, in which each step is a single transformation. By contrast, SQL statements are a set of constraints that, taken together, define the output. In many ways, programming in Pig Latin is like working at the level of an RDBMS query planner, which figures out how to turn a declarative statement into a system of steps.

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, but it's optional. Essentially, it will operate on any source of tuples (although the source should support being read in parallel, by being in multiple files, for example), where a UDF is used to read the tuples from their raw representation.[2] The most common representation is a text file with tab-separated fields, and Pig provides a built-in load function for this format. Unlike with a traditional database, there is no data import process to load the data into the RDBMS. The data is loaded from the filesystem (usually HDFS) as the first step in the processing.

Pig's support for complex, nested data structures further differentiates it from SQL, which operates on flatter data structures. Also, Pig's ability to use UDFs and streaming operators that are tightly integrated with the language and Pig's nested data structures makes Pig Latin more customizable than most SQL dialects.

RDBMSs have several features to support online, low-latency queries, such as transactions and indexes, that are absent in Pig. Pig does not support random reads or queries on the order of tens of milliseconds. Nor does it support random writes to update small portions of data; all writes are bulk streaming writes, just like with MapReduce.

Hive (covered in Chapter 17) sits between Pig and conventional RDBMSs. Like Pig, Hive is designed to use HDFS for storage, but otherwise there are some significant differences. Its query language, HiveQL, is based on SQL, and anyone who is familiar with SQL will have little trouble writing queries in HiveQL. Like RDBMSs, Hive mandates that all data be stored in tables, with a schema under its management; however, it can associate a schema with preexisting data in HDFS, so the load step is optional. Pig is able to work with Hive tables using HCatalog; this is discussed further in "Using Hive tables with HCatalog" on page 442.

2. Or as the Pig Philosophy has it, "Pigs eat anything."

Pig Latin

This section gives an informal description of the syntax and semantics of the Pig Latin programming language.[3] It is not meant to offer a complete reference to the language,[4] but there should be enough here for you to get a good understanding of Pig Latin's constructs.

Structure

A Pig Latin program consists of a collection of statements. A statement can be thought of as an operation or a command.[5] For example, a GROUP operation is a type of statement:

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

The command to list the files in a Hadoop filesystem is another example of a statement:

```
ls /
```

Statements are usually terminated with a semicolon, as in the example of the GROUP statement. In fact, this is an example of a statement that must be terminated with a semicolon; it is a syntax error to omit it. The `ls` command, on the other hand, does not have to be terminated with a semicolon. As a general guideline, statements or commands for interactive use in Grunt do not need the terminating semicolon. This group includes the interactive Hadoop commands, as well as the diagnostic operators such as DESCRIBE. It's never an error to add a terminating semicolon, so if in doubt, it's simplest to add one.

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 used for 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?
```

3. Not to be confused with Pig Latin, the language game. English words are translated into Pig Latin by moving the initial consonant sound to the end of the word and adding an "ay" sound. For example, "pig" becomes "ig-pay," and "Hadoop" becomes "Adoop-hay."

4. Pig Latin does not have a formal language definition as such, but there is a comprehensive guide to the language that you can find through a link on the Pig website.

5. You sometimes see these terms being used interchangeably in documentation on Pig Latin: for example, "GROUP command," "GROUP operation," "GROUP statement."

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 has a list of keywords that have a special meaning in the language and cannot be used as identifiers. These include the operators (`LOAD`, `ILLUSTRATE`), commands (`cat`, `ls`), expressions (`matches`, `FLATTEN`), and functions (`DIFF`, `MAX`)—all of which are covered in the following sections.

Pig Latin has mixed rules on case sensitivity. Operators and commands are not case sensitive (to make interactive use more forgiving); however, aliases and function names are case sensitive.

Statements

As a Pig Latin program is executed, each statement is parsed in turn. If there are syntax errors or other (semantic) problems, such as undefined aliases, the interpreter will halt and display an error message. The interpreter builds a *logical plan* for every relational operation, which forms the core of a Pig Latin program. The logical plan for the statement is added to the logical plan for the program so far, and then the interpreter moves on to the next statement.

It's important to note that no data processing takes place while the logical plan of the program is being constructed. For example, consider again the Pig Latin program from the first example:

```
-- 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 IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

When the Pig Latin interpreter sees the first line containing the `LOAD` statement, it confirms that it is syntactically and semantically correct and adds it to the logical plan, but it does *not* load the data from the file (or even check whether the file exists). Indeed, where would it load it? Into memory? Even if it did fit into memory, what would it do

with the data? Perhaps not all the input data is needed (because later statements filter it, for example), so it would be pointless to load it. The point is that it makes no sense to start any processing until the whole flow is defined. Similarly, Pig validates the `GROUP` and `FOREACH...GENERATE` statements, and adds them to the logical plan without executing them. The trigger for Pig to start execution is the `DUMP` statement. At that point, the logical plan is compiled into a physical plan and executed.

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 (this includes the `run` command), but in batch mode it will not (this includes the `exec` command). 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. Consider the following simple example:

```
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';
```

Relations `B` and `C` are both derived from `A`, so to save reading `A` twice, Pig can run this script as a single MapReduce job by reading `A` once and writing two output files from the job, one for each of `B` and `C`. This feature is called *multiquery execution*.

In previous versions of Pig that did not have multiquery execution, each `STORE` statement in a script run in batch mode triggered execution, resulting in a job for each `STORE` statement. It is possible to restore the old behavior by disabling multiquery execution with the `-M` or `-no_multiquery` option to `pig`.

The physical plan that Pig prepares is a series of MapReduce jobs, which in local mode Pig runs in the local JVM and in MapReduce mode Pig runs on a Hadoop cluster.

You can see the logical and physical plans created by Pig using the `EXPLAIN` command on a relation (`EXPLAIN max_temp;`, for example).

`EXPLAIN` will also show the MapReduce plan, which shows how the physical operators are grouped into MapReduce jobs. This is a good way to find out how many MapReduce jobs Pig will run for your query.

The relational operators that can be a part of a logical plan in Pig are summarized in Table 16-1. We go through the operators in more detail in "Data Processing Operators" on page 456.

Table 16-1. Pig Latin relational operators

Category	Operator	Description
Loading and storing	LOAD	Loads data from the filesystem or other storage into a relation
	STORE	Saves a relation to the filesystem or other storage
	DUMP (\d)	Prints a relation to the console
Filtering	FILTER	Removes unwanted rows from a relation
	DISTINCT	Removes duplicate rows from a relation
	FOREACH...GENERATE	Adds or removes fields to or from a relation
	MAPREDUCE	Runs a MapReduce job using a relation as input
	STREAM	Transforms a relation using an external program
	SAMPLE	Selects a random sample of a relation
	ASSERT	Ensures a condition is true for all rows in a relation; otherwise, fails
Grouping and joining	JOIN	Joins two or more relations
	COGROUP	Groups the data in two or more relations
	GROUP	Groups the data in a single relation
	CROSS	Creates the cross product of two or more relations
	CUBE	Creates aggregations for all combinations of specified columns in a relation
Sorting	ORDER	Sorts a relation by one or more fields
	RANK	Assign a rank to each tuple in a relation, optionally sorting by fields first
	LIMIT	Limits the size of a relation to a maximum number of tuples
Combining and splitting	UNION	Combines two or more relations into one
	SPLIT	Splits a relation into two or more relations

There are other types of statements that are not added to the logical plan. For example, the diagnostic operators—DESCRIBE, EXPLAIN, and ILLUSTRATE—are provided to allow the user to interact with the logical plan for debugging purposes (see Table 16-2). DUMP is a sort of diagnostic operator, too, since it is used only to allow interactive debugging of small result sets or in combination with LIMIT to retrieve a few rows from a larger relation. The STORE statement should be used when the size of the output is more than a few lines, as it writes to a file rather than to the console.

Table 16-2. Pig Latin diagnostic operators

Operator (Shortcut)	Description
`DESCRIBE` (`\de`)	Prints a relation's schema
`EXPLAIN` (`\e`)	Prints the logical and physical plans
`ILLUSTRATE` (`\i`)	Shows a sample execution of the logical plan, using a generated subset of the input

Pig Latin also provides three statements—`REGISTER`, `DEFINE`, and `IMPORT`—that make it possible to incorporate macros and user-defined functions into Pig scripts (see Table 16-3).

Table 16-3. Pig Latin macro and UDF statements

Statement	Description
`REGISTER`	Registers a JAR file with the Pig runtime
`DEFINE`	Creates an alias for a macro, UDF, streaming script, or command specification
`IMPORT`	Imports macros defined in a separate file into a script

Because they do not process relations, commands are not added to the logical plan; instead, they are executed immediately. Pig provides commands to interact with Hadoop filesystems (which are very handy for moving data around before or after processing with Pig) and MapReduce, as well as a few utility commands (described in Table 16-4).

Table 16-4. 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

Category	Command	Description
Utility	`clear`	Clears the screen in Grunt
	`exec`	Runs a script in a new Grunt shell in batch mode
	`help`	Shows the available commands and options
	`history`	Prints the query statements run in the current Grunt session
	`quit` (`\q`)	Exits the interpreter
	`run`	Runs a script within the existing Grunt shell
	`set`	Sets Pig options and MapReduce job properties
	`sh`	Runs a shell command from within Grunt

The filesystem commands can operate on files or directories in any Hadoop filesystem, and they are very similar to the `hadoop fs` commands (which is not surprising, as both are simple wrappers around the Hadoop `FileSystem` interface). You can access all of the Hadoop filesystem shell commands using Pig's `fs` command. For example, `fs -ls` will show a file listing, and `fs -help` will show help on all the available commands.

Precisely which Hadoop filesystem is used is determined by the `fs.defaultFS` property in the site file for Hadoop Core. See "The Command-Line Interface" on page 50 for more details on how to configure this property.

These commands are mostly self-explanatory, except `set`, which is used to set options that control Pig's behavior (including arbitrary MapReduce job properties). The `debug` option is used to turn debug logging on or off from within a script (you can also control the log level when launching Pig, using the `-d` or `-debug` option):

```
grunt> set debug on
```

Another useful option is the `job.name` option, which gives a Pig job a meaningful name, making it easier to pick out your Pig MapReduce jobs when running on a shared Hadoop cluster. If Pig is running a script (rather than operating as an interactive query from Grunt), its job name defaults to a value based on the script name.

There are two commands in Table 16-4 for running a Pig script, `exec` and `run`. The difference is that `exec` runs the script in batch mode in a new Grunt shell, so any aliases defined in the script are not accessible to the shell after the script has completed. On the other hand, when running a script with `run`, it is as if the contents of the script had been entered manually, so the command history of the invoking shell contains all the statements from the script. Multiquery execution, where Pig executes a batch of statements in one go (see "Multiquery Execution" on page 434), is used only by `exec`, not `run`.

Control Flow

By design, Pig Latin lacks native control flow statements. The recommended approach for writing programs that have conditional logic or loop constructs is to embed Pig Latin in another language, such as Python, JavaScript, or Java, and manage the control flow from there. In this model, the host script uses a compile-bind-run API to execute Pig scripts and retrieve their status. Consult the Pig documentation for details of the API.

Embedded Pig programs always run in a JVM, so for Python and JavaScript you use the `pig` command followed by the name of your script, and the appropriate Java scripting engine will be selected (Jython for Python, Rhino for JavaScript).

Expressions

An expression is something that is evaluated to yield a value. Expressions can be used in Pig as a part of a statement containing a relational operator. Pig has a rich variety of expressions, many of which will be familiar from other programming languages. They are listed in Table 16-5, with brief descriptions and examples. We will see examples of many of these expressions throughout the chapter.

Table 16-5. Pig Latin expressions

Category	Expressions	Description	Examples
Constant	Literal	Constant value (see also the "Literal example" column in Table 16-6)	`1.0`, `'a'`
Field (by position)	`$`*n*	Field in position *n* (zero-based)	`$0`
Field (by name)	*f*	Field named *f*	`year`
Field (disambiguate)	*r*`::`*f*	Field named *f* from relation *r* after grouping or joining	`A::year`
Projection	*c*`.$`*n*, *c*`.`*f*	Field in container *c* (relation, bag, or tuple) by position, by name	`records.$0`, `records.year`
Map lookup	*m*`#`*k*	Value associated with key *k* in map *m*	`items#'Coat'`
Cast	`(`*t*`)` *f*	Cast of field *f* to type *t*	`(int) year`
Arithmetic	*x* `+` *y*, *x* `-` *y*	Addition, subtraction	`$1 + $2`, `$1 - $2`
	x `*` *y*, *x* `/` *y*	Multiplication, division	`$1 * $2`, `$1 / $2`
	x `%` *y*	Modulo, the remainder of *x* divided by *y*	`$1 % $2`
	`+`*x*, `-`*x*	Unary positive, negation	`+1`, `-1`
Conditional	*x* `?` *y* `:` *z*	Bincond/ternary; *y* if *x* evaluates to true, *z* otherwise	`quality == 0 ? 0 : 1`
	`CASE`	Multi-case conditional	`CASE q WHEN 0 THEN 'good' ELSE 'bad' END`

Category	Expressions	Description	Examples
Comparison	`x == y`, `x != y`	Equals, does not equal	`quality == 0`, `temperature != 9999`
	`x > y`, `x < y`	Greater than, less than	`quality > 0`, `quality < 10`
	`x >= y`, `x <= y`	Greater than or equal to, less than or equal to	`quality >= 1`, `quality <= 9`
	`x matches y`	Pattern matching with regular expression	`quality matches '[01459]'`
	`x is null`	Is null	`temperature is null`
	`x is not null`	Is not null	`temperature is not null`
Boolean	`x OR y`	Logical OR	`q == 0 OR q == 1`
	`x AND y`	Logical AND	`q == 0 AND r == 0`
	`NOT x`	Logical negation	`NOT q matches '[01459]'`
	`IN x`	Set membership	`q IN (0, 1, 4, 5, 9)`
Functional	*fn*`(`*f1*`,`*f2*`,...)`	Invocation of function *fn* on fields *f1*, *f2*, etc.	`isGood(quality)`
Flatten	`FLATTEN(`*f*`)`	Removal of a level of nesting from bags and tuples	`FLATTEN(group)`

Types

So far you have seen some of the simple types in Pig, such as `int` and `chararray`. Here we will discuss Pig's built-in types in more detail.

Pig has a `boolean` type and six numeric types: `int`, `long`, `float`, `double`, `biginteger`, and `bigdecimal`, which are identical to their Java counterparts. There is also a `bytearray` type, like Java's `byte` array type for representing a blob of binary data, and `chararray`, which, like `java.lang.String`, represents textual data in UTF-16 format (although it can be loaded or stored in UTF-8 format). The `datetime` type is for storing a date and time with millisecond precision and including a time zone.

Pig does not have types corresponding to Java's `byte`, `short`, or `char` primitive types. These are all easily represented using Pig's `int` type, or `chararray` for `char`.

The Boolean, numeric, textual, binary, and temporal types are simple atomic types. Pig Latin also has three complex types for representing nested structures: `tuple`, `bag`, and `map`. All of Pig Latin's types are listed in Table 16-6.

Table 16-6. Pig Latin types

Category	Type	Description	Literal example
Boolean	`boolean`	True/false value	`true`
Numeric	`int`	32-bit signed integer	`1`
	`long`	64-bit signed integer	`1L`
	`float`	32-bit floating-point number	`1.0F`
	`double`	64-bit floating-point number	`1.0`
	`biginteger`	Arbitrary-precision integer	`'10000000000'`
	`bigdecimal`	Arbitrary-precision signed decimal number	`'0.1100010000000000000000001'`
Text	`chararray`	Character array in UTF-16 format	`'a'`
Binary	`bytearray`	Byte array	Not supported
Temporal	`datetime`	Date and time with time zone	Not supported, use `ToDate` built-in function
Complex	`tuple`	Sequence of fields of any type	`(1,'pomegranate')`
	`bag`	Unordered collection of tuples, possibly with duplicates	`{(1,'pomegranate'),(2)}`
	`map`	Set of key-value pairs; keys must be character arrays, but values may be any type	`['a'#'pomegranate']`

The complex types are usually loaded from files or constructed using relational operators. Be aware, however, that the literal form in Table 16-6 is used when a constant value is created from within a Pig Latin program. The raw form in a file is usually different when using the standard `PigStorage` loader. For example, the representation in a file of the bag in Table 16-6 would be `{(1,pomegranate),(2)}` (note the lack of quotation marks), and with a suitable schema, this would be loaded as a relation with a single field and row, whose value was the bag.

Pig provides the built-in functions `TOTUPLE`, `TOBAG`, and `TOMAP`, which are used for turning expressions into tuples, bags, and maps.

Although relations and bags are conceptually the same (unordered collections of tuples), in practice Pig treats them slightly differently. A relation is a top-level construct, whereas a bag has to be contained in a relation. Normally you don't have to worry about this, but there are a few restrictions that can trip up the uninitiated. For example, it's not possible to create a relation from a bag literal. So, the following statement fails:

```
A = {(1,2),(3,4)}; -- Error
```

The simplest workaround in this case is to load the data from a file using the `LOAD` statement.

As another example, you can't treat a relation like a bag and project a field into a new relation (`$0` refers to the first field of `A`, using the positional notation):

```
B = A.$0;
```

Instead, you have to use a relational operator to turn the relation `A` into relation `B`:

```
B = FOREACH A GENERATE $0;
```

It's possible that a future version of Pig Latin will remove these inconsistencies and treat relations and bags in the same way.

Schemas

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}
```

This time we've declared the year to be an integer rather than a `chararray`, even though the file it is being loaded from is the same. An integer may be more appropriate if we need to manipulate the year arithmetically (to turn it into a timestamp, for example), whereas the `chararray` representation might be more appropriate when it's being used as a simple identifier. Pig's flexibility in the degree to which schemas are declared contrasts with schemas in traditional SQL databases, which are declared before the data is loaded into the system. Pig is designed for analyzing plain input files with no associated type information, so it is quite natural to choose types for fields later than you would with an RDBMS.

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}
```

In this case, we have specified only the names of the fields in the schema: `year`, `temperature`, and `quality`. The types default to `bytearray`, the most general type, representing a binary string.

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}
```

However, if you specify a schema in this way, you do need to specify every field. Also, there's no way to specify the type of a field without specifying the name. On the other 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}
```

Although it can be convenient not to assign types to fields (particularly in the first stages of writing a query), doing so can improve the clarity and efficiency of Pig Latin programs and is generally recommended.

Using Hive tables with HCatalog

Declaring a schema as a part of the query is flexible but doesn't lend itself to schema reuse. A set of Pig queries over the same input data will often have the same schema repeated in each query. If the query processes a large number of fields, this repetition can become hard to maintain.

HCatalog (which is a component of Hive) solves this problem by providing access to Hive's metastore, so that Pig queries can reference schemas by name, rather than specifying them in full each time. For example, after running through "An Example" on page 474 to load data into a Hive table called `records`, Pig can access the table's schema and data as follows:

```
% pig -useHCatalog
grunt> records = LOAD 'records' USING org.apache.hcatalog.pig.HCatLoader();
grunt> DESCRIBE records;
records: {year: chararray,temperature: int,quality: int}
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

Validation and nulls

A SQL database will enforce the constraints in a table's schema at load time; for example, trying to load a string into a column that is declared to be a numeric type will fail. In

Pig, if the value cannot be cast to the type declared in the schema, it will substitute a `null` value. Let's see how this works when we have the following input for the weather data, which has an "e" character in place of an integer:

```
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)
```

Pig produces a warning for the invalid field (not shown here) but does not halt its processing. For large datasets, it is very common to have corrupt, invalid, or merely 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 (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)
```

Note the use of the `is null` operator, which is analogous to SQL. In practice, we would include more information from the original record, such as an identifier and the value that could not be parsed, to help our analysis of the bad data.

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 464 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 OTHERWISE;
grunt> DUMP good_records;
(1950,0,1)
(1950,22,1)
(1949,111,1)
(1949,78,1)
grunt> DUMP bad_records;
(1950,,1)
```

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 IN (0, 1, 4, 5, 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 loading and look for missing or corrupt values in the relations themselves before you do your main processing.

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)
```

Schema merging

In Pig, you don't declare the schema for every new relation in the data flow. In most cases, Pig can figure out the resulting schema for the output of a relational operation by considering the schema of the input relation.

How are schemas propagated to new relations? Some relational operators don't change the schema, so the relation produced by the `LIMIT` operator (which restricts a relation to a maximum number of tuples), for example, has the same schema as the relation it operates on. For other operators, the situation is more complicated. `UNION`, for example, combines two or more relations into one and tries to merge the input relations' schemas. If the schemas are incompatible, due to different types or number of fields, then the schema of the result of the `UNION` is unknown.

You can find out the schema for any relation in the data flow using the `DESCRIBE` operator. If you want to redefine the schema for a relation, you can use the `FOREACH...GENERATE` operator with `AS` clauses to define the schema for some or all of the fields of the input relation.

See "User-Defined Functions" on page 448 for a further discussion of schemas.

Functions

Functions in Pig come in four types:

Eval function
: A function that takes one or more expressions and returns another expression. An example of a built-in eval function is `MAX`, which returns the maximum value of the entries in a bag. Some eval functions are *aggregate functions*, which means they operate on a bag of data to produce a scalar value; `MAX` is an example of an aggregate function. Furthermore, many aggregate functions are *algebraic*, which means that the result of the function may be calculated incrementally. In MapReduce terms, algebraic functions make use of the combiner and are much more efficient to calculate (see "Combiner Functions" on page 34). `MAX` is an algebraic function, whereas a function to calculate the median of a collection of values is an example of a function that is not algebraic.

Filter function
: A special type of eval function that returns a logical Boolean result. As the name suggests, filter functions are used in the `FILTER` operator to remove unwanted rows. They can also be used in other relational operators that take Boolean conditions, and in general, in expressions using Boolean or conditional expressions. An example of a built-in filter function is `IsEmpty`, which tests whether a bag or a map contains any items.

Load function
: A function that specifies how to load data into a relation from external storage.

Store function
: A function that specifies how to save the contents of a relation to external storage. Often, load and store functions are implemented by the same type. For example, `PigStorage`, which loads data from delimited text files, can store data in the same format.

Pig comes with a collection of built-in functions, a selection of which are listed in Table 16-7. The complete list of built-in functions, which includes a large number of standard math, string, date/time, and collection functions, can be found in the documentation for each Pig release.

Table 16-7. A selection of Pig's built-in functions

Category	Function	Description
Eval	`AVG`	Calculates the average (mean) value of entries in a bag.
	`CONCAT`	Concatenates byte arrays or character arrays together.
	`COUNT`	Calculates the number of non-`null` entries in a bag.
	`COUNT_STAR`	Calculates the number of entries in a bag, including those that are `null`.
	`DIFF`	Calculates the set difference of two bags. If the two arguments are not bags, returns a bag containing both if they are equal; otherwise, returns an empty bag.
	`MAX`	Calculates the maximum value of entries in a bag.
	`MIN`	Calculates the minimum value of entries in a bag.
	`SIZE`	Calculates the size of a type. The size of numeric types is always 1; for character arrays, it is the number of characters; for byte arrays, the number of bytes; and for containers (tuple, bag, map), it is the number of entries.
	`SUM`	Calculates the sum of the values of entries in a bag.
	`TOBAG`	Converts one or more expressions to individual tuples, which are then put in a bag. A synonym for `()`.
	`TOKENIZE`	Tokenizes a character array into a bag of its constituent words.
	`TOMAP`	Converts an even number of expressions to a map of key-value pairs. A synonym for `[]`.
	`TOP`	Calculates the top *n* tuples in a bag.
	`TOTUPLE`	Converts one or more expressions to a tuple. A synonym for `{}`.
Filter	`IsEmpty`	Tests whether a bag or map is empty.
Load/Store	`PigStorage`	Loads or stores relations using a field-delimited text format. Each line is broken into fields using a configurable field delimiter (defaults to a tab character) to be stored in the tuple's fields. It is the default storage when none is specified.[a]
	`TextLoader`	Loads relations from a plain-text format. Each line corresponds to a tuple whose single field is the line of text.

Category	Function	Description
	`JsonLoader`, `JsonStorage`	Loads or stores relations from or to a (Pig-defined) JSON format. Each tuple is stored on one line.
	`AvroStorage`	Loads or stores relations from or to Avro datafiles.
	`ParquetLoader`, `ParquetStorer`	Loads or stores relations from or to Parquet files.
	`OrcStorage`	Loads or stores relations from or to Hive ORCFiles.
	`HBaseStorage`	Loads or stores relations from or to HBase tables.

[a] The default storage can be changed by setting `pig.default.load.func` and `pig.default.store.func` to the fully qualified load and store function classnames.

Other libraries

If the function you need is not available, you can write your own user-defined function (or UDF for short), as explained in "User-Defined Functions" on page 448. Before you do that, however, have a look in the Piggy Bank, a library of Pig functions shared by the Pig community and distributed as a part of Pig. For example, there are load and store functions in the Piggy Bank for CSV files, Hive RCFiles, sequence files, and XML files. The Piggy Bank JAR file comes with Pig, and you can use it with no further configuration. Pig's API documentation includes a list of functions provided by the Piggy Bank.

Apache DataFu is another rich library of Pig UDFs. In addition to general utility functions, it includes functions for computing basic statistics, performing sampling and estimation, hashing, and working with web data (sessionization, link analysis).

Macros

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`.

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 IN (0, 1, 4, 5, 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 IN (0, 1, 4, 5, 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
```

Normally you don't see the expanded form, because Pig creates it internally; however, in some cases it is useful to see it when writing and debugging macros. You can get Pig to perform macro expansion only (without executing the script) by passing the `-dryrun` argument to `pig`.

Notice that the parameters that were passed to the macro (`filtered_records`, `year`, and `temperature`) have been substituted for the names in the macro definition. Aliases in the macro definition that don't have a `$` prefix, such as `A` in this example, are local to the macro definition and are rewritten at expansion time to avoid conflicts with aliases in other parts of the program. In this case, `A` becomes `macro_max_by_group_A_0` in the expanded form.

To foster reuse, macros can be defined in separate files to Pig scripts, in which case they need to be imported into any script that uses them. An import statement looks like this:

```
IMPORT './ch16-pig/src/main/pig/max_temp.macro';
```

User-Defined Functions

Pig's designers realized that the ability to plug in custom code is crucial for all but the most trivial data processing jobs. For this reason, they made it easy to define and use user-defined functions. We only cover Java UDFs in this section, but be aware that you can also write UDFs in Python, JavaScript, Ruby, or Groovy, all of which are run using the Java Scripting API.

A Filter UDF

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 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. 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.

Filter UDFs are all subclasses of `FilterFunc`, which itself is a subclass of `EvalFunc`. We'll look at `EvalFunc` in more detail later, but for the moment just note that, in essence, `EvalFunc` looks like the following class:

```
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. For `FilterFunc`, `T` is `Boolean`, so the method should return `true` only for those tuples that should not be filtered out.

For the quality filter, we write a class, `IsGoodQuality`, that extends `FilterFunc` and implements the `exec()` method (see Example 16-1). The `Tuple` class is essentially a list of objects with associated types. Here we are concerned only with the first field (since the function only has a single argument), which we extract by index using the `get()` method on `Tuple`. The field is an integer, so if it's not `null`, we cast it and check whether the value is one that signifies the temperature was a good reading, returning the appropriate value, `true` or `false`.

Example 16-1. A FilterFunc UDF to remove records with unsatisfactory temperature quality readings

```
package com.hadoopbook.pig;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.apache.pig.FilterFunc;

import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataType;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.logicalLayer.FrontendException;

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);
    }
  }

}
```

To use the new function, we first compile it and package it in a JAR file (the example code that accompanies this book comes with build instructions for how to do this). 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);
```

Pig resolves function calls by treating the function's name as a Java classname and attempting to load a class of that name. (This, incidentally, is why function names are case sensitive: because Java classnames are.) 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.

For the UDF in this example, Pig looks for a class with the name `com.hadoopbook.pig.IsGoodQuality`, which it finds in the JAR file we registered.

Resolution of built-in functions proceeds in the same way, except for one difference: Pig has a set of built-in package names that it searches, so the function call does not have to be a fully qualified name. For example, the function `MAX` is actually implemented by a class `MAX` in the package `org.apache.pig.builtin`. This is one of the packages that Pig looks in, so we can write `MAX` rather than `org.apache.pig.builtin.MAX` in our Pig programs.

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);
```

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.

If you add the lines to register JAR files and define function aliases to the *.pigbootup* file in your home directory, they will be run whenever you start Pig.

Leveraging types

The filter works when the quality field is declared to be of type `int`, but if the type information is absent, the UDF fails! This happens because the field is the default type, `bytearray`, represented by the `DataByteArray` class. Because `DataByteArray` is not an `Integer`, the cast fails.

The obvious way to fix this is to convert the field to an integer in the `exec()` method. However, there is a better way, which is to tell Pig the types of the fields that the function expects. The `getArgToFuncMapping()` method on `EvalFunc` is provided for precisely this reason. We can override it to tell Pig that the first field should be an integer:

```
@Override
public List<FuncSpec> getArgToFuncMapping() throws FrontendException {
  List<FuncSpec> funcSpecs = new ArrayList<FuncSpec>();
  funcSpecs.add(new FuncSpec(this.getClass().getName(),
      new Schema(new Schema.FieldSchema(null, DataType.INTEGER))));

  return funcSpecs;
}
```

This method returns a `FuncSpec` object corresponding to each of the fields of the tuple that are passed to the `exec()` method. Here there is a single field, and we construct an anonymous `FieldSchema` (the name is passed as `null`, since Pig ignores the name when doing type conversion). The type is specified using the `INTEGER` constant on Pig's `DataType` class.

With the amended function, Pig will attempt to convert the argument passed to the function to an integer. If the field cannot be converted, then a `null` is passed for the field. The `exec()` method always returns `false` when the field is `null`. For this application, this behavior is appropriate, as we want to filter out records whose quality field is unintelligible.

An Eval UDF

Writing an eval function is a small step up from writing a filter function. Consider the UDF in Example 16-2, which trims the leading and trailing whitespace from `chararray` values using the `trim()` method on `java.lang.String`.[6]

Example 16-2. 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();
  }
}
```

In this case, we have taken advantage of `PrimitiveEvalFunc`, which is a specialization of `EvalFunc` for when the input is a single primitive (atomic) type. For the `Trim` UDF, the input and output types are both of type `String`.[7]

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 `udf`:

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

If `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.

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

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

6. Pig actually comes with an equivalent built-in function called `TRIM`.

7. Although not relevant for this example, eval functions that operate on a bag may additionally implement Pig's `Algebraic` or `Accumulator` interfaces for more efficient processing of the bag in chunks.

```
(apple)
(lychee)
grunt> DESCRIBE B;
B: {chararray}
```

`A` has `chararray` fields that have leading and trailing spaces. We create `B` from `A` by applying the `Trim` function to the first field in `A` (named `fruit`). `B`'s fields are correctly inferred to be of type `chararray`.

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.

The following snippet shows how we could define and use a `trim` UDF that uses the Apache Commons Lang `StringUtils` class:

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

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.

A Load UDF

We'll demonstrate a custom load function that can read plain-text column ranges as fields, very much like the Unix `cut` command.[8] It is used as follows:

```
grunt> records = LOAD 'input/ncdc/micro/sample.txt'
>>   USING com.hadoopbook.pig.CutLoadFunc('16-19,88-92,93-93')
>>   AS (year:int, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
```

8. There is a more fully featured UDF for doing the same thing in the Piggy Bank called `FixedWidthLoader`.

```
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

The string passed to `CutLoadFunc` is the column specification; each comma-separated range defines a field, which is assigned a name and type in the `AS` clause. Let's examine the implementation of `CutLoadFunc`, shown in Example 16-3.

Example 16-3. A LoadFunc UDF to load tuple fields as column ranges

```
public class CutLoadFunc extends LoadFunc {

  private static final Log LOG = LogFactory.getLog(CutLoadFunc.class);

  private final List<Range> ranges;
  private final TupleFactory tupleFactory = TupleFactory.getInstance();
  private RecordReader reader;

  public CutLoadFunc(String cutPattern) {
    ranges = Range.parse(cutPattern);
  }

  @Override
  public void setLocation(String location, Job job)
      throws IOException {
    FileInputFormat.setInputPaths(job, location);
  }

  @Override
  public InputFormat getInputFormat() {
    return new TextInputFormat();
  }

  @Override
  public void prepareToRead(RecordReader reader, PigSplit split) {
    this.reader = reader;
  }

  @Override
  public Tuple getNext() throws IOException {
    try {
      if (!reader.nextKeyValue()) {
        return null;
      }
      Text value = (Text) reader.getCurrentValue();
      String line = value.toString();
      Tuple tuple = tupleFactory.newTuple(ranges.size());
      for (int i = 0; i < ranges.size(); i++) {
        Range range = ranges.get(i);
        if (range.getEnd() > line.length()) {
          LOG.warn(String.format(
              "Range end (%s) is longer than line length (%s)",
              range.getEnd(), line.length()));
```

```
          continue;
        }
        tuple.set(i, new DataByteArray(range.getSubstring(line)));
      }
      return tuple;
    } catch (InterruptedException e) {
      throw new ExecException(e);
    }
  }
}
```

In Pig, like in Hadoop, data loading takes place before the mapper runs, so it is important that the input can be split into portions that are handled independently by each mapper (see "Input Splits and Records" on page 220 for background). A `LoadFunc` will typically use an existing underlying Hadoop `InputFormat` to create records, with the `LoadFunc` providing the logic for turning the records into Pig tuples.

`CutLoadFunc` is constructed with a string that specifies the column ranges to use for each field. The logic for parsing this string and creating a list of internal `Range` objects that encapsulates these ranges is contained in the `Range` class, and is not shown here (it is available in the example code that accompanies this book).

Pig calls `setLocation()` on a `LoadFunc` to pass the input location to the loader. Since `CutLoadFunc` uses a `TextInputFormat` to break the input into lines, we just pass the location to set the input path using a static method on `FileInputFormat`.

Pig uses the new MapReduce API, so we use the input and output formats and associated classes from the `org.apache.hadoop.mapreduce` package.

Next, Pig calls the `getInputFormat()` method to create a `RecordReader` for each split, just like in MapReduce. Pig passes each `RecordReader` to the `prepareToRead()` method of `CutLoadFunc`, which we store a reference to, so we can use it in the `getNext()` method for iterating through the records.

The Pig runtime calls `getNext()` repeatedly, and the load function reads tuples from the reader until the reader reaches the last record in its split. At this point, it returns `null` to signal that there are no more tuples to be read.

It is the responsibility of the `getNext()` implementation to turn lines of the input file into `Tuple` objects. It does this by means of a `TupleFactory`, a Pig class for creating `Tuple` instances. The `newTuple()` method creates a new tuple with the required number of fields, which is just the number of `Range` classes, and the fields are populated using substrings of the line, which are determined by the `Range` objects.

We need to think about what to do when the line is shorter than the range asked for. One option is to throw an exception and stop further processing. This is appropriate if your application cannot tolerate incomplete or corrupt records. In many cases, it is better to return a tuple with `null` fields and let the Pig script handle the incomplete data as it sees fit. This is the approach we take here; by exiting the `for` loop if the range end is past the end of the line, we leave the current field and any subsequent fields in the tuple with their default values of `null`.

Using a schema

Let's now consider the types of the fields being loaded. If the user has specified a schema, then the fields need to be converted to the relevant types. However, this is performed lazily by Pig, so the loader should always construct tuples of type `bytearrary`, using the `DataByteArray` type. The load function still has the opportunity to do the conversion, however, by overriding `getLoadCaster()` to return a custom implementation of the `LoadCaster` interface, which provides a collection of conversion methods for this purpose.

`CutLoadFunc` doesn't override `getLoadCaster()` because the default implementation returns `Utf8StorageConverter`, which provides standard conversions between UTF-8–encoded data and Pig data types.

In some cases, the load function itself can determine the schema. For example, if we were loading self-describing data such as XML or JSON, we could create a schema for Pig by looking at the data. Alternatively, the load function may determine the schema in another way, such as from an external file, or by being passed information in its constructor. To support such cases, the load function should implement the `LoadMetadata` interface (in addition to the `LoadFunc` interface) so it can supply a schema to the Pig runtime. Note, however, that if a user supplies a schema in the `AS` clause of `LOAD`, then it takes precedence over the schema specified through the `LoadMetadata` interface.

A load function may additionally implement the `LoadPushDown` interface as a means for finding out which columns the query is asking for. This can be a useful optimization for column-oriented storage, so that the loader loads only the columns that are needed by the query. There is no obvious way for `CutLoadFunc` to load only a subset of columns, because it reads the whole line for each tuple, so we don't use this optimization.

Data Processing Operators

Loading and Storing 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 16-7.

Filtering Data

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

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 this example, we do both:

```
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)
```

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`.

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;
```

```
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;
```

Using the cut UDF we developed earlier, we load various fields from the input dataset into the `records` relation. Next, we group `records` by year. Notice the `PARALLEL` keyword for setting the number of reducers to use; this is vital when running on a cluster. Then we process each group using a nested `FOREACH...GENERATE` operator. The first nested statement creates a relation for the distinct USAF identifiers for stations using the `DISTINCT` operator. The second nested statement creates a relation for the records with "good" readings using the `FILTER` operator and a UDF. The final nested statement is a `GENERATE` statement (a nested `FOREACH...GENERATE` must always have a `GENERATE` statement as the last nested statement) that generates the summary fields of interest using the grouped records, as well as the relations created in the nested block.

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.

STREAM

The `STREAM` operator allows you to transform data in a relation using an external program or script. It is named by analogy with Hadoop Streaming, which provides a similar capability for MapReduce (see "Hadoop Streaming" on page 37).

`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)
```

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. You can provide a custom serializer and deserializer by subclassing PigStreamingBase (in the org.apache.pig package), then using the DEFINE operator.

Pig streaming is most powerful when you 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)
```

To use the script, you 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;
```

Grouping and Joining Data

Joining datasets in MapReduce takes some work on the part of the programmer (see "Joins" on page 268), 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.

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)
```

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.

You 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, you 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:[9]

```
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).

Pig also supports outer joins using a syntax that is similar to SQL's (this is covered for Hive in "Outer joins" on page 506). 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)
```

9. There are more keywords that may be used in the `USING` clause, including `'skewed'` (for large datasets with a skewed keyspace), `'merge'` (to effect a merge join for inputs that are already sorted on the join key), and `'merge-sparse'` (where 1% or less of data is matched). See Pig's documentation for details on how to use these specialized joins.