10 Programming with Pig

This chapter covers

- Installing Pig and using the Grunt shell
- Understanding the Pig Latin language
- Extending the Pig Latin language with user-defined functions
- Computing similar documents efficiently, using a simple Pig Latin script

One frequent complaint about MapReduce is that it's difficult to program. When you first think through a data processing task, you may think about it in terms of data flow operations, such as loops and filters. However, as you implement the program in MapReduce, you'll have to think at the level of mapper and reducer functions and job chaining. Certain functions that are treated as first-class operations in higher-level languages become nontrivial to implement in MapReduce, as we've seen for joins in chapter 5. Pig is a Hadoop extension that simplifies Hadoop programming by giving you a high-level data processing language while keeping Hadoop's simple scalability and reliability. Yahoo, one of the heaviest user of Hadoop (and a backer of both the Hadoop Core and Pig), runs 40 percent of all its Hadoop jobs with Pig. Twitter is also another well-known user of Pig.[1]

[1] http://www.slideshare.net/kevinweil/hadoop-pig-and-twitter-nosql-east-2009.

Pig has two major components:

1. A high-level data processing language called Pig Latin.
2. A compiler that compiles and runs your Pig Latin script in a choice of *evaluation mechanisms*. The main evaluation mechanism is Hadoop. Pig also supports a local mode for development purposes.

Pig simplifies programming because of the ease of expressing your code in Pig Latin. The compiler helps to automatically exploit optimization opportunities in your script. This frees you from having to tune your program manually. As the Pig compiler improves, your Pig Latin program will also get an automatic speed-up.

10.1 Thinking like a Pig

Pig has a certain philosophy about its design. We expect ease of use, high performance, and massive scalability from any Hadoop subproject. More unique and crucial to understanding Pig are the design choices of its programming language (a data flow language called Pig Latin), the data types it supports, and its treatment of user-defined functions (UDFs) as first-class citizens.

10.1.1 Data flow language

You write Pig Latin programs in a sequence of steps where each step is a single high-level data transformation. The transformations support relational-style operations, such as filter, union, group, and join. An example Pig Latin program that processes a search query log may look like

```
log  = LOAD 'excite-small.log' AS (user, time, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd GENERATE group, COUNT(log);
DUMP cntd;
```

Even though the operations are relational in style, Pig Latin remains a data flow language. A data flow language is friendlier to programmers who think in terms of algorithms, which are more naturally expressed by the data and control flows. On the other hand, a declarative language such as SQL is sometimes easier for analysts who prefer to just state the results one expects from a program. Hive is a different Hadoop subproject that targets users who prefer the SQL model. We'll learn about Hive in detail in chapter 11.

10.1.2 Data types

We can summarize Pig's philosophy toward data types in its slogan of "Pigs eat anything." Input data can come in any format. Popular formats, such as tab-delimited text files, are natively supported. Users can add functions to support other data file formats as well. Pig doesn't require metadata or schema on data, but it can take advantage of them if they're provided.

Pig can operate on data that is relational, nested, semistructured, or unstructured. To support this diversity of data, Pig supports complex data types, such as bags and tuples that can be nested to form fairly sophisticated data structures.

10.1.3 User-defined functions

Pig was designed with many applications in mind—processing log data, natural language processing, analyzing network graphs, and so forth. It's expected that many of the computations will require custom processing. Pig is architected from the ground up with support for user-defined functions. Knowing how to write UDFs is a big part of learning to use Pig.

10.2 Installing Pig

You can download the latest release of Pig from http://hadoop.apache.org/pig/releases.html. As of this writing, the latest versions of Pig are 0.4 and 0.5. Both of them require Java 1.6. The main difference between them is that Pig version 0.4 targets Hadoop version 0.18 whereas Pig version 0.5 targets Hadoop version 0.20. As usual, make sure to set `JAVA_HOME` to the root of your Java installation, and Windows users should install Cygwin. Your Hadoop cluster should already be set up. Ideally it's a real cluster in fully distributed mode, although a pseudo-distributed setup is fine for practice.

You install Pig on your local machine by unpacking the downloaded distribution. There's nothing you have to modify on your Hadoop cluster. Think of the Pig distribution as a compiler and some development and deployment tools. It enhances your MapReduce programming but is otherwise only loosely coupled with the production Hadoop cluster.

Under the directory where you unpacked Pig, you should create the subdirectories `logs` and `conf` (unless they're already there). Pig will take custom configuration from files in `conf`. If you are creating the `conf` directory just now, there's obviously no configuration file, and you'll need to put in `conf` a new file named pig-env.sh. This script is executed when you run Pig, and it can be used to set up environment variables for configuring Pig. Besides `JAVA_HOME`, the environment variables of particular interest are `PIG_HADOOP_VERSION` and `PIG_CLASSPATH`. You set these variables to instruct Pig about your Hadoop cluster. For example, the following statements in pig-env.sh will tell Pig the version of Hadoop used by the cluster is 0.18, and to add the configuration directory of your local installation of Hadoop to Pig's classpath:

```
export PIG_HADOOP_VERSION=18
export PIG_CLASSPATH=$HADOOP_HOME/conf/
```

We assume `HADOOP_HOME` is set to Hadoop's installation directory on your local machine. By adding Hadoop's `conf` directory to Pig's classpath, Pig can automatically pick up the location of your Hadoop cluster's NameNode and JobTracker.

Instead of using Pig's classpath, you can also specify the location of your Hadoop cluster by creating a pig.properties file. This properties file will be under the `conf` directory you created earlier. It should define `fs.default.name` and `mapred.job.tracker`, the filesystem (i.e., HDFS's NameNode) and the location of the JobTracker. An example pig.properties file pointing to a Hadoop set up in pseudo-distributed mode is

```
fs.default.name=hdfs://localhost:9000
mapred.job.tracker=localhost:9001
```

For the sake of convenience, let's add the Pig installation's `bin` directory to your path. Assume `$PIG_HOME` is pointing to your Pig's installation:

```
export PATH=$PATH:$PIG_HOME/bin
```

With Pig's `bin` directory set as part of your command line path, you can start Pig with the command `pig`. You may want to first see its usage options:

```
pig -help
```

Let's start Pig's interactive shell to see that it's reading the configurations properly.

```
pig
2009-07-11 22:33:04,797 [main] INFO
➥ org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -
➥ Connecting to hadoop file system at: hdfs://localhost:9000
2009-07-11 22:33:09,533 [main] INFO
➥ org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -
➥ Connecting to map-reduce job tracker at: localhost:9001
grunt>
```

The filesystem and the JobTracker Pig reports should be consistent with your configuration setup. You're now inside Pig's interactive shell, also known as Grunt.

10.3 Running Pig

We can run Pig Latin commands in three ways—via the Grunt interactive shell, through a script file, and as embedded queries inside Java programs. Each way can work in one of two modes—local mode and Hadoop mode. (Hadoop mode is sometimes called Mapreduce mode in the Pig documentation.) At the end of the previous section we've entered the Grunt shell running in Hadoop mode.

The Grunt shell allows you to enter Pig commands manually. This is typically used for ad hoc data analysis or during the interactive cycles of program development. Large Pig programs or ones that will be run repeatedly are run in script files. To enter Grunt, use the command `pig`. To run a Pig script, execute the same `pig` command with the file name as the argument, such as `pig myscript.pig`. The convention is to use the `.pig` extension for Pig scripts.

You can think of Pig programs as similar to SQL queries, and Pig provides a `PigServer` class that allows any Java program to execute Pig queries. Conceptually this is analogous to using JDBC to execute SQL queries. Embedded Pig programs is a fairly advanced topic and you can find more details at http://wiki.apache.org/pig/EmbeddedPig.

When you run Pig in local mode, you don't use Hadoop at all.[2] Pig commands are compiled to run locally in their own JVM, accessing local files. This is typically used for development purposes, where you can get fast feedback by running locally against

[2] There are plans to change Pig such that it uses Hadoop even in local mode, which helps to make some programming more consistent. The discussion for this topic is taking place at https://issues.apache.org/jira/browse/PIG-1053.

a small development data set. Running Pig in Hadoop mode means the compile Pig program will physically execute in a Hadoop installation. Typically the Hadoop installation is a fully distributed cluster. (The pseudo-distributed Hadoop setup we used in section 10.2 was purely for demonstration. It's rarely used except to debug configurations.) The execution mode is specified to the `pig` command via the `-x` or `-exectype` option. You can enter the Grunt shell in local mode through:

```
pig -x local
```

Entering the Grunt shell in Hadoop mode is

```
pig -x mapreduce
```

or use the `pig` command without arguments, as it chooses the Hadoop mode by default.

10.3.1 Managing the Grunt shell

In addition to running Pig Latin statements (which we'll look at in a later section), the Grunt shell supports some basic utility commands.[3] Typing `help` will print out a help screen of such utility commands. You exit the Grunt shell with `quit`. You can stop a Hadoop job with the `kill` command followed by the Hadoop job ID. Some Pig parameters are set with the `set` command. For example,

```
grunt> set debug on
grunt> set job.name 'my job'
```

The `debug` parameter states whether debug-level logging is turned on or off. The `job.name` parameter takes a single-quoted string and will use that as the Pig program's Hadoop job name. It's useful to set a meaningful name to easily identify your Pig job in Hadoop's Web UI.

The Grunt shell also supports file utility commands, such as `ls` and `cp`. You can see the full list of utility commands and file commands in table 10.1. The file commands are mostly a subset of the HDFS filesystem shell commands, and their usage should be self-explanatory.

Table 10.1 Utility and file commands in the Grunt shell

Utility commands	`help`
	`quit`
	`kill` ***`jobid`***
	`set debug [on|off]`
	`set job.name` ***`'jobname'`***

File commands	`cat, cd, copyFromLocal, copyToLocal, cp, ls, mkdir, mv, pwd, rm, rmf, exec, run`

[3] Technically these are still considered Pig Latin commands, but you'll not likely use them outside of the Grunt shell.

Two new commands are `exec` and `run`. They run Pig scripts while inside the Grunt shell and can be useful in debugging Pig scripts. The `exec` command executes a Pig script in a separate space from the Grunt shell. Aliases defined in the script aren't visible to the shell and vice versa. The command `run` executes a Pig script in the same space as Grunt (also known as *interactive mode*). It has the same effect as manually typing in each line of the script into the Grunt shell.

10.4 Learning Pig Latin through Grunt

Before formally describing Pig's data types and data processing operators, let's run a few commands in the Grunt shell to get a feel for how to process data in Pig. For the purpose of learning, it's more convenient to run Grunt in local mode:

```
pig -x local
```

You may want to first try some of the file commands, such as `pwd` and `ls`, to orient yourself around the filesystem.

Let's look at some data. We'll later reuse the patent data we introduced in chapter 4, but for now let's dig into an interesting data set of query logs from the Excite search engine. This data set already comes with the Pig installation, and it's in the file `tutorial/data/excite-small.log` under the Pig installation directory. The data comes in a three-column, tab-separated format. The first column is an anonymized user ID. The second column is a Unix timestamp, and the third is the search query. A decidedly non-random sample from the 4,500 records of this file looks like

```
3F8AAC2372F6941C    970916093724    minors in possession
C5460576B58BB1CC    970916194352    hacking telenet
9E1707EE57C96C1E    970916073214    buffalo mob crime family
06878125BE78B42C    970916183900    how to make ecstacy
```

From within Grunt, enter the following statement to load this data into an "alias" (i.e., variable) called `log`.

```
grunt> log  = LOAD 'tutorial/data/excite-small.log' AS (user, time, query);
```

Note that nothing seems to have happened after you entered the statement. In the Grunt shell, Pig parses your statements but doesn't physically execute them until you use a `DUMP` or `STORE` command to ask for the results. The `DUMP` command prints out the content of an alias whereas the `STORE` command stores the content to a file. The fact that Pig doesn't physically execute any command until you explicitly request some end result will make sense once you remember that we're processing large data sets. There's no memory space to "load" the data, and in any case we want to verify the logic of the execution plan before spending the time and resources to physically execute it.

We use the `DUMP` command usually only for development. Most often you'll `STORE` significant results into a directory. (Like Hadoop, Pig will automatically partition the data into files named `part-`*nnnnn*.) When you `DUMP` an alias, you should be sure that

its content is small enough to be reasonably printed to screen. The common way to do that is to create another alias through the `LIMIT` command and `DUMP` the new, smaller alias. The `LIMIT` command allows you to specify how many tuples (rows) to return back. For example, to see four tuples of `log`

```
grunt> lmt = LIMIT log 4;
grunt> DUMP lmt;
(2A9EABFB35F5B954,970916105432L,+md foods +proteins)
(BED75271605EBD0C,970916001949L,yahoo chat)
(BED75271605EBD0C,970916001954L,yahoo chat)
(BED75271605EBD0C,970916003523L,yahoo chat)
```

Table 10.2 summarizes the read and write operators in Pig Latin. `LIMIT` is technically not a read or write operator, but as it's often used alongside, we've included it in the table.

Table 10.2 Data read/write operators in Pig Latin

`LOAD`	`alias = LOAD 'file' [USING function] [AS schema];` Load data from a file into a relation. Uses the `PigStorage` load function as default unless specified otherwise with the `USING` option. The data can be given a schema using the `AS` option.
`LIMIT`	`alias = LIMIT alias n;` Limit the number of tuples to *n*. When used right after `alias` was processed by an `ORDER` operator, `LIMIT` returns the first *n* tuples. Otherwise there's no guarantee which tuples are returned. The `LIMIT` operator defies categorization because it's certainly not a read/write operator but it's not a true relational operator either. We include it here for the practical reason that a reader looking up the `DUMP` operator, explained later, will remember to use the `LIMIT` operator right before it.
`DUMP`	`DUMP alias;` Display the content of a relation. Use mainly for debugging. The relation should be small enough for printing on screen. You can apply the `LIMIT` operation on an alias to make sure it's small enough for display.
`STORE`	`STORE alias INTO 'directory' [USING function];` Store data from a relation into a directory. The directory must not exist when this command is executed. Pig will create the directory and store the relation in files named `part-`*nnnnn* in it. Uses the `PigStorage` store function as default unless specified otherwise with the `USING` option.

You may find loading and storing data not terribly exciting. Let's execute a few data processing statements and see how we can explore Pig Latin through Grunt.

```
grunt> log  = LOAD 'tutorial/data/excite-small.log'
           ➥ AS (user:chararray, time:long, query:chararray);
grunt> grpd = GROUP log BY user;
grunt> cntd = FOREACH grpd GENERATE group, COUNT(log);
grunt> STORE cntd INTO 'output';
```

The preceding statements count the number of queries each user has searched for. The content of the output files (you'll have to look at the file from outside Grunt) look like this:

```
002BB5A52580A8ED        18
005BD9CD3AC6BB38        18
00A08A54CD03EB95        3
011ACA65C2BF70B2        5
01500FAFE317B7C0        15
0158F8ACC570947D        3
018FBF6BFB213E68        1
```

Conceptually we've performed an aggregating operation similar to the SQL query:

```
SELECT user, COUNT(*) FROM excite-small.log GROUP BY user;
```

Two main differences between the Pig Latin and SQL versions are worth pointing out. As we've mentioned earlier, Pig Latin is a data processing language. You're specifying a series of data processing steps instead of a complex SQL query with clauses. The other difference is more subtle—relations in SQL always have fixed schemas. In SQL, we define a relation's schema before it's populated with data. Pig takes a much looser approach to schema. In fact, you don't need to use schemas if you don't want to, which may be the case when handling semistructured or unstructured data. Here we do specify a schema for the relation `log`, but it's only in the load statement and it's not enforced until we're loading in the data. Any field that doesn't obey the schema in the load operation is casted to a null. In this way the relation `log` is guaranteed to obey our stated schema for subsequent operations.

As much as possible, Pig tries to figure out the schema for a relation based on the operation used to create it. You can expose Pig's schema for any relation with the `DESCRIBE` command. This can be useful in understanding what a Pig statement is doing. For example, we'll look at the schemas for `grpd` and `cntd`. Before doing this, let's first see how the `DESCRIBE` command describes `log`.

```
grunt> DESCRIBE log;
log: {user: chararray,time: long,query: chararray}
```

As expected, the load command gives `log` the exact schema we've specified. The relation `log` consists of three fields named `user`, `time`, and `query`. The fields `user` and `query` are both strings (`chararray` in Pig) whereas `time` is a `long` integer.

`A GROUP BY` operation on the relation `log` generates the relation `grpd`. Based on the operation and the schema for `log`, Pig infers a schema for `grpd`:

```
grunt> DESCRIBE grpd;
grpd: {group: chararray,log: {user: chararray,time: long,query: chararray}}
```

`group` and `log` are two fields in `grpd`. The field `log` is a *bag* with subfields `user`, `time`, and `query`. As we haven't covered Pig's type system and the `GROUP BY` operation, we don't expect you to understand this schema yet. The point is that relations in Pig can have fairly complex schemas, and `DESCRIBE` is your friend in understanding the relations you're working with:

```
grunt> DESCRIBE cntd;
cntd: {group: chararray,long}
```

Finally, the FOREACH command operates on the relation grpd to give us cntd. Having looked at the output of cntd, we know it has two fields—the user ID and a count of the number of queries. Pig's schema for cntd, as given by DESCRIBE, also has two fields. The first one's name—group—is taken from grpd's schema. The second field has no name, but it has a type of long. This field is generated by the COUNT function, and the function doesn't automatically provide a name, although it does tell Pig that the type has to be a long.

Whereas DESCRIBE can tell you the schema of a relation, ILLUSTRATE does a sample run to show a step-by-step process on how Pig would compute the relation. Pig tries to simulate the execution of the statements to compute a relation, but it uses only a small sample of data to make the execution fast. The best way to understand ILLUSTRATE is by applying it to a relation. In this case we use cntd. (The output is reformatted to fit the width of a printed page.)

```
grunt> ILLUSTRATE cntd;
-------------------------------------------------------------------
| log     | user: bytearray  | time: bytearray | query: bytearray    |
-------------------------------------------------------------------
|         | 0567639EB8F3751C | 970916161410    | "conan o'brien"     |
|         | 0567639EB8F3751C | 970916161413    | "conan o'brien"     |
|         | 972F13CE9A8E2FA3 | 970916063540    | finger AND download |
-------------------------------------------------------------------
-----------------------------------------------------------------
| log     | user: chararray  | time: long    | query: chararray    |
-----------------------------------------------------------------
|         | 0567639EB8F3751C | 970916161410 | "conan o'brien"     |
|         | 0567639EB8F3751C | 970916161413 | "conan o'brien"     |
|         | 972F13CE9A8E2FA3 | 970916063540 | finger AND download |
-----------------------------------------------------------------
-------------------------------------------------------------------------
| grpd     | group: chararray | log: bag({user: chararray,time: long,     |
|          |                  |           query: chararray})              |
-------------------------------------------------------------------------
|          | 0567639EB8F3751C | {(0567639EB8F3751C, 970916161410,         |
|          |                  |   "conan o'brien"),                       |
|          |                  |  (0567639EB8F3751C,970916161413,          |
|          |                  |   "conan o'brien")}                       |
|          | 972F13CE9A8E2FA3 | {(972F13CE9A8E2FA3, 970916063540,         |
|          |                  |   finger AND download)}                   |
-------------------------------------------------------------------------
--------------------------------------
| cntd     | group: chararray | long  |
--------------------------------------
|          | 0567639EB8F3751C | 2     |
|          | 972F13CE9A8E2FA3 | 1     |
--------------------------------------
```

The ILLUSTRATE command shows there to be four transformations to arrive at cntd. The header row of each table describes the schema of the output relation after transformation, and the rest of the table shows example data. The log relation is shown as two transformations. The data hasn't changed from one to the next, but the schema has changed from a generic bytearray (Pig's type for binary objects) to the specified

schema. The GROUP operation on log is executed on the three sample log tuples to arrive at the data for grpd. Based on this we can infer the GROUP operation to have taken the user field and made it the group field. In addition, it groups all tuples in log with the same user value into a bag in grpd. Seeing sample data in a simulated run by ILLUSTRATE can greatly aid the understanding of different operations. Finally, we see the FOREACH operation applied to grpd to arrive at cntd. Having seen the data in grpd in the previous table, one can easily infer that the COUNT() function provided the size of each bag.

Although DESCRIBE and ILLUSTRATE are your workhorses in understanding Pig Latin statements, Pig also has an EXPLAIN command to show the logical and physical execution plan in detail. We summarize the diagnostic operators in table 10.3.

Table 10.3 Diagnostic operators in Pig Latin

DESCRIBE	`DESCRIBE alias;` Display the schema of a relation.
EXPLAIN	`EXPLAIN [-out path] [-brief] [-dot] [-param ...] [-param_file ...] alias;` Display the execution plan used to compute a relation. When used with a script name, for example, `EXPLAIN myscript.pig`, it will show the execution plan of the script.

ILLUSTRATE	`ILLUSTRATE alias;`
	Display step-by-step how data is transformed, starting with a load command, to arrive at the resulting relation. To keep the display and processing manageable, only a (not completely random) sample of the input data is used to simulate the execution.
	In the unfortunate case where none of Pig's initial sample will survive the script to generate meaningful data, Pig will "fake" some similar initial data that will survive to generate data for alias. For example, consider these operations:
	`A = LOAD 'student.data' as (name, age);`
	`B = FILTER A by age > 18;`
	`ILLUSTRATE B;`
	If every tuple Pig samples for A happens to have age less than or equal to 18, B is empty and not much is "illustrated." Instead Pig will construct for A some tuples with age greater than 18. This way B won't be an empty relation and users can see how the script works.
	In order for ILLUSTRATE to work, the load command in the first step must include a schema. The subsequent transformations must not include the LIMIT or SPLIT operators, or the nested FOREACH operator, or the use of the map data type (to be explained in section 10.5.1).

10.5 Speaking Pig Latin

You now know how to use Grunt to run Pig Latin statements and investigate their execution and results. We can come back and give a more formal treatment of the language. You should feel free to use Grunt to explore these language concepts as we present them.

10.5.1 Data types and schemas

Let's first look at Pig data types from a bottom-up view. Pig has six simple atomic types and three complex types, shown in tables 10.4 and 10.5 respectively. The atomic types include numeric scalars as well as string and binary objects. Type casting is supported and done in the usual manner. Fields default to `bytearray` unless specified otherwise.

Table 10.4 Atomic data types in Pig Latin

`int`	Signed 32-bit integer
`long`	Signed 64-bit integer
`float`	32-bit floating point
`double`	64-bit floating point
`chararray`	Character array (string) in Unicode UTF-8
`bytearray`	Byte array (binary object)

The three complex types are tuple, bag, and map.

A field in a tuple or a value in a map can be null or any atomic or complex type. This enables nesting and complex data structures. Whereas data structures can be arbitrarily complex, some are definitely more useful and occur more often than others, and nesting usually doesn't go deeper than two levels. In the Excite log example earlier, the `GROUP BY` operator generated a relation `grpd` where each tuple has a field that is a bag. The schema for the relation seems more natural once you think of `grpd` as the query history of each user. Each tuple represents one user and has a field that is a bag of the user's queries.

We can also look at Pig's data model from the top down. At the top, Pig Latin statements work with relations, which is a bag of tuples. If you force all the tuples in a bag

Table 10.5 Complex data types in Pig Latin

`Tuple`	`(12.5,hello world,-2)` A tuple is an ordered set of fields. It's most often used as a row in a relation. It's represented by fields separated by commas, all enclosed by parentheses.
`Bag`	`{(12.5,hello world,-2),(2.87,bye world,10)}` A bag is an unordered collection of tuples. A relation is a special kind of bag, sometimes called an outer bag. An inner bag is a bag that is a field within some complex type. A bag is represented by tuples separated by commas, all enclosed by curly brackets. Tuples in a bag aren't required to have the same schema or even have the same number of fields. It's a good idea to do this though, unless you're handling semistructured or unstructured data.
`Map`	`[key#value]` A map is a set of key/value pairs. Keys must be unique and be a string (`chararray`). The value can be any type.

to have a fixed number of fields and each field has a fixed atomic type, then it behaves like a relational data model—the relation is a table, tuples are rows (records), and fields are columns. But, Pig's data model has more power and flexibility by allowing *nested* data types. Fields can themselves be tuples, bags, or maps. Maps are helpful in processing semistructured data such as JSON, XML, and sparse relational data. In addition, it isn't necessary that tuples in a bag have the same number of fields. This allows tuples to represent unstructured data.

Besides declaring types for fields, schemas can also assign names to fields to make them easier to reference. Users can define schemas for relations using the `AS` keyword with the `LOAD`, `STREAM`, and `FOREACH` operators. For example, in the `LOAD` statement for getting the Excite query log, we defined the data types for the fields in `log`, as well as named the fields user, time, and query.

```
grunt> log  = LOAD 'tutorial/data/excite-small.log'
           ➥ AS (user:chararray, time:long, query:chararray);
```

In defining a schema, if you leave out the type, Pig will default to bytearray as the most generic type. You can also leave out the name, in which case a field would be unnamed and you can only reference it by position.

10.5.2 Expressions and functions

You can apply expressions and functions to data fields to compute various values. The simplest expression is a constant value. Next is to reference the value of a field. You can reference the named fields' value directly by the name. You can reference an unnamed field by `$`*n*, where *n* is its position inside the tuple. (Position is numbered starting at 0.) For example, this `LOAD` command provides named fields to `log` through the schema.

```
log  = LOAD 'tutorial/data/excite-small.log'
       ➥ AS (user:chararray, time:long, query:chararray);
```

The three named fields are user, time, and query. For example, we can refer to the time field as either `time` or `$1`, because the time field is the second field in `log` (position number 1). Let's say we want to extract the time field into its own relation; we can use this statement:

```
projection = FOREACH log GENERATE time;
```

We can also achieve the same with

```
projection = FOREACH log GENERATE $1;
```

Most of the time you should give names to fields. One use of referring to fields by position is when you're working with unstructured data.

When using complex types, you use the dot notation to reference fields nested inside tuples or bags. For example, recall earlier that we'd grouped the Excite log by user ID and arrived at relation `grpd` with a nested schema.

```
-------------------------------------------------------------------------
| grpd      | group: chararray | log: bag({user: chararray,time: long,      |
|           |                  |           query: chararray})               |
-------------------------------------------------------------------------
|           | 0567639EB8F3751C | {(0567639EB8F3751C, 970916161410,          |
|           |                  |   "conan o'brien"),                        |
|           |                  |  (0567639EB8F3751C,970916161413,           |
|           |                  |   "conan o'brien")}                        |
|           | 972F13CE9A8E2FA3 | {(972F13CE9A8E2FA3, 970916063540,          |
|           |                  |   finger AND download)}                    |
-------------------------------------------------------------------------
```

The second field in `grpd` is named `log`, of type bag. Each bag has tuples with three named fields: user, time, and query. In this relation, `log.query` would refer to the two copies of "conan" "o'brien" when applied to the first tuple. You can get the same field with `log.$2`.

You reference fields inside maps through the pound operator instead of the dot operator. For a map named `m`, the value associated with key `k` is referenced through `m#k`.

Being able to refer to values is only a first step. Pig supports the standard arithmetic, comparison, conditional, type casting, and Boolean expressions that are common in most popular programming languages. See table 10.6.

Table 10.6 Expressions in Pig Latin

Constant	`12, 19.2, 'hello world'`	Constant values such as 19 or "hello world." Numeric values without decimal point are treated as `int` unless l or L is appended to the number to make it a `long`. Numeric values with a decimal point are treated as `double` unless f or F is appended to the number to make it a `float`.
Basic arithmetic	`+,-,*,/`	Plus, minus, multiply, and divide.
Sign	`+x, -x`	Negation (-) changes the sign of a number.
Cast	`(t)x`	Convert the value of *x* into type *t*.
Modulo	`x % y`	The remainder of x divided by *y*.
Conditional	`(x ? y : z)`	Returns *y* if x is true, z otherwise. This expression must be enclosed in parentheses.
Comparison	`==,!=,<,>, <=,>=`	Equals to, not equals to, greater than, less than, etc.
Pattern matching	`x matches regex`	Regular expression matching of string x. Uses Java's regular expression format (under the `java.util.regex.Pattern` class) to specify `regex`.
Null	`x is null, x is not null`	Check if x is null (or not).
Boolean	`x and y, x or y not x`	Boolean and, or, not.

Furthermore, Pig also supports functions. Table 10.7 shows Pig's built-in functions, most of which are self-explanatory. We'll discuss user-defined functions (UDF) in section 10.6.

Table 10.7 Built-in functions in Pig Latin

AVG	Calculate the average of numeric values in a single-column bag.
CONCAT	Concatenate two strings (chararray) or two bytearrays.
COUNT	Calculate the number of tuples in a bag. See SIZE for other data types.
DIFF	Compare two fields in a tuple. If the two fields are bags, it will return tuples that are in one bag but not the other. If the two fields are values, it will emit tuples where the values don't match.
MAX	Calculate the maximum value in a single-column bag. The column must be a numeric type or a chararray.
MIN	Calculate the minimum value in a single-column bag. The column must be a numeric type or a chararray.
SIZE	Calculate the number of elements. For a bag it counts the number of tuples. For a tuple it counts the number of elements. For a chararray it counts the number of characters. For a bytearray it counts the number of bytes. For numeric scalars it always returns 1.
SUM	Calculate the sum of numeric values in a single-column bag.
TOKENIZE	Split a string (chararray) into a bag of words (each word is a tuple in the bag). Word separators are space, double quote ("), comma, parentheses, and asterisk (*).
IsEmpty	Check if a bag or map is empty.

You can't use expressions and functions alone. You must use them within relational operators to transform data.

10.5.3 Relational operators

The most salient characteristic about Pig Latin as a language is its relational operators. These operators define Pig Latin as a data processing language. We'll quickly go over the more straightforward operators first, to acclimate ourselves to their style and syntax. Afterward we'll go into more details on the more complex operators such as COGROUP and FOREACH.

UNION combines multiple relations together whereas SPLIT partitions a relation into multiple ones. An example will make it clear:

```
grunt> a = load 'A' using PigStorage(',') as (a1:int, a2:int, a3:int);
grunt> b = load 'B' using PigStorage(',') as (b1:int, b2:int, b3:int);
grunt> DUMP a;
(0,1,2)
(1,3,4)
grunt> DUMP b;
(0,5,2)
(1,7,8)
grunt> c = UNION a, b;
grunt> DUMP c;
```

```
(0,1,2)
(0,5,2)
(1,3,4)
(1,7,8)
grunt> SPLIT c INTO d IF $0 == 0, e IF $0 == 1;
grunt> DUMP d;
(0,1,2)
(0,5,2)
grunt> DUMP e;
(1,3,4)
(1,7,8)
```

The `UNION` operator allows duplicates. You can use the `DISTINCT` operator to remove duplicates from a relation. Our `SPLIT` operation on `c` sends a tuple to `d` if its first field (`$0`) is 0, and to `e` if it's 1. It's possible to write conditions such that some rows will go to both `d` and `e` or to neither. You can simulate `SPLIT` by multiple `FILTER` operators. The `FILTER` operator alone trims a relation down to only tuples that pass a certain test:

```
grunt> f = FILTER c BY $1 > 3;
grunt> DUMP f;
(0,5,2)
(1,7,8)
```

We've seen `LIMIT` being used to take a specified number of tuples from a relation. `SAMPLE` is an operator that randomly samples tuples in a relation according to a specified percentage.

The operations 'till now are relatively simple in the sense that they operate on each tuple as an atomic unit. More complex data processing, on the other hand, will require working on groups of tuples together. We'll next look at operators for grouping. Unlike previous operators, these grouping operators will create new schemas in their output that rely heavily on bags and nested data types. The generated schema may take a little time to get used to at first. Keep in mind that these grouping operators are almost always for generating intermediate data. Their complexity is only temporary on your way to computing the final results.

The simpler of these operators is `GROUP`. Continuing with the same set of relations we used earlier,

```
grunt> g = GROUP c BY $2;
grunt> DUMP g;
(2,{(0,1,2),(0,5,2)})
(4,{(1,3,4)})
(8,{(1,7,8)})

grunt> DESCRIBE c;
c: {a1: int,a2: int,a3: int}
grunt> DESCRIBE g;
g: {group: int,c: {a1: int,a2: int,a3: int}}
```

We've created a new relation, `g`, from grouping tuples in `c` having the same value on the third column (`$2`, also named `a3`). The output of `GROUP` always has two fields. The first field is group key, which is `a3` in this case. The second field is a bag containing

all the tuples with the same group key. Looking at g's dump, we see that it has three tuples, corresponding to the three unique values in c's third column. The bag in the first tuple represents all tuples in c with the third column equal to 2. The bag in the second tuple represents all tuples in c with the third column equal to 4. And so forth. After you understand how g's data came about, you'll feel more comfortable looking at its schema. The first field of GROUP's output relation is always named group, for the group key. In this case it may seem more natural to call the first field a3, but currently Pig doesn't allow you to assign a name to replace group. You'll have to adapt yourself to refer to it as group. The second field of GROUP's output relation is always named after the relation it's operating on, which is c in this case, and as we said earlier it's always a bag. As we use this bag to hold tuples from c, the schema for this bag is exactly the schema for c —three fields of integers named a1, a2, and a3.

Before moving on, we want to note that one can GROUP by any function or expression. For example, if time is a timestamp and there exists a function DayOfWeek, one can conceivably do this grouping that would create a relation with seven tuples.

```
GROUP log BY DayOfWeek(time);
```

Finally, one can put all tuples in a relation into one big bag. This is useful for aggregate analysis on relations, as functions work on bags but not relations. For example:

```
grunt> h = GROUP c ALL;
grunt> DUMP h;
(all,{(0,1,2),(0,5,2),(1,3,4),(1,7,8)})
grunt> i = FOREACH h GENERATE COUNT($1);
grunt> dump i;
(4L)
```

This is one way to count the number of tuples in c. The first field in GROUP ALL's output is always the string all.

Now that you're comfortable with GROUP, we can look at COGROUP, which groups together tuples *from multiple relations*. It functions much like a join. For example, let's cogroup a and b on the third column.

```
grunt> j = COGROUP a BY $2, b BY $2;
grunt> DUMP j;
(2,{(0,1,2)},{(0,5,2)})
(4,{(1,3,4)},{})
(8,{},{(1,7,8)})
grunt> DESCRIBE j;
j: {group: int,a: {a1: int,a2: int,a3: int},b: {b1: int,b2: int,b3: int}}
```

Whereas GROUP always generates two fields in its output, COGROUP always generates three (more if cogrouping more than two relations). The first field is the group key, whereas the second and third fields are bags. These bags hold tuples from the cogrouping relations that match the grouping key. If a grouping key matches only tuples from one relation but not the other, then the field corresponding to the nonmatching relation will have an empty bag. To ignore group keys that don't exist for a relation, one can add the INNER keyword to the operation, like

```
grunt> j = COGROUP a BY $2, b BY $2 INNER;
grunt> dump j;
(2,{(0,1,2)},{(0,5,2)})
(8,{},{(1,7,8)})
grunt> j = COGROUP a BY $2 INNER, b BY $2 INNER;
grunt> dump j;
(2,{(0,1,2)},{(0,5,2)})
```

Conceptually, you can think of the default behavior of `COGROUP` as an outer join, and the `INNER` keyword can modify it to be left outer join, right outer join, or inner join. Another way to do inner join in Pig is to use the `JOIN` operator. The main difference between `JOIN` and an inner `COGROUP` is that `JOIN` creates a flat set of output records, as indicated by looking at the schema:

```
grunt> j = JOIN a BY $2, b BY $2;
grunt> dump j;
(0,1,2,0,5,2)
grunt> DESCRIBE j;
j: {a::a1: int,a::a2: int,a::a3: int,b::b1: int,b::b2: int,b::b3: int}
```

The last relational operator we look at is `FOREACH`. It goes through all tuples in a relation and generates new tuples in the output. Behind this seeming simplicity lies tremendous power though, particularly when it's applied to complex data types outputted by the grouping operators. There's even a nested form of `FOREACH` designed for handling complex types. First let's familiarize ourselves with different `FOREACH` operations on simple relations.

```
grunt> k = FOREACH c GENERATE a2, a2 * a3;
grunt> DUMP k;
(1,2)
(5,10)
(3,12)
(7,56)
```

`FOREACH` is always followed by an alias (name given to a relation) followed by the keyword `GENERATE`. The expressions after `GENERATE` control the output. At its simplest, we use `FOREACH` to project specific columns of a relation into the output. We can also apply arbitrary expressions, such as multiplication in the preceding example.

For relations with nested bags (e.g., ones generated by the grouping operations), `FOREACH` has special projection syntax, and a richer set of functions. For example, applying nested projection to have each bag retain only the first field:

```
grunt> k = FOREACH g GENERATE group, c.a1;
grunt> DUMP k;
(2,{(0),(0)})
(4,{(1)})
(8,{(1)})
```

To get two fields in each bag:

```
grunt> k = FOREACH g GENERATE group, c.(a1,a2);
grunt> DUMP k;
(2,{(0,1),(0,5)})
```

```
(4,{(1,3)})
(8,{(1,7)})
```

Most built-in Pig functions are geared toward working on bags.

```
grunt> k = FOREACH g GENERATE group, COUNT(c);
grunt> DUMP k;
(2,2L)
(4,1L)
(8,1L)
```

Recall that `g` is based on grouping `c` on the third column. This `FOREACH` statement therefore generates a frequency count of the values in `c`'s third column. As we said earlier, grouping operators are mainly for generating intermediate data that will be simplified by other operators such as `FOREACH`. The `COUNT` function is one of the aggregate functions. As we'll see, you can create many other functions via UDFs.

The `FLATTEN` function is designed to flatten nested data types. Syntactically it looks like a function, such as `COUNT` and `AVG`, but it's a special operator as it can change the structure of the output created by `FOREACH...GENERATE`. Its flattening behavior is also different depending on how it's applied and what it's applied to. For example, consider a relation with tuples of the form (a, (b, c)). The statement `FOREACH... GENERATE $0, FLATTEN($1)` will create one output tuple of the form (a, b, c) for each input tuple.

When applied to bags, `FLATTEN` modifies the `FOREACH...GENERATE` statement to generate new tuples. It removes one layer of nesting and behaves almost the opposite of grouping operations. If a bag contains *N* tuples, flattening it will remove the bag and create *N* tuples in its place.

```
grunt> k = FOREACH g GENERATE group, FLATTEN(c);
grunt> DUMP k;
(2,0,1,2)
(2,0,5,2)
(4,1,3,4)
(8,1,7,8)
grunt> DESCRIBE k;
k: {group: int,c::a1: int,c::a2: int,c::a3: int}
```

Another way to understand `FLATTEN` is to see that it produces a cross-product. This view is helpful when we use `FLATTEN` multiple times within a single `FOREACH` statement. For example, let's say we've somehow created a relation `l`.

```
grunt> dump l;
(1,{(1,2)},{(3)})
(4,{(4,2),(4,3)},{(6),(9)})
(8,{(8,3),(8,4)},{(9)})
grunt> describe l;
d: {group: int,a: {a1: int,a2: int},b: {b1: int}}
```

The following statement that flattens two bags outputs all combinations of those two bags for each tuple:

```
grunt> m = FOREACH l GENERATE group, FLATTEN(a), FLATTEN(b);
grunt> dump m;
```

```
(1,1,2,3)
(4,4,2,6)
(4,4,2,9)
(4,4,3,6)
(4,4,3,9)
(8,8,3,9)
(8,8,4,9)
```

We see that the tuple with group key 4 in relation `l` has a bag of size 2 in field `a` and also a bag size 2 in field `b`. The corresponding output in `m` therefore has four rows representing the full cross-product.

Finally, there's a nested form of `FOREACH` to allow for more complex processing of bags. Let's assume you have a relation (say `l`) and one of its fields (say `a`) is a bag, a `FOREACH` with nested block has this form:

```
alias = FOREACH l {
            tmp1 = operation on a;
            [more operations...]
            GENERATE expr [, expr...]
        }
```

The `GENERATE` statement must always be present at the end of the nested block. It will create some output for each tuple in `l`. The operations in the nested block can create new relations based on the bag `a`. For example, we can trim down the `a` bag in each element of `l`'s tuple.

```
grunt> m = FOREACH l {
        tmp1 = FILTER a BY a1 >= a2;
        GENERATE group, tmp1, b;
    }
grunt> DUMP m;
(1,{},{(3)})
(4,{(4,2),(4,3)},{(6),(9)})
(8,{(8,3),(8,4)},{(9)})
```

You can have multiple statements in the nested block. Each one can even be operating on different bags.

```
grunt> m = FOREACH l {
        tmp1 = FILTER a BY a1 >= a2;
        tmp2 = FILTER b by $0 < 7;
        GENERATE group, tmp1, tmp2;
    };
grunt> DUMP m;
(1,{},{(3)})
(4,{(4,2),(4,3)},{(6)})
(8,{(8,3),(8,4)},{})
```

As of this writing, only five operators are allowed in the nested block: `DISTINCT`, `FILTER`, `LIMIT`, `ORDER`, and `SAMPLE`. It's expected that more will be supported in the future.

NOTE Sometimes the output of `FOREACH` can have a completely different schema from its input. In those cases, users can specify the output schema using

the `AS` option after each field. This syntax differs from the `LOAD` command where the schema is specified as a list after the `AS` option, but in both cases we use `AS` to specify a schema.

Table 10.8 summarizes the relational operators in Pig Latin. On many operators you'll see an option for `PARALLEL n`. The number `n` is the degree of parallelism you want for executing that operator. In practice `n` is the number of reduce tasks in Hadoop that Pig will use. If you don't set `n` it'll default to the default setting of your Hadoop cluster. Pig documentation recommends setting the value of `n` according to the following guideline:

```
n = (#nodes - 1) * 0.45 * RAM
```

where `#nodes` is the number of nodes and `RAM` is the amount of memory in GB on each node.

Table 10.8 Relational operators in Pig Latin

`SPLIT`	`SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression ...];` Splits a relation into two or more relations, based on the given Boolean expressions. Note that a tuple can be assigned to more than one relation, or to none at all.
`UNION`	`alias = UNION alias, alias, [, alias ...]` Creates the union of two or more relations. Note that ■ As with any relation, there's no guarantee to the order of tuples ■ Doesn't require the relations to have the same schema or even the same number of fields ■ Doesn't remove duplicate tuples
`FILTER`	`alias = FILTER alias BY expression;` Selects tuples based on Boolean expression. Used to select tuples that you want or remove tuples that you don't want.
`DISTINCT`	`alias = DISTINCT alias [PARALLEL n];` Remove duplicate tuples.
`SAMPLE`	`alias = SAMPLE alias factor;` Randomly sample a relation. The sampling factor is given in `factor`. For example, a 1% sample of data in relation `large_data` is `small_data = SAMPLE large_data 0.01;` The operation is probabilistic in such a way that the size of `small_data` will not be exactly 1% of `large_data`, and there's no guarantee the operation will return the same number of tuples each time.
`FOREACH`	`alias = FOREACH alias GENERATE expression [,expression ...] [AS schema];` Loop through each tuple and generate new tuple(s). Usually applied to transform columns of data, such as adding or deleting fields. One can optionally specify a schema for the output relation; for example, naming new fields.

Table 10.8 Relational operators in Pig Latin (*continued*)

FOREACH (nested)	`alias = FOREACH nested_alias {`
	`  alias = nested_op;`
	`  [alias = nested_op; ...]`
	`  GENERATE expression [, expression ...];`
	`};`
	Loop through each tuple in `nested_alias` and generate new tuple(s). At least one of the fields of `nested_alias` should be a bag. `DISTINCT`, `FILTER`, `LIMIT`, `ORDER`, and `SAMPLE` are allowed operations in `nested_op` to operate on the inner bag(s).

JOIN	`alias = JOIN alias BY field_alias, alias BY field_alias [, alias BY field_alias …] [USING "replicated"] [PARALLEL n];` Compute inner join of two or more relations based on common field values. When using the replicated option, Pig stores all relations after the first one in memory for faster processing. You have to ensure that all those smaller relations together are indeed small enough to fit in memory. Under `JOIN`, when the input relations are flat, the output relation is also flat. In addition, the number of fields in the output relation is the sum of the number of fields in the input relations, and the output relation's schema is a concatenation of the input relations' schemas.
GROUP	`alias = GROUP alias { [ALL] | [BY {[field_alias [, field_alias]] | * | [expression]] } [PARALLEL n];` Within a single relation, group together tuples with the same group key. Usually the group key is one or more fields, but it can also be the entire tuple (`*`) or an expression. One can also use `GROUP alias ALL` to group all tuples into one group. The output relation has two fields with autogenerated names. The first field is always named "group" and it has the same type as the group key. The second field takes the name of the input relation and is a bag type. The schema for the bag is the same as the schema for the input relation.
COGROUP	`alias = COGROUP alias BY field_alias [INNER | OUTER] , alias BY field_alias [INNER | OUTER] [PARALLEL n];` Group tuples from two or more relations, based on common group values. The output relation will have a tuple for each unique group value. Each tuple will have the group value as its first field. The second field is a bag containing tuples from the first input relation with matching group value. Ditto for the third field of the output tuple. In the default `OUTER` join semantic, all group values appearing in any input relation are represented in the output relation. If an input relation doesn't have any tuple with a particular group value, it will have an empty bag in the corresponding output tuple. If the `INNER` option is set for a relation, then only group values that exist in that input relation are allowed in the output relation. There can't be an empty bag for that relation in the output. You can group on multiple fields. For this, you have to specify the fields in a comma-separated list enclosed by parentheses for `field_alias`. `COGROUP` (with `INNER`) and `JOIN` are similar except that `COGROUP` generates nested output tuples.
CROSS	`alias = CROSS alias, alias [, alias …] [PARALLEL n];` Compute the (flat) cross-product of two or more relations. This is an expensive operation and you should avoid it as far as possible.

Table 10.8 Relational operators in Pig Latin (*continued*)

ORDER	`alias = ORDER alias BY { * [ASC|DESC] | field_alias [ASC|DESC] [, field_alias [ASC|DESC] …] } [PARALLEL n];` Sort a relation based on one or more fields. If you retrieve the relation right after the ORDER operation (by DUMP or STORE), it's guaranteed to be in the desired sorted order. Further processing (FILTER, DISTINCT, etc.) may destroy the ordering.
STREAM	`alias = STREAM alias [, alias …] THROUGH {'command' | cmd_alias } [AS schema] ;` Process a relation with an external script.

At this point you've learned various aspects of the Pig Latin language—data types, expressions, functions, and relational operators. You can extend the language further with user-defined functions. But before discussing that we'll end this section with a note on Pig Latin compilation and optimization.

10.5.4 Execution optimization

As with many modern compilers, the Pig compiler can reorder the execution sequence to optimize performance, as long as the execution plan remains logically equivalent to the original program. For example, imagine a program that applies an expensive function (say, encryption) to a certain field (say, social security number) of every record, followed by a filtering function to select records based on a different field (say, limit only to people within a certain geography). The compiler can reverse the execution order of those two operations without affecting the final result, yet performance is much improved. Having the filtering step first can dramatically reduce the amount of data and work the encryption step will have to do.

As Pig matures, more optimization will be added to the compiler. Therefore it's important to try to always use the latest version. But there's always a limit to a compiler's ability to optimize arbitrary code. You can read Pig's web documentation for techniques to improve performance. A list of tips for enhancing performance under Pig version 0.3 is at http://hadoop.apache.org/pig/docs/r0.3.0/cookbook.html.

10.6 Working with user-defined functions

Fundamental to Pig Latin's design philosophy is its extensibility through user-defined functions (UDFs), and there's a well-defined set of APIs for writing UDFs. This doesn't mean that you'll have to write all the functions you need yourself. Part of Pig's ecosystem[4] is PiggyBank,[5] an online repository for users to share their functions. You should

[4] I thought about calling it a Pig pen, but PigPen is actually the name of an Eclipse plug-in for editing Pig Latin scripts. See http://wiki.apache.org/pig/PigPen.

[5] http://wiki.apache.org/pig/PiggyBank.

check PiggyBank first for any function you need. Only if you don't find an appropriate function should you consider writing your own. You should also consider contributing your UDF back to PiggyBank to benefit others in the Pig community.

10.6.1 Using UDFs

As of this writing UDFs are always written in Java and packaged in jar files. To use a particular UDF you'll need the jar file containing the UDF's class file(s). For example, when using functions from PiggyBank you'll most likely obtain a piggybank.jar file.

To use a UDF, you must first register the jar file with Pig using the `REGISTER` statement. Afterward, you invoke the UDF by its fully qualified Java class name. For example, there's an `UPPER` function in PiggyBank that transforms a string to uppercase:

```
REGISTER piggybank/java/piggybank.jar;
b = FOREACH a GENERATE
      ➥ org.apache.pig.piggybank.evaluation.string.UPPER($0);
```

If you need to use a function multiple times, it'll get annoying to write out the fully qualified class name every time. Pig offers the `DEFINE` statement to assign a name to a UDF. You can rewrite the above statements to

```
REGISTER piggybank/java/piggybank.jar;
DEFINE Upper org.apache.pig.piggybank.evaluation.string.UPPER();
b = FOREACH a GENERATE Upper($0);
```

Table 10.9 summarizes the UDF-related statements.

Table 10.9 UDF statements in Pig Latin

`DEFINE`	`DEFINE alias { function | 'command' [...] };` Assign an alias to a function or command.
`REGISTER`	`REGISTER alias;` Register UDFs with Pig. Currently UDFs are only written in Java, and *alias* is the path of the JAR file. All UDFs must be registered before they can be used.

If you're only using UDFs written by other people, this is all you need to know. But if you can't find the UDF you need, you'll have to write your own.

10.6.2 Writing UDFs

Pig supports two main categories of UDFs: eval[6] and load/store. We use the load/store functions only in `LOAD` and `STORE` statements to help Pig read and write special formats. Most UDFs are eval functions that take one field value and return another field value.

[6] Some eval functions are quite common and have special considerations. They're sometimes described in their own categories. These include filter functions (eval functions that return a Boolean) and aggregate functions (eval functions that take a bag and return a scalar value).

As of this writing, you can only write a UDF using Pig's Java API.[7] To create an eval UDF you make a Java class that extends the abstract `EvalFunc<T>` class. It has only one abstract method which you need to implement:

```
abstract public T exec(Tuple input) throws IOException;
```

This method is called on each tuple in a relation, where each tuple is represented by a `Tuple` object. The `exec()` method processes the tuple and returns a type `T` corresponding to a valid Pig Latin type. `T` can be any one of the Java classes in table 10.10, some of which are native Java classes and some of which are Pig extensions.

Table 10.10 Pig Latin types and their equivalent classes in Java.

Pig Latin type	Java class
Bytearray	DataByteArray
Chararray	String
Int	Integer
Long	Long
Float	Float
Double	Double
Tuple	Tuple
Bag	DataBag
Map	Map<Object, Object>

The best way to learn about writing UDFs is to dissect one of the existing UDFs in PiggyBank. Even when writing your own, it's often useful to start with a working UDF that's functionally similar to what you want and only modify the processing logic. For our purpose, let's explore the `UPPER` UDF we used earlier from PiggyBank. The `exec()` method looks like this:

```
public class UPPER extends EvalFunc<String>
{
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;

        try {
            String str = (String)input.get(0);
            return str.toUpperCase();
        } catch(Exception e){
            System.err.println("Failed to process input; error - " +
                               ➥ e.getMessage());
            return null;
        }
    }
}
```

[7] The Javadoc for the API is at http://hadoop.apache.org/pig/javadoc/docs/api/.

The object `input` belongs to the `Tuple` class, which has two methods for retrieving its content.

```
List<Object> getAll();
Object get(int fieldNum) throws ExecException;
```

The `getAll()` method return all fields in the tuple as an ordered list. `UPPER` instead uses the `get()` method to request for a specific field (at position 0). This method would throw an `ExecException` if the requested field number is greater than the number of fields in the tuple. In `UPPER` the retrieved field is casted to a Java String, which usually works but may cause a cast exception if we were casting between incompatible data types. We'll see later how to use Pig to ensure that our casting works. In any case, the try/catch block would've caught and handled any exception. If everything works, `UPPER`'s `exec()` method will return a String with characters uppercased. In addition, most UDFs should implement the default behavior that the output is null when the input tuple is null.

In addition to implementing `exec()`, `UPPER` also overrides a couple methods from `EvalFunc`, one of which is `getArgToFuncMapping`:

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

The `getArgToFuncMapping()` method returns a `List` of `FuncSpec` objects representing the schema of each field in the `input` tuple. Pig will handle typecasting for you by converting the types of all fields in a tuple to conform to this schema before passing it to `exec()`. It will pass fields that can't be converted to the desired type as null.

`UPPER` only cares about the type of the first field, so it adds only one `FuncSpec` to the list, and this `FuncSpec` states that the field must be of type `chararray`, represented as `DataType.CHARARRAY`. The instantiation of `FuncSpec` is quite convoluted, which is due to Pig's ability to handle complex nested types. Fortunately, unless you work with unusually complicated types, you'll probably find a `FuncSpec` instantiation for the type you want already in one of PiggyBank's UDFs. Reuse that in your code. You can even reuse the entire `getArgToFuncMapping()` function if you have the same tuple schema as another UDF.

Besides telling Pig the input schema, you can also tell Pig the schema of your output. You may not need to do this if the output of your UDF is a simple scalar, as Pig will use Java's Reflection mechanism to infer the schema automatically. But if your UDF returns a tuple or a bag, the Reflection mechanism will fail to figure out the schema completely. In that case you should specify it so that Pig can propagate the schema correctly.

In UPPER's case it only outputs a simple String, so it's not necessary to specify the output schema. But UPPER does do this by overriding outputSchema() to tell Pig that it's returning a string (DataType.CHARARRAY).

```
@Override
public Schema outputSchema(Schema input) {
    return new Schema(
        new Schema.FieldSchema(
            getSchemaName(this.getClass().getName().toLowerCase(), input),
            DataType.CHARARRAY
        )
    );
}
```

Again, the Schema object construction looks convoluted because of Pig's ability to have complex nested types. One special case is if the schema of your UDF's output is the same as the input. We can return a copy of the input schema:

```
public Schema outputSchema(Schema input) {
    return new Schema(input);
}
```

As with the construction of FuncSpec, you'll probably find some preexisting UDFs in PiggyBank with your desired output schema.

A few types of UDFs call for special considerations. Filter functions are eval functions that return a Boolean, and we use them in Pig Latin's FILTER and SPLIT statements. They should extend FilterFunc instead of EvalFunc. Aggregate functions are eval functions that take in a bag and return a scalar. They're usually used for computing aggregate metrics, such as COUNT, and we can sometimes optimize them in Hadoop by using a combiner. We haven't covered the load/save UDFs for reading and writing data sets. These more advanced topics are covered in Pig's documentation on UDFs: http://hadoop.apache.org/pig/docs/r0.3.0/udf.html.

10.7 Working with scripts

Writing Pig Latin scripts is largely about packaging together the Pig Latin statements that you've successfully tested in Grunt. Pig scripting does have a few unique topics though. They're comments, parameter substitution, and multiquery execution.

10.7.1 Comments

As you'll reuse your Pig Latin script, it's obviously a good idea to leave comments for other people (or yourself) to understand it in the future. Pig Latin supports two forms of comments, single-line and multiline. You start the single-line comment by a double hyphen and the comment ends at the end of the line. You enclose the multiline comment by the /* and */ markers, similar to multiline comments in Java. For example, a Pig Latin script with comments can look like

```
/*
 * Myscript.pig
 * Another line of comment
```

```
 */
log = LOAD 'excite-small.log' AS (user, time, query);
lmt = LIMIT log 4; -- Only show 4 tuples
DUMP lmt;
-- End of program
```

10.7.2 Parameter substitution

When you write a reusable script, it's generally parameterized such that you can vary its operation for each run. For example, the script may take the file paths of its input and output from the user each time. Pig supports parameter substitution to allow the user to specify such information at runtime. It denotes such parameters by the `$` prefix within the script. For example, the following script displays a user-specified number of tuples from a user-specified log file:

```
log = LOAD '$input' AS (user, time, query);
lmt = LIMIT log $size;
DUMP lmt;
```

The parameters in this script are `$input` and `$size`. If you run this script using the `pig` command, you specify the parameters using the `-param name=value` argument.

```
pig -param input=excite-small.log -param size=4 Myscript.pig
```

Note that you don't need the `$` prefix in the arguments. You can enclose a parameter value in single or double quotes, if it has multiple words. A useful technique is to use Unix commands to generate the parameter values, particularly for dates. This is accomplished through Unix's command substitution, which executes commands enclosed in back ticks (`` ` ``).

```
pig -param input=web-'date +%y-%m-%d'.log -param size=4 Myscript.pig
```

By doing this, the input file for `Myscript.pig` will be based on the date the script is run. For example, the input file will be `web-09-07-29.log` if the script is run on July 29, 2009.

If you have to specify many parameters, it may be more convenient to put them in a file and tell Pig to execute the script using parameter substitution based on that file. For example, we can create a file Myparams.txt with the following content:

```
# Comments in a parameter file start with hash
input=excite-small.log
size=4
```

The parameter file is passed to the pig command with the `-param_file filename` argument.

```
pig -param_file Myparams.txt Myscript.pig
```

You can specify multiple parameter files as well as mix parameter files with direct specification of parameters at the command line using `-param`. If you define a parameter multiple times, the last definition takes precedence. When in doubt about what parameter values a script ends up using, you can run the pig command with the `-debug` option.

This tells Pig to run the script and also output a file named `original_script_name.substituted` that has the original script but with all the parameters fully substituted. Executing `pig` with the `-dryrun` option outputs the same file but doesn't execute the script.

The `exec` and `run` commands allow you to run Pig Latin scripts from within the Grunt shell, and they support parameter substitution using the same `-param` and `-param_file` arguments; for example:

```
grunt> exec -param input=excite-small.log -param size=4 Myscript.pig
```

However, parameter substitution in `exec` and `run` doesn't support Unix commands, and there's no `debug` or `dryrun` option.

10.7.3 Multiquery execution

In the Grunt shell, a `DUMP` or `STORE` operation processes all previous statements needed for the result. On the other hand, Pig optimizes and processes an entire Pig script as a whole. This difference would have no effect at all if your script has only one `DUMP` or `STORE` command at the end. If your script has multiple `DUMP/STORE`, Pig script's *multiquery execution* improves efficiency by avoiding redundant evaluations. For example, let's say you have a script that stores intermediate data:

```
a = LOAD ...
b = some transformation of a
STORE b ...
c = some further transformation of b
STORE c ...
```

If you enter the statements in Grunt, where there's no multiquery execution, it will generate a chain of jobs on the `STORE b` command to compute `b`. On encountering `STORE c`, Grunt will run another chain of jobs to compute `c`, but this time it will evaluate both `a` and `b` again! You can manually avoid this reevaluation by inserting a `b = LOAD ...` statement right after `STORE b`, to force the computation of `c` to use the saved value of `b`. This works on the assumption that the stored value of `b` has not been modified, because Grunt, by itself, has no way of knowing.

On the other hand, if you run all the statements as a script, multiquery execution can optimize the execution by intelligently handling intermediate data. Pig compiles all the statements together and can locate the dependency and redundancy. Multiquery execution is enabled by default and usually has no effect on the computed results. But multiquery execution can fail if there are data dependencies that Pig is not aware of. This is quite rare but can happen with, for example, UDFs. Consider this script:

```
STORE a INTO 'out1';
b = LOAD ...
c = FOREACH b GENERATE MYUDF($0,'out1');
STORE c INTO 'out2';
```

If the custom function `MYUDF` is such that it accesses `a` through the file `out1`, the Pig compiler would have no way of knowing that. Not seeing the dependency, the Pig compiler may erroneously think it OK to evaluate `b` and `c` before evaluating `a`. To disable multiquery execution, run the `pig` command with `-M` or `-no_multiquery` option.

10.8 Seeing Pig in action—example of computing similar patents

Given the extra power that Pig provides, we can take on more challenging data processing applications. One interesting application from the patent data set is finding similar patents based on citation data. Patents that are often cited together must be similar (or at least related) in some way. This application has the essence of the Amazon.com style collaborative filtering ("Customers who have bought *this* have also bought *that*.") and finding similar documents (by looking for documents with a similar set of words). For our purpose here, let's suppose we want to look into patents that are cited together more than *N* times, where *N* is a fixed number we specify.[8]

For applications that involve pair-wise computations (e.g., computing number of cocitations for each pair of patents), it's often easy to imagine an implementation involving a pair of nested loops enumerating all pair combinations and performing the computation on each pair. Even though Hadoop makes it easy to scale by adding more hardware, we should continue to remember fundamental concepts in computational complexity. Quadratic complexity will still bring linear scalability to its knees. Even a small data set of 3 million patents can lead to 9 trillion pairs. We need smarter algorithms.

The main insight to leverage is that the resulting data is *sparse*. Most pairs will have zero similarity as most pairs of patents are never cited together. Our similarity computation will become much more manageable if we redesign it to only work on patent pairs that are known to have been cited together. Looking at our data, this approach is quite natural. This implementation involves these steps for each patent:

1. Get the list of patents it cites
2. Generate all pair-wise combinations of the list and record each pair
3. Count how many of each pair we have

If each patent cites a fixed number of patents, say 10, this implementation would generate 45 pairs for each patent. (45 is the number of pair combinations possible from 10 items, which mathematically is derived as 10 x 9 / 2.) With 3 million patents this creates 135 million pairs, which is orders of magnitude smaller than the brute force approach. This advantage would be even more apparent if the patent data set is larger.

Even though we've figured out the algorithm for this application, implementing it in MapReduce can still be tedious. It'll require chaining multiple jobs together, and each job will require its own class. Pig Latin, on the other hand, takes only a dozen lines to implement the three-step program (listing 10.1), and further optimization can eliminate more lines and increase efficiency still.

Listing 10.1 Pig Latin script to find patents that are often cited together

```
cite = LOAD 'input/cite75_99.txt' USING PigStorage(',')
       ➥ AS (citing:int, cited:int);
cite_grpd = GROUP cite BY citing;
```

[8] Variations of this may involve more advanced scoring functions, such as normalizing for frequent items, or computing a similarity ranking rather than a simple cutoff. The simple cutoff criterion we chose here is easier to implement and illustrates the essence of computing similarity.

```
  cite_grpd_dbl = FOREACH cite_grpd GENERATE group, cite.cited AS cited1,
                  ➥ cite.cited AS cited2;
         cocite = FOREACH cite_grpd_dbl
                  ➥ GENERATE FLATTEN(cited1), FLATTEN(cited2);
    cocite_fltrd = FILTER cocite BY cited1 != cited2;
    cocite_grpd = GROUP cocite_fltrd BY *;
     cocite_cnt = FOREACH cocite_grpd
                  ➥ GENERATE group, COUNT(cocite_fltrd) as cnt;
     cocite_flat = FOREACH cocite_cnt GENERATE FLATTEN(group), cnt;
cocite_cnt_grpd = GROUP cocite_flat BY cited1;
     cocite_bag = FOREACH cocite_cnt_grpd
                  ➥ GENERATE group, cocite_flat.(cited2, cnt);

cocite_final = FOREACH cocite_cnt_grpd {
    similar = FILTER cocite_flat BY cnt > 5;
    GENERATE group, similar;
}
STORE cocite_final INTO 'output';
```

Pig Latin, and probably complex data processing in general, can be hard to read. Fortunately, we can use Grunt's `ILLUSTRATE` command on `cocite_bag` to get a simulated sample run of the statements and see what each operation is generating. (We've reformatted the output to fit the width of the printed page.)

```
-------------------------------------------------
| cite     | citing: bytearray | cited: bytearray |
-------------------------------------------------
|          | 3858554           | 3601095          |
|          | 3858554           | 3685034          |
|          | 3859004           | 1730866          |
|          | 3859004           | 3022581          |
|          | 3859572           | 3206651          |
-------------------------------------------------
---------------------------------------
| cite     | citing: int | cited: int |
---------------------------------------
|          | 3858554     | 3601095    |
|          | 3858554     | 3685034    |
|          | 3859004     | 1730866    |
|          | 3859004     | 3022581    |
|          | 3859572     | 3206651    |
---------------------------------------
------------------------------------------------------------------------
| cite_grpd     | group: int | cite: bag({citing: int,cited: int})      |
------------------------------------------------------------------------
|               | 3858554    | {(3858554, 3601095), (3858554, 3685034)} |
|               | 3859004    | {(3859004, 1730866), (3859004, 3022581)} |
|               | 3859572    | {(3859572, 3206651)}                     |
------------------------------------------------------------------------
-----------------------------------------------------------------------------
| cite_grpd_dbl | group:  | cited1:                | cited2:                |
|               |    int  |     bag({cited: int})  |     bag({cited: int})|
-----------------------------------------------------------------------------
|               | 3858554 | {(3601095), (3685034)} | {(3601095),(3685034)}|
|               | 3859004 | {(1730866), (3022581)} | {(1730866),(3022581)}|
|               | 3859572 | {(3206651)}            | {(3206651)}          |
-----------------------------------------------------------------------------
```

The relation `cite_grpd` contains a bag for each patent, and in this bag are the cited patents. From this relation (in this example run), we can see that patents 3601095 and 3685034 are cited together in patent 3858554. Grouping cocited patents was done by the `GROUP` operation in creating `cite_grpd`. The relation `cite_grpd_dbl` only removes the redundant "citing" patent and creates a duplicate column. The columns `cited1` and `cited2` have the same values. This duplication will allow the cross-product operation to generate all pair-wise combinations.

```
-------------------------------------------------------
| cocite     | cited1::cited: int | cited2::cited: int |
-------------------------------------------------------
|            | 3601095            | 3601095            |
|            | 3601095            | 3685034            |
|            | 3685034            | 3601095            |
|            | 3685034            | 3685034            |
|            | 1730866            | 1730866            |
|            | 1730866            | 3022581            |
|            | 3022581            | 1730866            |
|            | 3022581            | 3022581            |
|            | 3206651            | 3206651            |
-------------------------------------------------------
```

The cross-product from flattening each row of `cite_grpd_dbl` creates `cocite`.[9] This is the record of all pairs of patents that have been cited together and is a major checkpoint for our algorithm. We know that cocite is a big relation, even under our scheme which is more efficient than brute force. There are three ways to trim down cocite further. We'll discuss them all but implement only one.

The first potential reduction is to notice that each cited patent is considered to have been cocited with itself. As we know that it's quite pointless for our application to figure out that a patent is similar to itself, we can ignore all such pairs. Note that if we keep these "identity" pairs in the calculation, the cocitation count for them will end up being exactly the citation count. These numbers can still be useful if we're looking for the percentage of times patents are cocited. As we're not computing percentages, that consideration wouldn't affect us.

As cocitation is symmetric, pairs always appear twice, in reverse order. For example, we see both (3601095,3685034) and (3685034,3601095) when they appear together once. Given our current application need to find patent pairs that are cocited more than *N* times together, we can put in a simple rule retaining only one of the two redundant pairs and trim `cocite`'s size by half. This rule can be thus: retain only pairs where the first field is smaller than the second field. But keeping the redundant pairs can be useful for lookup later in some applications. For example, we can find all patents cocited with X by searching for X in the first field. In the more condensed version we'd have to look for X in both fields.

Finally, we can use heuristics to remove cocitation pairs that we don't think are important. We compromise final precision to gain efficiency. The applicability

[9] Note that `cocite` can be computed from `cite_grpd` directly by using a more complicated `FOREACH` statement, and you may choose to do it when you feel more comfortable reading Pig Latin.

and usefulness of heuristics will depend on the application semantics and the data distribution. In our case, a patent that cites many patents together will generate a quadratic number of rows in `cocite`. If we believe that such "verbose" patents don't help us understand similar patent pairs, removing them can significantly reduce the size of data to process with little impact on final results. The benefit of this heuristic is much greater if we're looking at reverse patent citation or text documents, where frequency of items are extremely skewed and quadratic expansion on a few popular items can dominate the amount of data processed. In fact, in such situations approximate heuristics are almost necessary.

An important process check is to note that we've focused on a higher level of data processing issues. We've obviated any low-level discussion about MapReduce.

```
-------------------------------------------------------------
| cocite_fltrd     | cited1::cited: int | cited2::cited: int |
-------------------------------------------------------------
|                  | 3601095            | 3685034            |
|                  | 3685034            | 3601095            |
|                  | 1730866            | 3022581            |
|                  | 3022581            | 1730866            |
-------------------------------------------------------------
```

We've decided to only filter out "identity" patent pairs:

```
-------------------------------------------------------------------------
| cocite_grpd | group:                       | cocite_fltrd:              |
|             | tuple({cited1::cited: int,   | bag({cited1::cited: int,   |
|             |        cited2::cited: int})  |      cited2::cited: int})  |
-------------------------------------------------------------------------
|             | (1730866, 3022581)           | {(1730866, 3022581)}       |
|             | (3022581, 1730866)           | {(3022581, 1730866)}       |
|             | (3601095, 3685034)           | {(3601095, 3685034)}       |
|             | (3685034, 3601095)           | {(3685034, 3601095)}       |
-------------------------------------------------------------------------
-------------------------------------------------------------------------
| cocite_cnt | group:                                          | cnt:     |
|            |    tuple({cited1::cited: int,cited2::cited: int}) |   long |
-------------------------------------------------------------------------
|            | (1730866, 3022581)                              | 1        |
|            | (3022581, 1730866)                              | 1        |
|            | (3601095, 3685034)                              | 1        |
|            | (3685034, 3601095)                              | 1        |
-------------------------------------------------------------------------
-------------------------------------------------------------------------
| cocite_flat | group::cited1::cited: | group::cited2::cited: | cnt:      |
|             |                   int |                   int |     long  |
-------------------------------------------------------------------------
|             | 1730866               | 3022581               | 1         |
|             | 3022581               | 1730866               | 1         |
|             | 3601095               | 3685034               | 1         |
|             | 3685034               | 3601095               | 1         |
-------------------------------------------------------------------------
```

We grouped the patent pair citations together, counted them, and flattened out the relation. Unfortunately, `ILLUSTRATE` generates sample data that only has cocitation counts of 1. However, we see that the operations are doing basically what we wanted. If we stick to the original application requirement of only looking for patent pairs that have been cocited more than *N* times, we would apply a filter on `cocite_flat` and be finished. But we want to show how we can further group the tuples, which would be needed for other types of filtering. For example, you may want to find the *K* most cocited patents for each patent. Let's look at the rest of the output:

```
-------------------------------------------------------------------------
| cocite_cnt_grpd | group: int | cocite_flat: bag({group::cited1::cited:  |
|                 |            | int,group::cited2::cited: int,cnt: long})|
-------------------------------------------------------------------------
|                 | 1730866    | {(1730866, 3022581, 1)}                  |
|                 | 3022581    | {(3022581, 1730866, 1)}                  |
|                 | 3601095    | {(3601095, 3685034, 1)}                  |
|                 | 3685034    | {(3685034, 3601095, 1)}                  |
-------------------------------------------------------------------------
-------------------------------------------------------------------------
| cocite_bag  | group: int | cocite_flat:                                |
|             |            |   bag({group::cited2::cited: int,cnt: long}) |
-------------------------------------------------------------------------
|             | 1730866    | {(3022581, 1)}                              |
|             | 3022581    | {(1730866, 1)}                              |
|             | 3601095    | {(3685034, 1)}                              |
|             | 3685034    | {(3601095, 1)}                              |
-------------------------------------------------------------------------
```

If we had wanted to find each patent's *K* most cocited patents, we would use a `FOREACH` statement to process each tuple in `cocite_bag` and write our own UDF to take in a bag (`cocite_flat`) and return a bag of at most *K* tuples (the most cocited ones). You can do this final step as an exercise. Let's see an example of a nested `FOREACH` statement to filter out tuples inside bags that have counts of 5 or less.

```
cocite_final = FOREACH cocite_cnt_grpd {
    similar = FILTER cocite_flat BY cnt > 5;
    GENERATE group, similar;
}
```

As you can see, Pig has simplified the implementation of this data processing application tremendously. This "similar item" feature has been known to be useful in different applications, but it's also quite challenging to implement. Using Pig and Hadoop, this turns into only an afternoon's work. Furthermore, its improved ease of development enables rapid prototyping of alternative features. For your own exercise, instead of finding patents that are often cited together, can you find patents that have similar citations?

10.9 Summary

Pig is a higher-level data processing layer on top of Hadoop. Its Pig Latin language provides programmers a more intuitive way to specify data flows. It supports schemas in processing structured data, yet it's flexible enough to work with unstructured text or semistructured XML data. It's extensible with the use of UDFs. It vastly simplifies data joining and job chaining—two aspects of MapReduce programming that many developers found overly complicated. To demonstrate its usefulness, our example of computing patent cocitation shows a complex MapReduce program written in a dozen lines of Pig Latin.