CHAPTER 1
Introduction

What Is Pig?

Pig provides an engine for executing data flows in parallel on Hadoop. It includes a language, *Pig Latin*, for expressing these data flows. Pig Latin includes operators for many of the traditional data operations (join, sort, filter, etc.), as well as the ability for users to develop their own functions for reading, processing, and writing data.

Pig is an Apache open source project. This means users are free to download it as source or binary, use it for themselves, contribute to it, and—under the terms of the Apache License—use it in their products and change it as they see fit.

Pig on Hadoop

Pig runs on Hadoop. It makes use of both the Hadoop Distributed File System, *HDFS*, and Hadoop's processing system, *MapReduce*.

HDFS is a distributed filesystem that stores files across all of the nodes in a Hadoop cluster. It handles breaking the files into large blocks and distributing them across different machines, including making multiple copies of each block so that if any one machine fails no data is lost. It presents a POSIX-like interface to users. By default, Pig reads input files from HDFS, uses HDFS to store intermediate data between MapReduce jobs, and writes its output to HDFS. As you will see in Chapter 11, it can also read input from and write output to sources other than HDFS.

MapReduce is a simple but powerful parallel data-processing paradigm. Every job in MapReduce consists of three main phases: map, shuffle, and reduce. In the map phase, the application has the opportunity to operate on each record in the input separately. Many maps are started at once so that while the input may be gigabytes or terabytes in size, given enough machines, the map phase can usually be completed in under one minute.

Part of the specification of a MapReduce job is the key on which data will be collected. For example, if you were processing web server logs for a website that required users to log in, you might choose the user ID to be your key so that you could see everything done by each user on your website. In the shuffle phase, which happens after the map phase, data is collected together by the key the user has chosen and distributed to different machines for the reduce phase. Every record for a given key will go to the same reducer.

In the reduce phase, the application is presented each key, together with all of the records containing that key. Again this is done in parallel on many machines. After processing each group, the reducer can write its output. See the next section for a walkthrough of a simple MapReduce program. For more details on how MapReduce works, see "MapReduce" on page 189.

MapReduce's hello world

Consider a simple MapReduce application that counts the number of times each word appears in a given text. This is the "hello world" program of MapReduce. In this example the map phase will read each line in the text, one at a time. It will then split out each word into a separate string, and, for each word, it will output the word and a 1 to indicate it has seen the word one time. The shuffle phase will use the word as the key, hashing the records to reducers. The reduce phase will then sum up the number of times each word was seen and write that together with the word as output. Let's consider the case of the nursery rhyme "Mary Had a Little Lamb." Our input will be:

```
Mary had a little lamb
its fleece was white as snow
and everywhere that Mary went
the lamb was sure to go.
```

Let's assume that each line is sent to a different map task. In reality, each map is assigned much more data than this, but it makes the example easier to follow. The data flow through MapReduce is shown in Figure 1-1.

Once the map phase is complete, the shuffle phase will collect all records with the same word onto the same reducer. For this example we assume that there are two reducers: all words that start with `A-L` are sent to the first reducer, and `M-Z` are sent to the second reducer. The reducers will then output the summed counts for each word.

Pig uses MapReduce to execute all of its data processing. It compiles the Pig Latin scripts that users write into a series of one or more MapReduce jobs that it then executes. See Example 1-1 for a Pig Latin script that will do a word count of "Mary Had a Little Lamb."

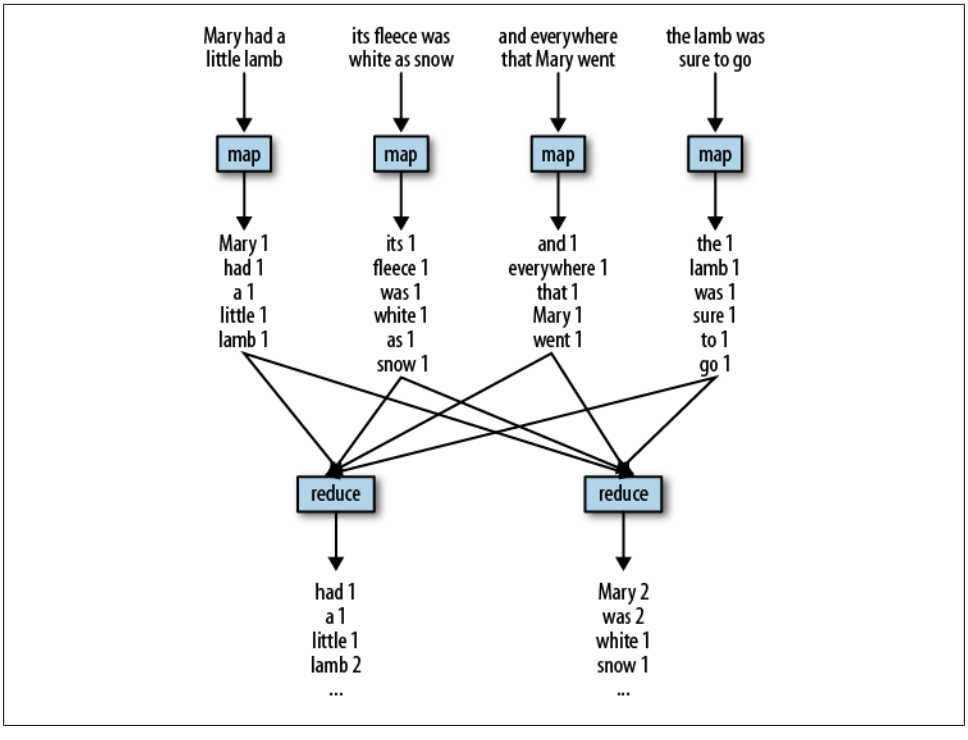


Figure 1-1. MapReduce illustration

Example 1-1. Pig counts Mary and her lamb

```
-- Load input from the file named Mary, and call the single
-- field in the record 'line'.
input = load 'mary' as (line);

-- TOKENIZE splits the line into a field for each word.
-- flatten will take the collection of records returned by
-- TOKENIZE and produce a separate record for each one, calling the single
-- field in the record word.
words = foreach input generate flatten(TOKENIZE(line)) as word;

-- Now group them together by each word.
grpd  = group words by word;

-- Count them.
cntd  = foreach grpd generate group, COUNT(words);
-- Print out the results.
dump cntd;
```

There is no need to be concerned with map, shuffle, and reduce phases when using Pig. It will manage decomposing the operators in your script into the appropriate MapReduce phases.

Pig Latin, a Parallel Dataflow Language

Pig Latin is a dataflow language. This means it allows users to describe how data from one or more inputs should be read, processed, and then stored to one or more outputs in parallel. These data flows can be simple linear flows like the word count example given previously. They can also be complex workflows that include points where multiple inputs are joined, and where data is split into multiple streams to be processed by different operators. To be mathematically precise, a Pig Latin script describes a *directed acyclic graph* (DAG), where the edges are data flows and the nodes are operators that process the data.

This means that Pig Latin looks different from many of the programming languages you have seen. There are no `if` statements or `for` loops in Pig Latin. This is because traditional procedural and object-oriented programming languages describe control flow, and data flow is a side effect of the program. Pig Latin instead focuses on data flow. For information on how to integrate the data flow described by a Pig Latin script with control flow, see Chapter 9.

Comparing query and dataflow languages

After a cursory look, people often say that Pig Latin is a procedural version of SQL. Although there are certainly similarities, there are more differences. SQL is a query language. Its focus is to allow users to form queries. It allows users to describe what question they want answered, but not how they want it answered. In Pig Latin, on the other hand, the user describes exactly how to process the input data.

Another major difference is that SQL is oriented around answering one question. When users want to do several data operations together, they must either write separate queries, storing the intermediate data into temporary tables, or write it in one query using subqueries inside that query to do the earlier steps of the processing. However, many SQL users find subqueries confusing and difficult to form properly. Also, using subqueries creates an inside-out design where the first step in the data pipeline is the innermost query.

Pig, however, is designed with a long series of data operations in mind, so there is no need to write the data pipeline in an inverted set of subqueries or to worry about storing data in temporary tables. This is illustrated in Examples 1-2 and 1-3.

Consider a case where a user wants to group one table on a key and then join it with a second table. Because joins happen before grouping in a SQL query, this must be expressed either as a subquery or as two queries with the results stored in a temporary table. Example 1-3 will use a temporary table, as that is more readable.

Example 1-2. Group then join in SQL

```
CREATE TEMP TABLE t1 AS
SELECT customer, sum(purchase) AS total_purchases
FROM transactions
GROUP BY customer;

SELECT customer, total_purchases, zipcode
FROM t1, customer_profile
WHERE t1.customer = customer_profile.customer;
```

In Pig Latin, on the other hand, this looks like Example 1-3.

Example 1-3. Group then join in Pig Latin

```
-- Load the transactions file, group it by customer, and sum their total purchases
txns    = load 'transactions' as (customer, purchase);
grouped = group txns by customer;
total   = foreach grouped generate group, SUM(txns.purchase) as tp;
-- Load the customer_profile file
profile = load 'customer_profile' as (customer, zipcode);
-- join the grouped and summed transactions and customer_profile data
answer  = join total by group, profile by customer;
-- Write the results to the screen
dump answer;
```

Furthermore, each was designed to live in a different environment. SQL is designed for the RDBMS environment, where data is normalized and schemas and proper constraints are enforced (that is, there are no nulls in places they do not belong, etc.). Pig is designed for the Hadoop data-processing environment, where schemas are sometimes unknown or inconsistent. Data may not be properly constrained, and it is rarely normalized. As a result of these differences, Pig does not require data to be loaded into tables first. It can operate on data as soon as it is copied into HDFS.

An analogy with human languages and cultures might help. My wife and I have been to France together a couple of times. I speak very little French. But because English is the language of commerce (and probably because Americans and the British like to vacation in France), there is enough English spoken in France for me to get by. My wife, on the other hand, speaks French. She has friends there to visit. She can talk to people we meet. She can explore the parts of France that are not on the common tourist itinerary. Her experience of France is much deeper than mine because she can speak the native language.

SQL is the English of data processing. It has the nice feature that everyone and every tool knows it, which means the barrier to adoption is very low. Our goal is to make Pig Latin the native language of parallel data-processing systems such as Hadoop. It may take some learning, but it will allow users to utilize the power of Hadoop much more fully.

How Pig differs from MapReduce

I have just made the claim that a goal of the Pig team is to make Pig Latin the native language of parallel data-processing environments such as Hadoop. But does MapReduce not provide enough? Why is Pig necessary?

Pig provides users with several advantages over using MapReduce directly. Pig Latin provides all of the standard data-processing operations, such as join, filter, group by, order by, union, etc. MapReduce provides the group by operation directly (that is what the shuffle plus reduce phases are), and it provides the order by operation indirectly through the way it implements the grouping. Filter and projection can be implemented trivially in the map phase. But other operators, particularly join, are not provided and must instead be written by the user.

Pig provides some complex, nontrivial implementations of these standard data operations. For example, because the number of records per key in a dataset is rarely evenly distributed, the data sent to the reducers is often skewed. That is, one reducer will get 10 or more times the data than other reducers. Pig has join and order by operators that will handle this case and (in some cases) rebalance the reducers. But these took the Pig team months to write, and rewriting these in MapReduce would be time consuming.

In MapReduce, the data processing inside the map and reduce phases is opaque to the system. This means that MapReduce has no opportunity to optimize or check the user's code. Pig, on the other hand, can analyze a Pig Latin script and understand the data flow that the user is describing. That means it can do early error checking (did the user try to add a string field to an integer field?) and optimizations (can these two grouping operations be combined?).

MapReduce does not have a type system. This is intentional, and it gives users the flexibility to use their own data types and serialization frameworks. But the downside is that this further limits the system's ability to check users' code for errors both before and during runtime.

All of these points mean that Pig Latin is much lower cost to write and maintain than Java code for MapReduce. In one very unscientific experiment, I wrote the same operation in Pig Latin and MapReduce. Given one file with user data and one with click data for a website, the Pig Latin script in Example 1-4 will find the five pages most visited by users between the ages of 18 and 25.

Example 1-4. Finding the top five URLs

```
Users = load 'users' as (name, age);
Fltrd = filter Users by age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd   = join Fltrd by name, Pages by user;
Grpd  = group Jnd by url;
Smmd  = foreach Grpd generate group, COUNT(Jnd) as clicks;
Srtd  = order Smmd by clicks desc;
Top5  = limit Srtd 5;
store Top5 into 'top5sites';
```

The first line of this program loads the file *users* and declares that this data has two fields: `name` and `age`. It assigns the name of `Users` to the input. The second line applies a filter to `Users` that passes through records with an `age` between 18 and 25, inclusive. All other records are discarded. Now the data has only records of users in the age range we are interested in. The results of this filter are named `Fltrd`.

The second `load` statement loads *pages* and names it `Pages`. It declares its schema to have two fields, `user` and `url`.

The line `Jnd = join` joins together `Fltrd` and `Pages` using `Fltrd.name` and `Pages.user` as the key. After this join we have found all the URLs each user has visited.

The line `Grpd = group` collects records together by URL. So for each value of `url`, such as `pignews.com/frontpage`, there will be one record with a collection of all records that have that value in the `url` field. The next line then counts how many records are collected together for each URL. So after this line we now know, for each URL, how many times it was visited by users aged 18–25.

The next thing to do is to sort this from most visits to least. The line `Srtd = order` sorts on the count value from the previous line and places it in `desc` (descending) order. Thus the largest value will be first. Finally, we need only the top five pages, so the last line limits the sorted results to only five records. The results of this are then stored back to HDFS in the file *top5sites*.

In Pig Latin this comes to nine lines of code and took about 15 minutes to write and debug. The same code in MapReduce (omitted here for brevity) came out to about 170 lines of code and took me four hours to get working. The Pig Latin will similarly be easier to maintain, as future developers can easily understand and modify this code.

There is, of course, a cost to all this. It is possible to develop algorithms in MapReduce that cannot be done easily in Pig. And the developer gives up a level of control. A good engineer can always, given enough time, write code that will out perform a generic system. So for less common algorithms or extremely performance-sensitive ones, MapReduce is still the right choice. Basically this is the same situation as choosing to code in Java versus a scripting language such as Python. Java has more power, but due to its lower-level nature, it requires more development time than scripting languages. Developers will need to choose the right tool for each job.

What Is Pig Useful For?

In my experience, Pig Latin use cases tend to fall into three separate categories: traditional extract transform load (ETL) data pipelines, research on raw data, and iterative processing.

The largest use case is data pipelines. A common example is web companies bringing in logs from their web servers, cleansing the data, and precomputing common aggregates before loading it into their data warehouse. In this case, the data is loaded onto

the grid, and then Pig is used to clean out records from bots and records with corrupt data. It is also used to join web event data against user databases so that user cookies can be connected with known user information.

Another example of data pipelines is using Pig offline to build behavior prediction models. Pig is used to scan through all the user interactions with a website and split the users into various segments. Then, for each segment, a mathematical model is produced that predicts how members of that segment will respond to types of advertisements or news articles. In this way the website can show ads that are more likely to get clicked on, or offer news stories that are more likely to engage users and keep them coming back to the site.

Traditionally, ad-hoc queries are done in languages such as SQL that make it easy to quickly form a question for the data to answer. However, for research on raw data, some users prefer Pig Latin. Because Pig can operate in situations where the schema is unknown, incomplete, or inconsistent, and because it can easily manage nested data, researchers who want to work on data before it has been cleaned and loaded into the warehouse often prefer Pig. Researchers who work with large data sets often use scripting languages such as Perl or Python to do their processing. Users with these backgrounds often prefer the dataflow paradigm of Pig over the declarative query paradigm of SQL.

Users building iterative processing models are also starting to use Pig. Consider a news website that keeps a graph of all news stories on the Web that it is tracking. In this graph each news story is a node, and edges indicate relationships between the stories. For example, all stories about an upcoming election are linked together. Every five minutes a new set of stories comes in, and the data-processing engine must integrate them into the graph. Some of these stories are new, some are updates of existing stories, and some supersede existing stories. Some data-processing steps need to operate on this entire graph of stories. For example, a process that builds a behavioral targeting model needs to join user data against this entire graph of stories. Rerunning the entire join every five minutes is not feasible because it cannot be completed in five minutes with a reasonable amount of hardware. But the model builders do not want to update these models only on a daily basis, as that means an entire day of missed serving opportunities.

To cope with this problem, it is possible to first do a join against the entire graph on a regular basis, for example, daily. Then, as new data comes in every five minutes, a join can be done with just the new incoming data, and these results can be combined with the results of the join against the whole graph. This combination step takes some care, as the five-minute data contains the equivalent of inserts, updates, and deletes on the entire graph. It is possible and reasonably convenient to express this combination in Pig Latin.

One point that is implicit in everything I have said so far is that Pig (like MapReduce) is oriented around the batch processing of data. If you need to process gigabytes or terabytes of data, Pig is a good choice. But it expects to read all the records of a file and write all of its output sequentially. For workloads that require writing single or small groups of records, or looking up many different records in random order, Pig (like MapReduce) is not a good choice. See "NoSQL Databases" on page 166 for a discussion of applications that are good for these use cases.

Pig Philosophy

Early on, people who came to the Pig project as potential contributors did not always understand what the project was about. They were not sure how to best contribute or which contributions would be accepted and which would not. So, the Pig team produced a statement of the project's philosophy that summarizes what Pig aspires to be:

Pigs eat anything
: Pig can operate on data whether it has metadata or not. It can operate on data that is relational, nested, or unstructured. And it can easily be extended to operate on data beyond files, including key/value stores, databases, etc.

Pigs live anywhere
: Pig is intended to be a language for parallel data processing. It is not tied to one particular parallel framework. It has been implemented first on Hadoop, but we do not intend that to be only on Hadoop.

Pigs are domestic animals
: Pig is designed to be easily controlled and modified by its users.

  Pig allows integration of user code wherever possible, so it currently supports user defined field transformation functions, user defined aggregates, and user defined conditionals. These functions can be written in Java or in scripting languages that can compile down to Java (e.g., Jython). Pig supports user provided load and store functions. It supports external executables via its `stream` command and MapReduce JARs via its `mapreduce` command. It allows users to provide a custom partitioner for their jobs in some circumstances, and to set the level of reduce parallelism for their jobs.

  Pig has an optimizer that rearranges some operations in Pig Latin scripts to give better performance, combines MapReduce jobs together, etc. However, users can easily turn this optimizer off to prevent it from making changes that do not make sense in their situation.

Pigs fly
: Pig processes data quickly. We want to consistently improve performance, and not implement features in ways that weigh Pig down so it can't fly.

Pig's History

Pig started out as a research project in Yahoo! Research, where Yahoo! scientists designed it and produced an initial implementation. As explained in a paper presented at SIGMOD in 2008,* the researchers felt that the MapReduce paradigm presented by Hadoop "is too low-level and rigid, and leads to a great deal of custom user code that is hard to maintain and reuse." At the same time they observed that many MapReduce users were not comfortable with declarative languages such as SQL. Thus they set out to produce "a new language called Pig Latin that we have designed to fit in a sweet spot between the declarative style of SQL, and the low-level, procedural style of MapReduce."

Yahoo! Hadoop users started to adopt Pig. So, a team of development engineers was assembled to take the research prototype and build it into a production-quality product. About this same time, in fall 2007, Pig was open sourced via the Apache Incubator. The first Pig release came a year later in September 2008. Later that same year, Pig graduated from the Incubator and became a subproject of Apache Hadoop.

Early in 2009 other companies started to use Pig for their data processing. Amazon also added Pig as part of its Elastic MapReduce service. By the end of 2009 about half of Hadoop jobs at Yahoo! were Pig jobs. In 2010, Pig adoption continued to grow, and Pig graduated from a Hadoop subproject, becoming its own top-level Apache project.

Why Is It Called Pig?

One question that is frequently asked is, "Why is it named Pig?" People also want to know whether Pig is an acronym. It is not. The story goes that the researchers working on the project initially referred to it simply as "the language." Eventually they needed to call it something. Off the top of his head, one researcher suggested Pig, and the name stuck. It is quirky yet memorable and easy to spell. While some have hinted that the name sounds coy or silly, it has provided us with an entertaining nomenclature, such as Pig Latin for a language, Grunt for a shell, and Piggybank for a CPAN-like shared repository.

* Christopher Olston et al, "Pig Latin: A Not-So-Foreign Language for Data Processing," available at *http://portal.acm.org/citation.cfm?id=1376726*.

CHAPTER 2

Installing and Running Pig

Downloading and Installing Pig

Before you can run Pig on your machine or your Hadoop cluster, you will need to download and install it. If someone else has taken care of this, you can skip ahead to "Running Pig" on page 13.

You can download Pig as a complete package or as source code that you build. You can also get it as part of a Hadoop distribution.

Downloading the Pig Package from Apache

This is the official version of Apache Pig. It comes packaged with all of the JAR files needed to run Pig. It can be downloaded by going to Pig's release page.

Pig does not need to be installed on your Hadoop cluster. It runs on the machine from which you launch Hadoop jobs. Though you can run Pig from your laptop or desktop, in practice, most cluster owners set up one or more machines that have access to their Hadoop cluster but are not part of the cluster (that is, they are not data nodes or task nodes). This makes it easier for administrators to update Pig and associated tools, as well as to secure access to the clusters. These machines are called *gateway machines* or *edge machines*. In this book I use the term gateway machine.

You will need to install Pig on these gateway machines. If your Hadoop cluster is accessible from your desktop or laptop, you can install Pig there as well. Also, you can install Pig on your local machine if you plan to use Pig in local mode.

The core of Pig is written in Java and is thus portable across operating systems. The shell script that starts Pig is a bash script, so it requires a Unix environment. Hadoop, which Pig depends on, even in local mode, also requires a Unix environment for its filesystem operations. In practice, most Hadoop clusters run a flavor of Linux. Many Pig developers develop and test Pig on Mac OS X.

Pig requires Java 1.6, and Pig versions 0.5 through 0.9 require Hadoop 0.20. For future versions, check the download page for information on what version(s) of Hadoop they require. The correct version of Hadoop is included with the Pig download. If you plan to use Pig in local mode or install it on a gateway machine where Hadoop is not currently installed, there is no need to download Hadoop separately.

Once you have downloaded Pig, you can place it anywhere you like on your machine, as it does not depend on being in a certain location. To install it, place the tarball in the directory of your choosing and type:

```
tar xzf filename
```

where *filename* is the TAR file you downloaded.

The only other setup in preparation for running Pig is making sure that the environment variable `JAVA_HOME` is set to the directory that contains your Java distribution. Pig will fail immediately if this value is not in the environment. You can set this in your shell, specify it on the command line when you invoke Pig, or set it explicitly in your copy of the Pig script `pig`, located in the *bin* directory that you just unpacked. You can find the appropriate value for `JAVA_HOME` by executing `which java` and stripping the `bin/java` from the end of the result.

Downloading Pig from Cloudera

In addition to the official Apache version, there are companies that repackage and distribute Hadoop and associated tools. Currently the most popular of these is Cloudera, which produces RPMs for Red Hat–based systems and packages for use with APT on Debian systems. It also provides tarballs for other systems that cannot use one of these package managers.

The upside of using a distribution like Cloudera's is that all of the tools are packaged and tested together. Also, if you need professional support, it is available. The downside is that you are constrained to move at the speed of your distribution provider. There is a delay between an Apache release of Pig and its availability in various distributions.

For complete instructions on downloading and installing Hadoop and Pig from Cloudera, see Cloudera's download site. Note that you have to download Pig separately; it is not part of the Hadoop package.

Downloading Pig Artifacts from Maven

In addition to the official release available from Pig's Apache site, it is possible to download Pig from Apache's Maven repository. This site includes JAR files for Pig, for the source code, and for the Javadocs, as well as the POM file that defines Pig's dependencies. Development tools that are Maven-aware can use this to pull down Pig's source and Javadoc. If you use `maven` or `ant` in your build process, you can also pull the Pig JAR from this repository automatically.

Downloading the Source

When you download Pig from Apache, you also get the Pig source code. This enables you to debug your version of Pig or just peruse the code to see how it works. But if you want to live on the edge and try out a feature or a bug fix before it is available in a release, you can download the source from Apache's Subversion repository. You can also apply patches that have been uploaded to Pig's issue-tracking system but that are not yet checked into the code repository. Information on checking out Pig using `svn` or cloning the repository via `git` is available on Pig's version control page.

Running Pig

You can run Pig locally on your machine or on your grid. You can also run Pig as part of Amazon's Elastic MapReduce service.

Running Pig Locally on Your Machine

Running Pig locally on your machine is referred to in Pig parlance as *local mode*. Local mode is useful for prototyping and debugging your Pig Latin scripts. Some people also use it for small data when they want to apply the same processing to large data—so that their data pipeline is consistent across data of different sizes—but they do not want to waste cluster resources on small files and small jobs.

In versions 0.6 and earlier, Pig executed scripts in local mode itself. Starting with version 0.7, it uses the Hadoop class `LocalJobRunner` that reads from the local filesystem and executes MapReduce jobs locally. This has the nice property that Pig jobs run locally in the same way as they will on your cluster, and they all run in one process, making debugging much easier. The downside is that it is slow. Setting up a local instance of Hadoop has approximately a 20-second overhead, so even tiny jobs take at least that long.*

Let's run a Pig Latin script in local mode. See "Code Examples in This Book" on page xi for how to download the data and Pig Latin for this example. The simple script in Example 2-1 loads the file *NYSE_dividends*, groups the file's rows by stock ticker symbol, and then calculates the average dividend for each symbol.

* Another reason for switching to MapReduce for local mode was that as Pig added features that took advantage of more advanced MapReduce features, it became difficult or impossible to replicate those features in local mode. Thus local mode and MapReduce mode were diverging in their feature set.

Example 2-1. Running Pig in local mode

```
--average_dividend.pig
-- load data from NYSE_dividends, declaring the schema to have 4 fields
dividends = load 'NYSE_dividends' as (exchange, symbol, date, dividend);
-- group rows together by stock ticker symbol
grouped   = group dividends by symbol;
-- calculate the average dividend per symbol
avg       = foreach grouped generate group, AVG(dividends.dividend);
-- store the results to average_dividend
store avg into 'average_dividend';
```

If you use `head -5` to look at the *NYSE_dividends* file, you will see:

```
NYSE    CPO 2009-12-30  0.14
NYSE    CPO 2009-09-28  0.14
NYSE    CPO 2009-06-26  0.14
NYSE    CPO 2009-03-27  0.14
NYSE    CPO 2009-01-06  0.14
```

This matches the schema we declared in our Pig Latin script. The first field is the exchange this stock is traded on, the second field is the stock ticker symbol, the third is the date the dividend was paid, and the fourth is the amount of the dividend.

Remember that to run Pig you will need to set the `JAVA_HOME` environment variable to the directory that contains your Java distribution.

Switch to the directory where *NYSE_dividends* is located. You can then run this example on your local machine by entering:

```
pig_path/bin/pig -x local average_dividend.pig
```

where ***pig_path*** is the path to the Pig installation on your local machine.

The result should be a lot of output on your screen. Much of this is MapReduce's `LocalJobRunner` generating logs. But some of it is Pig telling you how it will execute the script, giving you the status as it executes, etc. Near the bottom of the output you should see the simple message `Success!`. This means all went well. The script stores its output to *average_dividend*, so you might expect to find a file by that name in your local directory. Instead you will find a directory named *average_dividend* that contains a file named *part-r-00000*. Because Hadoop is a distributed system and usually processes data in parallel, when it outputs data to a "file" it creates a directory with the file's name, and each writer creates a separate *part file* in that directory. In this case we had one writer, so we have one part file. We can look in that part file for the results by entering:

```
cat average_dividend/part-r-00000 | head -5
```

which returns:

```
CA      0.04
CB      0.35
CE      0.04
CF      0.1
CI      0.04
```

Running Pig on Your Hadoop Cluster

Most of the time you will be running Pig on your Hadoop cluster. As was covered in "Downloading and Installing Pig" on page 11, Pig runs locally on your machine or your gateway machine. All of the parsing, checking, and planning is done locally. Pig then executes MapReduce jobs in your cluster.

When I say "your gateway machine," I mean the machine from which you are launching Pig jobs. Usually this will be one or more machines that have access to your Hadoop cluster. However, depending on your configuration, it could be your local machine as well.

The only thing Pig needs to know to run on your cluster is the location of your cluster's *NameNode* and *JobTracker*. The NameNode is the manager of HDFS, and the JobTracker coordinates MapReduce jobs. In Hadoop 0.18 and earlier, these locations are found in your *hadoop-site.xml* file. In Hadoop 0.20 and later, they are in three separate files: *core-site.xml*, *hdfs-site.xml*, and *mapred-site.xml*.

If you are already running Hadoop jobs from your gateway machine via MapReduce or another tool, you most likely have these files present. If not, the best course is to copy these files from nodes in your cluster to a location on your gateway machine. This guarantees that you get the proper addresses plus any site-specific settings.

If, for whatever reason, it is not possible to copy the appropriate files from your cluster, you can create a *hadoop-site.xml* file yourself. It will look like the following:

```
<configuration>
<property>
  <name>fs.default.name</name>
  <value>namenode_hostname:port</value>
</property>

<property>
  <name>mapred.job.tracker</name>
  <value>jobtrack_hostname:port</value>
</property>
</configuration>
```

You will need to find the names and ports for your NameNode and JobTracker from your cluster administrator.

Once you have located, copied, or created these files, you will need to tell Pig the directory they are in by setting the `PIG_CLASSPATH` environment variable to that directory. Note that this must point to the *directory* that the XML file is in, not the file itself. Pig will read all XML and properties files in that directory.

Let's run the same script on your cluster that we ran in the local mode example (Example 2-1). If you are running on a Hadoop cluster you have never used before, you will most likely need to create a home directory. Pig can do this for you:

```
PIG_CLASSPATH=hadoop_conf_dir pig_path/bin/pig -e fs -mkdir /user/username
```

where *`hadoop_conf_dir`* is the directory where your *hadoop-site.xml* or *core-site.xml*, *hdfs-site.xml*, and *mapred-site.xml* files are located; *`pig_path`* is the path to Pig on your gateway machine; and *`username`* is your username on the gateway machine. If you are using 0.5 or earlier, change `fs -mkdir` to `mkdir`.

Remember, you need to set `JAVA_HOME` before executing any Pig commands. See "Downloading the Pig Package from Apache" on page 11 for details.

In order to run this example on your cluster, you first need to copy the data to your cluster:

```
PIG_CLASSPATH=hadoop_conf_dir pig_path/bin/pig -e fs -copyFromLocal NYSE_dividends
    NYSE_dividends
```

If you are running Pig 0.5 or earlier, change `fs -copyFromLocal` to `copyFromLocal`.

Now you are ready to run the Pig Latin script itself:

```
PIG_CLASSPATH=hadoop_conf_dir pig_path/bin/pig average_dividend.pig
```

The first few lines of output will tell you how Pig is connecting to your cluster. After that it will describe its progress in executing your script. It is important for you to verify that Pig is connecting to the appropriate filesystem and JobTracker by checking that these values match the values for your cluster. If the filesystem is listed as *file:///* or the JobTracker says `localhost`, Pig did not connect to your cluster. You will need to check that you entered the values properly in your configuration files and properly set `PIG_CLASSPATH` to the directory that contains those files.

Near the end of the output there should be a line saying `Success!`. This means that your execution succeeded. You can see the results by entering:

```
PIG_CLASSPATH=hadoop_conf_dir pig_path/bin/pig -e cat average_dividend
```

which should give you the same connection information and then dump all of the stock ticker symbols and their average dividends.

In Example 2-1 you may have noticed that I made a point to say that *average_dividend* is a directory, and thus you have to `cat` the part file contained in that directory.

However, in this example I ran `cat` directly on *average_dividend*. If you list *average_dividend*, you will see that it is still a directory in this example, but in Pig, `cat` can operate on directories. See Chapter 3 for a discussion of this.

Running Pig in the Cloud

Cloud computing† along with the *software as a service* (SaaS) model have taken off in recent years. This has been fortuitous for hardware-intensive applications such as Hadoop. Setting up and maintaining a Hadoop cluster is an expensive proposition in terms of hardware acquisition, facility costs, and maintenance and administration. Many users find that it is cheaper to rent the hardware they need instead.

Whether you or your organization decides to use Hadoop and Pig in the cloud or on owned and operated machines, the instructions for running Pig on your cluster are the same; see "Running Pig on Your Hadoop Cluster" on page 15.

However, Amazon's *Elastic MapReduce* (EMR) cloud offering is different. Rather than allowing customers to rent machines for any type of process (like Amazon's Elastic Cloud Computing [EC2] service and other cloud services), EMR allows users to rent virtual Hadoop clusters. These clusters read data from and write data to Amazon's Simple Storage Service (S3). This means users do not even need to set up their own Hadoop cluster, which they would have to do if they used EC2 or a similar service.

EMR users can access their rented Hadoop cluster via their browser, SSH, or a web services API. For information about EMR, visit *http://aws.amazon.com/elasticmapreduce*. However, I suggest beginning with this nice tutorial, which will introduce you to the service.

Command-Line and Configuration Options

Pig has a number of command-line options that you can use with it. You can see the full list by entering `pig -h`. Most of these options will be discussed later, in the sections that cover the features these options control. In this section I discuss the remaining miscellaneous options:

`-e` *or* `-execute`
: Execute a single command in Pig. For example, `pig -e fs -ls` will list your home directory.

`-h` *or* `-help`
: List the available command-line options.

† Being the current flavor of the month, the term cloud computing is being used to describe just about anything that takes more than one computer and is not located on a person's desktop. In this chapter I use cloud computing to mean the ability to rent a cluster of computers and place software of your choosing on those computers.

`-h properties`
: List the properties that Pig will use if they are set by the user.

`-P` *or* `-propertyFile`
: Specify a property file that Pig should read.

`-version`
: Print the version of Pig.

Pig also uses a number of Java properties. The entire list can be printed out with `pig -h properties`. Specific properties are discussed later in sections that cover the features they control.

Hadoop also has a number of Java properties it uses to determine its behavior. For example, you can pass options to the JVM that runs your map and reduce tasks by setting `mapred.child.java.opts`. In Pig version 0.8 and later, these can be passed to Pig, and then Pig will pass them on to Hadoop when it invokes Hadoop. In earlier versions, the properties had to be in *hadoop-site.xml* so that the Hadoop client itself would pick them up.

Properties can be passed to Pig on the command line using `-D` in the same format as any Java property—for example, `bin/pig -D exectype=local`. When placed on the command line, these property definitions must come before any Pig-specific command-line options (such as `-x local`). They can also be specified in the *conf/pig.properties* file that is part of your Pig distribution. Finally, you can specify a separate properties file by using `-P`. If properties are specified on both the command line and in a properties file, the command-line specification takes precedence.

Return Codes

Pig uses return codes, described in Table 2-1, to communicate success or failure.

Table 2-1. Pig return codes

Value	Meaning	Comments
0	Success	
1	Retriable failure	
2	Failure	
3	Partial failure	Used with multiquery; see "Nonlinear Data Flows" on page 72
4	Illegal arguments passed to Pig	
5	`IOException` thrown	Would usually be thrown by a UDF
6	`PigException` thrown	Usually means a Python UDF raised an exception
7	`ParseException` thrown (can happen after parsing if variable substitution is being done)	
8	`Throwable` thrown (an unexpected exception)	

CHAPTER 3
Grunt

Grunt[*] is Pig's interactive shell. It enables users to enter Pig Latin interactively and provides a shell for users to interact with HDFS.

To enter Grunt, invoke Pig with no script or command to run. Typing:

```
pig -x local
```

will result in the prompt:

```
grunt>
```

This gives you a Grunt shell to interact with your local filesystem. If you omit the `-x local` and have a cluster configuration set in `PIG_CLASSPATH`, this will put you in a Grunt shell that will interact with HDFS on your cluster.

As you would expect with a shell, Grunt provides command-line history and editing, as well as Tab completion. It does not provide filename completion via the Tab key. That is, if you type `kil` and then press the Tab key, it will complete the command as `kill`. But if you have a file *foo* in your local directory and type `ls fo`, and then hit Tab, it will not complete it as `ls foo`. This is because the response time from HDFS to connect and find whether the file exists is too slow to be useful.

Although Grunt is a useful shell, remember that it is not a full-featured shell. It does not provide a number of commands found in standard Unix shells, such as pipes, redirection, and background execution.

To exit Grunt you can type `quit` or enter Ctrl-D.

[*] According to Ben Reed, one of the researchers at Yahoo! who helped start Pig, they named the shell "Grunt" because they felt the initial implementation was so limited that it was not worthy even of the name "oink."

Entering Pig Latin Scripts in Grunt

One of the main uses of Grunt is to enter Pig Latin in an interactive session. This can be particularly useful for quickly sampling your data and for prototyping new Pig Latin scripts.

You can enter Pig Latin directly into Grunt. Pig will not start executing the Pig Latin you enter until it sees either a `store` or `dump`. However, it will do basic syntax and semantic checking to help you catch errors quickly. If you do make a mistake while entering a line of Pig Latin in Grunt, you can reenter the line using the same alias, and Pig will take the last instance of the line you enter. For example:

```
pig  -x local
grunt> dividends = load 'NYSE_dividends' as (exchange, symbol, date, dividend);
grunt> symbols = foreach dividends generate symbl;
...Error during parsing. Invalid alias: symbl ...
grunt> symbols = foreach A generate symbol;
...
```

HDFS Commands in Grunt

Besides entering Pig Latin interactively, Grunt's other major use is to act as a shell for HDFS. In versions 0.5 and later of Pig, all `hadoop fs` shell commands are available. They are accessed using the keyword `fs`. The dash (`-`) used in the `hadoop fs` is also required:

```
grunt>fs -ls
```

You can see a complete guide to the available commands at *http://hadoop.apache.org/common/docs/r0.20.2/hdfs_shell.html*. A number of the commands come directly from Unix shells and will operate in ways that are familiar: `chgrp`, `chmod`, `chown`, `cp`, `du`, `ls`, `mkdir`, `mv`, `rm`, and `stat`. A few of them either look like Unix commands you are used to but behave slightly differently or are unfamiliar, including:

`cat` *`filename`*
: Print the contents of a file to `stdout`. You can apply this command to a directory and it will apply itself in turn to each file in the directory.

`copyFromLocal` *`localfile hdfsfile`*
: Copy a file from your local disk to HDFS. This is done serially, not in parallel.

`copyToLocal` *`hdfsfile localfile`*
: Copy a file from HDFS to your local disk. This is done serially, not in parallel.

`rmr` *`filename`*
: Remove files recursively. This is equivalent to `rm -r` in Unix. Use this with caution.

In versions of Pig before 0.5, `hadoop fs` commands were not available. Instead, Grunt had its own implementation of some of these commands: `cat`, `cd`, `copyFromLocal`, `copyToLocal`, `cp`, `ls`, `mkdir`, `mv`, `pwd`, `rm` (which acted like Hadoop's `rmr`, not Hadoop's `rm`), and `rmf`. As of Pig 0.8, all of these commands are still available. However, with the

exception of `cd` and `pwd`, these commands are deprecated in favor of using `hadoop fs`, and they might be removed at some point in the future.

In version 0.8, a new command was added to Grunt: `sh`. This command gives you access to the local shell, just as `fs` gives you access to HDFS. Simple shell commands that do not involve pipes or redirects can be executed. It is better to work with absolute paths, as `sh` does not always properly track the current working directory.

Controlling Pig from Grunt

Grunt also provides commands for controlling Pig and MapReduce:

`kill` *`jobid`*
: Kill the MapReduce job associated with *`jobid`*. The output of the `pig` command that spawned the job will list the ID of each job it spawns. You can also find the job's ID by looking at Hadoop's JobTracker GUI, which lists all jobs currently running on the cluster. Note that this command kills a particular MapReduce job. If your Pig job contains other MapReduce jobs that do not depend on the killed MapReduce job, these jobs will still continue. If you want to kill all of the MapReduce jobs associated with a particular Pig job, it is best to terminate the process running Pig, and then use this command to kill any MapReduce jobs that are still running. Make sure to terminate the Pig process with a Ctrl-C or a Unix `kill`, not a Unix `kill -9`. The latter does not give Pig the chance to clean up temporary files it is using, which can leave garbage in your cluster.

`exec [-param` *`param_name = param_value`*`] [-param_file` *`filename`*`] ` *`script`*
: Execute the Pig Latin script *`script`*. Aliases defined in *`script`* are not imported into Grunt. This command is useful for testing your Pig Latin scripts while inside a Grunt session. For information on the `-param` and `-param_file` options, see "Parameter Substitution" on page 77.

`run [-param` *`param_name = param_value`*`] [-param_file` *`filename`*`] ` *`script`*
: Execute the Pig Latin script *`script`* in the current Grunt shell. Thus all aliases referenced in *`script`* are available to Grunt, and the commands in *`script`* are accessible via the shell history. This is another option for testing Pig Latin scripts while inside a Grunt session. For information on the `-param` and `-param_file` options, see "Parameter Substitution" on page 77.

CHAPTER 4
Pig's Data Model

Before we take a look at the operators that Pig Latin provides, we first need to understand Pig's data model. This includes Pig's data types, how it handles concepts such as missing data, and how you can describe your data to Pig.

Types

Pig's data types can be divided into two categories: *scalar* types, which contain a single value, and *complex* types, which contain other types.

Scalar Types

Pig's scalar types are simple types that appear in most programming languages. With the exception of bytearray, they are all represented in Pig interfaces by `java.lang` classes, making them easy to work with in UDFs:

int
: An integer. Ints are represented in interfaces by `java.lang.Integer`. They store a four-byte signed integer. Constant integers are expressed as integer numbers, for example, `42`.

long
: A long integer. Longs are represented in interfaces by `java.lang.Long`. They store an eight-byte signed integer. Constant longs are expressed as integer numbers with an `L` appended, for example, `5000000000L`.

float
: A floating-point number. Floats are represented in interfaces by `java.lang.Float` and use four bytes to store their value. You can find the range of values representable by Java's `Float` type at *http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2.3*. Note that because this is a floating-point number, in some calculations it will lose precision. For calculations that require no loss of precision, you should use an int or long instead. Constant floats are expressed as a

floating-point number with an `f` appended. Floating-point numbers can be expressed in simple format, `3.14f`, or in exponent format, `6.022e23f`.

double
: A double-precision floating-point number. Doubles are represented in interfaces by `java.lang.Double` and use eight bytes to store their value. You can find the range of values representable by Java's `Double` type at *http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2.3*. Note that because this is a floating-point number, in some calculations it will lose precision. For calculations that require no loss of precision, you should use an int or long instead. Constant doubles are expressed as a floating-point number in either simple format, `2.71828`, or in exponent format, `6.626e-34`.

chararray
: A string or character array. Chararrays are represented in interfaces by `java.lang.String`. Constant chararrays are expressed as string literals with single quotes, for example, `'fred'`. In addition to standard alphanumeric and symbolic characters, you can express certain characters in chararrays by using backslash codes, such as `\t` for Tab and `\n` for Return. Unicode characters can be expressed as `\u` followed by their four-digit hexadecimal Unicode value. For example, the value for Ctrl-A is expressed as `\u0001`.

bytearray
: A blob or array of bytes. Bytearrays are represented in interfaces by a Java class `DataByteArray` that wraps a Java `byte[]`. There is no way to specify a constant bytearray.

Complex Types

Pig has three complex data types: maps, tuples, and bags. All of these types can contain data of any type, including other complex types. So it is possible to have a map where the value field is a bag, which contains a tuple where one of the fields is a map.

Map

A *map* in Pig is a chararray to data element mapping, where that element can be any Pig type, including a complex type. The chararray is called a key and is used as an index to find the element, referred to as the value.

Because Pig does not know the type of the value, it will assume it is a bytearray. However, the actual value might be something different. If you know what the actual type is (or what you want it to be), you can cast it; see "Casts" on page 30. If you do not cast the value, Pig will make a best guess based on how you use the value in your script. If the value is of a type other than bytearray, Pig will figure that out at runtime and handle it. See "Schemas" on page 27 for more information on how Pig handles unknown types.

By default there is no requirement that all values in a map must be of the same type. It is legitimate to have a map with two keys `name` and `age`, where the value for `name` is a chararray and the value for `age` is an int. Beginning in Pig 0.9, a map can declare its values to all be of the same type. This is useful if you know all values in the map will be of the same type, as it allows you to avoid the casting, and Pig can avoid the runtime type-massaging referenced in the previous paragraph.

Map constants are formed using brackets to delimit the map, a hash between keys and values, and a comma between key-value pairs. For example, `['name'#'bob', 'age'#55]` will create a map with two keys, "name" and "age". The first value is a chararray, and the second is an integer.

Tuple

A *tuple* is a fixed-length, ordered collection of Pig data elements. Tuples are divided into *fields*, with each field containing one data element. These elements can be of any type—they do not all need to be the same type. A tuple is analogous to a row in SQL, with the fields being SQL columns. Because tuples are ordered, it is possible to refer to the fields by position; see "Expressions in foreach" on page 37 for details. A tuple can, but is not required to, have a schema associated with it that describes each field's type and provides a name for each field. This allows Pig to check that the data in the tuple is what the user expects, and it allows the user to reference the fields of the tuple by name.

Tuple constants use parentheses to indicate the tuple and commas to delimit fields in the tuple. For example, `('bob', 55)` describes a tuple constant with two fields.

Bag

A *bag* is an unordered collection of tuples. Because it has no order, it is not possible to reference tuples in a bag by position. Like tuples, a bag can, but is not required to, have a schema associated with it. In the case of a bag, the schema describes all tuples within the bag.

Bag constants are constructed using braces, with tuples in the bag separated by commas. For example, `{('bob', 55), ('sally', 52), ('john', 25)}` constructs a bag with three tuples, each with two fields.

Pig users often notice that Pig does not provide a list or set type that can store items of any type. It is possible to mimic a set type using the bag, by wrapping the desired type in a tuple of one field. For instance, if you want to store a set of integers, you can create a bag with a tuple with one field, which is an int. This is a bit cumbersome, but it works.

Bag is the one type in Pig that is not required to fit into memory. As you will see later, because bags are used to store collections when grouping, bags can become quite large. Pig has the ability to spill bags to disk when necessary, keeping only partial sections of the bag in memory. The size of the bag is limited to the amount of local disk available for spilling the bag.

Memory Requirements of Pig Data Types

In the previous sections I often referenced the size of the value stored for each type (four bytes for integer, eight bytes for long, etc.). This tells you how large (or small) a value those types can hold. However, this does not tell you how much memory is actually used by objects of those types. Because Pig uses Java objects to represent these values internally, there is an additional overhead. This overhead depends on your JVM, but it is usually eight bytes per object. It is even worse for chararrays because Java's `String` uses two bytes per character rather than one.

So, if you are trying to figure out how much memory you need in Pig to hold all of your data (e.g., if you are going to do a join that needs to hold a hash table in memory), do not count the bytes on disk and assume that is how much memory you need. The multiplication factor between disk and memory is dependent on your data, whether your data is compressed on disk, your disk storage format, etc. As a rule of thumb, it takes about four times as much memory as it does disk to represent the uncompressed data.

Nulls

Pig includes the concept of a data element being null. Data of any type can be null. It is important to understand that in Pig the concept of null is the same as in SQL, which is completely different from the concept of null in C, Java, Python, etc. In Pig a null data element means the value is unknown. This might be because the data is missing, an error occurred in processing it, etc. In most procedural languages, a data value is said to be null when it is unset or does not point to a valid address or object. This difference in the concept of null is important and affects the way Pig treats null data, especially when operating on it. See "foreach" on page 37, "Group" on page 41, and "Join" on page 45 for details of how nulls are handled in expressions and relations in Pig.

Unlike SQL, Pig does not have a notion of constraints on the data. In the context of nulls, this means that any data element can always be null. As you write Pig Latin scripts and UDFs, you will need to keep this in mind.

Schemas

Pig has a very lax attitude when it comes to schemas. This is a consequence of Pig's philosophy of eating anything. If a schema for the data is available, Pig will make use of it, both for up-front error checking and for optimization. But if no schema is available, Pig will still process the data, making the best guesses it can based on how the script treats the data. First, we will look at ways that you can communicate the schema to Pig; then, we will examine how Pig handles the case where you do not provide it with the schema.

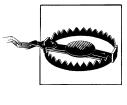
The easiest way to communicate the schema of your data to Pig is to explicitly tell Pig what it is when you load the data:

```
dividends = load 'NYSE_dividends' as
    (exchange:chararray, symbol:chararray, date:chararray, dividend:float);
```

Pig now expects your data to have four fields. If it has more, it will truncate the extra ones. If it has less, it will pad the end of the record with nulls.

It is also possible to specify the schema without giving explicit data types. In this case, the data type is assumed to be bytearray:

```
dividends = load 'NYSE_dividends' as (exchange, symbol, date, dividend);
```



You would expect that this also would force your data into a tuple with four fields, regardless of the number of actual input fields, just like when you specify both names and types for the fields. And in Pig 0.9 this is what happens. But in 0.8 and earlier versions it does not; no truncation or null padding is done in the case where you do not provide explicit types for the fields.

Also, when you declare a schema, you do not have to declare the schema of complex types, but you can if you want to. For example, if your data has a tuple in it, you can declare that field to be a tuple without specifying the fields it contains. You can also declare that field to be a tuple that has three columns, all of which are integers. Table 4-1 gives the details of how to specify each data type inside a schema declaration.

Table 4-1. Schema syntax

Data type	Syntax	Example
int	`int`	`as (a:int)`
long	`long`	`as (a:long)`
float	`float`	`as (a:float)`
double	`double`	`as (a:double)`
chararray	`chararray`	`as (a:chararray)`
bytearray	`bytearray`	`as (a:bytearray)`
map	`map[]` or `map[`*`type`*`]`, where *`type`* is any valid type. This declares all values in the map to be of this type.	`as (a:map[], b:map[int])`
tuple	`tuple()` or `tuple(`*`list_of_fields`*`)`, where *`list_of_fields`* is a comma-separated list of field declarations.	`as (a:tuple(), b:tuple(x:int, y:int))`
bag	`bag{}` or `bag{t:(`*`list_of_fields`*`)}`, where *`list_of_fields`* is a comma-separated list of field declarations. Note that, oddly enough, the tuple inside the bag must have a name, here specified as `t`, even though you will never be able to access that tuple `t` directly.	`(a:bag{}, b:bag{t:(x:int, y:int)})`

The runtime declaration of schemas is very nice. It makes it easy for users to operate on data without having to first load it into a metadata system. It also means that if you are interested in only the first few fields, you only have to declare those fields.

But for production systems that run over the same data every hour or every day, it has a couple of significant drawbacks. One, whenever your data changes, you have to change your Pig Latin. Two, although this works fine on data with 5 columns, it is painful when your data has 100 columns. To address these issues, there is another way to load schemas in Pig.

If the load function you are using already knows the schema of the data, the function can communicate that to Pig. (Load functions are how Pig reads data; see "Load" on page 34 for details.) Load functions might already know the schema because it is stored in a metadata repository such as HCatalog, or it might be stored in the data itself (if, for example, the data is stored in JSON format). In this case, you do not have to declare the schema as part of the `load` statement. And you can still refer to fields by name because Pig will fetch the schema from the load function before doing error checking on your script:

```
mdata = load 'mydata' using HCatLoader();
cleansed = filter mdata by name is not null;
...
```

But what happens when you cross the streams? What if you specify a schema and the loader returns one? If they are identical, all is well. If they are not identical, Pig will determine whether it can adapt the one returned by the loader to match the one you gave. For example, if you specified a field as a long and the loader said it was an int, Pig can and will do that cast. However, if it cannot determine a way to make the loader's schema fit the one you gave, it will give an error. See "Casts" on page 30 for a list of casts Pig can and cannot insert to make the schemas work together.

Now let's look at the case where neither you nor the load function tell Pig what the data's schema is. In addition to being referenced by name, fields can be referenced by position, starting from zero. The syntax is a dollar sign, then the position: `$0` refers to the first field. So it is easy to tell Pig which field you want to work with. But how does Pig know the data type? It does not, so it starts by assuming everything is a bytearray. Then it looks at how you use those fields in your script, drawing conclusions about what you think those fields are and how you want to use them. Consider the following:

```
--no_schema.pig
daily = load 'NYSE_daily';
calcs = foreach daily generate $7 / 1000, $3 * 100.0, SUBSTRING($0, 0, 1), $6 - $3;
```

In the expression `$7 / 1000`, `1000` is an integer, so it is a safe guess that the eighth field of *NYSE_daily* is an integer or something that can be cast to an integer. In the same way, `$3 * 100.0` indicates `$3` is a double, and the use of `$0` in a function that takes a chararray as an argument indicates the type of `$0`. But what about the last expression, `$6 - $3`? The `-` operator is used only with numeric types in Pig, so Pig can safely guess that `$3` and `$6` are numeric. But should it treat them as integers or floating-point numbers? Here Pig plays it safe and guesses that they are floating points, casting them to doubles. This is the safer bet because if they actually are integers, those can be represented as floating-point numbers, but the reverse is not true. However, because floating-point arithmetic is much slower and subject to loss of precision, if these values really are integers, you should cast them so that Pig uses integer types in this case.

There are also cases where Pig cannot make any intelligent guess:

```
--no_schema_filter
daily = load 'NYSE_daily';
fltrd = filter daily by $6 > $3;
```

`>` is a valid operator on numeric, chararray, and bytearray types in Pig Latin. So, Pig has no way to make a guess. In this case, it treats these fields as if they were bytearrays, which means it will do a byte-to-byte comparison of the data in these fields.

Pig also has to handle the case where it guesses wrong and must adapt on the fly. Consider the following:

```
--unintended_walks.pig
player     = load 'baseball' as (name:chararray, team:chararray,
                pos:bag{t:(p:chararray)}, bat:map[]);
unintended = foreach player generate bat#'base_on_balls' - bat#'ibbs';
```

Because the values in maps can be of any type, Pig has no idea what type `bat#'base_on_balls'` and `bat#'ibbs'` are. By the rules laid out previously, Pig will assume they are doubles. But let's say they actually turn out to be represented internally as integers.* In that case, Pig will need to adapt at runtime and convert what it thought was a cast from bytearray to double into a cast from int to double. Note that it will still produce a double output and not an int output. This might seem nonintuitive; see "How Strongly Typed Is Pig?" on page 32 for details on why this is so. It should be noted that in Pig 0.8 and earlier, much of this runtime adaption code was shaky and often failed. In 0.9, much of this has been fixed. But if you are using an older version of Pig, you might need to cast the data explicitly to get the right results.

Finally, Pig's knowledge of the schema can change at different points in the Pig Latin script. In all of the previous examples where we loaded data without a schema and then passed it to a `foreach` statement, the data started out without a schema. But after the `foreach`, the schema is known. Similarly, Pig can start out knowing the schema, but if the data is mingled with other data without a schema, the schema can be lost. That is, lack of schema is contagious:

```
--no_schema_join.pig
divs  = load 'NYSE_dividends' as (exchange, stock_symbol, date, dividends);
daily = load 'NYSE_daily';
jnd   = join divs by stock_symbol, daily by $1;
```

In this example, because Pig does not know the schema of `daily`, it cannot know the schema of the join of `divs` and `daily`.

Casts

The previous sections have referenced casts in Pig without bothering to define how casts work. The syntax for casts in Pig is the same as in Java—the type name in parentheses before the value:

```
--unintended_walks_cast.pig
player     = load 'baseball' as (name:chararray, team:chararray,
                pos:bag{t:(p:chararray)}, bat:map[]);
unintended = foreach player generate (int)bat#'base_on_balls' - (int)bat#'ibbs';
```

The syntax for specifying types in casts is exactly the same as specifying them in schemas, as shown previously in Table 4-1.

Not all conceivable casts are allowed in Pig. Table 4-2 describes which casts are allowed between scalar types. Casts to bytearrays are never allowed because Pig does not know how to represent the various data types in binary format. Casts from bytearrays to any type are allowed. Casts to and from complex types currently are not allowed, except from bytearray, although conceptually in some cases they could be.

* That is not the case in the example data. For that to be the case, you would need to use a loader that did load the `bat` map with these values as integers.

Table 4-2. Supported casts

	To int	To long	To float	To double	To chararray
From int		Yes.	Yes.	Yes.	Yes.
From long	Yes. Any values greater than 2^{31} or less than -2^{31} will be truncated.		Yes.	Yes.	Yes.
From float	Yes. Values will be truncated to int values.	Yes. Values will be truncated to long values.		Yes.	Yes.
From double	Yes. Values will be truncated to int values.	Yes. Values will be truncated to long values.	Yes. Values with precision beyond what float can represent will be truncated.		Yes.
From chararray	Yes. Chararrays with nonnumeric characters result in null.	Yes. Chararrays with nonnumeric characters result in null.	Yes. Chararrays with nonnumeric characters result in null.	Yes. Chararrays with nonnumeric characters result in null.	

One type of casting that requires special treatment is casting from bytearray to other types. Because bytearray indicates a string of bytes, Pig does not know how to convert its contents to any other type. Continuing the previous example, both `bat#'base_on_balls'` and `bat#'ibbs'` were loaded as bytearrays. The casts in the script indicate that you want them treated as ints.

Pig does not know whether integer values in *baseball* are stored as ASCII strings, Java serialized values, binary-coded decimal, or some other format. So it asks the load function, because it is that function's responsibility to cast bytearrays to other types. In general this works nicely, but it does lead to a few corner cases where Pig does not know how to cast a bytearray. In particular, if a UDF returns a bytearray, Pig will not know how to perform casts on it because that bytearray is not generated by a load function.

Before leaving the topic of casts, we need to consider cases where Pig inserts casts for the user. These casts are implicit, compared to explicit casts where the user indicates the cast. Consider the following:

```
--total_trade_estimate.pig
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float, close:float,
            volume:int, adj_close:float);
rough = foreach daily generate volume * close;
```

In this case, Pig will change the second line to `(float)volume * close` to do the operation without losing precision. In general, Pig will always widen types to fit when it needs to insert these implicit casts. So, int and long together will result in a long; int or long and float will result in a float; and int, long, or float and double will result in a double. There are no implicit casts between numeric types and chararrays or other types.

How Strongly Typed Is Pig?

In a strongly typed computer language (e.g., Java), the user must declare up front the type for all variables. In weakly typed languages (e.g., Perl), variables can take on values of different type and adapt as the occasion demands. So which is Pig? For the most part it is strongly typed. If you describe the schema of your data, Pig expects your data to be what you said. But when Pig does not know the schema, it will adapt to the actual types at runtime. (Perhaps we should say Pig is "gently typed." It is strong but willing to work with data that does not live up to its expectations.) To see the differences between these two cases, look again at this example:

```
--unintended_walks.pig
player     = load 'baseball' as (name:chararray, team:chararray,
                pos:bag{t:(p:chararray)}, bat:map[]);
unintended = foreach player generate bat#'base_on_balls' - bat#'ibbs';
```

In this example, remember we are pretending that the values for `base_on_balls` and `ibbs` turn out to be represented as integers internally (that is, the load function constructed them as integers). If Pig were weakly typed, the output of `unintended` would be records with one field typed as an integer. As it is, Pig will output records with one field typed as a double. Pig will make a guess and then do its best to massage the data into the types it guessed.

The downside here is that users coming from weakly typed languages are surprised, and perhaps frustrated, when their data comes out as a type they did not anticipate. However, on the upside, by looking at a Pig Latin script it is possible to know what the output data type will be in these cases without knowing the input data.

CHAPTER 5
Introduction to Pig Latin

It is time to dig into Pig Latin. This chapter provides you with the basics of Pig Latin, enough to write your first useful scripts. More advanced features of Pig Latin are covered in Chapter 6.

Preliminary Matters

Pig Latin is a dataflow language. Each processing step results in a new data set, or *relation*. In `input = load 'data'`, `input` is the name of the relation that results from loading the data set *data*. A relation name is referred to as an *alias*. Relation names look like variables, but they are not. Once made, an assignment is permanent. It is possible to reuse relation names; for example, this is legitimate:

```
A = load 'NYSE_dividends' (exchange, symbol, date, dividends);
A = filter A by dividends > 0;
A = foreach A generate UPPER(symbol);
```

However, it is not recommended. It looks here as if you are reassigning `A`, but really you are creating new relations called `A`, losing track of the old relations called `A`. Pig is smart enough to keep up, but it still is not a good practice. It leads to confusion when trying to read your programs (which `A` am I referring to?) and when reading error messages.

In addition to relation names, Pig Latin also has field names. They name a field (or column) in a relation. In the previous snippet of Pig Latin, `dividends` and `symbol` are examples of field names. These are somewhat like variables in that they will contain a different value for each record as it passes through the pipeline, but you cannot assign values to them.

Both relation and field names must start with an alphabetic character, and then they can have zero or more alphabetic, numeric, or _ (underscore) characters. All characters in the name must be ASCII.

Case Sensitivity

Unfortunately, Pig Latin cannot decide whether it is case-sensitive. Keywords in Pig Latin are not case-sensitive; for example, `LOAD` is equivalent to `load`. But relation and field names are. So `A = load 'foo';` is not equivalent to `a = load 'foo';`. UDF names are also case-sensitive, thus `COUNT` is not the same UDF as `count`.

Comments

Pig Latin has two types of comment operators: SQL-style single-line comments (`--`) and Java-style multiline comments (`/* */`). For example:

```
A = load 'foo'; --this is a single-line comment
/*
 * This is a multiline comment.
 */
B = load /* a comment in the middle */'bar';
```

Input and Output

Before you can do anything of interest, you need to be able to add inputs and outputs to your data flows.

Load

The first step to any data flow is to specify your input. In Pig Latin this is done with the `load` statement. By default, `load` looks for your data on HDFS in a tab-delimited file using the default load function `PigStorage`. `divs = load '/data/examples/NYSE_dividends';` will look for a file called *NYSE_dividends* in the directory */data/examples*. You can also specify relative path names. By default, your Pig jobs will run in your home directory on HDFS, */users/yourlogin*. Unless you change directories, all relative paths will be evaluated from there. You can also specify a full URL for the path, for example, *hdfs://nn.acme.com/data/examples/NYSE_dividends* to read the file from the HDFS instance that has `nn.acme.com` as a NameNode.

In practice, most of your data will not be in tab-separated text files. You also might be loading data from storage systems other than HDFS. Pig allows you to specify the function for loading your data with the `using` clause. For example, if you wanted to load your data from HBase, you would use the loader for HBase:

```
divs = load 'NYSE_dividends' using HBaseStorage();
```

If you do not specify a load function, the built-in function `PigStorage` will be used. You can also pass arguments to your load function via the `using` clause. For example, if you are reading comma-separated text data, `PigStorage` takes an argument to indicate which character to use as a separator:

```
divs = load 'NYSE_dividends' using PigStorage(',');
```

The `load` statement also can have an `as` clause, which allows you to specify the schema of the data you are loading. (The syntax and semantics of declaring schemas in Pig Latin is discussed in "Schemas" on page 27.)

```
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
```

When specifying a "file" to read from HDFS, you can specify directories. In this case, Pig will find all files under the directory you specify and use them as input for that `load` statement. So, if you had a directory *input* with two datafiles *today* and *yesterday* under it, and you specified *input* as your file to load, Pig will read both *today* and *yesterday* as input. If the directory you specify has other directories, files in those directories will be included as well.

`PigStorage` and `TextLoader`, the two built-in Pig load functions that operate on HDFS files, support globs.* With globs, you can read multiple files that are not under the same directory or read some but not all files under a directory. Table 5-1 describes globs that are valid in Hadoop 0.20. Be aware that glob meaning is determined by HDFS underneath Pig, so the globs that will work for you depend on your version of HDFS. Also, if you are issuing Pig Latin commands from a Unix shell command line, you will need to escape many of the glob characters to prevent your shell from expanding them.

Table 5-1. Globs in Hadoop 0.20

Glob	Meaning
`?`	Matches any single character.
`*`	Matches zero or more characters.
`[abc]`	Matches a single character from character set (a,b,c).
`[a-z]`	Matches a single character from the character range (a..z), inclusive. The first character must be lexicographically less than or equal to the second character.
`[^abc]`	Matches a single character that is not in the character set (a, b, c). The ^ character must occur immediately to the right of the opening bracket.
`[^a-z]`	Matches a single character that is not from the character range (a..z), inclusive. The ^ character must occur immediately to the right of the opening bracket.
`\c`	Removes (escapes) any special meaning of character c.
`{ab,cd}`	Matches a string from the string set {ab, cd}.

* Any loader that uses `FileInputFormat` as its InputFormat will support globs. Most loaders that load data from HDFS use this InputFormat.

Store

After you have finished processing your data, you will want to write it out somewhere. Pig provides the `store` statement for this purpose. In many ways it is the mirror image of the `load` statement. By default, Pig stores your data on HDFS in a tab-delimited file using `PigStorage`:†

```
store processed into '/data/examples/processed';
```

Pig will write the results of your processing into a directory *processed* in the directory */data/examples*. You can specify relative path names, as well as a full URL for the path, such as *hdfs://nn.acme.com/data/examples/processed*.

If you do not specify a store function, `PigStorage` will be used. You can specify a different store function with a `using` clause:

```
store processed into 'processed' using
      HBaseStorage();
```

You can also pass arguments to your store function. For example, if you want to store your data as comma-separated text data, `PigStorage` takes an argument to indicate which character to use as a separator:

```
store processed into 'processed' using PigStorage(',');
```

As noted in "Running Pig" on page 13, when writing to a filesystem, *processed* will be a directory with part files rather than a single file. But how many part files will be created? That depends on the parallelism of the last job before the `store`. If it has reduces, it will be determined by the parallel level set for that job. See "Parallel" on page 49 for information on how this is determined. If it is a map-only job, it will be determined by the number of maps, which is controlled by Hadoop and not Pig.

Dump

In most cases you will want to store your data somewhere when you are done processing it. But occasionally you will want to see it on the screen. This is particularly useful during debugging and prototyping sessions. It can also be useful for quick *ad hoc* jobs. `dump` directs the output of your script to your screen:

```
dump processed;
```

Up through version 0.7, the output of `dump` matches the format of constants in Pig Latin. So, longs are followed by an `L`, and floats by an `F`, and maps are surrounded by `[]` (brackets), tuples by `()` (parentheses), and bags by `{}` (braces). Starting with version 0.8, the `L` for longs and `F` for floats have been dropped, though the markers for the complex types have been kept. Nulls are indicated by missing values, and fields are separated by commas. Because each record in the output is a tuple, it is surrounded by `()`.

† A single function can be both a load and store function, as `PigStorage` is.

Relational Operations

Relational operators are the main tools Pig Latin provides to operate on your data. They allow you to transform it by sorting, grouping, joining, projecting, and filtering. This section covers the basic relational operators. More advanced features of these operators, as well as advanced relational operators, are covered in "Advanced Relational Operations" on page 57. What is covered here will be enough to get you started programming in Pig Latin.

foreach

`foreach` takes a set of expressions and applies them to every record in the data pipeline, hence the name `foreach`. From these expressions it generates new records to send down the pipeline to the next operator. For those familiar with database terminology, it is Pig's projection operator. For example, the following code loads an entire record, but then removes all but the `user` and `id` fields from each record:

```
A = load 'input' as (user:chararray, id:long, address:chararray, phone:chararray,
      preferences:map[]);
B = foreach A generate user, id;
```

Expressions in foreach

`foreach` supports an array of expressions. The simplest are constants and field references. The syntax for constants has already been discussed in "Types" on page 23. Field references can be by name (as shown in the preceding example) or by position. Positional references are preceded by a `$` (dollar sign) and start from 0:

```
prices = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close,
             volume, adj_close);
gain   = foreach prices generate close - open;
gain2  = foreach prices generate $6 - $3;
```

Relations `gain` and `gain2` will contain the same values. Positional style references are useful in situations where the schema is unknown or undeclared.

In addition to using names and positions, you can refer to all fields using * (asterisk), which produces a tuple that contains all the fields. Beginning in version 0.9, you can also refer to ranges of fields using `..` (two periods). This is particularly useful when you have many fields and do not want to repeat them all in your `foreach` command:

```
prices    = load 'NYSE_daily' as (exchange, symbol, date, open,
                high, low, close, volume, adj_close);
beginning = foreach prices generate ..open; -- produces exchange, symbol, date, open
middle    = foreach prices generate open..close; -- produces open, high, low, close
end       = foreach prices generate volume..; -- produces volume, adj_close
```

Standard arithmetic operators for integers and floating-point numbers are supported: `+` for addition, `-` for subtraction, * for multiplication, and `/` for division. These operators return values of their own type, so `5/2` is `2`, whereas `5.0/2.0` is `2.5`. In addition, for

integers the modulo operator `%` is supported. The unary negative operator (`-`) is also supported for both integers and floating-point numbers. Pig Latin obeys the standard mathematical precedence rules. For information on what happens when arithmetic operators are applied across different types (for example, `5/2.0`), see "Casts" on page 30.

Null values are viral for all arithmetic operators. That is, `x + null = null` for all values of `x`.

Pig also provides a binary condition operator, often referred to as *bincond*. It begins with a Boolean test, followed by a `?`, then the value to return if the test is true, then a `:`, and finally the value to return if the test is false. If the test returns null, bincond returns null. Both value arguments of the bincond must return the same type:

```
2 == 2 ? 1 : 4 --returns 1
2 == 3 ? 1 : 4 --returns 4
null == 2 ? 1 : 4 -- returns null
2 == 2 ? 1 : 'fred' -- type error; both values must be of the same type
```

To extract data from complex types, use the projection operators. For maps this is `#` (the pound or hash), followed by the name of the key as a string. Keep in mind that the value associated with a key may be of any type. If you reference a key that does not exist in the map, the result is a null:

```
bball = load 'baseball' as (name:chararray, team:chararray,
          position:bag{t:(p:chararray)}, bat:map[]);
avg = foreach bball generate bat#'batting_average';
```

Tuple projection is done with `.`, the dot operator. As with top-level records, the field can be referenced by name (if you have a schema for the tuple) or by position. Referencing a nonexistent positional field in the tuple will return null. Referencing a field name that does not exist in the tuple will produce an error:

```
A = load 'input' as (t:tuple(x:int, y:int));
B = foreach A generate t.x, t.$1;
```

Bag projection is not as straightforward as map and tuple projection. Bags do not guarantee that their tuples are stored in any order, so allowing a projection of the tuple inside the bag would not be meaningful. Instead, when you project fields in a bag, you are creating a new bag with only those fields:

```
A = load 'input' as (b:bag{t:(x:int, y:int)});
B = foreach A generate b.x;
```

This will produce a new bag whose tuples have only the field `x` in them. You can project multiple fields in a bag by surrounding the fields with parentheses and separating them by commas:

```
A = load 'input' as (b:bag{t:(x:int, y:int)});
B = foreach A generate b.(x, y);
```

This seemingly pedantic distinction that `b.x` is a bag and not a scalar value has consequences. Consider the following Pig Latin, which will not work:

```
A = load 'foo' as (x:chararray, y:int, z:int);
B = group A by x; -- produces bag A containing all the records for a given value of x
C = foreach B generate SUM(A.y + A.z);
```

It is clear what the programmer is trying to do here. But because `A.y` and `B.y` are bags and the addition operator is not defined on bags, this will produce an error.‡ The correct way to do this calculation in Pig Latin is:

```
A = load 'foo' as (x:chararray, y:int, z:int);
A1 = foreach A generate x, y + z as yz;
B = group A1 by x;
C = foreach B generate SUM(A1.yz);
```

UDFs in foreach

User Defined Functions (UDFs) can be invoked in `foreach`. These are called evaluation functions, or *eval funcs*. Because they are part of a `foreach` statement, these UDFs take one record at a time and produce one output. Keep in mind that either the input or the output can be a bag, so this one record can contain a bag of records:

```
-- udf_in_foreach.pig
divs  = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
 --make sure all strings are uppercase
upped = foreach divs generate UPPER(symbol) as symbol, dividends;
grpd  = group upped by symbol;   --output a bag upped for each value of symbol
--take a bag of integers, produce one result for each group
sums  = foreach grpd generate group, SUM(upped.dividends);
```

In addition, eval funcs can take * as an argument, which passes the entire record to the function. They can also be invoked with no arguments at all.

For a complete list of UDFs that are provided with Pig, see Appendix A. For a discussion of how to invoke UDFs not distributed as part of Pig, see "User Defined Functions" on page 51.

Naming fields in foreach

The result of each `foreach` statement is a new tuple, usually with a different schema than the tuple that was an input to `foreach`. Pig can infer the data types of the fields in this schema from the `foreach` statement. But it cannot always infer the names of those fields. For fields that are simple projections with no other operators applied, Pig keeps the same name as before:

```
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
          date:chararray, dividends:float);
```

‡ You might object and say that Pig could figure out what is intended here and do it, since `SUM(A.y + A.z)` could be decomposed to "foreach record in A, add y and z and then take the sum." This is true. But when we change the group to a cogroup so that there are two bags `A` and `B` involved (see "cogroup" on page 66) and change the sum to `SUM(A.y + B.z)`, because neither A nor B guarantee any ordering, this is not a well-defined operation. In designing the language, we thought it better to be consistent and always say that bags could not be added rather than allow it in some instances and not others.

```
sym  = foreach divs generate symbol;
describe sym;

sym: {symbol: chararray}
```

Once any expression beyond simple projection is applied, Pig does not assign a name to the field. If you do not explicitly assign a name, the field will be nameless and will be addressable only via a positional parameter, for example, `$0`. You can assign a name with the `as` clause:

```
divs     = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                date:chararray, dividends:float);
in_cents = foreach divs generate dividends * 100.0 as dividend, dividends * 100.0;
describe in_cents;

in_cents: {dividend: double,double}
```

Notice that in `foreach` the `as` is attached to each expression. This is different than `load`, where it is attached to the entire statement. The reason for this will become clear when we discuss `flatten` in "flatten" on page 57.

Filter

The `filter` statement allows you to select which records will be retained in your data pipeline. A `filter` contains a predicate. If that predicate evaluates to true for a given record, that record will be passed down the pipeline. Otherwise, it will not.

Predicates can contain the equality operators you expect, including `==` to test equality, and `!=`, `>`, `>=`, `<`, and `<=`. These comparators can be used on any scalar data type. `==` and `!=` can be applied to maps and tuples. To use these with two tuples, both tuples must have either the same schema or no schema. None of the equality operators can be applied to bags.

Pig Latin follows the operator precedence that is standard in most programming languages, where arithmetic operators have precedence over equality operators. So, `x + y == a + b` is equivalent to `(x + y) == (a + b)`.

For chararrays, users can test to see whether the chararray matches a regular expression:

```
-- filter_matches.pig
divs        = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                date:chararray, dividends:float);
startswithcm = filter divs by symbol matches 'CM.*';
```

Pig uses Java's regular expression format. This format requires the entire chararray to match, not just a portion as in Perl-style regular expressions. For example, if you are looking for all fields that contain the string "fred", you must say `'.*fred.*'` and not `'fred'`. The latter will match only the chararray `fred`.

You can find chararrays that do not match a regular expression by preceding the test with `not`:

```
-- filter_not_matches.pig
divs           = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                     date:chararray, dividends:float);
notstartswithcm = filter divs by not symbol matches 'CM.*';
```

You can combine multiple predicates into one by using the Boolean operators `and` and `or`, and you can reverse the outcome of any predicate by using the Boolean `not` operator. As is standard, the precedence of Boolean operators, from highest to lowest, is `not`, `and`, `or`. Thus `a and b or not c` is equivalent to `(a and b) or (not c)`.

Pig will short-circuit Boolean operations when possible. If the first (left) predicate of an `and` is false, the second (right) will not be evaluated. So in `1 == 2 and udf(x)`, the UDF will never be invoked. Similarly, if the first predicate of an `or` is true, the second predicate will not be evaluted. `1 == 1 or udf(x)` will never invoke the UDF.

For Boolean operators, nulls follow the SQL trinary logic. Thus `x == null` results in a value of `null`, not `true` (*even when x is null also*) or `false`. Filters pass through only those values that are `true`. So for a field that had three values `2`, `null`, and `4`, if you applied a filter `x == 2` to it, only the first record where the value is `2` would be passed through the filter. Likewise, `x != 2` would return only the last record where the value is `4`. The way to look for null values is to use the `is null` operator, which returns true whenever the value is `null`. To find values that are not null, use `is not null`.

Likewise, `null` neither matches nor fails to match any regular expression value.

Just as there are UDFs to be used in evaluation expressions, there are UDFs specifically for filtering records, called *filter funcs*. These are eval funcs that return a Boolean value and can be invoked in the `filter` statement. Filter funcs cannot be used in `foreach` statements.

Group

The `group` statement collects together records with the same key. It is the first operator we have looked at that shares its syntax with SQL, but it is important to understand that the grouping operator in Pig Latin is fundamentally different than the one in SQL. In SQL the `group by` clause creates a group that must feed directly into one or more aggregate functions. In Pig Latin there is no direct connection between `group` and aggregate functions. Instead, `group` does exactly what it says: collects all records with the same value for the provided key together into a bag. You can then pass this to an aggregate function if you want or do other things with it:

```
-- count.pig
daily = load 'NYSE_daily' as (exchange, stock);
grpd  = group daily by stock;
cnt   = foreach grpd generate group, COUNT(daily);
```

That example groups records by the key `stock` and then counts them. It is just as legitimate to group them and then store them for processing at a later time:

```
-- group.pig
daily = load 'NYSE_daily' as (exchange, stock);
grpd  = group daily by stock;
store grpd into 'by_group';
```

The records coming out of the `group by` statement have two fields, the key and the bag of collected records. The key field is named `group`.§ The bag is named for the alias that was grouped, so in the previous examples it will be named `daily` and have the same schema as the relation `daily`. If the relation `daily` has no schema, the bag `daily` will have no schema. For each record in the group, the entire record (including the key) is in the bag. Changing the last line of the previous script from `store grpd...` to `describe grpd;` will produce:

```
grpd: {group: bytearray,daily: {exchange: bytearray,stock: bytearray}}
```

You can also group on multiple keys, but the keys must be surrounded by parentheses. The resulting records still have two fields. In this case, the `group` field is a tuple with a field for each key:

```
--twokey.pig
daily = load 'NYSE_daily' as (exchange, stock, date, dividends);
grpd  = group daily by (exchange, stock);
avg   = foreach grpd generate group, AVG(daily.dividends);
describe grpd;
grpd: {group: (exchange: bytearray,stock: bytearray),daily: {exchange: bytearray,
    stock: bytearray,date: bytearray,dividends: bytearray}}
```

You can also use `all` to group together all of the records in your pipeline:

```
--countall.pig
daily = load 'NYSE_daily' as (exchange, stock);
grpd  = group daily all;
cnt   = foreach grpd generate COUNT(daily);
```

The record coming out of `group all` has the chararray literal `all` as a key. Usually this does not matter because you will pass the bag directly to an aggregate function such as `COUNT`. But if you plan to store the record or use it for another purpose, you might want to project out the artificial key first.

`group` is the first operator we have looked at that usually will force a reduce phase. Grouping means collecting all records where the key has the same value. If the pipeline is in a map phase, this will force it to shuffle and then reduce. If the pipeline is already in a reduce, this will force it to pass through map, shuffle, and reduce phases.

§ Thus the keyword `group` is overloaded in Pig Latin. This is unfortunate and confusing, but also hard to change now.

Because grouping collects *all* records together with the same value for the key, you often get skewed results. That is, just because you have specified that your job have 100 reducers, there is no reason to expect that the number of values per key will be distributed evenly. They might have a Gaussian or power law distribution.‖ For example, suppose you have an index of web pages and you group by the base URL. Certain values such as `yahoo.com` are going to have far more entries than most, which means that some reducers get far more data than others. Because your MapReduce job is not finished (and any subsequent ones cannot start) until all your reducers have finished, this skew will significantly slow your processing. In some cases it will also be impossible for one reducer to manage that much data.

Pig has a number of ways that it tries to manage this skew to balance out the load across your reducers. The one that applies to grouping is Hadoop's combiner. For details of how Hadoop's combiner works, see "Combiner Phase" on page 190. This does not remove all skew, but it places a bound on it. And because for most jobs the number of mappers will be at most in the tens of thousands, even if the reducers get a skewed number of records, the absolute number of records per reducer will be small enough that the reducers can handle them quickly.

Unfortunately, not all calculations can be done using the combiner. Calculations that can be decomposed into any number of steps, such as sum, are called *distributive*. These fit nicely into the combiner. Calculations that can be decomposed into an initial step, any number of intermediate steps, and a final step are referred to as *algebraic*. Count is an example of such a function, where the initial step is a count and the intermediate and final steps are sums. Distributive is a special case of algebraic, where the initial, intermediate, and final steps are all the same. Session analysis, where you want to track a user's actions on a website, is an example of a calculation that is not algebraic. You must have all the records sorted by timestamp before you can start analyzing their interaction with the site.

Pig's operators and built-in UDFs use the combiner whenever possible, because of its skew-reducing features and because early aggregation greatly reduces the amount of data shipped over the network and written to disk, thus speeding performance significantly. UDFs can indicate when they can work with the combiner by implementing the `Algebraic` interface. For information on how to make your UDFs use the combiner, see "Algebraic Interface" on page 135.

For information on how to determine the level of parallelism when executing your `group` operation, see "Parallel" on page 49. Also, keep in mind that when using `group all`, you are necessarily serializing your pipeline. That is, this step and any step after it until you split out the single bag now containing all of your records will not be done in parallel.

‖ In my experience, the vast majority of data tracking human activity follows a power law distribution.

Finally, `group` handles nulls in the same way that SQL handles them: by collecting all records with a null key into the same group. Note that this is in direct contradiction to the way expressions handle nulls (remember that neither `null == null` nor `null != null` are true) and to the way `join` (see "Join" on page 45) handles nulls.

Order by

The `order` statement sorts your data for you, producing a total order of your output data. Total order means that not only is the data sorted in each partition of your data, it is also guaranteed that all records in partition `n` are less than all records in partition `n - 1` for all `n`. When your data is stored on HDFS, where each partition is a part file, this means that `cat` will output your data in order.

The syntax of `order` is similar to `group`. You indicate a key or set of keys by which you wish to order your data. One glaring difference is that there are no parentheses around the keys when multiple keys are indicated in `order`:

```
--order.pig
daily   = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float, close:float,
            volume:int, adj_close:float);
bydate  = order daily by date;

--order2key.pig
daily          = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
                     date:chararray, open:float, high:float, low:float,
                     close:float, volume:int, adj_close:float);
bydatensymbol  = order daily by date, symbol;
```

It is also possible to reverse the order of the sort by appending `desc` to a key in the sort. In `order` statements with multiple keys, `desc` applies only to the key it immediately follows. Other keys will still be sorted in ascending order:

```
--orderdesc.pig
daily    = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float, close:float,
            volume:int, adj_close:float);
byclose  = order daily by close desc, open;
dump byclose; -- open still sorted in ascending order
```

Data is sorted based on the types of the indicated fields: numeric values are sorted numerically, chararray fields are sorted lexically, and bytearray fields are sorted lexically, using byte values rather than character values. Sorting by maps, tuples, or bags produces errors. For all data types, nulls are taken to be smaller than all possible values for that type, and thus will always appear first (or last when `desc` is used).

As discussed earlier in "Group" on page 41, skew of the values in data is very common. This affects `order` just as it does `group`, causing some reducers to take significantly longer than others. To address this, Pig balances the output across reducers. It does this by first sampling the input of the `order` statement to get an estimate of the key distribution. Based on this sample, it then builds a partitioner that produces a balanced total order (for details on what a partitioner is, see "Shuffle Phase" on page 191). For example, suppose you are ordering on a chararray field with the values `a, b, e, e, e, e, e, e, m, q, r, z`, and you have three reducers. The partitioner in this case would decide to partition your data such that values `a-e` go to reducer 1, `e` goes to reducer 2, and `m-z` go to reducer 3. Notice that the value `e` can be sent to either reducer 1 or 2. Some records with key `e` will be sent to reducer 1 and some to 2. This allows the partitioner to distribute the data evenly. In practice, we rarely see variance in reducer time exceed 10% when using this algorithm.

An important side effect of the way Pig distributes records to minimize skew is that it breaks the MapReduce convention that all instances of a given key are sent to the same partition. If you have other processing that depends on this convention, do not use Pig's `order` statement to sort data for it.

`order` always causes your data pipeline to go through a reduce phase. This is necessary to collect all equal records together. Also, Pig adds an additional MapReduce job to your pipeline to do the sampling. Because this sampling is very lightweight (it reads only the first record of every block), it generally takes less than 5% of the total job time.

Distinct

The `distinct` statement is very simple. It removes duplicate records. It works only on entire records, not on individual fields:

```
--distinct.pig
-- find a distinct list of ticker symbols for each exchange
-- This load will truncate the records, picking up just the first two fields.
daily   = load 'NYSE_daily' as (exchange:chararray, symbol:chararray);
uniq    = distinct daily;
```

Because it needs to collect like records together in order to determine whether they are duplicates, `distinct` forces a reduce phase. It does make use of the combiner to remove any duplicate records it can delete in the map phase.

The use of `distinct` shown here is equivalent to `select distinct x` in SQL. To learn how to do the equivalent of `select count(distinct x)`, see "Nested foreach" on page 59.

Join

`join` is one of the workhorses of data processing, and it is likely to be in many of your Pig Latin scripts. `join` selects records from one input to put together with records from

another input. This is done by indicating keys for each input. When those keys are equal,[#] the two rows are joined. Records for which no match is found are dropped:

```
--join.pig
daily = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close,
            volume, adj_close);
divs  = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
jnd   = join daily by symbol, divs by symbol;
```

You can also join on multiple keys. In all cases you must have the same number of keys, and they must be of the same or compatible types (where compatible means that an implicit cast can be inserted; see "Casts" on page 30):

```
-- join2key.pig
daily = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close,
            volume, adj_close);
divs  = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
jnd   = join daily by (symbol, date), divs by (symbol, date);
```

Like `foreach`, `join` preserves the names of the fields of the inputs passed to it. It also prepends the name of the relation the field came from, followed by a `::`. Adding `describe jnd;` to the end of the previous example produces:

```
jnd: {daily::exchange: bytearray,daily::symbol: bytearray,daily::date: bytearray,
daily::open: bytearray,daily::high: bytearray,daily::low: bytearray,
daily::close: bytearray,daily::volume: bytearray,daily::adj_close: bytearray,
divs::exchange: bytearray,divs::symbol: bytearray,divs::date: bytearray,
divs::dividends: bytearray}
```

The `daily::` prefix needs to be used only when the field name is no longer unique in the record. In this example, you will need to use `daily::date` or `divs::date` if you wish to refer to one of the `date` fields after the join. But fields such as `open` and `divs` do not need a prefix because there is no ambiguity.

Pig also supports *outer joins*. In outer joins, records that do not have a match on the other side are included, with null values being filled in for the missing fields. Outer joins can be `left`, `right`, or `full`. A left outer join means records from the left side will be included even when they do not have a match on the right side. Likewise, a right outer joins means records from the right side will be included even when they do not have a match on the left side. A full outer join means records from both sides are taken even when they do not have matches:

```
--leftjoin.pig
daily = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close,
            volume, adj_close);
divs  = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
jnd   = join daily by (symbol, date) left outer, divs by (symbol, date);
```

[#] Actually, joins can be on any condition, not just equality, but Pig only supports joins on equality (called equi-joins). See "cross" on page 68 for information on how to do non-equi-joins in Pig.

`outer` is a noise word and can be omitted. Unlike some SQL implementations, `full` is not a noise word. `C = join A by x outer, B by u;` will generate a syntax error, not a full outer join.

Outer joins are supported only when Pig knows the schema of the data on the side(s) for which it might need to fill in nulls. Thus for left outer joins, it must know the schema of the right side; for right outer joins, it must know the schema of the left side; and for full outer joins, it must know both. This is because, without the schema, Pig will not know how many null values to fill in.*

As in SQL, null values for keys do not match anything, *even null values from the other input*. So, for inner joins, all records with null key values are dropped. For outer joins, they will be retained but will not match any records from the other input.

Pig can also do multiple joins in a single operation, as long as they are all being joined on the same key(s). This can be done only for inner joins:

```
A = load 'input1' as (x, y);
B = load 'input2' as (u, v);
C = load 'input3' as (e, f);
alpha = join A by x, B by u, C by e;
```

Self joins are supported, though the data must be loaded twice:

```
--selfjoin.pig
-- For each stock, find all dividends that increased between two dates
divs1     = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                 date:chararray, dividends);
divs2     = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                 date:chararray, dividends);
jnd       = join divs1 by symbol, divs2 by symbol;
increased = filter jnd by divs1::date < divs2::date and
                 divs1::dividends < divs2::dividends;
```

If the preceding code were changed to the following, it would fail:

```
--selfjoin.pig
-- For each stock, find all dividends that increased between two dates
divs1     = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                 date:chararray, dividends);
jnd       = join divs1 by symbol, divs1 by symbol;
increased = filter jnd by divs1::date < divs2::date and
                 divs1::dividends < divs2::dividends;
```

It seems like this ought to work, since Pig could split the `divs1` data set and send it to join twice. But the problem is that field names would be ambiguous after the join, so the `load` statement must be written twice. The next best thing would be for Pig to figure

* You may object that Pig could determine this by looking at other records in the join and inferring the correct number of fields. However, this does not work for two reasons. First, when no schema is present, Pig does not enforce a semantic that every record has the same schema. So, assuming Pig can infer one record from another is not valid. Second, there might be no records in the join that match, and thus Pig might have no record to infer from.

out that these two `load` statements are loading the same input and then run the load only once, but it does not do that currently.

Pig does these joins in MapReduce by using the map phase to annotate each record with which input it came from. It then uses the join key as the shuffle key. Thus `join` forces a new reduce phase. Once all of the records with the same value for the key are collected together, Pig does a cross product between the records from both inputs. To minimize memory usage, it has MapReduce order the records coming into the reducer using the input annotation it added in the map phase. Thus all of the records for the left input arrive first. Pig caches these in memory. All of the records for the right input arrive second. As each of these records arrives, it is crossed with each record from the left side to produce an output record. In a multiway join, the left `n - 1` inputs are held in memory, and the `nth` is streamed through. It is important to keep this in mind when writing joins in your Pig queries if you know that one of your inputs has more records per value of the chosen key. Placing that input on the right side of your join will lower memory usage and possibly increase your script's performance.

Limit

Sometimes you want to see only a limited number of results. `limit` allows you do this:

```
--limit.pig
divs    = load 'NYSE_dividends';
first10 = limit divs 10;
```

The example here will return at most 10 lines (if your input has less than 10 lines total, it will return them all). Note that for all operators except `order`, Pig does not guarantee the order in which records are produced. Thus, because *NYSE_dividends* has more than 10 records, the example script could return different results every time. Putting an `order` *immediately* before the `limit` will guarantee that the same results are returned every time.

`limit` causes an additional reduce phase, since it needs to collect the records together to count how many it is returning. It does optimize this phase by limiting the output of each map and then applying the limit again in the reducer. In the case where `limit` is combined with `order`, the two are done together on the map and reduce. That is, on the map side, the records are sorted by MapReduce and the limit applied in the combiner. They are sorted again by MapReduce as part of the shuffle, and Pig applies the limit again in the reducer.

One possible optimization that Pig does not do is terminate reading of the input early once it has reached the number of records specified by `limit`. So, in the example, if you hoped to use this to read just a tiny slice of your input, you will be disappointed. Pig will still read it all.

Sample

`sample` offers a simple way to get a sample of your data. It reads through all of your data but returns only a percentage of rows. What percentage it returns is expressed as a double value, between 0 and 1. So, in the following example, `0.1` indicates 10%:

```
--sample.pig
divs = load 'NYSE_dividends';
some = sample divs 0.1;
```

Currently the sampling algorithm is very simple. The `sample A by 0.1` is rewritten to `filter A by random() <= 0.1`. Obviously this is nondeterministic, so results of a script with `sample` will vary with every run. Also, the percentage will not be an exact match, but close. There has been discussion about adding more sophisticated sampling techniques, but it has not been done yet.

Parallel

One of Pig's core claims is that it provides a language for parallel data processing. One of the tenets of Pig's philosophy is that Pigs are domestic animals (see "Pig Philosophy" on page 9), so Pig prefers that you tell it how parallel to be. To do this, it provides the `parallel` clause.

The `parallel` clause can be attached to any relational operator in Pig Latin. However, it controls only reduce-side parallelism, so it makes sense only for operators that force a reduce phase. These are: `group`*, `order`, `distinct`, `join`*, `limit`, `cogroup`*, and `cross`. Operators marked with an asterisk have multiple implementations, some of which force a reduce and some which do not. For details on this and on operators not covered in this chapter, see Chapter 6. `parallel` is ignored in local mode because all operations happen serially in local mode:

```
--parallel.pig
daily   = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close,
            volume, adj_close);
bysymbl = group daily by symbol parallel 10;
```

In this example, `parallel` will cause the MapReduce job spawned by Pig to have 10 reducers. `parallel` clauses apply only to the statement to which they are attached; they do not carry through the script. So if this `group` were followed by an `order`, `parallel` would need to be set for that `order` separately. Most likely the `group` will reduce your data size significantly and you will want to change the parallelism:

```
--parallel.pig
daily   = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close,
            volume, adj_close);
bysymbl = group daily by symbol parallel 10;
average = foreach bysymbl generate group, AVG(daily.close) as avg;
sorted  = order average by avg desc parallel 2;
```

If, however, you do not want to set `parallel` separately for every reduce-invoking operator in your script, you can set a script-wide value using the `set` command:

```
--defaultparallel.pig
set default_parallel 10;
daily   = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close,
            volume, adj_close);
bysymbl = group daily by symbol;
average = foreach bysymbl generate group, AVG(daily.close) as avg;
sorted  = order average by avg desc;
```

In this script, all MapReduce jobs will be done with 10 reduces. When you set a default parallel level, you can still add a `parallel` clause to any statement to override the default value. Thus it can be helpful to set a default value as a base to use in most cases, and specifically add a `parallel` clause only when you have an operator that needs a different value.

All of this is rather static, however. What happens if you run the same script across different inputs that have different characteristics? Or what if your input data varies significantly sometimes? You do not want to have to edit your script each time. Using parameter substitution, you can write your parallel clauses with variables, providing values for those variables at runtime. See "Parameter Substitution" on page 77 for details.

So far we have assumed that you know what your parallel value should be. See "Select the Right Level of Parallelism" on page 105 for information on how to determine that.

Finally, what happens if you do not specify a parallel level? Before version 0.8, Pig lets MapReduce set the parallelism in that case. The MapReduce default parallelism is controlled by your cluster configuration. The installation default value is one, and most people do not change that. This most likely means that you will be running with only one reducer. This is rarely what you want.

To avoid this situation, Pig added a heuristic in 0.8 to do a gross estimate of what the parallelism should be set to if it is not set. It looks at the initial input size, assumes there will be no data size changes, and then allocates a reducer for every 1G of data. It must be emphasized that this is not a good algorithm. It is provided only to prevent mistakes that result in scripts running very slowly, and, in some extreme cases, mistakes that cause MapReduce itself to have problems. This is a safety net, not an optimizer.

Map Parallelism

`parallel` only lets you set reduce parallelism. What about map parallelism? MapReduce only allows users to set reduce parallelism: it controls map parallelism itself. Because Pig cannot control map parallelism, it cannot expose that to its users either.

In MapReduce, data is read using a class called `InputFormat`. Part of `InputFormat`'s purpose is to tell MapReduce how many map tasks to run. It also suggests where they should be run.

Although Pig cannot give you direct control over how many map tasks to run, it does let you build and run your own `InputFormat` as part of building your own load function. See Chapter 11 for details on how to do this.

User Defined Functions

Much of the power of Pig lies in its ability to let users combine irs operators with their own or others' code via UDFs. Up through version 0.7, all UDFs must be written in Java and are implemented as Java classes.† This makes it very easy to add new UDFs to Pig by writing a Java class and telling Pig about your JAR file.

As of version 0.8, UDFs can also be written in Python. Pig uses Jython to execute Python UDFs, so they must be compatible with Python 2.5 and cannot use Python 3 features.

Pig itself comes packaged with some UDFs. Prior to version 0.8, this was a very limited set, including only the standard SQL aggregate functions and a few others. In 0.8, a large number of standard string-processing, math, and complex-type UDFs were added. For a complete list and description of built-in UDFs, see "Built-in UDFs" on page 171.

Piggybank is a collection of user-contributed UDFs that is packaged and released along with Pig. Piggybank UDFs are not included in the Pig JAR, and thus you have to register them manually in your script. See "Piggybank" on page 187 for more information.

Of course you can also write your own UDFs or use those written by other users. For details of how to write your own, see Chapter 10. Finally, you can use some static Java functions as UDFs as well.

Registering UDFs

When you use a UDF that is not already built into Pig, you have to tell Pig where to look for that UDF. This is done via the `register` command. For example, let's say you want to use the `Reverse` UDF provided in Piggybank (for information on where to find the Piggybank JAR, see "Piggybank" on page 187):

```
--register.pig
register 'your_path_to_piggybank/piggybank.jar';
divs      = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                date:chararray, dividends:float);
backwards = foreach divs generate
                org.apache.pig.piggybank.evaluation.string.Reverse(symbol);
```

† This is why UDF names are case-sensitive in Pig.

This example tells Pig that it needs to include code from *your_path_to_piggybank/piggybank.jar* when it produces a JAR to send to Hadoop. Pig opens all of the registered JARs, takes out the files, and places them in the JAR that it sends to Hadoop to run your jobs.

In this example, we have to give Pig the full package and class name of the UDF. This verbosity can be alleviated in two ways. The first option is to use the `define` command (see "define and UDFs" on page 53). The second option is to include a set of paths on the command line for Pig to search when looking for UDFs. So if instead of invoking Pig as `pig register.pig` we change our invocation to `pig -Dudf.import.list=org.apache.pig.piggybank.evaluation.string register.pig`, we can change our script to:

```
register 'your_path_to_piggybank/piggybank.jar';
divs      = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                date:chararray, dividends:float);
backwards = foreach divs generate Reverse(symbol);
```

Using yet another property, we can get rid of the register command as well. If we add `-Dpig.additional.jars=/usr/local/pig/piggybank/piggybank.jar` to our command line, the register command is no longer necessary.

In many cases it is better to deal with registration and definition issues explicitly in the script via the `register` and `define` commands than use these properties. Otherwise, everyone who runs your script has to know how to configure the command line. However, in some situations your scripts will always use the same set of JARs and always look in the same places for them. For instance, you might have a set of JARs used by everyone in your company. In this case, placing these properties in a shared properties file and using that with your Pig scripts will make sharing those UDFs easier and assure that everyone is using the correct versions of them.

In 0.8 and later versions, the `register` command can also take HDFS paths. If your JARs are stored in HDFS, you could then say `register 'hdfs://user/jar/acme.jar';`. Starting in 0.9, `register` accepts globs. So if all of the JARs you need are stored in one directory, you could include them all with `register '/usr/local/share/pig/udfs/*.jar'`.

Registering Python UDFs

`register` is also used to locate resources for Python UDFs that you use in your Pig Latin scripts. In this case you do not register a JAR, but rather a Python script that contains your UDF. The Python script must be in your current directory. Using the examples provided in the example code, copying *udfs/python/production.py* to the *data* directory looks like this:

```
--batting_production.pig
register 'production.py' using jython as bballudfs;
players  = load 'baseball' as (name:chararray, team:chararray,
                pos:bag{t:(p:chararray)}, bat:map[]);
nonnull  = filter players by bat#'slugging_percentage' is not null and
                bat#'on_base_percentage' is not null;
calcprod = foreach nonnull generate name, bballudfs.production(
                (float)bat#'slugging_percentage',
                (float)bat#'on_base_percentage');
```

The important differences here are the `using jython` and `as bballudfs` portions of the `register` statement. `using jython` tells Pig that this UDF is written in Python, not Java, and it should use Jython to compile that UDF. Pig does not know where on your system the Jython interpreter is, so you must include *jython.jar* in your classpath when invoking Pig. This can be done by setting the `PIG_CLASSPATH` environment variable.

`as bballudfs` defines a namespace that UDFs from this file are placed in. All UDFs from this file must now be invoked as `bballudfs.`*`udfname`*. Each Python file you load should be given a separate namespace. This avoids naming collisions when you register two Python scripts with duplicate function names.

One caveat: Pig does not trace dependencies inside your Python scripts and send the needed Python modules to your Hadoop cluster. You are required to make sure the modules you need reside on the task nodes in your cluster and that the `PYTHONPATH` environment variable is set on those nodes such that your UDFs will be able to find them for import. This issue has been fixed after 0.9, but as of this writing is not yet released.

define and UDFs

As was alluded to earlier, `define` can be used to provide an alias so that you do not have to use full package names for your Java UDFs. It can also be used to provide constructor arguments to your UDFs. `define` also is used in defining streaming commands, but this section covers only its UDF-related features. For information on using `define` with streaming, see "stream" on page 69. The following provides an example of using `define` to provide an alias for `org.apache.pig.piggybank.evaluation.string.Reverse`:

```
--define.pig
register 'your_path_to_piggybank/piggybank.jar';
define reverse org.apache.pig.piggybank.evaluation.string.Reverse();
divs      = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                date:chararray, dividends:float);
backwards = foreach divs generate reverse(symbol);
```

Eval and filter functions can also take one or more strings as constructor arguments. If you are using a UDF that takes constructor arguments, `define` is the place to provide those arguments. For example, consider a method `CurrencyConverter` that takes two constructor arguments, the first indicating which currency you are converting from and the second which currency you are converting to:

```
--define_constructor_args.pig
register 'acme.jar';
define convert com.acme.financial.CurrencyConverter('dollar', 'euro');
divs      = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                date:chararray, dividends:float);
backwards = foreach divs generate convert(dividends);
```

Calling Static Java Functions

Java has a rich collection of utilities and libraries. Because Pig is implemented in Java, some of these functions can be exposed to Pig users. Starting in version 0.8, Pig offers *invoker* methods that allow you to treat certain static Java functions as if they were Pig UDFs.

Any public static Java function that takes no arguments or some combination of `int`, `long`, `float`, `double`, `String`, or arrays thereof,‡ and returns `int`, `long`, `float`, `double`, or `String` can be invoked in this way.

Because Pig Latin does not support overloading on return types, there is an invoker for each return type: `InvokeForInt`, `InvokeForLong`, `InvokeForFloat`, `InvokeForDouble`, and `InvokeForString`. You must pick the appropriate invoker for the type you wish to return. This method takes two constructor arguments. The first is the full package, classname, and method name. The second is a space-separated list of parameters the Java function expects. Only the types of the parameters are given. If the parameter is an array, `[]` (square brackets) are appended to the type name. If the method takes no parameters, the second constructor argument is omitted.

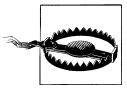
For example, if you wanted to use Java's `Integer` class to translate decimal values to hexadecimal values, you could do:

```
--invoker.pig
define hex InvokeForString('java.lang.Integer.toHexString', 'int');
divs  = load 'NYSE_daily' as (exchange, symbol, date, open, high, low,
           close, volume, adj_close);
nonnull = filter divs by volume is not null;
inhex = foreach nonnull generate symbol, hex((int)volume);
```

If your method takes an array of types, Pig will expect to pass it a bag where each tuple has a single field of that type. So if you had a Java method `com.yourcompany.Stats.stdev` that took an array of doubles, you could use it like this:

```
define stdev InvokeForDouble('com.acme.Stats.stdev', 'double[]');
A = load 'input' as (id: int, dp:double);
B = group A by id;
C = foreach B generate group, stdev(A.dp);
```

‡ For int, long, float, and double, invoker methods can call Java functions that take the scalar types but not the associated Java classes (so `int` but not `Integer`, etc.).



Invokers do not use the `Accumulator` or `Algebraic` interfaces, and are thus likely to be much slower and to use much more memory than UDFs written specifically for Pig. This means that before you pass an array argument to an invoked method, you should think carefully about whether those inefficiencies are acceptable. For more information on these interfaces, see "Accumulator Interface" on page 139 and "Algebraic Interface" on page 135.

Invoking Java functions in this way does have a small cost because reflection is used to find and invoke the methods.

Invoker functions throw Java an `IllegalArgumentException` when they are passed null input. You should place a filter before the invocation to prevent this.

CHAPTER 6
Advanced Pig Latin

In the previous chapter we worked through the basics of Pig Latin. In this chapter we will plumb its depths, and we will also discuss how Pig handles more complex data flows. Finally, we will look at how to use macros and modules to modularize your scripts.

Advanced Relational Operations

We will now discuss the more advanced Pig Latin operators, as well as additional options for operators that were introduced in the previous chapter.

Advanced Features of foreach

In our introduction to `foreach` (see "foreach" on page 37), we discussed how it could take a list of expressions to output for every record in your data pipeline. Now we will look at ways it can explode the number of records in your pipeline, and also how it can be used to apply a set of operations to each record.

flatten

Sometimes you have data in a bag or a tuple and you want to remove that level of nesting. The *baseball* data available on GitHub (see "Code Examples in This Book" on page xi) can be used as an example. Because a player can play more than one position, `position` is stored in a bag. This allows us to still have one entry per player in the *baseball* file.* But when you want to switch around your data on the fly and group

* Those with database experience will notice that this is a violation of the first normal form as defined by E. F. Codd. This intentional denormalization of data is very common in OLAP systems in general, and in large data-processing systems such as Hadoop in particular. RDBMS systems tend to make joins common and then work to optimize them. In systems such as Hadoop, where storage is cheap and joins are expensive, it is generally better to use nested data structures to avoid the joins.

by a particular position, you need a way to pull those entries out of the bag. To do this, Pig provides the `flatten` modifier in `foreach`:

```
--flatten.pig
players = load 'baseball' as (name:chararray, team:chararray,
            position:bag{t:(p:chararray)}, bat:map[]);
pos     = foreach players generate name, flatten(position) as position;
bypos   = group pos by position;
```

A `foreach` with a `flatten` produces a cross product of every record in the bag with all of the other expressions in the `generate` statement. Looking at the first record in *baseball*, we see it is the following (replacing tabs with commas for clarity):

```
Jorge Posada,New York Yankees,{(Catcher),(Designated_hitter)},...
```

Once this has passed through the `flatten` statement, it will be two records:

```
Jorge Posada,Catcher
Jorge Posada,Designated_hitter
```

If there is more than one bag and both are flattened, this cross product will be done with members of each bag as well as other expressions in the `generate` statement. So rather than getting `n` rows (where `n` is the number of records in one bag), you will get `n * m` rows.

One side effect that surprises many users is that if the bag is empty, no records are produced. So if there had been an entry in *baseball* with no position, either because the bag is null or empty, that record would not be contained in the output of *flatten.pig*. The record with the empty bag would be swallowed by `foreach`. There are a couple of reasons for this behavior. One, since Pig may or may not have the schema of the data in the bag, it might have no idea how to fill in nulls for the missing fields. Two, from a mathematical perspective, this is what you would expect. Crossing a set `S` with the empty set results in the empty set. If you wish to avoid this, use a bincond to replace empty bags with a constant bag:

```
--flatten_noempty.pig
players = load 'baseball' as (name:chararray, team:chararray,
            position:bag{t:(p:chararray)}, bat:map[]);
noempty = foreach players generate name,
            ((position is null or IsEmpty(position)) ? {('unknown')} : position)
            as position;
pos     = foreach noempty generate name, flatten(position) as position;
bypos   = group pos by position;
```

`flatten` can also be applied to a tuple. In this case, it does not produce a cross product; instead, it elevates each field in the tuple to a top-level field. Again, empty tuples will remove the entire record.

If the fields in a bag or tuple that is being flattened have names, Pig will carry those names along. As with `join`, to avoid ambiguity, the field name will have the bag's name and `::` prepended to it. As long as the field name is not ambiguous, you are not required to use the *`bagname`*`::` prefix.

If you wish to change the names of the fields, or if the fields initially did not have names, you can attach an `as` clause to your `flatten`, as in the preceding example. If there is more than one field in the bag or tuple that you are assigning names to, you must surround the set of field names with parentheses.

Finally, if you flatten a bag or tuple without a schema and do not provide an `as` clause, the resulting records coming out of your `foreach` will have a null schema. This is because Pig will not know how many fields the `flatten` will result in.†

Nested foreach

So far, all of the examples of `foreach` that we have seen immediately generate one or more lines of output. But `foreach` is more powerful than this. It can also apply a set of relational operations to each record in your pipeline. This is referred to as a *nested foreach*, or inner foreach. One example of how this can be used is to find the number of unique entries in a group. For example, to find the number of unique stock symbols for each exchange in the *NYSE_daily* data:

```
--distinct_symbols.pig
daily    = load 'NYSE_daily' as (exchange, symbol); -- not interested in other fields
grpd     = group daily by exchange;
uniqcnt  = foreach grpd {
           sym      = daily.symbol;
           uniq_sym = distinct sym;
           generate group, COUNT(uniq_sym);
};
```

There are several new things here to unpack; we will walk through each. In this example, rather than `generate` immediately following `foreach`, a `{` (open brace) signals that we will be nesting operators inside this `foreach`. In this nested code, each record passed to `foreach` is handled one at a time.

In the first line we see a syntax that we have not seen outside of `foreach`. In fact, `sym = daily.symbol` would not be legal outside of `foreach`. It is roughly equivalent to the top-level statement `sym = foreach grpd generate daily.symbol`, but it is not stated that way inside the `foreach` because it is not really another `foreach`. There is no relation for it to be associated with (that is, `grpd` is not defined here). This line takes the bag `daily` and produces a new relation `sym`, which is a bag with tuples that have only the field `symbol`.

The second line applies the `distinct` operator to the relation `sym`. Note that even inside `foreach`, relational operators can be applied only to relations; they cannot be applied to expressions. For example, the statement `uniq_sym = distinct daily.symbol` will produce a syntax error because `daily.symbol` is an expression, not a relation. `sym` is a relation. This distinction may seem arbitrary, but it results in Pig Latin having a

† In versions 0.8 and earlier, there is a bug where this `flatten` is assigned a schema of one field, which is a `bytearray`, instead of causing the schema to be null. This bug has been fixed in 0.9.

coherent definition as a language. Without this, strange statements such as `C = distinct 1 + 2` would be legal. One way to think about this is that the assignment operator inside `foreach` can be used to take an expression and create a relation, as happens in this example.

The last line in a nested `foreach` must always be `generate`. This tells Pig how to take the results of the nested operations and produce a record to be put in the outer relation (in this case, `uniqcnt`). So, `generate` is the operator that takes the inner relations and turns them back into expressions for inclusion in the outer relation. That is, if the script read `generate group, uniq_sym`, `uniq_sym` would be treated as a bag for the purpose of the `generate` statement.

Theoretically, any Pig Latin relational operator should be legal inside `foreach`. However, at the moment, only `distinct`, `filter`, `limit`, and `order` are supported.

Let's look at a few more examples of how this feature can be useful, such as to sort the contents of a bag before the bag is passed to a UDF. This is convenient for UDFs that require all of their input to come in a certain order. Consider a stock-analysis UDF that wants to track information about a particular stock over time. The UDF will want input sorted by timestamp:

```
--analyze_stock.pig
register 'acme.jar';
define analyze com.acme.financial.AnalyzeStock();
daily    = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
               date:chararray, open:float, high:float, low:float,
               close:float, volume:int, adj_close:float);
grpd     = group daily by symbol;
analyzed = foreach grpd {
            sorted = order daily by date;
            generate group, analyze(sorted);
};
```

Doing the sorting in Pig Latin, rather than in your UDF, is important for a couple of reasons. One, it means Pig can offload the sorting to MapReduce. MapReduce has the ability to sort data by a secondary key while grouping it. So, the `order` statement in this case does not require a separate sorting operation. Two, it means that your UDF does not need to wait for all data to be available before it starts processing. Instead, it can use the `Accumulator` interface (see "Accumulator Interface" on page 139), which is much more memory efficient.

This feature can be used to find the top *k* elements in a group. The following example will find the top three dividends payed for each stock:

```
--hightest_dividend.pig
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
grpd = group divs by symbol;
top3 = foreach grpd {
            sorted = order divs by dividends desc;
            top    = limit sorted 3;
```

```
            generate group, flatten(top);
};
```

Currently, these nested portions of code are always run serially for each record handed to them. Of course the `foreach` itself will be running in multiple map or reduce tasks, but each instance of the `foreach` will not spawn subtasks to do the nested operations in parallel. So if we added a `parallel 10` clause to the `grpd = group divs by symbol` statement in the previous example, this ordering and limiting would take place in 10 reducers. But each group of stocks would be sorted and the top three records taken serially within one of those 10 reducers.

There is, of course, no requirement that the pipeline inside the `foreach` be a simple linear pipeline. For example, if you wanted to calculate two distinct counts together, you could do the following:

```
--double_distinct.pig
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray);
grpd = group divs all;
uniq = foreach grpd {
            exchanges      = divs.exchange;
            uniq_exchanges = distinct exchanges;
            symbols        = divs.symbol;
            uniq_symbols   = distinct symbols;
            generate COUNT(uniq_exchanges), COUNT(uniq_symbols);
};
```

For simplicity, Pig actually runs this pipeline once for each expression in `generate`. Here this has no side effects because the two data flows are completely disjointed. However, if you constructed a pipeline where there was a split in the flow, and you put a UDF in the shared portion, you would find that it was invoked more often than you expected.

Using Different Join Implementations

When we covered `join` in the previous chapter (see "Join" on page 45), we discussed only the default join behavior. However, Pig offers multiple join implementations, which we will discuss here.

In RDBMS systems, traditionally the SQL optimizer chooses a join implementation for the user. This is nice as long as the optimizer chooses well, which it does in most cases. But Pig has taken a different approach. In the Pig team we like to say that our optimizer is located between the user's chair and keyboard. We empower the user to make these choices rather than having Pig make them. So for operators such as `join` where there are multiple implementations, Pig lets the user indicate his choice via a `using` clause.

This approach fits well with our philosophy that Pigs are domestic animals (i.e., Pig does what you tell it; see "Pig Philosophy" on page 9). Also, as a relatively new product, Pig has a lot of functionality to add. It makes more sense to focus on adding implementation choices and letting the user choose which ones to use, rather than focusing on building an optimizer capable of choosing well.

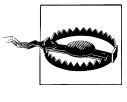
Joining small to large data

A common type of join is doing a lookup in a smaller input. For example, suppose you were processing data where you needed to translate a US ZIP code (postal code) to the state and city it referred to. As there are at most 100,000 zip codes in the US, this translation table should easily fit in memory. Rather than forcing a reduce phase that will sort your big file plus this tiny zip code translation file, it makes sense instead to send the zip code file to every machine, load it into memory, and then do the join by streaming through the large file and looking up each record in the zip code file. This is called a *fragment-replicate join* (because you fragment one file and replicate the other):

```
--repljoin.pig
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float,
            close:float, volume:int, adj_close:float);
divs  = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
jnd   = join daily by (exchange, symbol), divs by (exchange, symbol)
            using 'replicated';
```

The `using 'replicated'` tells Pig to use the fragment-replicate algorithm to execute this join. Because no reduce phase is necessary, all of this can be done in the map task.

The second input listed in the join (in this case, `divs`) is always the input that is loaded into memory. Pig does not check beforehand that the specified input will fit into memory. If Pig cannot fit the replicated input into memory, it will issue an error and fail.



Due to the way Java stores objects in memory, the size of the data on disk will not be the size of the data in memory. See "Memory Requirements of Pig Data Types" on page 26 for a discussion of how data expands in memory in Pig. You will need more memory for a replicated join than you need space on disk to store the replicated input.

Fragment-replicate join supports only inner and left outer joins. It cannot do a right outer join, because when a given map task sees a record in the replicated input that does not match any record in the fragmented input, it has no idea whether it would match a record in a different fragment. So, it does not know whether to emit a record. If you want a right or full outer join, you will need to use the default join operation.

Fragment-replicate join can be used with more than two tables. In this case, all but the first (left-most) table are read into memory.

Pig implements the fragment-replicate join by loading the replicated input into Hadoop's *distributed cache*. The distributed cache is a tool provided by Hadoop that preloads a file onto the local disk of nodes that will be executing the maps or reduces for that job. This has two important benefits. First, if you have a fragment-replicate join that is going to run on 1,000 maps, opening one file in HDFS from 1,000 different machines all at once puts a serious strain on the NameNode and the three data nodes that contain the block for that file. The distributed cache is built specifically to manage

these kinds of issues without straining HDFS. Second, if multiple map tasks are located on the same physical machine, the files in the distributed cache are shared between those instances, thus reducing the number of times the file has to be copied.

Pig runs a map-only MapReduce job to preprocess the file and get it ready for loading into the distributed cache. If there is a `filter` or `foreach` between the `load` and `join`, these will be done as part of this initial job so that the file to be stored in the distributed cache is as small as possible. The join itself will be done in a second map-only job.

Joining skewed data

As we have seen elsewhere, much of the data you will be processing with Pig has significant skew in the number of records per key. For example, if you were building a map of the Web and joining by the domain of the URL (your key), you would expect to see significant skew for values such as `yahoo.com`. Pig's default join algorithm is very sensitive to skew, because it collects all of the records for a given key together on a single reducer. In many data sets, there are a few keys that have three or more orders of magnitude more records than other keys. This results in one or two reducers that will take much longer than the rest. To deal with this, Pig provides *skew join*.

Skew join works by first sampling one input for the join. In that input it identifies any keys that have so many records that skew join estimates it will not be able to fit them all into memory. Then, in a second MapReduce job, it does the join. For all records except those identified in the sample, it does a standard join, collecting records with the same key onto the same reducer. Those keys identified as too large are treated differently. Based on how many records were seen for a given key, those records are split across the appropriate number of reducers. The number of reducers is chosen based on Pig's estimate of how wide the data must be split such that each reducer can fit its split into memory. For the input to the join that is not split, those keys that were split are then replicated to each reducer that contains that key.‡

For example, let's look at how the following Pig Latin script would work:

```
users = load 'users' as (name:chararray, city:chararray);
cinfo = load 'cityinfo' as (city:chararray, population:int);
jnd   = join cinfo by city, users by city using 'skewed';
```

Assume that the cities in *users* are distributed such that 20 users live in Barcelona, 100,000 in New York, and 350 in Portland. Let's further assume that Pig determined that it could fit 75,000 records into memory on each reducer. When this data was joined, New York would be identified as a key that needed to be split across reducers. During the join phase, all records with keys other than New York would be treated as in a default join. Records from *users* with New York as the key would be split between

‡ This algorithm was proposed in the paper "Practical Skew Handling in Parallel Joins," presented by David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri at the 18th International Conference on Very Large Databases.

two separate reducers. Records from *cityinfo* with New York as a key would be duplicated and sent to both of those reducers.

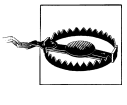
The second input in the join, in this case `users`, is the one that will be sampled and have its keys with a large number of values split across reducers. The first input will have records with those values replicated across reducers.

This algorithm addresses skew in only one input. If both inputs have skew, this algorithm will still work, but it will be slow. Much of the motivation behind this approach was that it guarantees the join will still finish, given time. Before Pig introduced skew join in version 0.4, data that was skewed on both sides could not be joined in Pig because it was not possible to fit all the records for the high-cardinality key values in memory for either side.

Skew join can be done on inner or outer joins. However, it can take only two join inputs. Multiway joins must be broken into a series of joins if they need to use skew join.

Since data often has skew, why not use skew join all of the time? There is a small performance penalty for using skew join, because one of the inputs must be sampled first to find any key values with a large number of records. This usually adds about 5% to the time it takes to calculate the join. If your data frequently has skew, it might be worth it to always use skew join and pay the 5% tax in order to avoid failing or running very slowly with the default join and then needing to rerun using skewed join.

As stated earlier, Pig estimates how much data it can fit into memory when deciding which key values to split and how wide to split them. For the purposes of this calculation, Pig looks at the record sizes in the sample and assumes it can use 30% of the JVM's heap to materialize records that will be joined. In your particular case you might find you need to increase or decrease this size. You should decrease the value if your join is still failing with out-of-memory errors even when using skew join. This indicates that Pig is estimating memory usage improperly, so you should tell it to use less. If profiling indicates that Pig is not utilizing all of your heap, you might want to increase the value in order to do the join more efficiently; the less ways the key values are split, the more efficient the join will be. You can do that by setting the property `pig.skewedjoin.reduce.memusage` to a value between 0 and 1. For example, if you wanted it to use 25% instead of 30%, you could add `-Dpig.skewedjoin.reduce.memusage=0.25` to your Pig command line or define the value in your properties file.



Like `order`, skew join breaks the MapReduce convention that all records with the same key will be processed by the same reducer. This means records with the same key might be placed in separate part files. If you plan to process the data in a way that depends on all records with the same key being in the same part file, you cannot use skew join.

Joining sorted data

A common database join strategy is to first sort both inputs on the join key and then walk through both inputs together, doing the join. This is referred to as a sort-merge join. In MapReduce, because a sort requires a full MapReduce job, as does Pig's default join, this technique is not more efficient than the default. However, if your inputs are already sorted on the join key, this approach makes sense. The join can be done in the map phase by opening both files and walking through them. Pig refers to this as a *merge join* because it is a sort-merge join, but the sort has already been done:

```
--mergejoin.pig
-- use sort_for_mergejoin.pig to build NYSE_daily_sorted and NYSE_dividends_sorted
daily = load 'NYSE_daily_sorted' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float,
            close:float, volume:int, adj_close:float);
divs  = load 'NYSE_dividends_sorted' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
jnd   = join daily by symbol, divs by symbol using 'merge';
```

To execute this join, Pig will first run a MapReduce job that samples the second input, *NYSE_dividends_sorted*. This sample builds an index that tells Pig the value of the join keys, `symbol` in the first record in every input split (usually each HDFS block). Because this sample reads only one record per split, it runs very quickly. Pig will then run a second MapReduce job that takes the first input, *NYSE_daily_sorted*, as its input. When each map reads the first record in its split of *NYSE_daily_sorted*, it takes the value of `symbol` and looks it up in the index built by the previous job. It looks for the last entry that is less than its value of `symbol`. It then opens *NYSE_dividends_sorted* at the corresponding block for that entry. For example, if the index contained entries `(CA, 1)`, `(CHY, 2)`, `(CP, 3)`, and the first `symbol` in a given map's input split of *NYSE_daily_sorted* was `CJA`, that map would open block 2 of *NYSE_dividends_sorted*. (Even if `CP` was the first user ID in *NYSE_daily_sorted*'s split, block 2 of *NYSE_dividends_sorted* would be opened, as there could be records with a key of `CP` in that block.) Once *NYSE_dividends_sorted* is opened, Pig throws away records until it reaches a record with `symbol` of `CJA`. Once it finds a match, it collects all the records with that value into memory and then does the join. It then advances the first input, *NYSE_daily_sorted*. If the key is the same, it again does the join. If not, it advances the second input, *NYSE_dividends_sorted*, again until it finds a value greater than or equal to the next value in the first input, *NYSE_daily_sorted*. If the value is greater, it advances the first input and continues. Because both inputs are sorted, it never needs to look in the index after the initial lookup.

All of this can be done without a reduce phase, and so it is more efficient than a default join. This algorithm, which was introduced in version 0.4, currently supports only two-way inner joins.

cogroup

`cogroup` is a generalization of `group`. Instead of collecting records of one input based on a key, it collects records of *n* inputs based on a key. The result is a record with a key and one bag for each input. Each bag contains all records from that input that have the given value for the key:

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

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

Another way to think of `cogroup` is as the first half of a join. The keys are collected together, but the cross product is not done. In fact, `cogroup` plus `foreach`, where each bag is flattened, is equivalent to a join—as long as there are no null values in the keys.

`cogroup` handles null values in the keys similarly to `group` and unlike `join`. That is, all records with a null value in the key will be collected together.

`cogroup` is useful when you want to do join-like things but not a full join. For example, Pig Latin does not have a semi-join operator, but you can do a semi-join:

```
--semijoin.pig
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float,
            close:float, volume:int, adj_close:float);
divs  = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
grpd  = cogroup daily by (exchange, symbol), divs by (exchange, symbol);
sjnd  = filter grpd by not IsEmpty(divs);
final = foreach sjnd generate flatten(daily);
```

Because `cogroup` needs to collect records with like keys together, it requires a reduce phase.

union

Sometimes you want to put two data sets together by concatenating them instead of joining them. Pig Latin provides `union` for this purpose. If you had two files you wanted to use for input and there was no glob that could describe them, you could do the following:

```
A = load '/user/me/data/files/input1';
B = load '/user/someoneelse/info/input2';
C = union A, B;
```

Unlike `union` in SQL, Pig does not require that both inputs share the same schema. If both do share the same schema, the output of the union will have that schema. If one schema can be produced from another by a set of implicit casts, the union will have that resulting schema. If neither of these conditions hold, the output will have no schema (that is, different records will have different fields). This schema comparison includes names, so even different field names will result in the output having no schema. You can get around this by placing a `foreach` before the union that renames fields.

```
A = load 'input1' as (x:int, y:float);
B = load 'input2' as (x:int, y:float);
C = union A, B;
describe C;

C: {x: int,y: float}

A = load 'input1' as (x:int, y:float);
B = load 'input2' as (x:int, y:double);
C = union A, B;
describe C;

C: {x: int,y: double}

A = load 'input1' as (x:int, y:float);
B = load 'input2' as (x:int, y:chararray);
C = union A, B;
describe C;

Schema for C unknown.
```

`union` does not perform a mathematical set union. That is, duplicate records are not eliminated. In this manner it is like SQL's `union all`. Also, `union` does not require a separate reduce phase.

Sometimes your data changes over time. If you have data you collect every month, you might add a new column this month. Now you are prevented from using `union` because your schemas do not match. If you want to union this data and force your data into a common schema, you can add the keyword `onschema` to your `union` statement:

```
A = load 'input1' as (w:chararray, x:int, y:float);
B = load 'input2' as (x:int, y:double, z:chararray);
C = union onschema A, B;
describe C;

C: {w: chararray,x: int,y: double,z: chararray}
```

`union onschema` requires that all inputs have schemas. It also requires that a shared schema for all inputs can be produced by adding fields and implicit casts. Matching of fields is done by name, not position. So, in the preceding example, `w:chararray` is added from *input1* and `z:chararray` is added from *input2*. Also, a cast from `float` to `double` is

added for *input1* so that field `y` is a `double`. If a shared schema cannot be produced by this method, an error is returned. When the data is read, nulls are inserted for fields not present in a given input.

cross

`cross` matches the mathematical set operation of the same name. In the following Pig Latin, `cross` takes every record in *NYSE_daily* and combines it with every record in *NYSE_dividends*:

```
--cross.pig
-- you may want to run this in a cluster, it produces about 3G of data
daily     = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
                date:chararray, open:float, high:float, low:float,
                close:float, volume:int, adj_close:float);
divs      = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                date:chararray, dividends:float);
tonsodata = cross daily, divs parallel 10;
```

`cross` tends to produce a lot of data. Given inputs with `n` and `m` records respectively, `cross` will produce output with `n x m` records.

Pig does implement `cross` in a parallel fashion. It does this by generating a synthetic join key, replicating rows, and then doing the cross as a join. The previous script is rewritten to:

```
daily     = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
                date:chararray, open:float, high:float, low:float,
                close:float, volume:int, adj_close:float);
divs      = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
                date:chararray, dividends:float);
A         = foreach daily generate flatten(GFCross(0, 2)), flatten(*);
B         = foreach divs generate flatten(GFCross(1, 2)), flatten(*);
C         = cogroup A by ($0, $1), B by ($0, $1) parallel 10;
tonsodata = foreach C generate flatten(A), flatten(B);
```

`GFCross` is an internal UDF. The first argument is the input number, and the second argument is the total number of inputs. In this example, the output is a bag that contains four records.§ These records have a schema of `(int, int)`. The field that is the same number as the first argument to `GFCross` contains a random number between zero and three. The other field counts from zero to three. So, if we assume for a given two records, one in each input, that the random number for the first input is `3` and for the second is `2`, then the outputs of `GFCross` would look like:

```
A {(3, 0), (3, 1), (3, 2), (3, 3)}
B {(0, 2), (1, 2), (2, 2), (3, 2)}
```

§ In 0.8 and earlier, the number of records is always 10. In 0.9, this is changed to be the square root of the parallel factor, rounded up.

When these records are flattened, four copies of each input record will be created in the map. They then are joined on the artificial keys. For every record in each input, it is guaranteed that there is one and only one instance of the artificial keys that will match and produce a record. Because the random numbers are chosen differently for each record, the resulting joins are done on an even distribution of the reducers.

This algorithm does enable crossing of data in parallel. However, it creates a burden on the shuffle phase by increasing the number of records in each input being shuffled. Also, no matter what you do, `cross` outputs a lot of data. Writing all of this data to disk is expensive, even when done in parallel.

This is not to say you should not use `cross`. There are instances when it is indispensable. Pig's `join` operator supports only equi-joins, that is, joins on an equality condition. Because general join implementations (ones that do not depend on the data being sorted or small enough to fit in memory) in MapReduce depend on collecting records with the same join key values onto the same reducer, non-equi-joins (also called *theta joins*) are difficult to do. They can be done in Pig using `cross` followed by `filter`:

```
--thetajoin.pig
--I recommend running this one on a cluster too
daily   = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float,
            close:float, volume:int, adj_close:float);
divs    = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
crossed = cross daily, divs;
tjnd    = filter crossed by daily::date < divs::date;
```

Fuzzy joins could also be done in this manner, where the fuzzy comparison is done after the cross. However, whenever possible, it is better to use a UDF to conform fuzzy values to a standard value and then do a regular join. For example, if you wanted to join two inputs on `city` but wanted to join any time two cities were in the same metropolitan area (e.g., you wanted "Los Angeles" and "Pasadena" to be viewed as equal), you could first run your records through a UDF that generated a single join key for all cities in a metropolitan area and then do the join.

Integrating Pig with Legacy Code and MapReduce

One tenet of Pig's philosophy is that Pig allows users to integrate their own code with Pig wherever possible (see "Pig Philosophy" on page 9). The most obvious way Pig does that is through its UDFs. But it also allows you to directly integrate other executables and MapReduce jobs.

stream

To specify an executable that you want to insert into your data flow, use `stream`. You may want to do this when you have a legacy program that you do not want to modify

or are unable to change. You can also use `stream` when you have a program you use frequently, or one you have tested on small data sets and now want to apply to a large data set. Let's look at an example where you have a Perl program *highdiv.pl* that filters out all stocks with a dividend below $1.00:

```
-- streamsimple.pig
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
highdivs = stream divs through `highdiv.pl` as (exchange, symbol, date, dividends);
```

Notice the `as` clause in the `stream` command. This is not required. But Pig has no idea what the executable will return, so if you do not provide the `as` clause, the relation `highdivs` will have no schema.

The executable *highdiv.pl* is invoked once on every map or reduce task. It is not invoked once per record. Pig instantiates the executable and keeps feeding data to it via *stdin*. It also keeps checking *stdout*, passing any results to the next operator in your data flow. The executable can choose whether to produce an output for every input, only every so many inputs, or only after all inputs have been received.

The preceding example assumes that you already have *highdiv.pl* installed on your grid, and that it is runnable from the working directory on the task machines. If that is not the case, which it usually will not be, you can ship the executable to the grid. To do this, use a `define` statement:

```
--streamship.pig
define hd `highdiv.pl` ship('highdiv.pl');
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
highdivs = stream divs through hd as (exchange, symbol, date, dividends);
```

This `define` does two things. First, it defines the executable that will be used. Now in `stream` we refer to *highdiv.pl* by the alias we gave it, `hp`, rather than referring to it directly. Second, it tells Pig to pick up the file *./highdiv.pl* and ship it to Hadoop as part of this job. This file will be picked up from the specified location on the machine where you launch the job. It will be placed in the working directory of the task on the task machines. So, the command you pass to `stream` must refer to it relative to the current working directory, not via an absolute path. If your executable depends on other modules or files, they can be specified as part of the `ship` clause as well. For example, if *highdiv.pl* depends on a Perl module called *Financial.pm*, you can send them both to the task machines:

```
define hd `highdiv.pl` ship('highdiv.pl', 'Financial.pm');
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
highdivs = stream divs through hd as (exchange, symbol, date, dividends);
```

Many scripting languages assume certain paths for modules based on their hierarchy. For example, Perl expects to find a module `Acme::Financial` in *Acme/Financial.pm*. However, the `ship` clause always puts files in your current working directory, and it does not take directories, so you could not ship *Acme*. The workaround for this is to create a TAR file and ship that, and then have a step in your executable that unbundles

the TAR file. You then need to set your module include path (for Perl, `-I` or the `PERL LIB` environment variables) to contain `.` (dot).

`ship` moves files into the grid from the machine where you are launching your job. But sometimes the file you want is already in the grid. If you have a grid file that will be accessed by every map or reduce task in your job, the proper way to access it is via the *distributed cache*. The distributed cache is a mechanism Hadoop provides to share files. It reduces the load on HDFS by preloading the file to the local disk on the machine that will be executing the task. You can use the distributed cache for your executable by using the `cache` clause in `define`:

```
crawl      = load 'webcrawl' as (url, pageid);
normalized = foreach crawl generate normalize(url);
define blc `blacklistchecker.py` cache('/data/shared/badurls#badurls');
goodurls   = stream normalized through blc as (url, pageid);
```

The string before the `#` is the path on HDFS, in this case, */data/shared/badurls*. The string after the `#` is the name of the file as viewed by the executable. So, Hadoop will put a copy of */data/shared/badurls* into the task's working directory and call it *badurls*.

So far we have assumed that your executable takes data on *stdin* and writes it to *stdout*. This might not work, depending on your executable. If your executable needs a file to read from, write to, or both, you can specify that with the `input` and `output` clauses in the `define` command. Continuing with our previous example, let's say that *blacklistchecker.py* expects to read its input from a file specified by `-i` on its command line and write to a file specified by `-o`:

```
crawl      = load 'webcrawl' as (url, pageid);
normalized = foreach crawl generate normalize(url);
define blc `blacklistchecker.py -i urls -o good` input('urls') output('good');
goodurls   = stream normalized through blc as (url, pageid);
```

Again, file locations are specified from the working directory on the task machines. In this example, Pig will write out all the input for a given task for *blacklistchecker.py* to *urls*, then invoke the executable, and then read *good* to get the results. Again, the executable will be invoked only once per map or reduce task, so Pig will first write out all the input to the file.

mapreduce

Beginning in Pig 0.8, you can also include MapReduce jobs directly in your data flow with the `mapreduce` command. This is convenient if you have processing that is better done in MapReduce than Pig but must be integrated with the rest of your Pig data flow. It can also make it easier to incorporate legacy processing written in MapReduce with newer processing you want to write in Pig Latin.

MapReduce jobs expect to read their input from and write their output to a storage device (usually HDFS). So to integrate them with your data flow, Pig first has to store the data, then invoke the MapReduce job, and then read the data back. This is done

via `store` and `load` clauses in the `mapreduce` statement that invoke regular load and store functions. You also provide Pig with the name of the JAR that contains the code for your MapReduce job.

As an example, let's continue with the blacklisting of URLs that we considered in the previous section. Only now let's assume that this is done by a MapReduce job instead of a Python script:

```
crawl      = load 'webcrawl' as (url, pageid);
normalized = foreach crawl generate normalize(url);
goodurls   = mapreduce 'blacklistchecker.jar'
                store normalized into 'input'
                load 'output' as (url, pageid);
```

`mapreduce` takes as its first argument the JAR containing the code to run a MapReduce job. It uses `load` and `store` phrases to specify how data will be moved from Pig's data pipeline to the MapReduce job. Notice that the input alias is contained in the `store` clause. As with `stream`, the output of `mapreduce` is opaque to Pig, so if we want the resulting relation `goodurls` to have a schema, we have to tell Pig what it is. This example also assumes that the Java code in *blacklistchecker.jar* knows which input and output files to look for and has a default class to run specified in its manifest. Often this will not be the case. Any arguments you wish to pass to the invocation of the Java command that will run the MapReduce task can be put in backquotes after the `load` clause:

```
crawl      = load 'webcrawl' as (url, pageid);
normalized = foreach crawl generate normalize(url);
goodurls   = mapreduce 'blacklistchecker.jar'
                store normalized into 'input'
                load 'output' as (url, pageid)
                `com.acmeweb.security.BlackListChecker -i input -o output`;
```

The string in the backquotes will be passed directly to your MapReduce job as is. So if you wanted to pass Java options, etc., you can do that as well.

The `load` and `store` clauses of the `mapreduce` command have the same syntax as the `load` and `store` statements, so you can use different load and store functions, pass constructor arguments, and so on. See "Load" on page 34 and "Store" on page 36 for full details.

Nonlinear Data Flows

So far our examples have been linear data flows or trees. In a linear data flow, one input is loaded, processed, and stored. We have looked at operators that combine multiple data flows: `join`, `cogroup`, `union`, and `cross`. With these you can build tree structures where multiple inputs all flow to a single output. But in complex data-processing situations, you often also want to split your data flow. That is, one input will result in more than one output. You might also have diamonds, places where the data flow is split and eventually joined back together. Pig supports these directed acyclic graph (DAG) data flows.

Splits in your data flow can be either implicit or explicit. In an implicit split, no specific operator or syntax is required in your script. You simply refer to a given relation multiple times. Let's consider data from our *baseball* example data. You might, for example, want to analyze players by position and by team at the same time:

```
--multiquery.pig
players    = load 'baseball' as (name:chararray, team:chararray,
                position:bag{t:(p:chararray)}, bat:map[]);
pwithba    = foreach players generate name, team, position,
                bat#'batting_average' as batavg;
byteam     = group pwithba by team;
avgbyteam  = foreach byteam generate group, AVG(pwithba.batavg);
store avgbyteam into 'by_team';
flattenpos = foreach pwithba generate name, team,
                flatten(position) as position, batavg;
bypos      = group flattenpos by position;
avgbypos   = foreach bypos generate group, AVG(flattenpos.batavg);
store avgbypos into 'by_position';
```

The `pwithba` relation is referred to by the `group` operators for both the `byteam` and `bypos` relations. Pig builds a data flow that takes every record from `pwithba` and ships it to both `group` operators.

Splitting data flows can also be done explicitly via the `split` operator, which allows you to split your data flow as many ways as you like. Let's take an example where you want to split data into different files depending on the date the record was created:

```
wlogs = load 'weblogs' as (pageid, url, timestamp);
split wlogs into apr03 if timestamp < '20110404',
          apr02 if timestamp < '20110403' and timestamp > '20110401',
          apr01 if timestamp < '20110402' and timestamp > '20110331';
store apr03 into '20110403';
store apr02 into '20110402';
store apr01 into '20110401';
```

At first glance, `split` looks like a `switch` or `case` statement, *but it is not*. A single record can go to multiple legs of the split since you use different filters for each `if` clause. And a record can go to no leg. In the preceding example, if a record were found with a date of `20110331`, it would be dropped. And there is no default clause—no way to send any leftover records to a particular alias.

`split` is semantically identical to an implicit split that users filters. The previous example could be rewritten as:

```
wlogs = load 'weblogs' as (pageid, url, timestamp);
apr03 = filter wlogs by timestamp < '20110404';
apr02 = filter wlogs by timestamp < '20110403' and timestamp > '20110401';
apr01 = filter wlogs by timestamp < '20110402' and timestamp > '20110331';
store apr03 into '20110403';
store apr02 into '20110402';
store apr01 into '20110401';
```

In fact, Pig will internally rewrite the original script that has `split` in exactly this way.

Let's take a look at how Pig executes these nonlinear data flows. Whenever possible, it combines them into single MapReduce jobs. This is referred to as a *multiquery*. In cases where all operators will fit into a single map task, this is easy. Pig creates separate pipelines inside the map and sends the appropriate records to each pipeline. The example using `split` to store data by date will be executed in this way.

Pig can also combine multiple `group` operators together in many cases. In the example given at the beginning of this section, where the baseball data is grouped by both team and position, this entire Pig Latin script will be executed inside one MapReduce job. Pig accomplishes this by duplicating records on the map side and annotating each record with its pipeline number. When the data is partitioned during the shuffle, the appropriate key is used for each record. That is, records from the pipeline grouping by `team` will use `team` as their shuffle key, and records from the pipeline grouping by `position` will use `position` as their shuffle key. This is done by declaring the key type to be `tuple` and placing the correct values in the key tuple for each record. Once the data has been collected to reducers, the pipeline number is used as part of the sort key so that records from each pipeline and group are collected together. In the reduce task, Pig instantiates multiple pipelines, one for each group operator. It sends each record down the appropriate pipeline based on its annotated pipeline number. In this way, input data can be scanned once but grouped many different ways. An example of how one record flows through this pipeline is shown in Figure 6-1. Although this does not provide linear speedup, we find it often approaches it.

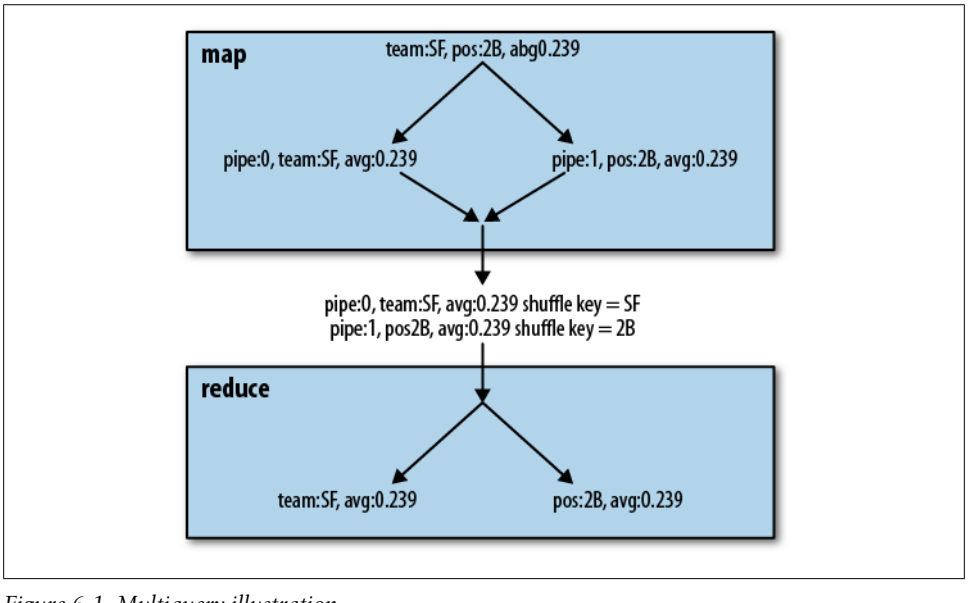


Figure 6-1. Multiquery illustration

There are cases where Pig will not combine multiple operators into a single MapReduce job. Pig does not use multiquery for any of the multiple-input operators: `join`, `union`,

`cross`, or `cogroup`. It does not use multiquery for `order` statements either. Also, if it has multiple `group` statements and some would use Hadoop's combiner and some would not, it combines only those statements that use Hadoop's combiner into a multiquery. This is because we have found that combining the Hadoop combiner and non-Hadoop combiner jobs together does not perform well.

Multiquery scripts tend to perform better than loading the same input multiple times, but this approach does have limits. Because it requires replicating records in the map, it does slow down the shuffle phase. Eventually the increased cost of the shuffle phase outweighs the reduced cost of rescanning the input data. Pig has no way to estimate when this will occur. Currently, the optimizer is optimistic and always combines jobs with multiquery whenever it can. If it combines too many jobs and becomes slower than splitting some of the jobs, you can turn off multiquery or you can rewrite your Pig Latin into separate scripts so Pig does not attempt to combine them all. To turn off multiquery, you can pass either `-M` or `-no_multiquery` on the command line or set the property `opt.multiquery` to `false`.

We must also consider what happens when one job in a multiquery fails but others succeed. If all jobs succeed, Pig will return 0, meaning success. If all of the jobs fail, Pig will return 2. If some jobs fail and some succeed, Pig will return 3. By default, if one of the jobs fails, Pig will continue processing the other jobs. However, if you want Pig to stop as soon as one of the jobs fails, you can pass `-F` or `-stop_on_failure`. In this case, any jobs that have not yet been finished will be terminated, and any that have not started will not be started. Any jobs that are already finished will not be cleaned up.

Controlling Execution

In addition to providing many relational and dataflow operators, Pig Latin provides ways for you to control how your jobs execute on MapReduce. It allows you to set values that control your environment and details of MapReduce, such as how your data is partitioned.

set

The `set` command is used to set the environment in which Pig runs the MapReduce jobs. Table 6-1 shows Pig-specific parameters that can be controlled via `set`.

Table 6-1. Pig-specific set parameters

Parameter	Value type	Description
debug	string	Sets the logging level to DEBUG. Equivalent to passing `-debug DEBUG` on the command line.
`default_parallel`	integer	Sets a default parallel level for all reduce operations in the script. See "Parallel" on page 49 for details.

Parameter	Value type	Description
job.name	string	Assigns a name to the Hadoop job. By default the name is the filename of the script being run, or a randomly generated name for interactive sessions.
job.priority	string	If your Hadoop cluster is using the Capacity Scheduler with priorities enabled for queues, this allows you to set the priority of your Pig job. Allowed values are very_low, low, normal, high, and very_high.

For example, to set the default parallelism of your Pig Latin script and set the job name to my_job:

```
set default_parallel 10;
set job.name my_job;
users = load 'users';
```

In addition to these predefined values, set can be used to pass Java property settings to Pig and Hadoop. Both Pig and Hadoop use a number of Java properties to control their behavior. Consider an example where you want to turn multiquery off for a given script, and you want to tell Hadoop to use a higher value than usual for its map-side sort buffer:

```
set opt.multiquery false;
set io.sort.mb 2048; --give it 2G
```

You can also use this mechanism to pass properties to UDFs. All of the properties are passed to the tasks on the Hadoop nodes when they are executed. They are not set as Java properties in that environment; rather, they are placed in a Hadoop object called JobConf. UDFs have access to the JobConf. Thus, anything you set in the script can be seen by your UDFs. This can be a convenient way to control UDF behavior. For information on how to retrieve this information in your UDFs, see "Constructors and Passing Data from Frontend to Backend" on page 128.

Values that are set in your script are global for the whole script. If they are reset later in the script, that second value will overwrite the first and be used *throughout the whole script*.

Setting the Partitioner

Hadoop uses a class called Partitioner to partition records to reducers during the shuffle phase. For details on partitioners, see "Shuffle Phase" on page 191. Pig does not override the default partitioner, except for order and skew join. The balancing operations in these require special Partitioners.

Beginning in version 0.8, Pig allows you to set the partitioner, except in the cases where it is already overriding it. To do this, you need to tell Pig which Java class to use to partition your data. This class must extend Hadoop's org.apache.hadoop.mapreduce.Partitioner<KEY,VALUE>. Note that this is the newer (version 0.20 and later) mapreduce API and not the older mapred:

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

Operators that reduce data can take the `partition` clause. These operators are `cogroup`, `cross`, `distinct`, `group`, and `join` (again, not in conjunction with skew join).

Pig Latin Preprocessor

Pig Latin has a preprocessor that runs before your Pig Latin script is parsed. In 0.8 and earlier, this provided parameter substitution, roughly similar to a very simple version of `#define` in C. Starting with 0.9, it also provides inclusion of other Pig Latin scripts and function-like macro definitions, so that you can write Pig Latin in a modular way.

Parameter Substitution

Pig Latin scripts that are used frequently often have elements that need to change based on when or where they are run. A script that is run every day is likely to have a date component in its input files or filters. Rather than edit and change the script every day, you want to pass in the date as a parameter. *Parameter substitution* provides this capability with a basic string-replacement functionality. Parameters must start with a letter or an underscore and can then have any amount of letters, numbers, or underscores. Values for the parameters can be passed in on the command line or from a parameter file:

```
--daily.pig
daily     = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
                date:chararray, open:float, high:float, low:float, close:float,
                volume:int, adj_close:float);
yesterday = filter daily by date == '$DATE';
grpd      = group yesterday all;
minmax    = foreach grpd generate MAX(yesterday.high), MIN(yesterday.low);
```

When you run *daily.pig*, you must provide a definition for the parameter `DATE`; otherwise, you will get an error telling you that you have undefined parameters:

```
pig -p DATE=2009-12-17 daily.pig
```

You can repeat the `-p` command-line switch as many times as needed. Parameters can also be placed in a file, which is convenient if you have more than a few of them. The format of the file is *`parameter=value`*, one per line. Comments in the file should be preceded by a `#`. You then indicate the file to be used with `-m` or `-param_file`:

```
pig -param_file daily.params daily.pig
```

Parameters passed on the command line take precedence over parameters provided in files. This way, you can provide all your standard parameters in a file and override a few as needed on the command line.

Parameters can contain other parameters. So, for example, you could have the following parameter file:

```
#Param file
YEAR=2009-
MONTH=12-
DAY=17
DATE=$YEAR$MONTH$DAY
```

A parameter must be defined before it is referenced. The parameter file here would produce an error if the `DAY` line came after the `DATE` line. The other caveat is that there is no special character to delimit the end of a parameter. Any alphanumeric or underscore character will be interpreted as part of the parameter, and any other character will be interpreted as itself. So, if you had a script that ran at the first of every month, you could not do the following:

```
wlogs = load 'clicks/$YEAR$MONTH01' as (url, pageid, timestamp);
```

This would try to resolve a parameter `MONTH01` when you meant `MONTH`.

When using parameter substitution, all parameters in your script must be resolved after the preprocessor is finished. If not, Pig will issue an error message and not continue. You can see the results of your parameter substitution by using the `-dryrun` flag on the Pig command line. Pig will write out a version of your Pig Latin script with the parameter substitution done, but it will not execute the script.

You can also define parameters inside your Pig Latin script using `%declare` and `%default`. `%declare` allows you to define a parameter in the script itself. `%default` is useful to provide a common default value that can be overridden when needed. Consider a case where most of the time your script is run on one Hadoop cluster, but occasionally it is run on a different cluster with different hardware:

```
%default parallel_factor 10;
wlogs = load 'clicks' as (url, pageid, timestamp);
grp   = group wlogs by pageid parallel $parallel_factor;
cntd  = foreach grp generate group, COUNT(wlogs);
```

When running your script in the usual configuration, there is no need to set the parameter `parallel_factor`. On the occasions it is run in a different setup, the parallel factor can be changed by passing a value on the command line.

Macros

Starting in 0.9, Pig added the ability to define macros. This makes it possible to make your Pig Latin scripts modular. It also makes it possible to share segments of Pig Latin code among users. This can be particularly useful for defining standard practices and making sure all data producers and consumers use them.

Macros are declared with the `define` statement. A macro takes a set of input parameters, which are string values that will be substituted for the parameters when the macro is expanded. By convention, input relation names are placed first before other parameters.

The output relation name is given in a `returns` statement. The operators of the macro are enclosed in {} (braces). Anywhere the parameters—including the output relation name—are referenced inside the macro, they must be preceded by a $ (dollar sign). The macro is then invoked in your Pig Latin by assigning it to a relation:

```
--macro.pig
-- Given daily input and a particular year, analyze how
-- stock prices changed on days dividends were paid out.
define dividend_analysis (daily, year, daily_symbol, daily_open, daily_close)
returns analyzed {
    divs          = load 'NYSE_dividends' as (exchange:chararray,
                         symbol:chararray, date:chararray, dividends:float);
    divsthisyear  = filter divs by date matches '$year-.*';
    dailythisyear = filter $daily by date matches '$year-.*';
    jnd           = join divsthisyear by symbol, dailythisyear by $daily_symbol;
    $analyzed     = foreach jnd generate dailythisyear::$daily_symbol,
                         $daily_close - $daily_open;
};

daily   = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float, close:float,
            volume:int, adj_close:float);
results = dividend_analysis(daily, '2009', 'symbol', 'open', 'close');
```

It is also possible to have a macro that does not return a relation. In this case, the `returns` clause of the `define` statement is changed to `returns void`. This can be useful when you want to define a macro that controls how data is partitioned and sorted before being stored to a particular output, such as HBase or a database.

These macros are expanded inline. This is where an important difference between macros and functions becomes apparent. Macros cannot be invoked recursively. Macros can invoke other macros, so a macro `A` can invoke a macro `B`, but `A` cannot invoke itself. And once `A` has invoked `B`, `B` cannot invoke `A`. Pig will detect these loops and throw an error.

Parameter substitution (see "Parameter Substitution" on page 77) cannot be used inside of macros. Parameters should be passed explicitly to macros, and parameter substitution should be used only at the top level.

You can use the `-dryrun` command-line argument to see how the macros are expanded inline. When the macros are expanded, the alias names are changed to avoid collisions with alias names in the place the macro is being expanded. If we take the previous example and use `-dryrun` to show us the resulting Pig Latin, we will see the following (reformatted slightly to fit on the page):

```
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float, close:float,
            volume:int, adj_close:float);
macro_dividend_analysis_divs_0 = load 'NYSE_dividends' as (exchange:chararray,
            symbol:chararray, date:chararray, dividends:float);
macro_dividend_analysis_divsthisyear_0 =
            filter macro_dividend_analysis_divs_0 BY (date matches '2009-.*');
```

```
macro_dividend_analysis_dailythisyear_0 = filter daily BY (date matches '2009-.*');
macro_dividend_analysis_jnd_0 =
            join macro_dividend_analysis_divsthisyear_0 by (symbol),
            macro_dividend_analysis_dailythisyear_0 by (symbol);
results = foreach macro_dividend_analysis_jnd_0 generate
            macro_dividend_analysis_dailythisyear_0::symbol, close - open;
```

As you can see, the aliases in the macro are expanded with a combination of the macro name and the invocation number. This provides a unique key so that if other macros use the same aliases, or the same macro is used multiple times, there is still no duplication.

Including Other Pig Latin Scripts

For a long time in Pig Latin, the entire script needed to be in one file. This produced some rather unpleasant multithousand-line Pig Latin scripts. Starting in 0.9, the preprocessor can be used to include one Pig Latin script in another. Taken together with the macros (also added in 0.9; see "Macros" on page 78), it is now possible to write modular Pig Latin that is easier to debug and reuse.

`import` is used to include one Pig Latin script in another:

```
--main.pig
import '../examples/ch6/dividend_analysis.pig';

daily   = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float, close:float,
            volume:int, adj_close:float);
results = dividend_analysis(daily, '2009', 'symbol', 'open', 'close');
```

`import` writes the imported file directly into your Pig Latin script in place of the `import` statement. In the preceding example, the contents of *dividend_analysis.pig* will be placed immediately before the `load` statement. Note that a file cannot be imported twice. If you wish to use the same functionality multiple times, you should write it as a macro and import the file with that macro.

In the example just shown, we used a relative path for the file to be included. Fully qualified paths also can be used. By default, relative paths are taken from the current working directory of Pig when you launch the script. You can set a search path by setting the `pig.import.search.path` property. This is a comma-separated list of paths that will be searched for your files. The current working directory, . (dot), is always in the search path:

```
set pig.import.search.path '/usr/local/pig,/grid/pig';
import 'acme/macros.pig';
```

Imported files are not in separate namespaces. This means that all macros are in the same namespace, even when they have been imported from separate files. Thus, care should be taken to choose unique names for your macros.

CHAPTER 7

Developing and Testing Pig Latin Scripts

The last few chapters focused on Pig Latin the language. Now we will turn to the practical matters of developing and testing your scripts. This chapter covers helpful debugging tools such as `describe` and `explain`. It also covers ways to test your scripts. Information on how to make your scripts perform better will be covered in the next chapter.

Development Tools

Pig provides several tools and diagnostic operators to help you develop your applications. In this section we will explore these and also look at some tools others have written to make it easier to develop Pig with standard editors and integrated development environments (IDEs).

Syntax Highlighting and Checking

Syntax highlighting often helps users write code correctly, at least syntactically, the first time around. Syntax highlighting packages exist for several popular editors. The packages listed in Table 7-1 were created and added at various times, so how their highlighting conforms with current Pig Latin syntax varies.

Table 7-1. Pig Latin syntax highlighting packages

Tool	URL
Eclipse	*http://code.google.com/p/pig-eclipse*
Emacs	*http://github.com/cloudera/piglatin-mode*, *http://sf.net/projects/pig-mode*
TextMate	*http://www.github.com/kevinweil/pig.tmbundle*
Vim	*http://www.vim.org/scripts/script.php?script_id=2186*

In addition to these syntax highlighting packages, Pig will also let you check the syntax of your script without running it. If you add `-c` or `-check` to the command line, Pig will just parse and run semantic checks on your script. The `-dryrun` command-line option will also check your syntax, expand any macros and `import`s, and perform parameter substitution.

describe

`describe` shows you the schema of a relation in your script. This can be very helpful as you are developing your scripts. It is especially useful as you are learning Pig Latin and understanding how various operators change the data. `describe` can be applied to any relation in your script, and you can have multiple `describe`s in a script:

```
--describe.pig
divs    = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
trimmed = foreach divs generate symbol, dividends;
grpd    = group trimmed by symbol;
avgdiv  = foreach grpd generate group, AVG(trimmed.dividends);

describe trimmed;
describe grpd;
describe avgdiv;

trimmed: {symbol: chararray,dividends: float}
grpd: {group: chararray,trimmed: {(symbol: chararray,dividends: float)}}
avgdiv: {group: chararray,double}
```

`describe` uses Pig's standard schema syntax. For information on this syntax, see "Schemas" on page 27. So, in this example, the relation `trimmed` has two fields: `symbol`, which is a chararray, and `dividends`, which is a float. `grpd` also has two fields, `group` (the name Pig always assigns to the group by key) and a bag `trimmed`, which matches the name of the relation that Pig grouped to produce the bag. Tuples in `trimmed` have two fields: `symbol` and `dividends`. Finally, in `avgdiv` there are two fields, `group` and a double, which is the result of the `AVG` function and is unnamed.

explain

One of Pig's goals is to allow you to think in terms of data flow instead of MapReduce. But sometimes you need to peek into the barn and see how Pig is compiling your script into MapReduce jobs. Pig provides `explain` for this. `explain` is particularly helpful when you are trying to optimize your scripts or debug errors. It was written so that Pig developers could examine how Pig handled various scripts, thus its output is not the most user-friendly. But with some effort, `explain` can help you write better Pig Latin.

There are two ways to use `explain`. You can `explain` any alias in your Pig Latin script, which will show the execution plan Pig would use if you stored that relation. You can also take an existing Pig Latin script and apply `explain` to the whole script in Grunt.

This has a couple of advantages. One, you do not have to edit your script to add the `explain` line. Two, it will work with scripts that do not have a single store, showing how Pig will execute the entire script:

```
--explain.pig
divs   = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
grpd   = group divs by symbol;
avgdiv = foreach grpd generate group, AVG(divs.dividends);
store avgdiv into 'average_dividend';

bin/pig -x local -e 'explain -script explain.pig'
```

This will produce a printout of several graphs in text format; we will examine this output momentarily. When using `explain` on a script in Grunt, you can also have it print out the plan in graphical format. To do this, add `-dot -out` *`filename`* to the preceding command line. This prints out a file in DOT language containing diagrams explaining how your script will be executed. Tools that can read this language and produce graphs can then be used to view the graphs. For some tools, you might need to split the three graphs in the file into separate files.

Pig goes through several steps to transform a Pig Latin script to a set of MapReduce jobs. After doing basic parsing and semantic checking, it produces a *logical plan*. This plan describes the logical operators that Pig will use to execute the script. Some optimizations are done on this plan. For example, filters are pushed as far up* as possible in the logical plan. The logical plan for the preceding example is shown in Figure 7-1. I have trimmed a few extraneous pieces to make the output more readable (scary that this is more readable, huh?). If you are using Pig 0.9, the output will look slightly different, but close enough that it will be recognizable.

The flow of this chart is bottom to top so that the `Load` operator is at the very bottom. The lines between operators show the flow. Each of the four operators created by the script (`Load`, `CoGroup`, `ForEach`, and `Store`) can be seen. Each of these operators also has a schema, described in standard schema syntax. The `CoGroup` and `ForEach` operators also have expressions attached to them (the lines dropping down from those operators). In the `CoGroup` operator, the projection indicates which field is the grouping key (in this case, field 1). The `ForEach` operator has a projection expression that projects field 0 (the group field) and a UDF expression, which indicates that the UDF being used is `org.apache.pig.builtin.AVG`. Notice how each of the `Project` operators has an `Input` field, indicating from which operator they are drawing their input. Figure 7-2 shows how this plan looks when the `-dot` option is used instead.

* Or down, whichever you prefer. Database textbooks usually talk of pushing filters down, closer to the scan. Because Pig Latin scripts start with a `load` at the top and go down, we tend to refer to it as pushing filters up toward the `load`.

```
avgdiv: Store 1-27 Schema: {group: bytearray,double} Type: Unknown
|
|---avgdiv: ForEach 1-26 Schema: {group: bytearray,double} Type: bag
    |   |
    |   Project 1-22 Projections: [0]
    |   FieldSchema: group: bytearray Type: bytearray
    |   Input: grpd: CoGroup 1-16
    |   |
    |   UserFunc 1-25 function: org.apache.pig.builtin.AVG
    |   FieldSchema: double Type: double
    |   |
    |   |---Project 1-23 Projections: [3]
    |       FieldSchema: dividends: bag({dividends: bytearray}) Type: bag
    |       Input: Project 1-24 Projections: [1] Overloaded: false|
    |       |---Project 1-24 Projections: [1]
    |           FieldSchema: divs: bag({exchange: bytearray,symbol: bytearray,
    |               date: bytearray,dividends: bytearray}) Type: bag
    |           Input: grpd: CoGroup 1-16
    |
    |---grpd: CoGroup 1-16 Schema: {group: bytearray,divs: {exchange: bytearray,
        |   |   symbol: bytearray,date: bytearray,dividends: bytearray}} Type: bag
        |   |
        |   Project 1-15 Projections: [1]
        |   FieldSchema: symbol: bytearray Type: bytearray
        |   Input: divs: Load 1-14
        |
        |---divs: Load 1-14 Schema: {exchange: bytearray,symbol: bytearray,
            date: bytearray,dividends: bytearray} Type: bag
```

Figure 7-1. Logical plan

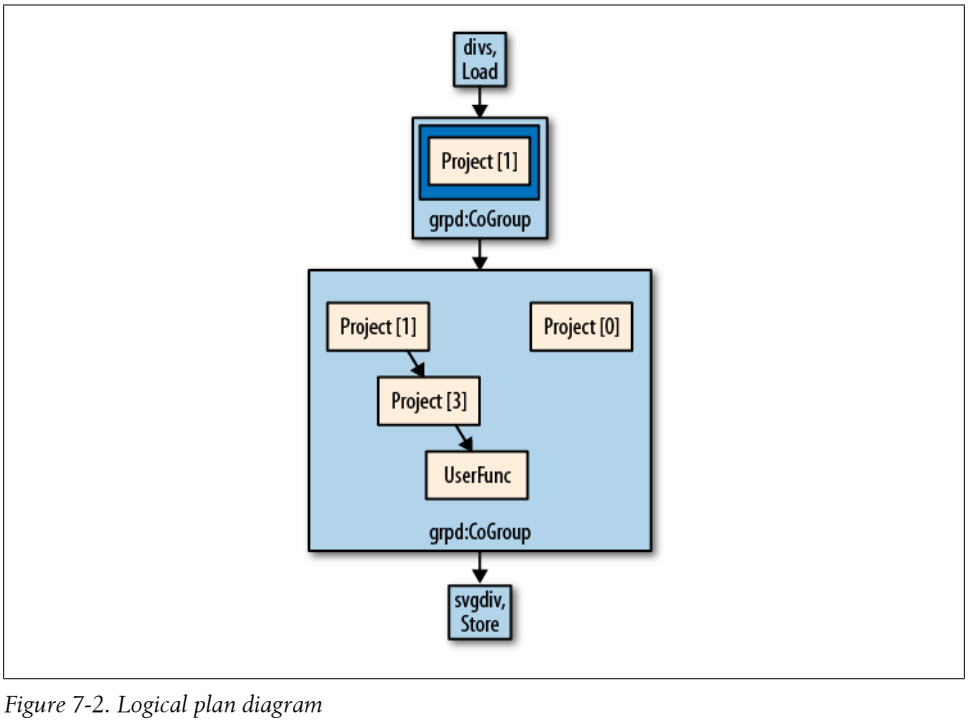


Figure 7-2. Logical plan diagram

After optimizing the logical plan, Pig produces a physical plan. This plan describes the physical operators Pig will use to execute the script, without reference to how they will be executed in MapReduce. The physical plan for our plan in Figure 7-1 is shown in Figure 7-3.

```
avgdiv: Store(file:///home/gates/git/programmingpig/data/average_dividend:
|       org.apache.pig.builtin.PigStorage)
|
|---avgdiv: New For Each(false,false)[bag]
    |   |
    |   Project[bytearray][0]
    |   |
    |   POUserFunc(org.apache.pig.builtin.AVG)[double]
    |   |
    |   |---Project[bag][3]
    |       |
    |       |---Project[bag][1]
    |
    |---grpd: Package[tuple]{bytearray}
        |
        |---grpd: Global Rearrange[tuple]
            |
            |---grpd: Local Rearrange[tuple]{bytearray}(false)
                |   |
                |   Project[bytearray][1]
                |
                |---divs: Load(file:///home/gates/git/programmingpig/data/
                    NYSE_dividends:org.apache.pig.builtin.PigStorage)
```

Figure 7-3. Physical plan

This looks like the logical plan, but with a few differences. The load and store functions that will be used have been resolved (in this case to `org.apache.pig.builtin.PigStorage`, the default load and store function), and the actual paths that will be used have been resolved. This example was run in local mode, so the paths are local files. If it had been run on a cluster, it would have showed a path like *hdfs://nn.machine.domain/filepath*.

The other noticeable difference is that the `CoGroup` operator was replaced by three operators, `Local Rearrange`, `Global Rearrange`, and `Package`. `Local Rearrange` is the operator Pig uses to prepare data for the shuffle by setting up the key. `Global Rearrange` is a stand-in for the shuffle. `Package` sits in the reduce phase and directs records to the proper bag. Figure 7-4 shows a graphical representation of this plan.

Finally, Pig takes the physical plan and decides how it will place its operators into one or more MapReduce jobs. First, it walks the physical plan looking for all operators that require a new reduce. This occurs anywhere there is a `Local Rearrange`, `Global Rearrange`, and `Package`. After it has done this, it sees whether there are places that it can do physical optimizations. For example, it looks for places the combiner can be used, and whether sorts can be avoided by including them as part of the sorting Hadoop does in the shuffle. After all of this is done, Pig has a MapReduce plan. This plan describes the

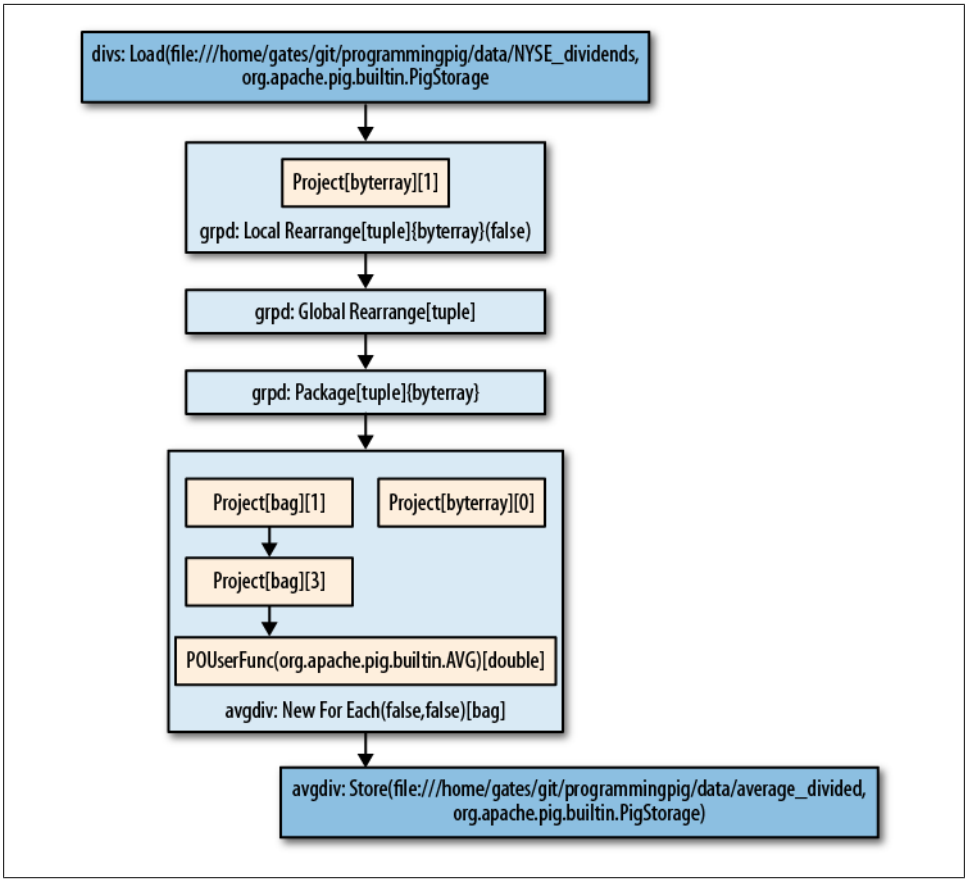


Figure 7-4. Physical plan diagram

maps, combines, and reduces, along with the physical operations Pig will perform in each stage. Completing our example, the MapReduce plan is shown in Figure 7-5.

This looks much the same as the physical plan. The pipeline is now broken into three stages: map, combine, and reduce. The `Global Rearrange` operator is gone because it was a stand-in for the shuffle. The `AVG` UDF has been broken up into three stages: `Initial` in the map, `Intermediate` in the combiner, and `Final` in the reduce. If there were multiple MapReduce jobs in this example, they would all be shown in this output. The graphical version is shown in Figure 7-6.

```
MapReduce node
Map Plan
grpd: Local Rearrange[tuple]{bytearray}(false)
|   |
|   Project[bytearray][0]
|
|---avgdiv: New For Each(false,false)[bag]
    |   |
    |   Project[bytearray][0]
    |   |
    |   POUserFunc(org.apache.pig.builtin.AVG$Initial)[tuple]
    |   |
    |   |---Project[bag][3]
    |       |
    |       |---Project[bag][1]
    |
    |---Pre Combiner Local Rearrange[tuple]{Unknown}
        |
        |---divs: Load(file:///home/gates/git/programmingpig/data/
                      NYSE_dividends:org.apache.pig.builtin.PigStorage)


Combine Plan
grpd: Local Rearrange[tuple]{bytearray}(false)
|   |
|   Project[bytearray][0]
|
|---avgdiv: New For Each(false,false)[bag]
    |   |
    |   Project[bytearray][0]
    |   |
    |   POUserFunc(org.apache.pig.builtin.AVG$Intermediate)[tuple]
    |   |
    |   |---Project[bag][1]
    |
    |---POCombinerPackage[tuple]{bytearray}


Reduce Plan
avgdiv: Store(file:///home/gates/git/programmingpig/data/average_dividend:
|         org.apache.pig.builtin.PigStorage)
|
|---avgdiv: New For Each(false,false)[bag]
    |   |
    |   Project[bytearray][0]
    |   |
    |   POUserFunc(org.apache.pig.builtin.AVG$Final)[double]
    |   |
    |   |---Project[bag][1]
    |
    |---POCombinerPackage[tuple]{bytearray}
Global sort: false
```

Figure 7-5. MapReduce plan