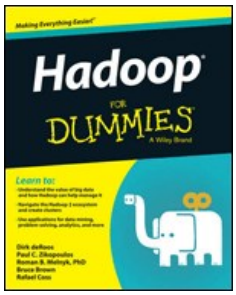


# Hadoop for Dummies

by Dirk deRoos et al.

# Chapter 8: Pig—Hadoop Programming Made Easier

## In This Chapter

- Looking at the Pig architecture
- Seeing the flow in the Pig Latin application flow
- Reciting the ABCs of Pig Latin
- Distinguishing between local and distributed modes of running Pig scripts
- Scripting with Pig Latin

Java MapReduce programs (see Chapter 6) and the Hadoop Distributed File System (HDFS; see Chapter 4) provide you with a powerful distributed computing framework, but they come with one major drawback — relying on them limits the use of Hadoop to Java programmers who can think in Map and Reduce terms when writing programs. More developers, data analysts, data scientists, and all-around good folks could leverage Hadoop if they had a way to harness the power of Map and Reduce while hiding some of the Map and Reduce complexities.

As with most things in life, where there's a need, somebody is bound to come up with an idea meant to fill that need. A growing list of MapReduce *abstractions* is now on the market — programming languages and/or tools such as Hive and Pig, which hide the messy details of MapReduce so that a programmer can concentrate on the important work.

Hive, for example, provides a limited SQL-like capability that runs over MapReduce, thus making said MapReduce more approachable for SQL developers. Hive also provides a *declarative* query language (the SQL-like HiveQL), which allows you to focus on *which* operation you need to carry out versus *how* it is carried out.

Though SQL is the common accepted language for querying structured data, some developers still prefer writing *imperative* scripts — scripts that define a set of operations that change the state of the data — and also want to have more data processing flexibility than what SQL or HiveQL provides. Again, this need led the engineers at Yahoo! Research to come up with a product meant to fulfill that need — and so Pig was born. Pig's claim to fame was its status as a programming tool attempting to have the best of both worlds: a declarative query language inspired by SQL and a low-level procedural programming language that can generate MapReduce code. This lowers the bar when it comes to the level of technical knowledge needed to exploit the power of Hadoop.

By taking a look at some murky computer programming language history, we can say that Pig was initially developed at Yahoo! in 2006 as part of a research project tasked with coming up with ways for people using Hadoop to focus more on analyzing large data sets rather than spending lots of time writing Java MapReduce programs. The goal here was a familiar one: Allow users to focus more on what they want to do and less on how it's done. Not long after, in 2007, Pig officially became an Apache project. As such, it is included in most Hadoop distributions.

And its name? That one's easy to figure out. The Pig programming language is designed to handle any kind of data tossed its way — structured, semi-structured, unstructured data, you name it. Pigs, of course, have a reputation for eating anything they come across. (We suppose they could have called it Goat — or maybe that name was already taken.) According to the Apache Pig philosophy, pigs eat anything, live anywhere, are domesticated and can fly to boot. (Flying Apache Pigs? Now we've seen everything.) Pigs "living anywhere" refers to the fact that Pig is a parallel data processing programming language and is not committed to any particular parallel framework — including Hadoop. What makes it a domesticated animal? Well, if "domesticated" means "plays well with humans," then it's definitely the case that Pig prides itself on being easy for humans to code and maintain. (Hey, it's easily integrated with other programming languages and it's extensible. What more could you ask?) Lastly, Pig is smart and in data processing lingo this means there is an optimizer that figures out how to do the hard work of figuring out how to get the data quickly. Pig is not just going to be quick — it's going to fly. (To see more about the Apache Pig philosophy, check out http://pig.apache.org/philosophy.)

## Admiring the Pig Architecture

"Simple" often means "elegant" when it comes to those architectural drawings for that new Silicon Valley mansion you have planned for when the money starts rolling in after you implement Hadoop. The same principle applies to software architecture. Pig is made up of two (count ‘em, two’ components:

- **The language itself**: As proof that programmers have a sense of humor, the programming language for Pig is known as Pig Latin, a high-level language that allows you to write data processing and analysis programs.
- **The Pig Latin compiler**: The Pig Latin compiler converts the Pig Latin code into executable code. The executable code is either in the form of MapReduce jobs or it can spawn a process where a virtual Hadoop instance is created to run the Pig code on a single node.

  **REMEMBER** The sequence of MapReduce programs enables Pig programs to do data processing and analysis in parallel, leveraging Hadoop MapReduce and HDFS. Running the Pig job in the virtual Hadoop instance is a useful strategy for testing your Pig scripts.

Figure 8-1 shows how Pig relates to the Hadoop ecosystem.

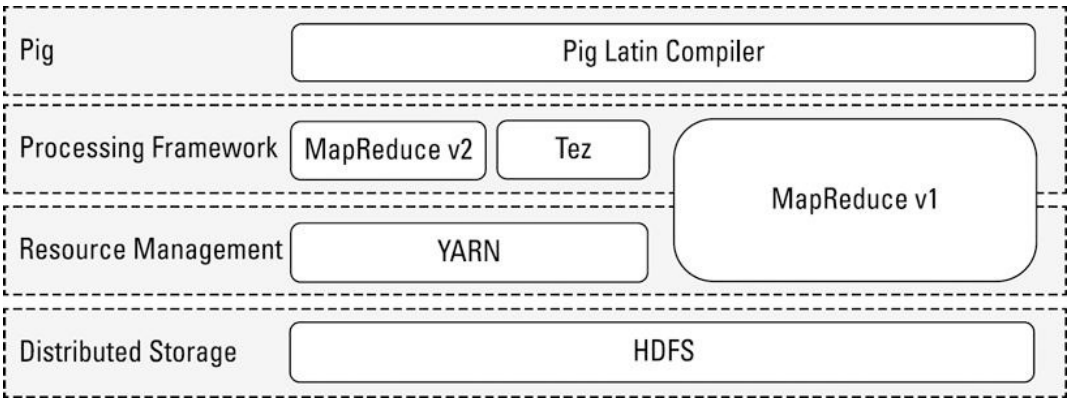


**Figure 8-1:** Pig archi-tecture

**TECHNICAL STUFF** Pig programs can run on MapReduce v1 or MapReduce v2 without any code changes, regardless of what mode your cluster is running. However, Pig scripts can also run using the Tez API instead. Apache Tez provides a more efficient execution framework than MapReduce. YARN enables application frameworks other than MapReduce (like Tez) to run on Hadoop. Hive can also run against the Tez framework. See Chapter 7 for more information on YARN and Tez.

## Going with the Pig Latin Application Flow

At its core, Pig Latin is a *dataflow* language, where you define a data stream and a series of transformations that are applied to the data as it flows through your application. This is in contrast to a *control flow* language (like C or Java), where you write a series of instructions. In control flow languages, we use constructs like loops and conditional logic (like an `if` statement). You won't find loops and `if` statements in Pig Latin.

If you need some convincing that working with Pig is a significantly easier row to hoe than having to write Map and Reduce programs, start by taking a look at some real Pig syntax:

**Listing 8-1: Sample Pig Code to illustrate the data processing dataflow**

```
A = LOAD 'data_file.txt';
...
B = GROUP ... ;
...
C= FILTER ...;
...
DUMP B;
..
STORE C INTO 'Results';
```

Some of the text in this example actually looks like English, right? Not too scary, at least at this point. Looking at each line in turn, you can see the basic flow of a Pig program. (Note that this code can either be part of a script or issued on the interactive shell called Grunt — we learn more about Grunt in a few pages.)

1. **Load**: You first load (`LOAD`) the data you want to manipulate. As in a typical MapReduce job, that data is stored in HDFS. For a Pig program to access the data, you first tell Pig what file or files to use. For that task, you use the `LOAD 'data_file'` command.

   Here, `'data_file'` can specify either an HDFS file or a directory. If a directory is specified, all files in that directory are loaded into the program.

   **TIP** If the data is stored in a file format that isn't natively accessible to Pig, you can optionally add the `USING` function to the `LOAD` statement to specify a user-defined function that can read in (and interpret) the data.

2. **Transform**: You run the data through a set of transformations that, way under the hood and far removed from anything you have to concern yourself with, are translated into a set of Map and Reduce tasks.

   **REMEMBER** The transformation logic is where all the data manipulation happens. Here, you can `FILTER` out rows that aren't of interest, `JOIN` two sets of data files, `GROUP` data to build aggregations, `ORDER` results, and do much, much more.

3. **Dump**: Finally, you dump (`DUMP`) the results to the screen

   or

   **Store** (`STORE`) the results in a file somewhere.

   **REMEMBER** You would typically use the `DUMP` command to send the output to the screen when you debug your programs. When your program goes into production, you simply change the `DUMP` call to a `STORE` call so that any results from running your programs are stored in a file for further processing or analysis.

## Working Through the ABCs of Pig Latin

Pig Latin is the language for Pig programs. Pig translates the Pig Latin script into MapReduce jobs that can be executed within Hadoop cluster. When coming up with Pig Latin, the development team followed three key design principles:

- **Keep it simple**. Pig Latin provides a streamlined method for interacting with Java MapReduce. It's an abstraction, in other words, that simplifies the creation of parallel programs on the Hadoop cluster for data flows and analysis. Complex tasks may require a series of interrelated data transformations — such series are encoded as *data flow sequences*.

  **REMEMBER** Writing data transformation and flows as Pig Latin scripts instead of Java MapReduce programs makes these programs easier to write, understand, and maintain because a) you don't have to write the job in Java, b) you don't have to think in terms of MapReduce, and c) you don't need to come up with custom code to support rich data types. Pig Latin provides a simpler language to exploit your Hadoop cluster, thus making it easier for more people to leverage the power of Hadoop and become productive sooner.

- **Make it smart**. You may recall that the Pig Latin Compiler does the work of transforming a Pig Latin program into a series of Java MapReduce jobs. The trick is to make sure that the compiler can optimize the execution of these Java MapReduce jobs automatically, allowing the user to focus on semantics rather than on how to optimize and access the data.

  For you SQL types out there, this discussion will sound familiar. SQL is set up as a declarative query that you use to access structured data stored in an RDBMS. The RDBMS engine first translates the query to a data access method and then looks at the statistics and generates a series of data access approaches. The cost-based optimizer chooses the most efficient approach for execution.

- **Don't limit development**. Make Pig extensible so that developers can add functions to address their particular business problems.

**REMEMBER** Traditional RDBMS data warehouses make use of the ETL data processing pattern, where you *e*xtract data from outside sources, *t*ransform it to fit your operational needs, and then *l*oad it into the end target, whether it's an operational data store, a data warehouse, or another variant of database. However, with big data, you typically want to reduce the amount of data you have moving about, so you end up bringing the processing to the data itself. The language for Pig data flows, therefore, takes a pass on the old ETL approach, and goes with ELT instead: *E*xtract the data from your various sources, *l*oad it into HDFS, and then *t*ransform it as necessary to prepare the data for further analysis.

## Uncovering Pig Latin Structures

To see how Pig Latin is put together, check out the following (bare-bones, training wheel) program for playing around in Hadoop. (To save time and money — hey, coming up with great examples can cost a pretty penny! — we'll reuse the Flight Data scenario from Chapter 6.) Compare and Contrast is often a good way to learn something new, so go ahead and review the problem we're solving in Chapter 6, and take a look at the code in Listings 6-3, 6-4, and 6-5.

The problem we're trying to solve involves calculating the total number of flights flown by every carrier. Following is the Pig Latin script we'll use to answer this question.

**Listing 8-2: Pig script calculating the total miles flown**

```
records = LOAD '2013_subset.csv' USING PigStorage(',') AS
          (Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDep
          Time,ArrTime,CRSArrTime,UniqueCarrier,FlightNum
          ,TailNum,ActualElapsedTime,CRSElapsedTime,AirTi
          me,ArrDelay,DepDelay,Origin,Dest,Distance:int,T
          axiIn,TaxiOut,Cancelled,CancellationCode,Divert
          ed,CarrierDelay,WeatherDelay,NASDelay,SecurityD
          elay,LateAircraftDelay);

milage_recs = GROUP records ALL;
tot_miles = FOREACH milage_recs GENERATE SUM(records.Distance);

DUMP tot_miles;
```

Before we walk through the code, here are a few high-level observations: The Pig script is a lot smaller than the MapReduce application you'd need to accomplish the same task — the Pig script only has 4 lines of code! Yes, that first line is rather long, but it's pretty simple, since we're just listing the names of the columns in the data set. And not only is the code shorter, but it's even semi-human readable. Just look at the key words in the script: LOADs the data, does a GROUP, calculates a SUM and finally DUMPs out an answer. You'll remember that one reason why SQL is so awesome is because it's a declarative query language, meaning you express queries on what you want the result to be, not how it is executed. Pig can be equally cool because it also gives you that declarative aspect and you don't have to tell it how to actually do it and in particular how to do the MapReduce stuff.

Ready for your walkthrough? As you make your way through the code, take note of these principles:

- **Most Pig scripts start with the `LOAD` statement to read data from HDFS**. In this case, we're loading data from a `.csv` file. Pig has a

data model it uses, so next we need to map the file's data model to the Pig data mode. This is accomplished with the help of the `USING` statement. (More on the Pig data model in the next section.) We then specify that it is a comma-delimited file with the `PigStorage ( ',' )` statement followed by the `AS` statement defining the name of each of the columns.

- **Aggregations are commonly used in Pig to summarize data sets**. The `GROUP` statement is used to aggregate the records into a single record `mileage_recs`. The `ALL` statement is used to aggregate all tuples into a single group. Note that some statements — including the following `SUM` statement — requires a preceding `GROUP ALL` statement for global sums.
- **`FOREACH` … `GENERATE` statements are used here to transform columns data**. In this case, we want to count the miles traveled in the `records_Distance` column. The `SUM` statement computes the sum of the `record_Distance` column into a single-column collection `total_miles`.
- **The `DUMP` operator is used to execute the Pig Latin statement and display the results on the screen**. `DUMP` is used in interactive mode, which means that the statements are executable immediately and the results are not saved. Typically, you will either use the DUMP or STORE operators at the end of your Pig script.

## Looking at Pig Data Types and Syntax

Pig's data types make up the data model for how Pig thinks of the structure of the data it is processing. With Pig, the data model gets defined when the data is loaded. Any data you load into Pig from disk is going to have a particular schema and structure. Pig needs to understand that structure, so when you do the loading, the data automatically goes through a mapping.

Luckily for you, the Pig data model is rich enough to handle most anything thrown its way, including table-like structures and nested hierarchical data structures. In general terms, though, Pig data types can be broken into two categories: scalar types and complex types. *Scalar* types contain a single value, whereas *complex* types contain other types, such as the Tuple, Bag, and Map types listed below.

Pig Latin has these four types in its data model:

- **Atom**: An *atom* is any single value, such as a string or a number — 'Diego', for example. Pig's atomic values are scalar types that appear in most programming languages — `int`, `long`, `float`, `double`, `chararray`, and `bytearray`, for example. See Figure 8-2 to see sample atom types.

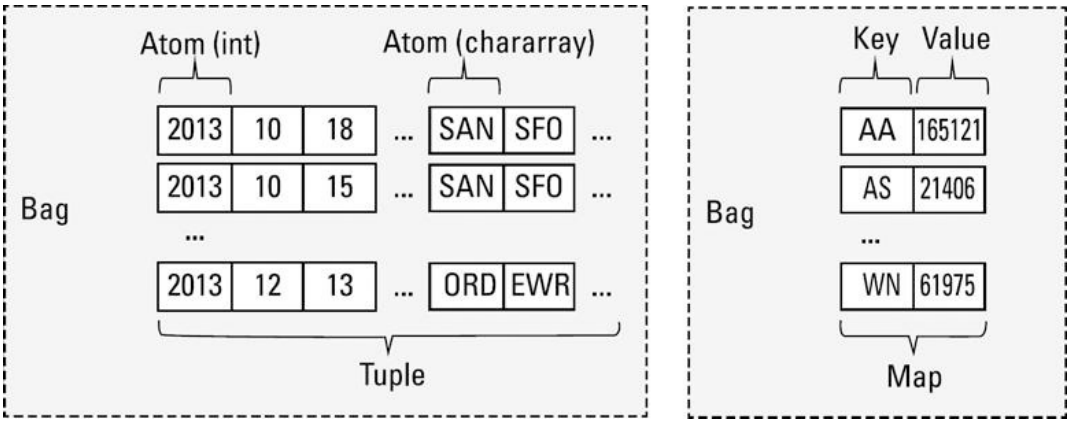


**Figure 8-2:** Sample Pig Data Types

- **Tuple**: A *tuple* is a record that consists of a sequence of fields. Each field can be of any type — 'Diego', 'Gomez', or 6, for example. Think of a tuple as a row in a table.
- **Bag**: A *bag* is a collection of non-unique tuples. The schema of the bag is flexible — each tuple in the collection can contain an arbitrary number of fields, and each field can be of any type.
- **Map**: A map is a collection of key value pairs. Any type can be stored in the value, and the key needs to be unique. The key of a map must be a `chararray` and the value can be of any type.

Figure 8-2 offers some fine examples of Tuple, Bag, and Map data types, as well.

**REMEMBER** The value of all these types can also be `null`. The semantics for `null` are similar to those used in SQL. The concept of `null` in Pig means that the value is unknown. Nulls can show up in the data in cases where values are unreadable or unrecognizable — for example, if you were to use a wrong data type in the LOAD statement. `Null` could be used as a placeholder until data is added or as a value for a field that is optional.

Pig Latin has a simple syntax with powerful semantics you'll use to carry out two primary operations: access and transform data. If you compare the Pig implementation for calculating miles traveled by airline (Listing 8-1) with the Java MapReduce implementations (Listings 6-1, 6-2, and 6-3), they both come up with the same result but the Pig implementation has a lot less code and is easier to understand.

**REMEMBER** In a Hadoop context, *accessing* data means allowing developers to load, store, and stream data, whereas *transforming* data means taking advantage of Pig's ability to group, join, combine, split, filter, and sort data. Table 8-1 gives an overview of the operators associated with each operation.

**Table 8-1: Pig Latin Operators**

| *Operation* | *Operator* | *Explanation* |
|---|---|---|
| Data Access | LOAD/STORE | Read and Write data to file system |
| | DUMP | Write output to standard output (stdout) |
| | STREAM | Send all records through external binary |
| | FOREACH | Apply expression to each record and output one or more records |
| | FILTER | Apply predicate and remove records that don't meet condition |
| | GROUP/COGROUP | Aggregate records with the same key from one or more inputs |
| | JOIN | Join two or more records based on a condition |
| Transformations | CROSS | Cartesian product of two or more inputs |
| | ORDER | Sort records based on key |
| | DISTINCT | Remove duplicate records |
| | UNION | Merge two data sets |
| | SPLIT | Divide data into two or more bags based on predicate |
| | LIMIT | subset the number of records |

Pig also provides a few operators that are helpful for debugging and troubleshooting, as shown in Table 8-2:

**Table 8-2: Operators for Debugging and Troubleshooting**

| *Operation* | *Operator* | *Description* |
|---|---|---|
| Debug | DESCRIBE | Return the schema of a relation. |
| | DUMP | Dump the contents of a relation to the screen. |
| | EXPLAIN | Display the MapReduce execution plans. |

**REMEMBER** Part of the paradigm shift of Hadoop is that you apply your schema at Read instead of Load. According to the old way of doing things — the RDBMS way — when you load data into your database system, you must load it into a well-defined set of tables. Hadoop allows you to store all that raw data upfront and apply the schema at Read. With Pig, you do this during the loading of the data, with the help of the `LOAD` operator. Back in Listing 8-2, we used the `LOAD` operator to read the flight data from a file.

The optional `USING` statement defines how to map the data structure within the file to the Pig data model — in this case, the `PigStorage()` data structure, which parses delimited text files. (This part of the `USING` statement is often referred to as a `LOAD Func` and works in a fashion similar to a custom deserializer.) The optional `AS` clause defines a schema for the data that is being mapped. If you don't use an `AS` clause, you're basically telling the default `LOAD Func` to expect a plain text file that is tab delimited. With no schema provided, the fields must be referenced by position because no name is defined.

Using `AS` clauses means that you have a schema in place at read-time for your text files, which allows users to get started quickly and provides agile schema modeling and flexibility so that you can add more data to your analytics.

**TECHNICAL STUFF** The `LOAD` operator operates on the principle of *lazy evaluation*, also referred to as *call-by-need*. Now lazy doesn't sound particularly praiseworthy, but all it means is that you delay the evaluation of an expression until you really need it. In the context of our Pig example, that means that after the `LOAD` statement is executed, no data is moved — nothing gets shunted around — until a statement to write data is encountered. You can have a Pig script that is a page long filled with complex transformations, but nothing gets executed until the `DUMP` or `STORE` statement is encountered.

## Evaluating Local and Distributed Modes of Running Pig scripts

Before you can run your first Pig script, you need to have a handle on how Pig programs can be packaged with the Pig server.

Pig has two modes for running scripts, as shown in Figure 8-3:

- **Local mode**: All scripts are run on a single machine without requiring Hadoop MapReduce and HDFS. This can be useful for developing and testing Pig logic. If you're using a small set of data to develope or test your code, then local mode could be faster than going through the MapReduce infrastructure.

  **REMEMBER** Local mode doesn't require Hadoop. When you run in Local mode, the Pig program runs in the context of a local Java Virtual Machine, and data access is via the local file system of a single machine. Local mode is actually a local simulation of MapReduce in Hadoop's LocalJobRunner class.

- **MapReduce mode (also known as Hadoop mode)**: Pig is executed on the Hadoop cluster. In this case, the Pig script gets converted into a series of MapReduce jobs that are then run on the Hadoop cluster.

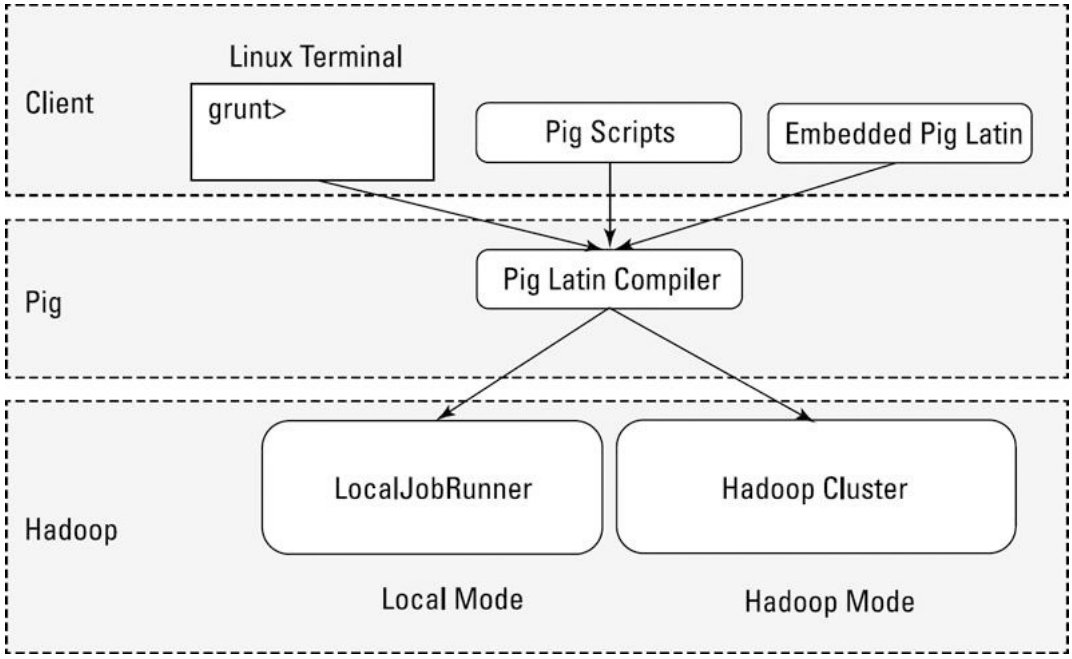


**Figure 8-3:** Pig modes

**REMEMBER** If you have a terabyte of data that you want to perform operations on and you want to interactively develop a program, you may soon find things slowing down considerably, and you may start growing your storage. Local mode allows you to work with a subset of your data in a more interactive manner so that you can figure out the logic (and work out the bugs) of your Pig program. After you have things set up as you want them and your operations are running smoothly, you can then run the script against the full data set using MapReduce mode.

## Checking Out the Pig Script Interfaces

Pig programs can be packaged in three different ways:

- **Script**: This method is nothing more than a file containing Pig Latin commands, identified by the `.pig` suffix (`FlightData.pig`, for example). Ending your Pig program with the `.pig` extension is a convention but not required. The commands are interpreted by the Pig Latin compiler and executed in the order determined by the Pig optimizer.
- **Grunt**: Grunt acts as a command interpreter where you can interactively enter Pig Latin at the Grunt command line and immediately see the response. This method is helpful for prototyping during initial development and with what-if scenarios.
- **Embedded**: Pig Latin statements can be executed within Java, Python, or JavaScript programs.

Pig scripts, Grunt shell Pig commands, and embedded Pig programs can run in either Local mode or MapReduce mode.

The Grunt shell provides an interactive shell to submit Pig commands or run Pig scripts. To start the Grunt shell in Interactive mode, just submit the command `pig` at your shell.

To specify whether a script or Grunt shell is executed locally or in Hadoop mode just specify it in the `-` flag to the `pig` command. The following is an example of how you'd specify running your Pig script in local mode:

```
pig -x local milesPerCarrier.pig
```

Here's how you'd run the Pig script in Hadoop mode, which is the default if you don't specify the flag:

```
pig -x mapreduce milesPerCarrier.pig
```

**REMEMBER** By default, when you specify the `pig` command without any parameters, it starts the Grunt shell in Hadoop mode. If you want to start the Grunt shell in local mode just add the `-x local` flag to the command. Here is an example:

```
pig -x local
```

## Scripting with Pig Latin

Hadoop is a rich and quickly evolving ecosystem with a growing set of new applications. Rather than try to keep up with all the requirements for new capabilities, Pig is designed to be extensible via *user-defined functions*, also known as UDFs. UDFs can be written in a number of programming languages, including Java, Python, and JavaScript. Developers are also posting and sharing a growing collection of UDFs online. (Look for Piggy Bank and DataFu, to name just two examples of such online collections.) Some of the Pig UDFs that are part of these

repositories are `LOAD`/`STORE` functions (XML, for example), date time functions, text, math, and stats functions.

Pig can also be embedded in host languages such as Java, Python, and JavaScript, which allows you to integrate Pig with your existing applications. It also helps overcome limitations in the Pig language. One of the most commonly referenced limitations is that Pig doesn't support control flow statements: `if/else`, `while loop`, `for loop`, and `condition` statements. Pig natively supports data flow, but needs to be embedded within another language to provide control flow. There are tradeoffs, however of embedding Pig in a control-flow language. For example if a Pig statement is embedded in a loop, every time the loop iterates and runs the Pig statement, this causes a separate MapReduce job to run.