The Pig Documentation provides the information you need to get started using Pig. If you haven't already, download Pig now. When you are ready to start writing your own scripts, review the [Pig Latin Basics](http://pig.apache.org/docs/r0.15.0/basic.html) manual to become familiar with the Pig Latin operators and the supported data types. Functions can be a part of almost every operator in Pig. The [Built in Functions](http://pig.apache.org/docs/r0.15.0/func.html) guide describes Pig's built in functions. The [User Defined Functions](http://pig.apache.org/docs/r0.15.0/udf.html) manual shows you how to how to write your own functions and how to access/contribute functions using the Piggy Bank repository. The mechanisms featured in the [Control Structures](http://pig.apache.org/docs/r0.15.0/cont.html) guide give you greater control over how your Pig scripts are structured and executed. The [Performance and Efficiency](http://pig.apache.org/docs/r0.15.0/cont.html) guide provides valuable examples and suggestions for optimizing your code. Use Pig's Administration features [Administration](http://pig.apache.org/docs/r0.15.0/admin.html) which provides properties that could be set to be used by all your users. Finally, use Pig's [Shell and Utility Commands](http://pig.apache.org/docs/r0.15.0/cmds.html) to run your programs and Pig's expanded [Testing and Diagnostics](http://pig.apache.org/docs/r0.15.0/test.html) tools to examine and/or debug your programs. If you have more questions, you can ask on the [Pig Mailing Lists](http://hadoop.apache.org/pig/mailing_lists.html).

**Pig Setup**

**Requirements**

**Mandatory**

Unix and Windows users need the following:

* **Hadoop 0.23.X, 1.X or 2.X** - <http://hadoop.apache.org/common/releases.html> (You can run Pig with different versions of Hadoop by setting HADOOP\_HOME to point to the directory where you have installed Hadoop. If you do not set HADOOP\_HOME, by default Pig will run with the embedded version, currently Hadoop 1.0.4.)
* **Java 1.7** - <http://java.sun.com/javase/downloads/index.jsp> (set JAVA\_HOME to the root of your Java installation)

**Optional**

* **Python 2.7** - [https://www.python.org](http://jython.org/downloads.html) (when using Streaming Python UDFs)
* **Ant 1.8** - <http://ant.apache.org/> (for builds)

**Download Pig**

To get a Pig distribution, do the following:

1. Download a recent stable release from one of the Apache Download Mirrors (see [Pig Releases](http://hadoop.apache.org/pig/releases.html)).
2. Unpack the downloaded Pig distribution, and then note the following:
   * The Pig script file, pig, is located in the bin directory (/pig-n.n.n/bin/pig). The Pig environment variables are described in the Pig script file.
   * The Pig properties file, pig.properties, is located in the conf directory (/pig-n.n.n/conf/pig.properties). You can specify an alternate location using the PIG\_CONF\_DIR environment variable.
3. Add /pig-n.n.n/bin to your path. Use export (bash,sh,ksh) or setenv (tcsh,csh). For example:   
   $ export PATH=/<my-path-to-pig>/pig-n.n.n/bin:$PATH
4. Test the Pig installation with this simple command: $ pig -help

**Build Pig**

To build pig, do the following:

1. Check out the Pig code from SVN: svn co http://svn.apache.org/repos/asf/pig/trunk
2. Build the code from the top directory: ant   
   If the build is successful, you should see the pig.jar file created in that directory.
3. Validate the pig.jar by running a unit test: ant test
4. If you are using Hadoop 0.23.X or 2.X, please add -Dhadoopversion=23 in your ant command line in the previous steps

**Running Pig**

You can run Pig (execute Pig Latin statements and Pig commands) using various modes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Local Mode** | **Tez Local Mode** | **Mapreduce Mode** | **Tez Mode** |
| **Interactive Mode** | yes | experimental | yes | yes |
| **Batch Mode** | yes | experimental | yes | yes |

**Execution Modes**

Pig has two execution modes or exectypes:

* **Local Mode** - To run Pig in local mode, you need access to a single machine; all files are installed and run using your local host and file system. Specify local mode using the -x flag (pig -x local).
* **Tez Local Mode** - To run Pig in tez local mode. It is similar to local mode, except internally Pig will invoke tez runtime engine. Specify Tez local mode using the -x flag (pig -x tez\_local).

**Note:** Tez local mode is experimental. There are some queries which just error out on bigger data in local mode.

* **Mapreduce Mode** - To run Pig in mapreduce mode, you need access to a Hadoop cluster and HDFS installation. Mapreduce mode is the default mode; you can, *but don't need to*, specify it using the -x flag (pig OR pig -x mapreduce).
* **Tez Mode** - To run Pig in Tez mode, you need access to a Hadoop cluster and HDFS installation. Specify Tez mode using the -x flag (-x tez).

You can run Pig in either mode using the "pig" command (the bin/pig Perl script) or the "java" command (java -cp pig.jar ...).

**Examples**

This example shows how to run Pig in local and mapreduce mode using the pig command.

/\* local mode \*/

$ pig -x local ...

/\* Tez local mode \*/

$ pig -x tez\_local ...

/\* mapreduce mode \*/

$ pig ...

or

$ pig -x mapreduce ...

/\* Tez mode \*/

$ pig -x tez ...

**Interactive Mode**

You can run Pig in interactive mode using the Grunt shell. Invoke the Grunt shell using the "pig" command (as shown below) and then enter your Pig Latin statements and Pig commands interactively at the command line.

**Example**

These Pig Latin statements extract all user IDs from the /etc/passwd file. First, copy the /etc/passwd file to your local working directory. Next, invoke the Grunt shell by typing the "pig" command (in local or hadoop mode). Then, enter the Pig Latin statements interactively at the grunt prompt (be sure to include the semicolon after each statement). The DUMP operator will display the results to your terminal screen.

grunt> A = load 'passwd' using PigStorage(':');

grunt> B = foreach A generate $0 as id;

grunt> dump B;

**Local Mode**

$ pig -x local

... - Connecting to ...

grunt>

**Tez Local Mode**

$ pig -x tez\_local

... - Connecting to ...

grunt>

**Mapreduce Mode**

$ pig -x mapreduce

... - Connecting to ...

grunt>

or

$ pig

... - Connecting to ...

grunt>

**Tez Mode**

$ pig -x tez

... - Connecting to ...

grunt>

**Batch Mode**

You can run Pig in batch mode using [Pig scripts](http://pig.apache.org/docs/r0.15.0/start.html#pig-scripts) and the "pig" command (in local or hadoop mode).

**Example**

The Pig Latin statements in the Pig script (id.pig) extract all user IDs from the /etc/passwd file. First, copy the /etc/passwd file to your local working directory. Next, run the Pig script from the command line (using local or mapreduce mode). The STORE operator will write the results to a file (id.out).

/\* id.pig \*/

A = load 'passwd' using PigStorage(':'); -- load the passwd file

B = foreach A generate $0 as id; -- extract the user IDs

store B into 'id.out'; -- write the results to a file name id.out

**Local Mode**

$ pig -x local id.pig

**Tez Local Mode**

$ pig -x tez\_local id.pig

**Mapreduce Mode**

$ pig id.pig

or

$ pig -x mapreduce id.pig

**Tez Mode**

$ pig -x tez id.pig

**Pig Scripts**

Use Pig scripts to place Pig Latin statements and Pig commands in a single file. While not required, it is good practice to identify the file using the \*.pig extension.

You can run Pig scripts from the command line and from the Grunt shell (see the [run](http://pig.apache.org/docs/r0.15.0/cmds.html#run) and [exec](http://pig.apache.org/docs/r0.15.0/cmds.html#exec) commands).

Pig scripts allow you to pass values to parameters using [parameter substitution](http://pig.apache.org/docs/r0.15.0/cont.html#Parameter-Sub).

**Comments in Scripts**

You can include comments in Pig scripts:

* For multi-line comments use /\* …. \*/
* For single-line comments use --

/\* myscript.pig

My script is simple.

It includes three Pig Latin statements.

\*/

A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float); -- loading data

B = FOREACH A GENERATE name; -- transforming data

DUMP B; -- retrieving results

**Scripts and Distributed File Systems**

Pig supports running scripts (and Jar files) that are stored in HDFS, Amazon S3, and other distributed file systems. The script's full location URI is required (see [REGISTER](http://pig.apache.org/docs/r0.15.0/basic.html#register) for information about Jar files). For example, to run a Pig script on HDFS, do the following:

$ pig hdfs://nn.mydomain.com:9020/myscripts/script.pig

**Pig Latin Statements**

Pig Latin statements are the basic constructs you use to process data using Pig. A Pig Latin statement is an operator that takes a [relation](http://pig.apache.org/docs/r0.15.0/basic.html#relations) as input and produces another relation as output. (This definition applies to all Pig Latin operators except LOAD and STORE which read data from and write data to the file system.) Pig Latin statements may include [expressions](http://pig.apache.org/docs/r0.15.0/basic.html#expressions) and [schemas](http://pig.apache.org/docs/r0.15.0/basic.html#schemas). Pig Latin statements can span multiple lines and must end with a semi-colon ( ; ). By default, Pig Latin statements are processed using [multi-query execution](http://pig.apache.org/docs/r0.15.0/perf.html#multi-query-execution).

Pig Latin statements are generally organized as follows:

* A LOAD statement to read data from the file system.
* A series of "transformation" statements to process the data.
* A DUMP statement to view results or a STORE statement to save the results.

Note that a DUMP or STORE statement is required to generate output.

* In this example Pig will validate, but not execute, the LOAD and FOREACH statements.
* A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);
* B = FOREACH A GENERATE name;
* In this example, Pig will validate and then execute the LOAD, FOREACH, and DUMP statements.
* A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);
* B = FOREACH A GENERATE name;
* DUMP B;
* (John)
* (Mary)
* (Bill)
* (Joe)

**Loading Data**

Use the [LOAD](http://pig.apache.org/docs/r0.15.0/basic.html#load) operator and the [load/store functions](http://pig.apache.org/docs/r0.15.0/udf.html#load-store-functions) to read data into Pig (PigStorage is the default load function).

**Working with Data**

Pig allows you to transform data in many ways. As a starting point, become familiar with these operators:

* Use the [FILTER](http://pig.apache.org/docs/r0.15.0/basic.html#filter) operator to work with tuples or rows of data. Use the [FOREACH](http://pig.apache.org/docs/r0.15.0/basic.html#foreach) operator to work with columns of data.
* Use the [GROUP](http://pig.apache.org/docs/r0.15.0/basic.html#group) operator to group data in a single relation. Use the [COGROUP](http://pig.apache.org/docs/r0.15.0/basic.html#cogroup), [inner JOIN](http://pig.apache.org/docs/r0.15.0/basic.html#join-inner), and [outer JOIN](http://pig.apache.org/docs/r0.15.0/basic.html#join-outer) operators to group or join data in two or more relations.
* Use the [UNION](http://pig.apache.org/docs/r0.15.0/basic.html#union) operator to merge the contents of two or more relations. Use the [SPLIT](http://pig.apache.org/docs/r0.15.0/basic.html#split) operator to partition the contents of a relation into multiple relations.

**Storing Intermediate Results**

Pig stores the intermediate data generated between MapReduce jobs in a temporary location on HDFS. This location must already exist on HDFS prior to use. This location can be configured using the pig.temp.dir property. The property's default value is "/tmp" which is the same as the hardcoded location in Pig 0.7.0 and earlier versions.

**Storing Final Results**

Use the [STORE](http://pig.apache.org/docs/r0.15.0/basic.html#store) operator and the [load/store functions](http://pig.apache.org/docs/r0.15.0/udf.html#load-store-functions) to write results to the file system (PigStorage is the default store function).

**Note:** During the testing/debugging phase of your implementation, you can use DUMP to display results to your terminal screen. However, in a production environment you always want to use the STORE operator to save your results (see [Store vs. Dump](http://pig.apache.org/docs/r0.15.0/perf.html#store-dump)).

**Debugging Pig Latin**

Pig Latin provides operators that can help you debug your Pig Latin statements:

* Use the [DUMP](http://pig.apache.org/docs/r0.15.0/test.html#dump) operator to display results to your terminal screen.
* Use the [DESCRIBE](http://pig.apache.org/docs/r0.15.0/test.html#describe) operator to review the schema of a relation.
* Use the [EXPLAIN](http://pig.apache.org/docs/r0.15.0/test.html#explain) operator to view the logical, physical, or map reduce execution plans to compute a relation.
* Use the [ILLUSTRATE](http://pig.apache.org/docs/r0.15.0/test.html#illustrate) operator to view the step-by-step execution of a series of statements.

**Shortcuts for Debugging Operators**

Pig provides shortcuts for the frequently used debugging operators (DUMP, DESCRIBE, EXPLAIN, ILLUSTRATE). These shortcuts can be used in Grunt shell or within pig scripts. Following are the shortcuts supported by pig

* \d alias - shourtcut for [DUMP](http://pig.apache.org/docs/r0.15.0/test.html#dump) operator. If alias is ignored last defined alias will be used.
* \de alias - shourtcut for [DESCRIBE](http://pig.apache.org/docs/r0.15.0/test.html#describe) operator. If alias is ignored last defined alias will be used.
* \e alias - shourtcut for [EXPLAIN](http://pig.apache.org/docs/r0.15.0/test.html#explain) operator. If alias is ignored last defined alias will be used.
* \i alias - shourtcut for [ILLUSTRATE](http://pig.apache.org/docs/r0.15.0/test.html#illustrate) operator. If alias is ignored last defined alias will be used.
* \q - To quit grunt shell

**Pig Properties**

Pig supports a number of Java properties that you can use to customize Pig behavior. You can retrieve a list of the properties using the [help properties](http://pig.apache.org/docs/r0.15.0/cmds.html#help) command. All of these properties are optional; none are required.

To specify Pig properties use one of these mechanisms:

* The pig.properties file (add the directory that contains the pig.properties file to the classpath)
* The -D and a Pig property in PIG\_OPTS environment variable (export PIG\_OPTS=-Dpig.tmpfilecompression=true)
* The -P command line option and a properties file (pig -P mypig.properties)
* The [set](http://pig.apache.org/docs/r0.15.0/cmds.html#set) command (set pig.exec.nocombiner true)

**Note:** The properties file uses standard Java property file format.

The following precedence order is supported: pig.properties < -D Pig property < -P properties file < set command. This means that if the same property is provided using the –D command line option as well as the –P command line option (properties file), the value of the property in the properties file will take precedence.

To specify Hadoop properties you can use the same mechanisms:

* Hadoop configuration files (include pig-cluster-hadoop-site.xml)
* The -D and a Hadoop property in PIG\_OPTS environment variable (export PIG\_OPTS=–Dmapreduce.task.profile=true)
* The -P command line option and a property file (pig -P property\_file)
* The [set](http://pig.apache.org/docs/r0.15.0/cmds.html#set) command (set mapred.map.tasks.speculative.execution false)

The same precedence holds: Hadoop configuration files < -D Hadoop property < -P properties\_file < set command.

Hadoop properties are not interpreted by Pig but are passed directly to Hadoop. Any Hadoop property can be passed this way.

All properties that Pig collects, including Hadoop properties, are available to any UDF via the UDFContext object. To get access to the properties, you can call the getJobConf method.

**Pig Tutorial**

The Pig tutorial shows you how to run Pig scripts using Pig's local mode, mapreduce mode and Tez mode (see [Execution Modes](http://pig.apache.org/docs/r0.15.0/start.html#execution-modes)).

To get started, do the following preliminary tasks:

1. Make sure the JAVA\_HOME environment variable is set the root of your Java installation.
2. Make sure your PATH includes bin/pig (this enables you to run the tutorials using the "pig" command).
3. $ export PATH=/<my-path-to-pig>/pig-0.14.0/bin:$PATH
4. Set the PIG\_HOME environment variable:
5. $ export PIG\_HOME=/<my-path-to-pig>/pig-0.14.0
6. Create the pigtutorial.tar.gz file:
   * Move to the Pig tutorial directory (.../pig-0.14.0/tutorial).
   * Run the "ant" command from the tutorial directory. This will create the pigtutorial.tar.gz file.
7. Copy the pigtutorial.tar.gz file from the Pig tutorial directory to your local directory.
8. Unzip the pigtutorial.tar.gz file.
9. $ tar -xzf pigtutorial.tar.gz
10. A new directory named pigtmp is created. This directory contains the [Pig Tutorial Files](http://pig.apache.org/docs/r0.15.0/start.html#pig-tutorial-files). These files work with Hadoop 0.20.2 and include everything you need to run [Pig Script 1](http://pig.apache.org/docs/r0.15.0/start.html#pig-script-1) and [Pig Script 2](http://pig.apache.org/docs/r0.15.0/start.html#pig-script-2).

**Running the Pig Scripts in Local Mode**

To run the Pig scripts in local mode, do the following:

1. Move to the pigtmp directory.
2. Execute the following command (using either script1-local.pig or script2-local.pig).
3. $ pig -x local script1-local.pig

Or if you are using Tez local mode:

$ pig -x tez\_local script1-local.pig

1. Review the result files, located in the script1-local-results.txt directory.

The output may contain a few Hadoop warnings which can be ignored:

2010-04-08 12:55:33,642 [main] INFO org.apache.hadoop.metrics.jvm.JvmMetrics

- Cannot initialize JVM Metrics with processName=JobTracker, sessionId= - already initialized

**Running the Pig Scripts in Mapreduce Mode or Tez Mode**

To run the Pig scripts in mapreduce mode, do the following:

1. Move to the pigtmp directory.
2. Copy the excite.log.bz2 file from the pigtmp directory to the HDFS directory.
3. $ hadoop fs –copyFromLocal excite.log.bz2 .
4. Set the PIG\_CLASSPATH environment variable to the location of the cluster configuration directory (the directory that contains the core-site.xml, hdfs-site.xml and mapred-site.xml files):
5. export PIG\_CLASSPATH=/mycluster/conf

If you are using Tez, you will also need to put Tez configuration directory (the directory that contains the tez-site.xml):

export PIG\_CLASSPATH=/mycluster/conf:/tez/conf

**Note:** The PIG\_CLASSPATH can also be used to add any other 3rd party dependencies or resource files a pig script may require. If there is also a need to make the added entries take the highest precedence in the Pig JVM's classpath order, one may also set the env-var PIG\_USER\_CLASSPATH\_FIRST to any value, such as 'true' (and unset the env-var to disable).

1. Set the HADOOP\_CONF\_DIR environment variable to the location of the cluster configuration directory:
2. export HADOOP\_CONF\_DIR=/mycluster/conf
3. Execute the following command (using either script1-hadoop.pig or script2-hadoop.pig):
4. $ pig script1-hadoop.pig

Or if you are using Tez:

$ pig -x tez script1-hadoop.pig

1. Review the result files, located in the script1-hadoop-results or script2-hadoop-results HDFS directory:
2. $ hadoop fs -ls script1-hadoop-results
3. $ hadoop fs -cat 'script1-hadoop-results/\*' | less

**Pig Tutorial Files**

The contents of the Pig tutorial file (pigtutorial.tar.gz) are described here.

|  |  |
| --- | --- |
| **File** | **Description** |
| pig.jar | Pig JAR file |
| tutorial.jar | User defined functions (UDFs) and Java classes |
| script1-local.pig | Pig Script 1, Query Phrase Popularity (local mode) |
| script1-hadoop.pig | Pig Script 1, Query Phrase Popularity (mapreduce mode) |
| script2-local.pig | Pig Script 2, Temporal Query Phrase Popularity (local mode) |
| script2-hadoop.pig | Pig Script 2, Temporal Query Phrase Popularity (mapreduce mode) |
| excite-small.log | Log file, Excite search engine (local mode) |
| excite.log.bz2 | Log file, Excite search engine (mapreduce) |

The user defined functions (UDFs) are described here.

|  |  |
| --- | --- |
| **UDF** | **Description** |
| ExtractHour | Extracts the hour from the record. |
| NGramGenerator | Composes n-grams from the set of words. |
| NonURLDetector | Removes the record if the query field is empty or a URL. |
| ScoreGenerator | Calculates a "popularity" score for the n-gram. |
| ToLower | Changes the query field to lowercase. |
| TutorialUtil | Divides the query string into a set of words. |

**Pig Script 1: Query Phrase Popularity**

The Query Phrase Popularity script (script1-local.pig or script1-hadoop.pig) processes a search query log file from the Excite search engine and finds search phrases that occur with particular high frequency during certain times of the day.

The script is shown here:

* Register the tutorial JAR file so that the included UDFs can be called in the script.

REGISTER ./tutorial.jar;

* Use the PigStorage function to load the excite log file (excite.log or excite-small.log) into the “raw” bag as an array of records with the fields **user**, **time**, and **query**.

raw = LOAD 'excite.log' USING PigStorage('\t') AS (user, time, query);

* Call the NonURLDetector UDF to remove records if the query field is empty or a URL.

clean1 = FILTER raw BY org.apache.pig.tutorial.NonURLDetector(query);

* Call the ToLower UDF to change the query field to lowercase.

clean2 = FOREACH clean1 GENERATE user, time, org.apache.pig.tutorial.ToLower(query) as query;

* Because the log file only contains queries for a single day, we are only interested in the hour. The excite query log timestamp format is YYMMDDHHMMSS. Call the ExtractHour UDF to extract the hour (HH) from the time field.

houred = FOREACH clean2 GENERATE user, org.apache.pig.tutorial.ExtractHour(time) as hour, query;

* Call the NGramGenerator UDF to compose the n-grams of the query.

ngramed1 = FOREACH houred GENERATE user, hour, flatten(org.apache.pig.tutorial.NGramGenerator(query)) as ngram;

* Use the DISTINCT operator to get the unique n-grams for all records.

ngramed2 = DISTINCT ngramed1;

* Use the GROUP operator to group records by n-gram and hour.

hour\_frequency1 = GROUP ngramed2 BY (ngram, hour);

* Use the COUNT function to get the count (occurrences) of each n-gram.

hour\_frequency2 = FOREACH hour\_frequency1 GENERATE flatten($0), COUNT($1) as count;

* Use the GROUP operator to group records by n-gram only. Each group now corresponds to a distinct n-gram and has the count for each hour.

uniq\_frequency1 = GROUP hour\_frequency2 BY group::ngram;

* For each group, identify the hour in which this n-gram is used with a particularly high frequency. Call the ScoreGenerator UDF to calculate a "popularity" score for the n-gram.

uniq\_frequency2 = FOREACH uniq\_frequency1 GENERATE flatten($0), flatten(org.apache.pig.tutorial.ScoreGenerator($1));

* Use the FOREACH-GENERATE operator to assign names to the fields.

uniq\_frequency3 = FOREACH uniq\_frequency2 GENERATE $1 as hour, $0 as ngram, $2 as score, $3 as count, $4 as mean;

* Use the FILTER operator to remove all records with a score less than or equal to 2.0.

filtered\_uniq\_frequency = FILTER uniq\_frequency3 BY score > 2.0;

* Use the ORDER operator to sort the remaining records by hour and score.

ordered\_uniq\_frequency = ORDER filtered\_uniq\_frequency BY hour, score;

* Use the PigStorage function to store the results. The output file contains a list of n-grams with the following fields: **hour**, **ngram**, **score**, **count**, **mean**.

STORE ordered\_uniq\_frequency INTO '/tmp/tutorial-results' USING PigStorage();

**Pig Script 2: Temporal Query Phrase Popularity**

The Temporal Query Phrase Popularity script (script2-local.pig or script2-hadoop.pig) processes a search query log file from the Excite search engine and compares the occurrence of frequency of search phrases across two time periods separated by twelve hours.

The script is shown here:

* Register the tutorial JAR file so that the user defined functions (UDFs) can be called in the script.

REGISTER ./tutorial.jar;

* Use the PigStorage function to load the excite log file (excite.log or excite-small.log) into the “raw” bag as an array of records with the fields **user**, **time**, and **query**.

raw = LOAD 'excite.log' USING PigStorage('\t') AS (user, time, query);

* Call the NonURLDetector UDF to remove records if the query field is empty or a URL.

clean1 = FILTER raw BY org.apache.pig.tutorial.NonURLDetector(query);

* Call the ToLower UDF to change the query field to lowercase.

clean2 = FOREACH clean1 GENERATE user, time, org.apache.pig.tutorial.ToLower(query) as query;

* Because the log file only contains queries for a single day, we are only interested in the hour. The excite query log timestamp format is YYMMDDHHMMSS. Call the ExtractHour UDF to extract the hour from the time field.

houred = FOREACH clean2 GENERATE user, org.apache.pig.tutorial.ExtractHour(time) as hour, query;

* Call the NGramGenerator UDF to compose the n-grams of the query.

ngramed1 = FOREACH houred GENERATE user, hour, flatten(org.apache.pig.tutorial.NGramGenerator(query)) as ngram;

* Use the DISTINCT operator to get the unique n-grams for all records.

ngramed2 = DISTINCT ngramed1;

* Use the GROUP operator to group the records by n-gram and hour.

hour\_frequency1 = GROUP ngramed2 BY (ngram, hour);

* Use the COUNT function to get the count (occurrences) of each n-gram.

hour\_frequency2 = FOREACH hour\_frequency1 GENERATE flatten($0), COUNT($1) as count;

* Use the FOREACH-GENERATE operator to assign names to the fields.

hour\_frequency3 = FOREACH hour\_frequency2 GENERATE $0 as ngram, $1 as hour, $2 as count;

* Use the FILTERoperator to get the n-grams for hour ‘00’

hour00 = FILTER hour\_frequency2 BY hour eq '00';

* Uses the FILTER operators to get the n-grams for hour ‘12’

hour12 = FILTER hour\_frequency3 BY hour eq '12';

* Use the JOIN operator to get the n-grams that appear in both hours.

same = JOIN hour00 BY $0, hour12 BY $0;

* Use the FOREACH-GENERATE operator to record their frequency.

same1 = FOREACH same GENERATE hour\_frequency2::hour00::group::ngram as ngram, $2 as count00, $5 as count12;

* Use the PigStorage function to store the results. The output file contains a list of n-grams with the following fields: **ngram**, **count00**, **count12**.

STORE same1 INTO '/tmp/tutorial-join-results' USING PigStorage();

## Conventions

Conventions for the syntax and code examples in the Pig Latin Reference Manual are described here.

|  |  |  |
| --- | --- | --- |
| Convention | Description | Example |
| ( ) | Parentheses enclose one or more items.  Parentheses are also used to indicate the tuple data type. | Multiple items:  (1, abc, (2,4,6) ) |
| [ ] | Straight brackets enclose one or more optional items.  Straight brackets are also used to indicate the map data type. In this case <> is used to indicate optional items. | Optional items:  [INNER | OUTER] |
| { } | Curly brackets enclose two or more items, one of which is required.  Curly brackets also used to indicate the bag data type. In this case <> is used to indicate required items. | Two items, one required:  { block | nested\_block } |
| … | Horizontal ellipsis points indicate that you can repeat a portion of the code. | Pig Latin syntax statement:  cat path [path …] |
| UPPERCASE  lowercase | In general, uppercase type indicates elements the system supplies.  In general, lowercase type indicates elements that you supply.  (These conventions are not strictly adherered to in all examples.)  See [Case Sensitivity](http://pig.apache.org/docs/r0.15.0/basic.html#case-sensitivity) | Pig Latin statement:  a = LOAD 'data' AS (f1:int);   * LOAD, AS - Pig keywords * a, f1 - aliases you supply * 'data' - data source you supply |

## Reserved Keywords

Pig reserved keywords are listed here.

|  |  |
| --- | --- |
| -- A | assert, and, any, all, arrange, as, asc, AVG |
| -- B | bag, BinStorage, by, bytearray, BIGINTEGER, BIGDECIMAL |
| -- C | cache, CASE, cat, cd, chararray, cogroup, CONCAT, copyFromLocal, copyToLocal, COUNT, cp, cross |
| -- D | datetime, %declare, %default, define, dense, desc, describe, DIFF, distinct, double, du, dump |
| -- E | e, E, eval, exec, explain |
| -- F | f, F, filter, flatten, float, foreach, full |
| -- G | generate, group |
| -- H | help |
| -- I | if, illustrate, import, inner, input, int, into, is |
| -- J | join |
| -- K | kill |
| -- L | l, L, left, limit, load, long, ls |
| -- M | map, matches, MAX, MIN, mkdir, mv |
| -- N | not, null |
| -- O | onschema, or, order, outer, output |
| -- P | parallel, pig, PigDump, PigStorage, pwd |
| -- Q | quit |
| -- R | register, returns, right, rm, rmf, rollup, run |
| -- S | sample, set, ship, SIZE, split, stderr, stdin, stdout, store, stream, SUM |
| -- T | TextLoader, TOKENIZE, through, tuple |
| -- U | union, using |
| -- V, W, X, Y, Z | void |

## Case Sensitivity

The names (aliases) of relations and fields are case sensitive. The names of Pig Latin functions are case sensitive. The names of parameters (see [Parameter Substitution](http://pig.apache.org/docs/r0.15.0/cont.html#Parameter-Sub)) and all other Pig Latin keywords (see [Reserved Keywords](http://pig.apache.org/docs/r0.15.0/basic.html#reserved-keywords)) are case insensitive.

In the example below, note the following:

* The names (aliases) of relations A, B, and C are case sensitive.
* The names (aliases) of fields f1, f2, and f3 are case sensitive.
* Function names PigStorage and COUNT are case sensitive.
* Keywords LOAD, USING, AS, GROUP, BY, FOREACH, GENERATE, and DUMP are case insensitive. They can also be written as load, using, as, group, by, etc.
* In the FOREACH statement, the field in relation B is referred to by positional notation ($0).

grunt> A = LOAD 'data' USING PigStorage() AS (f1:int, f2:int, f3:int);

grunt> B = GROUP A BY f1;

grunt> C = FOREACH B GENERATE COUNT ($0);

grunt> DUMP C;

## Data Types and More

### Identifiers

Identifiers include the names of relations (aliases), fields, variables, and so on. In Pig, identifiers start with a letter and can be followed by any number of letters, digits, or underscores.

Valid identifiers:

A

A123

abc\_123\_BeX\_

Invalid identifiers:

\_A123

abc\_$

A!B

### Relations, Bags, Tuples, Fields

[Pig Latin statements](http://pig.apache.org/docs/r0.15.0/start.html#pl-statements) work with relations. A relation can be defined as follows:

* A relation is a bag (more specifically, an outer bag).
* A bag is a collection of tuples.
* A tuple is an ordered set of fields.
* A field is a piece of data.

A Pig relation is a bag of tuples. A Pig relation is similar to a table in a relational database, where the tuples in the bag correspond to the rows in a table. Unlike a relational table, however, Pig relations don't require that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

Also note that relations are unordered which means there is no guarantee that tuples are processed in any particular order. Furthermore, processing may be parallelized in which case tuples are not processed according to any total ordering.

#### Referencing Relations

Relations are referred to by name (or alias). Names are assigned by you as part of the Pig Latin statement. In this example the name (alias) of the relation is A.

A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);

DUMP A;

(John,18,4.0F)

(Mary,19,3.8F)

(Bill,20,3.9F)

(Joe,18,3.8F)

You an assign an alias to another alias. The new alias can be used in the place of the original alias to refer the original relation.

A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);

B = A;

DUMP B;

#### Referencing Fields

Fields are referred to by positional notation or by name (alias).

* Positional notation is generated by the system. Positional notation is indicated with the dollar sign ($) and begins with zero (0); for example, $0, $1, $2.
* Names are assigned by you using schemas (or, in the case of the GROUP operator and some functions, by the system). You can use any name that is not a Pig keyword (see [Identifiers](http://pig.apache.org/docs/r0.15.0/basic.html#identifiers) for valid name examples).

Given relation A above, the three fields are separated out in this table.

|  |  |  |  |
| --- | --- | --- | --- |
|  | First Field | Second Field | Third Field |
| Data type | chararray | int | float |
| Positional notation (generated by system) | $0 | $1 | $2 |
| Possible name (assigned by you using a schema) | name | age | gpa |
| Field value (for the first tuple) | John | 18 | 4.0 |

As shown in this example when you assign names to fields (using the AS schema clause) you can still refer to the fields using positional notation. However, for debugging purposes and ease of comprehension, it is better to use field names.

A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);

X = FOREACH A GENERATE name,$2;

DUMP X;

(John,4.0F)

(Mary,3.8F)

(Bill,3.9F)

(Joe,3.8F)

In this example an error is generated because the requested column ($3) is outside of the declared schema (positional notation begins with $0). Note that the error is caught before the statements are executed.

A = LOAD 'data' AS (f1:int,f2:int,f3:int);

B = FOREACH A GENERATE $3;

DUMP B;

2009-01-21 23:03:46,715 [main] ERROR org.apache.pig.tools.grunt.GruntParser - java.io.IOException:

Out of bound access. Trying to access non-existent : 3. Schema {f1: bytearray,f2: bytearray,f3: bytearray} has 3 column(s).

etc ...

#### Referencing Fields that are Complex Data Types

As noted, the fields in a tuple can be any data type, including the complex data types: bags, tuples, and maps.

* Use the schemas for complex data types to name fields that are complex data types.
* Use the dereference operators to reference and work with fields that are complex data types.

In this example the data file contains tuples. A schema for complex data types (in this case, tuples) is used to load the data. Then, dereference operators (the dot in t1.t1a and t2.$0) are used to access the fields in the tuples. Note that when you assign names to fields you can still refer to these fields using positional notation.

cat data;

(3,8,9) (4,5,6)

(1,4,7) (3,7,5)

(2,5,8) (9,5,8)

A = LOAD 'data' AS (t1:tuple(t1a:int, t1b:int,t1c:int),t2:tuple(t2a:int,t2b:int,t2c:int));

DUMP A;

((3,8,9),(4,5,6))

((1,4,7),(3,7,5))

((2,5,8),(9,5,8))

X = FOREACH A GENERATE t1.t1a,t2.$0;

DUMP X;

(3,4)

(1,3)

(2,9)

### Data Types

#### Simple and Complex

|  |  |  |
| --- | --- | --- |
| **Simple Types** | Description | Example |
| int | Signed 32-bit integer | 10 |
| long | Signed 64-bit integer | Data:     10L or 10l  Display: 10L |
| float | 32-bit floating point | Data:     10.5F or 10.5f or 10.5e2f or 10.5E2F  Display: 10.5F or 1050.0F |
| double | 64-bit floating point | Data:     10.5 or 10.5e2 or 10.5E2  Display: 10.5 or 1050.0 |
| chararray | Character array (string) in Unicode UTF-8 format | hello world |
| bytearray | Byte array (blob) |  |
| boolean | boolean | true/false (case insensitive) |
| datetime | datetime | 1970-01-01T00:00:00.000+00:00 |
| biginteger | Java BigInteger | 200000000000 |
| bigdecimal | Java BigDecimal | 33.456783321323441233442 |
| **Complex Types** |  |  |
| tuple | An ordered set of fields. | (19,2) |
| bag | An collection of tuples. | {(19,2), (18,1)} |
| map | A set of key value pairs. | [open#apache] |

Note the following general observations about data types:

* Use schemas to assign types to fields.  If you don't assign types, fields default to type bytearray and implicit conversions are applied to the data depending on the context in which that data is used. For example, in relation B, f1 is converted to integer because 5 is integer. In relation C, f1 and f2 are converted to double because we don't know the type of either f1 or f2.
* A = LOAD 'data' AS (f1,f2,f3);
* B = FOREACH A GENERATE f1 + 5;
* C = FOREACH A generate f1 + f2;
* If a schema is defined as part of a load statement, the load function will attempt to enforce the schema. If the data does not conform to the schema, the loader will generate a null value or an error.
* A = LOAD 'data' AS (name:chararray, age:int, gpa:float);
* If an explicit cast is not supported, an error will occur. For example, you cannot cast a chararray to int.
* A = LOAD 'data' AS (name:chararray, age:int, gpa:float);
* B = FOREACH A GENERATE (int)name;

This will cause an error …

* If Pig cannot resolve incompatible types through implicit casts, an error will occur. For example, you cannot add chararray and float (see the Types Table for addition and subtraction).
* A = LOAD 'data' AS (name:chararray, age:int, gpa:float);
* B = FOREACH A GENERATE name + gpa;

This will cause an error …

All data types have corresponding [schemas](http://pig.apache.org/docs/r0.15.0/basic.html#schemas).

#### Tuple

A tuple is an ordered set of fields.

##### Syntax

|  |
| --- |
| ( field [, field …] ) |

##### Terms

|  |  |
| --- | --- |
| (  ) | A tuple is enclosed in parentheses ( ). |
| field | A piece of data. A field can be any data type (including tuple and bag). |

##### Usage

You can think of a tuple as a row with one or more fields, where each field can be any data type and any field may or may not have data. If a field has no data, then the following happens:

* In a load statement, the loader will inject null into the tuple. The actual value that is substituted for null is loader specific; for example, PigStorage substitutes an empty field for null.
* In a non-load statement, if a requested field is missing from a tuple, Pig will inject null.

Also see [tuple schemas](http://pig.apache.org/docs/r0.15.0/basic.html#tuple-schema).

##### Example

In this example the tuple contains three fields.

(John,18,4.0F)

#### Bag

A bag is a collection of tuples.

##### Syntax: Inner bag

|  |
| --- |
| { tuple [, tuple …] } |

##### Terms

|  |  |
| --- | --- |
| {  } | An inner bag is enclosed in curly brackets { }. |
| tuple | A tuple. |

##### Usage

Note the following about bags:

* A bag can have duplicate tuples.
* A bag can have tuples with differing numbers of fields. However, if Pig tries to access a field that does not exist, a null value is substituted.
* A bag can have tuples with fields that have different data types. However, for Pig to effectively process bags, the schemas of the tuples within those bags should be the same. For example, if half of the tuples include chararray fields and while the other half include float fields, only half of the tuples will participate in any kind of computation because the chararray fields will be converted to null.

Bags have two forms: outer bag (or relation) and inner bag.

Also see [bag schemas](http://pig.apache.org/docs/r0.15.0/basic.html#bag-schema).

##### Example: Outer Bag

In this example A is a relation or bag of tuples. You can think of this bag as an outer bag.

A = LOAD 'data' as (f1:int, f2:int, f3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

##### Example: Inner Bag

Now, suppose we group relation A by the first field to form relation X.

In this example X is a relation or bag of tuples. The tuples in relation X have two fields. The first field is type int. The second field is type bag; you can think of this bag as an inner bag.

X = GROUP A BY f1;

DUMP X;

(1,{(1,2,3)})

(4,{(4,2,1),(4,3,3)})

(8,{(8,3,4)})

#### Map

A map is a set of key/value pairs.

##### Syntax (<> denotes optional)

|  |
| --- |
| [ key#value <, key#value …> ] |

##### Terms

|  |  |
| --- | --- |
| [ ] | Maps are enclosed in straight brackets [ ]. |
| # | Key value pairs are separated by the pound sign #. |
| key | Must be chararray data type. Must be a unique value. |
| value | Any data type (the defaults to bytearray). |

##### Usage

Key values within a relation must be unique.

Also see [map schemas](http://pig.apache.org/docs/r0.15.0/basic.html#map-schema).

##### Example

In this example the map includes two key value pairs.

[name#John,phone#5551212]

### Nulls and Pig Latin

In Pig Latin, nulls are implemented using the SQL definition of null as unknown or non-existent. Nulls can occur naturally in data or can be the result of an operation.

#### Nulls, Operators, and Functions

Pig Latin operators and functions interact with nulls as shown in this table.

|  |  |
| --- | --- |
| Operator | Interaction |
| Comparison operators:  ==, !=  >, <  >=, <= | If either subexpression is null, the result is null. |
| Comparison operator:  matches | If either the string being matched against or the string defining the match is null, the result is null. |
| Arithmetic operators:   + , -, \*, /  % modulo  ? : bincond  CASE : case | If either subexpression is null, the resulting expression is null. |
| Null operator:  is null | If the tested value is null, returns true; otherwise, returns false (see [Null Operators](http://pig.apache.org/docs/r0.15.0/basic.html#null_operators)). |
| Null operator:  is not null | If the tested value is not null, returns true; otherwise, returns false (see [Null Operators](http://pig.apache.org/docs/r0.15.0/basic.html#null_operators)). |
| Dereference operators:  tuple (.) or map (#) | If the de-referenced tuple or map is null, returns null. |
| Operators:  COGROUP, GROUP, JOIN | These operators handle nulls differently (see examples below). |
| Function:  COUNT\_STAR | This function counts all values, including nulls. |
| Cast operator | Casting a null from one type to another type results in a null. |
| Functions:  AVG, MIN, MAX, SUM, COUNT | These functions ignore nulls. |
| Function:  CONCAT | If either subexpression is null, the resulting expression is null. |
| Function:  SIZE | If the tested object is null, returns null. |

For Boolean subexpressions, note the results when nulls are used with these operators:

* FILTER operator – If a filter expression results in null value, the filter does not pass them through (if X is null, !X is also null, and the filter will reject both).
* Bincond operator – If a Boolean subexpression results in null value, the resulting expression is null (see the interactions above for Arithmetic operators)

#### Nulls and Constants

Nulls can be used as constant expressions in place of expressions of any type.

In this example a and null are projected.

A = LOAD 'data' AS (a, b, c).

B = FOREACH A GENERATE a, null;

In this example of an outer join, if the join key is missing from a table it is replaced by null.

A = LOAD 'student' AS (name: chararray, age: int, gpa: float);

B = LOAD 'votertab10k' AS (name: chararray, age: int, registration: chararray, donation: float);

C = COGROUP A BY name, B BY name;

D = FOREACH C GENERATE FLATTEN((IsEmpty(A) ? null : A)), FLATTEN((IsEmpty(B) ? null : B));

Like any other expression, null constants can be implicitly or explicitly cast.

In this example both a and null will be implicitly cast to double.

A = LOAD 'data' AS (a, b, c).

B = FOREACH A GENERATE a + null;

In this example  both a and null will be cast to int, a implicitly, and null explicitly.

A = LOAD 'data' AS (a, b, c).

B = FOREACH A GENERATE a + (int)null;

#### Operations That Produce Nulls

As noted, nulls can be the result of an operation. These operations can produce null values:

* Division by zero
* Returns from user defined functions (UDFs)
* Dereferencing a field that does not exist.
* Dereferencing a key that does not exist in a map. For example, given a map, info, containing [name#john, phone#5551212] if a user tries to use info#address a null is returned.
* Accessing a field that does not exist in a tuple.

##### Example: Accessing a field that does not exist in a tuple

In this example nulls are injected if fields do not have data.

cat data;

2 3

4

7 8 9

A = LOAD 'data' AS (f1:int,f2:int,f3:int)

DUMP A;

(,2,3)

(4,,)

(7,8,9)

B = FOREACH A GENERATE f1,f2;

DUMP B;

(,2)

(4,)

(7,8)

#### Nulls and Load Functions

As noted, nulls can occur naturally in the data. If nulls are part of the data, it is the responsibility of the load function to handle them correctly. Keep in mind that what is considered a null value is loader-specific; however, the load function should always communicate null values to Pig by producing Java nulls.

The Pig Latin load functions (for example, PigStorage and TextLoader) produce null values wherever data is missing. For example, empty strings (chararrays) are not loaded; instead, they are replaced by nulls.

PigStorage is the default load function for the LOAD operator. In this example the is not null operator is used to filter names with null values.

A = LOAD 'student' AS (name, age, gpa);

B = FILTER A BY name is not null;

#### Nulls and GROUP/COGROUP Operators

When using the GROUP operator with a single relation, records with a null group key are grouped together.

A = load 'student' as (name:chararray, age:int, gpa:float);

dump A;

(joe,18,2.5)

(sam,,3.0)

(bob,,3.5)

X = group A by age;

dump X;

(18,{(joe,18,2.5)})

(,{(sam,,3.0),(bob,,3.5)})

When using the GROUP (COGROUP) operator with multiple relations, records with a null group key from different relations are considered different and are grouped separately. In the example below note that there are two tuples in the output corresponding to the null group key: one that contains tuples from relation A (but not relation B) and one that contains tuples from relation B (but not relation A).

A = load 'student' as (name:chararray, age:int, gpa:float);

B = load 'student' as (name:chararray, age:int, gpa:float);

dump B;

(joe,18,2.5)

(sam,,3.0)

(bob,,3.5)

X = cogroup A by age, B by age;

dump X;

(18,{(joe,18,2.5)},{(joe,18,2.5)})

(,{(sam,,3.0),(bob,,3.5)},{})

(,{},{(sam,,3.0),(bob,,3.5)})

#### Nulls and JOIN Operator

The JOIN operator - when performing inner joins - adheres to the SQL standard and disregards (filters out) null values. (See also [Drop Nulls Before a Join](http://pig.apache.org/docs/r0.15.0/perf.html#nulls).)

A = load 'student' as (name:chararray, age:int, gpa:float);

B = load 'student' as (name:chararray, age:int, gpa:float);

dump B;

(joe,18,2.5)

(sam,,3.0)

(bob,,3.5)

X = join A by age, B by age;

dump X;

(joe,18,2.5,joe,18,2.5)

### Constants

Pig provides constant representations for all data types except bytearrays.

|  |  |  |
| --- | --- | --- |
|  | **Constant Example** | **Notes** |
| **Simple Data Types** |  |  |
| int | 19 |  |
| long | 19L |  |
| float | 19.2F or 1.92e2f |  |
| double | 19.2 or 1.92e2 |  |
| chararray | 'hello world' |  |
| bytearray |  | Not applicable. |
| boolean | true/false | Case insensitive. |
| **Complex Data Types** |  |  |
| tuple | (19, 2, 1) | A constant in this form creates a tuple. |
| bag | { (19, 2), (1, 2) } | A constant in this form creates a bag. |
| map | [ 'name' # 'John', 'ext' # 5555 ] | A constant in this form creates a map. |

Please note the following:

* On UTF-8 systems you can specify string constants consisting of printable ASCII characters such as 'abc'; you can specify control characters such as '\t'; and, you can specify a character in Unicode by starting it with '\u', for instance, '\u0001' represents Ctrl-A in hexadecimal (see Wikipedia [ASCII](http://en.wikipedia.org/wiki/ASCII), [Unicode](http://en.wikipedia.org/wiki/Unicode), and [UTF-8](http://en.wikipedia.org/wiki/UTF-8)). In theory, you should be able to specify non-UTF-8 constants on non-UTF-8 systems but as far as we know this has not been tested.
* To specify a long constant, l or L must be appended to the number (for example, 12345678L). If the l or L is not specified, but the number is too large to fit into an int, the problem will be detected at parse time and the processing is terminated.
* Any numeric constant with decimal point (for example, 1.5) and/or exponent (for example, 5e+1) is treated as double unless it ends with f or F in which case it is assigned type float (for example,  1.5f).
* There is no native constant type for datetime field. You can use a ToDate udf with chararray constant as argument to generate a datetime value.

The data type definitions for tuples, bags, and maps apply to constants:

* A tuple can contain fields of any data type
* A bag is a collection of tuples
* A map key must be a chararray; a map value can be any data type

Complex constants (either with or without values) can be used in the same places scalar constants can be used; that is, in FILTER and GENERATE statements.

A = LOAD 'data' USING MyStorage() AS (T: tuple(name:chararray, age: int));

B = FILTER A BY T == ('john', 25);

D = FOREACH B GENERATE T.name, [25#5.6], {(1, 5, 18)};

### Expressions

In Pig Latin, expressions are language constructs used with the FILTER, FOREACH, GROUP, and SPLIT operators as well as the eval functions.

Expressions are written in conventional mathematical infix notation and are adapted to the UTF-8 character set. Depending on the context, expressions can include:

* Any Pig data type (simple data types, complex data types)
* Any Pig operator (arithmetic, comparison, null, boolean, dereference, sign, and cast)
* Any Pig built in function.
* Any user defined function (UDF) written in Java.

In Pig Latin,

* An arithmetic expression could look like this:
* X = GROUP A BY f2\*f3;
* A string expression could look like this, where a and b are both chararrays:
* X = FOREACH A GENERATE CONCAT(a,b);
* A boolean expression could look like this:
* X = FILTER A BY (f1==8) OR (NOT (f2+f3 > f1));

#### Field Expressions

Field expressions represent a field or a [dereference operator](http://pig.apache.org/docs/r0.15.0/basic.html#deref) applied to a field.

#### Star Expressions

Star expressions ( \* ) can be used to represent all the fields of a tuple. It is equivalent to writing out the fields explicitly. In the following example the definition of B and C are exactly the same, and MyUDF will be invoked with exactly the same arguments in both cases.

A = LOAD 'data' USING MyStorage() AS (name:chararray, age: int);

B = FOREACH A GENERATE \*, MyUDF(name, age);

C = FOREACH A GENERATE name, age, MyUDF(\*);

A common error when using the star expression is shown below. In this example, the programmer really wants to count the number of elements in the bag in the second field: COUNT($1).

G = GROUP A BY $0;

C = FOREACH G GENERATE COUNT(\*)

There are some restrictions on use of the star expression when the input schema is unknown (null):

* For GROUP/COGROUP, you can't include a star expression in a GROUP BY column.
* For ORDER BY, if you have project-star as ORDER BY column, you can’t have any other ORDER BY column in that statement.

#### Project-Range Expressions

Project-range ( .. ) expressions can be used to project a range of columns from input. For example:

* .. $x : projects columns $0 through $x, inclusive
* $x .. : projects columns through end, inclusive
* $x .. $y : projects columns through $y, inclusive

If the input relation has a schema, you can refer to columns by alias rather than by column position. You can also combine aliases and column positions in an expression; for example, "col1 .. $5" is valid.

Project-range can be used in all cases where the [star expression](http://pig.apache.org/docs/r0.15.0/basic.html#sexp) ( \* ) is allowed.

Project-range can be used in the following statements: [FOREACH](http://pig.apache.org/docs/r0.15.0/basic.html#foreach), [JOIN](http://pig.apache.org/docs/r0.15.0/basic.html#join-inner), [GROUP](http://pig.apache.org/docs/r0.15.0/basic.html#group), [COGROUP](http://pig.apache.org/docs/r0.15.0/basic.html#cogroup), and [ORDER BY](http://pig.apache.org/docs/r0.15.0/basic.html#order-by) (also when ORDER BY is used within a nested FOREACH block).

A few examples are shown here:

.....

grunt> F = foreach IN generate (int)col0, col1 .. col3;

grunt> describe F;

F: {col0: int,col1: bytearray,col2: bytearray,col3: bytearray}

.....

.....

grunt> SORT = order IN by col2 .. col3, col0, col4 ..;

.....

.....

J = join IN1 by $0 .. $3, IN2 by $0 .. $3;

.....

.....

g = group l1 by b .. c;

.....

There are some restrictions on the use of project-to-end form of project-range (eg "x .. ") when the input schema is unknown (null):

* For GROUP/COGROUP, the project-to-end form of project-range is not allowed.
* For ORDER BY, the project-to-end form of project-range is supported only as the last sort column.
* .....
* grunt> describe IN;
* Schema for IN unknown.
* /\* This statement is supported \*/
* SORT = order IN by $2 .. $3, $6 ..;
* /\* This statement is NOT supported \*/
* SORT = order IN by $2 .. $3, $6 ..;
* .....

#### Boolean Expressions

Boolean expressions can be made up of UDFs that return a boolean value or boolean operators (see [Boolean Operators](http://pig.apache.org/docs/r0.15.0/basic.html#boolops)).

#### Tuple Expressions

Tuple expressions form subexpressions into tuples. The tuple expression has the form (expression [, expression …]), where expression is a general expression. The simplest tuple expression is the star expression, which represents all fields.

#### General Expressions

General expressions can be made up of UDFs and almost any operator. Since Pig does not consider boolean a base type, the result of a general expression cannot be a boolean. Field expressions are the simpliest general expressions.

### Schemas

Schemas enable you to assign names to fields and declare types for fields. Schemas are optional but we encourage you to use them whenever possible; type declarations result in better parse-time error checking and more efficient code execution.

Schemas for [simple types](http://pig.apache.org/docs/r0.15.0/basic.html#schema-simple) and [complex types](http://pig.apache.org/docs/r0.15.0/basic.html#schema-complex) can be used anywhere a schema definition is appropriate.

Schemas are defined with the [LOAD](http://pig.apache.org/docs/r0.15.0/basic.html#load), [STREAM](http://pig.apache.org/docs/r0.15.0/basic.html#stream), and [FOREACH](http://pig.apache.org/docs/r0.15.0/basic.html#foreach) operators using the AS clause. If you define a schema using the LOAD operator, then it is the load function that enforces the schema (see [LOAD](http://pig.apache.org/docs/r0.15.0/basic.html#load) and [User Defined Functions](http://pig.apache.org/docs/r0.15.0/udf.html) for more information).

**Known Schema Handling**

Note the following:

* You can define a schema that includes both the field name and field type.
* You can define a schema that includes the field name only; in this case, the field type defaults to bytearray.
* You can choose not to define a schema; in this case, the field is un-named and the field type defaults to bytearray.

If you assign a name to a field, you can refer to that field using the name or by positional notation. If you don't assign a name to a field (the field is un-named) you can only refer to the field using positional notation.

If you assign a type to a field, you can subsequently change the type using the cast operators. If you don't assign a type to a field, the field defaults to bytearray; you can change the default type using the cast operators.

**Unknown Schema Handling**

Note the following:

* When you JOIN/COGROUP/CROSS multiple relations, if any relation has an unknown schema (or no defined schema, also referred to as a null schema), the schema for the resulting relation is null.
* If you FLATTEN a bag with empty inner schema, the schema for the resulting relation is null.
* If you UNION two relations with incompatible schema, the schema for resulting relation is null.
* If the schema is null, Pig treats all fields as bytearray (in the backend, Pig will determine the real type for the fields dynamically)

See the examples below. If a field's data type is not specified, Pig will use bytearray to denote an unknown type. If the number of fields is not known, Pig will derive an unknown schema.

/\* The field data types are not specified ... \*/

a = load '1.txt' as (a0, b0);

a: {a0: bytearray,b0: bytearray}

/\* The number of fields is not known ... \*/

a = load '1.txt';

a: Schema for a unknown

**How Pig Handles Schema**

As shown above, with a few exceptions Pig can infer the schema of a relationship up front. You can examine the schema of particular relation using [DESCRIBE](http://pig.apache.org/docs/r0.15.0/test.html#describe). Pig enforces this computed schema during the actual execution by casting the input data to the expected data type. If the process is successful the results are returned to the user; otherwise, a warning is generated for each record that failed to convert. Note that Pig does not know the actual types of the fields in the input data prior to the execution; rather, Pig determines the data types and performs the right conversions on the fly.

Having a deterministic schema is very powerful; however, sometimes it comes at the cost of performance. Consider the following example:

A = load 'input' as (x, y, z);

B = foreach A generate x+y;

If you do [DESCRIBE](http://pig.apache.org/docs/r0.15.0/test.html#describe) on B, you will see a single column of type double. This is because Pig makes the safest choice and uses the largest numeric type when the schema is not know. In practice, the input data could contain integer values; however, Pig will cast the data to double and make sure that a double result is returned.

If the schema of a relation can’t be inferred, Pig will just use the runtime data as is and propagate it through the pipeline.

#### Schemas with LOAD and STREAM

With LOAD and STREAM operators, the schema following the AS keyword must be enclosed in parentheses.

In this example the LOAD statement includes a schema definition for simple data types.

A = LOAD 'data' AS (f1:int, f2:int);

#### Schemas with FOREACH

With FOREACH operators, the schema following the AS keyword must be enclosed in parentheses when the FLATTEN operator is used. Otherwise, the schema should not be enclosed in parentheses.

In this example the FOREACH statement includes FLATTEN and a schema for simple data types.

X = FOREACH C GENERATE FLATTEN(B) AS (f1:int, f2:int, f3:int), group;

In this example the FOREACH statement includes a schema for simple expression.

X = FOREACH A GENERATE f1+f2 AS x1:int;

In this example the FOREACH statement includes a schemas for multiple fields.

X = FOREACH A GENERATE f1 as user, f2 as age, f3 as gpa;

#### Schemas for Simple Data Types

Simple data types include int, long, float, double, chararray, bytearray, boolean, datetime, biginteger and bigdecimal.

##### Syntax

|  |
| --- |
| (alias[:type]) [, (alias[:type]) …] ) |

##### Terms

|  |  |
| --- | --- |
| alias | The name assigned to the field. |
| type | (Optional) The simple data type assigned to the field.  The alias and type are separated by a colon ( : ).  If the type is omitted, the field defaults to type bytearray. |
| ( , ) | Multiple fields are enclosed in parentheses and separated by commas. |

##### Examples

In this example the schema defines multiple types.

cat student;

John 18 4.0

Mary 19 3.8

Bill 20 3.9

Joe 18 3.8

A = LOAD 'student' AS (name:chararray, age:int, gpa:float);

DESCRIBE A;

A: {name: chararray,age: int,gpa: float}

DUMP A;

(John,18,4.0F)

(Mary,19,3.8F)

(Bill,20,3.9F)

(Joe,18,3.8F)

In this example field "gpa" will default to bytearray because no type is declared.

cat student;

John 18 4.0

Mary 19 3.8

Bill 20 3.9

Joe 18 3.8

A = LOAD 'data' AS (name:chararray, age:int, gpa);

DESCRIBE A;

A: {name: chararray,age: int,gpa: bytearray}

DUMP A;

(John,18,4.0)

(Mary,19,3.8)

(Bill,20,3.9)

(Joe,18,3.8)

#### Schemas for Complex Data Types

Complex data types include tuples, bags, and maps.

#### Tuple Schemas

A tuple is an ordered set of fields.

##### Syntax

|  |
| --- |
| alias[:tuple] (alias[:type]) [, (alias[:type]) …] ) |

##### Terms

|  |  |
| --- | --- |
| alias | The name assigned to the tuple. |
| :tuple | (Optional) The data type, tuple (case insensitive). |
| ( ) | The designation for a tuple, a set of parentheses. |
| alias[:type] | The constituents of the tuple, where the schema definition rules for the corresponding type applies to the constituents of the tuple:   * alias – the name assigned to the field * type (optional) – the simple or complex data type assigned to the field |

##### Examples

In this example the schema defines one tuple. The load statements are equivalent.

cat data;

(3,8,9)

(1,4,7)

(2,5,8)

A = LOAD 'data' AS (T: tuple (f1:int, f2:int, f3:int));

A = LOAD 'data' AS (T: (f1:int, f2:int, f3:int));

DESCRIBE A;

A: {T: (f1: int,f2: int,f3: int)}

DUMP A;

((3,8,9))

((1,4,7))

((2,5,8))

In this example the schema defines two tuples.

cat data;

(3,8,9) (mary,19)

(1,4,7) (john,18)

(2,5,8) (joe,18)

A = LOAD data AS (F:tuple(f1:int,f2:int,f3:int),T:tuple(t1:chararray,t2:int));

DESCRIBE A;

A: {F: (f1: int,f2: int,f3: int),T: (t1: chararray,t2: int)}

DUMP A;

((3,8,9),(mary,19))

((1,4,7),(john,18))

((2,5,8),(joe,18))

#### Bag Schemas

A bag is a collection of tuples.

##### Syntax

|  |
| --- |
| alias[:bag] {tuple} |

##### Terms

|  |  |
| --- | --- |
| alias | The name assigned to the bag. |
| :bag | (Optional) The data type, bag (case insensitive). |
| { } | The designation for a bag, a set of curly brackets. |
| tuple | A tuple (see Tuple Schema). |

##### Examples

In this example the schema defines a bag. The two load statements are equivalent.

cat data;

{(3,8,9)}

{(1,4,7)}

{(2,5,8)}

A = LOAD 'data' AS (B: bag {T: tuple(t1:int, t2:int, t3:int)});

A = LOAD 'data' AS (B: {T: (t1:int, t2:int, t3:int)});

DESCRIBE A:

A: {B: {T: (t1: int,t2: int,t3: int)}}

DUMP A;

({(3,8,9)})

({(1,4,7)})

({(2,5,8)})

#### Map Schemas

A map is a set of key value pairs.

##### Syntax (<> demotes optional)

|  |
| --- |
| alias<:map> [ <type> ] |

##### Terms

|  |  |
| --- | --- |
| alias | The name assigned to the map. |
| :map | (Optional) The data type, map (case insensitive). |
| [ ] | The designation for a map, a set of straight brackets [ ]. |
| type | (Optional) The datatype (all types allowed, bytearray is the default).  The type applies to the map value only; the map key is always type chararray (see [Map](http://pig.apache.org/docs/r0.15.0/basic.html#map)).  If a type is declared then ALL values in the map must be of this type. |

##### Examples

In this example the schema defines an untyped map (the map values default to bytearray). The load statements are equivalent.

cat data;

[open#apache]

[apache#hadoop]

A = LOAD 'data' AS (M:map []);

A = LOAD 'data' AS (M:[]);

DESCRIBE A;

a: {M: map[ ]}

DUMP A;

([open#apache])

([apache#hadoop])

This example shows the use of a typed maps.

/\* Map types are declared\*/

a = load '1.txt' as(map[int]); --Map value is int

b = foreach a generate (map[(i:int)])a0; -- Map value is tuple

b = stream a through `cat` as (m:map[{(i:int,j:chararray)}]); -- Map value is bag

/\* The MapLookup of a typed map will result in a datatype of the map value \*/

a = load '1.txt' as(map[int]);

b = foreach a generate $0#'key';

/\* Schema for b \*/

b: {int}

#### Schemas for Multiple Types

You can define schemas for data that includes multiple types.

##### Example

In this example the schema defines a tuple, bag, and map.

A = LOAD 'mydata' AS (T1:tuple(f1:int, f2:int), B:bag{T2:tuple(t1:float,t2:float)}, M:map[] );

A = LOAD 'mydata' AS (T1:(f1:int, f2:int), B:{T2:(t1:float,t2:float)}, M:[] );

##### Previous Relation Shortcut

There is a shortcut form to reference the relation on the previous line of a pig script or grunt session:

a = load 'thing' as (x:int);

b = foreach @ generate x;

c = foreach @ generate x;

d = foreach @ generate x;

## Arithmetic Operators and More

### Arithmetic Operators

#### Description

|  |  |  |
| --- | --- | --- |
| Operator | Symbol | Notes |
| addition | + |  |
| subtraction | - |  |
| multiplication | \* |  |
| division | / |  |
| modulo | % | Returns the remainder of a divided by b (a%b).  Works with integral numbers (int, long). |
| bincond | ? : | (condition ? value\_if\_true : value\_if\_false)  The bincond should be enclosed in parenthesis.  The schemas for the two conditional outputs of the bincond should match.  Use expressions only (relational operators are not allowed). |
| case | CASE WHEN THEN ELSE END | CASE expression [ WHEN value THEN value ]+ [ ELSE value ]? END  CASE [ WHEN condition THEN value ]+ [ ELSE value ]? END  Case operator is equivalent to nested bincond operators.  The schemas for all the outputs of the when/else branches should match.  Use expressions only (relational operators are not allowed). |

##### Examples

Suppose we have relation A.

A = LOAD 'data' AS (f1:int, f2:int, B:bag{T:tuple(t1:int,t2:int)});

DUMP A;

(10,1,{(2,3),(4,6)})

(10,3,{(2,3),(4,6)})

(10,6,{(2,3),(4,6),(5,7)})

In this example the modulo operator is used with fields f1 and f2.

X = FOREACH A GENERATE f1, f2, f1%f2;

DUMP X;

(10,1,0)

(10,3,1)

(10,6,4)

In this example the bincond operator is used with fields f2 and B. The condition is "f2 equals 1"; if the condition is true, return 1; if the condition is false, return the count of the number of tuples in B.

X = FOREACH A GENERATE f2, (f2==1?1:COUNT(B));

DUMP X;

(1,1L)

(3,2L)

(6,3L)

In this example the case operator is used with field f2. The expression is "f2 % 2"; if the expression is equal to 0, return 'even'; if the expression is equal to 1, return 'odd'.

X = FOREACH A GENERATE f2, (

CASE f2 % 2

WHEN 0 THEN 'even'

WHEN 1 THEN 'odd'

END

);

DUMP X;

(1,odd)

(3,odd)

(6,even)

This can be also written as follows:

X = FOREACH A GENERATE f2, (

CASE

WHEN f2 % 2 == 0 THEN 'even'

WHEN f2 % 2 == 1 THEN 'odd'

END

);

DUMP X;

(1,odd)

(3,odd)

(6,even)

##### Types Table: addition (+) and subtraction (-) operators

\* bytearray cast as this data type

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | bag | tuple | map | int | long | float | double | chararray | bytearray |
| bag | error | error | error | error | error | error | error | error | error |
| tuple |  | not yet | error | error | error | error | error | error | error |
| map |  |  | error | error | error | error | error | error | error |
| int |  |  |  | int | long | float | double | error | cast as int |
| long |  |  |  |  | long | float | double | error | cast as long |
| float |  |  |  |  |  | float | double | error | cast as float |
| double |  |  |  |  |  |  | double | error | cast as double |
| chararray |  |  |  |  |  |  |  | error | error |
| bytearray |  |  |  |  |  |  |  |  | cast as double |

##### Types Table: multiplication (\*) and division (/) operators

\* bytearray cast as this data type

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | bag | tuple | map | int | long | float | double | chararray | bytearray |
| bag | error | error | error | not yet | not yet | not yet | not yet | error | error |
| tuple |  | error | error | not yet | not yet | not yet | not yet | error | error |
| map |  |  | error | error | error | error | error | error | error |
| int |  |  |  | int | long | float | double | error | cast as int |
| long |  |  |  |  | long | float | double | error | cast as long |
| float |  |  |  |  |  | float | double | error | cast as float |
| double |  |  |  |  |  |  | double | error | cast as double |
| chararray |  |  |  |  |  |  |  | error | error |
| bytearray |  |  |  |  |  |  |  |  | cast as double |

##### Types Table: modulo (%) operator

|  |  |  |  |
| --- | --- | --- | --- |
|  | int | long | bytearray |
| int | int | long | cast as int |
| long |  | long | cast as long |
| bytearray |  |  | error |

### Boolean Operators

#### Description

|  |  |  |
| --- | --- | --- |
| Operator | Symbol | Notes |
| AND | and |  |
| OR | or |  |
| IN | in | IN operator is equivalent to nested OR operators. |
| NOT | not |  |

The result of a boolean expression (an expression that includes boolean and comparison operators) is always of type boolean (true or false).

##### Example

X = FILTER A BY (f1==8) OR (NOT (f2+f3 > f1)) OR (f1 IN (9, 10, 11));

### Cast Operators

#### Description

Pig Latin supports casts as shown in this table.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **from / to** | bag | tuple | map | int | long | float | double | chararray | bytearray | boolean |
| bag |  | error | error | error | error | error | error | error | error | error |
| tuple | error |  | error | error | error | error | error | error | error | error |
| map | error | error |  | error | error | error | error | error | error | error |
| int | error | error | error |  | yes | yes | yes | yes | error | error |
| long | error | error | error | yes |  | yes | yes | yes | error | error |
| float | error | error | error | yes | yes |  | yes | yes | error | error |
| double | error | error | error | yes | yes | yes |  | yes | error | error |
| chararray | error | error | error | yes | yes | yes | yes |  | error | yes |
| bytearray | yes | yes | yes | yes | yes | yes | yes | yes |  | yes |
| boolean | error | error | error | error | error | error | error | yes | error |  |

##### Syntax

|  |
| --- |
| {(data\_type) |  (tuple(data\_type))  | (bag{tuple(data\_type)}) | (map[]) } field |

##### Terms

|  |  |
| --- | --- |
| (data\_type) | The data type you want to cast to, enclosed in parentheses. You can cast to any data type except bytearray (see the table above). |
| field | The field whose type you want to change.  The field can be represented by positional notation or by name (alias). For example, if f1 is the first field and type int, you can cast to type long using (long)$0 or (long)f1. |

##### Usage

Cast operators enable you to cast or convert data from one type to another, as long as conversion is supported (see the table above). For example, suppose you have an integer field, myint, which you want to convert to a string. You can cast this field from int to chararray using (chararray)myint.

Please note the following:

* A field can be explicitly cast. Once cast, the field remains that type (it is not automatically cast back). In this example $0 is explicitly cast to int.
* B = FOREACH A GENERATE (int)$0 + 1;
* Where possible, Pig performs implicit casts. In this example $0 is cast to int (regardless of underlying data) and $1 is cast to double.
* B = FOREACH A GENERATE $0 + 1, $1 + 1.0
* When two bytearrays are used in arithmetic expressions or a bytearray expression is used with built in aggregate functions (such as SUM) they are implicitly cast to double. If the underlying data is really int or long, you’ll get better performance by declaring the type or explicitly casting the data.
* Downcasts may cause loss of data. For example casting from long to int may drop bits.

#### Examples

In this example an int is cast to type chararray (see relation X).

A = LOAD 'data' AS (f1:int,f2:int,f3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

B = GROUP A BY f1;

DUMP B;

(1,{(1,2,3)})

(4,{(4,2,1),(4,3,3)})

(7,{(7,2,5)})

(8,{(8,3,4),(8,4,3)})

DESCRIBE B;

B: {group: int,A: {f1: int,f2: int,f3: int}}

X = FOREACH B GENERATE group, (chararray)COUNT(A) AS total;

(1,1)

(4,2)

(7,1)

(8,2)

DESCRIBE X;

X: {group: int,total: chararray}

In this example a bytearray (fld in relation A) is cast to type tuple.

cat data;

(1,2,3)

(4,2,1)

(8,3,4)

A = LOAD 'data' AS fld:bytearray;

DESCRIBE A;

a: {fld: bytearray}

DUMP A;

((1,2,3))

((4,2,1))

((8,3,4))

B = FOREACH A GENERATE (tuple(int,int,float))fld;

DESCRIBE B;

b: {(int,int,float)}

DUMP B;

((1,2,3))

((4,2,1))

((8,3,4))

In this example a bytearray (fld in relation A) is cast to type bag.

cat data;

{(4829090493980522200L)}

{(4893298569862837493L)}

{(1297789302897398783L)}

A = LOAD 'data' AS fld:bytearray;

DESCRIBE A;

A: {fld: bytearray}

DUMP A;

({(4829090493980522200L)})

({(4893298569862837493L)})

({(1297789302897398783L)})

B = FOREACH A GENERATE (bag{tuple(long)})fld;

DESCRIBE B;

B: {{(long)}}

DUMP B;

({(4829090493980522200L)})

({(4893298569862837493L)})

({(1297789302897398783L)})

In this example a bytearray (fld in relation A) is cast to type map.

cat data;

[open#apache]

[apache#hadoop]

[hadoop#pig]

[pig#grunt]

A = LOAD 'data' AS fld:bytearray;

DESCRIBE A;

A: {fld: bytearray}

DUMP A;

([open#apache])

([apache#hadoop])

([hadoop#pig])

([pig#grunt])

B = FOREACH A GENERATE ((map[])fld;

DESCRIBE B;

B: {map[ ]}

DUMP B;

([open#apache])

([apache#hadoop])

([hadoop#pig])

([pig#grunt])

#### Casting Relations to Scalars

Pig allows you to cast the elements of a single-tuple relation into a scalar value. The tuple can be a single-field or multi-field tulple. If the relation contains more than one tuple, however, a runtime error is generated: "Scalar has more than one row in the output".

The cast relation can be used in any place where an expression of the type would make sense, including FOREACH, FILTER, and SPLIT. Note that if an explicit cast is not used an implict cast will be inserted according to Pig rules. Also, when the schema can't be inferred bytearray is used.

The primary use case for casting relations to scalars is the ability to use the values of global aggregates in follow up computations.

In this example the percentage of clicks belonging to a particular user are computed. For the FOREACH statement, an explicit cast is used. If the SUM is not given a name, a position can be used as well (userid, clicks/(double)C.$0).

A = load 'mydata' as (userid, clicks);

B = group A all;

C = foreach B genertate SUM(A.clicks) as total;

D = foreach A generate userid, clicks/(double)C.total;

dump D;

In this example a multi-field tuple is used. For the FILTER statement, Pig performs an implicit cast. For the FOREACH statement, an explicit cast is used.

A = load 'mydata' as (userid, clicks);

B = group A all;

C = foreach B genertate SUM(A.clicks) as total, COUNT(A) as cnt;

D = FILTER A by clicks > C.total/3

E = foreach D generate userid, clicks/(double)C.total, cnt;

dump E;

### Comparison Operators

#### Description

|  |  |  |
| --- | --- | --- |
| Operator | Symbol | Notes |
| equal | == |  |
| not equal | != |  |
| less than | < |  |
| greater than | > |  |
| less than or equal to | <= |  |
| greater than or equal to | >= |  |
| pattern matching | matches | Takes an expression on the left and a string constant on the right.  expression matches string-constant  Use the Java [format](http://docs.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html) for regular expressions. |

Use the comparison operators with numeric and string data.

#### Examples

**Numeric Example**

X = FILTER A BY (f1 == 8);

**String Example**

X = FILTER A BY (f2 == 'apache');

**Matches Example**

X = FILTER A BY (f1 matches '.\*apache.\*');

#### Types Table: equal (==) operator

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | bag | tuple | map | int | long | float | double | chararray | bytearray | boolean | datetime | biginteger | bigdecimal |
| bag | error | error | error | error | error | error | error | error | error | error | error | error | error |
| tuple |  | boolean  (see Note 1) | error | error | error | error | error | error | error | error | error | error | error |
| map |  |  | boolean  (see Note 2) | error | error | error | error | error | error | error | error | error | error |
| int |  |  |  | boolean | boolean | boolean | boolean | error | cast as boolean | error | error | error | error |
| long |  |  |  |  | boolean | boolean | boolean | error | cast as boolean | error | error | error | error |
| float |  |  |  |  |  | boolean | boolean | error | cast as boolean | error | error | error | error |
| double |  |  |  |  |  |  | boolean | error | cast as boolean | error | error | error | error |
| chararray |  |  |  |  |  |  |  | boolean | cast as boolean | error | error | error | error |
| bytearray |  |  |  |  |  |  |  |  | boolean | error | error | error | error |
| boolean |  |  |  |  |  |  |  |  |  | boolean | error | error | error |
| datetime |  |  |  |  |  |  |  |  |  |  | boolean | error | error |
| biginteger |  |  |  |  |  |  |  |  |  |  |  | boolean | error |
| bigdecimal |  |  |  |  |  |  |  |  |  |  |  |  | boolean |

Note 1: boolean (Tuple A is equal to tuple B if they have the same size s, and for all 0 <= i < s A[i] == B[i])

Note 2: boolean (Map A is equal to map B if A and B have the same number of entries, and for every key k1 in A with a value of v1, there is a key k2 in B with a value of v2, such that k1 == k2 and v1 == v2)

#### Types Table: not equal (!=) operator

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | bag | tuple | map | int | long | float | double | chararray | bytearray | boolean | datetime | biginteger | bigdecimal |
| bag | error | error | error | error | error | error | error | error | error | error | error | error | error |
| tuple |  | error | error | error | error | error | error | error | error | error | error | error | error |
| map |  |  | error | error | error | error | error | error | error | error | error | error | error |
| int |  |  |  | boolean | boolean | boolean | boolean | error | boolean (bytearray cast as int) | error | error | error | error |
| long |  |  |  |  | boolean | boolean | boolean | error | boolean (bytearray cast as long) | error | error | error | error |
| float |  |  |  |  |  | boolean | boolean | error | boolean (bytearray cast as float) | error | error | error | error |
| double |  |  |  |  |  |  | boolean | error | boolean (bytearray cast as double) | error | error | error | error |
| chararray |  |  |  |  |  |  |  | boolean | boolean (bytearray cast as chararray) | error | error | error | error |
| bytearray |  |  |  |  |  |  |  |  | boolean | error | error | error | error |
| boolean |  |  |  |  |  |  |  |  |  | boolean | error | error | error |
| datetime |  |  |  |  |  |  |  |  |  |  | boolean | error | error |
| biginteger |  |  |  |  |  |  |  |  |  |  |  | boolean | error |
| bigdecimal |  |  |  |  |  |  |  |  |  |  |  |  | boolean |

#### Types Table: matches operator

\*Cast as chararray (the second argument must be chararray)

|  |  |  |
| --- | --- | --- |
|  | chararray | bytearray\* |
| chararray | boolean | boolean |
| bytearray | boolean | boolean |

### Type Construction Operators

#### Description

|  |  |  |
| --- | --- | --- |
| Operator | Symbol | Notes |
| tuple constructor | ( ) | Use to construct a tuple from the specified elements. Equivalent to [TOTUPLE](http://pig.apache.org/docs/r0.15.0/func.html#totuple). |
| bag constructor | { } | Use to construct a bag from the specified elements. Equivalent to [TOBAG](http://pig.apache.org/docs/r0.15.0/func.html#tobag). |
| map constructor | [ ] | Use to construct a map from the specified elements. Equivalent to [TOMAP](http://pig.apache.org/docs/r0.15.0/func.html#tomap). |

Note the following:

* These operators can be used anywhere where the expression of the corresponding type is acceptable including FOREACH GENERATE, FILTER, etc.
* A single element enclosed in parens ( ) like (5) is not considered to be a tuple but rather an arithmetic operator.
* For bags, every element is put in the bag; if the element is not a tuple Pig will create a tuple for it:
  + Given this {$1, $2} Pig creates this {($1), ($2)} a bag with two tuples

... neither $1 and $2 are tuples so Pig creates a tuple around each item

* + Given this {($1), $2} Pig creates this {($1), ($2)} a bag with two tuples

... since ($1) is treated as $1 (one cannot create a single element tuple using this syntax), {($1), $2} becomes {$1, $2} and Pig creates a tuple around each item

* + Given this {($1, $2)} Pig creates this {($1, $2)} a bag with a single tuple

... Pig creates a tuple ($1, $2) and then puts this tuple into the bag

#### Examples

**Tuple Construction**

A = load 'students' as (name:chararray, age:int, gpa:float);

B = foreach A generate (name, age);

store B into 'results';

Input (students):

joe smith 20 3.5

amy chen 22 3.2

leo allen 18 2.1

Output (results):

(joe smith,20)

(amy chen,22)

(leo allen,18)

**Bag Construction**

A = load 'students' as (name:chararray, age:int, gpa:float);

B = foreach A generate {(name, age)}, {name, age};

store B into 'results';

Input (students):

joe smith 20 3.5

amy chen 22 3.2

leo allen 18 2.1

Output (results):

{(joe smith,20)} {(joe smith),(20)}

{(amy chen,22)} {(amy chen),(22)}

{(leo allen,18)} {(leo allen),(18)}

**Map Construction**

A = load 'students' as (name:chararray, age:int, gpa:float);

B = foreach A generate [name, gpa];

store B into 'results';

Input (students):

joe smith 20 3.5

amy chen 22 3.2

leo allen 18 2.1

Output (results):

[joe smith#3.5]

[amy chen#3.2]

[leo allen#2.1]

### Dereference Operators

#### Description

|  |  |  |
| --- | --- | --- |
| Operator | Symbol | Notes |
| tuple dereference | tuple.id or tuple.(id,…) | Tuple dereferencing can be done by name (tuple.field\_name) or position (mytuple.$0). If a set of fields are dereferenced (tuple.(name1, name2) or tuple.($0, $1)), the expression represents a tuple composed of the specified fields. Note that if the dot operator is applied to a bytearray, the bytearray will be assumed to be a tuple. |
| bag dereference | bag.id or bag.(id,…) | Bag dereferencing can be done by name (bag.field\_name) or position (bag.$0). If a set of fields are dereferenced (bag.(name1, name2) or bag.($0, $1)), the expression represents a bag composed of the specified fields. |
| map dereference | map#'key' | Map dereferencing must be done by key (field\_name#key or $0#key). If the pound operator is applied to a bytearray, the bytearray is assumed to be a map. If the key does not exist, the empty string is returned. |

#### Examples

**Tuple Example**

Suppose we have relation A.

A = LOAD 'data' as (f1:int, f2:tuple(t1:int,t2:int,t3:int));

DUMP A;

(1,(1,2,3))

(2,(4,5,6))

(3,(7,8,9))

(4,(1,4,7))

(5,(2,5,8))

In this example dereferencing is used to retrieve two fields from tuple f2.

X = FOREACH A GENERATE f2.t1,f2.t3;

DUMP X;

(1,3)

(4,6)

(7,9)

(1,7)

(2,8)

**Bag Example**

Suppose we have relation B, formed by grouping relation A (see the GROUP operator for information about the field names in relation B).

A = LOAD 'data' AS (f1:int, f2:int,f3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

B = GROUP A BY f1;

DUMP B;

(1,{(1,2,3)})

(4,{(4,2,1),(4,3,3)})

(7,{(7,2,5)})

(8,{(8,3,4),(8,4,3)})

ILLUSTRATE B;

etc …

----------------------------------------------------------

| b | group: int | a: bag({f1: int,f2: int,f3: int}) |

----------------------------------------------------------

In this example dereferencing is used with relation X to project the first field (f1) of each tuple in the bag (a).

X = FOREACH B GENERATE a.f1;

DUMP X;

({(1)})

({(4),(4)})

({(7)})

({(8),(8)})

**Tuple/Bag Example**

Suppose we have relation B, formed by grouping relation A  (see the GROUP operator for information about the field names in relation B).

A = LOAD 'data' AS (f1:int, f2:int, f3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

B = GROUP A BY (f1,f2);

DUMP B;

((1,2),{(1,2,3)})

((4,2),{(4,2,1)})

((4,3),{(4,3,3)})

((7,2),{(7,2,5)})

((8,3),{(8,3,4)})

((8,4),{(8,4,3)})

ILLUSTRATE B;

etc …

-------------------------------------------------------------------------------

| b | group: tuple({f1: int,f2: int}) | a: bag({f1: int,f2: int,f3: int}) |

-------------------------------------------------------------------------------

| | (8, 3) | {(8, 3, 4), (8, 3, 4)} |

-------------------------------------------------------------------------------

In this example dereferencing is used to project a field (f1) from a tuple (group) and a field (f1) from a bag (a).

X = FOREACH B GENERATE group.f1, a.f1;

DUMP X;

(1,{(1)})

(4,{(4)})

(4,{(4)})

(7,{(7)})

(8,{(8)})

(8,{(8)})

**Map Example**

Suppose we have relation A.

A = LOAD 'data' AS (f1:int, f2:map[]);

DUMP A;

(1,[open#apache])

(2,[apache#hadoop])

(3,[hadoop#pig])

(4,[pig#grunt])

In this example dereferencing is used to look up the value of key 'open'.

X = FOREACH A GENERATE f2#'open';

DUMP X;

(apache)

()

()

()

### Disambiguate Operator

Use the disambiguate operator ( :: ) to identify field names after JOIN, COGROUP, CROSS, or FLATTEN operators.

In this example, to disambiguate y, use A::y or B::y. In cases where there is no ambiguity, such as z, the :: is not necessary but is still supported.

A = load 'data1' as (x, y);

B = load 'data2' as (x, y, z);

C = join A by x, B by x;

D = foreach C generate y; -- which y?

### Flatten Operator

The FLATTEN operator looks like a UDF syntactically, but it is actually an operator that changes the structure of tuples and bags in a way that a UDF cannot. Flatten un-nests tuples as well as bags. The idea is the same, but the operation and result is different for each type of structure.

For tuples, flatten substitutes the fields of a tuple in place of the tuple. For example, consider a relation that has a tuple of the form (a, (b, c)). The expression GENERATE $0, flatten($1), will cause that tuple to become (a, b, c).

For bags, the situation becomes more complicated. When we un-nest a bag, we create new tuples. If we have a relation that is made up of tuples of the form ({(b,c),(d,e)}) and we apply GENERATE flatten($0), we end up with two tuples (b,c) and (d,e). When we remove a level of nesting in a bag, sometimes we cause a cross product to happen. For example, consider a relation that has a tuple of the form (a, {(b,c), (d,e)}), commonly produced by the GROUP operator. If we apply the expression GENERATE $0, flatten($1) to this tuple, we will create new tuples: (a, b, c) and (a, d, e).

Also note that the flatten of empty bag will result in that row being discarded; no output is generated. (See also [Drop Nulls Before a Join](http://pig.apache.org/docs/r0.15.0/perf.html#nulls).)

grunt> cat empty.bag

{} 1

grunt> A = LOAD 'empty.bag' AS (b : bag{}, i : int);

grunt> B = FOREACH A GENERATE flatten(b), i;

grunt> DUMP B;

grunt>

For examples using the FLATTEN operator, see [FOREACH](http://pig.apache.org/docs/r0.15.0/basic.html#flatten-example).

### Null Operators

#### Description

|  |  |  |
| --- | --- | --- |
| Operator | Symbol | Notes |
| is null | is null |  |
| is not null | is not null |  |

For a detailed discussion of nulls see [Nulls and Pig Latin](http://pig.apache.org/docs/r0.15.0/basic.html#nulls).

#### Examples

In this example, values that are not null are obtained.

X = FILTER A BY f1 is not null;

#### Types Table

The null operators can be applied to all data types (see [Nulls and Pig Latin](http://pig.apache.org/docs/r0.15.0/basic.html#nulls)).

### Sign Operators

#### Description

|  |  |  |
| --- | --- | --- |
| Operator | Symbol | Notes |
| positive | + | Has no effect. |
| negative (negation) | - | Changes the sign of a positive or negative number. |

#### Examples

In this example, the negation operator is applied to the "x" values.

A = LOAD 'data' as (x, y, z);

B = FOREACH A GENERATE -x, y;

#### Types Table: negative ( - ) operator

|  |  |
| --- | --- |
| bag | error |
| tuple | error |
| map | error |
| int | int |
| long | long |
| float | float |
| double | double |
| chararray | error |
| bytearray | double (as double) |
| datetime | error |
| biginteger | biginteger |
| bigdecimal | bigdecimal |

## Relational Operators

### ASSERT

Assert a condition on the data.

#### Syntax

|  |
| --- |
| ASSERT alias BY expression [, message]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of the relation. |
| BY | Required keyword. |
| expression | A boolean expression. |
| message | Error message when assertion fails. |

#### Usage

Use assert to ensure a condition is true on your data. Processing fails if any of the records voilate the condition.

#### Examples

Suppose we have relation A.

A = LOAD 'data' AS (a0:int,a1:int,a2:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

Now, you can assert that a0 column in your data is >0, fail if otherwise

ASSERT A by a0 > 0, 'a0 should be greater than 0';

### COGROUP

See the [GROUP](http://pig.apache.org/docs/r0.15.0/basic.html#group) operator.

### CROSS

Computes the cross product of two or more relations.

#### Syntax

|  |
| --- |
| alias = CROSS alias, alias [, alias …] [PARTITION BY partitioner] [PARALLEL n]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| PARTITION BY partitioner | Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs.   * For more details, see <http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/Partitioner.html> * For usage, see [Example: PARTITION BY](http://pig.apache.org/docs/r0.15.0/basic.html#partitionby) |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.  For more information, see [Use the Parallel Features](http://pig.apache.org/docs/r0.15.0/perf.html#parallel). |

#### Usage

Use the CROSS operator to compute the cross product (Cartesian product) of two or more relations.

CROSS is an expensive operation and should be used sparingly.

#### Example

Suppose we have relations A and B.

A = LOAD 'data1' AS (a1:int,a2:int,a3:int);

DUMP A;

(1,2,3)

(4,2,1)

B = LOAD 'data2' AS (b1:int,b2:int);

DUMP B;

(2,4)

(8,9)

(1,3)

In this example the cross product of relation A and B is computed.

X = CROSS A, B;

DUMP X;

(1,2,3,2,4)

(1,2,3,8,9)

(1,2,3,1,3)

(4,2,1,2,4)

(4,2,1,8,9)

(4,2,1,1,3)

### CUBE

Performs cube/rollup operations.

#### Cube operation

Cube operation computes aggregates for all possbile combinations of specified group by dimensions. The number of group by combinations generated by cube for n dimensions will be 2^n.

#### Rollup operation

Rollup operations computes multiple levels of aggregates based on hierarchical ordering of specified group by dimensions. Rollup is useful when there is hierarchical ordering on the dimensions. The number of group by combinations generated by rollup for n dimensions will be n+1.

#### Syntax

|  |
| --- |
| alias = CUBE alias BY { CUBE expression | ROLLUP expression }, [ CUBE expression | ROLLUP expression ] [PARALLEL n]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of the relation. |
| CUBE | Keyword |
| BY | Keyword |
| expression | Projections (dimensions) of the relation. Supports field, star and project-range expressions. |
| ROLLUP | Keyword |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.  For more information, see [Use the Parallel Features](http://pig.apache.org/docs/r0.15.0/perf.html#parallel). |

#### Example

#### Basic usage of CUBE operation

salesinp = LOAD '/pig/data/salesdata' USING PigStorage(',') AS

(product:chararray, year:int, region:chararray, state:chararray, city:chararray, sales:long);

cubedinp = CUBE salesinp BY CUBE(product,year);

result = FOREACH cubedinp GENERATE FLATTEN(group), SUM(cube.sales) AS totalsales;

For a sample input tuple (car, 2012, midwest, ohio, columbus, 4000), the above query with cube operation will output

(car,2012,4000)

(car,,4000)

(,2012,4000)

(,,4000)

#### Output schema

grunt> describe cubedinp;

cubedinp: {group: (product: chararray,year: int),cube: {(product: chararray,year: int,region: chararray,

state: chararray,city: chararray,sales: long)}}

Note the second column, ‘cube’ field which is a bag of all tuples that belong to ‘group’. Also note that the measure attribute ‘sales’ along with other unused dimensions in load statement are pushed down so that it can be referenced later while computing aggregates on the measure, like in this case SUM(cube.sales).

#### Basic usage of ROLLUP operation

salesinp = LOAD '/pig/data/salesdata' USING PigStorage(',') AS

(product:chararray, year:int, region:chararray, state:chararray, city:chararray, sales:long);

rolledup = CUBE salesinp BY ROLLUP(region,state,city);

result = FOREACH rolledup GENERATE FLATTEN(group), SUM(cube.sales) AS totalsales;

For a sample input tuple (car, 2012, midwest, ohio, columbus, 4000), the above query with rollup operation will output

(midwest,ohio,columbus,4000)

(midwest,ohio,,4000)

(midwest,,,4000)

(,,,4000)

#### Output schema

grunt> describe rolledup;

rolledup: {group: (region: chararray,state: chararray,city: chararray),cube: {(region: chararray,

state: chararray,city: chararray,product: chararray,year: int,sales: long)}}

#### Basic usage of CUBE and ROLLUP operation combined

If CUBE and ROLLUP operations are used together, the output groups will be the cross product of all groups generated by cube and rollup operation. If there are m dimensions in cube operations and n dimensions in rollup operation then overall number of combinations will be (2^m) \* (n+1).

salesinp = LOAD '/pig/data/salesdata' USING PigStorage(',') AS

(product:chararray, year:int, region:chararray, state:chararray, city:chararray, sales:long);

cubed\_and\_rolled = CUBE salesinp BY CUBE(product,year), ROLLUP(region, state, city);

result = FOREACH cubed\_and\_rolled GENERATE FLATTEN(group), SUM(cube.sales) AS totalsales;

For a sample input tuple (car, 2012, midwest, ohio, columbus, 4000), the above query with cube and rollup operation will output

(car,2012,midwest,ohio,columbus,4000)

(car,2012,midwest,ohio,,4000)

(car,2012,midwest,,,4000)

(car,2012,,,,4000)

(car,,midwest,ohio,columbus,4000)

(car,,midwest,ohio,,4000)

(car,,midwest,,,4000)

(car,,,,,4000)

(,2012,midwest,ohio,columbus,4000)

(,2012,midwest,ohio,,4000)

(,2012,midwest,,,4000)

(,2012,,,,4000)

(,,midwest,ohio,columbus,4000)

(,,midwest,ohio,,4000)

(,,midwest,,,4000)

(,,,,,4000)

#### Output schema

grunt> describe cubed\_and\_rolled;

cubed\_and\_rolled: {group: (product: chararray,year: int,region: chararray,

state: chararray,city: chararray),cube: {(product: chararray,year: int,region: chararray,

state: chararray,city: chararray,sales: long)}}

#### Handling null values in dimensions

Since null values are used to represent subtotals in cube and rollup operation, in order to differentiate the legitimate null values that already exists as dimension values, CUBE operator converts any null values in dimensions to "unknown" value before performing cube or rollup operation. For example, for CUBE(product,location) with a sample tuple (car,) the output will be

(car,unknown)

(car,)

(,unknown)

(,)

### DEFINE

See:

* [DEFINE (UDFs, streaming)](http://pig.apache.org/docs/r0.15.0/basic.html#define-udfs)
* [DEFINE (macros)](http://pig.apache.org/docs/r0.15.0/cont.html#define-macros)

### DISTINCT

Removes duplicate tuples in a relation.

#### Syntax

|  |
| --- |
| alias = DISTINCT alias [PARTITION BY partitioner] [PARALLEL n]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of the relation. |
| PARTITION BY partitioner | Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs.   * For more details, see <http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/Partitioner.html> * For usage, see [Example: PARTITION BY](http://pig.apache.org/docs/r0.15.0/basic.html#partitionby). |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.  For more information, see [Use the Parallel Features](http://pig.apache.org/docs/r0.15.0/perf.html#parallel). |

#### Usage

Use the DISTINCT operator to remove duplicate tuples in a relation. DISTINCT does not preserve the original order of the contents (to eliminate duplicates, Pig must first sort the data). You cannot use DISTINCT on a subset of fields; to do this, use FOREACH and a nested block to first select the fields and then apply DISTINCT (see [Example: Nested Block](http://pig.apache.org/docs/r0.15.0/basic.html#nestedblock)).

#### Example

Suppose we have relation A.

A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;

(8,3,4)

(1,2,3)

(4,3,3)

(4,3,3)

(1,2,3)

In this example all duplicate tuples are removed.

X = DISTINCT A;

DUMP X;

(1,2,3)

(4,3,3)

(8,3,4)

### FILTER

Selects tuples from a relation based on some condition.

#### Syntax

|  |
| --- |
| alias = FILTER alias  BY expression; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of the relation. |
| BY | Required keyword. |
| expression | A boolean expression. |

#### Usage

Use the FILTER operator to work with tuples or rows of data (if you want to work with columns of data, use the FOREACH...GENERATE operation).

FILTER is commonly used to select the data that you want; or, conversely, to filter out (remove) the data you don’t want.

#### Examples

Suppose we have relation A.

A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

In this example the condition states that if the third field equals 3, then include the tuple with relation X.

X = FILTER A BY f3 == 3;

DUMP X;

(1,2,3)

(4,3,3)

(8,4,3)

In this example the condition states that if the first field equals 8 or if the sum of fields f2 and f3 is not greater than first field, then include the tuple relation X.

X = FILTER A BY (f1 == 8) OR (NOT (f2+f3 > f1));

DUMP X;

(4,2,1)

(8,3,4)

(7,2,5)

(8,4,3)

### FOREACH

Generates data transformations based on columns of data.

#### Syntax

|  |
| --- |
| alias  = FOREACH { block | nested\_block }; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of relation (outer bag). |
| block | FOREACH…GENERATE block used with a relation (outer bag). Use this syntax:  alias = FOREACH alias GENERATE expression [AS schema] [expression [AS schema]….];  See [Schemas](http://pig.apache.org/docs/r0.15.0/basic.html#schemas) |
| nested\_block | Nested FOREACH...GENERATE block used with a inner bag. Use this syntax:  alias = FOREACH nested\_alias {     alias = {nested\_op | nested\_exp}; [{alias = {nested\_op | nested\_exp}; …]     GENERATE expression [AS schema] [expression [AS schema]….]  };  Where:  The nested block is enclosed in opening and closing brackets { … }.  The GENERATE keyword must be the last statement within the nested block.  See [Schemas](http://pig.apache.org/docs/r0.15.0/basic.html#schemas)  Macros are NOT alllowed inside a nested block. |
| expression | An expression. |
| nested\_alias | The name of the inner bag. |
| nested\_op | Allowed operations are CROSS, DISTINCT, FILTER, FOREACH, LIMIT, and ORDER BY.  Note: FOREACH statements can be nested to two levels only. FOREACH statements that are nested to three or more levels will result in a grammar error.  You can also perform projections within the nested block.  For examples, see [Example: Nested Block](http://pig.apache.org/docs/r0.15.0/basic.html#nestedblock). |
| nested\_exp | Any arbitrary, supported expression. |
| AS | Keyword |
| schema | A schema using the AS keyword (see [Schemas](http://pig.apache.org/docs/r0.15.0/basic.html#schemas)).   * If the [FLATTEN](http://pig.apache.org/docs/r0.15.0/basic.html#flatten) operator is used, enclose the schema in parentheses. * If the FLATTEN operator is not used, don't enclose the schema in parentheses. |

#### Usage

Use the FOREACH…GENERATE operation to work with columns of data (if you want to work with tuples or rows of data, use the FILTER operation).

FOREACH...GENERATE works with relations (outer bags) as well as inner bags:

* If A is a relation (outer bag), a FOREACH statement could look like this.
* X = FOREACH A GENERATE f1;
* If A is an inner bag, a FOREACH statement could look like this.
* X = FOREACH B {
* S = FILTER A BY 'xyz';
* GENERATE COUNT (S.$0);
* }

#### Example: Projection

In this example the asterisk (\*) is used to project all fields from relation A to relation X. Relation A and X are identical.

X = FOREACH A GENERATE \*;

DUMP X;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

In this example two fields from relation A are projected to form relation X.

X = FOREACH A GENERATE a1, a2;

DUMP X;

(1,2)

(4,2)

(8,3)

(4,3)

(7,2)

(8,4)

#### Example: Nested Projection

In this example if one of the fields in the input relation is a tuple, bag or map, we can perform a projection on that field (using a deference operator).

X = FOREACH C GENERATE group, B.b2;

DUMP X;

(1,{(3)})

(4,{(6),(9)})

(8,{(9)})

In this example multiple nested columns are retained.

X = FOREACH C GENERATE group, A.(a1, a2);

DUMP X;

(1,{(1,2)})

(4,{(4,2),(4,3)})

(8,{(8,3),(8,4)})

#### Example: Schema

In this example two fields in relation A are summed to form relation X. A schema is defined for the projected field.

X = FOREACH A GENERATE a1+a2 AS f1:int;

DESCRIBE X;

x: {f1: int}

DUMP X;

(3)

(6)

(11)

(7)

(9)

(12)

Y = FILTER X BY f1 > 10;

DUMP Y;

(11)

(12)

#### Example: Applying Functions

In this example the built in function SUM() is used to sum a set of numbers in a bag.

X = FOREACH C GENERATE group, SUM (A.a1);

DUMP X;

(1,1)

(4,8)

(8,16)

#### Example: Flatten

In this example the [FLATTEN](http://pig.apache.org/docs/r0.15.0/basic.html#flatten) operator is used to eliminate nesting.

X = FOREACH C GENERATE group, FLATTEN(A);

DUMP X;

(1,1,2,3)

(4,4,2,1)

(4,4,3,3)

(8,8,3,4)

(8,8,4,3)

Another FLATTEN example.

X = FOREACH C GENERATE GROUP, FLATTEN(A.a3);

DUMP X;

(1,3)

(4,1)

(4,3)

(8,4)

(8,3)

Another FLATTEN example. Note that for the group '4' in C, there are two tuples in each bag. Thus, when both bags are flattened, the cross product of these tuples is returned; that is, tuples (4, 2, 6), (4, 3, 6), (4, 2, 9), and (4, 3, 9).

X = FOREACH C GENERATE FLATTEN(A.(a1, a2)), FLATTEN(B.$1);

DUMP X;

(1,2,3)

(4,2,6)

(4,2,9)

(4,3,6)

(4,3,9)

(8,3,9)

(8,4,9)

Another FLATTEN example. Here, relations A and B both have a column x. When forming relation E, you need to use the :: operator to identify which column x to use - either relation A column x (A::x) or relation B column x (B::x). This example uses relation A column x (A::x).

A = LOAD 'data' AS (x, y);

B = LOAD 'data' AS (x, z);

C = COGROUP A BY x, B BY x;

D = FOREACH C GENERATE flatten(A), flatten(b);

E = GROUP D BY A::x;

……

#### Example: Nested Block

In this example a CROSS is performed within the nested block.

user = load 'user' as (uid, age, gender, region);

session = load 'session' as (uid, region);

C = cogroup user by uid, session by uid;

D = foreach C {

crossed = cross user, session;

generate crossed;

}

dump D;

In this example FOREACH is nested to the second level.

a = load '1.txt' as (a0, a1:chararray, a2:chararray);

b = group a by a0;

c = foreach b {

c0 = foreach a generate TOMAP(a1,a2);

generate c0;

}

dump c;

This example shows a CROSS and FOREACH nested to the second level.

a = load '1.txt' as (a0, a1, a2);

b = load '2.txt' as (b0, b1);

c = cogroup a by a0, b by b0;

d = foreach c {

d0 = cross a, b;

d1 = foreach d0 generate a1+b1;

generate d1;

}

dump d;

Suppose we have relations A and B. Note that relation B contains an inner bag.

A = LOAD 'data' AS (url:chararray,outlink:chararray);

DUMP A;

(www.ccc.com,www.hjk.com)

(www.ddd.com,www.xyz.org)

(www.aaa.com,www.cvn.org)

(www.www.com,www.kpt.net)

(www.www.com,www.xyz.org)

(www.ddd.com,www.xyz.org)

B = GROUP A BY url;

DUMP B;

(www.aaa.com,{(www.aaa.com,www.cvn.org)})

(www.ccc.com,{(www.ccc.com,www.hjk.com)})

(www.ddd.com,{(www.ddd.com,www.xyz.org),(www.ddd.com,www.xyz.org)})

(www.www.com,{(www.www.com,www.kpt.net),(www.www.com,www.xyz.org)})

In this example we perform two of the operations allowed in a nested block, FILTER and DISTINCT. Note that the last statement in the nested block must be GENERATE. Also, note the use of projection (PA = FA.outlink;) to retrieve a field. DISTINCT can be applied to a subset of fields (as opposed to a relation) only within a nested block.

X = FOREACH B {

FA= FILTER A BY outlink == 'www.xyz.org';

PA = FA.outlink;

DA = DISTINCT PA;

GENERATE group, COUNT(DA);

}

DUMP X;

(www.aaa.com,0)

(www.ccc.com,0)

(www.ddd.com,1)

(www.www.com,1)

### GROUP

Groups the data in one or more relations.

Note: The GROUP and COGROUP operators are identical. Both operators work with one or more relations. For readability GROUP is used in statements involving one relation and COGROUP is used in statements involving two or more relations. You can COGROUP up to but no more than 127 relations at a time.

#### Syntax

|  |
| --- |
| alias = GROUP alias { ALL | BY expression} [, alias ALL | BY expression …] [USING 'collected' | 'merge'] [PARTITION BY partitioner] [PARALLEL n]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation.  You can COGROUP up to but no more than 127 relations at a time. |
| ALL | Keyword. Use ALL if you want all tuples to go to a single group; for example, when doing aggregates across entire relations.  B = GROUP A ALL; |
| BY | Keyword. Use this clause to group the relation by field, tuple or expression.  B = GROUP A BY f1; |
| expression | A tuple expression. This is the group key or key field. If the result of the tuple expression is a single field, the key will be the value of the first field rather than a tuple with one field. To group using multiple keys, enclose the keys in parentheses:  B = GROUP A BY (key1,key2); |
| USING | Keyword |
| 'collected' | Use the ‘collected’ clause with the GROUP operation (works with one relation only).  The following conditions apply:   * The loader must implement the {CollectableLoader} interface. * Data must be sorted on the group key.   If your data and loaders satisfy these conditions, use the ‘collected’ clause to perform an optimized version of GROUP; the operation will execute on the map side and avoid running the reduce phase. |
| 'merge' | Use the ‘merge’ clause with the COGROUP operation (works with two or more relations only).  The following conditions apply:   * No other operations can be done between the LOAD and COGROUP statements. * Data must be sorted on the COGROUP key for all tables in ascending (ASC) order. * Nulls are considered smaller than evertyhing. If data contains null keys, they should occur before anything else. * Left-most loader must implement the {CollectableLoader} interface as well as {OrderedLoadFunc} interface. * All other loaders must implement IndexableLoadFunc. * Type information must be provided in the schema for all the loaders.   If your data and loaders satisfy these conditions, the ‘merge’ clause to perform an optimized version of COGROUP; the operation will execute on the map side and avoid running the reduce phase. |
| PARTITION BY partitioner | Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs.   * For more details, see <http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/Partitioner.html> * For usage, see [Example: PARTITION BY](http://pig.apache.org/docs/r0.15.0/basic.html#partitionby) |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.  For more information, see [Use the Parallel Features](http://pig.apache.org/docs/r0.15.0/perf.html#Parallel). |

#### Usage

The GROUP operator groups together tuples that have the same group key (key field). The key field will be a tuple if the group key has more than one field, otherwise it will be the same type as that of the group key. The result of a GROUP operation is a relation that includes one tuple per group. This tuple contains two fields:

* The first field is named "group" (do not confuse this with the GROUP operator) and is the same type as the group key.
* The second field takes the name of the original relation and is type bag.
* The names of both fields are generated by the system as shown in the example below.

Note the following about the GROUP/COGROUP and JOIN operators:

* The GROUP and JOIN operators perform similar functions. GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples
* The GROUP/COGROUP and JOIN operators handle null values differently (see [Nulls and GROUP/COGROUP Operataors](http://pig.apache.org/docs/r0.15.0/basic.html#nulls_group)).

#### Example

Suppose we have relation A.

A = load 'student' AS (name:chararray,age:int,gpa:float);

DESCRIBE A;

A: {name: chararray,age: int,gpa: float}

DUMP A;

(John,18,4.0F)

(Mary,19,3.8F)

(Bill,20,3.9F)

(Joe,18,3.8F)

Now, suppose we group relation A on field "age" for form relation B. We can use the DESCRIBE and ILLUSTRATE operators to examine the structure of relation B. Relation B has two fields. The first field is named "group" and is type int, the same as field "age" in relation A. The second field is name "A"  after relation A and is type bag.

B = GROUP A BY age;

DESCRIBE B;

B: {group: int, A: {name: chararray,age: int,gpa: float}}

ILLUSTRATE B;

etc ...

----------------------------------------------------------------------

| B | group: int | A: bag({name: chararray,age: int,gpa: float}) |

----------------------------------------------------------------------

| | 18 | {(John, 18, 4.0), (Joe, 18, 3.8)} |

| | 20 | {(Bill, 20, 3.9)} |

----------------------------------------------------------------------

DUMP B;

(18,{(John,18,4.0F),(Joe,18,3.8F)})

(19,{(Mary,19,3.8F)})

(20,{(Bill,20,3.9F)})

Continuing on, as shown in these FOREACH statements, we can refer to the fields in relation B by names "group" and "A" or by positional notation.

C = FOREACH B GENERATE group, COUNT(A);

DUMP C;

(18,2L)

(19,1L)

(20,1L)

C = FOREACH B GENERATE $0, $1.name;

DUMP C;

(18,{(John),(Joe)})

(19,{(Mary)})

(20,{(Bill)})

#### Example

Suppose we have relation A.

A = LOAD 'data' as (f1:chararray, f2:int, f3:int);

DUMP A;

(r1,1,2)

(r2,2,1)

(r3,2,8)

(r4,4,4)

In this example the tuples are grouped using an expression, f2\*f3.

X = GROUP A BY f2\*f3;

DUMP X;

(2,{(r1,1,2),(r2,2,1)})

(16,{(r3,2,8),(r4,4,4)})

#### Example

Suppose we have two relations, A and B.

A = LOAD 'data1' AS (owner:chararray,pet:chararray);

DUMP A;

(Alice,turtle)

(Alice,goldfish)

(Alice,cat)

(Bob,dog)

(Bob,cat)

B = LOAD 'data2' AS (friend1:chararray,friend2:chararray);

DUMP B;

(Cindy,Alice)

(Mark,Alice)

(Paul,Bob)

(Paul,Jane)

In this example tuples are co-grouped using field “owner” from relation A and field “friend2” from relation B as the key fields. The DESCRIBE operator shows the schema for relation X, which has three fields, "group", "A" and "B" (see the GROUP operator for information about the field names).

X = COGROUP A BY owner, B BY friend2;

DESCRIBE X;

X: {group: chararray,A: {owner: chararray,pet: chararray},B: {friend1: chararray,friend2: chararray}}

Relation X looks like this. A tuple is created for each unique key field. The tuple includes the key field and two bags. The first bag is the tuples from the first relation with the matching key field. The second bag is the tuples from the second relation with the matching key field. If no tuples match the key field, the bag is empty.

(Alice,{(Alice,turtle),(Alice,goldfish),(Alice,cat)},{(Cindy,Alice),(Mark,Alice)})

(Bob,{(Bob,dog),(Bob,cat)},{(Paul,Bob)})

(Jane,{},{(Paul,Jane)})

#### Example

This example shows how to group using multiple keys.

A = LOAD 'allresults' USING PigStorage() AS (tcid:int, tpid:int, date:chararray, result:chararray, tsid:int, tag:chararray);

B = GROUP A BY (tcid, tpid);

#### Example: PARTITION BY

To use the Hadoop Partitioner add PARTITION BY clause to the appropriate operator:

A = LOAD 'input\_data';

B = GROUP A BY $0 PARTITION BY org.apache.pig.test.utils.SimpleCustomPartitioner PARALLEL 2;

Here is the code for SimpleCustomPartitioner:

public class SimpleCustomPartitioner extends Partitioner <PigNullableWritable, Writable> {

//@Override

public int getPartition(PigNullableWritable key, Writable value, int numPartitions) {

if(key.getValueAsPigType() instanceof Integer) {

int ret = (((Integer)key.getValueAsPigType()).intValue() % numPartitions);

return ret;

}

else {

return (key.hashCode()) % numPartitions;

}

}

}

### IMPORT

See [IMPORT (macros)](http://pig.apache.org/docs/r0.15.0/cont.html#import-macros)

### JOIN (inner)

Performs an inner join of two or more relations based on common field values.

#### Syntax

|  |
| --- |
| alias = JOIN alias BY {expression|'('expression [, expression …]')'} (, alias BY {expression|'('expression [, expression …]')'} …) [USING 'replicated' | 'skewed' | 'merge' | 'merge-sparse'] [PARTITION BY partitioner] [PARALLEL n]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| BY | Keyword |
| expression | A field expression.  Example: X = JOIN A BY fieldA, B BY fieldB, C BY fieldC; |
| USING | Keyword |
| 'replicated' | Use to perform replicated joins (see [Replicated Joins](http://pig.apache.org/docs/r0.15.0/perf.html#replicated-joins)). |
| 'skewed' | Use to perform skewed joins (see [Skewed Joins](http://pig.apache.org/docs/r0.15.0/perf.html#skewed-joins)). |
| 'merge' | Use to perform merge joins (see [Merge Joins](http://pig.apache.org/docs/r0.15.0/perf.html#merge-joins)). |
| 'merge-sparse' | Use to perform merge-sparse joins (see [Merge-Sparse Joins](http://pig.apache.org/docs/r0.15.0/perf.html#merge-sparse-joins)). |
| PARTITION BY partitioner | Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs.   * For more details, see <http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/Partitioner.html> * For usage, see [Example: PARTITION BY](http://pig.apache.org/docs/r0.15.0/basic.html#partitionby)   This feature CANNOT be used with skewed joins. |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.  For more information, see [Use the Parallel Features](http://pig.apache.org/docs/r0.15.0/perf.html#parallel). |

#### Usage

Use the JOIN operator to perform an inner, equijoin join of two or more relations based on common field values. Inner joins ignore null keys, so it makes sense to filter them out before the join.

Note the following about the GROUP/COGROUP and JOIN operators:

* The GROUP and JOIN operators perform similar functions. GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples.
* The GROUP/COGROUP and JOIN operators handle null values differently (see [Nulls and JOIN Operator](http://pig.apache.org/docs/r0.15.0/basic.html#nulls_join)).

**Self Joins**

To perform self joins in Pig load the same data multiple times, under different aliases, to avoid naming conflicts.

In this example the same data is loaded twice using aliases A and B.

grunt> A = load 'mydata';

grunt> B = load 'mydata';

grunt> C = join A by $0, B by $0;

grunt> explain C;

#### Example

Suppose we have relations A and B.

A = LOAD 'data1' AS (a1:int,a2:int,a3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

B = LOAD 'data2' AS (b1:int,b2:int);

DUMP B;

(2,4)

(8,9)

(1,3)

(2,7)

(2,9)

(4,6)

(4,9)

In this example relations A and B are joined by their first fields.

X = JOIN A BY a1, B BY b1;

DUMP X;

(1,2,3,1,3)

(4,2,1,4,6)

(4,3,3,4,6)

(4,2,1,4,9)

(4,3,3,4,9)

(8,3,4,8,9)

(8,4,3,8,9)

### JOIN (outer)

Performs an outer join of two relations based on common field values.

#### Syntax

|  |
| --- |
| alias = JOIN left-alias BY left-alias-column [LEFT|RIGHT|FULL] [OUTER], right-alias BY right-alias-column [USING 'replicated' | 'skewed' | 'merge'] [PARTITION BY partitioner] [PARALLEL n]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. Applies to alias, left-alias and right-alias. |
| alias-column | The name of the join column for the corresponding relation. Applies to left-alias-column and right-alias-column. |
| BY | Keyword |
| LEFT | Left outer join. |
| RIGHT | Right outer join. |
| FULL | Full outer join. |
| OUTER | (Optional) Keyword |
| USING | Keyword |
| 'replicated' | Use to perform replicated joins (see [Replicated Joins](http://pig.apache.org/docs/r0.15.0/perf.html#replicated-joins)).  Only left outer join is supported for replicated joins. |
| 'skewed' | Use to perform skewed joins (see [Skewed Joins](http://pig.apache.org/docs/r0.15.0/perf.html#skewed-joins)). |
| 'merge' | Use to perform merge joins (see [Merge Joins](http://pig.apache.org/docs/r0.15.0/perf.html#merge-joins)). |
| PARTITION BY partitioner | Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs.   * For more details, see <http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/Partitioner.html> * For usage, see [Example: PARTITION BY](http://pig.apache.org/docs/r0.15.0/basic.html#partitionby)   This feature CANNOT be used with skewed joins. |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.  For more information, see [Use the Parallel Features](http://pig.apache.org/docs/r0.15.0/perf.html#parallel). |

#### Usage

Use the JOIN operator with the corresponding keywords to perform left, right, or full outer joins. The keyword OUTER is optional for outer joins; the keywords LEFT, RIGHT and FULL will imply left outer, right outer and full outer joins respectively when OUTER is omitted. The Pig Latin syntax closely adheres to the SQL standard.

Please note the following:

* Outer joins will only work provided the relations which need to produce nulls (in the case of non-matching keys) have schemas.
* Outer joins will only work for two-way joins; to perform a multi-way outer join, you will need to perform multiple two-way outer join statements.

#### Examples

This example shows a left outer join.

A = LOAD 'a.txt' AS (n:chararray, a:int);

B = LOAD 'b.txt' AS (n:chararray, m:chararray);

C = JOIN A by $0 LEFT OUTER, B BY $0;

This example shows a full outer join.

A = LOAD 'a.txt' AS (n:chararray, a:int);

B = LOAD 'b.txt' AS (n:chararray, m:chararray);

C = JOIN A BY $0 FULL, B BY $0;

This example shows a replicated left outer join.

A = LOAD 'large';

B = LOAD 'tiny';

C= JOIN A BY $0 LEFT, B BY $0 USING 'replicated';

This example shows a skewed full outer join.

A = LOAD 'studenttab' as (name, age, gpa);

B = LOAD 'votertab' as (name, age, registration, contribution);

C = JOIN A BY name FULL, B BY name USING 'skewed';

### LIMIT

Limits the number of output tuples.

#### Syntax

|  |
| --- |
| alias = LIMIT alias  n; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| n | The number of output tuples, either:   * a constant (for example, 3) * a scalar used in an expression (for example, c.sum/100)   Note: The expression can consist of constants or scalars; it cannot contain any columns from the input relation.  Note: Using a scalar instead of a constant in LIMIT automatically disables most optimizations (only push-before-foreach is performed). |

#### Usage

Use the LIMIT operator to limit the number of output tuples.

If the specified number of output tuples is equal to or exceeds the number of tuples in the relation, all tuples in the relation are returned.

If the specified number of output tuples is less than the number of tuples in the relation, then n tuples are returned. There is no guarantee which n tuples will be returned, and the tuples that are returned can change from one run to the next. A particular set of tuples can be requested using the ORDER operator followed by LIMIT.

Note: The LIMIT operator allows Pig to avoid processing all tuples in a relation. In most cases a query that uses LIMIT will run more efficiently than an identical query that does not use LIMIT. It is always a good idea to use limit if you can.

#### Examples

In this example the limit is expressed as a scalar.

a = load 'a.txt';

b = group a all;

c = foreach b generate COUNT(a) as sum;

d = order a by $0;

e = limit d c.sum/100;

Suppose we have relation A.

A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

In this example output is limited to 3 tuples. Note that there is no guarantee which three tuples will be output.

X = LIMIT A 3;

DUMP X;

(1,2,3)

(4,3,3)

(7,2,5)

In this example the ORDER operator is used to order the tuples and the LIMIT operator is used to output the first three tuples.

B = ORDER A BY f1 DESC, f2 ASC;

DUMP B;

(8,3,4)

(8,4,3)

(7,2,5)

(4,2,1)

(4,3,3)

(1,2,3)

X = LIMIT B 3;

DUMP X;

(8,3,4)

(8,4,3)

(7,2,5)

### LOAD

Loads data from the file system.

#### Syntax

|  |
| --- |
| LOAD 'data' [USING function] [AS schema]; |

#### Terms

|  |  |
| --- | --- |
| 'data' | The name of the file or directory, in single quotes.  If you specify a directory name, all the files in the directory are loaded.  You can use Hadoop globing to specify files at the file system or directory levels (see Hadoop [globStatus](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/fs/FileSystem.html#globStatus%28org.apache.hadoop.fs.Path%29) for details on globing syntax).  **Note:** Pig uses Hadoop globbing so the functionality is IDENTICAL. However, when you run from the command line using the Hadoop fs command (rather than the Pig LOAD operator), the Unix shell may do some of the substitutions; this could alter the outcome giving the impression that globing works differently for Pig and Hadoop. For example:   * This works  hadoop fs -ls /mydata/20110423{00,01,02,03,04,05,06,07,08,09,{10..23}}00//**part** * This does not work  LOAD '/mydata/20110423{00,01,02,03,04,05,06,07,08,09,{10..23}}00//**part** ' |
| USING | Keyword.  If the USING clause is omitted, the default load function PigStorage is used. |
| function | The load function.   * You can use a built in function (see [Load/Store Functions](http://pig.apache.org/docs/r0.15.0/func.html#load-store-functions)). PigStorage is the default load function and does not need to be specified (simply omit the USING clause). * You can write your own load function if your data is in a format that cannot be processed by the built in functions (see [User Defined Functions](http://pig.apache.org/docs/r0.15.0/udf.html)). |
| AS | Keyword. |
| schema | A schema using the AS keyword, enclosed in parentheses (see [Schemas](http://pig.apache.org/docs/r0.15.0/basic.html#schemas)).  The loader produces the data of the type specified by the schema. If the data does not conform to the schema, depending on the loader, either a null value or an error is generated.  Note: For performance reasons the loader may not immediately convert the data to the specified format; however, you can still operate on the data assuming the specified type. |

#### Usage

Use the LOAD operator to load data from the file system.

#### Examples

Suppose we have a data file called myfile.txt. The fields are tab-delimited. The records are newline-separated.

1 2 3

4 2 1

8 3 4

In this example the default load function, PigStorage, loads data from myfile.txt to form relation A. The two LOAD statements are equivalent. Note that, because no schema is specified, the fields are not named and all fields default to type bytearray.

A = LOAD 'myfile.txt';

A = LOAD 'myfile.txt' USING PigStorage('\t');

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

In this example a schema is specified using the AS keyword. The two LOAD statements are equivalent. You can use the DESCRIBE and ILLUSTRATE operators to view the schema.

A = LOAD 'myfile.txt' AS (f1:int, f2:int, f3:int);

A = LOAD 'myfile.txt' USING PigStorage('\t') AS (f1:int, f2:int, f3:int);

DESCRIBE A;

a: {f1: int,f2: int,f3: int}

ILLUSTRATE A;

---------------------------------------------------------

| a | f1: bytearray | f2: bytearray | f3: bytearray |

---------------------------------------------------------

| | 4 | 2 | 1 |

---------------------------------------------------------

---------------------------------------

| a | f1: int | f2: int | f3: int |

---------------------------------------

| | 4 | 2 | 1 |

---------------------------------------

For examples of how to specify more complex schemas for use with the LOAD operator, see [Schemas for Complex Data Types](http://pig.apache.org/docs/r0.15.0/basic.html#schema-complex) and [Schemas for Multiple Types](http://pig.apache.org/docs/r0.15.0/basic.html#schema-multi).

### MAPREDUCE

Executes native MapReduce jobs inside a Pig script.

#### Syntax

|  |
| --- |
| alias1 = MAPREDUCE 'mr.jar' STORE alias2 INTO 'inputLocation' USING storeFunc LOAD 'outputLocation' USING loadFunc AS schema [`params, ... `]; |

#### Terms

|  |  |
| --- | --- |
| alias1, alias2 | The names of relations. |
| mr.jar | The MapReduce jar file (enclosed in single quotes).  You can specify any MapReduce jar file that can be run through the hadoop jar mymr.jar params command.  The values for inputLocation and outputLocation can be passed in the params. |
| STORE ... INTO ... USING | See [STORE](http://pig.apache.org/docs/r0.15.0/basic.html#store)  Store alias2 into the inputLocation using storeFunc, which is then used by the MapReduce job to read its data. |
| LOAD ... USING ... AS | See [LOAD](http://pig.apache.org/docs/r0.15.0/basic.html#load)  After running mr.jar's MapReduce job, load back the data from outputLocation into alias1 using loadFunc as schema. |
| `params, ...` | Extra parameters required for the mapreduce job (enclosed in back tics). |

#### Usage

Use the MAPREDUCE operator to run native MapReduce jobs from inside a Pig script.

The input and output locations for the MapReduce program are conveyed to Pig using the STORE/LOAD clauses. Pig, however, does not pass this information (nor require that this information be passed) to the MapReduce program. If you want to pass the input and output locations to the MapReduce program you can use the params clause or you can hardcode the locations in the MapReduce program.

#### Example

This example demonstrates how to run the wordcount MapReduce progam from Pig. Note that the files specified as input and output locations in the MAPREDUCE statement will NOT be deleted by Pig automatically. You will need to delete them manually.

A = LOAD 'WordcountInput.txt';

B = MAPREDUCE 'wordcount.jar' STORE A INTO 'inputDir' LOAD 'outputDir'

AS (word:chararray, count: int) `org.myorg.WordCount inputDir outputDir`;

### ORDER BY

Sorts a relation based on one or more fields.

#### Syntax

|  |
| --- |
| alias = ORDER alias BY { \* [ASC|DESC] | field\_alias [ASC|DESC] [, field\_alias [ASC|DESC] …] } [PARALLEL n]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| \* | The designator for a tuple. |
| field\_alias | A field in the relation. The field must be a simple type. |
| ASC | Sort in ascending order. |
| DESC | Sort in descending order. |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.  For more information, see [Use the Parallel Features](http://pig.apache.org/docs/r0.15.0/perf.html#parallel). |

#### Usage

**Note:** ORDER BY is NOT stable; if multiple records have the same ORDER BY key, the order in which these records are returned is not defined and is not guarantted to be the same from one run to the next.

In Pig, relations are unordered (see [Relations, Bags, Tuples, Fields](http://pig.apache.org/docs/r0.15.0/basic.html#relations)):

* If you order relation A to produce relation X (X = ORDER A BY \* DESC;) relations A and X still contain the same data.
* If you retrieve relation X (DUMP X;) the data is guaranteed to be in the order you specified (descending).
* However, if you further process relation X (Y = FILTER X BY $0 > 1;) there is no guarantee that the data will be processed in the order you originally specified (descending).

Pig currently supports ordering on fields with simple types or by tuple designator (\*). You cannot order on fields with complex types or by expressions.

A = LOAD 'mydata' AS (x: int, y: map[]);

B = ORDER A BY x; -- this is allowed because x is a simple type

B = ORDER A BY y; -- this is not allowed because y is a complex type

B = ORDER A BY y#'id'; -- this is not allowed because y#'id' is an expression

#### Examples

Suppose we have relation A.

A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

In this example relation A is sorted by the third field, f3 in descending order. Note that the order of the three tuples ending in 3 can vary.

X = ORDER A BY a3 DESC;

DUMP X;

(7,2,5)

(8,3,4)

(1,2,3)

(4,3,3)

(8,4,3)

(4,2,1)

### RANK

Returns each tuple with the rank within a relation.

#### Syntax

|  |
| --- |
| alias = RANK alias [ BY { \* [ASC|DESC] | field\_alias [ASC|DESC] [, field\_alias [ASC|DESC] …] } [DENSE] ]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| \* | The designator for a tuple. |
| field\_alias | A field in the relation. The field must be a simple type. |
| ASC | Sort in ascending order. |
| DESC | Sort in descending order. |
| DENSE | No gap in the ranking values. |

#### Usage

When specifying no field to sort on, the RANK operator simply prepends a sequential value to each tuple.

Otherwise, the RANK operator uses each field (or set of fields) to sort the relation. The rank of a tuple is one plus the number of different rank values preceding it. If two or more tuples tie on the sorting field values, they will receive the same rank.

**NOTE:** When using the option **DENSE**, ties do not cause gaps in ranking values.

#### Examples

Suppose we have relation A.

A = load 'data' AS (f1:chararray,f2:int,f3:chararray);

DUMP A;

(David,1,N)

(Tete,2,N)

(Ranjit,3,M)

(Ranjit,3,P)

(David,4,Q)

(David,4,Q)

(Jillian,8,Q)

(JaePak,7,Q)

(Michael,8,T)

(Jillian,8,Q)

(Jose,10,V)

In this example, the RANK operator does not change the order of the relation and simply prepends to each tuple a sequential value.

B = rank A;

dump B;

(1,David,1,N)

(2,Tete,2,N)

(3,Ranjit,3,M)

(4,Ranjit,3,P)

(5,David,4,Q)

(6,David,4,Q)

(7,Jillian,8,Q)

(8,JaePak,7,Q)

(9,Michael,8,T)

(10,Jillian,8,Q)

(11,Jose,10,V)

In this example, the RANK operator works with f1 and f2 fields, and each one with different sorting order. RANK sorts the relation on these fields and prepends the rank value to each tuple. Otherwise, the RANK operator uses each field (or set of fields) to sort the relation. The rank of a tuple is one plus the number of different rank values preceding it. If two or more tuples tie on the sorting field values, they will receive the same rank.

C = rank A by f1 DESC, f2 ASC;

dump C;

(1,Tete,2,N)

(2,Ranjit,3,M)

(2,Ranjit,3,P)

(4,Michael,8,T)

(5,Jose,10,V)

(6,Jillian,8,Q)

(6,Jillian,8,Q)

(8,JaePak,7,Q)

(9,David,1,N)

(10,David,4,Q)

(10,David,4,Q)

Same example as previous, but DENSE. In this case there are no gaps in ranking values.

C = rank A by f1 DESC, f2 ASC DENSE;

dump C;

(1,Tete,2,N)

(2,Ranjit,3,M)

(2,Ranjit,3,P)

(3,Michael,8,T)

(4,Jose,10,V)

(5,Jillian,8,Q)

(5,Jillian,8,Q)

(6,JaePak,7,Q)

(7,David,1,N)

(8,David,4,Q)

(8,David,4,Q)

### SAMPLE

Selects a random sample of data based on the specified sample size.

#### Syntax

|  |
| --- |
| SAMPLE alias size; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| size | Sample size, either   * a constant, range 0 to 1 (for example, enter 0.1 for 10%) * a scalar used in an expression   Note: The expression can consist of constants or scalars; it cannot contain any columns from the input relation. |

#### Usage

Use the SAMPLE operator to select a random data sample with the stated sample size. SAMPLE is a probabalistic operator; there is no guarantee that the exact same number of tuples will be returned for a particular sample size each time the operator is used.

#### Example

In this example relation X will contain 1% of the data in relation A.

A = LOAD 'data' AS (f1:int,f2:int,f3:int);

X = SAMPLE A 0.01;

In this example, a scalar expression is used (it will sample approximately 1000 records from the input).

a = load 'a.txt';

b = group a all;

c = foreach b generate COUNT(a) as num\_rows;

e = sample a 1000/c.num\_rows;

### SPLIT

Partitions a relation into two or more relations.

#### Syntax

|  |
| --- |
| SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression …] [, alias OTHERWISE]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| INTO | Required keyword. |
| IF | Required keyword. |
| expression | An expression. |
| OTHERWISE | Optional keyword. Designates a default relation. |

#### Usage

Use the SPLIT operator to partition the contents of a relation into two or more relations based on some expression. Depending on the conditions stated in the expression:

* A tuple may be assigned to more than one relation.
* A tuple may not be assigned to any relation.

#### Example

In this example relation A is split into three relations, X, Y, and Z.

A = LOAD 'data' AS (f1:int,f2:int,f3:int);

DUMP A;

(1,2,3)

(4,5,6)

(7,8,9)

SPLIT A INTO X IF f1<7, Y IF f2==5, Z IF (f3<6 OR f3>6);

DUMP X;

(1,2,3)

(4,5,6)

DUMP Y;

(4,5,6)

DUMP Z;

(1,2,3)

(7,8,9)

#### Example

In this example, the SPLIT and FILTER statements are essentially equivalent. However, because SPLIT is implemented as "split the data stream and then apply filters" the SPLIT statement is more expensive than the FILTER statement because Pig needs to filter and store two data streams.

SPLIT input\_var INTO output\_var IF (field1 is not null), ignored\_var IF (field1 is null);

-- where ignored\_var is not used elsewhere

output\_var = FILTER input\_var BY (field1 is not null);

### STORE

Stores or saves results to the file system.

#### Syntax

|  |
| --- |
| STORE alias INTO 'directory' [USING function]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| INTO | Required keyword. |
| 'directory' | The name of the storage directory, in quotes. If the directory already exists, the STORE operation will fail.  The output data files, named part-nnnnn, are written to this directory. |
| USING | Keyword. Use this clause to name the store function.  If the USING clause is omitted, the default store function PigStorage is used. |
| function | The store function.   * You can use a built in function (see the [Load/Store Functions](http://pig.apache.org/docs/r0.15.0/func.html#load-store-functions)). PigStorage is the default store function and does not need to be specified (simply omit the USING clause). * You can write your own store function if your data is in a format that cannot be processed by the built in functions (see [User Defined Functions](http://pig.apache.org/docs/r0.15.0/udf.html)). |

#### Usage

Use the STORE operator to run (execute) Pig Latin statements and save (persist) results to the file system. Use STORE for production scripts and batch mode processing.

Note: To debug scripts during development, you can use [DUMP](http://pig.apache.org/docs/r0.15.0/test.html#dump) to check intermediate results.

#### Examples

In this example data is stored using PigStorage and the asterisk character (\*) as the field delimiter.

A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

STORE A INTO 'myoutput' USING PigStorage ('\*');

CAT myoutput;

1\*2\*3

4\*2\*1

8\*3\*4

4\*3\*3

7\*2\*5

8\*4\*3

In this example, the CONCAT function is used to format the data before it is stored.

A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

B = FOREACH A GENERATE CONCAT('a:',(chararray)f1), CONCAT('b:',(chararray)f2), CONCAT('c:',(chararray)f3);

DUMP B;

(a:1,b:2,c:3)

(a:4,b:2,c:1)

(a:8,b:3,c:4)

(a:4,b:3,c:3)

(a:7,b:2,c:5)

(a:8,b:4,c:3)

STORE B INTO 'myoutput' using PigStorage(',');

CAT myoutput;

a:1,b:2,c:3

a:4,b:2,c:1

a:8,b:3,c:4

a:4,b:3,c:3

a:7,b:2,c:5

a:8,b:4,c:3

### STREAM

Sends data to an external script or program.

#### Syntax

|  |
| --- |
| alias = STREAM alias [, alias …] THROUGH {`command` | cmd\_alias } [AS schema] ; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| THROUGH | Keyword. |
| `command` | A command, including the arguments, enclosed in back tics (where a command is anything that can be executed). |
| cmd\_alias | The name of a command created using the DEFINE operator (see [DEFINE (UDFs, streaming)](http://pig.apache.org/docs/r0.15.0/basic.html#define-udfs) for additional streaming examples). |
| AS | Keyword. |
| schema | A schema using the AS keyword, enclosed in parentheses (see [Schemas](http://pig.apache.org/docs/r0.15.0/basic.html#schemas)). |

#### Usage

Use the STREAM operator to send data through an external script or program. Multiple stream operators can appear in the same Pig script. The stream operators can be adjacent to each other or have other operations in between.

When used with a command, a stream statement could look like this:

A = LOAD 'data';

B = STREAM A THROUGH `stream.pl -n 5`;

When used with a cmd\_alias, a stream statement could look like this, where mycmd is the defined alias.

A = LOAD 'data';

DEFINE mycmd `stream.pl –n 5`;

B = STREAM A THROUGH mycmd;

#### About Data Guarantees

Data guarantees are determined based on the position of the streaming operator in the Pig script.

* Unordered data – No guarantee for the order in which the data is delivered to the streaming application.
* Grouped data – The data for the same grouped key is guaranteed to be provided to the streaming application contiguously
* Grouped and ordered data – The data for the same grouped key is guaranteed to be provided to the streaming application contiguously. Additionally, the data within the group is guaranteed to be sorted by the provided secondary key.

In addition to position, data grouping and ordering can be determined by the data itself. However, you need to know the property of the data to be able to take advantage of its structure.

#### Example: Data Guarantees

In this example the data is unordered.

A = LOAD 'data';

B = STREAM A THROUGH `stream.pl`;

In this example the data is grouped.

A = LOAD 'data';

B = GROUP A BY $1;

C = FOREACH B FLATTEN(A);

D = STREAM C THROUGH `stream.pl`;

In this example the data is grouped and ordered.

A = LOAD 'data';

B = GROUP A BY $1;

C = FOREACH B {

D = ORDER A BY ($3, $4);

GENERATE D;

}

E = STREAM C THROUGH `stream.pl`;

#### Example: Schemas

In this example a schema is specified as part of the STREAM statement.

X = STREAM A THROUGH `stream.pl` as (f1:int, f2:int, f3:int);

### UNION

Computes the union of two or more relations.

#### Syntax

|  |
| --- |
| alias = UNION [ONSCHEMA] alias, alias [, alias …]; |

#### Terms

|  |  |
| --- | --- |
| alias | The name of a relation. |
| ONSCHEMA | Use the ONSCHEMA clause to base the union on named fields (rather than positional notation). All inputs to the union must have a non-unknown (non-null) [schema](http://pig.apache.org/docs/r0.15.0/basic.html#schemas). |

#### Usage

Use the UNION operator to merge the contents of two or more relations. The UNION operator:

* Does not preserve the order of tuples. Both the input and output relations are interpreted as unordered bags of tuples.
* Does not ensure (as databases do) that all tuples adhere to the same schema or that they have the same number of fields. In a typical scenario, however, this should be the case; therefore, it is the user's responsibility to either (1) ensure that the tuples in the input relations have the same schema or (2) be able to process varying tuples in the output relation.
* Does not eliminate duplicate tuples.

**Schema Behavior**

The behavior of schemas for UNION (positional notation / data types) and UNION ONSCHEMA (named fields / data types) is the same, except where noted.

Union on relations with two different sizes result in a null schema (union only):

A: (a1:long, a2:long)

B: (b1:long, b2:long, b3:long)

A union B: null

Union columns with incompatible types result in a bytearray type:

A: (a1:long, a2:long)

B: (b1:(b11:long, b12:long), b2:long)

A union B: (a1:bytearray, a2:long)

Union columns of compatible type will produce an "escalate" type. The priority is:

* double > float > long > int > bytearray
* tuple|bag|map|chararray > bytearray

A: (a1:int, a2:bytearray, a3:int)

B: (b1:float, b2:chararray, b3:bytearray)

A union B: (a1:float, a2:chararray, a3:int)

Union of different inner types results in an empty complex type:

A: (a1:(a11:long, a12:int), a2:{(a21:charray, a22:int)})

B: (b1:(b11:int, b12:int), b2:{(b21:int, b22:int)})

A union B: (a1:(), a2:{()})

The alias of the first relation is always taken as the alias of the unioned relation field.

#### Example

In this example the union of relation A and B is computed.

A = LOAD 'data' AS (a1:int,a2:int,a3:int);

DUMP A;

(1,2,3)

(4,2,1)

B = LOAD 'data' AS (b1:int,b2:int);

DUMP A;

(2,4)

(8,9)

(1,3)

X = UNION A, B;

DUMP X;

(1,2,3)

(4,2,1)

(2,4)

(8,9)

(1,3)

#### Example

This example shows the use of ONSCHEMA.

L1 = LOAD 'f1' USING (a : int, b : float);

DUMP L1;

(11,12.0)

(21,22.0)

L2 = LOAD 'f1' USING (a : long, c : chararray);

DUMP L2;

(11,a)

(12,b)

(13,c)

U = UNION ONSCHEMA L1, L2;

DESCRIBE U ;

U : {a : long, b : float, c : chararray}

DUMP U;

(11,12.0,)

(21,22.0,)

(11,,a)

(12,,b)

(13,,c)

## UDF Statements

### DEFINE (UDFs, streaming)

Assigns an alias to a UDF or streaming command.

#### Syntax: UDF and streaming

|  |
| --- |
| DEFINE alias {function | [`command` [input] [output] [ship] [cache] [stderr] ] }; |

#### Terms

|  |  |
| --- | --- |
| alias | The name for a UDF function or the name for a streaming command (the cmd\_alias for the [STREAM](http://pig.apache.org/docs/r0.15.0/basic.html#stream) operator). |
| function | For use with functions.  The name of a UDF function. |
| `command` | For use with streaming.  A command, including the arguments, enclosed in back tics (where a command is anything that can be executed).  The clauses (input, output, ship, cache, stderr) are described below. Note the following:   * All clauses are optional. * The clauses can be specified in any order (for example, stderr can appear before input) * Each clause can be specified at most once (for example, multiple inputs are not allowed) |
| input | For use with streaming.  INPUT ( {stdin | 'path'} [USING serializer] [, {stdin | 'path'} [USING serializer] …] )  Where:   * INPUT – Keyword. * 'path' – A file path, enclosed in single quotes. * USING – Keyword. * serializer – PigStreaming is the default serializer. |
| output | For use with streaming.  OUTPUT ( {stdout | stderr | 'path'} [USING deserializer] [, {stdout | stderr | 'path'} [USING deserializer] …] )  Where:   * OUTPUT – Keyword. * 'path' – A file path, enclosed in single quotes. * USING – Keyword. * deserializer – PigStreaming is the default deserializer. |
| ship | For use with streaming.  SHIP('path' [, 'path' …])  Where:   * SHIP – Keyword. * 'path' – A file path, enclosed in single quotes. |
| cache | For use with streaming.  CACHE('dfs\_path#dfs\_file' [, 'dfs\_path#dfs\_file' …])  Where:   * CACHE – Keyword. * 'dfs\_path#dfs\_file' – A file path/file name on the distributed file system, enclosed in single quotes. Example: '/mydir/mydata.txt#mydata.txt' |
| stderr | For use with streaming.  STDERR( '/dir') or STDERR( '/dir' LIMIT n)  Where:   * '/dir' is the log directory, enclosed in single quotes. * (optional) LIMIT n is the error threshold where n is an integer value. If not specified, the default error threshold is unlimited. |

#### Usage

Use the DEFINE statement to assign a name (alias) to a UDF function or to a streaming command.

Use DEFINE to specify a UDF function when:

* The function has a long package name that you don't want to include in a script, especially if you call the function several times in that script.
* The constructor for the function takes string parameters. If you need to use different constructor parameters for different calls to the function you will need to create multiple defines – one for each parameter set.

Use DEFINE to specify a streaming command when:

* The streaming command specification is complex.
* The streaming command specification requires additional parameters (input, output, and so on).

##### About Input and Output

Serialization is needed to convert data from tuples to a format that can be processed by the streaming application. Deserialization is needed to convert the output from the streaming application back into tuples. PigStreaming is the default serialization/deserialization function.

Streaming uses the same default format as PigStorage to serialize/deserialize the data. If you want to explicitly specify a format, you can do it as show below (see more examples in the Examples: Input/Output section).

DEFINE CMD `perl PigStreaming.pl - nameMap` input(stdin using PigStreaming(',')) output(stdout using PigStreaming(','));

A = LOAD 'file';

B = STREAM B THROUGH CMD;

If you need an alternative format, you will need to create a custom serializer/deserializer by implementing the following interfaces.

interface PigToStream {

/\*\*

\* Given a tuple, produce an array of bytes to be passed to the streaming

\* executable.

\*/

public byte[] serialize(Tuple t) throws IOException;

}

interface StreamToPig {

/\*\*

\* Given a byte array from a streaming executable, produce a tuple.

\*/

public Tuple deserialize(byte[]) throws IOException;

/\*\*

\* This will be called on the front end during planning and not on the back

\* end during execution.

\*

\* @return the {@link LoadCaster} associated with this object.

\* @throws IOException if there is an exception during LoadCaster

\*/

public LoadCaster getLoadCaster() throws IOException;

}

##### About Ship

Use the ship option to send streaming binary and supporting files, if any, from the client node to the compute nodes. Pig does not automatically ship dependencies; it is your responsibility to explicitly specify all the dependencies and to make sure that the software the processing relies on (for instance, perl or python) is installed on the cluster. Supporting files are shipped to the task's current working directory and only relative paths should be specified. Any pre-installed binaries should be specified in the PATH.

Only files, not directories, can be specified with the ship option. One way to work around this limitation is to tar all the dependencies into a tar file that accurately reflects the structure needed on the compute nodes, then have a wrapper for your script that un-tars the dependencies prior to execution.

Note that the ship option has two components: the source specification, provided in the ship( ) clause, is the view of your machine; the command specification is the view of the actual cluster. The only guarantee is that the shipped files are available in the current working directory of the launched job and that your current working directory is also on the PATH environment variable.

Shipping files to relative paths or absolute paths is not supported since you might not have permission to read/write/execute from arbitrary paths on the clusters.

Note the following:

* It is safe only to ship files to be executed from the current working directory on the task on the cluster.
* OP = stream IP through 'script';
* or
* DEFINE CMD 'script' ship('/a/b/script');
* OP = stream IP through CMD;
* Shipping files to relative paths or absolute paths is undefined and mostly will fail since you may not have permissions to read/write/execute from arbitraty paths on the actual clusters.

##### About Cache

The ship option works with binaries, jars, and small datasets. However, loading larger datasets at run time for every execution can severely impact performance. Instead, use the cache option to access large files already moved to and available on the compute nodes. Only files, not directories, can be specified with the cache option.

##### About Auto-Ship

If the ship and cache options are not specified, Pig will attempt to auto-ship the binary in the following way:

* If the first word on the streaming command is perl or python, Pig assumes that the binary is the first non-quoted string it encounters that does not start with dash.
* Otherwise, Pig will attempt to ship the first string from the command line as long as it does not come from /bin, /usr/bin, /usr/local/bin. Pig will determine this by scanning the path if an absolute path is provided or by executing which. The paths can be made configurable using the [set stream.skippath](http://pig.apache.org/docs/r0.15.0/cmds.html#set) option (you can use multiple set commands to specify more than one path to skip).

If you don't supply a DEFINE for a given streaming command, then auto-shipping is turned off.

Note the following:

* If Pig determines that it needs to auto-ship an absolute path it will not ship it at all since there is no way to ship files to the necessary location (lack of permissions and so on).
* OP = stream IP through `/a/b/c/script`;
* or
* OP = stream IP through `perl /a/b/c/script.pl`;
* Pig will not auto-ship files in the following system directories (this is determined by executing 'which <file>' command).
* /bin /usr/bin /usr/local/bin /sbin /usr/sbin /usr/local/sbin
* To auto-ship, the file in question should be present in the PATH. So if the file is in the current working directory then the current working directory should be in the PATH.

#### Examples: Input/Output

In this example PigStreaming is the default serialization/deserialization function. The tuples from relation A are converted to tab-delimited lines that are passed to the script.

X = STREAM A THROUGH `stream.pl`;

In this example PigStreaming is used as the serialization/deserialization function, but a comma is used as the delimiter.

DEFINE Y 'stream.pl' INPUT(stdin USING PigStreaming(',')) OUTPUT (stdout USING PigStreaming(','));

X = STREAM A THROUGH Y;

In this example user defined serialization/deserialization functions are used with the script.

DEFINE Y 'stream.pl' INPUT(stdin USING MySerializer) OUTPUT (stdout USING MyDeserializer);

X = STREAM A THROUGH Y;

#### Examples: Ship/Cache

In this example ship is used to send the script to the cluster compute nodes.

DEFINE Y 'stream.pl' SHIP('/work/stream.pl');

X = STREAM A THROUGH Y;

In this example cache is used to specify a file located on the cluster compute nodes.

DEFINE Y 'stream.pl data.gz' SHIP('/work/stream.pl') CACHE('/input/data.gz#data.gz');

X = STREAM A THROUGH Y;

#### Example: DEFINE with STREAM

In this example a command is defined for use with the [STREAM](http://pig.apache.org/docs/r0.15.0/basic.html#stream) operator.

A = LOAD 'data';

DEFINE mycmd 'stream\_cmd –input file.dat';

B = STREAM A through mycmd;

#### Examples: Logging

In this example the streaming stderr is stored in the \_logs/<dir> directory of the job's output directory. Because the job can have multiple streaming applications associated with it, you need to ensure that different directory names are used to avoid conflicts. Pig stores up to 100 tasks per streaming job.

DEFINE Y 'stream.pl' stderr('<dir>' limit 100);

X = STREAM A THROUGH Y;

#### Examples: DEFINE a function

In this example a function is defined for use with the FOREACH …GENERATE operator.

REGISTER /src/myfunc.jar

DEFINE myFunc myfunc.MyEvalfunc('foo');

A = LOAD 'students';

B = FOREACH A GENERATE myFunc($0);

### REGISTER

Registers a JAR file so that the UDFs in the file can be used.

#### Syntax

|  |
| --- |
| REGISTER path; |

#### Terms

|  |  |
| --- | --- |
| path | The path to the JAR file (the full location URI is required). Do not place the name in quotes. |

#### Usage

**Pig Scripts**

Use the REGISTER statement inside a Pig script to specify a JAR file or a Python/JavaScript module. Pig supports JAR files and modules stored in local file systems as well as remote, distributed file systems such as HDFS and Amazon S3 (see [Pig Scripts](http://pig.apache.org/docs/r0.15.0/start.html#pig-scripts)).

Additionally, JAR files stored in local file systems can be specified as a glob pattern using “\*”. Pig will search for matching jars in the local file system, either the relative path (relative to your working directory) or the absolute path. Pig will pick up all JARs that match the glob.

**Command Line**

You can register additional files (to use with your Pig script) via PIG\_OPTS environment variable using the -Dpig.additional.jars.uris option. For more information see [User Defined Functions](http://pig.apache.org/docs/r0.15.0/udf.html).

#### Examples

In this example REGISTER states that the JavaScript module, myfunc.js, is located in the /src directory.

/src $ java -jar pig.jar –

REGISTER /src/myfunc.js;

A = LOAD 'students';

B = FOREACH A GENERATE myfunc.MyEvalFunc($0);

In this example additional JAR files are registered via PIG\_OPTS environment variable.

export PIG\_OPTS="-Dpig.additional.jars.uris=my.jar,your.jar"

In this example a JAR file stored in HDFS and a local JAR file are registered.

export PIG\_OPTS="-Dpig.additional.jars.uris=hdfs://nn.mydomain.com:9020/myjars/my.jar,file:///home/root/pig/your.jar"

Note, the legacy property pig.additional.jars which use colon as separator is still supported. But we recommend to use pig.additional.jars.uris since colon is also used in URL scheme, and thus we cannot use full scheme in the list. We will deprecate pig.additional.jar in future releases.

This example shows how to specify a glob pattern using either a relative path or an absolute path.

register /homes/user/pig/myfunc\*.jar

register count\*.jar

register jars/\*.jar