# Unit 5:

Pig: Hadoop Programming Made Easier Admiring the Pig Architecture, Going with the Pig Latin Application Flow, Working through the ABCs of Pig Latin, Evaluating Local and Distributed Modes of Running Pig Scripts, Checking out the Pig Script Interfaces, Scripting with Pig Latin

## Pig: Hadoop Programming Made Easier Admiring the Pig Architecture,

Apache Pig is an abstraction over MapReduce. It is a tool/platform which is used to analyze larger sets of data representing them as data flows. Pig is generally used with **Hadoop**; we can perform all the data manipulation operations in Hadoop using Pig.

To write data analysis programs, Pig provides a high-level language known as**Pig Latin**. This language provides various operators using which programmers can develop their own functions for reading, writing, and processing data.

To analyze data using **Apache Pig**, programmers need to write scripts using Pig Latin language. All these scripts are internally converted to Map and Reduce tasks. Apache Pig has a component known as **Pig Engine** that accepts the Pig Latin scripts as input and converts those scripts into MapReduce jobs.

# Why Do WeNeed Apache Pig?

Programmers who are not so good at Java normally used to struggle working with Hadoop, especially while performing any MapReduce tasks. Apache Pig is a boon for all such programmers.

- Using **Pig Latin**, programmers can perform MapReduce tasks easily without having to type complex codes in Java.
- Apache Pig uses **multi-query approach**, thereby reducing the length of codes. For example, an operation that would require you to type 200 lines of code (LoC) in Java can be easily done by typing as less as just 10 LoC in Apache Pig. Ultimately Apache Pig reduces the development time by almost 16 times.
- Pig Latin is **SQL-like language** and it is easy to learn Apache Pig when you are familiar with SQL.

- Apache Pig provides many built-in operators to support data operations like joins, filters, ordering, etc. In addition, it also provides nested data types like tuples, bags, and maps that are missing from MapReduce.

# Features of Pig

Apache Pig comes with the following features –

- **Rich set of operators** – It provides many operators to perform operations like join, sort, filer, etc.
- **Ease of programming** – Pig Latin is similar to SQL and it is easy to write a Pig script if you are good at SQL.
- **Optimization opportunities** – The tasks in Apache Pig optimize their execution automatically, so the programmers need to focus only on semantics of the language.
- **Extensibility** – Using the existing operators, users can develop their own functions to read, process, and write data.
- **UDF's** – Pig provides the facility to create **User-defined Functions** in other programming languages such as Java and invoke or embed them in Pig Scripts.
- **Handles all kinds of data** – Apache Pig analyzes all kinds of data, both structured as well as unstructured. It stores the results in HDFS.

# Apache Pig Vs MapReduce

Listed below are the major differences between Apache Pig and MapReduce.

| Apache Pig | MapReduce |
|---|---|
| Apache Pig is a data flow language. | MapReduce is a data processing paradigm. |
| It is a high level language. | MapReduce is low level and rigid. |
| Performing a Join operation in Apache Pig is | It is quite difficult in MapReduce to perform a Join operation between |

| | |
|---|---|
| pretty simple. | datasets. |
| Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig. | Exposure to Java is must to work with MapReduce. |
| Apache Pig uses multi-query approach, thereby reducing the length of the codes to a great extent. | MapReduce will require almost 20 times more the number of lines to perform the same task. |
| There is no need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job. | MapReduce jobs have a long compilation process. |

# Apache Pig Vs SQL

Listed below are the major differences between Apache Pig and SQL.

| Pig | SQL |
|---|---|
| Pig Latin is a **procedural** language. | SQL is a **declarative** language. |
| In Apache Pig, **schema** is optional. We can store data without designing a schema (values are stored as $01, $02 etc.) | Schema is mandatory in SQL. |
| The data model in Apache Pig is **nested relational**. | The data model used in SQL **is flat relational**. |
| Apache Pig provides limited opportunity for **Query optimization**. | There is more opportunity for query optimization in SQL. |

In addition to above differences, Apache Pig Latin –

- Allows splits in the pipeline.
- Allows developers to store data anywhere in the pipeline.

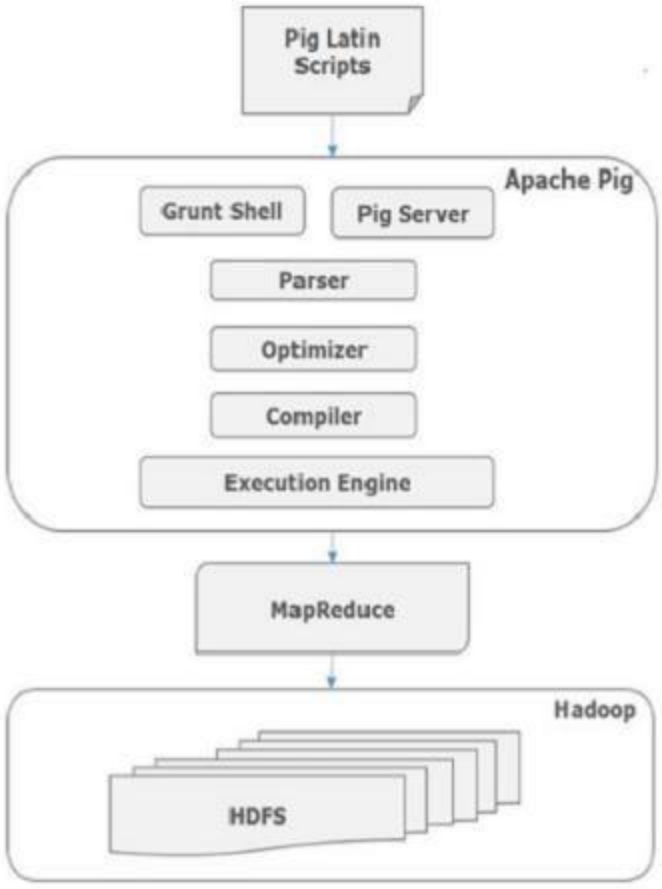
- Declares execution plans.
- Provides operators to perform ETL (Extract, Transform, and Load) functions.

## Apache Pig - Architecture

The language used to analyze data in Hadoop using Pig is known as **Pig Latin**. It is a highlevel data processing language which provides a rich set of data types and operators to perform various operations on the data.

To perform a particular task Programmers using Pig, programmers need to write a Pig script using the Pig Latin language, and execute them using any of the execution mechanisms (Grunt Shell, UDFs, Embedded). After execution, these scripts will go through a series of transformations applied by the Pig Framework, to produce the desired output.

Internally, Apache Pig converts these scripts into a series of MapReduce jobs, and thus, it makes the programmer's job easy. The architecture of Apache Pig is shown below.



# Apache Pig Components

As shown in the figure, there are various components in the Apache Pig framework. Let us take a look at the major components.

## Parser

Initially the Pig Scripts are handled by the Parser. It checks the syntax of the script, does type checking, and other miscellaneous checks. The output of the parser will be a DAG (directed acyclic graph), which represents the Pig Latin statements and logical operators.

In the DAG, the logical operators of the script are represented as the nodes and the data flows are represented as edges.

## Optimizer

The logical plan (DAG) is passed to the logical optimizer, which carries out the logical optimizations such as projection and pushdown.

## Compiler

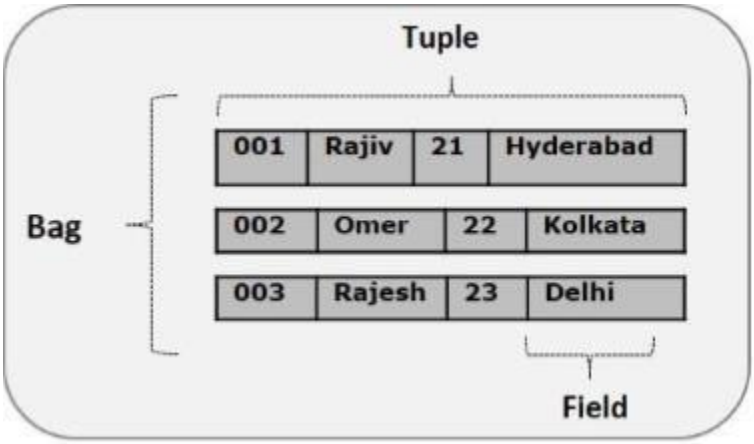
The compiler compiles the optimized logical plan into a series of MapReduce jobs.

## Execution engine

Finally the MapReduce jobs are submitted to Hadoop in a sorted order. Finally, these MapReduce jobs are executed on Hadoop producing the desired results.

# Pig Latin Data Model

The data model of Pig Latin is fully nested and it allows complex non-atomic datatypes such as **map** and **tuple**. Given below is the diagrammatical representation of Pig Latin's data model.




## Atom

Any single value in Pig Latin, irrespective of their data, type is known as an**Atom**. It is stored as string and can be used as string and number. int, long, float, double, chararray, and bytearray are the atomic values of Pig. A piece of data or a simple atomic value is known as a **field**.

**Example** – 'raja' or '30'

## Tuple

A record that is formed by an ordered set of fields is known as a tuple, the fields can be of any type. A tuple is similar to a row in a table of RDBMS.

**Example** – (Raja, 30)

## Bag

A bag is an unordered set of tuples. In other words, a collection of tuples (non-unique) is known as a bag. Each tuple can have any number of fields (flexible schema). A bag is represented by '{}'. It is similar to a table in RDBMS, but unlike a table in RDBMS, it is not necessary that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

**Example** – {(Raja, 30), (Mohammad, 45)}

A bag can be a field in a relation; in that context, it is known as **inner bag**.

**Example** – {Raja, 30, **{9848022338, raja@gmail.com,}**}

### Map

A map (or data map) is a set of key-value pairs. The **key** needs to be of type chararray and should be unique. The **value** might be of any type. It is represented by '[]'

**Example** – [name#Raja, age#30]

### Relation

A relation is a bag of tuples. The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).

## Apache Pig Execution Modes

You can run Apache Pig in two modes, namely, **Local Mode** and **HDFS mode**.

### Local Mode

In this mode, all the files are installed and run from your local host and local file system. There is no need of Hadoop or HDFS. This mode is generally used for testing purpose.

### MapReduce Mode

MapReduce mode is where we load or process the data that exists in the Hadoop File System (HDFS) using Apache Pig. In this mode, whenever we execute the Pig Latin statements to process the data, a MapReduce job is invoked in the back-end to perform a particular operation on the data that exists in the HDFS.

## Apache Pig Execution Mechanisms

Apache Pig scripts can be executed in three ways, namely, interactive mode, batch mode, and embedded mode.

- **Interactive Mode** (Grunt shell) – You can run Apache Pig in interactive mode using the Grunt shell. In this shell, you can enter the Pig Latin statements and get the output (using Dump operator).
- **Batch Mode** (Script) – You can run Apache Pig in Batch mode by writing the Pig Latin script in a single file with **.pig** extension.

- **Embedded Mode** (UDF) – Apache Pig provides the provision of defining our own functions (**U**ser **D**efined **F**unctions) in programming languages such as Java, and using them in our script.

# Invoking the Grunt Shell

You can invoke the Grunt shell in a desired mode (local/MapReduce) using the **–x** option as shown below.

| Local mode | MapReduce mode |
|---|---|
| **Command –**<br><br>$ ./pig –x local | **Command –**<br><br>$ ./pig -x mapreduce |
| **Output** –<br><br>15/09/28 10:13:03 INFO pig.Main: Logging error messages to: /home/Hadoop/pig_1443415383991.log<br>2015-09-28 10:13:04,838 [main] INFO org.apache.pig.backend.hadoop.execution engine.HExecutionEngine - Connecting to hadoop file system at: file:///<br><br>**grunt>** | **Output** –<br><br>15/09/28 10:28:46 INFO pig.Main: Logging error messages to: /home/Hadoop/pig_1443416326123.log<br>2015-09-28 10:28:46,427 [main] INFO org.apache.pig.backend.hadoop.execution engine.HExecutionEngine - Connecting to hadoop file system at: file:///<br><br>**grunt>** |

Either of these commands gives you the Grunt shell prompt as shown below.

```
grunt>
```

You can exit the Grunt shell using **'ctrl + d'.**

After invoking the Grunt shell, you can execute a Pig script by directly entering the Pig Latin statements in it.

```
grunt> customers = LOAD 'customers.txt' USING PigStorage(',');
```

# Executing Apache Pig in Batch Mode

You can write an entire Pig Latin script in a file and execute it using the **–x command**. Let us suppose we have a Pig script in a file named**sample_script.pig** as shown below.

## Sample_script.pig

```
student = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING
   PigStorage(',') as (id:int,name:chararray,city:chararray);


Dump student;
```

Now, you can execute the script in the above file as shown below.

| Local mode | MapReduce mode |
|---|---|
| $ pig -x local **Sample_script.pig** | $ pig -x mapreduce **Sample_script.pig** |

**Note** – We will discuss in detail how to run a Pig script in **Bach mode** and in**embedded mode** in subsequent chapters.

# Conventions

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

| Convention | Description |
|---|---|
| ( ) | Parentheses enclose one or more items.<br><br>Parentheses are also used to indicate the tuple data type. |
| [ ] | Straight brackets enclose one or more optional items.<br><br>Straight brackets are also used to indicate the map data type. In this case <> is used to indicate optional items. |
| { } | 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.

| | |
|---|---|
| -- O | onschema, or, order, outer, output |
| -- P | parallel, pig, PigDump, PigStorage, pwd |
| -- Q | quit |
| -- R | register, right, rm, rmf, 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 | |

# 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) and all other Pig Latin keywords (see 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 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)
```

### 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 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 | name | age | gpa |

| | | | |
|---|---|---|---|
| by you using a schema) | | | |
| 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 Data Types** | Description | Example |
|---|---|---|
| Scalars | | |
| int | Signed 32-bit integer | 10 |
| long | Signed 64-bit integer | Data: 10L or 10l<br>Display: 10L |
| float | 32-bit floating point | Data: 10.5F or 10.5f or 10.5e2f or 10.5E2F<br>Display: 10.5F or 1050.0F |
| double | 64-bit floating point | Data: 10.5 or 10.5e2 or 10.5E2<br>Display: 10.5 or 1050.0 |
| Arrays | | |
| chararray | Character array (string) in Unicode UTF-8 format | hello world |
| bytearray | Byte array (blob) | |
| **Complex Data 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.

### 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.

#### 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.

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.

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:<br>==, !=<br>>, <<br>>=, <= | If either subexpression is null, the result is null. |
| Comparison operator:<br>matches | If either the string being matched against or the string defining the match is null, t |
| Arithmetic operators:<br>+ , -, *, /<br>% modulo<br>? : bincond | If either subexpression is null, the resulting expression is null. |

| | |
|---|---|
| Null operator:<br><br>is null | If the tested value is null, returns true; otherwise, returns false (see Null Operators |
| Null operator:<br><br>is not null | If the tested value is not null, returns true; otherwise, returns false (see Null Opera |
| Dereference operators:<br><br>tuple (.) or map (#) | If the de-referenced tuple or map is null, returns null. |
| Operators:<br><br>COGROUP, GROUP, JOIN | These operators handle nulls differently (see examples below). |
| Function:<br><br>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:<br><br>AVG, MIN, MAX, SUM, COUNT | These functions ignore nulls. |
| Function:<br><br>CONCAT | If either subexpression is null, the resulting expression is null. |
| Function:<br><br>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 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.)

```
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 | | |
| Scalars | | |
| int | 19 | |
| long | 19L | |
| float | 19.2F or 1.92e2f | |
| double | 19.2 or 1.92e2 | |
| Arrays | | |
| chararray | 'hello world' | |
| bytearray | | Not applicab |
| Complex Data Types | | |
| tuple | (19, 2, 1) | A constant i |
| bag | { (19, 2), (1, 2) } | A constant i |
| map | [ 'name' # 'John', 'ext' # 5555 ] | A constant i |

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, Unicode, and 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).

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 scalar; 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 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 ( * ) is allowed.

Project-range can be used in the following statements: FOREACH, JOIN, GROUP, COGROUP, and 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).

### 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 and complex types can be used anywhere a schema definition is appropriate.

Schemas are defined with the LOAD, STREAM, and 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 and User Defined Functions 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. 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 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 relationship 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, and bytearray.

Syntax

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

Terms

| | |
|---|---|
| alias | The name assigned to the field. |
| type | (Optional) The simple data type assigned to the field.<br>The alias and type are separated by a colon ( : ).<br>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<br><br>alias – the name assigned to the field<br>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).<br><br>The type applies to the map value only; the map key is always type chararray (see Map).<br><br>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:[]
);
```

# 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).<br><br>Works with integral numbers (int, long). |
| bincond | ? : | (condition ? value_if_true : value_if_false)<br><br>The bincond should be enclosed in parenthesis.<br><br>The schemas for the two conditional outputs of the bincond should match.<br><br>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)
```

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

* bytearray cast as this data type

| | bag | tuple | map | int | long | float | double |
|---|---|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| map | error | error | error | error | error |

## Cast Operators

### Description

Pig Latin supports casts as shown in this table.

| | to |
|---|---|

- 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 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;
{(48290904939805222OOL)}
{(48932985698628374 93L)}
{(12977893028973987 83L)}

A = LOAD 'data' AS fld:bytearray;

DESCRIBE A;
A: {fld: bytearray}

DUMP A;
({(48290904939805222OOL)})
({(48932985698628374 93L)})
({(12977893028973987 83L)})

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

DESCRIBE B;
B: {{(long)}}

DUMP B;
({(48290904939805222OOL)})
({(48932985698628374 93L)})
({(12977893028973987 83L)})
```

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 if 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.<br>*expression* matches *string-constant*<br>Use the Java format for regular expressions. |

Use the comparison operators with numeric and string data.

Example: numeric

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

Example: string

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

Example: matches

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

| | |
|---|---|
| | chararray |
| chararray | boolean |
| bytearray | boolean |

## Dereference Operators

### Description

| Operator | Symbol | Notes |
|---|---|---|
| tuple dereference | tuple.id or tuple.(id,…) | Tuple dereferencing can be done by name (tuple.field_name) or positio<br>If a set of fields are dereferenced (tuple.(name1, name2) or tuple.($0, $<br>the expression represents a tuple composed of the specified fields.<br>Note that if the dot operator is applied to a bytearray, the bytearraywil |
| bag dereference | bag.id or bag.(id,…) | Bag dereferencing can be done by name (bag.field_name) or position (<br>If a set of fields are dereferenced (bag.(name1, name2) or bag.($0, $1))<br>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).<br>If the pound operator is applied to a bytearray, |

| | | |
|---|---|---|
| | | the bytearray is assumed to be a map.<br><br>If the key does not exist, the empty string is returned. |

Example: Tuple

Suppose we have relation 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)
```

Example: Bag

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)})
```

Example: Tuple and Bag

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)})
```

Example: Map

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.)

```
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.

## Null Operators

| Description | |
|---|---|
| Operator | Symbol |

| | |
|---|---|
| float | float |
| double | double |
| chararray | error |
| bytearray | double (as double) |

# Relational Operators

## COGROUP

See the 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<br><br>For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoop/<br>For usage, see Example: PARTITION BY |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.<br><br>For more information, see Use the Parallel Features. |

### 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)
```

## DEFINE

See:

- DEFINE (UDFs, streaming)
- 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<br><br>For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoop/<br>For usage, see Example: PARTITION BY. |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.<br><br>For more information, see Use the Parallel Features. |

### 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 (seeExample: Nested Block).

### 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:<br>alias = FOREACH alias GENERATE expression [AS schema] [expression [AS schema]….];<br>See Schemas |
| nested_block | Nested FOREACH...GENERATE block used with a inner bag. Use this syntax:<br>alias = FOREACH nested_alias {<br>alias = {nested_op \| nested_exp}; [{alias = {nested_op \| nested_exp}; …]<br>GENERATE expression [AS schema] [expression [AS schema]….]<br>};<br>Where:<br>The nested block is enclosed in opening and closing brackets { … }. |

| | |
|---|---|
| | The GENERATE keyword must be the last statement within the nested block.<br><br>See Schemas<br><br>Macros are NOT alllowed inside a nested block. |
| expression | An expression. |
| nested_alias | The name of the inner bag. |
| nested_op | Allowed operations are DISTINCT, FILTER, LIMIT, and ORDER BY.<br><br>The FOREACH…GENERATE operation itself is not allowed since this could lead to an arbitrary number of nesting lev<br><br>You can also perform projections (see Example: Nested Block). |
| nested_exp | Any arbitrary, supported expression. |
| AS | Keyword |
| schema | A schema using the AS keyword (see Schemas).<br><br>If the FLATTEN operator is used, enclose the schema in parentheses.<br>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 tuples 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 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**

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] [P |

### Terms

| | |
|---|---|
| alias | The name of a relation.<br><br>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 r |

| | |
|---|---|
| | B = GROUP A ALL; |
| BY | Keyword. Use this clause to group the relation by field, tuple or expression.<br><br>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 wil multiple keys, enclose the keys in parentheses:<br><br>B = GROUP A BY (key1,key2); |
| USING | Keyword |
| 'collected' | Use the 'collected' clause with the GROUP operation (works with one relation only).<br><br>The following conditions apply:<br><br>The loader must implement the {CollectableLoader} interface.<br>Data must be sorted on the group key.<br><br>If your data and loaders satisfy these conditions, use the 'collected' clause to perform an optimized version of GROU<br><br>Note that the Zebra loader satisfies the conditions (see Zebra and Pig). |
| 'merge' | Use the 'merge' clause with the COGROUP operation (works with two or more relations only).<br><br>The following conditions apply:<br><br>No other operations can be done between the LOAD and COGROUP statements.<br>Data must be sorted on the COGROUP key for all tables in ascending (ASC)order.<br>Nulls are considered smaller than evertyhing. If data contains null keys, they should occur before anything e<br>Left-most loader must implement the {CollectableLoader} interface as well as {OrderedLoadFunc} interfac<br>All other loaders must implement IndexableLoadFunc.<br>Type information must be provided in the schema for all the loaders.<br><br>If your data and loaders satisfy these conditions, the 'merge' clause to perform an optimized version of COGROUP; t<br><br>Note that the Zebra loader satisfies the conditions (see Zebra and Pig). |
| PARTITION BY partitioner | Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermed<br><br>For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoop/mapred/Partitio<br>For usage, see Example: PARTITION BY |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.<br><br>For more information, see Use the Parallel Features. |

### 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).

**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 two fields, "group" and "A" (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: {firend1:
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,Ali
ce)})
(Bob,{(Bob,dog),(Bob,cat)},{(Paul,Bob)})
(Jane,{},{(Paul,Jane)})
```

In this example tuples are co-grouped and the INNER keyword is used asymmetrically on only one of the relations.

```
X = COGROUP A BY owner, B BY friend2 INNER;

DUMP X;
(Bob,{(Bob,dog),(Bob,cat)},{(Paul,Bob)})
(Jane,{},{(Paul,Jane)})
(Alice,{(Alice,turtle),(Alice,goldfish),(Alice,cat)},{(Cindy,Alice),(Mark,Ali
ce)})
```

**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**

This example shows how to group using the collected keyword.

```
 register zebra.jar;
 A = LOAD 'studentsortedtab' USING
org.apache.hadoop.zebra.pig.TableLoader('name, age, gpa', 'sorted');
 B = GROUP A BY name USING 'collected';
 C = FOREACH b GENERATE group, MAX(a.age), COUNT STAR(a);
```

**Example**

This example shows how to use COGROUP with the merge keyword.

```
 register zebra.jar;
 A = LOAD 'data1' USING org.apahce.hadoop.zebra.pig.TableLoader('id:int',
'sorted');
 B = LOAD 'data2' USING org.apahce.hadoop.zebra.pig.TableLoader('id:int',
'sorted');'
 C = COGROUP A BY id, B BY id USING 'merge';
```

**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)

## 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 'rep |

### Terms

| | |
|---|---|
| alias | The name of a relation. |
| BY | Keyword |
| expression | A field expression.<br><br>Example: X = JOIN A BY fieldA, B BY fieldB, C BY fieldC; |
| USING | Keyword |
| 'replicated' | Use to perform replicated joins (see Replicated Joins). |
| 'skewed' | Use to perform skewed joins (see Skewed Joins). |
| 'merge' | Use to perform merge joins (see Merge Joins). |
| PARTITION BY partitioner | Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys<br><br>For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoop/<br>For usage, see Example: PARTITION BY<br><br>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. |

### Usage

Use the JOIN operator to perform an inner, equijoin join of two or more relations based on common field values. The JOIN operator always performs an inner join. 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).

### 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 or more 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 |

### 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-alia |
| 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).<br><br>Only left outer join is supported for replicated joins. |
| 'skewed' | Use to perform skewed joins (see Skewed Joins). |
| 'merge' | Use to perform merge joins (see Merge Joins). |
| PARTITION BY partitioner | Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys<br><br>For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoop/<br>For usage, see Example: PARTITION BY<br><br>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. |

### 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 (a constant). |

### 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

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.<br><br>If you specify a directory name, all the files in the directory are loaded.<br><br>You can use Hadoop globing to specify files at the file system or directory levels (see Hadoop globStatus for details on globing synta<br><br>**Note:** Pig uses Hadoop globbing so the functionality is IDENTICAL. However, when you run from the command line using the Hado of the substitutions; this could alter the outcome giving the impression that globing works differently for Pig and Hadoop. For examp<br><br>This works<br>hadoop fs -ls /mydata/20110423{00,01,02,03,04,05,06,07,08,09,{10..23}}00//**part**<br><br>This does not work<br>LOAD '/mydata/20110423{00,01,02,03,04,05,06,07,08,09,{10..23}}00//**part** ' |
| USING | Keyword.<br><br>If the USING clause is omitted, the default load function PigStorage is used. |
| function | The load function.<br><br>You can use a built in function (see Load/Store Functions). PigStorage is the default load function and does not need to be s<br>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 |
| AS | Keyword. |
| schema | A schema using the AS keyword, enclosed in parentheses (see Schemas).<br><br>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<br><br>Note: For performance reasons the loader may not immediately convert the data to the specified format; however, you can still operat |

## 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 and Schemas for Multiple Types.

## 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 |

### Terms

| | |
|---|---|
| alias1, alias2 | The names of relations. |
| mr.jar | The MapReduce jar file (enclosed in single quotes).<br>You can specify any MapReduce jar file that can be run through the `hadoop jar mymr.jar`<br>The values for inputLocation and outputLocation can be passed in the params. |

| | |
|---|---|
| STORE ... INTO ... USING | See STORE<br><br>Store alias2 into the inputLocation using storeFunc, which is then used by the MapReduce job to rea |
| LOAD ... USING ... AS | See LOAD<br><br>After running mr1.jar's MapReduce job, load back the data from outputLocation into alias1 using lo |
| `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. |

### 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):

- 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)
```

## SAMPLE

Partitions a relation into two or more relations.

### Syntax

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

### Terms

| | |
|---|---|
| alias | The name of a relation. |
| size | Sample size, range 0 to 1 (for example, enter 0.1 for 10%). |

### 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;
```

## SPLIT

Partitions a relation into two or more relations.

### Syntax

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

### Terms

| | |
|---|---|
| alias | The name of a relation. |
| INTO | Required keyword. |
| IF | Required keyword. |
| expression | An expression. |

### 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.<br><br>The output data files, named part-nnnnn, are written to this directory. |
| USING | Keyword. Use this clause to name the store function.<br><br>If the USING clause is omitted, the default store function PigStorage is used. |

| | |
|---|---|
| function | The store function.<br><br>You can use a built in function (see the Load/Store Functions). PigStorage is the default store function and does not nee<br>You can write your own store function if your data is in a format that cannot be processed by the built in functions (see |

### 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 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) for additional streaming ex |
| AS | Keyword. |
| schema | A schema using the AS keyword, enclosed in parentheses (see 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 mus |

### 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 operator). |
| function | For use with functions.<br><br>The name of a UDF function. |
| `command` | For use with streaming.<br><br>A command, including the arguments, enclosed in back tics (where a command is anything that can be executed).<br><br>The clauses (input, output, ship, cache, stderr) are described below. Note the following:<br><br>All clauses are optional.<br><br>The clauses can be specified in any order (for example, stderr can appear before input)<br><br>Each clause can be specified at most once (for example, multiple inputs are not allowed) |
| input | For use with streaming.<br><br>INPUT ( {stdin \| 'path'} [USING serializer] [, {stdin \| 'path'} [USING serializer] …] )<br><br>Where:<br><br>INPUT – Keyword.<br>'path' – A file path, enclosed in single quotes.<br>USING – Keyword.<br>serializer – PigStreaming is the default serializer. |
| output | For use with streaming.<br><br>OUTPUT ( {stdout \| stderr \| 'path'} [USING deserializer] [, {stdout \| stderr \| 'path'} [USING deserializer] …] )<br><br>Where:<br><br>OUTPUT – Keyword.<br>'path' – A file path, enclosed in single quotes.<br>USING – Keyword.<br>deserializer – PigStreaming is the default deserializer. |
| ship | For use with streaming.<br><br>SHIP('path' [, 'path' …])<br><br>Where:<br><br>SHIP – Keyword.<br>'path' – A file path, enclosed in single quotes. |
| cache | For use with streaming.<br><br>CACHE('dfs_path#dfs_file' [, 'dfs_path#dfs_file' …])<br><br>Where:<br><br>CACHE – Keyword. |

| | |
|---|---|
| | 'dfs_path#dfs_file' – A file path/file name on the distributed file system, enclosed in single quotes.<br>Example: '/mydir/mydata.txt#mydata.txt' |
| stderr | For use with streaming.<br>STDERR( '/dir') or STDERR( '/dir' LIMIT n)<br>Where:<br>'/dir' is the log directory, enclosed in single quotes.<br>(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 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 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;
```

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).

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 the command line using the -Dpig.additional.jars option. For more information see User Defined Functions.

## 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 the command line.

```
pig -Dpig.additional.jars=my.jar:your.jar script.pig
```

In this example a JAR file stored in HDFS is registered.

```
java -cp pig.jar org.apache.pig.Main
hdfs://nn.mydomain.com:9020/myscripts/script.pig
```

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
```