Pig Latin Manual

by

Table of contents

1. Conventions

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

Convention	Description	Example
( )	Parentheses enclose one or more items. Parentheses are also used to indicate the tuple data type.	Multiple items: (1, abc, (2,4,6))
[ ]	Straight brackets enclose one or more optional items. Straight brackets are also used to indicate the map data type. In this case <> is used to indicate optional items.	Optional items: [INNER | OUTER]
{ }	Curly brackets enclose two or more items, one of which is required. Curly brackets also used to indicate the bag data type. In this case <> is used to indicate required items.	Two items, one required: { gen_blk | nested_gen_blk }
…	Horizontal ellipsis points indicate that you can repeat a portion of the code.	Pig Latin syntax statement: cat path [path …]
UPPERCASE lowercase	In general, uppercase type indicates elements the system supplies. In general, lowercase type indicates elements that you supply. Note: The names (aliases) of relations and fields are case sensitive. The names of Pig Latin functions are case sensitive. All other Pig Latin keywords are case	Pig Latin statement: A = LOAD 'data' AS (f1:int); 1. LOAD, AS supplied BY system 2. A, f1 are names (aliases) 3. data supplied by you

	insensitive.	
italics	Italic type indicates placeholders or variables for which you must supply values.	Pig Latin syntax: alias = LIMIT alias n; You supply the values for placeholder alias and variable n.

2. Pig Latin Statements

A Pig Latin statement is an operator that takes a relation as input and produces another relation as output. (This definition applies to all Pig Latin operators except LOAD and STORE which read data from and write data to the file system.) Pig Latin statements can span multiple lines and must end with a semi-colon (;). Pig Latin statements are generally organized in the following manner.

1. A LOAD statement reads data from the file system.
2. A series of "transformation" statements process the data.
3. A STORE statement writes output to the file system; or, a DUMP statement displays output to the screen.

2.1. Processing Pig Latin Statements

You can execute Pig Latin statements interactively using the Grunt shell or you can place Pig Latin statements in a script and run the script. Either way, Pig processes Pig Latin statements as follows:

1. First, Pig validates the syntax and semantics of all statements.
2. Next, if Pig has encountered a DUMP or STORE, Pig will execute all statements connected to the specified DUMP or STORE.

In this example Pig will validate, but not execute, the LOAD and FOREACH statements.

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

B = FOREACH A GENERATE name;
```

In this example, Pig will validate the LOAD, FOREACH, and DUMP statements. Then, if there are no errors, Pig will execute these statements.

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

```
B = FOREACH A GENERATE name;
DUMP B;
(John)
(Mary)
(Bill)
(Joe)
```

2.2. Using Comments in Scripts

If you place Pig Latin statements in a script, the script can include comments.

1. For multi-line comments use /* …. */
2. For single line comments use --

```
/* myscript.pig
My script includes three simple Pig Latin Statements.
*/
A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float); -- load statement
B = FOREACH A GENERATE name;  -- foreach statement
DUMP B;  --dump statement
```

3. Relations, Bags, Tuples, and Fields

As noted, Pig Latin statements work with relations. A relation can be defined as follows:

1. A relation is a bag (more specifically, an outer bag).
2. A bag is a collection of tuples.
3. A tuple is an ordered set of fields.
4. 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.

3.1. 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)
```

3.2. Referencing Fields

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

1. 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.
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; for example, f1, f2, f3 or a, b, c or name, age, gpa.

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

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

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

3.3. 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.

1. Use the schemas for complex data types to name fields that are complex data types.
2. 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)
```

1.

4. 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 are case insensitive.

In the example below, note the following:

1. The names (aliases) of relations A, B, and C are case sensitive.
2. The names (aliases) of fields f1, f2, and f3 are case sensitive.
3. Function names PigStorage and COUNT are case sensitive.
4. 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.
5. 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;
```

5. Working with Data

Pig Latin allows you to work with data in many ways. In general, and as a starting point:

1. Use the FILTER operator to work with tuples or rows of data. Use the FOREACH operator to work with columns of data.
2. Use the GROUP operator to group data in a single relation. Use the COGROUP and JOIN operators to group or join data in two or more relations.
3. Use the UNION operator to merge the contents of two or more relations. Use the SPLIT operator to partition the contents of a relation into multiple relations.

6. Increasing Parallelism

To increase the parallelism of a job, include the PARALLEL clause with the COGROUP, CROSS, DISTINCT, GROUP, JOIN and ORDER operators. PARALLEL controls the number of reducers only; the number of maps is determined by the input data (see the Pig User Cookbook).

7. Increasing Performance

You can increase or optimize the performance of your Pig Latin scripts by following a few simple rules (see the Pig User Cookbook).

8. Retrieving Results

Pig Latin includes operators you can use to retrieve the results of your Pig Latin statements:

1. Use the DUMP operator to display results to a screen.
2. Use the STORE operator to write results to a file on the file system.

9. Debugging Pig Latin Scripts

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

1. Use the DESCRIBE operator to review the schema of a relation.
2. Use the EXPLAIN operator to view the logical, physical, or map reduce execution plans

to compute a relation.

3. Use the ILLUSTRATE operator to view the step-by-step execution of a series of statements.

10. Data Types

Simple Data Types	Description	Example
Scalars		
int	Signed 32-bit integer	10
long	Signed 64-bit integer	Data: 10L or 10l Display: 10L
float	32-bit floating point	Data: 10.5F or 10.5f or 10.5e2f or 10.5E2F Display: 10.5F or 1050.0F
double	64-bit floating point	Data: 10.5 or 10.5e2 or 10.5E2 Display: 10.5 or 1050.0
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:

1. 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;
```

1. 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);
```

1. 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 …
```

1. 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 …
```

10.1. Tuple

A tuple is an ordered set of fields.

10.1.1. Syntax

(field [, field …])

10.1.2. Terms

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

10.1.3. 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:

1. 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.
2. In a non-load statement, if a requested field is missing from a tuple, Pig will inject null.

10.1.4. Examples

In this example the tuple contains three fields.

(John,18,4.0F)

10.2. Bag

A bag is a collection of tuples.

10.2.1. Syntax: Inner bag

{ tuple [, tuple …] }

10.2.2. Terms

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

10.2.3. Usage

Note the following about bags:

1. A bag can have duplicate tuples.
2. 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.
3. 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.

10.2.4. 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)
```

10.2.5. 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)})

10.3. Map

A map is a set of key value pairs.

10.3.1. Syntax (<> denotes optional)

[key#value <, key#value …>]

10.3.2. Terms

[ ]	Maps are enclosed in straight brackets [ ].
#	Key value pairs are separated by the pound sign #.
key	Must be a scalar data type. Must be a unique value.
value	Any data type.

10.3.3. Usage

Key values within a relation must be unique.

10.3.4. Example

In this example the map includes two key value pairs.

[name#John,phone#5551212]

11. Nulls

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.

11.1. Nulls and Operators

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

Operator	Interaction

Comparison operators: ==, != >, < >=, <=	If either sub-expression is null, the result is null.
Comparison operator: matches	If either the string being matched against or the string defining the match is null, the result is null.
Arithmetic operators: + , -, *, / % modulo ? bincond	If either sub-expression is null, the resulting expression is null.
Null operator: is null	If the tested value is null, returns true; otherwise, returns false.
Null operator: is not null	If the tested value is not null, returns true; otherwise, returns false.
Dereference operators: tuple (.) or map (#)	If the de-referenced tuple or map is null, returns null.
Cast operator	Casting a null from one type to another type results in a null.
Functions: AVG, MIN, MAX, SUM	These functions ignore nulls.
Function: COUNT	This function counts all values, including nulls.
Function: CONCAT	If either sub-expression is null, the resulting expression is null.

Function: SIZE	If the tested object is null, returns null.

For Boolean sub-expressions, note the results when nulls are used with these operators:

1. 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).
2. Bincond operator – If a Boolean sub-expression results in null value, the resulting expression is null (see the interactions above for Arithmetic operators)

11.1.1. Example: COUNT function

As noted, the COUNT function counts all values, including nulls. If you don't want the function to count null values, you can use one of the methods shown here.

In this example the is not null operator is used to filter (remove) all null values before subsequent operations, including the COUNT function, are applied.

```
A = LOAD 'data';
B = FILTER A BY $1 is not null;
C = GROUP A BY $0;
D = FOREACH B GENERATE GROUP, COUNT(B.$1);
```

Suppose you have written a function, RemoveNulls, to filter null values. In this example RemoveNulls is used to filter nulls values for the COUNT function only.

```
A = LOAD 'data';
B = GROUP A BY $0;
D = FOREACH B GENERATE GROUP, COUNT(RemoveNulls($1));
```

11.2. 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;
```

11.3. Operations That Produce Nulls

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

1. Division by zero
2. Returns from user defined functions (UDFs)
3. Dereferencing a field that does not exist.
4. 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.
5. Accessing a field that does not exist in a tuple. As a further explanation, see the examples below.

11.3.1. 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)
```

11.4. 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;
```

12. 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 applicable.
Complex Data Types		
tuple	(19, 2, 1)	A constant in this form creates a tuple.
bag	{ (19, 2), (1, 2) }	A constant in this form creates a bag.
map	['name' # 'John', 'ext' # 5555]	A constant in this form creates a map.

Please note the following:

1. 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.

2. 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.
3. 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:

1. A tuple can contain fields of any data type
2. A bag is a collection of tuples
3. A map key must be a scalar; a map value can be any data type

Complex constants 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)};
```

13. 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:

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

In a Pig Latin statement, an arithmetic expression could look like this:

```
X = GROUP A BY f2*f3;
```

1. A string expression could look like this, where a and b are both chararrays:

```
X = FOREACH A GENERATE CONCAT(a,b);
```

1. A boolean expression could look like this:

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

14. Schemas

Schemas enable you to assign names to 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 are defined using the AS keyword with the LOAD, STREAM, and FOREACH operators. If you define a schema using the LOAD operator, then it is the load function that enforces the schema (see the LOAD operator and the User-Defined Function Manual for more information).

Note the following:

1. You can define a schema that includes both the field name and field type.
2. You can define a schema that includes the field name only; in this case, the field type defaults to bytearray.
3. 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.

14.1. Schemas with LOAD and STREAM Statements

With LOAD and STREAM statements, 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);
```

14.2. Schemas with FOREACH Statements

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

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

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

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

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

14.3. Schemas for Simple Data Types

Simple data types include int, long, float, double, chararray, and bytearray.

14.3.1. Syntax

```
(alias[:type]) [, (alias[:type]) …] )
```

14.3.2. Terms

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

14.3.3. 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)
```

14.4. Schemas for Complex Data Types

Complex data types include tuples, bags, and maps.

14.5. Tuple Schema

A tuple is an ordered set of fields.

14.5.1. Syntax

```
alias[:tuple] (alias[:type]) [, (alias[:type]) …] )
```

14.5.2. Terms

alias	The name assigned to the tuple.
:tuple	(Optional) The data type, tuple (case insensitive).
( )	The designation for a tuple, a set of parentheses.
alias[:type]	The constituents of the tuple, where the schema definition rules for the corresponding type applies to the constituents of the tuple: 1. alias – the name assigned to the field 2. type (optional) – the simple or complex data type assigned to the field

14.5.3. 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))
```

14.6. Bag Schema

A bag is a collection of tuples.

14.6.1. Syntax

```
alias[:bag] {tuple}
```

14.6.2. 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).

14.6.3. 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)})
```

14.7. Map Schema

A map is a set of key value pairs.

14.7.1. Syntax (where <> means optional)

alias<:map> [ ]

14.7.2. 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 [ ].

14.7.3. Example

In this example the schema defines a map. 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])
```

14.8. Schemas for Multiple Types

You can define schemas for data that includes multiple types.

14.8.1. 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:[] );
```

15. Parameter Substitution

15.1. Description

Substitute values for parameters at run time.

15.1.1. Syntax: Specifying parameters using the Pig command line

```
pig {–param param_name = param_value | –param_file file_name} [-debug | -dryrun] script
```

15.1.2. Syntax: Specifying parameters using preprocessor statements in a Pig script

```
{%declare | %default} param_name param_value
```

15.1.3. Terms

pig	Keyword
–param	Flag. Use this option when the parameter is included in the command line. Multiple parameters can be specified. If the same parameter is specified multiple times, the last value will be used and a warning will be generated. Command line parameters and parameter files can be combined with command line parameters taking precedence.
param_name	The name of the parameter. The parameter name has the structure of a standard language identifier: it must start with a letter or underscore followed by any number of letters, digits, and underscores. Parameter names are case insensitive. If you pass a parameter to a script that the script does not use, this parameter is silently ignored. If the script

	has a parameter and no value is supplied or substituted, an error will result.
param_value	The value of the parameter. A parameter value can take two forms: 1. A sequence of characters enclosed in single or double quotes. In this case the unquoted version of the value is used during substitution. Quotes within the value can be escaped with the backslash character (\ ). Single word values that don't use special characters such as % or = don't have to be quoted. 2. A command enclosed in back ticks. The value of a parameter, in either form, can be expressed in terms of other parameters as long as the values of the dependent parameters are already defined.
–param_file	Flag. Use this option when the parameter is included in a file. Multiple files can be specified. If the same parameter is present multiple times in the file, the last value will be used and a warning will be generated. If a parameter present in multiple files, the value from the last file will be used and a warning will be generated. Command line parameters and parameter files can be combined with command line parameters taking precedence.
file_name	The name of a file containing one or more parameters. A parameter file will contain one line per parameter. Empty lines are allowed. Perl-style (#) comment lines are also allowed. Comments must take a full line and # must be the first character on the line. Each parameter line will be of the form: param_name = param_value. White spaces around = are allowed but are optional.
–debug	Flag. With this option, the script is run and a fully substituted Pig script produced in the current working directory named original_script_name.substituted

–dryrun	Flag. With this option, the script is not run and a fully substituted Pig script produced in the current working directory named original_script_name.substituted
script	A pig script. The pig script must be the last element in the Pig command line. 1. If parameters are specified in the Pig command line or in a parameter file, the script should include a $param_name for each para_name included in the command line or parameter file. 2. If parameters are specified using the preprocessor statements, the script should include either %declare or %default. 3. In the script, parameter names can be escaped with the backslash character (\ ) in which case substitution does not take place.
%declare	Preprocessor statement included in a Pig script. Use to describe one parameter in terms of other parameters. The declare statement is processed prior to running the Pig script. The scope of a parameter value defined using declare is all the lines following the declare statement until the next declare statement that defines the same parameter is encountered.
%default	Preprocessor statement included in a Pig script. Use to provide a default value for a parameter. The default value has the lowest priority and is used if a parameter value has not been defined by other means. The default statement is processed prior to running the Pig script. The scope is the same as for %declare.

15.1.4. Usage

Parameter substitution enables you to write Pig scripts that include parameters and to supply values for these parameters at run time. For instance, suppose you have a job that needs to

run every day using the current day's data. You can create a Pig script that includes a parameter for the date. Then, when you run this script you can specify or supply a value for the date parameter using one of the supported methods.

15.1.4.1. Specifying Parameters

You can specify parameter names and parameter values as follows:

1. As part of a command line.
2. In parameter file, as part of a command line.
3. With the declare statement, as part of Pig script.
4. With default statement, as part of a Pig script.

15.1.4.2. Precedence

Precedence for parameters is as follows:

1. Highest - parameters defined using the declare statement
2. Next - parameters defined in the command line
3. Lowest - parameters defined in a script

15.1.4.3. Processing Order and Precedence

Parameters are processed as follows:

1. Command line parameters are scanned in the order they are specified on the command line.
2. Parameter files are scanned in the order they are specified on the command line. Within each file, the parameters are processed in the order they are listed.
3. Declare and default preprocessors statements are processed in the order they appear in the Pig script.

15.1.5. Example: Specifying parameters in the command line

Suppose we have a data file called 'mydata' and a pig script called 'myscript.pig'.

1. mydata

```
1	2	3
4	2	1
8	3	4
```

1. myscript.pig

```
A = LOAD '$data' USING PigStorage() AS (f1:int, f2:int, f3:int);
DUMP A;
```

In this example the parameter (data) and the parameter value (mydata) are specified in the command line. If the parameter name in the command line (data) and the parameter name in the script ($data) do not match, the script will not run. If the value for the parameter (mydata) is not found, an error is generated.

```
$ pig –param data=mydata myscript.pig
(1,2,3)
(4,2,1)
(8,3,4)
```

15.1.6. Example: Specifying parameters using a parameter file

Suppose we have a parameter file called 'myparams.'

```
# my parameters
data1 = mydata1
cmd = `generate_name`
```

In this example the parameters and values are passed to the script using the parameter file.

```
$ pig –param_file myparams script2.pig
```

15.1.7. Example: Specifying parameters using the declare statement

In this example the command is executed and its stdout is used as the parameter value.

```
%declare CMD `generate_date`
A = LOAD '/data/mydata/$CMD';
B = FILTER A BY $0>'5';
etc …
```

15.1.8. Example: Specifying parameters using the default statement

In this example the parameter (DATE) and value ('20090101') are specified in the Pig script using the default statement. If a value for DATE is not specified elsewhere, the default value 20090101 is used.

```
%default DATE '20090101';

A = load '/data/mydata/$DATE';

etc …
```

15.1.9. Examples: Specifying parameter values as a sequence of characters

In this example the characters (in this case, Joe's URL) can be enclosed in single or double quotes, and quotes within the sequence of characters can be escaped.

```
%declare DES 'Joe\'s URL';

A = LOAD 'data' AS (name, description, url);

B = FILTER A BY description == '$DES';

etc …
```

In this example single word values that don't use special characters (in this case, mydata) don't have to be enclosed in quotes.

```
$ pig –param data=mydata myscript.pig
```

15.1.10. Example: Specifying parameter values as a command

In this example the command is enclosed in back ticks. First, the parameters mycmd and date are substituted when the declare statement is encountered. Then the resulting command is executed and its stdout is placed in the path before the load statement is run.

```
%declare CMD `$mycmd $date`

A = LOAD '/data/mydata/$CMD';

B = FILTER A BY $0>'5';

etc …
```

16. Keywords

A	F	M	Functions
and	f	map	AVG
all	F	matches	BinaryDeserializer
as	filter	mkdir	BinarySerializer
asc	flatten	mv	BinStorage
	float	N	CONCAT
B	foreach	not	COUNT
bag	G	null	DIFF
by	generate	O	MIN
bytearray	group	or	MAX
C	H	order	PigDump
cache	help	outer	PigStorage
cat	I	output	SIZE
cd	if	P	SUM
chararray	illustrate	parallel	TextLoader
cogroup	inner	pig	TOKENIZE
copyFromLocal	input	pwd	
copyToLocal	int	Q	Symbols

cp	into	quit	== != < > <= >=
cross	is	R	+ - * / %
D	J	register	? $. # ( ) [ ] { }
distinct	join	rm	
define	K	rmf	Preprocessor Statements
desc	kill	run	%declare
describe	L	S	%default
double	l	set	
du	L	ship	
dump	limit	split	
E	load	stderr	
e	long	stdin	
E	ls	stdout	
exec		store	
explain		stream	
		T	
		through	
		tuple	
		U	

		union	
		using	

17. Arithmetic Operators

17.1. Description

Operator	Symbol	Notes
addition	+	
subtraction	-	
multiplication	*	
division	/	
modulo	%	Returns the remainder of a divided by b (a%b).
bincond	? :	condition ? value_if_true : value_if_false

17.1.1. 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)})
```

17.1.2. 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)
```

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

* bytearray cast as this data type

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

									double
chararray								error	error
bytearray									cast as double

17.1.4. Types Table: multiplication (*) and division (/) operators

* bytearray cast as this data type

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

17.1.5. Types Table: modulo (%) operator

	int	long	bytearray

int	int	long	cast as int
long		long	cast as long
bytearray			error

18. Comparison Operators

18.1. Description

Operator	Symbol	Notes
equal	==	
not equal	!=	
less than	<	
greater than	>	
less than or equal to	<=	
greater than or equal to	>=	
pattern matching	matches	Regular expression matching. Use the Java format for regular expressions.

Use the comparison operators with numeric and string data.

18.1.1. Example: numeric

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

18.1.2. Example: string

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

18.1.3. Example: matches

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

18.1.4. Types Table: equal (==) and not equal (!=) operators

* bytearray cast as this data type

	bag	tuple	map	int	long	float	double	chararray	bytearray
bag	error	error	error	error	error	error	error	error	error
tuple		boolean (see Note 1)	error	error	error	error	error	error	error
map			boolean (see Note 2)	error	error	error	error	error	error
int				boolean	boolean	boolean	boolean	error	cast as boolean
long					boolean	boolean	boolean	error	cast as boolean
float						boolean	boolean	error	cast as boolean
double							boolean	error	cast as boolean
chararray								boolean	cast as boolean
bytearray									boolean

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

s A[i] = = B[i])

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

18.1.5.

	bag	tuple	map	int	long	float	double	chararray	bytearray
bag	error	error	error	error	error	error	error	error	error
tuple		error	error	error	error	error	error	error	error
map			error	error	error	error	error	error	error
int				boolean	boolean	boolean	boolean	error	boolean (bytearray cast as int)
long					boolean	boolean	boolean	error	boolean (bytearray cast as long)
float						boolean	boolean	error	boolean (bytearray cast as float)
double							boolean	error	boolean (bytearray cast as double)
chararray								boolean	boolean (bytearray cast as chararray)
bytearray									boolean

18.1.6. Types Table: matches operator

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

	chararray	bytearray*
chararray	boolean	boolean
bytearray	boolean	boolean

19. Null Operators

19.1. Description

Operator	Symbol	Notes
is null	is null	
is not null	is not null	

19.1.1. Example

```
X = FILTER A BY f1 is not null;
```

19.2. Types Table

The null operators can be applied to all data types. For more information, see Nulls.

20. Boolean Operators

20.1. Description

Operator	Symbol	Notes
AND	and	
OR	or	

NOT	not	

Pig does not support a boolean data type. However, the result of a boolean expression (an expression that includes boolean and comparison operators) is always of type boolean (true or false).

20.1.1. Example

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

21. Dereference Operators

21.1. Description

Operator	Symbol	Notes
tuple dereference	. (dot)	Retrieve a field from a tuple.
bag dereference	. (dot)	Retrieve a column from a bag.
map dereference	#	For a key#value pair, look up the value for the specified key.

Note the following:

1. Tuple dereferencing can be done by name (tuple.field_name) or position (mytuple.$0). Note that if the dot operator is applied to a bytearray, the bytearray will be assumed to be a tuple.
2. Bag dereferencing can be done by name (bag.field_name) or position (bag.$0).
3. Map dereferencing must be done by key (field_name#key or $0#key). If the pound operator is applied to a bytearray, the bytearray is assumed to be a map. If the key does not exist, the empty string is returned.

21.1.1. Example: Tuple

1. 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)
```

1.

21.1.2. 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}) |
-------------------------------------------------------------
|     | 8          | {(8, 3, 4), (8, 4, 3)}            |
-------------------------------------------------------------
```

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)})
```

21.1.3. Example: Tuple and Bag

1. 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)})
```

21.1.4. Example: Map

1. 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])
```

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

```
X = FOREACH A GENERATE f2#'open';
DUMP X;
(apache)
()
()
()
```

1.

22. Sign Operators

22.1. Description

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

22.1.1. Example

```
A = LOAD 'data' as (x, y, z);
B = FOREACH A GENERATE -x, y;
```

22.1.2. Types Table: negation (-) operator

bag	error
tuple	error
map	error
int	int
long	long
float	float
double	double
chararray	error
bytearray	double (as double)

23. Cast Operators

23.1. Description

Pig Latin supports casts as shown in this table.

	to								
from	bag	tuple	map	int	long	float	double	chararray	bytearray
bag		error	error	error	error	error	error	error	error
tuple	error		error	error	error	error	error	error	error
map	error	error		error	error	error	error	error	error

int	error	error	error		yes	yes	yes	yes	error
long	error	error	error	yes		yes	yes	yes	error
float	error	error	error	yes	yes		yes	yes	error
double	error	error	error	yes	yes	yes		yes	error
chararray	error	error	error	error	error	error	error		error
bytearray	yes	yes	yes	yes	yes	yes	yes	yes	

23.1.1. Syntax

{(data_type) | (tuple(data_type)) | (bag{tuple(data_type)}) | (map[]) } field

23.1.2. Terms

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

23.1.3. Usage

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

Please note the following:

1. 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;
```

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
```

2. 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.
3. Downcasts may cause loss of data. For example casting from long to int may drop bits.

23.1.4. Examples

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

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

DUMP A;

(1,2,3)

(4,2,1)

(8,3,4)

(4,3,3)

(7,2,5)

(8,4,3)

B = GROUP A BY f1;

DUMP B;

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

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

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

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

DESCRIBE B;

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

```
X = FOREACH B GENERATE group, (chararray)COUNT(A) AS total;
(1,1)
(4,2)
(7,1)
(8,2)

DESCRIBE X;
X: {group: int,total: chararray}
```

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

```
cat data;
(1,2,3)
(4,2,1)
(8,3,4)

A = LOAD 'data' AS fld:bytearray;
DESCRIBE A;
a: {fld: bytearray}
DUMP A;
((1,2,3))
((4,2,1))
((8,3,4))

B = FOREACH A GENERATE (tuple(int,int,float))fld;
DESCRIBE B;
b: {(int,int,float)}
DUMP B;
((1,2,3))
((4,2,1))
((8,3,4))
```

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

```
cat data
{(4829090493980522200L)}
{(4893298569862837493L)}
{(1297789302897398783L)}

A = LOAD 'data' AS fld:bytearray;
DESCRIBE A;
A: {fld: bytearray}
DUMP A;
({(4829090493980522200L)})
({(4893298569862837493L)})
({(1297789302897398783L)})

B = FOREACH A GENERATE (bag{tuple(long)})fld;
DESCRIBE B;
B: {{(long)}}
DUMP B;
({(4829090493980522200L)})
({(4893298569862837493L)})
({(1297789302897398783L)})
```

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

```
cat data
[open#apache]
[apache#hadoop]
[hadoop#pig]
[pig#grunt]

A = LOAD 'data' AS fld:bytearray;
```

```
DESCRIBE A;
A: {fld: bytearray}
DUMP A;
([open#apache])
([apache#hadoop])
([hadoop#pig])
([pig#grunt])

B = FOREACH A GENERATE ((map[])fld;
DESCRIBE B;
B: {map[ ]}
DUMP B;
([open#apache])
([apache#hadoop])
([hadoop#pig])
([pig#grunt])
```

24. Relational Operators

24.1. COGROUP

Groups the data in two or more relations.

24.1.1. Syntax

```
alias =COGROUP alias BY field_alias [INNER | OUTER] , alias  BY field_alias [INNER | OUTER] [PARALLEL n] ;
```

24.1.2. Terms

alias	The name a relation.
field_alias	The name of one or more fields in a relation.

	If multiple fields are specified, separate with commas and enclose in parentheses. For example, X = COGROUP A BY (f1, f2); The number of fields specified in each BY clause must match. For example, X = COGROUP A BY (a1,a2,a3), B BY (b1,b2,b3);
BY	Keyword.
INNER	Keyword.
OUTER	Keyword.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. The optimal number of parallel tasks depends on the amount of memory on each node and the memory required by each of the tasks. To determine n, use the following as a general guideline: n = (nr_nodes - 1) * 0.45 * nr_GB where nr_nodes is the number of nodes used and nr_GB is the amount of physical memory on each node. Note the following: 1. Parallel only affects the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block. 2. If you don't specify parallel, you still get the same map parallelism but only one reduce task.

24.1.3. Usage

The COGOUP operator groups the data in two or more relations based on the common field values. Note that the COGROUP and JOIN operators perform similar functions. COGROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples.

24.1.4. Examples

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,Alice)})
(Bob,{(Bob,dog),(Bob,cat)},{(Paul,Bob)})
(Jane,{},{(Paul,Jane)})
```

In this example tuples are co-grouped and the INNER keyword is used to ensure that only

bags with at least one tuple are returned.

```
X = COGROUP A BY owner INNER, B BY friend2 INNER;
DUMP X;
(Alice,{(Alice,turtle),(Alice,goldfish),(Alice,cat)},{(Cindy,Alice),(Mark,Alice)})
(Bob,{(Bob,dog),(Bob,cat)},{(Paul,Bob)})
```

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,Alice)})
```

24.2. CROSS

Computes the cross product of two or more relations.

24.2.1. Syntax

```
alias = CROSS alias, alias [, alias …] [PARALLEL n];
```

24.2.2. Terms

alias	The name of a relation.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. The optimal number of parallel tasks depends on the amount of memory on each node and the memory required by each of the tasks. To determine n, use the following as a general guideline: n = (nr_nodes - 1) * 0.45 * nr_GB where nr_nodes is the number of nodes used and

	nr_GB is the amount of physical memory on each node. Note the following: 1. Parallel only affects the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block. 2. If you don't specify parallel, you still get the same map parallelism but only one reduce task.

24.2.3. 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.

24.2.4. 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)

24.3. DISTINCT

Removes duplicate tuples in a relation.

24.3.1. Syntax

alias = DISTINCT alias [PARALLEL n];

24.3.2. Terms

alias	The name of the relation.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. The optimal number of parallel tasks depends on the amount of memory on each node and the memory required by each of the tasks. To determine n, use the following as a general guideline: n = (nr_nodes - 1) * 0.45 * nr_GB where nr_nodes is the number of nodes used and nr_GB is the amount of physical memory on each node. Note the following: 1. Parallel only affects the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block. 2. If you don't specify parallel, you still get the same map parallelism but only one reduce task.

24.3.3. 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 … GENERATE to select the fields, and then use DISTINCT.

24.3.4. 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)
```

24.4. DUMP

Displays the contents of a relation.

24.4.1. Syntax

```
DUMP alias;
```

24.4.2. Terms

alias	The name of a relation.

24.4.3. Usage

Use the DUMP operator to run (execute) a Pig Latin statement and to display the contents of an alias. You can use DUMP as a debugging device to make sure the results you are expecting are being generated.

24.4.4. Example

In this example a dump is performed after each statement.

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);
DUMP A;
(John,18,4.0F)
(Mary,19,3.7F)
(Bill,20,3.9F)
(Joe,22,3.8F)
(Jill,20,4.0F)

B = FILTER A BY name matches 'J.+';
DUMP B;
(John,18,4.0F)
(Joe,22,3.8F)
(Jill,20,4.0F)
```

24.5. FILTER

Selects tuples from a relation based on some condition.

24.5.1. Syntax

```
alias = FILTER alias  BY expression;
```

24.5.2. Terms

alias	The name of the relation.

BY	Required keyword.
expression	An expression.

24.5.3. 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.

24.5.4. 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)
```

24.6. FOREACH … GENERATE

Generates data transformations based on columns of data.

24.6.1. Syntax

```
alias  = FOREACH { gen_blk | nested_gen_blk } [AS schema];
```

24.6.2. Terms

alias	The name of relation (outer bag).
gen_blk	FOREACH … GENERATE used with a relation (outer bag). Use this syntax: alias = FOREACH alias GENERATE expression [expression ….]

nested_gen_blk	FOREACH … GENERATE used with a inner bag. Use this syntax:
	
	alias = FOREACH nested_alias {
	  alias = nested_op; [alias = nested_op; …]
	  GENERATE expression [expression ….]
	};
	
	Where:
	The nested block is enclosed in opening and closing brackets { … }.

	The GENERATE keyword must be the last statement within the nested block.
expression	An expression.
nested_alias	The name of the inner bag.
nested_op	Allowed operations are FILTER, ORDER, and DISTINCT. The FOREACH … GENERATE operation itself is not allowed since this could lead to an arbitrary number of nesting levels.
AS	Keyword.
schema	A schema using the AS keyword (see Schemas). 1. If the FLATTEN keyword is used, enclose the schema in parentheses. 2. If the FLATTEN keyword is not used, don't enclose the schema in parentheses.

24.6.3. 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:

1. If A is a relation (outer bag), a FOREACH statement could look like this.

```
X = FOREACH A GENERATE f1;
```

1. 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);
}
```

24.6.4. Examples

Suppose we have relations A, B, and C (see the GROUP operator for information about the field names in relation C).

```
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)

C = COGROUP A BY a1 inner, B BY b1 inner;
DUMP C;
(1,{(1,2,3)},{(1,3)})
(4,{(4,2,1),(4,3,3)},{(4,6),(4,9)})
(8,{(8,3,4),(8,4,3)},{(8,9)})

ILLUSTRATE C;
```

```
etc …
---------------------------------------------------------------------------------
| c     | group: int | a: bag({a1: int,a2: int,a3: int}) | B: bag({b1: int,b2: int}) |
---------------------------------------------------------------------------------
|       | 1          | {(1, 2, 3)}                       | {(1, 3)}                  |
---------------------------------------------------------------------------------
```

24.6.5. Example: Projection

In this example the asterisk (*) is used to project all fields from relation A to relation X (this is similar to SQL Select *). 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)
```

24.6.6. 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)})
```

24.6.7. 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)
```

24.6.8. 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)
```

24.6.9. Example: Flattening

In this example the FLATTEN keyword 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)
```

24.6.10. Example: Nested Block

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

```
A = LOAD 'data' AS (url:chararray,outline: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.

```
X = foreach B {
      FA= FILTER A BY outlink == 'www.xyz.org';
      PA = FA.outlink;
      DA = DISTINCT PA;
      GENERATE GROUP, COUNT(DA);
}
DUMP X;
(www.ddd.com,1L)
(www.www.com,1L)
```

24.7. GROUP

Groups the data in a single relation.

24.7.1. Syntax

```
alias = GROUP alias [BY {[field_alias [, field_alias]] | * | [expression] } ]  [ALL]  [PARALLEL n];
```

24.7.2. Terms

alias	The name of a relation.
BY	Keyword. Use this clause to group the relation by fields or by expression.
field_alias	The name of a field in a relation. This is the group

	key or key field. A relation can be grouped by a single field (f1) or by the composite value of multiple fields (f1,f2).
*	The asterisk. A designator for all fields in the relation.
expression	An expression.
ALL	Keyword. Use ALL if you want all tuples to go to a single group; for example, when doing aggregates across entire relations.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. The optimal number of parallel tasks depends on the amount of memory on each node and the memory required by each of the tasks. To determine n, use the following as a general guideline: n = (nr_nodes - 1) * 0.45 * nr_GB where nr_nodes is the number of nodes used and nr_GB is the amount of physical memory on each node. Note the following: 1. Parallel only affects the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block. 2. If you don't specify parallel, you still get the same map parallelism but only one reduce task.

24.7.3. Usage

The GROUP operator groups together tuples that have the same group key (key field). The result of a GROUP operation is a relation that includes one tuple per group. This tuple contains two fields:

1. The first field is named "group" (do not confuse this with the GROUP operator) and is the same type of the group key.
2. 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.

24.7.4. Example

1. 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)})
```

1. 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)})
```

24.8. 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)})
```

24.9. JOIN

Joins two or more relations based on common field values.

24.9.1. Syntax

```
alias = JOIN alias BY field_alias, alias BY field_alias [, alias BY field_alias …] [USING "replicated"] [PARALLEL n];
```

24.9.2. Terms

alias	The name of a relation.
BY	Keyword
field_alias	The name of a field in a relation. For the BY clause, field_alias must be in alias. Example: X = JOIN A BY fieldA, B BY fieldB, C BY fieldC;
USING	Keyword
"replicated"	Use to perform fragment replicate join where one or more relations are small enough to fit into main memory.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. The optimal number of parallel tasks depends on the amount of memory on each node and the memory required by each of the tasks. To determine n, use the following as a general guideline: n = (nr_nodes - 1) * 0.45 * nr_GB

	where nr_nodes is the number of nodes used and nr_GB is the amount of physical memory on each node. Note the following: 1. Parallel only affects the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block. 2. If you don't specify parallel, you still get the same map parallelism but only one reduce task.

24.9.3. Usage

Use the JOIN operator to join two or more relations based on common field values. The JOIN operator always performs an inner join. Note that the JOIN and COGROUP operators perform similar functions. JOIN creates a flat set of output records while COGROUP creates a nested set of output records.

Fragment replicate join is a special type of join that works well if one relation is small enough to fit into main memory. In such cases, Pig can perform a very efficient join because all of the hadoop work is done on the map side. In this type of join the large relation is followed by one or more small relations. The small relations must be small enough to fit into main memory; if they don't, the process fails and an error is generated.

Note: Fragment replicate joins are experimental; we don't have a strong sense of how small the small relation must be to fit into memory. In our tests with a simple query that involves just a JOIN, a relation of up to 100 M can be used if the process overall gets 1 GB of memory. Please share your observations and experience with us.

24.9.4. 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)
```

24.9.5. Example: Fragment Replicate Join

In this example, a large relation is joined with two smaller relations. Note that the large relation comes first followed by the smaller relations; and, all small relations together must fit into main memory, otherwise an error is generated.

```
big = LOAD 'big_data' AS (b1,b2,b3);
tiny = LOAD 'tiny_data' AS (t1,t2,t3);
```

```
mini = LOAD 'mini_data' AS (m1,m2,m3);
C = JOIN big BY b1, tiny BY t1, mini BY m1 USING "replicated";
```

24.10. LIMIT

Limits the number of output tuples.

24.10.1. Syntax

```
alias = LIMIT alias  n;
```

24.10.2. Terms

alias	The name of a relation.
n	The number of tuples.

24.10.3. 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, the output will include all tuples in the relation.

There is no guarantee which 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.

24.10.4. 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)
```

24.11. LOAD

Loads data from the file system.

24.11.1. Syntax

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

24.11.2. Terms

'data'	The name of the file or directory, in single quotes. If you specify a directory name, all the files in the directory are loaded. You can use hadoop-supported globing to specify files at the file system or directory levels (see hadoop glob documentation for details on globing syntax).
USING	Keyword. If the USING clause is omitted, the default load function PigStorage is used.
function	The load function. 1. You can use a built-in function (see the load/store functions). PigStorage is the default load function and does not need to be specified (simply omit the USING clause). 2. You can write your own load function (see the User-Defined Function Manual) if your data is in a format that cannot be processed by the built-in functions.
AS	Keyword.
schema	A schema using the AS keyword, enclosed in parentheses (see Schemas). The loader produces the data of the type specified by the schema. If the data does not conform to the schema, depending on the loader, either a null value or an error is generated.

	Note: For performance reasons the loader may not immediately convert the data to the specified format; however, you can still operate on the data assuming the specified type.

24.11.3. Usage

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

24.11.4. 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.

24.12. ORDER

Sorts a relation based on one or more fields.

24.12.1. Syntax

```
alias = ORDER alias BY { * [ASC|DESC] | field_alias [ASC|DESC] [, field_alias [ASC|DESC] …] }
[PARALLEL n];
```

24.12.2. Terms

alias	The name of a relation.
BY	Required keyword.
*	Represents all fields in a relation. If relation A has three fields a1, a2, a3, then these statements are equivalent: 1. X = ORDER A BY a1,a2,a3; 2. X = ORDER A BY * ;

ASC	Sort in ascending order.
DESC	Sort in descending order.
field_alias	A field in the relation.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. The optimal number of parallel tasks depends on the amount of memory on each node and the memory required by each of the tasks. To determine n, use the following as a general guideline: n = (nr_nodes - 1) * 0.45 * nr_GB where nr_nodes is the number of nodes used and nr_GB is the amount of physical memory on each node. Note the following: 1. Parallel only affects the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block. 2. If you don't specify parallel, you still get the same map parallelism but only one reduce task.

24.12.3. Usage

In Pig, relations are unordered (see Relations, Bags, Tuples, and Fields):

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

24.12.4. 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)
```

24.13. SPLIT

Partitions a relation into two or more relations.

24.13.1. Syntax

```
SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression …];
```

24.13.2. Terms

alias	The name of a relation.
INTO	Required keyword.

IF	Required keyword.
expression	An expression.

24.13.3. 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:

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

24.13.4. 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)
```

24.14. STORE

Stores data to the file system.

24.14.1. Syntax

```
STORE alias INTO 'directory' [USING function];
```

24.14.2. Terms

alias	The name of a relation.
INTO	Required keyword.
'directory'	The name of the storage directory, in quotes. If the directory already exists, the STORE operation will fail. The output data files, named part-nnnnn, are written to this directory.
USING	Keyword. Use this clause to name the store function. If the USING clause is omitted, the default store function PigStorage is used.
function	The store function. 1. You can use a built-in function (see the Load/Store Functions). PigStorage is the default load function and does not need to be specified (simply omit the USING clause). 2. You can write your own store function (see the User-Defined Function Manual) if your data is in a format that cannot be processed by the built-in functions.

24.14.3. Usage

Use the STORE operator to run (execute) Pig Latin statements and to store data on the file system.

24.14.4. 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
```

24.15. STREAM

Sends data to an external script or program.

24.15.1. Syntax

```
alias = STREAM alias [, alias …] THROUGH {`command` | cmd_alias } [AS schema] ;
```

24.15.2. 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.
AS	Keyword.
schema	A schema using the AS keyword, enclosed in parentheses (see Schemas).

24.15.3. 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 cmd is the defined alias.

```
A = LOAD 'data';

DEFINE cmd `stream.pl –n 5`;

B = STREAM A THROUGH cmd;
```

24.15.4. About Data Guarantees

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

1. Unordered data – No guarantee for the order in which the data is delivered to the streaming application.
2. Grouped data – The data for the same grouped key is guaranteed to be provided to the

streaming application contiguously

3. 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.

24.15.5. 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`;
```

24.15.6. 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);
```

24.15.7. Additional Examples

See the UDF statement DEFINE for additional examples.

24.16. UNION

Computes the union of two or more relations.

24.16.1. Syntax

```
alias = UNION alias, alias [, alias …];
```

24.16.2. Terms

alias	The name of a relation.

24.16.3. Usage

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

1. Does not preserve the order of tuples. Both the input and output relations are interpreted as unordered bags of tuples.
2. 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.
3. Does not eliminate duplicate tuples.

24.16.4. 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)
```

25. DESCRIBE

Returns the schema of an alias.

25.1. Syntax

```
DESCRIBE alias;
```

25.2. Terms

alias	The name of a relation.

25.3. Usage

Use the DESCRIBE operator to review the schema of a particular alias.

25.4. Example

In this example a schema is specified using the AS clause. If all data conforms to the schema, Pig will use the assigned types.

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

B = FILTER A BY name matches 'J.+';

C = GROUP B BY name;

D = FOREACH B GENERATE COUNT(B.age);

DESCRIBE A;

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

DESCRIBE B;

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

DESCRIBE C;

C: {group, chararry,B: (name: chararray,age: int,gpa: float}

DESCRIBE D;

D: {long}
```

In this example no schema is specified. All fields default to type bytearray or long (see Data Types).

```
a = LOAD 'student';

b = FILTER a BY $0 matches 'J.+';

c = GROUP b BY $0;

d = FOREACH c GENERATE COUNT(b.$1);


DESCRIBE a;

Schema for a unknown.


DESCRIBE b;

2008-12-05 01:17:15,316 [main] WARN  org.apache.pig.PigServer - bytearray is implicitly cast to chararray under LORegexp Operator

Schema for b unknown.


DESCRIBE c;
```

```
2008-12-05 01:17:23,343 [main] WARN  org.apache.pig.PigServer - bytearray is implicitly caste to chararray under LORegexp Operator

c: {group: bytearray,b: {null}}


DESCRIBE d;

2008-12-05 03:04:30,076 [main] WARN  org.apache.pig.PigServer - bytearray is implicitly caste to chararray under LORegexp Operator

d: {long}
```

26. EXPLAIN

Displays execution plans.

26.1. Syntax

```
EXPLAIN alias;
```

26.2. Terms

alias	The name of a relation.

26.3. Usage

Use the EXPLAIN operator to review the logical, physical, and map reduce execution plans that are used to compute the specified relationship.

1. The logical plan shows a pipeline of operators to be executed to build the relation. Type checking and backend-independent optimizations (such as applying filters early on) also apply.
2. The physical plan shows how the logical operators are translated to backend-specific physical operators. Some backend optimizations also apply.
3. The map reduce plan shows how the physical operators are grouped into map reduce jobs.

26.4. Example

In this example the EXPLAIN operator produces all three plans. (Note that only a portion of the output is shown in this example.)

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);
B = GROUP A BY name;
C = FOREACH B GENERATE COUNT(A.age);
EXPLAIN C;

Logical Plan:
Store xxx-Fri Dec 05 19:42:29 UTC 2008-23 Schema: {long} Type: Unknown
|
|---ForEach xxx-Fri Dec 05 19:42:29 UTC 2008-15 Schema: {long} Type: bag
etc …
------------------------------------------------
Physical Plan:
------------------------------------------------
Store(fakefile:org.apache.pig.builtin.PigStorage) - xxx-Fri Dec 05 19:42:29 UTC 2008-40
|
|---New For Each(false)[bag] - xxx-Fri Dec 05 19:42:29 UTC 2008-39
   |  |
   |  POUserFunc(org.apache.pig.builtin.COUNT)[long] - xxx-Fri Dec 05
etc …

--------------------------------------------------
| Map Reduce Plan                          |
--------------------------------------------------
MapReduce node xxx-Fri Dec 05 19:42:29 UTC 2008-41
Map Plan
Local Rearrange[tuple]{chararray}(false) - xxx-Fri Dec 05 19:42:29 UTC 2008-34
|  |
|  Project[chararray][0] - xxx-Fri Dec 05 19:42:29 UTC 2008-35
etc …
```

27. ILLUSTRATE

Displays a step-by-step execution of a sequence of statements.

27.1. Syntax

```
ILLUSTRATE alias;
```

27.2. Terms

alias	The name of a relation.

27.3. Usage

Use the ILLUSTRATE operator to review how data is transformed through a sequence of Pig Latin statements:

1. The data load statement must include a schema.
2. The Pig Latin statement used to form the relation that is used with the ILLUSTRATE command cannot include the map data type, the LIMIT and SPLIT operators, or nested FOREACH statements.

ILLUSTRATE accesses the ExampleGenerator algorithm which can select an appropriate and concise set of example data automatically. It does a better job than random sampling would do; for example, random sampling suffers from the drawback that selective operations such as filters or joins can eliminate all the sampled data, giving you empty results which will not help with debugging.

With the ILLUSTRATE operator you can test your programs on small datasets and get faster turnaround times. The ExampleGenerator algorithm uses Pig's Local mode (rather than Hadoop mode) which means that illustrative example data is generated in near real-time.

Relation X can be used with the ILLUSTRATE operator.

```
X = FOREACH A GENERATE f1;

ILLUSTRATE X;
```

Relation Y cannot be used with the ILLUSTRATE operator.

```
Y = LIMIT A 3;
```

```
ILLUSTRATE Y;
```

27.4. Example

In this example we count the number of sites a user has visited since 12/1/08. The ILLUSTRATE statement will show how the results for num_user_visits are derived.

```
visits = LOAD 'visits' AS (user:chararray, ulr:chararray, timestamp:chararray);
DUMP visits;
(Amy,cnn.com,20080218)
(Fred,harvard.edu,20081204)
(Amy,bbc.com,20081205)
(Fred,stanford.edu,20081206)

recent_visits = FILTER visits BY timestamp >= '20081201';
user_visits = GROUP recent_visits BY user;
num_user_visits = FOREACH user_visits GENERATE COUNT(recent_visits);
DUMP num_user_visits;
(1L)
(2L)

ILLUSTRATE num_user_visits;
-------------------------------------------------------------------------
| visits     | user: bytearray | ulr: bytearray | timestamp: bytearray |
-------------------------------------------------------------------------
|          | Amy            | cnn.com        | 20080218            |
|          | Fred           | harvard.edu    | 20081204            |
|          | Amy            | bbc.com        | 20081205            |
|          | Fred           | stanford.edu   | 20081206            |
-------------------------------------------------------------------------
-------------------------------------------------------------------------
```

```
| visits     | user: chararray | ulr: chararray | timestamp: chararray |
-------------------------------------------------------------------------
|            | Amy             | cnn.com        | 20080218             |
|            | Fred            | harvard.edu    | 20081204             |
|            | Amy             | bbc.com        | 20081205             |
|            | Fred            | stanford.edu   | 20081206             |
-------------------------------------------------------------------------

-------------------------------------------------------------------------------
| recent_visits     | user: chararray | ulr: chararray | timestamp: chararray |
-------------------------------------------------------------------------------
|                   | Fred            | harvard.edu    | 20081204             |
|                   | Amy             | bbc.com        | 20081205             |
|                   | Fred            | stanford.edu   | 20081206             |
-------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------------------
| user_visits     | group: chararray | recent_visits: bag({user: chararray,ulr: chararray,timestamp: chararray}) |
-----------------------------------------------------------------------------------------------------------------
|                 | Amy              | {(Amy, bbc.com, 20081205)}                                               |
|                 | Fred             | {(Fred, harvard.edu, 20081204), (Fred, stanford.edu, 20081206)}          |
-----------------------------------------------------------------------------------------------------------------

------------------------------
| num_user_visits     | long  |
------------------------------
|                     | 1     |
|                     | 2     |
------------------------------
```

28. DEFINE

Assigns an alias to a function or command.

28.1. Syntax

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

28.2. Terms

alias	The name for the function or command.
function	The name of a function.
`command `	A command, including the arguments, enclosed in back tics (where a command is anything that can be executed).
input	INPUT ({stdin | 'path'} [USING serializer] [, {stdin | 'path'} [USING serializer] …]) Where: 1. INPUT – Keyword. 2. 'path' – A file path, enclosed in single quotes. 3. USING – Keyword. 4. serializer – A function that converts data from tuples to stream format. PigStorage is the default serializer. You can also write your own UDF.
output	OUTPUT ({stdout | stderr | 'path'} [USING deserializer] [, {stdout | stderr | 'path'} [USING deserializer] …]) Where: 1. OUTPUT – Keyword. 2. 'path' – A file path, enclosed in single quotes. 3. USING – Keyword. 4. deserializer – A function that converts data from stream format to tuples. PigStorage is the default deserializer. You can also write your own UDF.
ship	SHIP('path' [, 'path' …])

	Where: 1. SHIP – Keyword. 2. 'path' – A file path, enclosed in single quotes.
cache	CACHE('dfs_path#dfs_file' [, 'dfs_path#dfs_file' …]) Where: 1. CACHE – Keyword. 2. 'dfs_path#dfs_file' – A file path/file name on the distributed file system, enclosed in single quotes. Example: '/mydir/mydata.txt#mydata.txt'

28.3. Usage

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

Use DEFINE to specify a function when:

1. The function has a log package name that you don't want to include in a script, especially if you call the function several times in that script.
2. 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 command when the streaming command specification is complex or requires additional parameters (input, output, and so on).

28.3.1. 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.

PigStorage, the default serialization/deserialization function, converts tuples to tab-delimited lines. Pig's BinarySerializer and BinaryDeserializer functions treat the entire file as a byte stream (no formatting or interpretation takes place). You can also write your own serialization/deserialization functions.

28.3.2. 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 cluster.The only guarantee is that the shipped files are available is 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.

28.3.3. 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.

28.4. Example: Input/Output

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

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

28.5. Example: 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;
```

28.6. Example: 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);
```

In this example a command is defined for use with the STREAM operator.

```
A = LOAD 'data';
DEFINE cmd `stream_cmd –input file.dat`
B = STREAM A through cmd.
```

29. REGISTER

Registers a JAR file so that the UDFs in the file can be used.

29.1. Syntax

```
REGISTER alias;
```

29.2. Terms

alias	The path of a Java JAR file. Do not place the name in quotes.

29.3. Usage

Use the REGISTER statement to specify the path of a Java JAR file containing UDFs.

For more information about UDFs, see the User Defined Function Guide. Note that Pig currently only supports functions written in Java.

29.4. Example

In this example REGISTER states that myfunc.jar is located in the /src directory.

```
/src $ java -jar pig.jar –

REGISTER /src/myfunc.jar;

A = LOAD 'students';

B = FOREACH A GENERATE myfunc.MyEvalFunc($0);
```

30. Eval Functions

30.1. AVG

Computes the average of the numeric values in a single-column bag.

30.1.1. Syntax

```
AVG(expression)
```

30.1.2. Terms

expression	Any expression whose result is a bag. The elements of the bag should be data type int, long, float, or double.

30.1.3. Usage

Use the AVG function to compute the average of the numeric values in a single-column bag. AVG requires a preceding GROUP ALL statement for global averages and a GROUP BY statement for group averages.

30.1.4. Example

In this example the average GPA for each student is computed (see the GROUP operators for information about the field names in relation B).

```
A = LOAD 'student.txt' AS (name:chararray, term:chararray, gpa:float);

DUMP A;

(John,fl,3.9F)

(John,wt,3.7F)

(John,sp,4.0F)

(John,sm,3.8F)

(Mary,fl,3.8F)

(Mary,wt,3.9F)

(Mary,sp,4.0F)

(Mary,sm,4.0F)

B = GROUP A BY name;

DUMP B;

(John,{(John,fl,3.9F),(John,wt,3.7F),(John,sp,4.0F),(John,sm,3.8F)})

(Mary,{(Mary,fl,3.8F),(Mary,wt,3.9F),(Mary,sp,4.0F),(Mary,sm,4.0F)})
```

```
C = FOREACH B GENERATE A.name, AVG(A.gpa);
DUMP C;
({(John),(John),(John),(John)},3.850000023841858)
({(Mary),(Mary),(Mary),(Mary)},3.925000011920929)
```

30.1.5. Types Tables

	int	long	float	double	chararray	bytearray
AVG	long	long	double	double	error	cast as double

30.2. CONCAT

Concatenates two fields of type chararray or two fields of type bytearray.

30.2.1. Syntax

```
CONCAT (expression, expression)
```

30.2.2. Terms

expression	An expression with data types chararray or bytearray.

30.2.3. Usage

Use the CONCAT function to concatenate two elements. The data type of the two elements must be the same, either chararray or bytearray.

30.2.4. Example

In this example fields f2 and f3 are concatenated.

```
A = LOAD 'data' as (f1:chararray, f2:chararray, f3:chararray);
DUMP A;
(apache,open,source)
(hadoop,map,reduce)
```

```
(pig,pig,latin)

X = FOREACH A GENERATE CONCAT(f2,f3);
DUMP X;
(opensource)
(mapreduce)
(piglatin)
```

30.2.5. Types Tables

	chararray	bytearray
chararray	chararray	cast as chararray
bytearray		bytearray

30.3. COUNT

Computes the number of elements in a bag. COUNT requires a preceding GROUP ALL statement for global counts and a GROUP BY statement for group counts.

30.3.1. Syntax

```
COUNT(expression)
```

30.3.2. Terms

expression	An expression with data type bag.

30.3.3. Usage

Use the COUNT function to compute the number of elements in a bag.

30.3.4. Example

In this example the tuples in the bag are counted (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)})

X = FOREACH B GENERATE COUNT(A);
DUMP X;
(1L)
(2L)
(1L)
(2L)
```

30.3.5. Types Tables

	int	long	float	double	chararray	bytearray
COUNT	long	long	long	long	long	long

30.4. DIFF

Compares two fields in a tuple.

30.4.1. Syntax

DIFF (expression, expression)

30.4.2. Terms

expression	An expression with any data type.

30.4.3. Usage

The DIFF function compares two fields in a tuple. If the field values match, null is returned. If the field values do not match, the non-matching elements are returned.

30.4.4. Example

In this example the two fields are bags. DIFF compares the tuples in each bag.

```
A = LOAD 'bag_data' AS (B1:bag{T1:tuple(t1:int,t2:int)},B2:bag{T2:tuple(f1:int,f2:int)});
DUMP A;
({(8,9),(0,1)},{(8,9),(1,1)})
({(2,3),(4,5)},{(2,3),(4,5)})
({(6,7),(3,7)},{(2,2),(3,7)})
DESCRIBE A;
a: {B1: {T1: (t1: int,t2: int)},B2: {T2: (f1: int,f2: int)}}

X = FOREACH A DIFF(B1,B2);
grunt> dump x;
({(0,1),(1,1)})
({})
({(6,7),(2,2)})
```

30.5. MAX

Computes the maximum of the numeric values or chararrays in a single-column bag. MAX requires a preceding GROUP ALL statement for global maximums and a GROUP BY

statement for group maximums.

30.5.1. Syntax

MAX(expression)

30.5.2. Terms

expression	An expression with data types int, long, float, double, or chararray.

30.5.3. Usage

Use the MAX function to compute the maximum of the numeric values or chararrays in a single-column bag.

30.5.4. Example

In this example the maximum GPA for all terms is computed for each student (see the GROUP operator for information about the field names in relation B).

```
A = LOAD 'student' AS (name:chararray, session:chararray, gpa:float);

DUMP A;

(John,fl,3.9F)

(John,wt,3.7F)

(John,sp,4.0F)

(John,sm,3.8F)

(Mary,fl,3.8F)

(Mary,wt,3.9F)

(Mary,sp,4.0F)

(Mary,sm,4.0F)


B = GROUP A BY name;

DUMP B;

(John,{(John,fl,3.9F),(John,wt,3.7F),(John,sp,4.0F),(John,sm,3.8F)})
```

```
(Mary,{(Mary,fl,3.8F),(Mary,wt,3.9F),(Mary,sp,4.0F),(Mary,sm,4.0F)})

X = FOREACH B GENERATE group, MAX(A.gpa);
DUMP X;
(John,4.0F)
(Mary,4.0F)
```

30.5.5. Types Tables

	int	long	float	double	chararray	bytearray
MAX	int	long	float	double	chararray	cast as double

30.6. MIN

Computes the minimum of the numeric values or chararrays in a single-column bag. MIN requires a preceding GROUP… ALL statement for global minimums and a GROUP … BY statement for group minimums.

30.6.1. Syntax

```
MIN(expression)
```

30.6.2. Terms

expression	An expression with data types int, long, float, double, or chararray.

30.6.3. Usage

Use the MIN function to compute the minimum of a set of numeric values or chararrays in a single-column bag.

30.6.4. Example

In this example the minimum GPA for all terms is computed for each student (see the GROUP operator for information about the field names in relation B).

```
A = LOAD 'student' AS (name:chararray, session:chararray, gpa:float);
DUMP A;
(John,fl,3.9F)
(John,wt,3.7F)
(John,sp,4.0F)
(John,sm,3.8F)
(Mary,fl,3.8F)
(Mary,wt,3.9F)
(Mary,sp,4.0F)
(Mary,sm,4.0F)

B = GROUP A BY name;
DUMP B;
(John,{(John,fl,3.9F),(John,wt,3.7F),(John,sp,4.0F),(John,sm,3.8F)})
(Mary,{(Mary,fl,3.8F),(Mary,wt,3.9F),(Mary,sp,4.0F),(Mary,sm,4.0F)})

X = FOREACH B GENERATE group, MIN(A.gpa);
DUMP X;
(John,3.7F)
(Mary,3.8F)
```

30.6.5. Types Tables

	int	long	float	double	chararray	bytearray
MIN	int	long	float	double	chararray	cast as double

30.7. SIZE

Computes the number of elements based on the data type.

30.7.1. Syntax

SIZE(expression)

30.7.2. Terms

expression	An expression with any data type.

30.7.3. Usage

Use the SIZE function to compute the number of elements based on the data type (see the Types Tables below).

30.7.4. Example

In this example the number of characters in the first field is computed.

```
A = LOAD 'data' as (f1:chararray, f2:chararray, f3:chararray);
(apache,open,source)
(hadoop,map,reduce)
(pig,pig,latin)

X = FOREACH A GENERATE SIZE(f1);
DUMP X;
(6L)
(6L)
(3L)
```

30.7.5. Types Tables

int	returns 1
long	returns 1
float	returns 1

double	returns 1
chararray	returns number of characters in the array
bytearray	returns number of bytes in the array
tuple	returns number of fields in the tuple
bag	returns number of tuples in bag
map	returns number of key/value pairs in map

30.8. SUM

Computes the sum of the numeric values in a single-column bag. SUM requires a preceding GROUP ALL statement for global sums and a GROUP BY statement for group sums.

30.8.1. Syntax

SUM(expression)

30.8.2. Terms

expression	An expression with data types int, long, float, double, or bytearray cast as double.

30.8.3. Usage

Use the SUM function to compute the sum of a set of numeric values in a single-column bag.

30.8.4. Example

In this example the number of pets is computed. (see the GROUP operator for information about the field names in relation B).

```
A = LOAD 'data' AS (owner:chararray, pet_type:chararray, pet_num:int);

DUMP A;

(Alice,turtle,1)
```

```
(Alice,goldfish,5)
(Alice,cat,2)
(Bob,dog,2)
(Bob,cat,2)

B = GROUP A BY owner;
DUMP B;
(Alice,{(Alice,turtle,1),(Alice,goldfish,5),(Alice,cat,2)})
(Bob,{(Bob,dog,2),(Bob,cat,2)})

X = FOREACH B GENERATE group, SUM(A.pet_num);
DUMP X;
(Alice,8L)
(Bob,4L)
```

30.8.5. Types Tables

	int	long	float	double	chararray	bytearray
SUM	long	long	double	double	error	cast as double

30.9. TOKENIZE

Splits a string and outputs a bag of words.

30.9.1. Syntax

```
TOKENIZE(expression)
```

30.9.2. Terms

expression	An expression with data type chararray.

30.9.3. Usage

Use the TOKENIZE function to split a string of words (all words in a single tuple) into a bag of words (each word in a single tuple). The following characters are considered to be word separators: space, double quote("), coma(,) parenthesis(()), star(*).

30.9.4. Example

In this example the strings in each row are split.

```
A  = LOAD 'data' AS (f1:chararray);
DUMP A;
(Here is the first string.)
(Here is the second string.)
(Here is the third string.)

X = FOREACH A GENERATE TOKENIZE(f1);
DUMP X;
({(Here),(is),(the),(first),(string.)})
({(Here),(is),(the),(second),(string.)})
({(Here),(is),(the),(third),(string.)})
```

31. Load/Store Functions

Load/Store functions determine how data goes into Pig and comes out of Pig. In addition to the Pig built-in load/store functions, you can also write your functions (see the User-Defined Function Manual).

31.1. BinarySerializer

Converts a file to a byte stream.

31.1.1. Syntax

```
BinarySerializer()
```

31.1.2. Terms

none	no parameters

31.1.3. Usage

Use the BinarySerializer with the DEFINE operator to convert a file to a byte stream. No Formatting or interpretation takes place.

31.1.4. Example

In this example the BinarySerializer and BinaryDeserializer are use to convert data to and from streaming format.

```
DEFINE Y `stream.pl` INPUT(stdin USING BinarySerializer()) OUTPUT (stdout USING BinaryDeserializer());

X = STREAM A THROUGH Y;
```

31.2. BinaryDeserializer

Converts a byte stream into a file.

31.2.1. Syntax

```
BinarySerializer()
```

31.2.2. Terms

none	no parameters

31.2.3. Usage

Use the BinaryDeserializer with the DEFINE operator to convert a byte stream into a file. No Formatting or interpretation takes place.

31.2.4. Example

In this example the BinarySerializer and BinaryDeserializer are use to convert data to and from streaming format.

```
DEFINE Y `stream.pl` INPUT(stdin USING BinarySerializer()) OUTPUT (stdout USING
BinaryDeserializer());

X = STREAM A THROUGH Y;
```

31.3. BinStorage

Loads and stores data in machine-readable format.

31.3.1. Syntax

```
BinStorage()
```

31.3.2. Terms

none	no parameters

31.3.3. Usage

BinStorage works with data that is represented on disk in machine-readable format.

BinStorage is used internally by Pig to store the temporary data that is created between multiple map/reduce jobs.

31.3.4. Example

In this example BinStorage is used with the LOAD and STORE functions.

```
A = LOAD 'data' USING BinStorage();

STORE X into 'output' USING BinStorage();
```

31.4. PigStorage

Loads and stores data in UTF-8 format.

31.4.1. Syntax

```
PigStorage(field_delimiter)
```

31.4.2. Terms

field_delimiter	Parameter. The default field delimiter is tab ('\t'). You can specify other characters as field delimiters.

31.4.3. Usage

PigStorage works with structured text files in human-readable UTF-8 format. PigStorage also works with simple and complex data types and is the default function for the LOAD and STORE operators.

1. For load statements, PigStorage expects data to be formatted as delimiter-separated fields and newline-separated records.
2. For store statements, PigStorage outputs data as delimiter-separated fields and newline-separated records.

For both load and store statements the default field delimiter is the tab character ('\t'). You can use other characters as field delimiters, but separators such as ^A or Ctrl-A should be represented in Unicode (\u0001) using UTF-16 encoding (see Wikipedia ASCII, Unicode, and UTF-16).

31.4.4. Example

In this example PigStorage expects input.txt to contain tab-separated fields and newline-separated records. The statements are equivalent.

```
A = LOAD 'student' USING PigStorage('\t') AS (name: chararray, age:int, gpa: float);

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

In this example PigStorage stores the contents of X into files with fields that are delimited with an asterisk (*). The STORE function specifies that the files will be located in a directory named output and that the files will be named part-nnnnn (for example, part-00000).

```
STORE X INTO  'output' USING PigStorage('*');
```

31.5. PigDump

Stores data in UTF-8 format.

31.5.1. Syntax

```
PigDump()
```

31.5.2. Terms

none	no parameters

31.5.3. Usage

PigDump stores data as tuples in human-readable UTF-8 format.

31.5.4. Example

In this example PigDump is used with the STORE function.

```
STORE X INTO 'output' USING PigDump();
```

31.6. TextLoader

Loads unstructured data in UTF-8 format.

31.6.1. Syntax

```
TextLoader()
```

31.6.2. Terms

none	no parameters

31.6.3. Usage

TextLoader works with unstructured data in UTF8 format. Each resulting tuple contains a single field with one line of input text. TextLoader cannot be used to store data.

31.6.4. Example

In this example TextLoader is used with the LOAD function.

```
A = LOAD 'data' USING TextLoader();
```

32. cat

Prints the content of one or more files to the screen.

32.1. Syntax

```
cat path [ path …]
```

32.2. Terms

path	The location of a file or directory.

32.3. Usage

The cat command is similar to the Unix cat command. If multiple files are specified, content from all files is concatenated together. If multiple directories are specified, content from all files in all directories is concatenated together.

32.4. Example

In this example the students file in the data directory is printed.

```
grunt> cat data/students
joe smith
john adams
anne white
grunt>
```

33. cd

Changes the current directory to another directory.

33.1. Syntax

```
cd [dir]
```

33.2. Terms

dir	The name of the directory you want to navigate to.

33.3. Usage

The cd command is similar to the Unix cd command and can be used to navigate the file system. If a directory is specified, this directory is made your current working directory and all other operations happen relatively to this directory. If no directory is specified, your home directory (/user/NAME) becomes the current working directory.

33.4. Example

In this example we move to the /data directory.

```
grunt> cd /data
```

34. copyFromLocal

Copies a file or directory from the local file system to HDFS.

34.1. Syntax

```
copyFromLocal src_path dst_path
```

34.2. Terms

src_path	The path on the local file system for a file or directory
dst_path	The path on HDFS.

34.3. Usage

The copyFromLocal command enables you to copy a file or a director from the local file system to the Hadoop Distributed File System (HDFS). If a directory is specified, it is recursively copied over. Dot "." can be used to specify that the new file/directory should be created in the current working directory and retain the name of the source file/directory.

34.4. Example

In this example a file (students) and a directory (/data/tests) are copied from the local file system to HDFS.

```
grunt> copyFromLocal /data/students students
grunt> ls students
/data/students <r 3> 8270
grunt>  copyFromLocal  /data/tests new_tests
grunt> ls new_test
/data/new_test/test1.data<r 3>   664
/data/new_test/test2.data<r 3>    344
/data/new_test/more_data
```

35. copyToLocal

Copies a file or directory from HDFS to a local file system.

35.1. Syntax

```
copyToLocal src_path dst_path
```

35.2. Terms

src_path	The path on HDFS.
dst_path	The path on the local file system for a file or directory.

35.3. Usage

The copyToLocal command enables you to copy a file or a director from Hadoop Distributed File System (HDFS) to a local file system. If a directory is specified, it is recursively copied over. Dot "." can be used to specify that the new file/directory should be created in the current working directory (directory from which the script was executed or grunt shell started) and retain the name of the source file/directory.

35.4. Example

In this example two files are copied from HDFS to the local file system.

```
grunt> copyToLocal students /data
grunt> copyToLocal data /data/mydata
```

36. cp

Copies a file or directory within HDFS.

36.1. Syntax

```
cp src_path dst_path
```

36.2. Terms

src_path	The path on HDFS.
dst_path	The path on HDFS.

36.3. Usage

The cp command is similar to the Unix cp command and enables you to copy files or directories within DFS. If a directory is specified, it is recursively copied over. Dot "." can be used to specify that the new file/directory should be created in the current working directory and retain the name of the source file/directory.

36.4. Example

In this example a file (students) is copied to another file (students_save).

```
grunt> cp students students_save
```

37. exec

Run a Pig script.

37.1. Syntax

```
run script
```

37.2. Terms

script	The name of a Pig script.

37.3. Usage

Use the exec command to run a Pig script with no interaction between the script and the Grunt shell. Aliases defined in the script are not available to the shell; however, the files produced as the output of the script and stored on the system are visible after the script is run. Aliases defined via the shell are not available to the script.

For comparison, see the run command. Both the exec and run commands are useful for debugging because you can modify a Pig script in an editor and then rerun the script in the Grunt shell without leaving the shell. Also, both commands promote Pig script modularity as they allow you to reuse existing components.

The exec command supports parameter substitution.

37.4. Example

In this example the script is displayed and run.

```
grunt> cat myscript.pig
a = LOAD 'student' AS (name, age, gpa);
b = LIMIT a 3;
DUMP b;
grunt> exec myscript.pig
(alice,20,2.47)
(luke,18,4.00)
(holly,24,3.27)
```

In this example parameter substitution is used with the exec command.

```
grunt> cat myscript.pig
a = LOAD 'student' AS (name, age, gpa);
b = ORDER a BY name;
```

```
STORE b into '$out';
grunt> exec –param out=myoutput myscript.pig
```

38. ls

Lists the contents of a directory.

38.1. Syntax

```
ls [path]
```

38.2. Terms

path	The name of the path/directory.

38.3. Usage

The ls command is similar to the Unix ls command and enables you to list the contents of a directory. If DIR is specified, the command lists the content of the specified directory. Otherwise, the content of the current working directory is listed.

38.4. Example

In this example the contents of the data directory are listed.

```
grunt> ls /data
/data/DDLs  <dir>
/data/count <dir>
/data/data  <dir>
/data/schema <dir>
```

39. mkdir

Creates a new directory.

39.1. Syntax

```
mkdir path
```

39.2. Terms

path	The name of the path/directory.

39.3. Usage

The mkdir command is similar to the Unix mkdir command and enables you to create a new directory. If you specify a directory or path that does not exist, it will be created.

39.4. Example

In this example a directory and subdirectory are created.

```
grunt> mkdir data/20070905
```

40. mv

Moves a file or directory within the Hadoop Distributed File System (HDFS).

40.1. Syntax

```
mv src_path dst_path
```

40.2. Terms

src_path	The path on HDFS.
dst_path	The path on HDFS.

40.3. Usage

The mv command is identical to the Unix mv command (which copies files or directories within DFS) except that it deletes the source file or directory as soon as it is copied.

If a directory is specified, it is recursively moved. Dot "." can be used to specify that the new file/directory should be created in the current working directory and retain the name of the source file/directory.

40.4. Example

In this example the output directory is copied to output2 and then deleted.

```
grunt> mv output output2
grunt> ls output
File or directory output does not exist.
grunt> ls output2
/data/output2/map-000000<r 3>     508844
/data/output2/output3     <dir>
/data/output2/part-00000<r 3>     0
```

41. pwd

Prints the name of the current working directory.

41.1. Syntax

```
pwd
```

41.2. Terms

none	no parameters

41.3. Usage

The pwd command is identical to Unix pwd command and it prints the name of the current working directory.

41.4. Example

In this example the name of the current working directory is /data.

```
grunt> pwd
/data
```

42. rm

Removes one or more files or directories.

42.1. Syntax

rm path [path…]

42.2. Terms

path	The name of the path/directory/file.

42.3. Usage

The rm command is similar to the Unix rm command and enables you to remove one or more files or directories.

Note: This command recursively removes a directory even if it is not empty and it does not confirm remove and the removed data is not recoverable.

42.4. Example

In this example files are removed.

```
grunt> rm /data/students
grunt> rm students students_sav
```

43. rmf

Forcibly removes one or more files or directories.

43.1. Syntax

rmf path [path …]

43.2. Terms

path	The name of the path/directory/file.

43.3. Usage

The rmf command is similar to the Unix rm -f command and enables you to forcibly remove one or more files or directories.

Note: This command recursively removes a directory even if it is not empty and it does not confirm remove and the removed data is not recoverable.

43.4. Example

In this example files are forcibly removed.

```
grunt> rmf /data/students
grunt> rmf students students_sav
```

44. run

Run a Pig script.

44.1. Syntax

```
run script
```

44.2. Terms

script	The name of a Pig script.

44.3. Usage

Use the run command to run a Pig script that can interact with the Grunt shell. The script has access to aliases defined externally via the Grunt shell. The Grunt shell has access to aliases defined within the script. All commands from the script are visible in the command history.

For comparison, see the exec command. Both the run and exec commands are useful for debugging because you can modify a Pig script in an editor and then rerun the script in the Grunt shell without leaving the shell. Also, both commands promote Pig script modularity as they allow you to reuse existing components.

The run command supports parameter substitution.

44.4. Example

In this example the script interacts with the results of commands issued via the Grunt shell.

```
grunt> cat myscript.pig
b = ORDER a BY name;
c = LIMIT b 10;
grunt> a = LOAD 'student' AS (name, age, gpa);
grunt> run myscript.pig
grunt> d = LIMIT c 3;
grunt> DUMP d;
(alice,20,2.47)
(alice,27,1.95)
(alice,36,2.27)
```

In this example parameter substitution is used with the run command.

```
grunt> a = LOAD 'student' AS (name, age, gpa);
grunt> cat myscript.pig
b = ORDER a BY name;
STORE b into '$out';
grunt> run –param out=myoutput myscript.pig
```

45. Utility Commands

45.1. help

Prints a list of Pig commands.

45.1.1. Syntax

```
help
```

45.1.2. Terms

none	no parameters

45.1.3. Usage

The help command prints a list of Pig commands.

45.1.4. Example

In this example the students file in the data directory is printed out.

```
grunt> help
Commands:
<pig latin statement>;
store <alias> into <filename> [using <functionSpec>]
dump <alias>
etc ….
```

45.2. kill

Kills a job.

45.2.1. Syntax

```
kill jobid
```

45.2.2. Terms

jobid	The job id.

45.2.3. Usage

The kill command enables you to kill a job based on a job id.

45.2.4. Example

In this example the job with id job_0001 is killed.

```
grunt> kill job_0001
```

45.3. quit

Quits from the Pig grunt shell.

45.3.1. Syntax

```
exit
```

45.3.2. Terms

none	no parameters

45.3.3. Usage

The quit command enables you to quit or exit the Pig grunt shell.

45.3.4. Example

In this example the quit command exits the Pig grunt shall.

```
grunt> quit
```

45.4. set

Assigns values to keys used in Pig.

45.4.1. Syntax

```
set key 'value'
```

45.4.2. Terms

key	Key (see table). Case sensitive.
value	Value for key (see table). Case sensitive.

45.4.3. Usage

The set command enables you to assign values to keys, as shown here:

Key	Value	Description
debug	on/off	enables/disables debug-level logging
job.name	single quoted string that contains the name	sets user-specified name for the job

45.4.4. Example

In this example debug is set on and the job is assigned a name.

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