Pig Latin Reference Manual 2

by

Table of contents

1. Overview

Use this manual together with Pig Latin Reference Manual 1.

Also, be sure to review the information in the Pig Cookbook.

1.1. Conventions

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

Convention	Description	Example
( )	Parentheses enclose one or more items. Parentheses are also used to indicate the tuple data type.	Multiple items: (1, abc, (2,4,6))
[ ]	Straight brackets enclose one or more optional items. Straight brackets are also used to indicate the map data type. In this case <> is used to indicate optional items.	Optional items: [INNER | OUTER]
{ }	Curly brackets enclose two or more items, one of which is required. Curly brackets also used to indicate the bag data type. In this case <> is used to indicate required items.	Two items, one required: { 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	Pig Latin statement: A = LOAD 'data' AS (f1:int); • LOAD, AS supplied BY

	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 insensitive.	system • A, f1 are names (aliases) • data supplied by you
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.

1.2. Reserved Keywords

Pig reserved keywords are listed here.

-- A	and, any, all, arrange, as, asc, AVG
-- B	bag, BinStorage, by, bytearray
-- C	cache, cat, cd, chararray, cogroup, CONCAT, copyFromLocal, copyToLocal, COUNT, cp, cross
-- D	%declare, %default, define, desc, describe, DIFF, distinct, double, du, dump
-- E	e, E, eval, exec, explain
-- F	f, F, filter, flatten, float, foreach, full
-- G	generate, group
-- H	help
-- I	if, illustrate, inner, input, int, into, is
-- J	join

-- K	kill
-- L	l, L, left, limit, load, long, ls
-- M	map, matches, MAX, MIN, mkdir, mv
-- N	not, null
-- O	onschema, or, order, outer, output
-- P	parallel, pig, PigDump, PigStorage, pwd
-- Q	quit
-- R	register, right, rm, rmf, run
-- S	sample, set, ship, SIZE, split, stderr, stdin, stdout, store, stream, SUM
-- T	TextLoader, TOKENIZE, through, tuple
-- U	union, using
-- V, W, X, Y, Z	
-- Symbols	= = != < > <= >= + - * / % ? $. # :: ( ) [ ] { }

2. Data Types and More

2.1. Relations, Bags, Tuples, Fields

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

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

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

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

2.1.1. 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)
```

2.1.2. Referencing Fields

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

- Positional notation is generated by the system. Positional notation is indicated with the dollar sign ($) and begins with zero (0); for example, $0, $1, $2.
- Names are assigned by you using schemas (or, in the case of the GROUP operator and some functions, by the system). You can use any name that is not a Pig keyword; 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 ...
```

2.1.3. 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.

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

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

```
cat data;
(3,8,9) (4,5,6)
```

```
(1,4,7) (3,7,5)
(2,5,8) (9,5,8)

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

DUMP A;
((3,8,9),(4,5,6))
((1,4,7),(3,7,5))
((2,5,8),(9,5,8))

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

DUMP X;
(3,4)
(1,3)
(2,9)
```

2.2. Data Types

2.2.1. Simple and Complex

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:

- Use schemas to assign types to fields. If you don't assign types, fields default to type bytearray and implicit conversions are applied to the data depending on the context in which that data is used. For example, in relation B, f1 is converted to integer because 5 is integer. In relation C, f1 and f2 are converted to double because we don't know the type of either f1 or f2.

```
A = LOAD 'data' AS (f1,f2,f3);
B = FOREACH A GENERATE f1 + 5;
C = FOREACH A generate f1 + f2;
```

- If a schema is defined as part of a load statement, the load function will attempt to enforce the schema. If the data does not conform to the schema, the loader will generate a null value or an error.

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

- If an explicit cast is not supported, an error will occur. For example, you cannot cast a chararray to int.

```
A = LOAD 'data' AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE (int)name;

This will cause an error …
```

- If Pig cannot resolve incompatible types through implicit casts, an error will occur. For example, you cannot add chararray and float (see the Types Table for addition and subtraction).

```
A = LOAD 'data' AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE name + gpa;
```

```
This will cause an error …
```

2.2.2. Tuple

A tuple is an ordered set of fields.

2.2.2.1. Syntax

(field [, field …])

2.2.2.2. Terms

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

2.2.2.3. 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:

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

2.2.2.4. Example

In this example the tuple contains three fields.

```
(John,18,4.0F)
```

2.2.3. Bag

A bag is a collection of tuples.

2.2.3.1. Syntax: Inner bag

{ tuple [, tuple …] }

2.2.3.2. Terms

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

2.2.3.3. Usage

Note the following about bags:

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

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

2.2.3.4. 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)
```

2.2.3.5. 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)})
```

2.2.4. Map

A map is a set of key value pairs.

2.2.4.1. Syntax (<> denotes optional)

[key#value <, key#value …>]

2.2.4.2. Terms

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

2.2.4.3. Usage

Key values within a relation must be unique.

2.2.4.4. Example

In this example the map includes two key value pairs.

```
[name#John,phone#5551212]
```

2.3. 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.

2.3.1. Nulls, Operators, and Functions

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

Operator	Interaction
Comparison operators:	If either subexpression is null, the result is null.

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

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

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

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

2.3.2. 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;
```

2.3.3. Operations That Produce Nulls

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

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

2.3.3.1. 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)
```

2.3.4. 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;
```

2.3.5. Nulls and GROUP/COGROUP Operators

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

```
A = load 'student' as (name:chararray, age:int, gpa:float);
dump A;
(joe,18,2.5)
(sam,,3.0)
(bob,,3.5)

X = group A by age;
dump X;
(18,{(joe,18,2.5)})
(,{(sam,,3.0),(bob,,3.5)})
```

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

```
A = load 'student' as (name:chararray, age:int, gpa:float);
B = load 'student' as (name:chararray, age:int, gpa:float);
dump B;
(joe,18,2.5)
(sam,,3.0)
(bob,,3.5)

X = cogroup A by age, B by age;
dump X;
(18,{(joe,18,2.5)},{(joe,18,2.5)})
(,{(sam,,3.0),(bob,,3.5)},{})
(,{},{(sam,,3.0),(bob,,3.5)})
```

2.3.6. Nulls and JOIN Operator

The JOIN operator - when performing inner joins - adheres to the SQL standard and disregards (filters out) null values. (See also Drop Nulls Before a Join.)

```
A = load 'student' as (name:chararray, age:int, gpa:float);
B = load 'student' as (name:chararray, age:int, gpa:float);
dump B;
(joe,18,2.5)
(sam,,3.0)
```

```
(bob,,3.5)

X = join A by age, B by age;
dump X;
(joe,18,2.5,joe,18,2.5)
```

2.4. 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:

- On UTF-8 systems you can specify string constants consisting of printable ASCII characters such as 'abc'; you can specify control characters such as '\t'; and, you can specify a character in Unicode by starting it with '\u', for instance, '\u0001' represents Ctrl-A in hexadecimal (see Wikipedia ASCII, Unicode, and UTF-8). In theory, you should be able to specify non-UTF-8 constants on non-UTF-8 systems but as far as we know this has not been tested.
- To specify a long constant, l or L must be appended to the number (for example, 12345678L). If the l or L is not specified, but the number is too large to fit into an int, the problem will be detected at parse time and the processing is terminated.
- Any numeric constant with decimal point (for example, 1.5) and/or exponent (for example, 5e+1) is treated as double unless it ends with f or F in which case it is assigned type float (for example, 1.5f).

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

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

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

```
A = LOAD 'data' USING MyStorage() AS (T: tuple(name:chararray, age: int));
B = FILTER A BY T == ('john', 25);
D = FOREACH B GENERATE T.name, [25#5.6], {(1, 5, 18)};
```

2.5. Expressions

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

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

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

In Pig Latin,

- An arithmetic expression could look like this:

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

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

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

- A boolean expression could look like this:

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

2.5.1. Field expressions

Field expressions represent a field or a dereference operator applied to a field. See Dereference Operators for more details.

2.5.2. Star expression

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

```
A = LOAD 'data' USING MyStorage() AS (name:chararray, age: int);
B = FOREACH A GENERATE *, MyUDF(name, age);
C = FOREACH A GENERATE name, age, MyUDF(*);
```

A common error when using the star expression is the following:

```
G = GROUP A BY $0;
C = FOREACH G GENERATE COUNT(*)
```

In this example, the programmer really wants to count the number of elements in the bag in the second field: COUNT($1).

2.5.3. Boolean expressions

Boolean expressions can be made up of UDFs that return a boolean value or boolean operators (see Boolean Operators).

2.5.4. Tuple expressions

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

2.5.5. General expressions

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

2.6. 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 Pig UDF Manual for more information).

Note the following:

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

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

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

2.6.1. 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);
```

2.6.2. Schemas with FOREACH Statements

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

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

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

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

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

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

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

2.6.3. Schemas for Simple Data Types

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

2.6.3.1. Syntax

(alias[:type]) [, (alias[:type]) …])

2.6.3.2. 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.

2.6.3.3. 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)
```

2.6.4. Schemas for Complex Data Types

Complex data types include tuples, bags, and maps.

2.6.5. Tuple Schema

A tuple is an ordered set of fields.

2.6.5.1. Syntax

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

2.6.5.2. 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: • alias – the name assigned to the field • type (optional) – the simple or complex data type assigned to the field

2.6.5.3. 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))
```

2.6.6. Bag Schema

A bag is a collection of tuples.

2.6.6.1. Syntax

alias[:bag] {tuple}

2.6.6.2. 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).

2.6.6.3. 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)})
```

2.6.7. Map Schema

A map is a set of key value pairs.

2.6.7.1. Syntax (where <> means optional)

alias<:map> [ ]

2.6.7.2. 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 [ ].

2.6.7.3. 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])
```

2.6.8. Schemas for Multiple Types

You can define schemas for data that includes multiple types.

2.6.8.1. 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:[]
);
```

2.7. Parameter Substitution

2.7.1. Description

Substitute values for parameters at run time.

2.7.1.1. Syntax: Specifying parameters using the Pig command line

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

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

{%declare | %default} param_name param_value

2.7.1.3. Terms

pig	Keyword Note: exec, run, and explain also support parameter substitution.
–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: • 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. • 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. There are no hard limits on the size except that parameters need to fit into memory.
–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. • 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. • If parameters are specified using the preprocessor statements, the script should include either %declare or %default. • 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.

2.7.1.4. 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.

Specifying Parameters

You can specify parameter names and parameter values as follows:

- As part of a command line.
- In parameter file, as part of a command line.
- With the declare statement, as part of Pig script.
- With default statement, as part of a Pig script.

Precedence

Precedence for parameters is as follows:

- Highest - parameters defined using the declare statement
- Next - parameters defined in the command line
- Lowest - parameters defined in a script

Processing Order and Precedence

Parameters are processed as follows:

- Command line parameters are scanned in the order they are specified on the command line.
- 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.
- Declare and default preprocessors statements are processed in the order they appear in the Pig script.

2.7.1.5. Example: Specifying parameters in the command line

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

mydata

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

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)
```

2.7.1.6. 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
```

2.7.1.7. 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...
```

2.7.1.8. 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...
```

2.7.1.9. 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
```

2.7.1.10. 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...
```

3. Arithmetic Operators and More

3.1. Arithmetic Operators

3.1.1. Description

Operator	Symbol	Notes
addition	+	
subtraction	-	
multiplication	*	
division	/	
modulo	%	Returns the remainder of a divided by b (a%b). Works with integral numbers (int, long).
bincond	? :	(condition ? value_if_true : value_if_false) The bincond should be enclosed in parenthesis. The schemas for the two conditional outputs of the bincond should match. Use expressions only (relational operators are not allowed).

3.1.1.1. 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)})
```

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)
```

3.1.1.2. 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

3.1.1.3. 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

3.1.1.4. Types Table: modulo (%) operator

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

3.2. Comparison Operators

3.2.1. 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.

3.2.1.1. Example: numeric

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

3.2.1.2. Example: string

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

3.2.1.3. Example: matches

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

3.2.1.4. 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)

3.2.1.5.

	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

3.2.1.6. Types Table: matches operator

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

	chararray	bytearray*
chararray	boolean	boolean
bytearray	boolean	boolean

3.3. Null Operators

3.3.1. Description

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

3.3.1.1. Example

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

3.3.2. Types Table

The null operators can be applied to all data types (see Nulls).

3.4. Boolean Operators

3.4.1. 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).

3.4.1.1. Example

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

3.5. Dereference Operators

3.5.1. Description

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

3.5.1.1. Example: Tuple

Suppose we have relation A.

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

DUMP A;
(1,(1,2,3))
(2,(4,5,6))
(3,(7,8,9))
```

```
(4,(1,4,7))
(5,(2,5,8))
```

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

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

DUMP X;
(1,3)
(4,6)
(7,9)
(1,7)
(2,8)
```

3.5.1.2. 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}) |
-------------------------------------------------------------
```

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)})
```

3.5.1.3. Example: Tuple and Bag

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

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

DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)

B = GROUP A BY (f1,f2);

DUMP B;
((1,2),{(1,2,3)})
((4,2),{(4,2,1)})
((4,3),{(4,3,3)})
((7,2),{(7,2,5)})
((8,3),{(8,3,4)})
((8,4),{(8,4,3)})

ILLUSTRATE B;
etc …
-------------------------------------------------------------------------------
| b     | group: tuple({f1: int,f2: int}) | a: bag({f1: int,f2: int,f3:
int}) |
-------------------------------------------------------------------------------
|       | (8, 3)                                  | {(8, 3, 4), (8, 3, 4)} |
-------------------------------------------------------------------------------
```

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

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

DUMP X;
(1,{(1)})
(4,{(4)})
(4,{(4)})
(7,{(7)})
(8,{(8)})
(8,{(8)})
```

3.5.1.4. Example: Map

Suppose we have relation A.

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

DUMP A;
(1,[open#apache])
(2,[apache#hadoop])
(3,[hadoop#pig])
(4,[pig#grunt])
```

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

```
X = FOREACH A GENERATE f2#'open';

DUMP X;
(apache)
()
()
()
```

3.6. Sign Operators

3.6.1. Description

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

3.6.1.1. Example

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

B = FOREACH A GENERATE -x, y;
```

3.6.1.2. Types Table: negation (-) operator

bag	error
tuple	error
map	error
int	int

long	long
float	float
double	double
chararray	error
bytearray	double (as double)

3.7. Flatten Operator

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

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

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

Also note that the flatten of empty bag will result in that row being discarded; no output is generated. (See also Drop Nulls Before a Join.)

```
grunt> cat empty.bag
{}      1
grunt> A = LOAD 'empty.bag' AS (b : bag{}, i : int);
grunt> B = FOREACH A GENERATE flatten(b), i;
grunt> DUMP B;
grunt>
```

For examples using the FLATTEN operator, see FOREACH.

3.8. Cast Operators

3.8.1. 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	yes	yes	yes	yes		error
bytearray	yes	yes	yes	yes	yes	yes	yes	yes	

3.8.1.1. Syntax

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

3.8.1.2. 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.

3.8.1.3. 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:

- A field can be explicitly cast. Once cast, the field remains that type (it is not automatically cast back). In this example $0 is explicitly cast to int.

```
B = FOREACH A GENERATE (int)$0 + 1;
```

- Where possible, Pig performs implicit casts. In this example $0 is cast to int (regardless of underlying data) and $1 is cast to double.

```
B = FOREACH A GENERATE $0 + 1, $1 + 1.0
```

- When two bytearrays are used in arithmetic expressions or with built-in aggregate functions (such as SUM) they are implicitly cast to double. If the underlying data is really int or long, you'll get better performance by declaring the type or explicitly casting the data.
- Downcasts may cause loss of data. For example casting from long to int may drop bits.

3.8.1.4. 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])
```

3.9. Casting Relations to Scalars

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

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

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

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

```
A = load 'mydata' as (userid, clicks);
B = group A all;
C = foreach B genertate SUM(A.clicks) as total;
D = foreach A generate userid, clicks/(double)C.total;
dump D;
```

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

```
A = load 'mydata' as (userid, clicks);
B = group A all;
C = foreach B genertate SUM(A.clicks) as total, COUNT(A) as cnt;
D = FILTER A by clicks > C.total/3
E = foreach D generate userid, clicks/(double)C.total, cnt;
dump E;
```

4. Relational Operators

4.1. COGROUP

See the GROUP operator.

4.2. CROSS

Computes the cross product of two or more relations.

4.2.1. Syntax

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

4.2.2. Terms

alias	The name of a relation.
PARTITION BY partitioner	Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs. • For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoo • For usage, see Example: PARTITION BY
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. For more information, see Use the Parallel Features.

4.2.3. 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.

4.2.4. 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)
```

4.3. DISTINCT

Removes duplicate tuples in a relation.

4.3.1. Syntax

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

4.3.2. Terms

alias	The name of the relation.
PARTITION BY partitioner	Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs. • For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoo • For usage, see Example: PARTITION BY.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. For more information, see Use the Parallel Features.

4.3.3. 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 (see Example: Nested Block).

4.3.4. 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)
```

4.4. FILTER

Selects tuples from a relation based on some condition.

4.4.1. Syntax

alias = FILTER alias BY expression;

4.4.2. Terms

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

4.4.3. 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.

4.4.4. 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)
```

4.5. FOREACH

Generates data transformations based on columns of data.

4.5.1. Syntax

```
alias  = FOREACH { gen_blk | nested_gen_blk };
```

4.5.2. 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 [AS schema] [expression [AS schema]….]; See Schemas with FOREACH Statements

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 [AS schema] [expression [AS schema]….]
	};
	
	Where:
	
	The nested block is enclosed in opening and closing brackets { … }.
	
	The GENERATE keyword must be the last statement within the nested block.
	
	See Schemas with FOREACH Statements

expression	An expression.
nested_alias	The name of the inner bag.
nested_op	Allowed operations are DISTINCT, FILTER, LIMIT, and ORDER BY. The FOREACH…GENERATE operation itself is not allowed since this could lead to an arbitrary number of nesting levels. You can also perform projections (see Example: Nested Block).
AS	Keyword
schema	A schema using the AS keyword (see Schemas). • If the FLATTEN operator is used, enclose the

	schema in parentheses. • If the FLATTEN operator is not used, don't enclose the schema in parentheses.

4.5.3. 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:

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

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

- If A is an inner bag, a FOREACH statement could look like this.

```
X = FOREACH B {
        S = FILTER A BY 'xyz';
        GENERATE COUNT (S.$0);
}
```

4.5.4. 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)}
|
-------------------------------------------------------------------------------
```

4.5.5. Example: Projection

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

```
X = FOREACH A GENERATE *;

DUMP X;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)
```

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

```
X = FOREACH A GENERATE a1, a2;

DUMP X;
(1,2)
(4,2)
(8,3)
(4,3)
(7,2)
(8,4)
```

4.5.6. 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)})
```

4.5.7. 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)
```

4.5.8. 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)
```

4.5.9. Example: Flattening

In this example the FLATTEN operator is used to eliminate nesting.

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

DUMP X;
(1,1,2,3)
(4,4,2,1)
(4,4,3,3)
(8,8,3,4)
(8,8,4,3)
```

Another FLATTEN example.

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

DUMP X;
(1,3)
(4,1)
(4,3)
(8,4)
(8,3)
```

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

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

DUMP X;
(1,2,3)
(4,2,6)
(4,2,9)
(4,3,6)
(4,3,9)
(8,3,9)
(8,4,9)
```

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

```
A = LOAD 'data' AS (x, y);
B = LOAD 'data' AS (x, z);
C = COGROUP A BY x, B BY x;
D = FOREACH C GENERATE flatten(A), flatten(b);
E = GROUP D BY A::x;
……
```

4.5.10. Example: Nested Block

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

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

DUMP A;
(www.ccc.com,www.hjk.com)
(www.ddd.com,www.xyz.org)
(www.aaa.com,www.cvn.org)
(www.www.com,www.kpt.net)
(www.www.com,www.xyz.org)
(www.ddd.com,www.xyz.org)

B = GROUP A BY url;

DUMP B;
(www.aaa.com,{(www.aaa.com,www.cvn.org)})
(www.ccc.com,{(www.ccc.com,www.hjk.com)})
(www.ddd.com,{(www.ddd.com,www.xyz.org),(www.ddd.com,www.xyz.org)})
(www.www.com,{(www.www.com,www.kpt.net),(www.www.com,www.xyz.org)})
```

In this example we perform two of the operations allowed in a nested block, FILTER and DISTINCT. Note that the last statement in the nested block must be GENERATE. Also, note the use of projection (PA = FA.outlink;).

```
X = FOREACH B {
        FA= FILTER A BY outlink == 'www.xyz.org';
        PA = FA.outlink;
        DA = DISTINCT PA;
        GENERATE group, COUNT(DA);
}

DUMP X;
(www.aaa.com,0)
(www.ccc.com,0)
(www.ddd.com,1)
(www.www.com,1)
```

4.6. GROUP

Groups the data in one or more relations.

Note: The GROUP and COGROUP operators are identical. Both operators work with one or more relations. For readability GROUP is used in statements involving one relation and COGROUP is used in statements involving two or more relations.

4.6.1. Syntax

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

4.6.2. Terms

alias	The name of a relation.
ALL	Keyword. Use ALL if you want all tuples to go to a single group; for example, when doing aggregates across entire relations. B = GROUP A ALL;
BY	Keyword. Use this clause to group the relation by field, tuple or expression. B = GROUP A BY f1;
expression	A tuple expression. This is the group key or key field. If the result of the tuple expression is a single field, the key will be the value of the first field rather than a tuple with one field. To group using multiple keys, enclose the keys in parentheses: B = GROUP A BY (key1,key2);
USING	Keyword
'collected'	Use the ‘collected’ clause with the GROUP operation (works with one relation only). The following conditions apply: • The loader must implement the {CollectableLoader} interface. • Data must be sorted on the group key. If your data and loaders satisfy these conditions, use the ‘collected’ clause to perform an optimized version of GROUP; the operation will execute on the map side and avoid running the reduce phase. Note that the Zebra loader satisfies the conditions (see Zebra and Pig).
'merge'	Use the ‘merge’ clause with the COGROUP

	operation (works with two or more relations only). The following conditions apply: • No other operations can be done between the LOAD and COGROUP statements. • Data must be sorted on the cogroup key for all tables in ascending (ASC) order. • Nulls are considered smaller than evertyhing. If data contains null keys, they should occur before anything else. • Left-most loader must implement the {CollectableLoader} interface as well as {OrderedLoadFunc} interface. • All other loaders must implement IndexableLoadFunc. • Type information must be provided in the schema for all the loaders. If your data and loaders satisfy these conditions, the 'merge' clause to perform an optimized version of COGROUP; the operation will execute on the map side and avoid running the reduce phase. Note that the Zebra loader satisfies the conditions (see Zebra and Pig).
PARTITION BY partitioner	Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs. • For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hado • For usage, see Example: PARTITION BY
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. For more information, see Use the Parallel Features.

4.6.3. Usage

The GROUP operator groups together tuples that have the same group key (key field). The key field will be a tuple if the group key has more than one field, otherwise it will be the

same type as that of the group key. The result of a GROUP operation is a relation that includes one tuple per group. This tuple contains two fields:

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

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

- The GROUP and JOIN operators perform similar functions. GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples
- The GROUP/COGROUP and JOIN operators handle null values differently (see Nulls and GROUP/COGROUP Operators).

4.6.4. Example

Suppose we have relation A.

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

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

DUMP A;
(John,18,4.0F)
(Mary,19,3.8F)
(Bill,20,3.9F)
(Joe,18,3.8F)
```

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

```
B = GROUP A BY age;

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

ILLUSTRATE B;
etc …
-------------------------------------------------------------------------
| B     | group: int | A: bag({name: chararray,age: int,gpa: float}) |
-------------------------------------------------------------------------
```

```
|       | 18          | {(John, 18, 4.0), (Joe, 18, 3.8)}              |
|       | 20          | {(Bill, 20, 3.9)}                              |
-------------------------------------------------------------------------

DUMP B;
(18,{(John,18,4.0F),(Joe,18,3.8F)})
(19,{(Mary,19,3.8F)})
(20,{(Bill,20,3.9F)})
```

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

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

DUMP C;
(18,2L)
(19,1L)
(20,1L)

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

DUMP C;
(18,{(John),(Joe)})
(19,{(Mary)})
(20,{(Bill)})
```

4.6.5. 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)})
```

4.6.6. Example

Suppose we have two relations, A and B.

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

```
DUMP A;
(Alice,turtle)
(Alice,goldfish)
(Alice,cat)
(Bob,dog)
(Bob,cat)

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

DUMP B;
(Cindy,Alice)
(Mark,Alice)
(Paul,Bob)
(Paul,Jane)
```

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

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

DESCRIBE X;
X: {group: chararray,A: {owner: chararray,pet: chararray},b: {firend1:
chararray,friend2: chararray}}
```

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

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

4.6.7. Example

This example shows how to compute the number of tuples in an inner join between two relations.

```
A = LOAD …
B = LOAD …
C = COGROUP A BY f1 INNER, B BY f2 INNER;
D = FOREACH C GENERATE group, COUNT(A)*COUNT(B) AS count;   -- cross
product in each co-group
E = GROUP D ALL;
F = FOREACH E GENERATE SUM(D.count) AS sum;  -- sum of cross products
DUMP F;
```

4.6.8. Example

This example shows how to group using multiple keys.

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

4.6.9. Example

This example shows how to group using the collected keyword.

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

4.6.10. Example

This example shows how to cogroup using the merge keyword.

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

4.6.11. Example: PARTITION BY

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

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

Here is the code for SimpleCustomPartitioner:

```
public class SimpleCustomPartitioner extends Partitioner
<PigNullableWritable, Writable> {
     //@Override
    public int getPartition(PigNullableWritable key, Writable value, int
numPartitions) {
        if(key.getValueAsPigType() instanceof Integer) {
            int ret = (((Integer)key.getValueAsPigType()).intValue() %
numPartitions);
            return ret;
       }
       else {
            return (key.hashCode()) % numPartitions;
        }
    }
}
```

4.7. JOIN (inner)

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

4.7.1. Syntax

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

4.7.2. Terms

alias	The name of a relation.
BY	Keyword
expression	A field expression. Example: X = JOIN A BY fieldA, B BY fieldB, C BY fieldC;
USING	Keyword

'replicated'	Use to perform replicated joins (see Replicated Joins).
'skewed'	Use to perform skewed joins (see Skewed Joins).
'merge'	Use to perform merge joins (see Merge Joins).
PARTITION BY partitioner	Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs. • For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoo • For usage, see Example: PARTITION BY This feature CANNOT be used with skewed joins.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. For more information, see Use the Parallel Features.

4.7.3. Usage

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

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

- The GROUP and JOIN operators perform similar functions. GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples.
- The GROUP/COGROUP and JOIN operators handle null values differently (see Nulls and JOIN Operator).

4.7.4. 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)
```

4.8. JOIN (outer)

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

4.8.1. Syntax

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

4.8.2. Terms

alias	The name of a relation. Applies to alias, left-alias and right-alias.
alias-column	The name of the join column for the corresponding relation. Applies to left-alias-column and right-alias-column.

BY	Keyword
LEFT	Left outer join.
RIGHT	Right outer join.
FULL	Full outer join.
OUTER	(Optional) Keyword
USING	Keyword
'replicated'	Use to perform replicated joins (see Replicated Joins). Only left outer join is supported for replicated joins.
'skewed'	Use to perform skewed joins (see Skewed Joins).
'merge'	Use to perform merge joins (see Merge Joins).
PARTITION BY partitioner	Use this feature to specify the Hadoop Partitioner. The partitioner controls the partitioning of the keys of the intermediate map-outputs. • For more details, see http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hado • For usage, see Example: PARTITION BY This feature CANNOT be used with skewed joins.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. For more information, see Use the Parallel Features.

4.8.3. Usage

Use the JOIN operator with the corresponding keywords to perform left, right, or full outer joins. The keyword OUTER is optional for outer joins; the keywords LEFT, RIGHT and

FULL will imply left outer, right outer and full outer joins respectively when OUTER is omitted. The Pig Latin syntax closely adheres to the SQL standard.

Please note the following:

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

4.8.4. Examples

This example shows a left outer join.

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
C = JOIN A by $0 LEFT OUTER, B BY $0;
```

This example shows a full outer join.

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
C = JOIN A BY $0 FULL, B BY $0;
```

This example shows a replicated left outer join.

```
A = LOAD 'large';
B = LOAD 'tiny';
C= JOIN A BY $0 LEFT, B BY $0 USING 'replicated';
```

This example shows a skewed full outer join.

```
A = LOAD 'studenttab' as (name, age, gpa);
B = LOAD 'votertab' as (name, age, registration, contribution);
C = JOIN A BY name FULL, B BY name USING 'skewed';
```

4.9. LIMIT

Limits the number of output tuples.

4.9.1. Syntax

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

4.9.2. Terms

alias	The name of a relation.
n	The number of output tuples (a constant).

4.9.3. 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, all tuples in the relation are returned.

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.

4.9.4. 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)
```

4.10. LOAD

Loads data from the file system.

4.10.1. Syntax

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

4.10.2. 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 gobStatus for details on globing syntax).
USING	Keyword. If the USING clause is omitted, the default load function PigStorage is used.
function	The load function. • You can use a built-in function (see the Load/Store Functions). PigStorage is the default load function and does not need to be specified (simply omit the USING clause). • You can write your own load function if your data is in a format that cannot be processed by

	the built-in functions (see the Pig UDF Manual).
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.

4.10.3. Usage

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

4.10.4. 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.

4.11. MAPREDUCE

Executes native MapReduce jobs inside a Pig script.

4.11.1. Syntax

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

4.11.2. Terms

alias	The name of a relation.
mr.jar	The MapReduce jar file (enclosed in single quotes). You can specify nny MapReduce jar file that can be run through the "hadoop jar mymr.jar params" command. The values for inputLocation and outputLocation can be passed in the params.
STORE ... INTO ... USING	See STORE Store alias2 into the inputLocation using storeFunc,

	which is then used by native mapreduce to read its data.
LOAD ... USING ... AS	See LOAD After running mr.jar's mapreduce, load back the data from outputLocation into alias1 using loadFunc as schema
'params, ...'	Extra parameters required for native MapReduce job (enclosed in back tics).

4.11.3. Usage

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

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

4.11.4. Example

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

```
A = LOAD 'WordcountInput.txt';
B = MAPREDUCE 'wordcount.jar' STOE A INTO 'inputDir' LOAD 'outputDir'
    AS (word:chararray, count: int) `org.myorg.WordCount inputDir
outputDir`;
```

4.12. ORDER BY

Sorts a relation based on one or more fields.

4.12.1. Syntax

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

4.12.2. Terms

alias	The name of a relation.
*	The designator for a tuple.
field_alias	A field in the relation. The field must be a simple type.
ASC	Sort in ascending order.
DESC	Sort in descending order.
PARALLEL n	Increase the parallelism of a job by specifying the number of reduce tasks, n. For more information, see Use the Parallel Features.

4.12.3. Usage

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

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

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

```
A = LOAD 'mydata' AS (x: int, y: map[]);
B = ORDER A BY x; -- this is allowed because x is a simple type
B = ORDER A BY y; -- this is not allowed because y is a complex type
B = ORDER A BY y#'id'; -- this is not allowed because y#'id' is an
expression
```

4.12.4. 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)
```

4.13. SAMPLE

Partitions a relation into two or more relations.

4.13.1. Syntax

SAMPLE alias size;

4.13.2. Terms

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

4.13.3. Usage

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

4.13.4. Example

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

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
X = SAMPLE A 0.01;
```

4.14. SPLIT

Partitions a relation into two or more relations.

4.14.1. Syntax

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

4.14.2. Terms

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

4.14.3. 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:

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

4.14.4. 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)
```

4.14.5. Example

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

```
SPLIT input_var INTO output_var IF (field1 is not null), ignored_var IF
(field1 is null);
-- where ignored_var is not used elsewhere

output_var = FILTER input_var BY (field1 is not null);
```

4.15. STORE

Stores or saves results to the file system.

4.15.1. Syntax

STORE alias INTO 'directory' [USING function];

4.15.2. 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. • You can use a built-in function (see the Load/Store Functions). PigStorage is the default store function and does not need to be specified (simply omit the USING clause). • You can write your own store function if your data is in a format that cannot be processed by the built-in functions (see the Pig UDF Manual).

4.15.3. Usage

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

Note: To debug scripts during development, you can use DUMP to check intermediate results.

4.15.4. 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
```

4.16. STREAM

Sends data to an external script or program.

4.16.1. Syntax

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

4.16.2. 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 (see the DEFINE operator for additional streaming examples).
AS	Keyword.
schema	A schema using the AS keyword, enclosed in parentheses (see Schemas).

4.16.3. 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 mycmd is the defined alias.

```
A = LOAD 'data';

DEFINE mycmd 'stream.pl –n 5';

B = STREAM A THROUGH mycmd;
```

4.16.4. About Data Guarantees

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

script.

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

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

4.16.5. 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';
```

4.16.6. 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);
```

4.17. UNION

Computes the union of two or more relations.

4.17.1. Syntax

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

4.17.2. Terms

alias	The name of a relation.
ONSCHEMA	Use the keyword ONSCHEMA with UNION so that the union is based on column names of the input relations, and not column position. If the following requirements are not met, the statement will throw an error: • All inputs to the union should have a non null schema. • The data type for columns with same name in different input schemas should be compatible. Numeric types are compatible, and if column having same name in different input schemas have different numeric types , an implicit conversion will happen. bytearray type is considered compatible with all other types, a cast will be added to convert to other type. Bags or tuples having different inner schema are considered incompatible.

4.17.3. Usage

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

- Does not preserve the order of tuples. Both the input and output relations are interpreted as unordered bags of tuples.
- Does not ensure (as databases do) that all tuples adhere to the same schema or that they

have the same number of fields. In a typical scenario, however, this should be the case; therefore, it is the user's responsibility to either (1) ensure that the tuples in the input relations have the same schema or (2) be able to process varying tuples in the output relation.

- Does not eliminate duplicate tuples.

4.17.4. 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)
```

4.17.5. Example

This example shows the use of ONSCHEMA.

```
L1 = LOAD 'f1' USING (a : int, b : float);
DUMP L1;
(11,12.0)
(21,22.0)

L2 = LOAD  'f1' USING (a : long, c : chararray);
DUMP L2;
(11,a)
(12,b)
(13,c)

U = UNION ONSCHEMA L1, L2;
DESCRIBE U ;
```

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

DUMP U;
(11,12.0,)
(21,22.0,)
(11,,a)
(12,,b)
(13,,c)
```

5. Diagnostic Operators

5.1. DESCRIBE

Returns the schema of a relation.

5.1.1. Syntax

DESCRIBE alias;

5.1.2. Terms

alias	The name of a relation.

5.1.3. Usage

Use the DESCRIBE operator to view the schema of a relation. You can view outer relations as well as relations defined in a nested FOREACH statement.

See also: Debugging Pig Latin

5.1.4. 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}
```

This example shows how to view the schema of a nested relation using the :: operator.

```
A = LOAD 'studentab10k' AS (name, age, gpa);
B = GROUP A BY name;
C = FOREACH B {
     D = DISTINCT A.age;
     GENERATE COUNT(D), group;}

DESCRIBE C::D;
D: {age: bytearray}
```

5.2. DUMP

Dumps or displays results to screen.

5.2.1. Syntax

DUMP alias;

5.2.2. Terms

alias	The name of a relation.

5.2.3. Usage

Use the DUMP operator to run (execute) Pig Latin statements and display the results to your screen. DUMP is meant for interactive mode; statements are executed immediately and the results are not saved (persisted). You can use DUMP as a debugging device to make sure that the results you are expecting are actually generated.

Note that production scripts *should not* use DUMP as it will disable multi-query optimizations and is likely to slow down execution (see Store vs. Dump).

See also: Debugging Pig Latin

5.2.4. 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)
```

5.3. EXPLAIN

Displays execution plans.

5.3.1. Syntax

EXPLAIN [–script pigscript] [–out path] [–brief] [–dot] [–param param_name = param_value] [–param_file file_name] alias;

5.3.2. Terms

–script	Use to specify a pig script.
–out	Use to specify the output path (directory). Will generate a logical_plan[.txt|.dot], physical_plan[.text|.dot], exec_plan[.text|.dot] file in the specified path. Default (no path specified): Stdout
–brief	Does not expand nested plans (presenting a smaller graph for overview).
–dot	Text mode (default): multiple output (split) will be broken out in sections. Dot mode: outputs a format that can be passed to the dot utility for graphical display – will generate a directed-acyclic-graph (DAG) of the plans in any supported format (.gif, .jpg ...).
–param param_name = param_value	See Parameter Substitution.
–param_file file_name	See Parameter Substitution.
alias	The name of a relation.

5.3.3. Usage

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

If no script is given:

- 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.

- The physical plan shows how the logical operators are translated to backend-specific physical operators. Some backend optimizations also apply.
- The map reduce plan shows how the physical operators are grouped into map reduce jobs.

If a script without an alias is specified, it will output the entire execution graph (logical, physical, or map reduce).

If a script with a alias is specified, it will output the plan for the given alias.

See also: Debugging Pig Latin

5.3.4. 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 …
```

5.4. ILLUSTRATE

(Note! This feature is NOT maintained at the moment. We are looking for someone to adopt it.)

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

5.4.1. Syntax

ILLUSTRATE alias;

5.4.2. Terms

alias	The name of a relation.

5.4.3. Usage

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

- The data load statement must include a schema.
- 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;
```

5.4.4. 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     |
-------------------------------
```

6. UDF Statements

6.1. DEFINE

Assigns an alias to a UDF function or a streaming command.

6.1.1. Syntax

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

6.1.2. Terms

alias	The name for a UDF function or the name for a streaming command (the cmd_alias for the STREAM operator).
function	For use with functions.

	The name of a UDF function.
`command`	For use with streaming. A command, including the arguments, enclosed in back tics (where a command is anything that can be executed).
input	For use with streaming. INPUT ({stdin | 'path'} [USING serializer] [, {stdin | 'path'} [USING serializer] …]) Where: • INPUT – Keyword. • 'path' – A file path, enclosed in single quotes. • USING – Keyword. • serializer – PigStreaming is the default serializer.
output	For use with streaming. OUTPUT ({stdout | stderr | 'path'} [USING deserializer] [, {stdout | stderr | 'path'} [USING deserializer] …]) Where: • OUTPUT – Keyword. • 'path' – A file path, enclosed in single quotes. • USING – Keyword. • deserializer – PigStreaming is the default deserializer.
ship	For use with streaming. SHIP('path' [, 'path' …]) Where: • SHIP – Keyword. • 'path' – A file path, enclosed in single quotes.
cache	For use with streaming.

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

6.1.3. Usage

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

Use DEFINE to specify a UDF function when:

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

Use DEFINE to specify a streaming command when:

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

6.1.3.1. 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. PigStreaming is the default serialization/deserialization function.

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

```
DEFINE CMD 'perl PigStreaming.pl - nameMap' input(stdin using
PigStreaming(',')) output(stdout using PigStreaming(','));
A = LOAD 'file';
B = STREAM B THROUGH CMD;
```

If you need an alternative format, you will need to create a custom serializer/deserializer by

implementing the following interfaces.

```
interface PigToStream {

        /**
         * Given a tuple, produce an array of bytes to be passed to the
streaming
         * executable.
         */
        public byte[] serialize(Tuple t) throws IOException;
    }

    interface StreamToPig {

        /**
         *  Given a byte array from a streaming executable, produce a
tuple.
         */
        public Tuple deserialize(byte[]) throws IOException;

        /**
         * This will be called on the front end during planning and not on
the back
         * end during execution.
         *
         * @return the {@link LoadCaster} associated with this object.
         * @throws IOException if there is an exception during LoadCaster
         */
        public LoadCaster getLoadCaster() throws IOException;
    }
```

6.1.3.2. 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 actual cluster. The only guarantee is that the shipped files are available in the current working

directory of the launched job and that your current working directory is also on the PATH environment variable.

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

Note the following:

1. It is safe only to ship files to be executed from the current working directory on the task on the cluster.

```
OP = stream IP through 'script';
or
DEFINE CMD 'script' ship('/a/b/script');
OP = stream IP through 'CMD';
```

2. Shipping files to relative paths or absolute paths is undefined and mostly will fail since you may not have permissions to read/write/execute from arbitraty paths on the actual clusters.

6.1.3.3. 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.

6.1.3.4. About Auto-Ship

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

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

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

Note the following:

1. If Pig determines that it needs to auto-ship an absolute path it will not ship it at all since there is no way to ship files to the necessary location (lack of permissions and so on).

```
OP = stream IP through '/a/b/c/script';
or
OP = stream IP through 'perl /a/b/c/script.pl';
```

2. Pig will not auto-ship files in the following system directories (this is determined by executing 'which <file>' command).

```
/bin /usr/bin /usr/local/bin /sbin /usr/sbin /usr/local/sbin
```

3. To auto-ship, the file in question should be present in the PATH. So if the file is in the current working directory then the current working directory should be in the PATH.

6.1.4. Examples: Input/Output

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

```
X = STREAM A THROUGH 'stream.pl';
```

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

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

X = STREAM A THROUGH Y;
```

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

```
DEFINE Y 'stream.pl' INPUT(stdin USING MySerializer) OUTPUT (stdout USING
MyDeserializer);

X = STREAM A THROUGH Y;
```

6.1.5. Examples: Ship/Cache

In this example ship is used to send the script to the cluster compute nodes.

```
DEFINE Y 'stream.pl' SHIP('/work/stream.pl');

X = STREAM A THROUGH Y;
```

In this example cache is used to specify a file located on the cluster compute nodes.

```
DEFINE Y 'stream.pl data.gz' SHIP('/work/stream.pl')
CACHE('/input/data.gz#data.gz');
```

```
X = STREAM A THROUGH Y;
```

6.1.6. Example: DEFINE with STREAM

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

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

6.1.7. Examples: Logging

In this example the streaming stderr is stored in the _logs/<dir> directory of the job's output directory. Because the job can have multiple streaming applications associated with it, you need to ensure that different directory names are used to avoid conflicts. Pig stores up to 100 tasks per streaming job.

```
DEFINE Y 'stream.pl' stderr('<dir>' limit 100);
X = STREAM A THROUGH Y;
```

In this example a function is defined for use with the FOREACH …GENERATE operator.

```
REGISTER /src/myfunc.jar
DEFINE myFunc myfunc.MyEvalfunc('foo');
A = LOAD 'students';
B = FOREACH A GENERATE myFunc($0);
```

6.2. REGISTER

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

6.2.1. Syntax

REGISTER alias;

6.2.2. Terms

alias	The path to the JAR file (the full location URI is required). Do not place the name in quotes.

	Pig supports functions written in Java only. Pig supports JAR files stored in HDFS, Amazon S3, and other distributed files systems (see also Pig Latin Scripts).

6.2.3. Usage

Use the REGISTER statement inside a Pig script to specify the path of the JAR file containing UDFs.

You can register additional files (to use with your Pig script) via the command line using the -Dpig.additional.jars option.

For more information see the Pig UDF Manual.

6.2.4. Examples

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);
```

In this example additional jar files are registered via the command line.

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

In this example a jar file stored in HDFS is registered

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

7. Eval Functions

7.1. AVG

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

7.1.1. Syntax

AVG(expression)

7.1.2. Terms

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

7.1.3. 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.

The AVG function now ignores NULL values.

7.1.4. 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)
```

7.1.5. Types Tables

	int	long	float	double	chararray	bytearray

AVG	long	long	double	double	error	cast as double

7.2. CONCAT

Concatenates two expressions of identical type.

7.2.1. Syntax

CONCAT (expression, expression)

7.2.2. Terms

expression	Any expression.

7.2.3. Usage

Use the CONCAT function to concatenate two expressions. The result values of the two expressions must have identical types.

7.3. 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)
```

7.4. COUNT

Computes the number of elements in a bag.

7.4.1. Syntax

COUNT(expression)

7.4.2. Terms

expression	An expression with data type bag.

7.4.3. Usage

Use the COUNT function to compute 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.

The COUNT function follows syntax semantics and ignores nulls. What this means is that a tuple in the bag will not be counted if the *first field* in this tuple is NULL. If you want to include NULL values in the count computation, use COUNT_STAR.

Note: You cannot use the tuple designator (*) with COUNT; that is, COUNT(*) will not work.

7.4.4. 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)
```

7.4.5. Types Tables

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

7.5. COUNT_STAR

Computes the number of elements in a bag.

7.5.1. Syntax

COUNT_STAR(expression)

7.5.2. Terms

expression	An expression with data type bag.

7.5.3. Usage

Use the COUNT_STAR function to compute the number of elements in a bag. COUNT_STAR requires a preceding GROUP ALL statement for global counts and a GROUP BY statement for group counts.

COUNT_STAR includes NULL values in the count computation (unlike COUNT, which ignores NULL values).

7.5.4. Example

In this example COUNT_STAR is used the count the tuples in a bag.

```
X = FOREACH B GENERATE COUNT_STAR(A);
```

7.6. DIFF

Compares two fields in a tuple.

7.6.1. Syntax

DIFF (expression, expression)

7.6.2. Terms

expression	An expression with any data type.

7.6.3. Usage

The DIFF function takes two bags as arguments and compares them. Any tuples that are in one bag but not the other are returned in a bag. If the bags match, an empty bag is returned. If the fields are not bags then they will be wrapped in tuples and returned in a bag if they do not match, or an empty bag will be returned if the two records match. The implementation assumes that both bags being passed to the DIFF function will fit entirely into memory simultaneously. If this is not the case the UDF will still function but it will be VERY slow.

7.6.4. Example

In this example DIFF compares the tuples in two bags.

```
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)})
```

7.7. IsEmpty

Checks if a bag or map is empty.

7.7.1. Syntax

IsEmpty(expression)

7.7.2. Terms

expression	An expression with any data type.

7.7.3. Usage

The IsEmpty function checks if a bag or map is empty (has no data). The function can be used to filter data.

7.7.4. Example

In this example all students with an SSN but no name are located.

```
SSN = load 'ssn.txt' using PigStorage() as (ssn:long);

SSN_NAME = load 'students.txt' using PigStorage() as (ssn:long,
name:chararray);

-- do a left out join of SSN with SSN_Name
X = cogroup SSN by ssn inner, SSN_NAME by ssn;

-- only keep those ssn's for which there is no name
Y = filter X by IsEmpty(SSN_NAME);
```

7.8. 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.

7.8.1. Syntax

MAX(expression)

7.8.2. Terms

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

7.8.3. Usage

Use the MAX function to compute the maximum of the numeric values or chararrays in a

single-column bag.

7.8.4. 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)
```

7.8.5. Types Tables

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

7.9. 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.

7.9.1. Syntax

MIN(expression)

7.9.2. Terms

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

7.9.3. Usage

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

7.9.4. 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)
```

7.9.5. Types Tables

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

7.10. SIZE

Computes the number of elements based on any Pig data type.

7.10.1. Syntax

SIZE(expression)

7.10.2. Terms

expression	An expression with any data type.

7.10.3. Usage

Use the SIZE function to compute the number of elements based on the data type (see the Types Tables below). SIZE includes NULL values in the size computation. SIZE is not algebraic.

7.10.4. 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)
```

7.10.5. 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

7.11. 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.

7.11.1. Syntax

SUM(expression)

7.11.2. Terms

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

7.11.3. Usage

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

7.11.4. 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)
```

7.11.5. Types Tables

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

7.12. TOKENIZE

Splits a string and outputs a bag of words.

7.12.1. Syntax

TOKENIZE(expression)

7.12.2. Terms

expression	An expression with data type chararray.

7.12.3. 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(*).

7.12.4. 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.)})
```

8. Load/Store Functions

Load/Store functions determine how data goes into Pig and comes out of Pig. Pig provides a set of built-in load/store functions, described in the sections below. You can also write your own load/store functions (see the Pig UDF Manual).

8.1. Handling Compression

Support for compression is determined by the load/store function. PigStorage and TextLoader support gzip and bzip compression for both read (load) and write (store). BinStorage does not support compression.

To work with gzip compressed files, input/output files need to have a .gz extension. Gzipped files cannot be split across multiple maps; this means that the number of maps created is equal to the number of part files in the input location.

```
A = load ‘myinput.gz’;
store A into ‘myoutput.gz’;
```

To work with bzip compressed files, the input/output files need to have a .bz or .bz2 extension. Because the compression is block-oriented, bzipped files can be split across multiple maps.

```
A = load ‘myinput.bz’;
store A into ‘myoutput.bz’;
```

Note: PigStorage and TextLoader correctly read compressed files as long as they are NOT CONCATENATED FILES generated in this manner:

- cat *.gz > text/concat.gz
- cat *.bz > text/concat.bz
- cat *.bz2 > text/concat.bz2

If you use concatenated gzip or bzip files with your Pig jobs, you will NOT see a failure but

the results will be INCORRECT.

8.2. BinStorage

Loads and stores data in machine-readable format.

8.2.1. Syntax

BinStorage()

8.2.2. Terms

none	no parameters

8.2.3. Usage

BinStorage works with data that is represented on disk in machine-readable format. BinStorage does NOT support compression.

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

BinStorage supports multiple locations (files, directories, globs) as input.

8.2.4. Example

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

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

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

In this example BinStorage is used to load multiple locations.

```
A = LOAD 'input1.bin, input2.bin' USING BinStorage();
```

8.3. PigStorage

Loads and stores data in UTF-8 format.

8.3.1. Syntax

PigStorage(field_delimiter)

8.3.2. Terms

field_delimiter	Parameter. The default field delimiter is tab ('\t'). You can specify other characters as field delimiters; however, be sure to encase the characters in single quotes.

8.3.3. Usage

PigStorage is the default function for the LOAD and STORE operators and works with both simple and complex data types.

PigStorage supports structured text files (in human-readable UTF-8 format). PigStorage also supports compression.

PigStorage supports multiple locations (files, directories, globs) as input.

Load statements – PigStorage expects data to be formatted using field delimiters, either the tab character ('\t') or other specified character.

Store statements – PigStorage outputs data using field deliminters, either the tab character ('\t') or other specified character, and the line feed record delimiter ('\n').

Field Delimiters – For 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).

Record Deliminters – For load statements Pig interprets the line feed ('\n'), carriage return ('\r' or CTRL-M) and combined CR + LF ('\r\n') characters as record delimiters (do not use these characters as field delimiters). For store statements Pig uses the line feed ('\n') character as the record delimiter.

8.3.4. 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('*');
```

8.4. PigDump

Stores data in UTF-8 format.

8.4.1. Syntax

PigDump()

8.4.2. Terms

none	no parameters

8.4.3. Usage

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

8.4.4. Example

In this example PigDump is used with the STORE function.

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

8.5. TextLoader

Loads unstructured data in UTF-8 format.

8.5.1. Syntax

TextLoader()

8.5.2. Terms

none	no parameters

8.5.3. Usage

TextLoader works with unstructured data in UTF8 format. Each resulting tuple contains a single field with one line of input text. TextLoader also supports compression.

Currently, TextLoader support for compression is limited.

TextLoader cannot be used to store data.

8.5.4. Example

In this example TextLoader is used with the LOAD function.

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

9. Math Functions

For general information about these functions, see the Java API Specification, Class Math. Note the following:

- Pig function names are case sensitive and UPPER CASE.
- Pig may process results differently than as stated in the Java API Specification:
  - If the result value is null or empty, Pig returns null.
  - If the result value is not a number (NaN), Pig returns null.
  - If Pig is unable to process the expression, Pig returns an exception.

9.1. ABS

Returns the absolute value of an expression.

9.1.1. Syntax

ABS(expression)

9.1.2. Terms

expression	Any expression whose result is type int, long, float, or double.

9.1.3. Usage

Use the ABS function to return the absolute value of an expression. If the result is not negative (x # 0), the result is returned. If the result is negative (x < 0), the negation of the result is returned.

9.2. ACOS

Returns the arc cosine of an expression.

9.2.1. Syntax

ACOS(expression)

9.2.2. Terms

expression	An expression whose result is type double.

9.2.3. Usage

Use the ACOS function to return the arc cosine of an expression.

9.3. ASIN

Returns the arc sine of an expression.

9.3.1. Syntax

ASIN(expression)

9.3.2. Terms

expression	An expression whose result is type double.

9.3.3. Usage

Use the ASIN function to return the arc sine of an expression.

9.4. ATAN

Returns the arc tangent of an expression.

9.4.1. Syntax

ATAN(expression)

9.4.2. Terms

expression	An expression whose result is type double.

9.4.3. Usage

Use the ATAN function to return the arc tangent of an expression.

9.5. CBRT

Returns the cube root of an expression.

9.5.1. Syntax

CBRT(expression)

9.5.2. Terms

expression	An expression whose result is type double.

9.5.3. Usage

Use the CBRT function to return the cube root of an expression.

9.6. CEIL

Returns the value of an expression rounded up to the nearest integer.

9.6.1. Syntax

CEIL(expression)

9.6.2. Terms

expression	An expression whose result is type double.

9.6.3. Usage

Use the CEIL function to return the value of an expression rounded up to the nearest integer. This function never decreases the result value.

x	CEIL(x)
4.6	5
3.5	4
2.4	3
1.0	1
-1.0	-1
-2.4	-2
-3.5	-3
-4.6	-4

9.7. COSH

Returns the hyperbolic cosine of an expression.

9.7.1. Syntax

COSH(expression)

9.7.2. Terms

expression	An expression whose result is type double.

9.7.3. Usage

Use the COSH function to return the hyperbolic cosine of an expression.

9.8. COS

Returns the trigonometric cosine of an expression.

9.8.1. Syntax

COS(expression)

9.8.2. Terms

expression	An expression (angle) whose result is type double.

9.8.3. Usage

Use the COS function to return the trigonometric cosine of an expression.

9.9. EXP

Returns Euler's number e raised to the power of x.

9.9.1. Syntax

EXP(expression)

9.9.2. Terms

expression	An expression whose result is type double.

9.9.3. Usage

Use the EXP function to return the value of Euler's number e raised to the power of x (where x is the result value of the expression).

9.10. FLOOR

Returns the value of an expression rounded down to the nearest integer.

9.10.1. Syntax

FLOOR(expression)

9.10.2. Terms

expression	An expression whose result is type double.

9.10.3. Usage

Use the FLOOR function to return the value of an expression rounded down to the nearest integer. This function never increases the result value.

x	CEIL(x)
4.6	4
3.5	3
2.4	2
1.0	1
-1.0	-1
-2.4	-3
-3.5	-4
-4.6	-5

9.11. LOG

Returns the natural logarithm (base e) of an expression.

9.11.1. Syntax

LOG(expression)

9.11.2. Terms

expression	An expression whose result is type double.

9.11.3. Usage

Use the LOG function to return the natural logarithm (base e) of an expression.

9.12. LOG10

Returns the base 10 logarithm of an expression.

9.12.1. Syntax

LOG10(expression)

9.12.2. Terms

expression	An expression whose result is type double.

9.12.3. Usage

Use the LOG10 function to return the base 10 logarithm of an expression.

9.13. RANDOM

Returns a pseudo random number.

9.13.1. Syntax

RANDOM( )

9.13.2. Terms

N/A	No terms.

9.13.3. Usage

Use the RANDOM function to return a pseudo random number (type double) greater than or equal to 0.0 and less than 1.0.

9.14. ROUND

Returns the value of an expression rounded to an integer.

9.14.1. Syntax

ROUND(expression)

9.14.2. Terms

expression	An expression whose result is type float or double.

9.14.3. Usage

Use the ROUND function to return the value of an expression rounded to an integer (if the result type is float) or rounded to a long (if the result type is double).

x	CEIL(x)
4.6	5
3.5	4
2.4	2
1.0	1
-1.0	-1
-2.4	-2
-3.5	-3
-4.6	-5

9.15. SIN

Returns the sine of an expression.

9.15.1. Syntax

SIN(expression)

9.15.2. Terms

expression	An expression whose result is double.

9.15.3. Usage

Use the SIN function to return the sine of an expession.

9.16. SINH

Returns the hyperbolic sine of an expression.

9.16.1. Syntax

SINH(expression)

9.16.2. Terms

expression	An expression whose result is double.

9.16.3. Usage

Use the SINH function to return the hyperbolic sine of an expression.

9.17. SQRT

Returns the positive square root of an expression.

9.17.1. Syntax

SQRT(expression)

9.17.2. Terms

expression	An expression whose result is double.

9.17.3. Usage

Use the SQRT function to return the positive square root of an expression.

9.18. TAN

Returns the trignometric tangent of an angle.

9.18.1. Syntax

```
TAN(expression)
```

9.18.2. Terms

expression	An expression (angle) whose result is double.

9.18.3. Usage

Use the TAN function to return the trignometric tangent of an angle.

9.19. TANH

Returns the hyperbolic tangent of an expression.

9.19.1. Syntax

```
TANH(expression)
```

9.19.2. Terms

expression	An expression whose result is double.

9.19.3. Usage

Use the TANH function to return the hyperbolic tangent of an expression.

10. String Functions

For general information about these functions, see the Java API Specification, Class String. Note the following:

- Pig function names are case sensitive and UPPER CASE.
- Pig string functions have an extra, first parameter: the string to which all the operations are applied.
- Pig may process results differently than as stated in the Java API Specification. If any of the input parameters are null or if an insufficient number of parameters are supplied, NULL is returned.

10.1. INDEXOF

Returns the index of the first occurrence of a character in a string, searching forward from a start index.

10.1.1. Syntax

INDEXOF(string, 'character', startIndex)

10.1.2. Terms

string	The string to be searched.
'character'	The character being searched for, in quotes.
startIndex	The index from which to begin the forward search. The string index begins with zero (0).

10.1.3. Usage

Use the INDEXOF function to determine the index of the first occurrence of a character in a string. The forward search for the character begins at the designated start index.

10.2. LAST_INDEX_OF

Returns the index of the last occurrence of a character in a string, searching backward from a start index.

10.2.1. Syntax

LAST_INDEX_OF(expression)

10.2.2. Terms

string	The string to be searched.
'character'	The character being searched for, in quotes.
startIndex	The index from which to begin the backward search. The string index begins with zero (0).

10.2.3. Usage

Use the LAST_INDEX_OF function to determine the index of the last occurrence of a character in a string. The backward search for the character begins at the designated start index.

10.3. LCFIRST

Converts the first character in a string to lower case.

10.3.1. Syntax

LCFIRST(expression)

10.3.2. Terms

expression	An expression whose result type is chararray.

10.3.3. Usage

Use the LCFIRST function to convert only the first character in a string to lower case.

10.4. LOWER

Converts all characters in a string to lower case.

10.4.1. Syntax

LOWER(expression)

10.4.2. Terms

expression	An expression whose result type is chararray.

10.4.3. Usage

Use the LOWER function to convert all characters in a string to lower case.

10.5. REGEX_EXTRACT

Performs regular expression matching and extracts the matched group defined by an index parameter.

10.5.1. Syntax

REGEX_EXTRACT (string, regex, index)

10.5.2. Terms

string	The string in which to perform the match.
regex	The regular expression.
index	The index of the matched group to return.

10.5.3. Usage

Use the REGEX_EXTRACT function to perform regular expression matching and to extract the matched group defined by the index parameter (where the index is a 1-based parameter.) The function uses Java regular expression form.

The function returns a string that corresponds to the matched group in the position specified by the index. If there is no matched expression at that position, NULL is returned.

10.5.4. Example

This example will return the string '192.168.1.5'.

```
REGEX_EXTRACT('192.168.1.5:8020', '(.*)\:(.*)', 1);
```

10.6. REGEX_EXTRACT_ALL

Performs regular expression matching and extracts all matched groups.

10.6.1. Syntax

REGEX_EXTRACT (string, regex)

10.6.2. Terms

string	The string in which to perform the match.
regex	The regular expression.

10.6.3. Usage

Use the REGEX_EXTRACT_ALL function to perform regular expression matching and to extract all matched groups. The function uses Java regular expression form.

The function returns a tuple where each field represents a matched expression. If there is no match, an empty tuple is returned.

10.6.4. Example

This example will return the tuple (192.168.1.5,8020).

```
REGEX_EXTRACT_ALL('192.168.1.5:8020', '(.*)\:(.*)');
```

10.7. REPLACE

Replaces existing characters in a string with new characters.

10.7.1. Syntax

REPLACE(string, 'oldChar', 'newChar');

10.7.2. Terms

string	The string to be updated.
'oldChar'	The existing characters being replaced, in quotes.
'newChar'	The new characters replacing the existing characters, in quotes.

10.7.3. Usage

Use the REPLACE function to replace existing characters in a string with new characters.

For example, to change "open source software" to "open source wiki" use this statement:
REPLACE(string,'software','wiki');

10.8. STRSPLIT

Splits a string around matches of a given regular expression.

10.8.1. Syntax

STRSPLIT(string, regex, limit)

10.8.2. Terms

string	The string to be split.
regex	The regular expression.
Limit	The number of times the pattern (the compiled representation of the regular expression) is applied.

10.8.3. Usage

Use the STRSPLIT function to split a string around matches of a given regular expression.

For example, given the string (open:source:software), STRSPLIT (string, ':',2) will return ((open,source:software)) and STRSPLIT (string, ':',3) will return ((open,source,software)).

10.9. SUBSTRING

Returns a substring from a given string.

10.9.1. Syntax

SUBSTRING(string, startIndex, stopIndex)

10.9.2. Terms

string	The string from which a substring will be extracted.
startIndex	The index (type integer) of the first character of the substring. The index of a string begins with zero (0).
stopIndex	The index (type integer) of the character *following* the last character of the substring.

10.9.3. Usage

Use the SUBSTRING function to return a substring from a given string.

Given a field named alpha whose value is ABCDEF, to return substring BCD use this statement: SUBSTRING(alpha,1,4). Note that 1 is the index of B (the first character of the substring) and 4 is the index of E (the character *following* the last character of the substring).

10.10. TRIM

Returns a copy of a string with leading and trailing white space removed.

10.10.1. Syntax

TRIM(expression)

10.10.2. Terms

expression	An expression whose result is chararray.

10.10.3. Usage

Use the TRIM function to remove leading and trailing white space from a string.

10.11. UCFIRST

Returns a string with the first character converted to upper case.

10.11.1. Syntax

UCFIRST(expression)

10.11.2. Terms

expression	An expression whose result type is chararray.

10.11.3. Usage

Use the UCFIRST function to convert only the first character in a string to upper case.

10.12. UPPER

Returns a string converted to upper case.

10.12.1. Syntax

UPPER(expression)

10.12.2. Terms

expression	An expression whose result type is chararray.

10.12.3. Usage

Use the UPPER function to convert all characters in a string to upper case.

11. Bag and Tuple Functions

11.1. TOBAG

Converts one or more expressions to type bag.

11.1.1. Syntax

TOBAG(expression [, expression ...])

11.1.2. Terms

expression	An expression with any data type.

11.1.3. Usage

Use the TOBAG function to convert one or more expressions to individual tuples which are then placed in a bag.

11.1.4. Example

In this example, fields f1 and f3 are converted to tuples that are then placed in a bag.

```
a = LOAD 'student' AS (f1:chararray, f2:int, f3:float);
DUMP a;

(John,18,4.0)
(Mary,19,3.8)
(Bill,20,3.9)
(Joe,18,3.8)

b = FOREACH a GENERATE TOBAG(f1,f3);
DUMP b;

({(John),(4.0)})
({(Mary),(3.8)})
({(Bill),(3.9)})
({(Joe),(3.8)})
```

11.2. TOP

Returns the top-n tuples from a bag of tuples.

11.2.1. Syntax

TOP(topN,column,relation)

11.2.2. Terms

topN	The number of top tuples to return (type integer).
column	The tuple column whose values are being compared.
relation	The relation (bag of tuples) containing the tuple column.

11.2.3. Usage

TOP function returns a bag containing top N tuples from the input bag where N is controlled by the first parameter to the function. The tuple comparison is performed based on a single column from the tuple. The column position is determined by the second parameter to the function. The function assumes that all tuples in the bag contain an element of the same type in the compared column

11.2.4. Example

In this example the top 10 occurrences are returned.

```
A = LOAD 'data' as (first: chararray, second: chararray);
B = GROUP A BY (first, second);
C = FOREACH B generate FLATTEN(group), COUNT(*) as count;
D = GROUP C BY first; // again group by first
topResults = FOREACH D {
    result = TOP(10, 2, C); // and retain top 10 occurrences of 'second' in
first
    GENERATE FLATTEN(result);
}
```

11.3. TOTUPLE

Converts one or more expressions to type tuple.

11.3.1. Syntax

TOTUPLE(expression [, expression ...])

11.3.2. Terms

expression	An expression of any datatype.

11.3.3. Usage

Use the TOTUPLE function to convert one or more expressions to a tuple.

11.3.4. Example

In this example, fields f1, f2 and f3 are converted to a tuple.

```
a = LOAD 'student' AS (f1:chararray, f2:int, f3:float);
DUMP a;

(John,18,4.0)
(Mary,19,3.8)
(Bill,20,3.9)
(Joe,18,3.8)

b = FOREACH a GENERATE TOTUPLE(f1,f2,f3);
DUMP b;

((John,18,4.0))
((Mary,19,3.8))
((Bill,20,3.9))
((Joe,18,3.8))
```

12. File Commands

Note: Beginning with Pig 0.6.0, the file commands are now deprecated and will be removed in a future release. Start using Pig's -fs command to invoke the shell commands shell commands.

12.1. cat

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

12.1.1. Syntax

cat path [path …]

12.1.2. Terms

path	The location of a file or directory.

12.1.3. 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.

12.1.4. Example

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

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

12.2. cd

Changes the current directory to another directory.

12.2.1. Syntax

cd [dir]

12.2.2. Terms

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

12.2.3. 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.

12.2.4. Example

In this example we move to the /data directory.

```
grunt> cd /data
```

12.3. copyFromLocal

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

12.3.1. Syntax

copyFromLocal src_path dst_path

12.3.2. Terms

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

12.3.3. 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.

12.3.4. 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
```

12.4. copyToLocal

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

12.4.1. Syntax

copyToLocal src_path dst_path

12.4.2. Terms

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

12.4.3. 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.

12.4.4. Example

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

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

12.5. cp

Copies a file or directory within HDFS.

12.5.1. Syntax

cp src_path dst_path

12.5.2. Terms

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

12.5.3. 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.

12.5.4. Example

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

```
grunt> cp students students_save
```

12.6. ls

Lists the contents of a directory.

12.6.1. Syntax

ls [path]

12.6.2. Terms

path	The name of the path/directory.

12.6.3. 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.

12.6.4. 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>
```

12.7. mkdir

Creates a new directory.

12.7.1. Syntax

mkdir path

12.7.2. Terms

path	The name of the path/directory.

12.7.3. 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.

12.7.4. Example

In this example a directory and subdirectory are created.

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

12.8. mv

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

12.8.1. Syntax

mv src_path dst_path

12.8.2. Terms

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

12.8.3. 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.

12.8.4. 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
```

12.9. pwd

Prints the name of the current working directory.

12.9.1. Syntax

pwd

12.9.2. Terms

none	no parameters

12.9.3. Usage

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

12.9.4. Example

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

```
grunt> pwd
/data
```

12.10. rm

Removes one or more files or directories.

12.10.1. Syntax

rm path [path…]

12.10.2. Terms

path	The name of the path/directory/file.

12.10.3. 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.

12.10.4. Example

In this example files are removed.

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

12.11. rmf

Forcibly removes one or more files or directories.

12.11.1. Syntax

rmf path [path …]

12.11.2. Terms

path	The name of the path/directory/file.

12.11.3. 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.

12.11.4. Example

In this example files are forcibly removed.

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

13. Shell Commands

13.1. fs

Invokes any FSShell command from within a Pig script or the Grunt shell.

13.1.1. Syntax

fs subcommand subcommand_parameters

13.1.2. Terms

subcommand	The FSShell command.
subcommand_parameters	The FSShell command parameters.

13.1.3. Usage

Use the fs command to invoke any FSShell command from within a Pig script or Grunt shell. The fs command greatly extends the set of supported file system commands and the capabilities supported for existing commands such as ls that will now support globing. For a complete list of FSShell commands, see File System Shell Guide

13.1.4. Examples

In these examples a directory is created, a file is copied, a file is listed.

```
fs -mkdir /tmp
```

```
fs -copyFromLocal file-x file-y
fs -ls file-y
```

13.2. sh

Invokes any sh shell command from within a Pig script or the Grunt shell.

13.2.1. Syntax

sh subcommand subcommand_parameters

13.2.2. Terms

subcommand	The sh shell command.
subcommand_parameters	The sh shell command parameters.

13.2.3. Usage

Use the sh command to invoke any sh shell command from within a Pig script or Grunt shell.

Note that only real programs can be run form the sh command. Commands such as cd are not programs but part of the shell environment and as such cannot be executed unless the user invokes the shell explicitly, like "bash cd".

13.2.4. Example

In this example the ls command is invoked.

```
grunt> sh ls
bigdata.conf
nightly.conf
.....
grunt>
```

14. Utility Commands

14.1. exec

Run a Pig script.

14.1.1. Syntax

exec [–param param_name = param_value] [–param_file file_name] script

14.1.2. Terms

–param param_name = param_value	See Parameter Substitution.
–param_file file_name	See Parameter Substitution.
script	The name of a Pig script.

14.1.3. Usage

Use the exec command to run a Pig script with no interaction between the script and the Grunt shell (batch mode). 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.

With the exec command, store statements will not trigger execution; rather, the entire script is parsed before execution starts. Unlike the run command, exec does not change the command history or remembers the handles used inside the script. Exec without any parameters can be used in scripts to force execution up to the point in the script where the exec occurs.

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.

14.1.4. Examples

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
```

In this example multiple parameters are specified.

```
grunt> exec –param p1=myparam1 –param p2=myparam2 myscript.pig
```

14.2. help

Prints a list of Pig commands or properties.

14.2.1. Syntax

-help [properties]

14.2.2. Terms

properties	List Pig properties.

14.2.3. Usage

The help command prints a list of Pig commands or properties.

14.2.4. Example

Use "help" to get a list of commands.

```
$ pig -help

Apache Pig version 0.8.1-dev (r1094835)
compiled Apr 18 2011, 19:26:53

USAGE: Pig [options] [-] : Run interactively in grunt shell.
       Pig [options] -e[xecute] cmd [cmd ...] : Run cmd(s).
       Pig [options] [-f[ile]] file : Run cmds found in file.
  options include:
    -4, -log4jconf - Log4j configuration file, overrides log conf
    -b, -brief - Brief logging (no timestamps)
    -c, -check - Syntax check
etc …
```

Use "-help properties" to get a list of properties.

```
$ pig -help properties

The following properties are supported:
    Logging:
        verbose=true|false; default is false. This property is the same as
-v switch
        brief=true|false; default is false. This property is the same as -b
switch
        debug=OFF|ERROR|WARN|INFO|DEBUG; default is INFO. This property is
the same as -d switch
        aggregate.warning=true|false; default is true. If true, prints
count of warnings
            of each type rather than logging each warning.
etc …
```

14.3. kill

Kills a job.

14.3.1. Syntax

kill jobid

14.3.2. Terms

jobid	The job id.

14.3.3. Usage

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

14.3.4. Example

In this example the job with id job_0001 is killed.

```
grunt> kill job_0001
```

14.4. quit

Quits from the Pig grunt shell.

14.4.1. Syntax

exit

14.4.2. Terms

none	no parameters

14.4.3. Usage

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

14.4.4. Example

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

```
grunt> quit
```

14.5. run

Run a Pig script.

14.5.1. Syntax

run [–param param_name = param_value] [–param_file file_name] script

14.5.2. Terms

–param param_name = param_value	See Parameter Substitution.
–param_file file_name	See Parameter Substitution.
script	The name of a Pig script.

14.5.3. Usage

Use the run command to run a Pig script that can interact with the Grunt shell (interactive mode). 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.

With the run command, every store triggers execution. The statements from the script are put

into the command history and all the aliases defined in the script can be referenced in subsequent statements after the run command has completed. Issuing a run command on the grunt command line has basically the same effect as typing the statements manually.

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.

14.5.4. 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
```

14.6. set

Assigns values to keys used in Pig.

14.6.1. Syntax

```
set key 'value'
```

14.6.2. Terms

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

14.6.3. Usage

Use the set command to assign values to keys, as shown in the table. All keys and their corresponding values (for Pig and Hadoop) are case sensitive.

Key	Value	Description
default_parallel	a whole number	Sets the number of reducers for all MapReduce jobs generated by Pig (see Use the Parallel Features).
debug	on/off	Turns debug-level logging on or off.
job.name	Single-quoted string that contains the job name.	Sets user-specified name for the job
job.priority	Acceptable values (case insensitive): very_low, low, normal, high, very_high	Sets the priority of a Pig job.
stream.skippath	String that contains the path.	For streaming, sets the path from which not to ship data (see DEFINE and About Auto-Ship).

All Pig and Hadoop properties can be set, either in the Pig script or via the Grunt command line.

14.6.4. Examples

In this example key value pairs are set at the command line.

```
grunt> SET debug 'on'
grunt> SET job.name 'my job'
grunt> SET default_parallel 100
```

In this example default_parallel is set in the Pig script; all MapReduce jobs that get launched will use 20 reducers.

```
SET default_parallel 20;
A = LOAD 'myfile.txt' USING PigStorage() AS (t, u, v);
B = GROUP A BY t;
C = FOREACH B GENERATE group, COUNT(A.t) as mycount;
D = ORDER C BY mycount;
STORE D INTO 'mysortedcount' USING PigStorage();
```

In this example multiple key value pairs are set in the Pig script. These key value pairs are put in job-conf by Pig (making the pairs available to Pig and Hadoop). This is a script-wide setting; if a key value is defined multiple times in the script the last value will take effect and will be set for all jobs generated by the script.

```
...
SET mapred.map.tasks.speculative.execution false;
SET pig.logfile mylogfile.log;
SET my.arbitrary.key my.arbitary.value;
...
```