Pig

In this session, you will learn about:

- **Introduction to Pig**
- **Uses, comparisons**
- **Programming in Pig**
- **Use cases, real time data analytics using Pig**

Pig - introduction

- Apache Pig is an abstraction over MapReduce.
- It is a tool/platform which is used to analyze large sets of data representing them as data flows.
- Pig is generally used with Hadoop; we can perform all the data manipulation operations in Hadoop using Apache Pig.
- To write programs, Pig provides a high-level language known as Pig Latin.
- This language provides various operators using which programmers can develop their own functions for reading, writing, and processing data.
- All these scripts are internally converted to Map and Reduce tasks.
- Apache Pig has a component known as Pig Engine that accepts the Pig Latin scripts as input and converts those scripts into MapReduce jobs.

Pig - introduction

- How it benefits: Java is complicated. Pig allows an easier, more intuitive interface to execute in mapreduce framework.
- Using Pig Latin, programmers can perform MapReduce tasks easily without having to type complex codes in Java.
- Pig Latin is SQL-like language and it is easy to learn Apache Pig when you are familiar with SQL.
- Apache Pig provides many built-in operators to support data operations like joins, filters, ordering, etc.
- In addition, it also provides nested data types like tuples, bags, and maps that are missing from MapReduce.
- Code size, development, debugging time is expectedly shorter with Pig as compared to Java.

Pig - features

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

Pig - comparison with MapReduce

Pig	Mapreduce
Apache Pig is a data flow language.	MapReduce is a data processing paradigm.
It is a high level language.	MapReduce is low level and rigid.
Performing a Join operation in Apache Pig is pretty simple.	It is quite difficult in MapReduce to perform a Join operation between datasets.
Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig.	Exposure to Java is must to work with MapReduce.
Apache Pig uses multi-query approach, thereby reducing the length of the code to a great extent.	MapReduce will require almost 20 times more the number of lines to perform the same task.
There is no need for compilation.	MapReduce jobs have a long compilation process.
On execution, every Apache Pig operator is converted internally into a MapReduce job.	

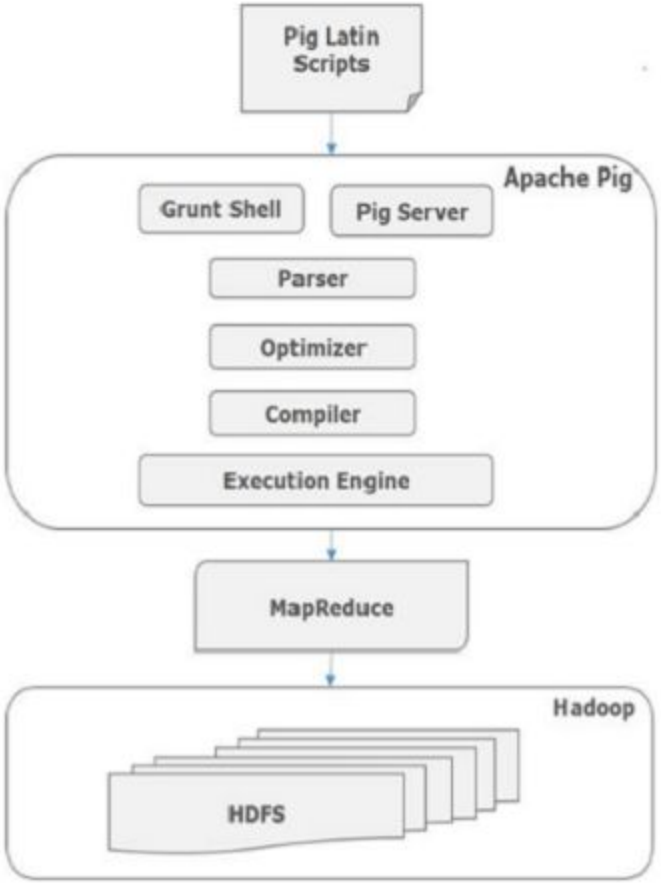
Pig - comparison with SQL

Pig	SQL
Pig Latin is a procedural language.	SQL is a declarative language.
In Apache Pig, schema is optional. We can store data without designing a schema	Schema is mandatory in SQL.
The data model in Apache Pig is nested relational.	The data model used in SQL is flat relational.
Apache Pig provides limited opportunity for Query optimization.	There is more opportunity for query optimization in SQL.

Pig - some applications

- To process huge data sources such as web logs.
- To perform data processing for search platforms.
- To process time sensitive data loads.
- Easier implementation of ETL

Pig - architecture



Pig - architecture

Parser

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

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

Optimizer

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

Pig - data types

int Represents a signed 32-bit integer.

Example : 8

long Represents a signed 64-bit integer.

Example : 5L

float Represents a signed 32-bit floating point.

Example : 5.5F

Pig - data types

double Represents a 64-bit floating point.

Example : 10.5

chararray Represents a character array (string) in Unicode UTF-8 format.

Example : ‘Lord Voldemort looks worse than a pig.’

Bytearray Represents a Byte array (blob).

Boolean Represents a Boolean value.

Example : true/ false.

Datetime Represents a date-time.

Example : 1970-01-01T00:00:00.000+00:00

Pig - data types

Biginteger Represents a Java BigInteger.

Example : 60708090709

Bigdecimal Represents a Java BigDecimal

Example : 185.98376256272893883

Pig - NULL treatment

Values for all the data types can be NULL.

Apache Pig treats null values in a similar way as SQL does.

A null can be an unknown value or a non-existent value.

It is used as a placeholder for optional values.

These nulls can occur naturally or can be the result of an operation.

Pig - complex data types and data model

Atom

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

Tuple

- A tuple is an ordered set of fields.
- Example : ('Harry Potter', 27)

Bag

- A bag is a collection of tuples.
- Example : {('Harry Potter',27), ('Ronald Weasly',28)}

Pig - complex data types and data model

Map

- A Map is a set of key-value pairs.
- Example : [‘name’#’Tony Stark’, ‘age’#45]

Relation

- A bag of bags
- Contains many tuples
- Tuples don’t need to have the same structure
- Relations contain unordered tuples hence execution order can change at runtime

Pig - execution modes

Local Mode

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

MapReduce Mode

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

Pig - execution mechanisms

Interactive Mode (Grunt shell)

- You can run Apache Pig in interactive mode using the Grunt shell.
- You can enter the Pig Latin statements and get the output.
- pig -x local
- pig -x mapreduce

Batch Mode (Script)

- You can run Apache Pig in Batch mode by writing the Pig Latin script in a single file with .pig extension.
- pig -x local <pig script name>
- pig -x mapreduce <pig script name>

Embedded Mode (UDF)

- Apache Pig provides the provision of defining our own functions (User Defined Functions) in programming languages such as Java, and using them in our script.

Pig - statements

- These statements work with relations. They include expressions and schemas.
- Every statement ends with a semicolon (;).
- Except LOAD and STORE, while performing all other operations, Pig Latin statements take a relation as input and produce another relation as output.

Pig - Installation

As root:

cd /usr/local

wget http://www.us.apache.org/dist/pig/pig-0.16.0/pig-0.16.0.tar.gz

tar zxvf pig-0.16.0tar.gz

mv pig-0.16.0 pig

Update .bashrc and then source it

export PIG_HOME=/usr/local/pig

export PIG_CLASSPATH=$HADOOP_HOME/etc/hadoop

export PATH=$PATH:${PIG_HOME}/bin

test install by: pig -version

Pig installation - Quick check and a first attempt

Objective:

- Create file imsi_data.txt containing columns: number_source, number_dest,minutes,msgs,data
- Add 5 rows to it. All values must be integers
- Run a simple aggregation - group by the first column (Date)
- segregate Date in a separate variable

Note: Just ensure there is a duplicate row in your file.
Ensure to turn on the jobhistory server: mr-jobhistory-daemon.sh start historyserver

Initiate your grunt shell: pig -x local

```
grunt > pc=LOAD '/<your path>/imsi_data.txt' USING PigStorage('\t')  AS (f1:int, f2:int, f3:int, f4:int, f5:int);
grunt > B=GROUP pc BY f1;
grunt > C=FOREACH B GENERATE ($0);
grunt > Dump C;
```

Repeat for mapreduce mode on the same file.

Pig Latin statements

Pig Latin statements are generally organized as follows:

- A LOAD statement to read data from the file system.
- A series of "transformation" statements to process the data.
- A DUMP statement to view results or a STORE statement to save the results.
- Note that a DUMP or STORE statement is required to generate output.

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

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

B = FOREACH A GENERATE name;
```

In this example, Pig will validate and then execute the LOAD, FOREACH, and DUMP statements.

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

B = FOREACH A GENERATE name;

DUMP B;
```

Pig - Schemas

- Schemas enable you to assign names to fields and declare types for fields.
- Schemas are optional.
- Defined with the LOAD, STREAM, and FOREACH operators using the AS clause.
- If you define a schema using the LOAD operator, then it is the load function that enforces the schema.
- Schemas for simple types and complex types can be used anywhere a schema definition is appropriate.

Known Schema Handling

- 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 assign a type to a field, you can subsequently change the type using the cast operators.

Unknown Schema Handling

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

Pig - Schemas

Schemas with LOAD and STREAM

With LOAD and STREAM operators, the schema following the AS keyword must be enclosed in parentheses. In this example the LOAD statement includes a schema definition for simple data types.

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

Schemas with FOREACH

With FOREACH operators, the schema following the AS keyword must be enclosed in parentheses when the FLATTEN operator is used. Otherwise, the schema should not be enclosed in parentheses. In this example the FOREACH statement includes FLATTEN and a schema for simple data types.

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

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

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

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

Pig - Schemas for simple data types

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

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

Terms
alias: The name assigned to the field.
type: (Optional) The simple data type assigned to the field.

The alias and type are separated by a colon (:). Multiple fields are enclosed in parentheses and separated by commas.

In the following example the schema defines multiple types.

```
cat student;
John    18      4.0
Mary    19      3.8
Bill    20      3.9
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)
```

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

```
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)
```

Pig - Schemas for complex types

Complex data types include tuples, bags, and maps.

Tuple Schemas

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

- `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

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)}
```

Pig - Schemas for complex types

Bag Schemas

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

Terms

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

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)})
```

Pig - Schemas for complex types

Map Schemas

Syntax: alias<:map> [<type>]

Terms

- alias: The name assigned to the map.
- :map: (Optional) The data type, map (case insensitive).
- [ ]: The designation for a map, a set of straight brackets [ ].
- type: (Optional) The datatype (all types allowed, bytearray is the default). The type applies to the map value only; the map key is always type chararray. If a type is declared then ALL values in the map must be of this type.

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

```
cat data;
[open#apache]
[apache#hadoop]

A = LOAD 'data' AS (M:map []);
A = LOAD 'data' AS (M:[]);
DESCRIBE A;
a: {M: map[ ]}

DUMP A;
([open#apache])
([apache#hadoop])
```

Pig - Schemas for complex types

You can define schemas for data that includes multiple types.

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:[] );
```

Pig - LOAD

Use the LOAD operator to read data into Pig.

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

- '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.
- USING: If the USING clause is omitted, the default load function PigStorage is used.
- function: The load function. You can use a built in function (see Load/Store Functions). 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.
- AS: Keyword.
- schema: A schema using the AS keyword, enclosed in parentheses.

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.

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.

Example:

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

Pig - FILTER

Selects tuples from a relation based on some condition.

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

Terms

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

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).

```
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)
```

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)
```

Pig - FOREACH

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).

```
alias  = FOREACH { block | nested_block };
```

Terms

alias: The name of relation (outer bag).

block:

FOREACH…GENERATE block used with a relation (outer bag). Use this syntax:

```
alias = FOREACH alias GENERATE expression [AS schema] [expression [AS schema]….];
```

nested_block: Nested FOREACH...GENERATE block used with a inner bag. Use this syntax:

```
alias = FOREACH nested_alias {

    alias = {nested_op | nested_exp}; [{alias = {nested_op | nested_exp}; …]

    GENERATE expression [AS schema] [expression [AS schema]….]

};
```

Where:

The nested block is enclosed in opening and closing brackets { … }.

The GENERATE keyword must be the last statement within the nested block.

Pig - FOREACH

nested_op

- Allowed operations are CROSS, DISTINCT, FILTER, FOREACH, LIMIT, and ORDER BY.
- Note: FOREACH statements can be nested to two levels only. FOREACH statements that are nested to three or more levels will result in a grammar error.
- You can also perform projections within the nested block.

nested_exp

- Any arbitrary, supported expression.

schema

- 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.

Pig - FOREACH

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);
}
```

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

Example: Projection

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

```
X = FOREACH A GENERATE *;
```

```
DUMP X;
```

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

Pig - GROUP

Groups the data in one or more relations.

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

Syntax

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

Terms

alias The name of a relation.

ALL Keyword. Use ALL if you want all tuples to go to a single group; for example, when doing aggregates across entire relations.

BY Keyword. Use this clause to group the relation by field, tuple or expression.

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:

USING Keyword

PARTITION BY partitioner

PARALLEL n Increase the parallelism of a job by specifying the number of reduce tasks, n.

```
A = load 'student' AS (name:chararray,age:int,gpa:float);
B = GROUP A BY age;
C = FOREACH B GENERATE group, COUNT(A);
X = GROUP A BY f2*f3;
```

Pig - UNION

Computes the union of two or more relations.

Syntax

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

Terms

alias The name of a relation.

ONSCHEMA Use the ONSCHEMA clause to base the union on named fields (rather than positional notation). All inputs to the union must have a non-unknown (non-null) schema.

- 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.

```
UNION A, B;
```

Pig - STORE

Stores or saves results to the file system.

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

Terms
alias The name of a relation.
INTO Required keyword.
'directory' The name of the storage directory, in quotes. If the directory already exists, the STORE operation will fail. The output data files, named part-nnnnn, are written to this directory.
USING Keyword. Use this clause to name the store function. If the USING clause is omitted, the default store function PigStorage is used.
function The store function. You can use a built in function (see the Load/Store Functions). PigStorage is the default store function.

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

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

CAT myoutput;
```

Pig - Debugging

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

- Use the DUMP operator to display results to your terminal screen.
- Use the DESCRIBE operator to review the schema of a relation.
- Use the EXPLAIN operator to view the logical, physical, or mapreduce execution plans to compute a relation.
- Use the ILLUSTRATE operator to view the step-by-step execution of a series of statements.

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

```
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 ...
```

Pig - Arithmetic Operators

Description

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

Pig - Boolean Operators

Description

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

Pig - Comparison operators

Operator	Description	Example
"=="	Equal− Checks if the values of two operands are equal or not; if yes, then the condition becomes true.	(a = b) is not true
!=	Not Equal− Checks if the values of two operands are equal or not. If the values are not equal, then condition becomes true.	(a != b) is true.
>	Greater than− Checks if the value of the left operand is greater than the value of the right operand. If yes, then the condition becomes true.	(a > b) is not true.
<	Less than− Checks if the value of the left operand is less than the value of the right operand. If yes, then the condition becomes true.	(a < b) is true.
>=	Greater than or equal to− Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition becomes true.	(a >= b) is not true.
<=	Less than or equal to− Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true.	(a <= b) is true.
matches	Pattern matching− Checks whether the string in the left-hand side matches with the constant in the right-hand side.	f1 matches '.*text.*'

Pig - Comparison operators

Operator	Description	Example
"=="	Equal− Checks if the values of two operands are equal or not; if yes, then the condition becomes true.	(a = b) is not true
!=	Not Equal− Checks if the values of two operands are equal or not. If the values are not equal, then condition becomes true.	(a != b) is true.
>	Greater than− Checks if the value of the left operand is greater than the value of the right operand. If yes, then the condition becomes true.	(a > b) is not true.
<	Less than− Checks if the value of the left operand is less than the value of the right operand. If yes, then the condition becomes true.	(a < b) is true.
>=	Greater than or equal to− Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition becomes true.	(a >= b) is not true.
<=	Less than or equal to− Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true.	(a <= b) is true.
matches	Pattern matching− Checks whether the string in the left-hand side matches with the constant in the right-hand side.	f1 matches '.*text.*'

Pig - Construction operators

Operator	Description	Example
()	Tuple constructor operator− This operator is used to construct a tuple.	(Harry Potter, 27)
{}	Bag constructor operator− This operator is used to construct a bag.	{(Harry, 27), (Dumbledoor, 105)}
[]	Map constructor operator− This operator is used to construct a tuple.	[name#Nic Flamal, age#550]

Pig - More functions

DISTINCT

Removes duplicate tuples in a relation.

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

CUBE

Cube operation computes aggregates for all possible combinations of specified group by dimensions. The number of group by combinations generated by cube for n dimensions will be 2^n. Rollup operations computes multiple levels of aggregates based on hierarchical ordering of specified group by dimensions. Rollup is useful when there is hierarchical ordering on the dimensions. The number of group by combinations generated by rollup for n dimensions will be n+1.

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

JOIN (inner)

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

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

ORDER BY

Sorts a relation based on one or more fields.

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

RANK

Returns each tuple with the rank within a relation.

```
alias = RANK alias [ BY { * [ASC|DESC] | field_alias [ASC|DESC] [, field_alias [ASC|DESC] …] } [DENSE] ];
```

Pig - More functions

SAMPLE

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

```
SAMPLE alias size;
```

SPLIT

Partitions a relation into two or more relations.

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

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

STREAM

Sends data to an external script or program.

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

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

Pig - Eval functions

AVG

Computes 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 ignores NULL values.

Syntax

AVG(expression)

Terms

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

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

B = GROUP A BY name;

C = FOREACH B GENERATE A.name, AVG(A.gpa);

DUMP C;
```

Pig - Eval functions

BagToString

Concatenate the elements of a Bag into a chararray string, placing an optional delimiter between each value. BagToString creates a single string from the elements of a bag, similar to SQL's GROUP_CONCAT function.

Syntax

```
BagToString(vals:bag [, delimiter:chararray])
```

Terms

vals A bag of arbitrary values. They will each be cast to chararray if they are not already.

delimiter A chararray value to place between elements of the bag; defaults to underscore '_'.

```
team_parks = LOAD 'team_parks' AS (team_id:chararray,
park_id:chararray, years:bag{(year_id:int)});

-- BOS     BOS07   {(1995),(1997),(1996),(1998),(1999)}

-- NYA     NYC16   {(1995),(1999),(1998),(1997),(1996)}

-- NYA     NYC17   {(1998)}

-- SDN     HON01   {(1997)}

-- SDN     MNT01   {(1996),(1999)}

-- SDN     SAN01   {(1999),(1997),(1998),(1995),(1996)}

team_parkslist = FOREACH (GROUP team_parks BY team_id)
GENERATE  group AS team_id, BagToString(team_parks.park_id, ';');

-- BOS     BOS07

-- NYA     NYC17;NYC16

-- SDN     SAN01;MNT01;HON01
```

Pig - Eval functions

BagToTuple

Un-nests the elements of a bag into a tuple. BagToTuple creates a tuple from the elements of a bag. It removes only the first level of nesting; it does not recursively un-nest nested bags. Unlike FLATTEN, BagToTuple will not generate multiple output records per input record.

Syntax

```
BagToTuple(expression)
```

Terms

expression An expression with data type bag.

Examples

```
A = LOAD 'bag_data' AS (B1:bag{T1:tuple(f1:chararray)});

DUMP A;

({('a'),('b'),('c')})

({('d'),('e'),('f')})

X = FOREACH A GENERATE BagToTuple(B1);

DUMP X;

(('a','b','c'))

(('d','e','f'))
```

Pig - Eval functions

CONCAT

Concatenates two or more expressions of identical type. The result values of the expressions must have identical types. If any subexpression is null, the resulting expression is null.

Syntax

```
CONCAT (expression, expression, [...expression])
```

Terms

`expression` Any expression.

```
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(f1, '_', f2,f3);

DUMP X;

(apache_opensource)

(hadoop_mapreduce)

(pig_piglatin)
```

Pig - Eval functions

COUNT

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.

Syntax

`COUNT(expression)`

Terms

`expression` An expression with data type bag.

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

DUMP A;

(1,2,3)

(4,2,1)

(4,3,3)

(7,2,5)

B = GROUP A BY f1;

X = FOREACH B GENERATE COUNT(A);

DUMP X;

(1L)

(2L)

(1L)
```

Pig - Eval functions

DIFF

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.

Syntax

```
DIFF (expression, expression)
```

Terms

expression An expression with any data type.

```
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 GENERATE DIFF(B1,B2);

Dump x;

({(0,1),(1,1)})

({})

({(6,7),(2,2)})
```

Pig - Eval functions

IsEmpty

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

Syntax

IsEmpty(expression)

Terms

expression An expression with any data type.

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

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

/* do a cogroup of SSN with SSN_Name */

X = COGROUP SSN by ssn, SSN_NAME by ssn;



/* only keep those ssn's for which there is no name */

Y = filter X by IsEmpty(SSN_NAME);
```

Pig - Eval functions

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. The MAX function ignores NULL values.

Syntax

MAX(expression)

Terms

expression An expression with data types int, long, float, double, bigdecimal, biginteger, chararray, datetime or bytearray.

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

DUMP A;

(John,fl,3.9F)

(John,sp,4.0F)

(Mary,wt,3.9F)

(Mary,sp,4.0F)

(Mary,sm,4.0F)

B = GROUP A BY name;

X = FOREACH B GENERATE group, MAX(A.gpa);

DUMP X;

(John,4.0F)

(Mary,4.0F)
```

Pig - Eval functions

PluckTuple

Allows the user to specify a string prefix, and then filter for the columns in a relation that begin with that prefix or match that regex pattern. Optionally, include flag 'false' to filter for columns that do not match that prefix or match that regex pattern

Syntax

`DEFINE pluck PluckTuple(expression1)`

`DEFINE pluck PluckTuple(expression1,expression3)`

`pluck(expression2)`

Terms

`expression1` A prefix to pluck by or an regex pattern to pluck by

`expression2` The fields to apply the pluck to, usually '*'

`expression3` A boolean flag to indicate whether to include or exclude matching columns

```
a = load 'a' as (x, y);

b = load 'b' as (x, y);

c = join a by x, b by x;

DEFINE pluck PluckTuple('a::');

d = foreach c generate FLATTEN(pluck(*));

describe c;

c: {a::x: bytearray,a::y: bytearray,b::x: bytearray,b::y: bytearray}

describe d;

d: {plucked::a::x: bytearray,plucked::a::y: bytearray}

DEFINE pluckNegative PluckTuple('a::','false');

d = foreach c generate FLATTEN(pluckNegative(*));

describe d;

d: {plucked::b::x: bytearray,plucked::b::y: bytearray}
```

Pig - Eval functions

SIZE

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.

Syntax

SIZE(expression)

Terms

expression An expression with any data type.

```
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)
```

Pig - Eval functions

SUBTRACT

SUBTRACT takes two bags as arguments and returns a new bag composed of the tuples of first bag are not in the second bag.

If arguments are not bags, an IOException is thrown.

Syntax

```
SUBTRACT(expression, expression)
```

Terms

expression	An expression with data type bag.

```
A = LOAD 'bag_data' AS
(B1:bag{T1:tuple(t1:int,t2:int)},B2:bag{T2:tuple(f1:int,f2:int)});

DUMP A;

({(8,9),(0,1),(1,2)},{(8,9),(1,1)})

({(2,3),(4,5)},{(2,3),(4,5)})

({(6,7),(3,7),(3,7)},{(2,2),(3,7)})

DESCRIBE A;

A: {B1: {T1: (t1: int,t2: int)},B2: {T2: (f1: int,f2: int)}}

X = FOREACH A GENERATE SUBTRACT(B1,B2);

DUMP X;

({(0,1),(1,2)})

({})

({(6,7)})
```

Pig - Eval functions

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.

Syntax

SUM(expression)

Terms

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

```
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;

X = FOREACH B GENERATE group, SUM(A.pet_num);

DUMP X;
```

(Alice,8L)

(Bob,4L)

Pig - Eval functions

IN

IN operator allows you to easily test if an expression matches any value in a list of values. It is used to reduce the need for multiple OR conditions.

Syntax

IN (expression)

Terms

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

```
A = load 'data' using PigStorage(',') AS (id:int, first:chararray, last:chararray, gender:chararray);

DUMP A;

(1,Christine,Romero,Female)

(3,Albert,Rogers,Male)

(4,Kimberly,Morrison,Female)

(5,Eugene,Baker,Male)

(6,Ann,Alexander,Female)

X = FILTER A BY id IN (4, 6);

DUMP X;

(4,Kimberly,Morrison,Female)

(6,Ann,Alexander,Female)
```

Pig - Eval functions

TOKENIZE

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).

Syntax

```
TOKENIZE(expression [, 'field_delimiter'])
```

Terms

expression	An expression with data type chararray.
'field_delimiter'	An optional field delimiter (in single quotes). If field_delimiter is null or not passed, the following will be used as delimiters: space [ ], double quote ["], coma [,] parenthesis [()], star [*].

```
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.)})
```

Pig - Load/store functions

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 bz/bz2 FILES generated in this manner:

```
cat *.bz > text/concat.bz

cat *.bz2 > text/concat.bz2
```

If you use concatenated bzip files with your Pig jobs, you will NOT see a failure but the results will be INCORRECT.

Pig - Load/store functions

BinStorage

Pig uses BinStorage to load and store the temporary data that is generated between multiple MapReduce jobs.

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

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

Syntax

`BinStorage()`

And then later:

```
a = load 'b.txt' as (id, f);

b = group a by id;

store b into 'g' using BinStorage();
```

Pig - Load/store functions

JsonLoader, JsonStorage

Use JsonLoader to load JSON data. Use JsonStorage to store JSON data. Note that there is no concept of delimit in JsonLoader or JsonStorage. The data is encoded in standard JSON format. JsonLoader optionally takes a schema as the construct argument.

Syntax

```
JsonLoader( ['schema'] )
```

```
JsonStorage( )
```

Terms

schema An optional Pig schema, in single quotes.

```
a = load 'a.json' using
JsonLoader('a0:int,a1:{(a10:int,a11:chararray)},a2:(a20:double,a21:byte
array),a3:[chararray]');
```

```
a = load 'a.json' using JsonLoader();
```

Pig - Load/store functions

PigDump

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

Syntax

```
PigDump()
```

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

PigStorage

PigStorage is the default function used by Pig to load/store the data. PigStorage supports structured text files (in human-readable UTF-8 format) in compressed or uncompressed form (see Handling Compression). All Pig data types (both simple and complex) can be read/written using this function. The input data to the load can be a file, a directory or a glob.

Syntax

```
PigStorage( [field_delimiter] , ['options'] )
```

Terms

`field_delimiter` The default field delimiter is tab ('\t').

`'options'` A string that contains space-separated options ('optionA optionB optionC')

Currently supported options are:

`('schema')` - Stores the schema of the relation using a hidden JSON file.

`('noschema')` - Ignores a stored schema during the load.

`('tagsource')` - (deprecated, Use tagPath instead) Add a first column indicates the input file of the record.

`('tagPath')` - Add a first column indicates the input path of the record.

`('tagFile')` - Add a first column indicates the input file name of the record.

Pig - Load/store functions

TextLoader

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.

Syntax

TextLoader()

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

HBaseStorage

HBaseStorage stores and loads data from HBase. The function takes two arguments. The first argument is a space seperated list of columns. The second optional argument is a space seperated list of options. Column syntax and available options are listed below. Note that HBaseStorage always disable split combination.

Syntax

HBaseStorage('columns', ['options'])

Terms

columns

A list of qualified HBase columns to read data from or store data to. The column family name and column qualifier are seperated by a colon (:). Only the columns used in the Pig script need to be specified. Columns are specified in one of three different ways as described below.

Explicitly specify a column family and column qualifier (e.g., user_info:id).

Pig - Load/store functions

This will produce a scalar in the resultant tuple.

Specify a column family and a portion of column qualifier name as a prefix followed by an asterisk (i.e., user_info:address_*). This approach is used to read one or more columns from the same column family with a matching descriptor prefix. The datatype for this field will be a map of column descriptor name to field value. Note that combining this style of prefix with a long list of fully qualified column descriptor names could cause perfomance degredation on the HBase scan. This will produce a Pig map in the resultant tuple with column descriptors as keys.

Specify all the columns of a column family using the column family name followed by an asterisk (i.e., user_info:*). This will produce a Pig map in the resultant tuple with column descriptors as keys.

'options'

-loadKey=(true|false) Load the row key as the first value in every tuple returned from HBase (default=false)

-gt=minKeyVal Return rows with a rowKey greater than minKeyVal

-lt=maxKeyVal Return rows with a rowKey less than maxKeyVal

-regex=regex Return rows with a rowKey that match this regex on KeyVal

-gte=minKeyVal Return rows with a rowKey greater than or equal to minKeyVal

-lte=maxKeyVal Return rows with a rowKey less than or equal to maxKeyVal

-limit=numRowsPerRegion Max number of rows to retrieve per region

-caching=numRows Number of rows to cache (faster scans, more memory)

-delim=delimiter Column delimiter in columns list (default is whitespace)

-ignoreWhitespace=(true|false) When delim is set to something other than whitespace, ignore spaces when parsing column list (default=true)

Pig - Load/store functions

-minTimestamp=timestamp Return cell values that have a creation timestamp greater or equal to this value

-maxTimestamp=timestamp Return cell values that have a creation timestamp less than this value

-timestamp=timestamp Return cell values that have a creation timestamp equal to this value

-includeTimestamp=Record will include the timestamp after the rowkey on store (rowkey, timestamp, ...)

-includeTombstone=Record will include a tombstone marker on store after the rowKey and timestamp (if included) (rowkey, [timestamp,] tombstone, ...)

Load Example

```
raw = LOAD 'hbase://SomeTableName'

    USING org.apache.pig.backend.hadoop.hbase.HBaseStorage(

    'info:first_name info:last_name tags:work_* info:*', '-loadKey=true
-limit=5') AS

    (id:bytearray, first_name:chararray, last_name:chararray,
tags_map:map[], info_map:map[]);
```

Store Example

```
A = LOAD 'hdfs_users' AS (id:bytearray, first_name:chararray,
last_name:chararray);

STORE A INTO 'hbase://users_table' USING
org.apache.pig.backend.hadoop.hbase.HBaseStorage(

   'info:first_name info:last_name');
```

Pig - Math functions

ABS(expression)

To get the absolute value of an expression.

ACOS(expression)

To get the arc cosine of an expression.

ASIN(expression)

To get the arc sine of an expression.

ATAN(expression)

This function is used to get the arc tangent of an expression.

CBRT(expression)

This function is used to get the cube root of an expression.

CEIL(expression)

This function is used to get the value of an expression rounded up to the nearest integer.

COS(expression)

This function is used to get the trigonometric cosine of an expression.

COSH(expression)

This function is used to get the hyperbolic cosine of an expression.

Pig - Math functions

EXP(expression)

This function is used to get the Euler's number e raised to the power of x.

FLOOR(expression)

To get the value of an expression rounded down to the nearest integer.

LOG(expression)

To get the natural logarithm (base e) of an expression.

LOG10(expression)

To get the base 10 logarithm of an expression.

RANDOM( )

To get a pseudo random number (type double) greater than or equal to 0.0 and less than 1.0.

ROUND(expression)

To get 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).

SIN(expression)

To get the sine of an expression.

SQRT(expression)

To get the positive square root of an expression.

Pig - String functions

String Functions

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.

ENDSWITH

Tests inputs to determine if the first argument ends with the string in the second.

Syntax

```
ENDSWITH(string, testAgainst)
```

EqualsIgnoreCase

Compares two Strings ignoring case considerations.

Syntax

```
EqualsIgnoreCase(string1, string2)
```

INDEXOF

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

Syntax

```
INDEXOF(string, 'character', startIndex)
```

Pig - String functions

LAST_INDEX_OF

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

Syntax

```
LAST_INDEX_OF(string, 'character')
```

LCFIRST

Converts the first character in a string to lower case.

Syntax

```
LCFIRST(expression)
```

LOWER

Converts all characters in a string to lower case.

Syntax

```
LOWER(expression)
```

LTRIM

Returns a copy of a string with only leading white space removed.

Syntax

```
LTRIM(expression)
```

REGEX_EXTRACT

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

Syntax

```
REGEX_EXTRACT (string, regex, index)
```

Pig - String functions

REGEX_EXTRACT_ALL

Performs regular expression matching and extracts all matched groups.

Syntax

```
REGEX_EXTRACT_ALL (string, regex)
```

REGEX_SEARCH

Performs regular expression matching and searches all matched characters in a string.

Syntax

```
REGEX_SEARCH(string, 'regExp');
```

REPLACE

Replaces existing characters in a string with new characters.

Syntax

```
REPLACE(string, 'regExp', 'newChar');
```

RTRIM

Returns a copy of a string with only trailing white space removed.

Syntax

```
RTRIM(expression)
```

SPRINTF

Formats a set of values according to a printf-style template, using the native Java Formatter library.

Syntax

```
SPRINTF(format, [...vals])
```

Pig - String functions

REGEX_EXTRACT_ALL

Performs regular expression matching and extracts all matched groups.

Syntax

```
REGEX_EXTRACT_ALL (string, regex)
```

REGEX_SEARCH

Performs regular expression matching and searches all matched characters in a string.

Syntax

```
REGEX_SEARCH(string, 'regExp');
```

REPLACE

Replaces existing characters in a string with new characters.

Syntax

```
REPLACE(string, 'regExp', 'newChar');
```

RTRIM

Returns a copy of a string with only trailing white space removed.

Syntax

```
RTRIM(expression)
```

SPRINTF

Formats a set of values according to a printf-style template, using the native Java Formatter library.

Syntax

```
SPRINTF(format, [...vals])
```

Pig - String functions

STARTSWITH

Tests inputs to determine if the first argument starts with the string in the second.

Syntax

STARTSWITH(string, testAgainst)

STRSPLIT

Splits a string around matches of a given regular expression.

Syntax

STRSPLIT(string, regex, limit)

SUBSTRING

Returns a substring from a given string.

Syntax

SUBSTRING(string, startIndex, stopIndex)

TRIM

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

Syntax

TRIM(expression)

UPPER

Returns a string converted to upper case.

Syntax

UPPER(expression)

Pig - Datetime functions

AddDuration

Returns the result of a DateTime object plus a Duration object.

Syntax

`AddDuration(datetime, duration)`

CurrentTime

Returns the DateTime object of the current time.

Syntax

`CurrentTime()`

DaysBetween

Returns the number of days between two DateTime objects.

Syntax

`DaysBetween(datetime1, datetime2)`

GetDay

Returns the day of a month from a DateTime object.

Syntax

`GetDay(datetime)`

GetHour

Returns the hour of a day from a DateTime object.

Syntax

`GetHour(datetime)`

Pig - Datetime functions

GetMilliSecond

Returns the millisecond of a second from a DateTime object.

Syntax

`GetMilliSecond(datetime)`

GetMinute

Returns the minute of a hour from a DateTime object.

Syntax

`GetMinute(datetime)`

GetMonth

Returns the month of a year from a DateTime object.

Syntax

`GetMonth(datetime)`

GetSecond

Returns the second of a minute from a DateTime object.

Syntax

`GetSecond(datetime)`

GetWeek

Returns the week of a week year from a DateTime object.

Syntax

`GetWeek(datetime)`

Pig - Datetime functions

GetWeekYear

Returns the week year from a DateTime object.

Syntax

```
GetWeekYear(datetime)
```

GetYear

Returns the year from a DateTime object.

Syntax

```
GetYear(datetime)
```

HoursBetween

Returns the number of hours between two DateTime objects.

Syntax

```
HoursBetween(datetime1, datetime2)
```

MilliSecondsBetween

Returns the number of milliseconds between two DateTime objects.

Syntax

```
MilliSecondsBetween(datetime1, datetime2)
```

MinutesBetween

Returns the number of minutes between two DateTime objects.

Syntax

```
MinutesBetween(datetime1, datetime2)
```

Pig - Datetime functions

MonthsBetween

Returns the number of months between two DateTime objects.

Syntax

```
MonthsBetween(datetime1, datetime2)
```

SecondsBetween

Returns the number of seconds between two DateTime objects.

Syntax

```
SecondsBetween(datetime1, datetime2)
```

SubtractDuration

Returns the result of a DateTime object minus a Duration object.

Syntax

```
SubtractDuration(datetime, duration)
```

ToDate

Returns a DateTime object according to parameters.

Syntax

```
ToDate(milliseconds)
```

ToMilliSeconds

Returns the number of milliseconds elapsed since January 1, 1970, 00:00:00.000 GMT for a DateTime object.

Syntax

```
ToMilliSeconds(datetime)
```

Pig - Datetime functions

ToString

ToString converts the DateTime object to the ISO or the customized string.

Syntax

```
ToString(datetime [, format string])
```

ToUnixTime

Returns the Unix Time as long for a DateTime object. UnixTime is the number of seconds elapsed since January 1, 1970, 00:00:00.000 GMT.

Syntax

```
ToUnixTime(datetime)
```

WeeksBetween

Returns the number of weeks between two DateTime objects.

Syntax

```
WeeksBetween(datetime1, datetime2)
```

YearsBetween

Returns the number of years between two DateTime objects.

Syntax

```
YearsBetween(datetime1, datetime2)
```

Pig - Tuple/Bag/Map functions

TOTUPLE

Converts one or more expressions to type tuple.

Syntax

`TOTUPLE(expression [, expression ...])`

Terms

`expression` An expression of any datatype.

```
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))
```

Pig - Tuple/Bag/Map functions

TOBAG

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

Syntax

`TOBAG(expression [, expression ...])`

Terms

`expression`	An expression with any data type.

```
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)})
```

Pig - Tuple/Bag/Map functions

TOMAP

Use the TOMAP function to convert pairs of expressions into a map.
Note the following:

- You must supply an even number of expressions as parameters
- The elements must comply with map type rules:
- Every odd element (key-expression) must be a chararray since only chararrays can be keys into the map
- Every even element (value-expression) can be of any type supported by a map.

Syntax

TOMAP(key-expression, value-expression [, key-expression, value-expression ...])

Terms

key-expression An expression of type chararray.

value-expression An expression of any type supported by a map.

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

B = foreach A generate TOMAP(name, gpa);

store B into 'results';
```

Input (students)

```
joe smith 20 3.5

amy chen 22 3.2

leo allen 18 2.1
```

Output (results)

```
[joe smith#3.5]

[amy chen#3.2]

[leo allen#2.1]
```

Pig - Tuple/Bag/Map functions

TOP

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.

By default, TOP function uses descending order. But it can be configured via DEFINE statement.

```
DEFINE asc TOP('ASC'); -- ascending order
```

```
DEFINE desc TOP('DESC'); -- descending order
```

Syntax

```
TOP(topN,column,relation)
```

Terms

topN The number of top tuples to return (type integer).

column The tuple column whose values are being compared, note 0 denotes the first column.

relation The relation (bag of tuples) containing the tuple column.

```
DEFINE asc TOP('ASC'); -- ascending order

A = LOAD 'data' as (first: chararray, second: chararray);

B = GROUP A BY (first, second);

C = FOREACH B generate FLATTEN(group), COUNT(A) as count;

D = GROUP C BY first; -- again group by first

topResults = FOREACH D {result = asc(10, 1, C); -- and retain top 10 (in ascending order) occurrences of 'second' in first

   GENERATE FLATTEN(result);}

bottomResults = FOREACH D {    result = desc(10, 1, C); -- and retain top 10  (in descending order) occurrences of 'second' in first

   GENERATE FLATTEN(result);}
```

Pig - User Defined Functions

Pig provides extensive support for user defined functions (UDFs) as a way to specify custom processing.

Pig UDFs can currently be implemented in six languages: Java, Jython, Python, JavaScript, Ruby and Groovy.

The most extensive support is provided for Java functions. You can customize all parts of the processing including data load/store, column transformation, and aggregation. J

ava functions are also more efficient because they are implemented in the same language as Pig and because additional interfaces are supported such as the Algebraic Interface and the Accumulator Interface.

Limited support is provided for Jython, Python, JavaScript, Ruby and Groovy functions.

At runtime note that Pig will automatically detect the usage of a scripting UDF in the Pig script and will automatically ship the corresponding scripting jar, either Jython, Rhino, JRuby or Groovy-all, to the backend.

Pig also provides support for Piggy Bank, a repository for JAVA UDFs. Through Piggy Bank you can access Java UDFs written by other users and contribute Java UDFs that you have written.

Usage involves: writing the java class, compiling and creating jars, registering jar with pig, invoking the UDF in pig script, executing.

Pig - UDF types

- **Filter Functions** − The filter functions are used as conditions in filter statements. These functions accept a Pig value as input and return a Boolean value.
- **Eval Functions** − The Eval functions are used in FOREACH-GENERATE statements. These functions accept a Pig value as input and return a Pig result.
- **Algebraic Functions** − The Algebraic functions act on inner bags in a FOREACH GENERATE statement. These functions are used to perform full MapReduce operations on an inner bag.

Pig - writing Eval UDF

```
package myudfs;

import java.io.IOException;

import org.apache.pig.EvalFunc;

import org.apache.pig.data.Tuple;

public class UPPER extends EvalFunc<String>

{

  public String exec(Tuple input) throws IOException {

    if (input == null || input.size() == 0 || input.get(0) == null)

      return null;

    try{

      String str = (String)input.get(0);

      return str.toUpperCase();
```

Create the `myudfs` package.

`EvalFunc` class is the base class for all eval functions.

Return type of the UDF is a Java `String` in this case.

This function is invoked on every input tuple. The input into the function is a tuple with input parameters in the order they are passed to the function in the Pig script. In our example, it will contain a single string field corresponding to the student name.

The first thing to decide is what to do with invalid data. This depends on the format of the data. If the data is of type `bytearray` it means that it has not yet been converted to its proper type. In this case, if the format of the data does not match the expected type, a NULL value should be returned. If, on the other hand, the input data is of another type, this means that the conversion has already happened and the data should be in the correct format. This is the case with our example and that's why it throws an error (line 15.)

Now it needs to be compiled and included in a jar.

```
javac -cp pig.jar UPPER.java

jar -cf myudfs.jar myudfs
```

Pig - writing Eval UDF

```
-- myscript.pig

REGISTER myudfs.jar;

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

B = FOREACH A GENERATE myudfs.UPPER(name);

DUMP B;


pig -x local myscript.pig
```

Pig - Writing Algebraic UDF

```
public class COUNT extends EvalFunc<Long> implements Algebraic{

    public Long exec(Tuple input) throws IOException {return count(input);}

    public String getInitial() {return Initial.class.getName();}

    public String getIntermed() {return Intermed.class.getName();}

    public String getFinal() {return Final.class.getName();}

    static public class Initial extends EvalFunc<Tuple> {

        public Tuple exec(Tuple input) throws IOException {return
TupleFactory.getInstance().newTuple(count(input));}

    }

    static public class Intermed extends EvalFunc<Tuple> {

        public Tuple exec(Tuple input) throws IOException {return
TupleFactory.getInstance().newTuple(sum(input));}

    }

static public class Final extends EvalFunc<Long> {

        public Tuple exec(Tuple input) throws IOException {return sum(input);}

    }

    static protected Long count(Tuple input) throws ExecException {

        Object values = input.get(0);

        if (values instanceof DataBag) return ((DataBag)values).size();

        else if (values instanceof Map) return new Long(((Map)values).size());

    }

    static protected Long sum(Tuple input) throws ExecException,
NumberFormatException {

        DataBag values = (DataBag)input.get(0);

        long sum = 0;

        for (Iterator (Tuple) it = values.iterator(); it.hasNext();) {

            Tuple t = it.next();
```

Pig - Writing Algebraic UDF

An aggregate function is an eval function that takes a bag and returns a scalar value. One interesting and useful property of many aggregate functions is that they can be computed incrementally in a distributed fashion. We call these functions `algebraic`. `COUNT` is an example of an algebraic function because we can count the number of elements in a subset of the data and then sum the counts to produce a final output. In the Hadoop world, this means that the partial computations can be done by the map and combiner, and the final result can be computed by the reducer.

It is very important for performance to make sure that aggregate functions that are algebraic are implemented as such. Let's look at the implementation of the COUNT function to see what this means. (Error handling and some other code is omitted to save space.

Pig - UDF and schemas

Pig uses type information for validation and performance. It is important for UDFs to participate in type propagation. Our UDFs generally make no effort to communicate their output schema to Pig. This is because Pig can usually figure out this information by using Java's Reflection. If your UDF returns a scalar or a map, no work is required.

However, if your UDF returns a tuple or a bag (of tuples), it needs to help Pig figure out the structure of the tuple.

If a UDF returns a tuple or a bag and schema information is not provided, Pig assumes that the tuple contains a single field of type bytearray. If this is not the case, then not specifying the schema can cause failures.

Pig - More on UDFs

Schemas

Pig uses type information for validation and performance. It is important for UDFs to participate in type propagation. Our UDFs generally make no effort to communicate their output schema to Pig. This is because Pig can usually figure out this information by using Java's Reflection. If your UDF returns a scalar or a map, no work is required.

However, if your UDF returns a tuple or a bag (of tuples), it needs to help Pig figure out the structure of the tuple.

If a UDF returns a tuple or a bag and schema information is not provided, Pig assumes that the tuple contains a single field of type bytearray. If this is not the case, then not specifying the schema can cause failures.

Pig - More on UDFs

Error Handling

There are several types of errors that can occur in a UDF:

1. An error that affects a particular row but is not likely to impact other rows. An example of such an error would be a malformed input value or divide by zero problem. A reasonable handling of this situation would be to emit a warning and return a null value. `ABS` function in the next section demonstrates this approach. The current approach is to write the warning to `stderr`. Eventually we would like to pass a logger to the UDFs. Note that returning a NULL value only makes sense if the malformed value is of type `bytearray`. Otherwise the proper type has been already created and should have an appropriate value. If this is not the case, it is an internal error and should cause the system to fail. Both cases can be seen in the implementation of the `ABS` function in the next section.
2. An error that affects the entire processing but can succeed on retry. An example of such a failure is the inability to open a lookup file because the file could not be found. This could be a temporary environmental issue that can go away on retry. A UDF can signal this to Pig by throwing an `IOException` as with the case of the `ABS` function below.
3. An error that affects the entire processing and is not likely to succeed on retry. An example of such a failure is the inability to open a lookup file because of file permission problems. Pig currently does not have a way to handle this case. Hadoop does not have a way to handle this case either. It will be handled the same way as 2 above.

Pig - More on UDFs

Function Overloading

1. Before the type system was available in Pig, all values for the purpose of arithmetic calculations were assumed to be doubles as the safest choice. However, this is not very efficient if the data is actually of type integer or long. (We saw about a 2x slowdown of a query when using double where integer could be used.) Now that Pig supports types we can take advantage of the type information and choose the function that is most efficient for the provided operands.
2. UDF writers are encouraged to provide type-specific versions of a function if this can result in better performance. On the other hand, we don't want the users of the functions to worry about different functions - the right thing should just happen. Pig allows for this via a function table mechanism