Apache Pig

Apache Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs.

Apache Pig raises the level of abstraction for processing large datasets. MapReduce allows you, as the programmer, to specify a map function followed by a reduce function, but working out how to fit your data processing into this pattern, which often requires multiple MapReduce stages, can be a challenge. With Pig, the data structures are much

richer, typically being multivalued and nested, and the transformations you can apply to the data are much more powerful.

Pig is made up of two pieces:
• The language used to express data flows, called *Pig Latin*.
• The execution environment to run Pig Latin programs. There are currently two environments: local execution in a single JVM and distributed execution on a Hadoop cluster.

Need of Apache Pig

Writing MapReduce jobs in Java is not an easy task for everyone. Thus, Apache Pig emerged as a boon for programmers who were not good with Java or Python. Even if someone who knows Java and is good with MapReduce, they will also prefer Apache Pig due to the ease working with Pig.

Apache Pig vs MapReduce

Programmers face difficulty writing MapReduce tasks as it requires Java or Python programming knowledge. For them, Apache Pig is a savior.

- Pig Latin is a high-level data flow language, whereas MapReduce is a low-level data processing paradigm.
- Without writing complex Java implementations in MapReduce, programmers can achieve the same implementations very easily using Pig Latin.
- Apache Pig uses multi-query approach (i.e. using a single query of Pig Latin we can accomplish multiple MapReduce

tasks), which reduces the length of the code by 20 times. Hence, this reduces the development period by almost 16 times.
- Pig provides many built-in operators to support data operations like joins, filters, ordering, sorting etc. Whereas to perform the same function in MapReduce is a humongous task.
- Performing a Join operation in Apache Pig is simple. Whereas it is difficult in MapReduce to perform a Join operation between the data sets, as it requires multiple MapReduce tasks to be executed sequentially to fulfill the job.
- In addition, it also provides nested data types like tuples, bags, and maps that are missing from MapReduce. I will explain you these data types in a while.

Execution Types

Pig has two execution types or modes: local mode and MapReduce mode.

Local mode:

In local mode, Pig runs in a single JVM and accesses the local filesystem. This mode is suitable only for small datasets and when trying out Pig. The execution type is set using the -x or -exectype option. To run in local mode, set the option to local:

```
% pig -x local
```

MapReduce mode:

In MapReduce mode, Pig translates queries into MapReduce jobs and runs them on a Hadoop cluster. The cluster may be a pseudo- or fully distributed cluster. MapReduce mode (with a fully distributed cluster) is what you use when you want to run Pig on large datasets.

Once you have configured Pig to connect to a Hadoop cluster, you can launch Pig, setting the -x option to mapreduce or omitting it entirely, as MapReduce mode is the default.

```
% pig
```

Running Pig Programs

There are three ways of executing Pig programs, all of which work in both local and MapReduce mode:

1. ***Script :*** Pig can run a script file that contains Pig commands. For example, pig script.pig runs the commands in the local file *script.pig*. Alternatively, for very short scripts, you can use the -e option to run a script specified as a string on the command line.
2. ***Grunt :*** Grunt is an interactive shell for running Pig commands. Grunt is started when no file is specified for Pig to run and the -e option is not used. It is also possible to run Pig scripts from within Grunt using run and exec.
3. ***Embedded :*** You can run Pig programs from Java using the PigServer class, much like you can use JDBC to run SQL programs from Java. For programmatic access to Grunt, use PigRunner.

Pig Latin

Pig Latin is a data flow programming language used to query and analyzing large datasets.

Structure

A Pig Latin program consists of a collection of statements. A statement can be thought of as an operation or a command.[5] For example, a GROUP operation is a type of statement:

```
grouped_records = GROUP records BY year;
```

The command to list the files in a Hadoop filesystem is another example of a statement:

```
ls /
```

Pig Lating Commands:

Pig Latin commands

Hadoop filesystem	cat	Prints the contents of one or more files
	cd	Changes the current directory
	copyFromLocal	Copies a local file or directory to a Hadoop filesystem
	copyToLocal	Copies a file or directory on a Hadoop filesystem to the local filesystem
	cp	Copies a file or directory to another directory
	fs	Accesses Hadoop's filesystem shell

	ls	Lists files
	mkdir	Creates a new directory
	mv	Moves a file or directory to another directory
	Pwd	Prints the path of the current working directory
	Rm	Deletes a file or directory
	Rmf	Forcibly deletes a file or directory (does not fail if the file or directory does not exist)
Hadoop MapReduce	kill	Kills a MapReduce job
Utility	Clear	Clears the screen in Grunt
	exec	Runs a script in a new Grunt shell in batch mode
	Help	Shows the available commands and options
	history	Prints the query statements run in the current Grunt session
	quit	Exits the interpreter
	run	Runs a script within the existing Grunt shell
	Sets	Pig options and MapReduce job properties
	sh	Runs a shell command from within Grunt

Pig Latin Relational operators

Loading and storing	LOAD	Loads data from the filesystem or other storage into a relation
	STORE	Saves a relation to the filesystem or other storage
	DUMP (\d)	Prints a relation to the console
Filtering	FILTER	Removes unwanted rows from a relation
	DISTINCT	Removes duplicate rows from a relation
	FOREACH...GENERATE	Adds or removes fields to or from a relation
	MAPREDUCE	Runs a MapReduce job using a relation as input
	STREAM	Transforms a relation using an external program
	SAMPLE	Selects a random sample of a relation
	ASSERT	Ensures a condition is true for all rows in a relation; otherwise, fails
Grouping and joining	JOIN	Joins two or more relations
	COGROUP	Groups the data in two or more relations
	GROUP	Groups the data in a single relation
	CROSS	Creates the cross product of two or more relations
	CUBE	Creates aggregations for all combinations of specified columns in a relation
Sorting	ORDER	Sorts a relation by one or more fields
	RANK	Assign a rank to each tuple in a relation, optionally sorting by fields first
	LIMIT	Limits the size of a relation to a maximum number of tuples
Combining and splitting	UNION	Combines two or more relations into one
	SPLIT	Splits a relation into two or more relations

Pig Latin diagnostic operators

DESCRIBE (/de)	Prints a relation's schema
EXPLAIN (/e)	Prints the logical and physical plans
ILLUSTRATE (/i)	Shows a sample execution of the logical plan, using a generated subset of the input

Pig Latin macro and UDF statements

REGISTER	Registers a JAR file with the Pig runtime
DEFINE	Creates an alias for a macro, UDF, streaming script, or command specification
IMPORT	Imports macros defined in a separate file into a script

Pig Latin expressions

Constant	Literal	Constant value	1.0, 'a'
Field (by position)	*$n*	Field in position *n* (zero-based)	$0
Field (by name)	*f*	Field named *f*	Year
Field (disambiguate)	*r*::*f*	Field named *f* from relation *r* after grouping or joining	A::year
Projection	*c*.$*n*, *c*.*f*	Field in container *c* (relation, bag, or tuple) by position, by name	records.$0, records.year
Map lookup	*m*#*k*	Value associated with key *k* in map *m*	items#'Coat'
Cast	(*t*) *f*	Cast of field *f* to type *t*	(int) year
Arithmetic	*x* + *y*, *x* - *y*	Addition, subtraction	$1 + $2, $1 - $2
	x * *y*, *x* / *y*	Multiplication, division	$1 * $2, $1 / $2
	x % *y*	Modulo, the remainder of *x* divided by *y*	$1 % $2
	+*x*, -*x*	Unary positive, negation	+1, -1
Conditional	*x* ? *y* : *z*	Bincond/ternary; *y* if *x* evaluates to true, *z* otherwise	quality == 0 ? 0 : 1
	CASE	Multi-case conditional	CASE q WHEN 0 THEN 'good' ELSE 'bad' END
Comparison	*x* == *y*, *x* != *y*	Equals, does not equal	quality == 0, tempera ture != 9999
	x > *y*, *x* < *y*	Greater than, less than	quality > 0, quality < 10
	x >= *y*, *x* <= *y*	Greater than or equal to, less than or equal to	quality >= 1, quality <= 9

	x matches *y*	Pattern matching with regular Expression	quality matches '[01459]'
	x is null	Is null	temperature is null
	x is not null	Is not null	temperature is not null
Boolean	*x* OR *y*	Logical OR	q == 0 OR q == 1
	x AND *y*	Logical AND	q == 0 AND r == 0
	NOT *x*	Logical negation	NOT q matches '[01459]'
	IN *x*	Set membership	q IN (0, 1, 4, 5, 9)
Functional	*f n*(*f1*,*f2*,...)	Invocation of function *fn* on fields *f1*, *f2*, etc.	isGood(quality)
Flatten	FLATTEN(*f*)	Removal of a level of nesting from bags and tuples	FLATTEN(group)

Pig Latin types

Boolean	boolean	True/false value	True
Numeric	Int	True/false value	1
	Long	32-bit signed integer	1L
	Float	64-bit signed integer	1.0F
	Double	32-bit floating-point number	1.0
	Biginteger	64-bit floating-point number	'10000000000'
	bigdecimal	Arbitrary-precision integer	'0.11000100000000000000000001'
Text	Chararray	Character array in UTF-16 format	'a'
Binary	Bytearray	Byte array	
Temporal	Datetime	B Date and time with time zone	
Complex	Tuple	Sequence of fields of any type	(1,'pomegranate')
	Bag	Unordered collection of tuples, possibly with duplicates	{(1,'pomegranate'),(2)
	Map	Set of key-value pairs; keys must be character arrays, but values may be any type	['a'#'pomegranate']