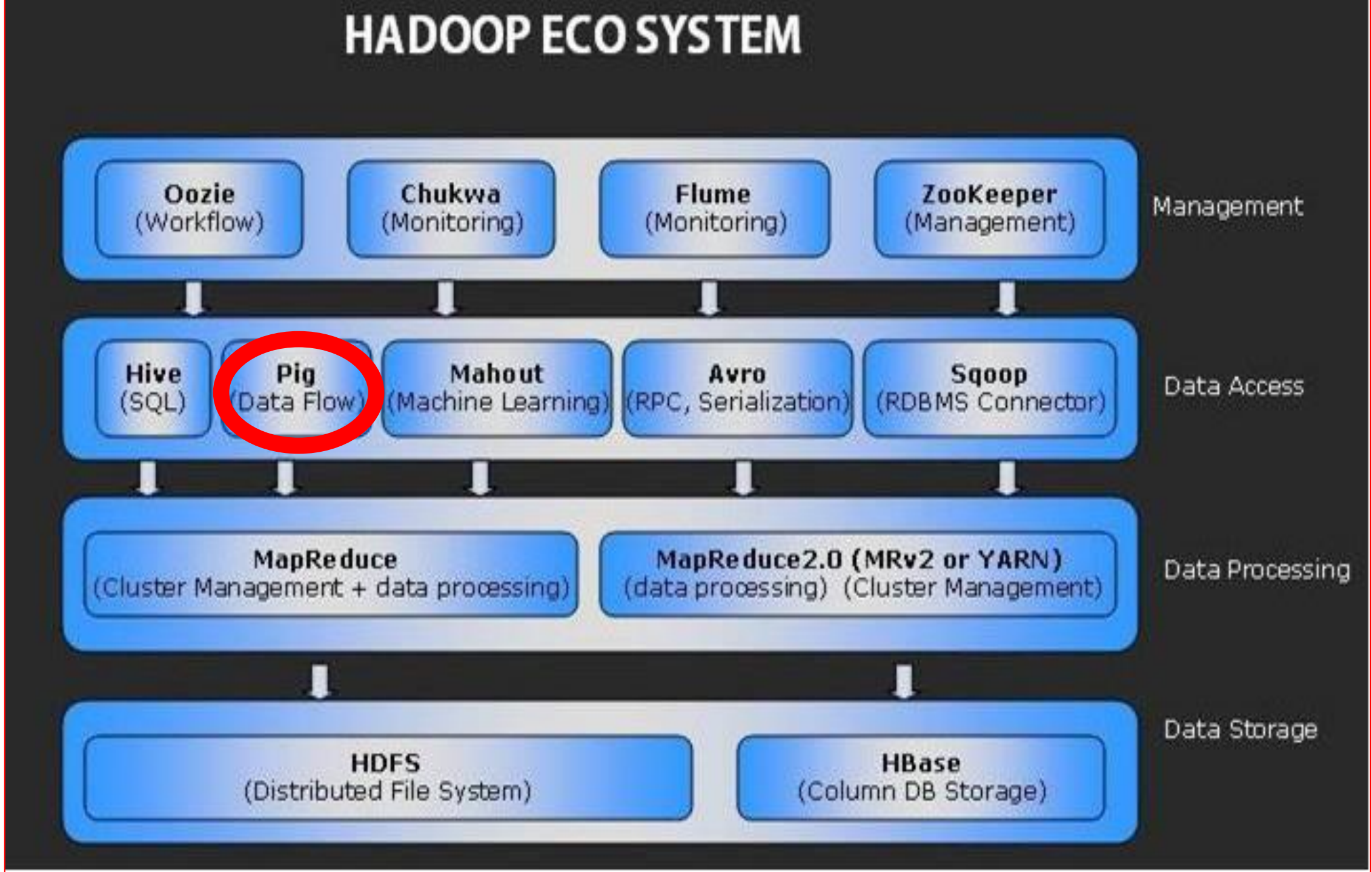
# Chapter 2

## PIG

# Topics

- Pig overview.
- Execution modes.
- Pig Latin Basics.
- Developing Pig Script.


HADOOP ECO SYSTEM
Oozie
(Workflow)
Chukwa
(Monitoring)
Flume
(Monitoring)
ZooKeeper
(Management)
Management
Hive
(SQL)
Pig
(Data Flow)
Mahout
(Machine Learning)
Avro
(RPC, Serialization)
Sqoop
(RDBMS Connector)
Data Access
MapReduce
(Cluster Management + data processing)
MapReduce2.0 (MRv2 or YARN)
(data processing) (Cluster Management)
Data Processing
HDFS
(Distributed File System)
HBase
(Column DB Storage)
Data Storage

# Pig Definition :

- 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.

- Pig is widely used and accepted by Yahoo!, Twitter, Netflix etc.

# Pig and Map reduce:

- Map reduce programmers need to think programmes in terms of map and reduce functions.
- It normally requires JAVA programmers.
- Pig requires its own scripting language named ‘Pig Latin’.

# Pig’s Features:

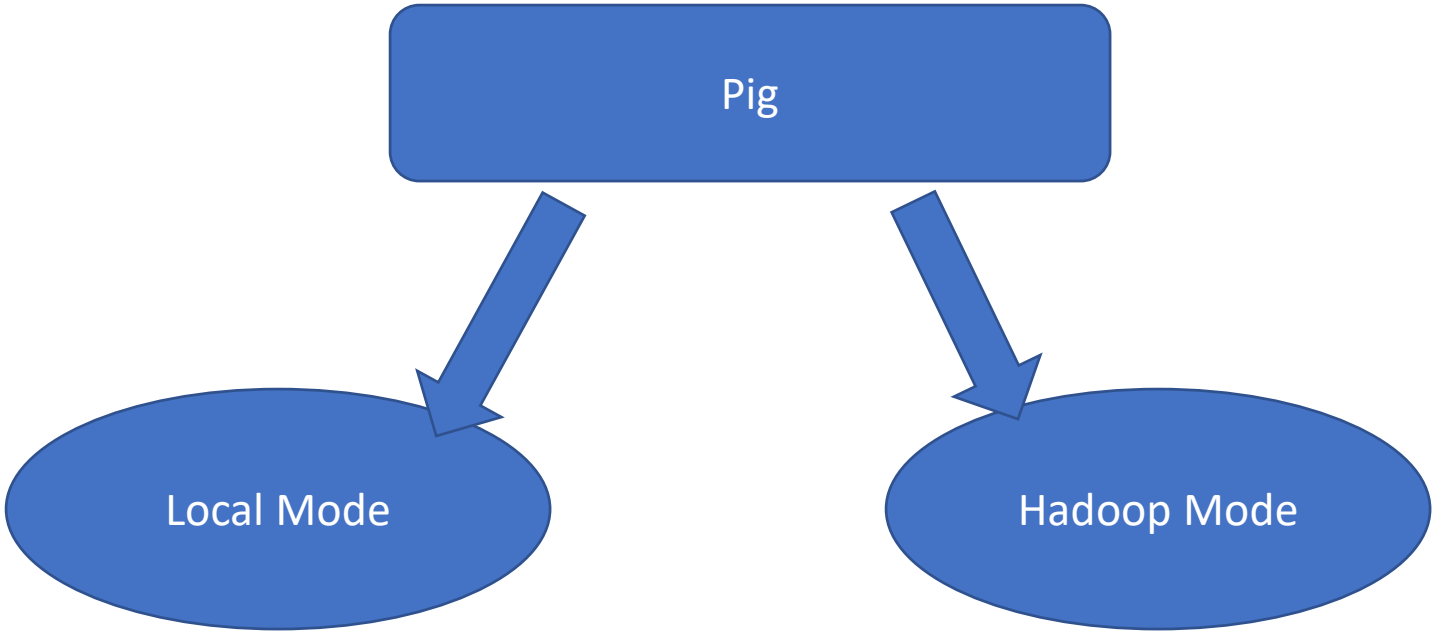
- Join Datasets.
- Sort Datasets.
- Data Types.
- Group By.
- Etc..

# Pig Components:

- **Pig Latin :**
- Command based language.
- Designed specifically for data transformation and flow expression**.**
- **Execution Environment :**
- The environment in which Pig Latin commands are executed .
- Currently it supports local and Hadoop modes.
- **Pig compiler converts Pig Latin into Map Reduce**
- Compiler strives to optimize execution.

# Execution Modes:

- There are 2 execution modes supported by Pig :




- To enter into different modes :

$Pig –x local  (for local mode)

$Pig –x mapreduce  (for hadoop mode)

# Pig Latin Concepts:

- Building Blocks :

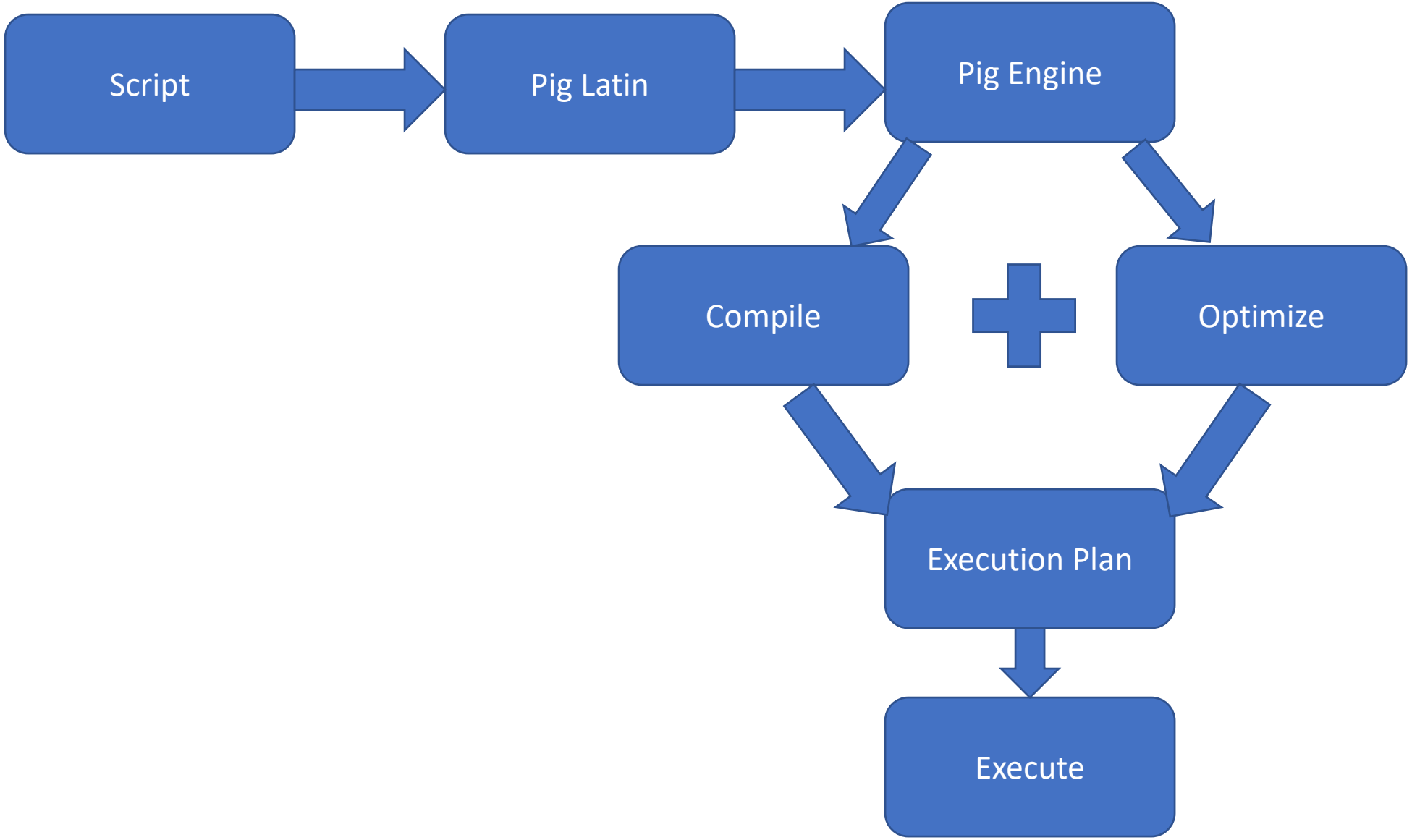
1) Field – Piece of data.
2) Tuple – ordered set of fields represented with “(“and“)” can also be referred as collection of fields.

Example : (abc, 12, M, 1000).

3) Bag – collection of tuples can be also represented with “{“and”}”.

Example : {(abc, 12, M, 1000), (xyz, 14, F, 2000)}.

# Working :


Script
Pig Latin
Pig Engine
Compile
Optimize
Execution Plan
Execute

# Schema Data Types:

| Type | Description | Example |
|---|---|---|
| | Simple | |
| int | Signed 32-bit integer | 10 |
| long | Signed 64-bit integer | 10L or 10l |
| float | 32-bit floating point | 10.5F or 10.5f |
| double | 64-bit floating point | 10.5 or 10.5e2 or 10.5E2 |
| | Arrays | |
| chararray | Character array (string) in Unicode UTF-8 | 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 | An collection of tuples | [open#apache] |

# Pig LOAD Command:

LOAD ‘data’ [USING Function] [AS schema];

- data – name of the directory or file usually in single inverted commas.
- USING – specifies the load function to use. By default uses PigStorage which parses each line into fields using delimiter as \t etc.
- AS – assigns a schema to incoming data.
- assigns names to fields.
- declares types to fields.

# Simple Pig Example:

- $ pig

Start grunt with default mapreduce mode.

```
grunt> cat /training/playarea/pig/a.txt
A  100
B  200
C  300
```

Create text file using cat command

```
grunt> records= LOAD '/training/playarea/pig/a.txt'
USING PigStorage() AS (letter:chararray, count:int);
```

Load data into records

```
grunt> dump records;
```

Display result using dump command

```
(A,100)
(B,200)
(C,300)
```

Output

# Dump and Store commands

No action will be taken unless dump and store commands are executed in Pig.

1) **Dump :**

Dump is used to display the results.

**2) Store :**

Store is used to save the results.

- Hadoop data is usually too large so it is not always feasible to print it to the screen so usually it gets stored in the Hadoop (HDFS, Hbase).

# Pig Latin - Grouping

```
grunt> chars = LOAD '/training/playarea/pig/b.txt' USING PigStorage()
   AS (c:chararray);
grunt> dump chars;
(a)
(b)
(k)
:
(k)
grunt> charGroup = GROUP chars by c;
grunt> dump charGroup;
(a,{(a),(a)})
(b,{(b),(b),(b)})
```

Creates a new bag with element named group and element named chars

The chars bag is grouped by "c" therefore 'group element will contain unique values

'chars' element is a bag itself and contains all tuples from 'chars' bag that match the values from 'c'

# Pig Latin Diagnostic tools:

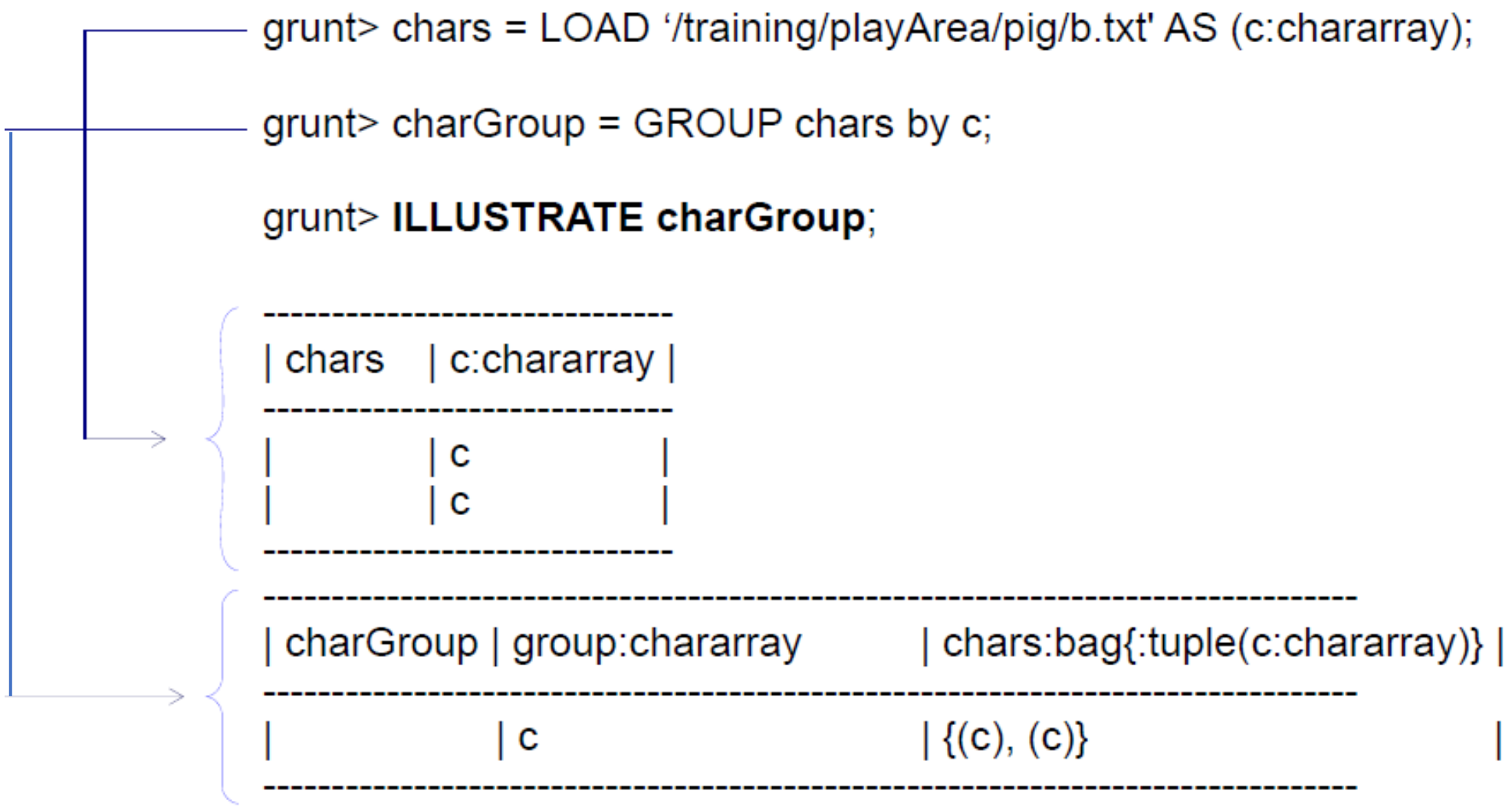
- Display the **structure of Bag**

grunt> DESCRIBE <bag_name>;

- Illustrate how Pig engine **transforms data**

grunt> ILLUSTRATE <bag_name>;

# ILLUSTRATE Command:



# Inner and Outer Bags:

```
grunt> chars = LOAD '/training/playArea/pig/b.txt' AS (c:chararray);
grunt> charGroup = GROUP chars by c;
grunt> ILLUSTRATE charGroup;
-----------------------------
| chars    | c:chararray    |
-----------------------------
|          | c              |
|          | c              |
-----------------------------
-------------------------------------------------------------------------
| charGroup      | group:chararray | chars:bag{:tuple(c:chararray)}    |
-------------------------------------------------------------------------
|                | c               | {(c), (c)}                        |
-------------------------------------------------------------------------
```

**Inner Bag**

**Outer Bag**

# Example of Inner Bag and Outer Bag:

```
grunt> chars = LOAD '/training/playArea/pig/b.txt' AS
(c:chararray);
grunt> charGroup = GROUP chars by c;
grunt> dump charGroup;
(a,{(a),(a),(a)})
(c,{(c),(c)})
(i,{(i),(i),(i)})
(k,{(k),(k),(k),(k)})
(l,{(l),(l)})
```

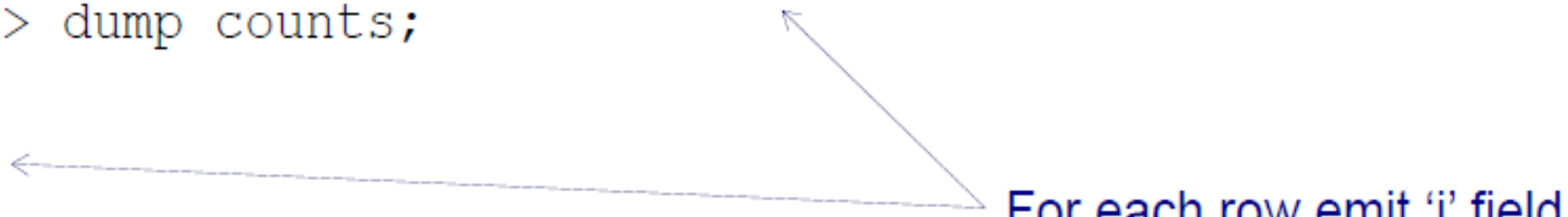
**Inner Bag**

**Outer Bag**

# Pig Latin - FOREACH

- FOREACH <bag> GENERATE <data>
- Iterates over each element in a bag and produces a result

```
grunt> records = LOAD 'data/a.txt' AS (c:chararray, i:int);
grunt> dump records;
(a,1)
(d,4)
(c,9)
(k,6)
grunt> counts = foreach records generate i;
grunt> dump counts;
(1)
(4)
(9)
(6)
```

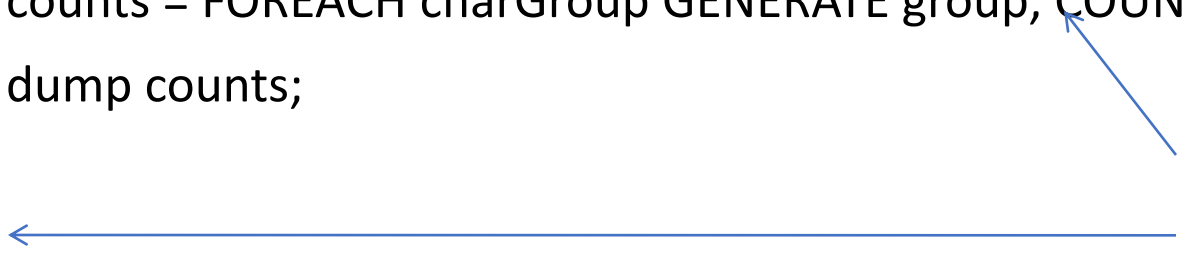


# FOREACH with Functions

- FOREACH B GENERATE group, FUNCTION(A);

Pig comes with many functions including COUNT, FLATTEN, CONCAT etc.

```
grunt> chars = LOAD ‘data/b.txt’ AS (c:chararray);
grunt> charGroup = GROUP chars by c;
grunt> dump charGroup;
(a,{(a),(a)})
(b,{(b),(b),(b)})
(c,{(c)})
grunt> counts = FOREACH charGroup GENERATE group, COUNT (chars);
grunt> dump counts;
(a,2)
(b,3)
(c,1)
```



For each row in ‘charGroup‘ bag emit group field and count the number of items in ‘chars’ bag

# TOKENIZE Function:

- Splits a string into tokens and outputs as a bag of tokens.

Seperators are space, comma(,), double quote(“”), parenthesis(()), star(*).

```
grunt> linesOfText = LOAD 'data/c.txt' AS (line:chararray);
grunt> dump linesOfText;
(this is a line of text)
(yet another line of text)
(third line of words)

grunt> tokenBag = FOREACH linesOfText GENERATE TOKENIZE(line);

grunt> dump tokenBag;
({(this),(is),(a),(line),(of),(text)})
({(yet),(another),(line),(of),(text)})
({(third),(line),(of),(words)})
grunt> describe tokenBag;
tokenBag: {bag_of_tokenTuples: {tuple_of_tokens: (token: chararray)}}
```

Split each row line by space and return a bag of tokens

Each row is a bag of words produced by TOKENIZE function

# Flatten Function:

Rearranges the output.

```
grunt> dump tokenBag;
({(this),(is),(a),(line),(of),(text)})
({(yet),(another),(line),(of),(text)})
({(third),(line),(of),(words)})
grunt> flatBag = FOREACH tokenBag GENERATE flatten($0);
grunt> dump flatBag;
(this)
(is)
(a)
...
...
(text)
(third)
(line)
(of)
(words)
```

Nested structure: bag of bags of tuples

Each row is flatten resulting in a bag of simple tokens

Elements in a bag can be referenced by index

# Pig Latin - JOINS

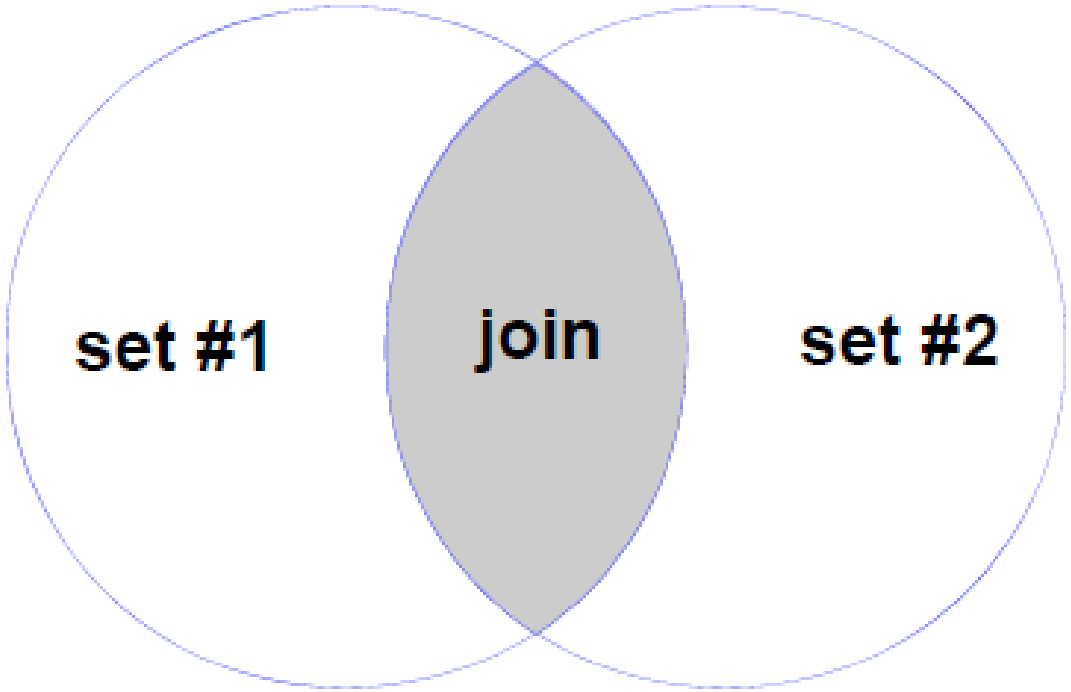
- Pig supports
- Inner Joins.
- Outer Joins.
- Full Joins.

- How to Join in Pig?

1. Load records into a bag from input #1.
2. Load records into a bag from input #2.
3. Join the two bags by provided join key.

# Inner Join:

- Inner Join is termed as a Default Join.
- Rows are joined where the keys match.



# Inner Join Example:

1) **Load records into a bag from input #1.**

grunt> posts = load ‘/training/data/user-posts.txt’ using PigStorage(‘,’) as (user:chararray, post:chararray, date:long);

2) **Load records into a bag from input #2.**

grunt> likes = load ‘/training/data/user-likes.txt’ using PigStorage(‘,’) as (user:chararray, likes:int, date:long);

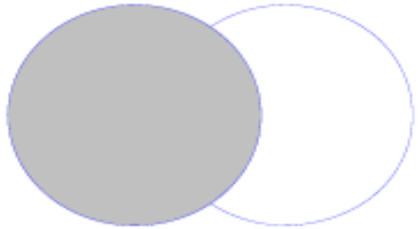
3) **Join the  data sets.**

grunt> userInfo = join posts by user, likes by user;

grunt> dump userInfo;

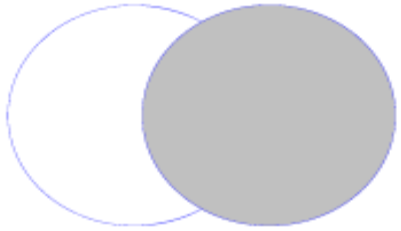
# Outer Join:

- Records which will not join with the other record set are still included in the result.



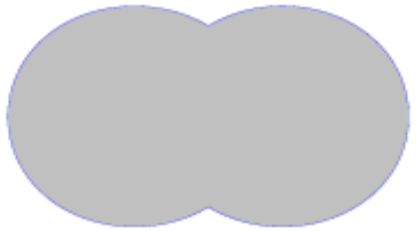
**Left Outer**

- Records from the first data-set are included whether they have a match or not. Fields from the unmatched (second) bag are set to null.



**Right Outer**

- The opposite of Left Outer Join: Records from the second data-set are included no matter what. Fields from the unmatched (first) bag are set to null.



**Full Outer**

- Records from both sides are included. For unmatched records the fields from the 'other' bag are set to null.

# Left Outer Join example:

1) **Load records into a bag from input #1.**

```
grunt> posts = load ‘/training/data/user-posts.txt’ using
  PigStorage(‘,’) as (user:chararray, post:chararray, date:long);
```

2) **Load records into a bag from input #2.**

```
grunt> likes = load ‘/training/data/user-likes.txt’ using
  PigStorage(‘,’) as (user:chararray, likes:int, date:long);
```

3) **Join the  data sets.**

```
grunt> userInfo = join posts by user LEFT OUTER, likes by user;

grunt> dump userInfo;
```

# Right Outer Join example:

1) **Load records into a bag from input #1.**

grunt> posts = load ‘/training/data/user-posts.txt’ using PigStorage(‘,’) as (user:chararray, post:chararray, date:long);

2) **Load records into a bag from input #2.**

grunt> likes = load ‘/training/data/user-likes.txt’ using PigStorage(‘,’) as (user:chararray, likes:int, date:long);

3) **Join the  data sets.**

grunt> userInfo = join posts by user RIGHT OUTER, likes by user;

grunt> dump userInfo;

# Full Outer Join example:

1) **Load records into a bag from input #1.**

grunt> posts = load ‘/training/data/user-posts.txt’ using PigStorage(‘,’) as (user:chararray, post:chararray, date:long);

2) **Load records into a bag from input #2.**

grunt> likes = load ‘/training/data/user-likes.txt’ using PigStorage(‘,’) as (user:chararray, likes:int, date:long);

3) **Join the  data sets.**

grunt> userInfo = join posts by user FULL OUTER, likes by user;

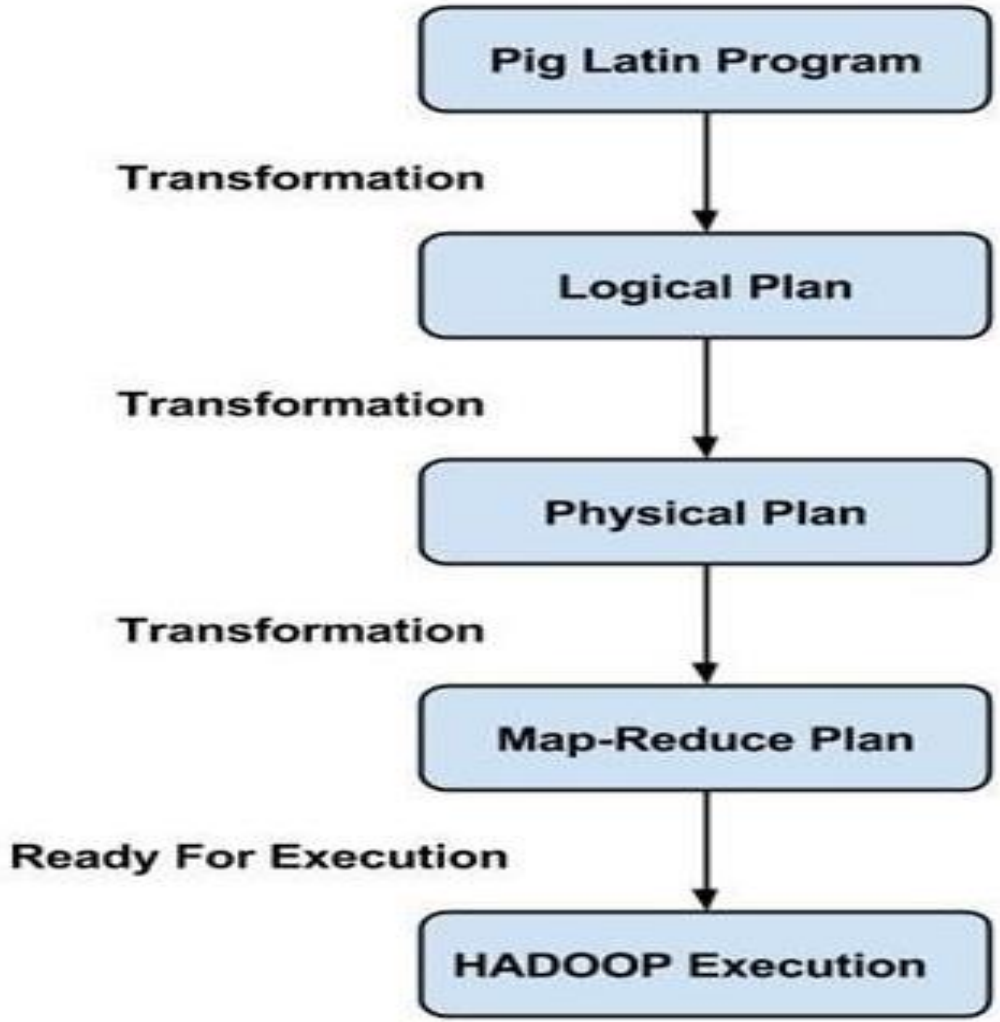
grunt> dump userInfo;

# Introduction

- Pig is a high-level programming language useful for analyzing large data sets. Pig was a result of development effort at Yahoo!
- In a MapReduce framework, programs need to be translated into a series of Map and Reduce stages. However, this is not a programming model which data analysts are familiar with. So, in order to bridge this gap, an abstraction called Pig was built on top of Hadoop.
- Apache Pig enables people to focus more on **analyzing bulk data sets and to spend less time writing Map-Reduce programs.** Similar to Pigs, who eat anything, the Apache Pig programming language is designed to work upon any kind of data.

# Pig Architecture

- The Architecture of Pig consists of two components:
- **Pig Latin,** which is a language
- **A runtime environment,** for running PigLatin programs.
- A Pig Latin program consists of a series of operations or transformations which are applied to the input data to produce output. These operations describe a data flow which is translated into an executable representation, by Hadoop Pig execution environment. Underneath, results of these transformations are series of MapReduce jobs which a programmer is unaware of. So, in a way, Pig in Hadoop allows the programmer to focus on data rather than the nature of execution.
- PigLatin is a relatively stiffened language which uses familiar keywords from data processing e.g., Join, Group and Filter.




PIG Architecture

# Execution modes:

- Pig in Hadoop has two execution modes:
- Local mode: In this mode, Hadoop Pig language runs in a single JVM and makes use of local file system. This mode is suitable only for analysis of small datasets using Pig in Hadoop
- Map Reduce mode: In this mode, queries written in Pig Latin are translated into MapReduce jobs and are run on a Hadoop cluster (cluster may be pseudo or fully distributed). MapReduce mode with the fully distributed cluster is useful of running Pig on large datasets.

# Pig Latin – Data Model

- A **Relation** is the outermost structure of the Pig Latin data model. And it is a **bag** where −
- A bag is a collection of tuples.
- A tuple is an ordered set of fields.
- A field is a piece of data.

- Pig Latin – Statemets
- While processing data using Pig Latin, **statements** are the basic constructs.
- These statements work with **relations**. They include **expressions** and **schemas**.
- Every statement ends with a semicolon (;).
- We will perform various operations using operators provided by Pig Latin, through statements.
- Except LOAD and STORE, while performing all other operations, Pig Latin statements take a relation as input and produce another relation as output.
- As soon as you enter a **Load** statement in the Grunt shell, its semantic checking will be carried out. To see the contents of the schema, you need to use the **Dump** operator. Only after performing the **dump** operation, the MapReduce job for loading the data into the file system will be carried out.

# Pig Latin – Data types

| S.N. | Data Type | Description & Example |
|---|---|---|
| 1 | int | Represents a signed 32-bit integer.<br>**Example** : 8 |
| 2 | long | Represents a signed 64-bit integer.<br>**Example** : 5L |
| 3 | float | Represents a signed 32-bit floating point.<br>**Example** : 5.5F |
| 4 | double | Represents a 64-bit floating point.<br>**Example** : 10.5 |
| 5 | chararray | Represents a character array (string) in Unicode UTF-8 format.<br>**Example** : 'tutorials point' |
| 6 | Bytearray | Represents a Byte array (blob). |

# Pig Latin – Data types

| | | |
|---|---|---|
| 7 | Boolean | Represents a Boolean value.<br>**Example** : true/ false. |
| 8 | Datetime | Represents a date-time.<br>**Example** : 1970-01-01T00:00:00.000+00:00 |
| 9 | Biginteger | Represents a Java BigInteger.<br>**Example** : 60708090709 |
| 10 | Bigdecimal | Represents a Java BigDecimal<br>**Example** : 185.98376256272893883 |
| **Complex Types** | | |
| 11 | Tuple | A tuple is an ordered set of fields.<br>**Example** : (raja, 30) |
| 12 | Bag | A bag is a collection of tuples.<br>**Example** : {(raju,30),(Mohhammad,45)} |
| 13 | Map | A Map is a set of key-value pairs.<br>**Example** : [ 'name'#'Raju', 'age'#30] |

# Thank You