BIG DATA ANALYTICS

Part – 4

"PIG"

By: Dr.Rashmi L Malghan &
Ms. Shavantrevva S Bilakeri

Agenda:

- Entry of Apache Pig
- Pig vs MapReduce
- Twitter Case Study on Apache Pig
- Apache Pig Architecture
- Pig Components
- Pig Data Model & Operators
- Running Pig Commands and Pig Scripts (Log Analysis)

Apache PIG

- Pig was introduced by yahoo and later on it was made fully open source by Apache Hadoop.

- It also provides a bridge to query data over Hadoop clusters but unlike hive, it implements a script implementation to make Hadoop data accessable by developers and business persons.

- Apache pig provides a high level programming platform for developers to process and analyses Big Data using user defined functions and programming efforts.

- In January 2013 Apache released Pig 0.10.1 which is defined for use with Hadoop 0.10.1 or later releases.

MapReduce Way:

In MapReduce, you need to write a program in Java/Python to process the data.

Apache PIG:

- Focus on the data transformations rather than the underlying MapReduce implementation.
- Apache Pig's high-level dataflow engine simplifies the development of large-scale data processing tasks on Hadoop clusters by providing an abstraction layer and leveraging the power of MapReduce without requiring users to write complex Java code.

Key Features and example of Pig Latin code

- Declarative Language (Pig Latin)
- Abstraction from MapReduce
- Data Flow Model
- Schema Flexibility
- Optimization Opportunities
- Ease of Use

```
-- Load data
data = LOAD 'input_data' USING PigStorage(',');

-- Filter data
filtered_data = FILTER data BY $1 > 50;

-- Group and aggregate
grouped_data = GROUP filtered_data BY $0;
result = FOREACH grouped_data GENERATE
group, AVG(filtered_data.$1);

-- Store the result
STORE result INTO 'output';
```

Apache PIG:

- An open-source high-level dataflow system
- Introduced by Yahoo
- Provides abstraction over MapReduce
- Two main components – the *Pig Latin* language and the *Pig Execution*

Fun Fact:

✓ ***10 lines of pig latin= approx. 200 lines of Map-Reduce Java Program***

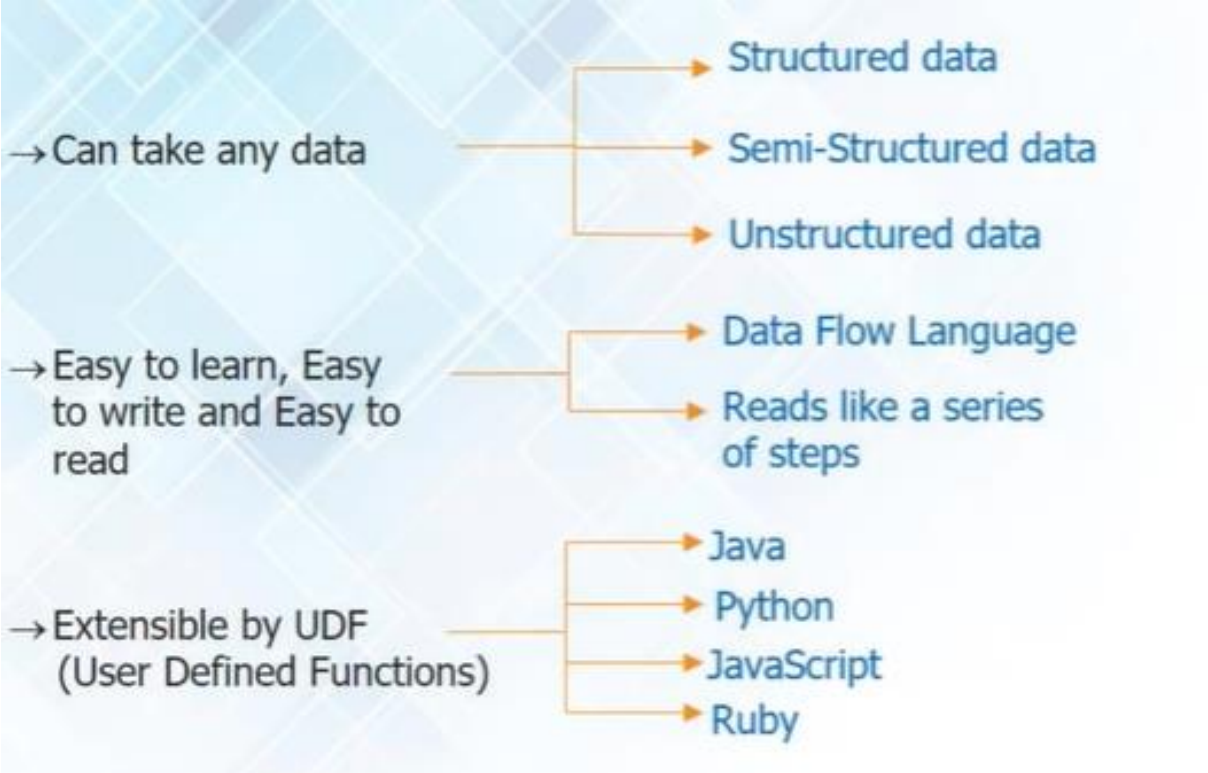
Why to Opt Pig instead of MapReduce:

PIG VS MapReduce

- High-level data flow tool
- No need to write complex programs
- Built-in support for data operations like joins, filters, ordering, sorting etc.
- Provides nested data types like tuples, bags, and maps

- Low-level data processing paradigm
- You need write programs in Java/Python etc.
- Performing data operations in MapReduce is a humongous task
- Nested data types are not there in MapReduce

Apache Pig: Advantages



→Provides common data operations filters, joins, ordering, etc. and nested data types tuples, bags, and maps missing from MapReduce.

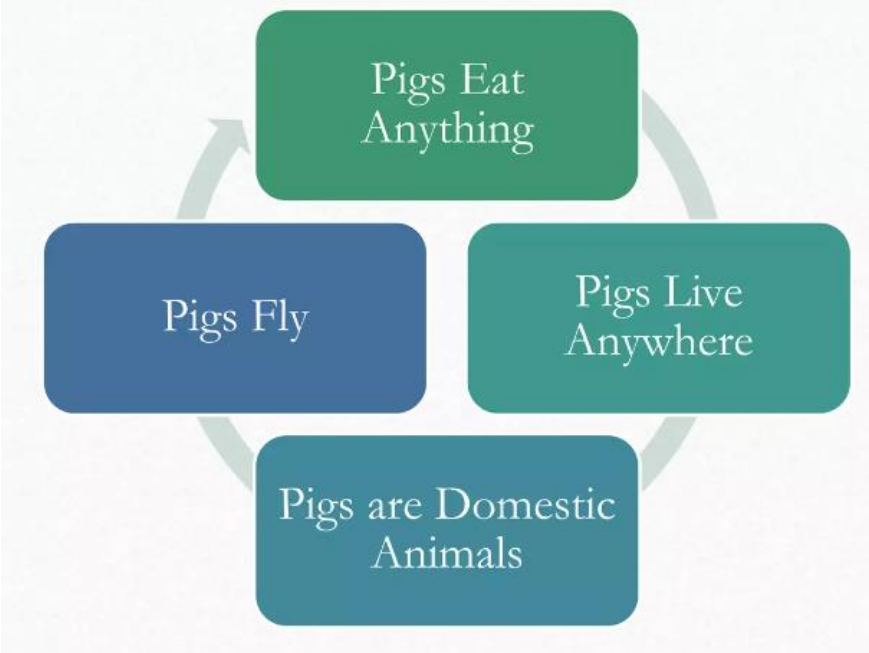
→An ad-hoc way of creating and executing map-reduce jobs on very large data sets

→Open source and actively supported by a community of developers.

PIG Anatomy

1. Data flow Language- **pig latin**
2. Interactive shell- **Grunt**
3. Prig interpreter and execution engine

PIG Philosophy



PIG supports:

1. HDFS commands
2. UNIX shell operators
3. Relational operators
4. Positional operators
5. Mathematical functions
6. User defined functions
7. Complex data structures

Apache Pig – Architecture / Pig MapReduce Engine

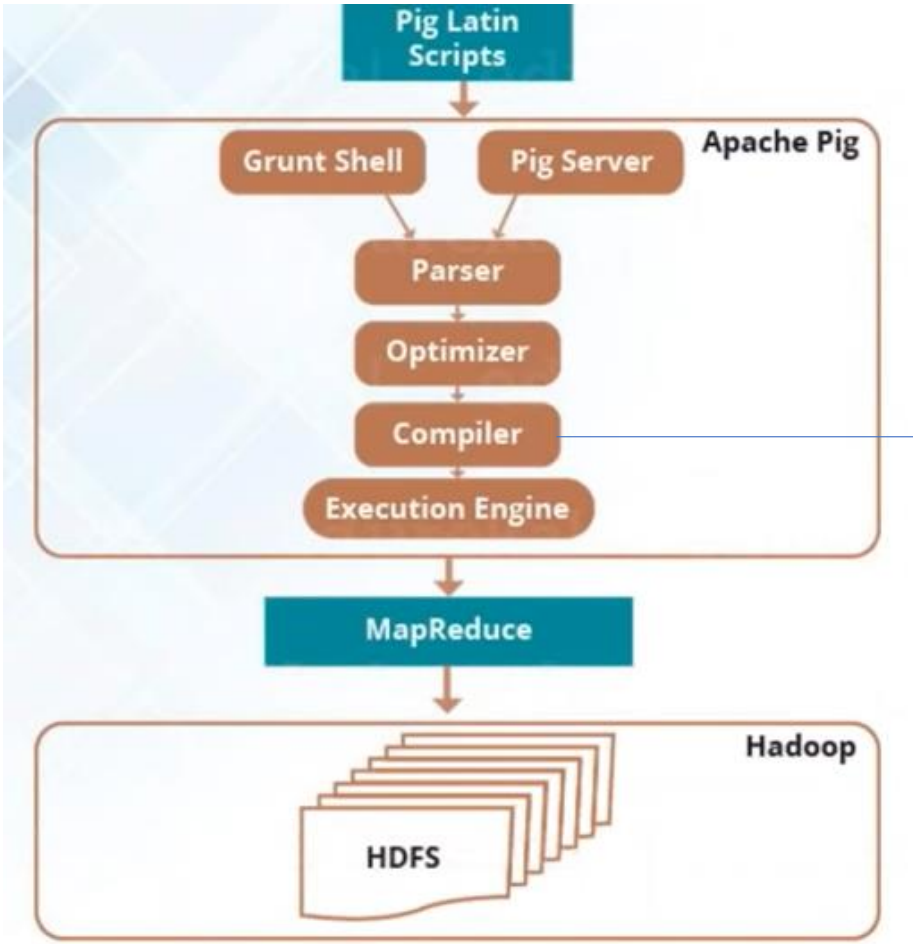


Figure: Apache Pig Architecture

- Pig Latin Scripts: Execute queries over big data.

- Grunt Shell: Native shell provided by Apache Pig, to execute pig queries.
- Submit pig scripts to java client to pig server & execute over Apache pig.

- Compiler: Converts pig latin scripts to Apache MapReduce code
- Executed using executing engine over Hadoop cluster

Operators in Apache Pig

Pig Latin Operators are the basic constructs that allow data manipulation in Apache Pig. Some commonly used operators include:

- LOAD and STORE: These operators are used to read and write data.
- FILTER: The FILTER operator is used to remove unwanted data based on a condition.
- GROUP: The GROUP operator is used to group the data in one or more relations.
- JOIN: The JOIN operator merges two or more relations.
- SPLIT: The SPLIT operator is used to split a single relation into two or more relations based on some condition.

Pig Syntax used for Data Processing

```
X= LOAD  ‘file name.txt’;     #directory name
……….

Y= GROUP ….;


……..
Z= FILTER …;


…..
DUMP Y;       #view result on screen
……..
STORE Z into ‘temp’
```

Example Latin Script: find the total distance travelled by a flight

```
-- Load the flight data
flight_data = LOAD 'path/to/flight_data' USING PigStorage(',') AS
(date:chararray, distance:int);

-- Filter out empty or invalid distance values
filtered_data = FILTER flight_data BY distance is not null and distance >= 0;

-- Calculate the total distance covered
total_distance = FOREACH (GROUP filtered_data ALL) GENERATE
SUM(filtered_data.distance) as total_distance;

-- Display the result
DUMP total_distance;
```

Question: load the student data (assuming data contains rollno, name, gpa),

romove the students whose gpa is less than 5.0.

From the filtered data, find the student name with highest gpa.

Display and Store the result to output file

SOLUTION

A = load ‘student’ (rollno, name, gpa) #A is a relational table not a variable

A = filter A by gpa>5.0

A = foreach A generate Upper (name);

STORE A INTO ‘myreport’

Note: by default columns are indexed with $01, $02, $03 when **USING PigStorage** is not used.

```
--Load student data
student_data = LOAD 'path/to/student_data' USING PigStorage(',')
AS (rollno:chararray, name:chararray, gpa:float);

 -- Filter out students with GPA less than 5.0
filtered_data = FILTER student_data BY gpa >= 5.0;
 -- Find the student with the highest GPA
 max_gpa_student = ORDER filtered_data BY gpa DESC; top_student
= LIMIT max_gpa_student 1;
 -- Display the result
DUMP top_student;
 -- Store the result in an output file
STORE top_student INTO 'path/to/output' USING PigStorage(',');
```

PIG LATIN IDENTIFIERS and COMMENTS

- Identifiers are the names assigned to field or the other data structures
- Should begin with a letter and should be followed only by letters and underscores
- Examples for valid**: Y, A1, A1_2014, Sample**
- Examples for invalid**: 5, sales$, sales%, _sales**
- Single line comment begin with **“--”**
- Multiline comment begin with **“/* and end with */”**

Case Study: Twitter

- **Objective:** To increase their user base / Enhance their offerings
- Procedures: To Extract insights monthly/weekly/daily
- **Results:** To scale up their infrastructure so that they will be able to handle larger user base they are targeting.

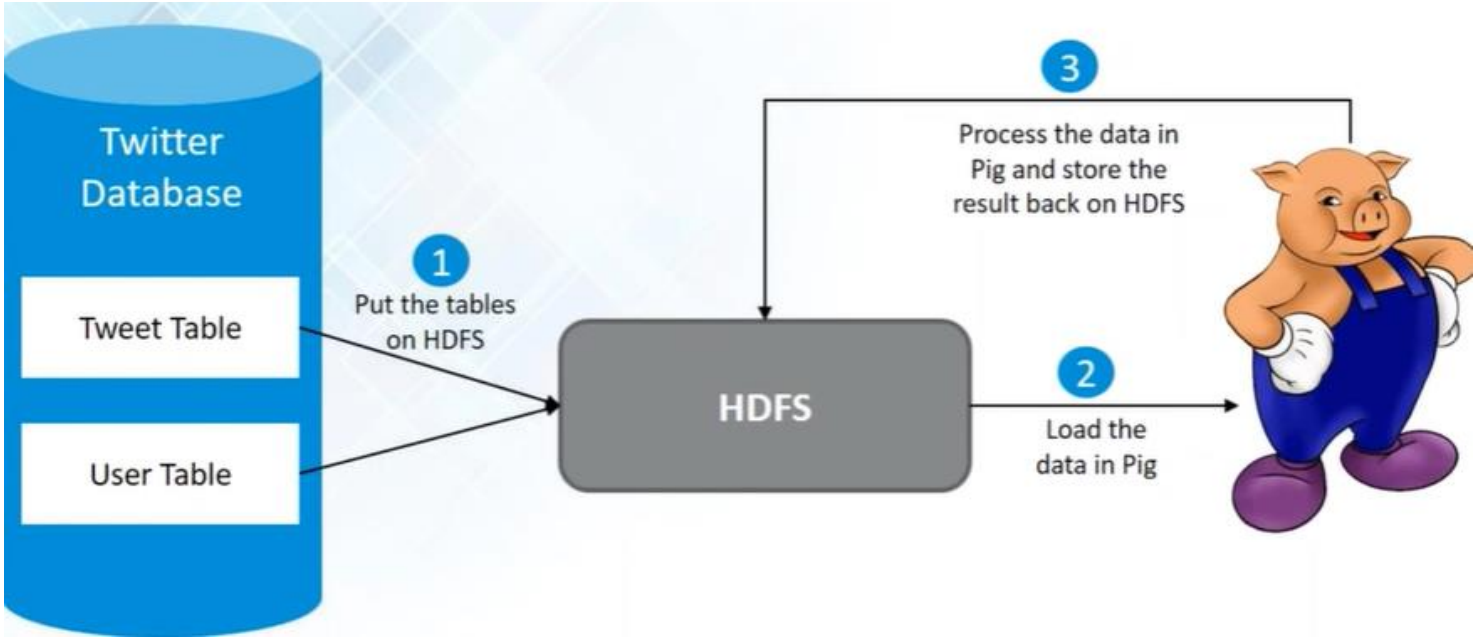
Case Study: Twitter

- Twitter's data was growing at an accelerating rate (i.e. 10 TB/day).
- Thus, Twitter decided to move the archived data to HDFS and adopt Hadoop for extracting the business values out of it.
- Their major aim was to analyse data stored in Hadoop to come up with the multiple insights on a daily, weekly or monthly basis.

Let me talk about one of the insight they wanted to know.

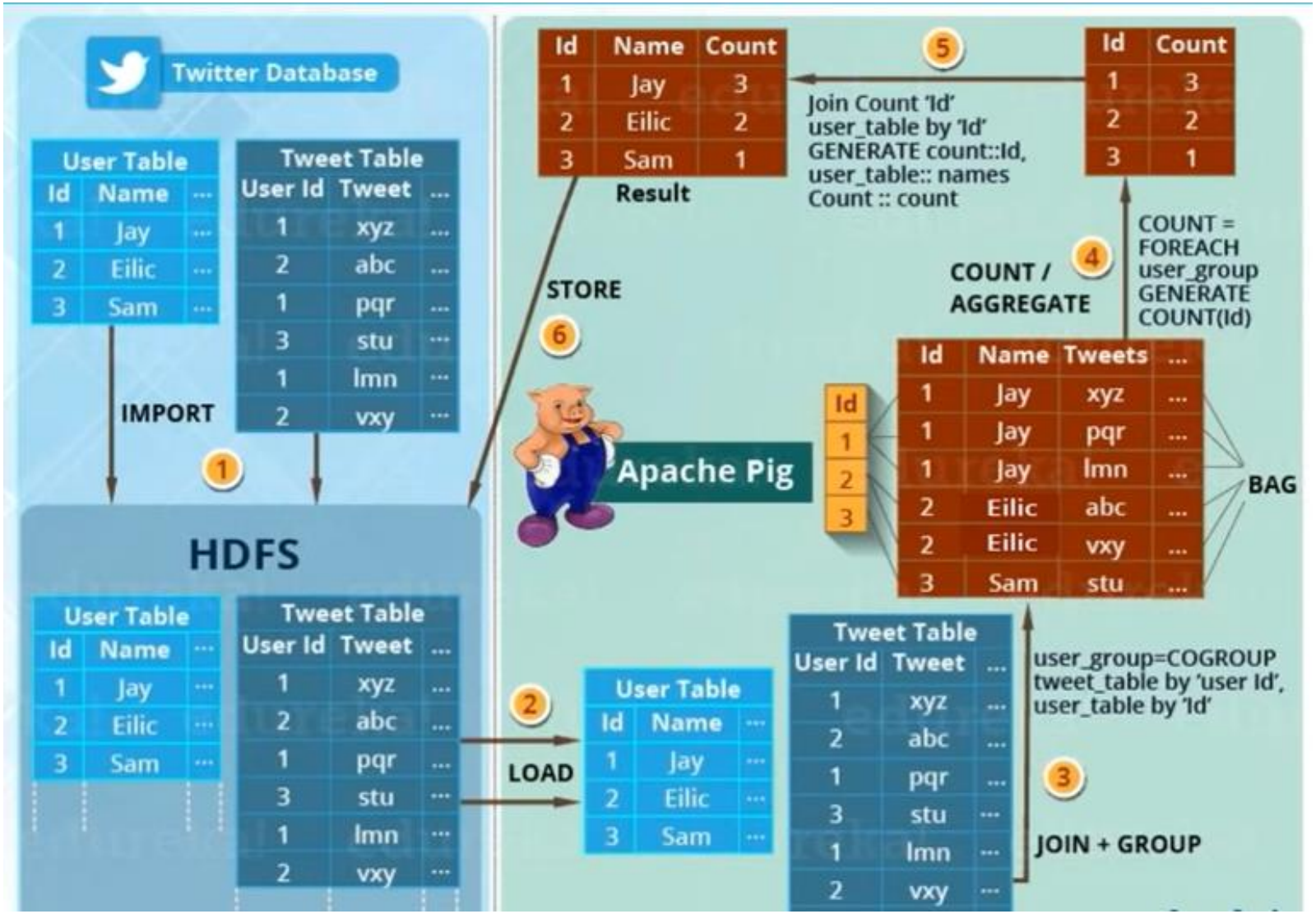
Analyzing how many tweets are stored per user, in the given tweet tables?

High Level implementation – HDFS & Pig



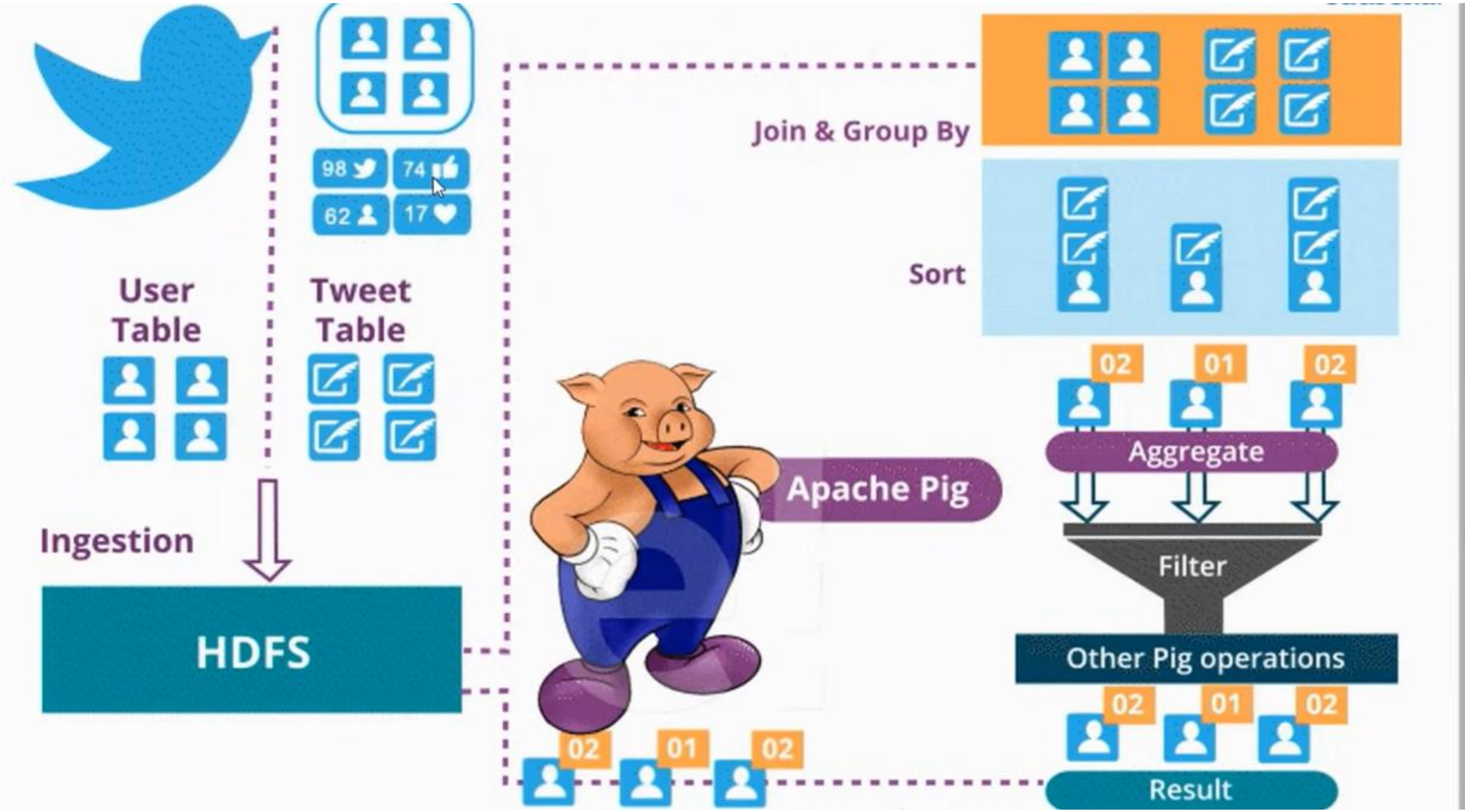
- Twitter Database had many tables, in which archive data was stored.
- The insight they want to extract was related to :Tweet & user table.

Implementation Flow - Detail



- **Import -Sqoop-** transfer data between RDBM to HDFS or HDFS to RDMS
- Twitter used Pig instead of mapreduce thus –saved their time and effort.

Case Study: Twitter



What happens underneath the covers when you run/submit a Sqoop import job

- Sqoop will connect to the database.
- Sqoop uses JDBC to examine the table by retrieving a list of all the columns and their SQL data types. These SQL types (varchar, integer and more) can then be mapped to Java data types (String, Integer etc.)
- Sqoop's code generator will use this information to create a table-specific class to hold records extracted from the table.
- Sqoop will connect to cluster and submit a MapReduce job.
- The dataset being transferred is sliced up into different partitions and a map-only job is launched with individual mappers responsible for transferring a slice of this dataset.
- For databases, Sqoop will read the table row-by-row into HDFS.
- For mainframe datasets, sqoop will read records from each mainframe dataset into HDFS.
- The output of this import process is a set of files containing a copy of imported table or datasets.
- The import process is performed in parallel for this reason, the output will be in multiple files.
- These files may be delimited text files CSV, TSV or binary Avro or Sequence files containing serialized record data. By default it is CSV.

Pig Latin: Case Sensitivity

- Keywords/ operators are not case sensitive.

Ex: LOAD, STORE, GROUP, FOREACH DUMP

- Relations and paths are case sensitive
- Function names are case sensitive Ex:

  PigStorage, COUNT

Complex Types		
11	Tuple	A tuple is an ordered set of fields. **Example** : (raja, 30)
12	Bag	A bag is a collection of tuples. **Example** : {(raju,30),(Mohhammad,45)}
13	Map	A Map is a set of key-value pairs. **Example** : ['name'#'Raju', 'age'#30]

S.N.	Data Type	Description & Example
1	int	Represents a signed 32-bit integer. **Example** : 8
2	long	Represents a signed 64-bit integer. **Example** : 5L
3	float	Represents a signed 32-bit floating point. **Example** : 5.5F
4	double	Represents a 64-bit floating point. **Example** : 10.5
5	chararray	Represents a character array (string) in Unicode UTF-8 format. **Example** : 'tutorials point'
6	Bytearray	Represents a Byte array (blob).
7	Boolean	Represents a Boolean value. **Example** : true/ false.
8	Datetime	Represents a date-time. **Example** : 1970-01-01T00:00:00.000+00:00
9	Biginteger	Represents a Java BigInteger. **Example** : 60708090709
10	Bigdecimal	Represents a Java BigDecimal **Example** : 185.98376256272893883

Pig Latin – Arithmetic Operators

Operator	Description	Example
+	**Addition** – Adds values on either side of the operator	a + b will give 30
–	**Subtraction** – Subtracts right hand operand from left hand operand	a – b will give –10
*	**Multiplication** – Multiplies values on either side of the operator	a * b will give 200
/	**Division** – Divides left hand operand by right hand operand	b / a will give 2
%	**Modulus** – Divides left hand operand by right hand operand and returns remainder	b % a will give 0
? :	**Bincond** – Evaluates the Boolean operators. It has three operands as shown below. variable **x** = (expression) ? **value1** if true : **value2** if false.	b = (a == 1)? 20: 30; if a = 1 the value of b is 20. if a!=1 the value of b is 30.
CASE WHEN THEN ELSE END	**Case** – The case operator is equivalent to nested bincond operator.	CASE f2 % 2 WHEN 0 THEN 'even' WHEN 1 THEN 'odd' END

Pig Latin – 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 '.*tutorial.*'

Pig Latin – Relational Operations

Operator	Description
Loading and Storing	
LOAD	To Load the data from the file system (local/HDFS) into a relation.
STORE	To save a relation to the file system (local/HDFS).
Filtering	
FILTER	To remove unwanted rows from a relation.
DISTINCT	To remove duplicate rows from a relation.
FOREACH, GENERATE	To generate data transformations based on columns of data.
STREAM	To transform a relation using an external program.
Grouping and Joining	
JOIN	To join two or more relations.
COGROUP	To group the data in two or more relations.
GROUP	To group the data in a single relation.
CROSS	To create the cross product of two or more relations.

Pig Latin – Relational Operations

Sorting	
ORDER	To arrange a relation in a sorted order based on one or more fields (ascending or descending).
LIMIT	To get a limited number of tuples from a relation.
Combining and Splitting	
UNION	To combine two or more relations into a single relation.
SPLIT	To split a single relation into two or more relations.
Diagnostic Operators	
DUMP	To print the contents of a relation on the console.
DESCRIBE	To describe the schema of a relation.
EXPLAIN	To view the logical, physical, or MapReduce execution plans to compute a relation.
ILLUSTRATE	To view the step-by-step execution of a series of statements.

Pig Latin – Type Construction Operators

Operator	Description	Example
()	**Tuple constructor operator** – This operator is used to construct a tuple.	(Raju, 30)
{}	**Bag constructor operator** – This operator is used to construct a bag.	{(Raju, 30), (Mohammad, 45)}
[]	**Map constructor operator** – This operator is used to construct a tuple.	[name#Raja, age#30]

Apache Pig - Diagnostic Operators

To verify the execution of the **Load** statement, you have to use the **Diagnostic Operators**.

1. Dump operator
2. Describe operator
3. Explanation operator
4. Illustration operator

Dump Operator

The **Dump** operator is used to run the Pig Latin statements and display the results on the screen. It is generally used for debugging Purpose.

syntax of the **Dump** operator:

grunt> Dump Relation_Name

Example

Assume we have a file **student_data.txt** in HDFS with the following content.

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

And we have read it into a relation **student** using the LOAD operator as shown below.

```
grunt> student = LOAD 'hdfs://localhost:9000/pig_data/student_data.txt'
   USING PigStorage(',')
   as ( id:int, firstname:chararray, lastname:chararray, phone:chararray,
   city:chararray );
```

Now, let us print the contents of the relation using the **Dump operator** as shown below.

```
grunt> Dump student
```

```
2015-10-01 15:05:27,642 [main]
INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLaun
100% complete
2015-10-01 15:05:27,652 [main]
INFO  org.apache.pig.tools.pigstats.mapreduce.SimplePigStats - Script Statistics:
HadoopVersion  PigVersion  UserId    StartedAt          FinishedAt      Features
2.6.0          0.15.0      Hadoop  2015-10-01 15:03:11  2015-10-01 05:27    UNKNOWN

Success!
Job Stats (time in seconds):

JobId          job_14459_0004
Maps               1
Reduces            0
MaxMapTime         n/a
MinMapTime         n/a
AvgMapTime         n/a
MedianMapTime      n/a
MaxReduceTime      0
MinReduceTime      0
AvgReduceTime      0
MedianReducetime   0
Alias              student
Feature            MAP_ONLY
Outputs            hdfs://localhost:9000/tmp/temp580182027/tmp757878456,
```

```
2015-10-01 15:06:28,485 [main]
INFO  org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths
to process : 1
2015-10-01 15:06:28,485 [main]
INFO  org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input pat
to process : 1

(1,Rajiv,Reddy,9848022337,Hyderabad)
(2,siddarth,Battacharya,9848022338,Kolkata)
(3,Rajesh,Khanna,9848022339,Delhi)
(4,Preethi,Agarwal,9848022330,Pune)
(5,Trupthi,Mohanthy,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,9848022335,Chennai)
```

Describe operator

- The **describe** operator is used to view the schema of a relation.

Syntax

grunt> Describe Relation_name

Example

grunt> describe student;

grunt> student: { id: int,firstname: chararray,lastname: chararray,phone: chararray,city: chararray }

ILLUSTRATE OPERATOR

The **illustrate** operator gives you the step-by-step execution of a sequence of statements.

Syntax

grunt> illustrate Relation_name;

```
grunt> illustrate student;

INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.PigMapOnly$M a
being processed per job phase (AliasName[line,offset]): M: student[1,10] C:  R:
------------------------------------------------------------------------------------
|student | id:int | firstname:chararray | lastname:chararray | phone:chararray | city:char
------------------------------------------------------------------------------------
|       | 002    | siddarth            | Battacharya        | 9848022338      | Kolkata      |
------------------------------------------------------------------------------------
```

GROUP operator

The **GROUP** operator is used to group the data in one or more relations. It collects the data having the same key.

Syntax

grunt> Group_data = GROUP Relation_name BY age;

Example

grunt> group_data = GROUP student_details by age;

(21,{(4,Preethi,Agarwal,21,9848022330,Pune),(1,Rajiv,Reddy,21,9848022337,Hydera bad)})
(22,{(3,Rajesh,Khanna,22,9848022339,Delhi),(2,siddarth,Battacharya,22,984802233 8,Kolkata)})
(23,{(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,9848022336 ,Bhuwaneshwar)})
(24,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,9848022334, trivendram)})

- The **COGROUP** operator works more or less in the same way as the GROUP operator.
- The only difference between the two operators is that the **group** operator is normally used with one relation, while the **cogroup** operator is used in statements involving two or more relations.

Grouping Two Relations using Cogroup

Assume that we have two files namely **student_details.txt** and **employee_details.txt** in the HDFS directory **/pig_data/** .

```
grunt> cogroup_data = COGROUP student_details by age, employee_details by age;
```

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

employee_details.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuwaneshwar
006,Maggy,22,Chennai
```

The UNION operator of Pig Latin is used to merge the content of two relations.

To perform UNION operation on two relations, their columns and domains must be identical.

Syntax: **grunt> Relation_name3 = UNION Relation_name1, Relation_name2;**

Student_data1.txt

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

Student_data2.txt

```
7,Komal,Nayak,9848022334,trivendram.
8,Bharathi,Nambiayar,9848022333,Chennai.
```

```
grunt> student = UNION student1,
student2;
```

Output

```
(1,Rajiv,Reddy,9848022337,Hyderabad) (2,siddarth,Battacharya,9848022338,Kolkata)
(3,Rajesh,Khanna,9848022339,Delhi)
(4,Preethi,Agarwal,9848022330,Pune)
(5,Trupthi,Mohanthy,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,9848022335,Chennai)
(7,Komal,Nayak,9848022334,trivendram)
(8,Bharathi,Nambiayar,9848022333,Chennai)
```

The **SPLIT** operator is used to split a relation into two or more relations.

Syntax: grunt> SPLIT Relation1_name INTO Relation2_name IF (condition1), Relation2_name (condition2)

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

Output

```
grunt> Dump student_details1;
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(4,Preethi,Agarwal,21,9848022330,Pune)

grunt> Dump student_details2;
(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,23,9848022335,Chennai)
(7,Komal,Nayak,24,9848022334,trivendram)
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
```

```
SPLIT student_details into student_details1 if
age<23, student_details2 if (22<age and age>25);
```

The **FILTER** operator is used to select the required tuples from a relation based on a condition.

Syntax

grunt> Relation2_name = FILTER Relation1_name BY (condition);

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

Output

(6,Archana,Mishra,23,9848022335,Chennai)
(8,Bharathi,Nambiayar,24,9848022333,Chennai)

```
filter_data = FILTER student_details BY
city == 'Chennai';
```

The **DISTINCT** operator is used to remove redundant (duplicate) tuples from a relation.

Syntax: grunt> Relation_name2 = DISTINCT Relatin_name1;

The **FOREACH** operator is used to generate specified data transformations based on the column data.

Syntax: grunt> Relation_name2 = FOREACH Relatin_name1 GENERATE (required data);

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

```
grunt> foreach_data = FOREACH
student_details GENERATE id,age,city;
```

Output

```
(1,21,Hyderabad)
(2,22,Kolkata)
(3,22,Delhi)
(4,21,Pune)
(5,23,Bhuwaneshwar)
(6,23,Chennai)
(7,24,trivendram)
(8,24,Chennai)
```

The **ORDER BY** operator is used to display the contents of a relation in a sorted order based on one or more fields.

Syntax

grunt> Relation_name2 = ORDER Relatin_name1 BY (ASC|DESC);

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

```
grunt> order_by_data = ORDER
student_details BY age DESC;
```

Output

```
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
(7,Komal,Nayak,24,9848022334,trivendram)
(6,Archana,Mishra,23,9848022335,Chennai)
(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(4,Preethi,Agarwal,21,9848022330,Pune)
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
```

The **LIMIT** operator is used to get a limited number of tuples from a relation.

Syntax

grunt> Result = LIMIT Relation_name required number of tuples;

Example:

```
grunt> limit_data = LIMIT student_details 4;
```

Apache Pig - Eval Functions

Apache Pig provides various built-in functions namely **eval, load, store, math, string, bag** and **tuple** functions.

S.N.	Function & Description
1	**AVG()** To compute the average of the numerical values within a bag.
2	**BagToString()** To concatenate the elements of a bag into a string. While concatenating, we can place a delimiter between these values (optional).
3	**CONCAT()** To concatenate two or more expressions of same type.
4	**COUNT()** To get the number of elements in a bag, while counting the number of tuples in a bag.
5	**COUNT_STAR()** It is similar to the **COUNT()** function. It is used to get the number of elements in a bag.
6	**DIFF()** To compare two bags (fields) in a tuple.
7	**IsEmpty()** To check if a bag or map is empty.

Apache Pig - Eval Functions

8	**MAX()** To calculate the highest value for a column (numeric values or chararrays) in a single-column bag.
9	**MIN()** To get the minimum (lowest) value (numeric or chararray) for a certain column in a single-column bag.
10	**PluckTuple()** Using the Pig Latin **PluckTuple()** function, we can define a string Prefix and filter the columns in a relation that begin with the given prefix.
11	**SIZE()** To compute the number of elements based on any Pig data type.
12	**SUBTRACT()** To subtract two bags. It takes two bags as inputs and returns a bag which contains the tuples of the first bag that are not in the second bag.
13	**SUM()** To get the total of the numeric values of a column in a single-column bag.
14	**TOKENIZE()** To split a string (which contains a group of words) in a single tuple and return a bag which contains the output of the split operation.

Apache Pig - Load & Store Functions

The **Load** and **Store** functions in Apache Pig are used to determine how the data goes ad comes out of Pig. These functions are used with the load and store operators. Given below is the list of load and store functions available in Pig.

S.N.	Function & Description
1	**PigStorage()** To load and store structured files.
2	**TextLoader()** To load unstructured data into Pig.
3	**BinStorage()** To load and store data into Pig using machine readable format.
4	**Handling Compression** In Pig Latin, we can load and store compressed data.

Apache Pig - Bag & Tuple Functions

S.N.	Function & Description
1	**TOBAG()** To convert two or more expressions into a bag.
2	**TOP()** To get the top **N** tuples of a relation.
3	**TOTUPLE()** To convert one or more expressions into a tuple.
4	**TOMAP()** To convert the key-value pairs into a Map.

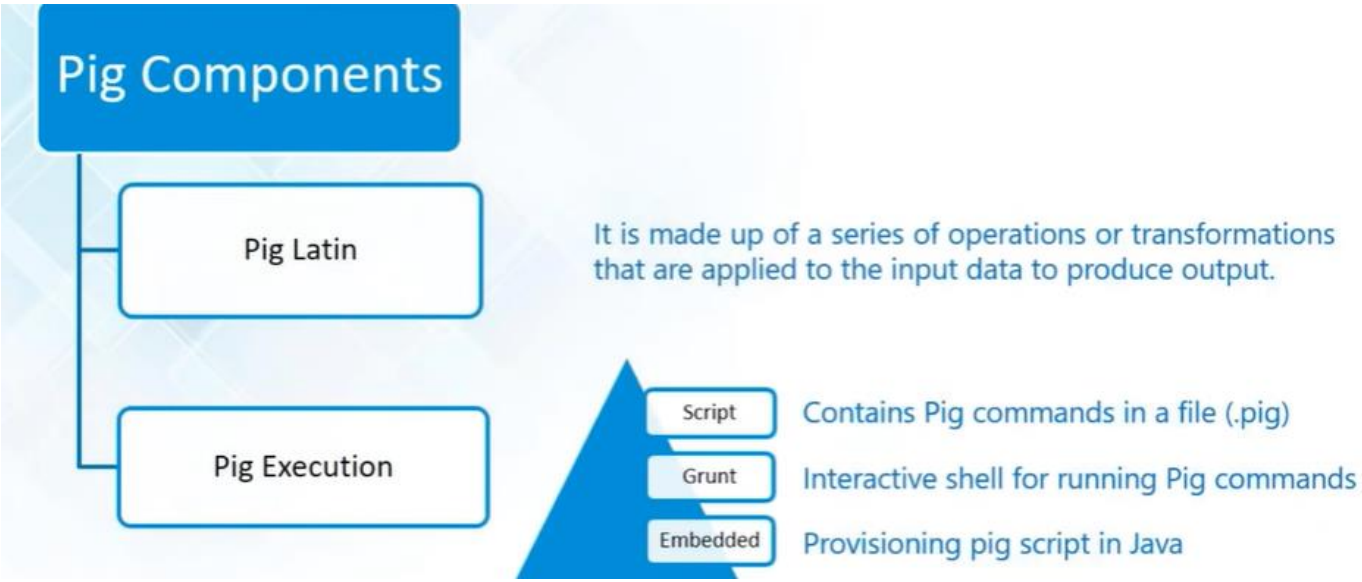
Apache Pig - String Functions

S.N.	Functions & Description
1	**ENDSWITH(string, testAgainst)** To verify whether a given string ends with a particular substring.
2	**STARTSWITH(string, substring)** Accepts two string parameters and verifies whether the first string starts with the second.
3	**SUBSTRING(string, startIndex, stopIndex)** Returns a substring from a given string.
4	**EqualsIgnoreCase(string1, string2)** To compare two stings ignoring the case.
5	**INDEXOF(string, 'character', startIndex)** Returns the first occurrence of a character in a string, searching forward from a start index.
6	**LAST_INDEX_OF(expression)** Returns the index of the last occurrence of a character in a string, searching backward from a start index.
7	**LCFIRST(expression)** Converts the first character in a string to lower case.
8	**UCFIRST(expression)** Returns a string with the first character converted to upper case.
9	**UPPER(expression)** UPPER(expression) Returns a string converted to upper case.
10	**LOWER(expression)** Converts all characters in a string to lower case.
11	**REPLACE(string, 'oldChar', 'newChar');** To replace existing characters in a string with new characters.
12	**STRSPLIT(string, regex, limit)** To split a string around matches of a given regular expression.
13	**STRSPLITTOBAG(string, regex, limit)** Similar to the **STRSPLIT()** function, it splits the string by given delimiter and returns the result in a bag.
14	**TRIM(expression)** Returns a copy of a string with leading and trailing whitespaces removed.
15	**LTRIM(expression)** Returns a copy of a string with leading whitespaces removed.
16	**RTRIM(expression)** Returns a copy of a string with trailing whitespaces removed.

Apache Pig - Date-time Functions

S.N.	Functions & Description
1	**ToDate(milliseconds)** This function returns a date-time object according to the given parameters. The other alternative for this function are ToDate(iosstring), ToDate(userstring, format), ToDate(userstring, format, timezone)
2	**CurrentTime()** returns the date-time object of the current time.
3	**GetDay(datetime)** Returns the day of a month from the date-time object.
4	**GetHour(datetime)** Returns the hour of a day from the date-time object.
5	**GetMilliSecond(datetime)** Returns the millisecond of a second from the date-time object.
6	**GetMinute(datetime)** Returns the minute of an hour from the date-time object.
7	**GetMonth(datetime)** Returns the month of a year from the date-time object.
8	**GetSecond(datetime)** Returns the second of a minute from the date-time object.
9	**GetWeek(datetime)** Returns the week of a year from the date-time object.
10	**GetWeekYear(datetime)** Returns the week year from the date-time object.
11	**GetYear(datetime)** Returns the year from the date-time object.
12	**AddDuration(datetime, duration)** Returns the result of a date-time object along with the duration object.
13	**SubtractDuration(datetime, duration)** Subtracts the Duration object from the Date-Time object and returns the result.
14	**DaysBetween(datetime1, datetime2)** Returns the number of days between the two date-time objects.
15	**HoursBetween(datetime1, datetime2)** Returns the number of hours between two date-time objects.
16	**MilliSecondsBetween(datetime1, datetime2)** Returns the number of milliseconds between two date-time objects.
17	**MinutesBetween(datetime1, datetime2)** Returns the number of minutes between two date-time objects.
18	**MonthsBetween(datetime1, datetime2)** Returns the number of months between two date-time objects.

Apache Pig - Components



- **Pig Latin :** Very simple data flow language given by Apache Pig .
- Write , transformation and analysis can be performed over input data set

- Various ways to execute Pig Scripts
- Embedded : Execute over pigserver.

Pig – Execution Modes

You can run Apache Pig in 2 modes:

MapReduce Mode – This is the default mode, which requires access to a Hadoop cluster and HDFS installation. The input and output in this mode are present on HDFS.

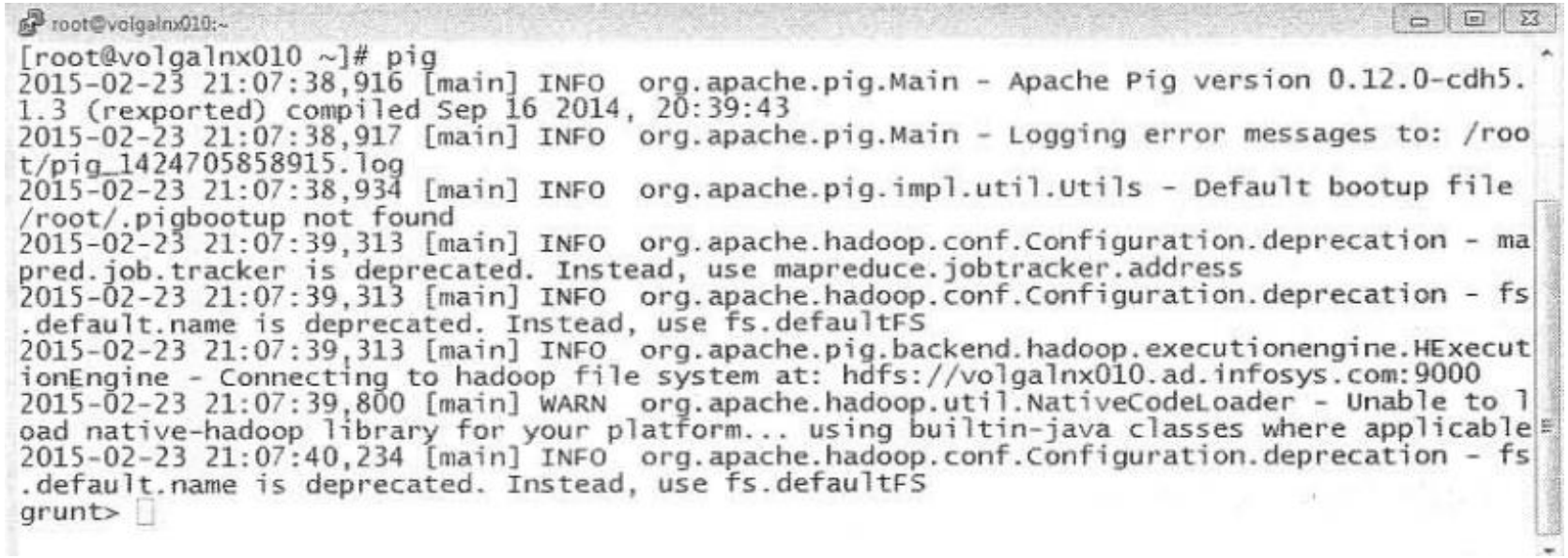
Command: pig

Local Mode – With access to a single machine, all files are installed and run using a local host and file system. Here the local mode is specified using '-x flag' (pig -x local). The input and output in this mode are present on local file system.

Command: pig –x local

Running PIG

1. Interactive mode: Run pig in interactive mode by invoking **grunt** shell.



2. Batch mode: Create **pig script** to run in batch mode. Write pig latin statements in a file and save it **with .pig extension**

Executing Pig Script in Batch mode

While executing Apache Pig statements in batch mode, follow the steps given below.

Step 1

Write all the required Pig Latin statements in a single file. We can write all the Pig Latin statements and commands in a single file and save it as **.pig** file.

Step 2

Execute the Apache Pig script. You can execute the Pig script from the shell (Linux) as shown below.

Local mode	MapReduce mode
$ pig -x local **Sample_script.pig**	$ pig -x mapreduce **Sample_script.pig**

You can execute it from the Grunt shell as well using the exec command as shown below.

```
grunt> exec /sample_script.pig
```

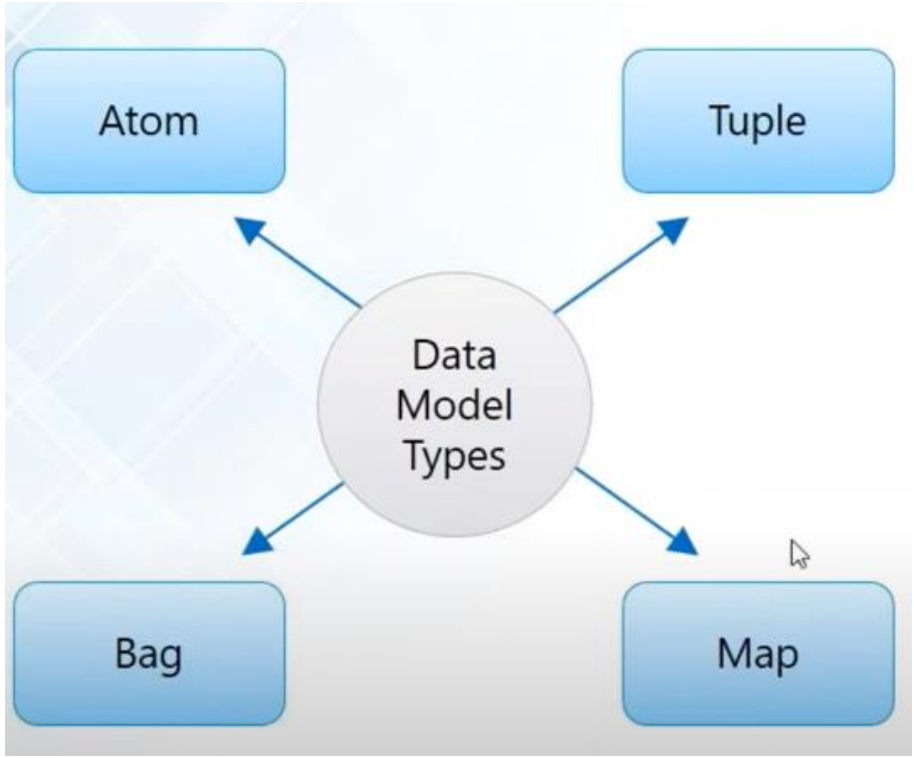
Executing a Pig Script from HDFS

We can also execute a Pig script that resides in the HDFS.

Suppose there is a Pig script with the name **Sample_script.pig** in the HDFS directory named **/pig_data/**. We can execute it as shown below.

```
$ pig -x mapreduce hdfs://localhost:9000/pig_data/Sample_script.pig
```

Data Model : Pig



Pig Data Model: Bag & Tuple

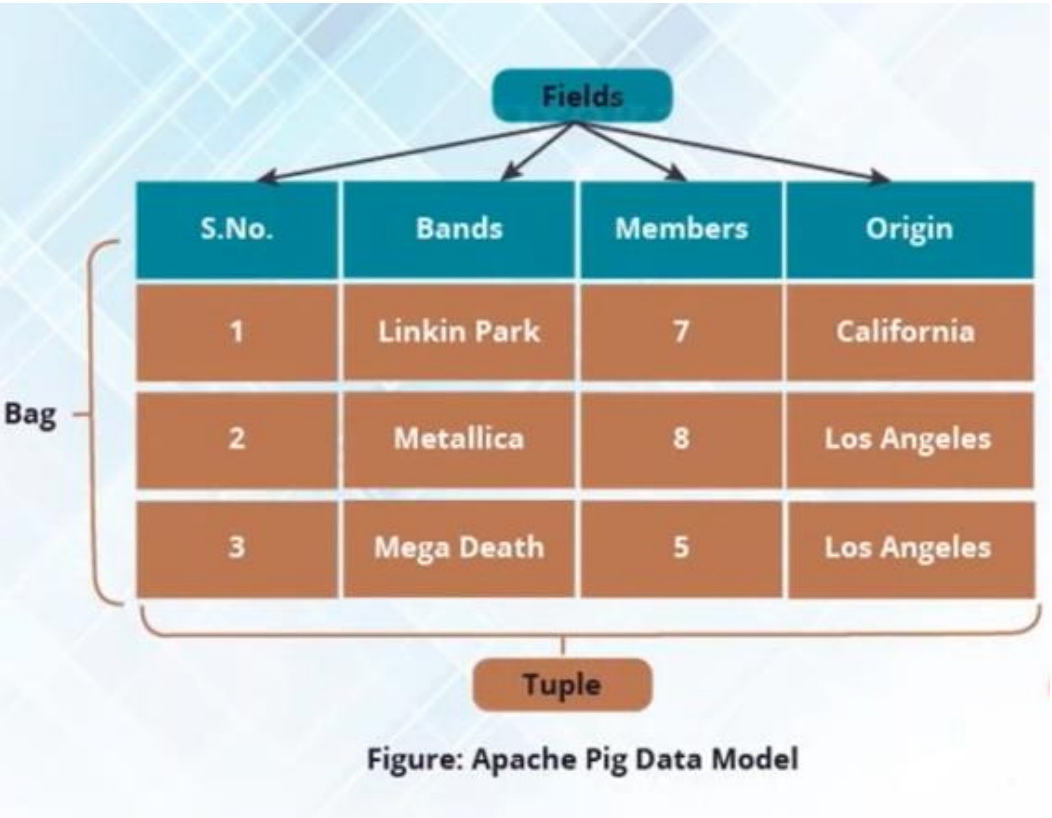


Figure: Apache Pig Data Model

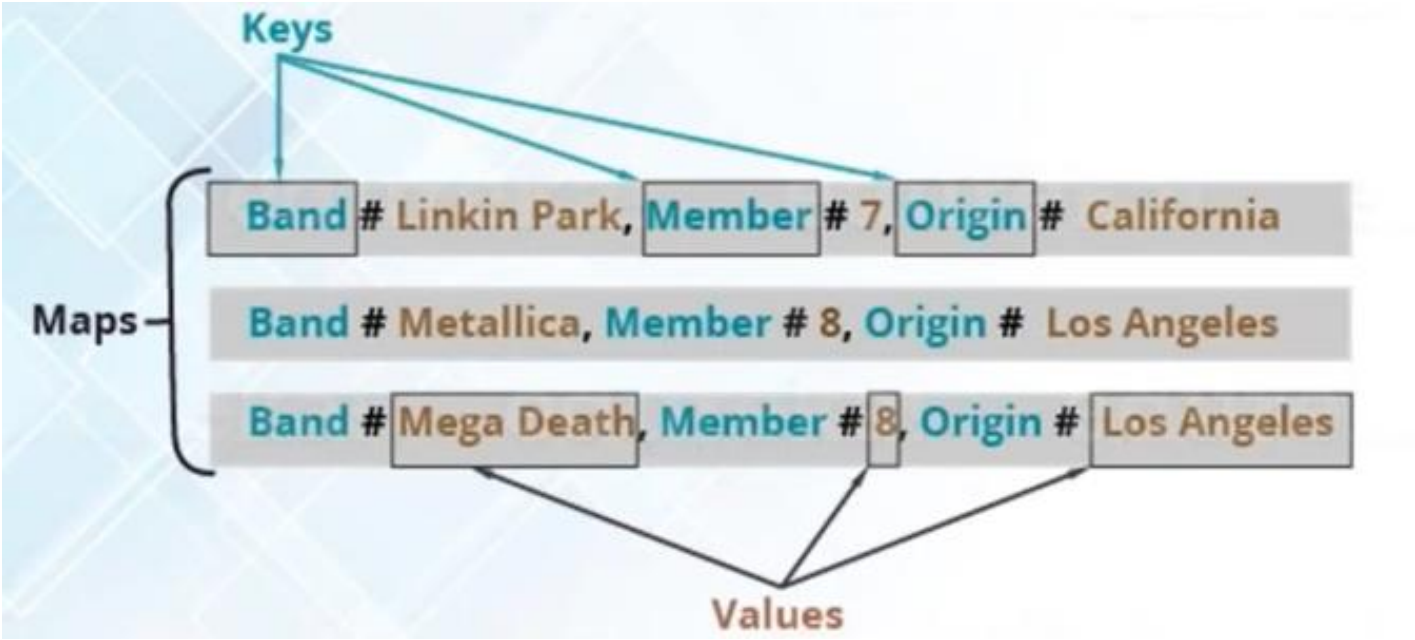
- Tuple is an ordered set of fields which may contain different data types for each field.

  Example of tuple – (1, Linkin Park, 7, California)

- A Bag is a collection of a set of tuples and these tuples are subset of rows or entire rows of a table.

  Example of a bag – {(Linkin Park, 7, California), (Metallica, 8), (Mega Death, Los Angeles)}

Pig Data Model: Map & Atom



- A Map is key-value pairs used to represent data elements.

  Example of maps– [band#Linkin Park, members#7], [band#Metallica, members#8]

- Atoms are basic data types which are used in all the languages like string, int, float, long, double, char[], byte[]

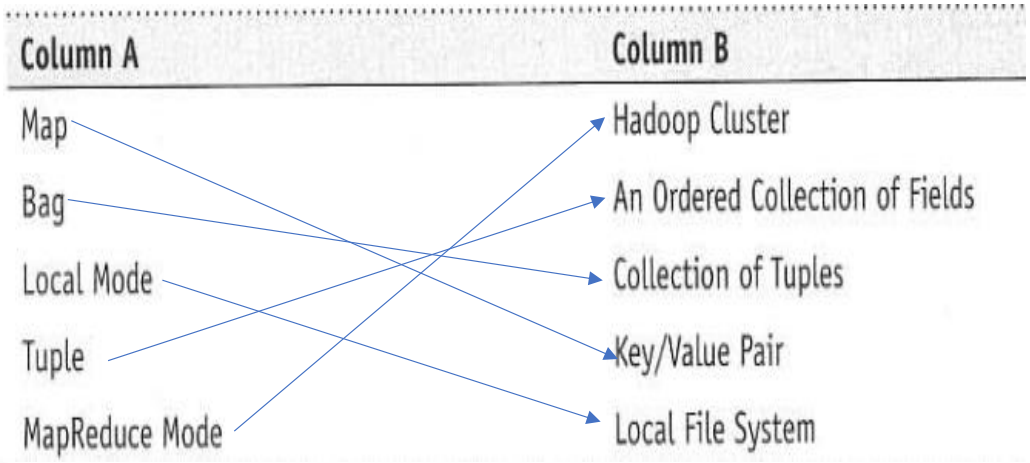
Pig: Operators

Operator	Description
LOAD	Load data from the local file system or HDFS storage into Pig
FOREACH	Generates data transformations based on columns of data
FILTER	Selects tuples from a relation based on a condition
JOIN	Join the relations based on the column
ORDER BY	Sort a relation based on one or more fields
STORE	Save results to the local file system or HDFS
DISTINCT	Removes duplicate tuples in a relation
GROUP	Groups together the tuples with the same group key (key field)
COGROUP	It is same as GROUP. But COGROUP is used when multiple relations re involved

Fill in??????

Pig is a scripting language.
In Pig, Pig latin is used to specify data flow.
Pig provides an Pig engine to execute data flow.
local, mapreduce are execution modes of Pig.
The interactive mode of Pig is grunt.
relation and path are case sensitive in Pig.
Bag, tuple, map are Complex Data Types of Pig.
Pig is used in ETL process.

Can you Match ??????????



True / False ??????????

PigStorage() function is case sensitive. ✓
Local Mode is the default mode of Pig. ✗
DISTINCT Keyword removes duplicate fields. ✗
LIMIT keyword is used to display limited number of tuples in Pig. ✓
ORDER BY is used for sorting. ✓

Piggy Bank

- Apache Pig provides extensive support for **U**ser **D**efined **F**unctions (UDF's).
- The UDF support is provided in six programming languages, namely, Java, Jython, Python, JavaScript, Ruby and Groovy.
- For writing UDF's, complete support is provided in Java and limited support is provided in all the remaining languages.
- Since Apache Pig has been written in Java, the UDF's written using Java language work efficiently compared to other languages.
- In Apache Pig, we also have a Java repository for UDF's named **Piggybank**.
- User can use piggy bank functions in pig latin script and

can share their functions in piggy bank

PIG EXECUTION : Load and Store data locally and on Hadoop

Step1: Create input.txt file

Step2: Transfer to HDFS

```
hdfs dfs put  /home/hdoop/input.txt  bda1/
```

Step3: Create Pigscript file

```
sudo gedit pigscript.pig
        OR
vi pigscript.pig
```

Code to be typed in pigscript.pig

```
record = load '/bda1/input.txt/';
store record into '/bda1/out';
```

Step4: **Run pigscript in mapreduce mode**

```
pig -x mapreduce pigscript.pig
```

Step5: **Check the status of execution**

Output: Found 2 items

```
-rw-r--r--   2 hdoop supergroup          0 2024-01-12 15:05 /bda1/out/_SUCCESS
-rw-r--r--   2 hdoop supergroup        112 2024-01-12 15:05 /bda1/out/part-m-00000
```

Step 6: View the output file

```
hdfs dfs -cat /bda1/out/part-m-00000
```

WORD COUNT PROGRAM

Step1: Create a text file and add some contents to text file

Step 2: Open .pig file and edit the following script into that

```
--LOAD THE DATA
records = LOAD '/pig1/input.txt';
-- SPLIT EACH LINE OF TEXT AND ELIMINATE NESTING
terms = FOREACH records GENERATE FLATTEN(TOKENIZE((chararray) $0)) AS word;
--GROUP SIMILAR TERMS
grouped_terms = GROUP terms BY word;
--COUNT THE NUMBER OF TUPLES IN EACH GROUP
word_counts = FOREACH grouped_terms GENERATE COUNT(terms), group;
--STORE THE RESULT
STORE word_counts INTO '/pig1/output';
```

Step 3: pig (type this at commond prompt)

Step4: grunt>pig wordcount.pig

Step 5: grunt>run wordcount.pig

Step 6: grunt> pwd

Step7: grunt> cd /pig1/output

Step8: grunt> ls

Output:

hdfs://192.168.159.101:9000/pig1/output/_SUCCESS<r 2> 0

hdfs://192.168.159.101:9000/pig1/output/part-r-00000<r 2>127

Step9: grunt> cat part-r-00000

Output:

2	i
2	am
1	hi
2	in
1	are
2	how
1	too
2	you
1	Data
2	good
1	hope
1	you.
1	Btech
1	about
1	doing
1	manipal
1	science.
1	studying


Thank You!


Thank you