BDA – UNIT-IV

1. What is Apache Pig?

Apache Pig is a high-level data flow platform for executing MapReduce programs of Hadoop. The Pig scripts get internally converted to Map Reduce jobs and get executed on data stored in HDFS. Apart from that, Pig can also execute its job in Apache Tez or Apache Spark.

Pig can handle any type of data, i.e., structured, semi-structured or unstructured and stores the corresponding results into Hadoop Data File System. Every task which can be achieved using PIG can also be achieved using java used in MapReduce.

Apache Pig is an abstraction over MapReduce. It is a tool/platform which is used to analyze larger 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 analyze data using **Apache Pig**, programmers need to write scripts using Pig Latin language. 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.

Apache Pig – History

In **2006**, Apache Pig was developed as a research project at Yahoo, especially to create and execute MapReduce jobs on every dataset. In **2007**, Apache Pig was open sourced via Apache incubator. In **2008**, the first release of Apache Pig came out. In **2010**, Apache Pig graduated as an Apache top-level project.

Need of Apache Pig:

Programmers who are not so good at Java normally used to struggle working with Hadoop, especially while performing any MapReduce tasks. Apache Pig is a boon for all such programmers.

- Using **Pig Latin**, programmers can perform MapReduce tasks easily without having to type complex codes in Java.
- Apache Pig uses **multi-query approach**, thereby reducing the length of codes. For example, an operation that would require you to type 200 lines of code (LoC) in Java can be easily done by typing as less as just 10 LoC in Apache Pig. Ultimately Apache Pig reduces the development time by almost 16 times.
- 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.

Features of Pig

Apache Pig comes with the following 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.

Advantages of Apache Pig

- Less code - The Pig consumes less line of code to perform any operation.
- Reusability - The Pig code is flexible enough to reuse again.
- Nested data types - The Pig provides a useful concept of nested data types like tuple, bag, and map.

Applications of Apache Pig

Apache Pig is generally used by data scientists for performing tasks involving ad-hoc processing and quick prototyping. Apache Pig is used −

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

2. Compare Pig with Other Databases?

Apache Pig Vs MapReduce

Apache 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 codes 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. On execution, every Apache Pig operator is converted internally into a MapReduce job.	MapReduce jobs have a long compilation process.

Apache Pig Vs 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 (values are stored as $01, $02 etc.)	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.

In addition to above differences, Apache Pig Latin −

- Allows splits in the pipeline.
- Allows developers to store data anywhere in the pipeline.
- Declares execution plans.
- Provides operators to perform ETL (Extract, Transform, and Load) functions.

Apache Pig Vs Hive

Both Apache Pig and Hive are used to create MapReduce jobs. And in some cases, Hive operates on HDFS in a similar way Apache Pig does. In the following table, we have listed a few significant points that set Apache Pig apart from Hive.

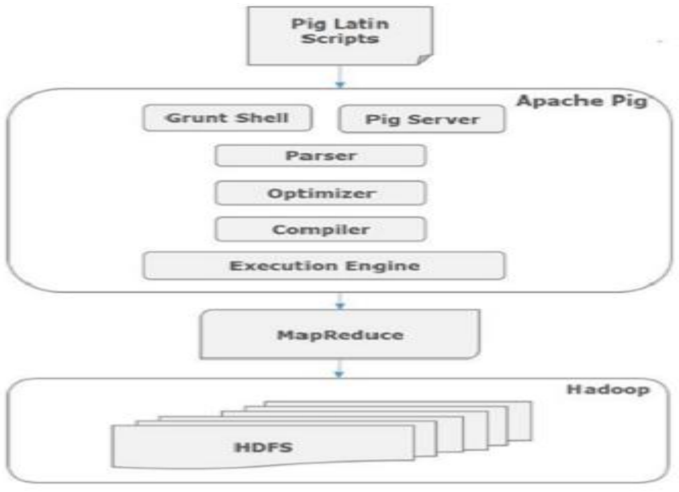
Apache Pig	Hive
Apache Pig uses a language called **Pig Latin**. It was originally created at **Yahoo**.	Hive uses a language called **HiveQL**. It was originally created at **Facebook**.
Pig Latin is a data flow language.	HiveQL is a query processing language.
Pig Latin is a procedural language and it fits in pipeline paradigm.	HiveQL is a declarative language.
Apache Pig can handle structured, unstructured, and semi-structured data.	Hive is mostly for structured data.

3) Explain Pig-Architecture?

Apache Pig – Architecture: The language used to analyze data in Hadoop using Pig is known as **Pig Latin**. It is a highlevel data processing language which provides a rich set of data types and operators to perform various operations on the data.

To perform a particular task Programmers using Pig, programmers need to write a Pig script using the Pig Latin language, and execute them using any of the execution mechanisms (Grunt Shell, UDFs, Embedded). After execution, these scripts will go through a series of transformations applied by the Pig Framework, to produce the desired output.

Internally, Apache Pig converts these scripts into a series of MapReduce jobs, and thus, it makes the programmer's job easy. The architecture of Apache Pig is shown below.



Apache Pig Components

As shown in the figure, there are various components in the Apache Pig framework. Let us take a look at the major components.

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.

Compiler
The compiler compiles the optimized logical plan into a series of MapReduce jobs.

Execution engine
Finally the MapReduce jobs are submitted to Hadoop in a sorted order. Finally, these MapReduce jobs are executed on Hadoop producing the desired results.

3) Give Brief note on Apache Pig Installation?

Apache Pig Installation

Pre-requisite

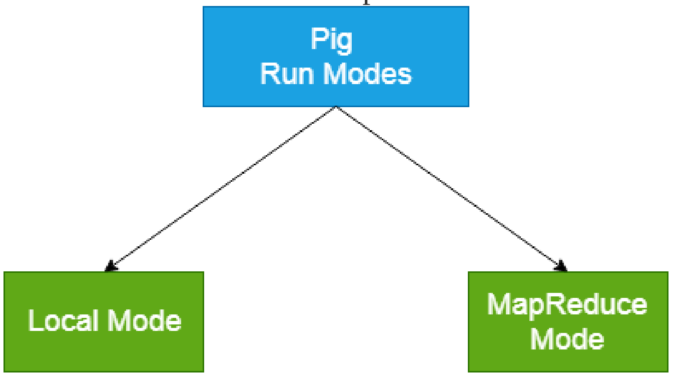
- **Java Installation** - Check whether the Java is installed or not using the following command.
  1. $java -version
- **Hadoop Installation** - Check whether the Hadoop is installed or not using the following command.
  1. $hadoop version

Steps to install Apache Pig

- Download the Apache Pig tar file.
- Unzip the downloaded tar file.
  1. $ tar -xvf pig-0.16.0.tar.gz
- Open the bashrc file.
  1. $ sudo nano ~/.bashrc
- Now, provide the following PIG_HOME path.
  1. export PIG_HOME=/home/hduser/pig-0.16.0
  2. export PATH=$PATH:$PIG_HOME/bin
- Update the environment variable
  1. $ source ~/.bashrc
- Let's test the installation on the command prompt type
  1. $ pig -h
- Let's start the pig in MapReduce mode.
  1. $ pig

Apache Pig Run Modes

Apache Pig executes in two modes: Local Mode and MapReduce Mode.



Local Mode

- It executes in a single JVM and is used for development experimenting and prototyping.
- Here, files are installed and run using localhost.
- The local mode works on a local file system. The input and output data stored in the local file system.

The command for local mode grunt shell:

```
1. $ pig-x local
```

MapReduce Mode

- The MapReduce mode is also known as Hadoop Mode.
- It is the default mode.
- In this Pig renders Pig Latin into MapReduce jobs and executes them on the cluster.
- It can be executed against semi-distributed or fully distributed Hadoop installation.
- Here, the input and output data are present on HDFS.

The command for Map reduce mode:

```
1. $ pig
```

Or,

```
1. $ pig -x mapreduce
```

Ways to execute Pig Program

These are the following ways of executing a Pig program on local and MapReduce mode: -

- **Interactive Mode** - In this mode, the Pig is executed in the Grunt shell. To invoke Grunt shell, run the pig command. Once the Grunt mode executes, we can provide Pig Latin statements and command interactively at the command line.
- **Batch Mode** - In this mode, we can run a script file having a .pig extension. These files contain Pig Latin commands.
- **Embedded Mode** - In this mode, we can define our own functions. These functions can be called as UDF (User Defined Functions). Here, we use programming languages like Java and Python.

Invoking the Grunt Shell

You can invoke the Grunt shell in a desired mode (local/MapReduce) using the **−x** option as shown below.

Local mode	**MapReduce mode**
Command − $ ./pig –x local	**Command −** $ ./pig -x mapreduce

Output −
`15/09/28 10:13:03 INFO pig.Main: Logging error messages to: /home/Hadoop/pig_1443415383991.log 2015-09-28 10:13:04,838 [main] INFO org.apache.pig.backend.hadoop.execution engine.HExecutionEngine - Connecting to hadoop file system at: file:///`
 	
`grunt>`
	Output −
	`15/09/28 10:28:46 INFO pig.Main: Logging error messages to: /home/Hadoop/pig_1443416326123.log 2015-09-28 10:28:46,427 [main] INFO org.apache.pig.backend.hadoop.execution engine.HExecutionEngine - Connecting to hadoop file system at: file:///`
	
	`grunt>`

Either of these commands gives you the Grunt shell prompt as shown below.

```
grunt>
```

You can exit the Grunt shell using **'ctrl + d'.**

After invoking the Grunt shell, you can execute a Pig script by directly entering the Pig Latin statements in it.

```
grunt> customers = LOAD 'customers.txt' USING PigStorage(',');
```

4) What are the components of Pig?

Components of Pig

There are two major components of the Pig:

- Pig Latin script language
- A runtime engine

Pig Latin script language

The Pig Latin script is a procedural data flow language. It contains syntax and commands that can be applied to implement business logic. Examples of Pig Latin are LOAD and STORE.

A runtime engine

The runtime engine is a compiler that produces sequences of MapReduce programs. It uses HDFS to store and retrieve data. It is also used to interact with the Hadoop system (HDFS and MapReduce). The runtime engine parses, validates, and compiles the script operations into a sequence of MapReduce jobs.

Working Stages of Pig Operations: Pig operations can be explained in the following three stages:

Stage 1: Load data and write Pig script

In this stage, data is loaded and Pig script is written.

```
A = LOAD ‘myfile’
AS (x, y, z);
B = FILTER A by x > 0;
C = GROUP B BY x;
D = FOREACH A GENERATE
x, COUNT(B);
STORE D INTO ‘output’;
```

Stage 2: Pig Operations

In the second stage, the Pig execution engine Parses and checks the script. If it passes the script optimized and a logical and physical plan is generated for execution. The job is submitted to Hadoop as a job defined as a MapReduce Task. Pig Monitors the status of the job using Hadoop API and reports to the client.

Stage 3: Execution of the plan

In the final stage, results are dumped on the section or stored in HDFS depending on the user command.

5) What is Pig Latin? Give the list of Pig Data Types, Data Models, Operators?

Pig Latin: The Pig Latin is a data flow language used by Apache Pig to analyze the data in Hadoop. It is a textual language that abstracts the programming from the Java MapReduce idiom into a notation.

Pig Latin Statements: The Pig Latin statements are used to process the data. It is an operator that accepts a relation as an input and generates another relation as an output.

- It can span multiple lines.
- Each statement must end with a semi-colon.
- It may include expression and schemas.
- By default, these statements are processed using multi-query execution

Pig Latin Conventions

Convention	Description
( )	The parenthesis can enclose one or more items. It can also be used to indicate the tuple data type. Example - (10, xyz, (3,6,9))
[ ]	The straight brackets can enclose one or more items. It can also be used to indicate the map data type. Example - [INNER | OUTER]

{ }	The curly brackets enclose two or more items. It can also be used to indicate the bag data type Example - { block | nested_block }
...	The horizontal ellipsis points indicate that you can repeat a portion of the code. Example - cat path [path ...]

Latin Data Types

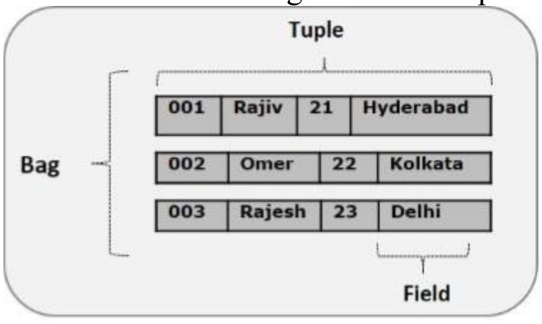
Simple Data Types

Type	Description
int	It defines the signed 32-bit integer. Example - 2
long	It defines the signed 64-bit integer. Example - 2L or 2l
float	It defines 32-bit floating point number. Example - 2.5F or 2.5f or 2.5e2f or 2.5.E2F
double	It defines 64-bit floating point number. Example - 2.5 or 2.5 or 2.5e2f or 2.5.E2F
chararray	It defines character array in Unicode UTF-8 format. Example - javatpoint
bytearray	It defines the byte array.
boolean	It defines the boolean type values. Example - true/false
datetime	It defines the values in datetime order. Example - 1970-01- 01T00:00:00.000+00:00
biginteger	It defines Java BigInteger values. Example - 5000000000000
bigdecimal	It defines Java BigDecimal values. Example - 52.232344535345

Complex Types

Type	Description
tuple	It defines an ordered set of fields. Example - (15,12)
bag	It defines a collection of tuples. Example - {(15,12), (12,15)}
map	It defines a set of key-value pairs. Example - [open#apache]

Pig Latin Data Model: The data model of Pig Latin is fully nested and it allows complex non-atomic datatypes such as map and tuple. Given below is the diagrammatical representation of Pig Latin's 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**.

Example − ‘raja’ or ‘30’

Tuple

A record that is formed by an ordered set of fields is known as a tuple, the fields can be of any type. A tuple is similar to a row in a table of RDBMS.

Example − (Raja, 30)

Bag

A bag is an unordered set of tuples. In other words, a collection of tuples (non-unique) is known as a bag. Each tuple can have any number of fields (flexible schema). A bag is represented by ‘{}’. It is similar to a table in RDBMS, but unlike a table in RDBMS, it is not necessary that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

Example − {(Raja, 30), (Mohammad, 45)}

A bag can be a field in a relation; in that context, it is known as **inner bag**.

Example − {Raja, 30, **{9848022338, raja@gmail.com,}**}

Map

A map (or data map) is a set of key-value pairs. The **key** needs to be of type chararray and should be unique. The **value** might be of any type. It is represented by ‘[]’

Example − [name#Raja, age#30]

Relation

A relation is a bag of tuples. The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).

Pig Latin – Statements

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.

Example

Given below is a Pig Latin statement, which loads data to Apache Pig.

```
grunt> Student_data = LOAD 'student_data.txt' USING PigStorage(',')as
  ( id:int, firstname:chararray, lastname:chararray, phone:chararray, city:chararray );
```

Pig Latin – Data types

Given below table describes the Pig Latin data types.

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
		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]

Null Values

Values for all the above 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 Latin – Arithmetic Operators

The following table describes the arithmetic operators of Pig Latin. Suppose a = 10 and b = 20.

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 – 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]

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.

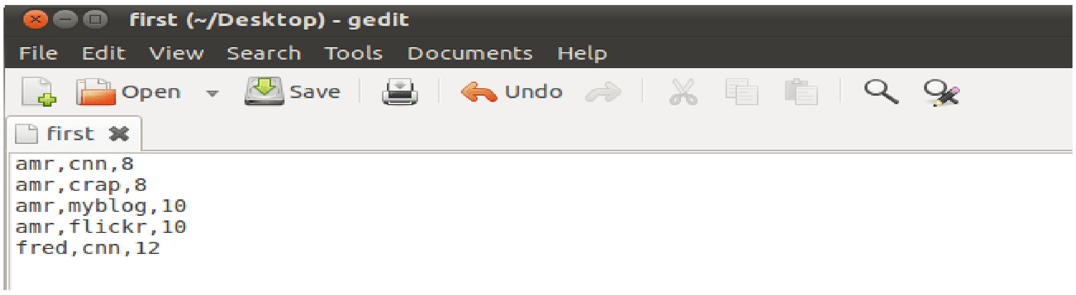
Sorting	
ORDER	To arrange a relation in a sorted order based on one or more fields (ascending or descending).
LIMIT	To get a limited number of tuples from a relation.
Combining and Splitting	
UNION	To combine two or more relations into a single relation.
SPLIT	To split a single relation into two or more 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.

6) Explain briefly the Pig in Practice?
Apache Pig Operators:

Apache Pig Operators is a high-level procedural language for querying large data sets using Hadoop and the Map Reduce Platform. A Pig Latin statement is an operator that takes a relation as input and produces another relation as output. These operators are the main tools Pig Latin provides to operate on the data. They allow you to transform it by sorting, grouping, joining, projecting, and filtering. The Apache Pig operators can be classified as: ***Relational and Diagnostic.***

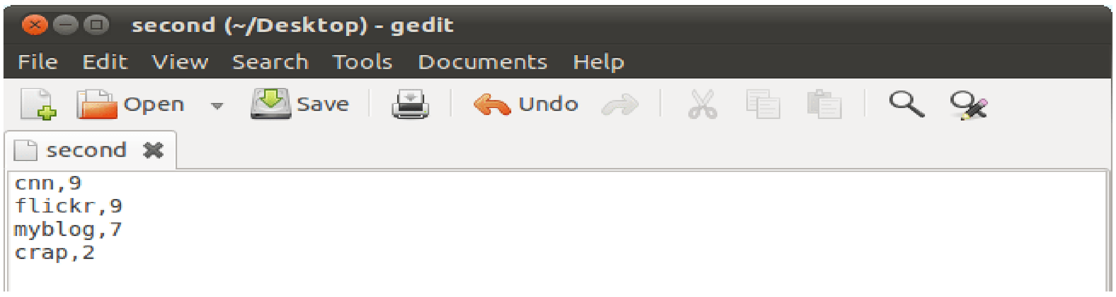
Relational Operators: Relational operators are the main tools Pig Latin provides to operate on the data. It allows you to transform the data by sorting, grouping, joining, projecting and filtering.

Let's create two files to run the commands:
We have two files with name 'first' and 'second.' The first file contains three fields: user, url & id.


first (~/Desktop) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
first

```
amr,cnn,8
amr,crap,8
amr,myblog,10
amr,flickr,10
fred,cnn,12
```

The second file contain two fields: url & rating. These two files are CSV files.

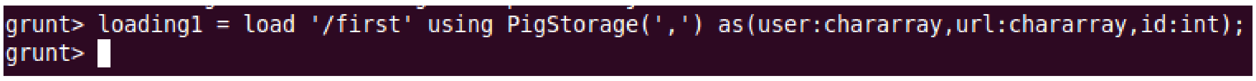

second (~/Desktop) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
second

```
cnn,9
flickr,9
myblog,7
crap,2
```

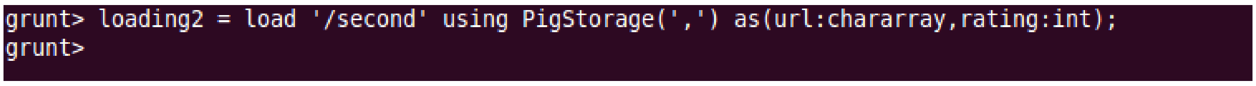
LOAD:

LOAD operator is used to load data from the file system or HDFS storage into a Pig relation.

In this example, the Load operator loads data from file 'first' to form relation 'loading1'. The field names are user, url, id.



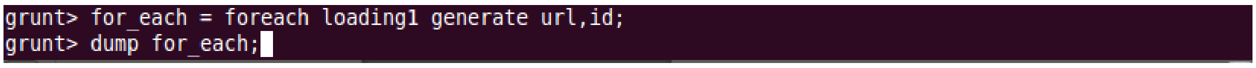
```
grunt> loading1 = load '/first' using PigStorage(',') as(user:chararray,url:chararray,id:int);
grunt> 
```



```
grunt> loading2 = load '/second' using PigStorage(',') as(url:chararray,rating:int);
grunt>
```

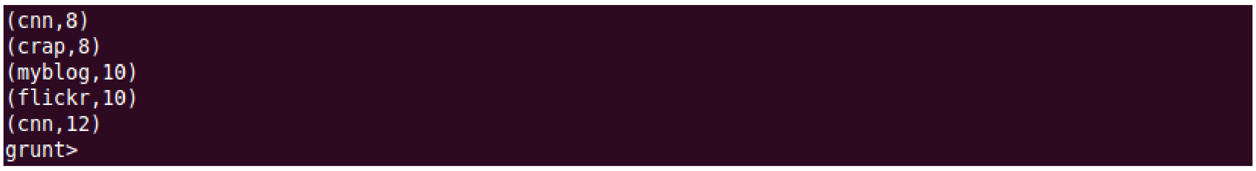
FOREACH:

This operator generates data transformations based on columns of data. It is used to add or remove fields from a relation. Use FOREACH-GENERATE operation to work with columns of data.



```
grunt> for_each = foreach loading1 generate url,id;
grunt> dump for_each;
```

FOREACH Result:

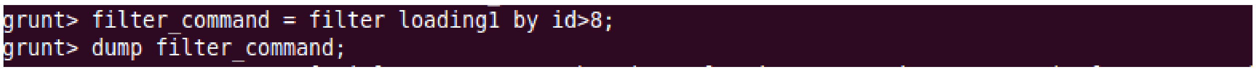


```
(cnn,8)
(crap,8)
(myblog,10)
(flickr,10)
(cnn,12)
grunt>
```

FILTER:

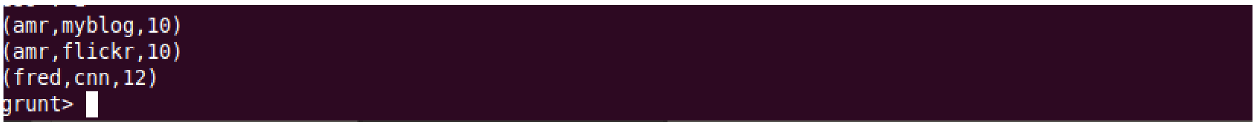
This operator selects tuples from a relation based on a condition.

In this example, we are filtering the record from 'loading1' when the condition 'id' is greater than 8.



```
grunt> filter_command = filter loading1 by id>8;
grunt> dump filter_command;
```

FILTER Result:

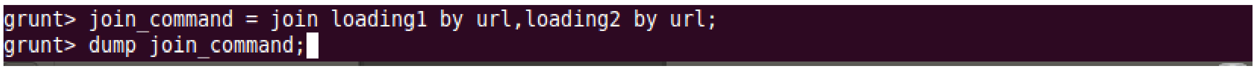


```
(amr,myblog,10)
(amr,flickr,10)
(fred,cnn,12)
grunt> 
```

JOIN:

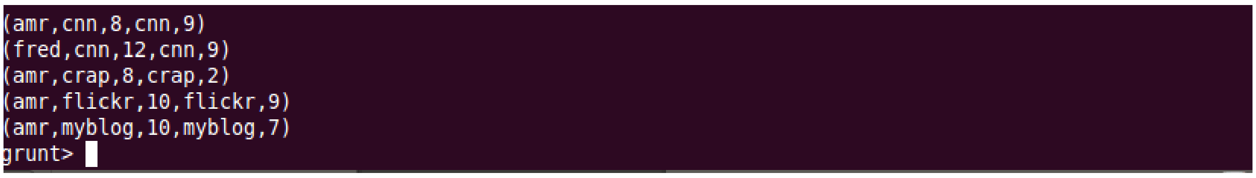
JOIN operator is used to perform an inner, equijoin join of two or more relations based on common field values. The JOIN operator always performs an inner join. Inner joins ignore null keys, so it makes sense to filter them out before the join.

In this example, join the two relations based on the column 'url' from 'loading1' and 'loading2'.



```
grunt> join_command = join loading1 by url,loading2 by url;
grunt> dump join_command;
```

JOIN Result:



```
(amr,cnn,8,cnn,9)
(fred,cnn,12,cnn,9)
(amr,crap,8,crap,2)
(amr,flickr,10,flickr,9)
(amr,myblog,10,myblog,7)
grunt> 
```

ORDER BY:
Order By is used to sort a relation based on one or more fields. You can do sorting in ascending or descending order using ASC and DESC keywords.

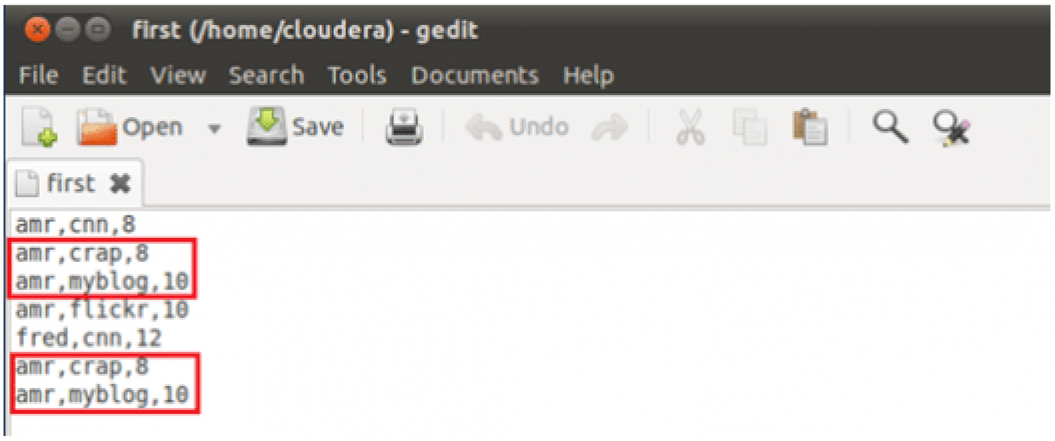
In below example, we are sorting data in loading2 in ascending order on ratings field.

```
grunt> loading4 = ORDER loading2 by rating ASC;
grunt> dump loading4;
```

ORDER BY Result:

```
(crap,2)
(myblog,7)
(cnn,9)
(flickr,9)
grunt>
```

***DISTINCT**:*
Distinct removes duplicate tuples in a relation.Lets take an input file as below, which has **amr,crap,8** and **amr,myblog,10** twice in the file. When we apply distinct on the data in this file, duplicate entries are removed.


first (/home/cloudera) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
first

```
amr,cnn,8
amr,crap,8
amr,myblog,10
amr,flickr,10
fred,cnn,12
amr,crap,8
amr,myblog,10
```

```
grunt> loading1 = load '/first' using PigStorage(',') as (user:chararray,url:chararray,id:int);
grunt> loading3 = DISTINCT loading1;
grunt> dump loading3;
```

DISTINCT Result:

```
(amr,cnn,8)
(amr,crap,8)
(amr,flickr,10)
(amr,myblog,10)
(fred,cnn,12)
grunt>
```

STORE:
Store is used to save results to the file system.

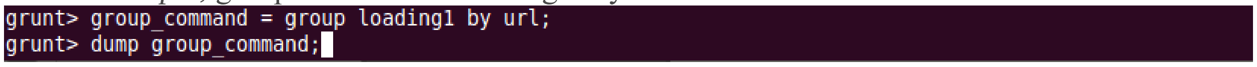
Here we are saving **loading3** data into a file named **storing** on HDFS.

```
grunt> store loading3 into '/storing';
```

GROUP:

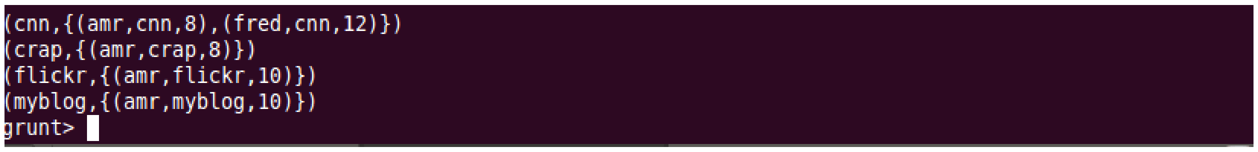
The GROUP operator groups together the tuples with the same group key (key field). The key field will be a tuple if the group key has more than one field, otherwise it will be the same type as that of the group key. The result of a GROUP operation is a relation that includes one tuple per group.

In this example, group the relation ‘loading1’ by column url.



```
grunt> group_command = group loading1 by url;
grunt> dump group_command;
```

GROUP Result:

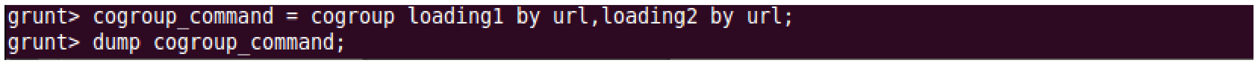


```
(cnn,{(amr,cnn,8),(fred,cnn,12)})
(crap,{(amr,crap,8)})
(flickr,{(amr,flickr,10)})
(myblog,{(amr,myblog,10)})
grunt>
```

COGROUP:

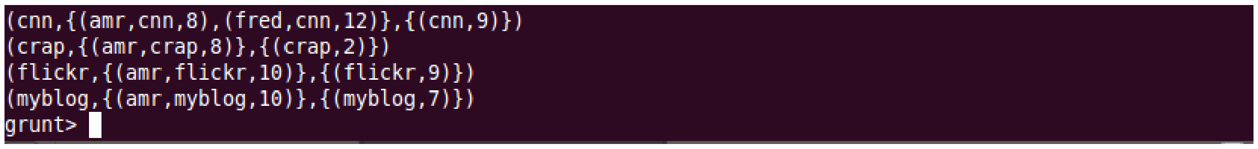
COGROUP is same as GROUP operator. For readability, programmers usually use GROUP when only one relation is involved and COGROUP when multiple relations re involved.

In this example group the ‘loading1’ and ‘loading2’ by url field in both relations.



```
grunt> cogroup_command = cogroup loading1 by url,loading2 by url;
grunt> dump cogroup_command;
```

COGROUP Result:

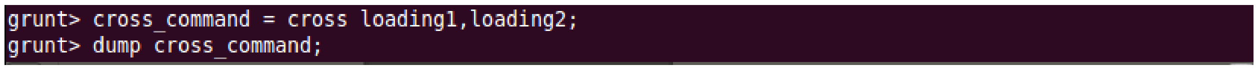


```
(cnn,{(amr,cnn,8),(fred,cnn,12)},{(cnn,9)})
(crap,{(amr,crap,8)},{(crap,2)})
(flickr,{(amr,flickr,10)},{(flickr,9)})
(myblog,{(amr,myblog,10)},{(myblog,7)})
grunt>
```

CROSS:

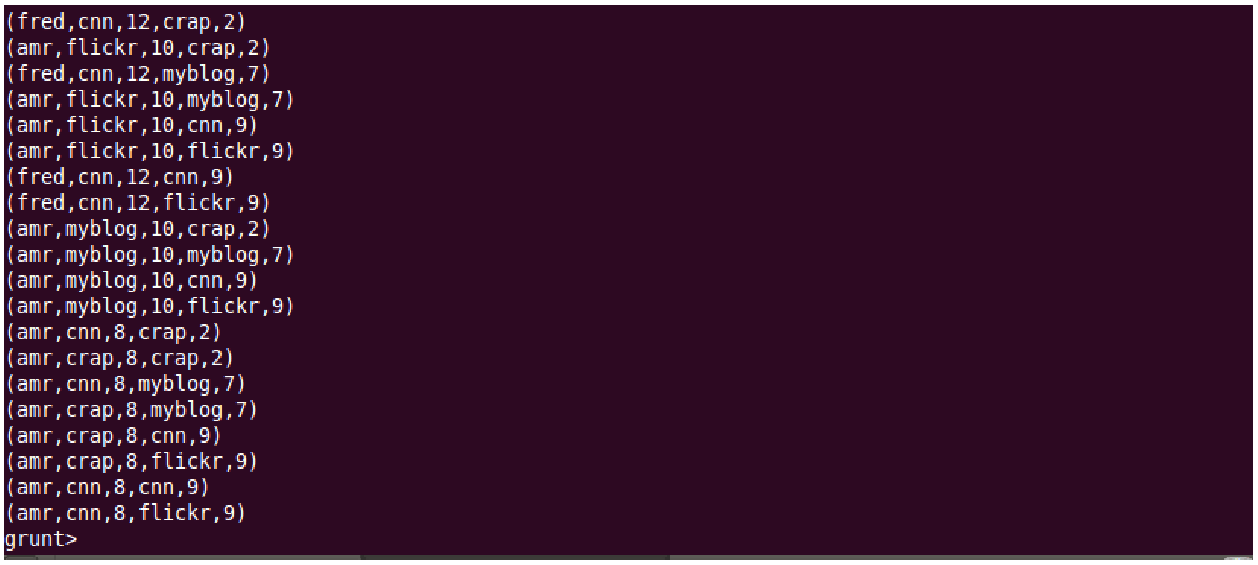
The CROSS operator is used to compute the cross product (Cartesian product) of two or more relations.

Applying cross product on loading1 and loading2.



```
grunt> cross_command = cross loading1,loading2;
grunt> dump cross_command;
```

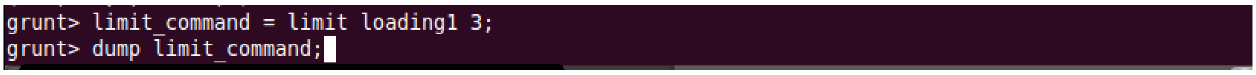
CROSS Result:



```
(fred,cnn,12,crap,2)
(amr,flickr,10,crap,2)
(fred,cnn,12,myblog,7)
(amr,flickr,10,myblog,7)
(amr,flickr,10,cnn,9)
(amr,flickr,10,flickr,9)
(fred,cnn,12,cnn,9)
(fred,cnn,12,flickr,9)
(amr,myblog,10,crap,2)
(amr,myblog,10,myblog,7)
(amr,myblog,10,cnn,9)
(amr,myblog,10,flickr,9)
(amr,cnn,8,crap,2)
(amr,crap,8,crap,2)
(amr,cnn,8,myblog,7)
(amr,crap,8,myblog,7)
(amr,crap,8,cnn,9)
(amr,crap,8,flickr,9)
(amr,cnn,8,cnn,9)
(amr,cnn,8,flickr,9)
grunt>
```

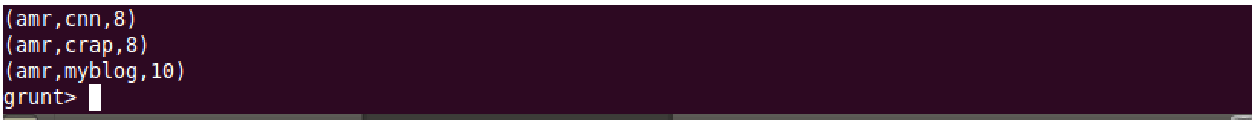
LIMIT:

LIMIT operator is used to limit the number of output tuples. If the specified number of output tuples is equal to or exceeds the number of tuples in the relation, the output will include all tuples in the relation.



```
grunt> limit_command = limit loading1 3;
grunt> dump limit_command;
```

LIMIT Result:

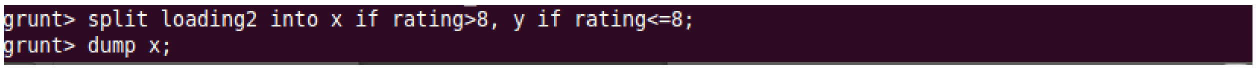


```
(amr,cnn,8)
(amr,crap,8)
(amr,myblog,10)
grunt>
```

SPLIT:

SPLIT operator is used to partition the contents of a relation into two or more relations based on some expression. Depending on the conditions stated in the expression.

Split the loading2 into two relations x and y. x relation created by loading2 contain the fields that the rating is greater than 8 and y relation contain fields that rating is less than or equal to 8.

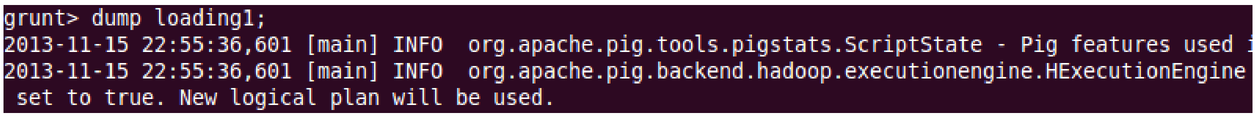


```
grunt> split loading2 into x if rating>8, y if rating<=8;
grunt> dump x;
```

Diagnostic Operators:

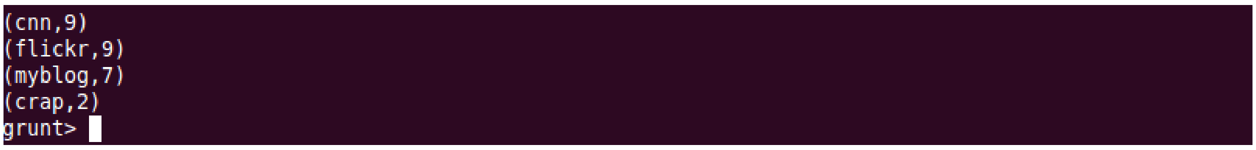
DUMP:

The DUMP operator is used to run Pig Latin statements and display the results on the screen. *In this example*, the operator prints ‘loading1’ on to the screen.



```
grunt> dump loading1;
2013-11-15 22:55:36,601 [main] INFO  org.apache.pig.tools.pigstats.ScriptState - Pig features used i
2013-11-15 22:55:36,601 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine
 set to true. New logical plan will be used.
```

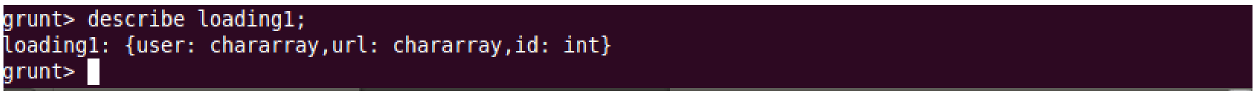
DUMP Result:



```
(cnn,9)
(flickr,9)
(myblog,7)
(crap,2)
grunt>
```

DESCRIBE*:*

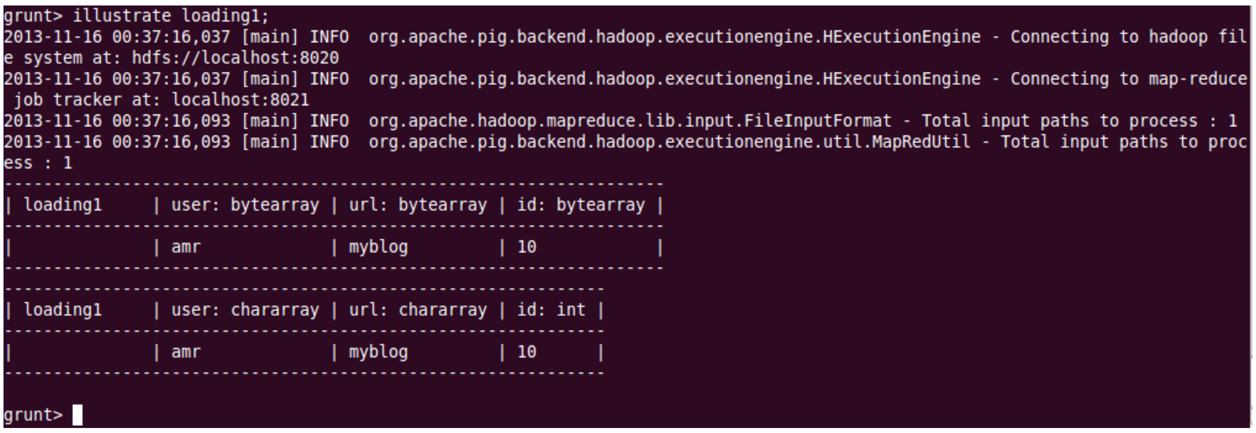
Use the DESCRIBE operator to review the schema of a particular relation. The DESCRIBE operator is best used for debugging a script.



```
grunt> describe loading1;
loading1: {user: chararray,url: chararray,id: int}
grunt>
```

ILLUSTRATE*:*

ILLUSTRATE operator is used to review how data is transformed through a sequence of Pig Latin statements. ILLUSTRATE command is your best friend when it comes to debugging a script. This command alone might be a good reason for choosing Pig over something else.



```
grunt> illustrate loading1;
2013-11-16 00:37:16,037 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop fil
e system at: hdfs://localhost:8020
2013-11-16 00:37:16,037 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to map-reduce
 job tracker at: localhost:8021
2013-11-16 00:37:16,093 [main] INFO  org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths to process : 1
2013-11-16 00:37:16,093 [main] INFO  org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input paths to proc
ess : 1
-------------------------------------------------------------------
| loading1     | user: bytearray | url: bytearray | id: bytearray |
-------------------------------------------------------------------
|              | amr             | myblog         | 10            |
-------------------------------------------------------------------
-------------------------------------------------------------
| loading1     | user: chararray | url: chararray | id: int |
-------------------------------------------------------------
|              | amr             | myblog         | 10      |
-------------------------------------------------------------

grunt> 
```

EXPLAIN:
The EXPLAIN operator prints the logical and physical plane.

7) Briefly Explain about Pig UDF?

Pig UDF (User Defined Functions)
To specify custom processing, Pig provides support for user-defined functions (UDFs). Thus, Pig allows us to create our own functions. Currently, Pig UDFs can be implemented using the following programming languages: -

- Java
- Python
- Jython
- JavaScript
- Ruby
- Groovy

Among all the languages, Pig provides the most extensive support for Java functions. However, limited support is provided to languages like Python, Jython, JavaScript, Ruby, and Groovy.

For writing UDF's, complete support is provided in Java and limited support is provided in all the remaining languages. Using Java, you can write UDF's involving all parts of the processing like data load/store, column transformation, and aggregation. 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**. Using Piggybank, we can access Java UDF's written by other users, and contribute our own UDF's.

Types of UDF's in Java

While writing UDF's using Java, we can create and use the following three types of functions −

- **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 FOREACHGENERATE statement. These functions are used to perform full MapReduce operations on an inner bag.

Writing UDF’s using Java

To write a UDF using Java, we have to integrate the jar file **Pig-0.15.0.jar**. In this section, we discuss how to write a sample UDF using Eclipse. Before proceeding further, make sure you have installed Eclipse and Maven in your system.

Follow the steps given below to write a UDF function −

- Open Eclipse and create a new project (say **myproject**).
- Convert the newly created project into a Maven project.
- Copy the following content in the pom.xml. This file contains the Maven dependencies for Apache Pig and Hadoop-core jar files.
- Save the file and refresh it. In the **Maven Dependencies** section, you can find the downloaded jar files.
- Create a new class file with name **Sample_Eval** and copy the following content in it.

While writing UDF’s, it is mandatory to inherit the EvalFunc class and provide implementation to **exec()** function. Within this function, the code required for the UDF is written. In the above example, we have return the code to convert the contents of the given column to uppercase.

- After compiling the class without errors, right-click on the Sample_Eval.java file. It gives you a menu. Select **export** as shown in the following screenshot.
- On clicking **export**, you will get the following window. Click on **JAR file**.
- Proceed further by clicking **Next>** button. You will get another window where you need to enter the path in the local file system, where you need to store the jar file.
- Finally click the **Finish** button. In the specified folder, a Jar file **sample_udf.jar** is created. This jar file contains the UDF written in Java.

Using the UDF

After writing the UDF and generating the Jar file, follow the steps given below −

Step 1: Registering the Jar file

After writing UDF (in Java) we have to register the Jar file that contain the UDF using the Register operator. By registering the Jar file, users can intimate the location of the UDF to Apache Pig.

Syntax

```
REGISTER path;
```

Example

As an example let us register the sample_udf.jar created earlier in this chapter.
Start Apache Pig in local mode and register the jar file sample_udf.jar as shown below.

```
$cd PIG_HOME/bin

$./pig –x local


REGISTER '/$PIG_HOME/sample_udf.jar'
```

Note − assume the Jar file in the path − /$PIG_HOME/sample_udf.jar

Step 2: Defining Alias

After registering the UDF we can define an alias to it using the **Define** operator.

Syntax

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

Example
DEFINE sample_eval sample_eval();

Step 3: Using the UDF

After defining the alias you can use the UDF same as the built-in functions. Suppose there is a file named emp_data in the HDFS **/Pig_Data/** directory with the following content.

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuwaneshwar
006,Maggy,22,Chennai
007,Robert,22,newyork
008,Syam,23,Kolkata
009,Mary,25,Tokyo
010,Saran,25,London
011,Stacy,25,Bhuwaneshwar
012,Kelly,22,Chennai
```

And assume we have loaded this file into Pig as shown below.

```
grunt> emp_data = LOAD 'hdfs://localhost:9000/pig_data/emp1.txt' USING PigStorage(',')
   as (id:int, name:chararray, age:int, city:chararray);
```

Let us now convert the names of the employees in to upper case using the UDF **sample_eval**.

```
grunt> Upper_case = FOREACH emp_data GENERATE sample_eval(name);
```

Verify the contents of the relation **Upper_case** as shown below.

grunt> Dump Upper_case;

8) What is Hive? Describe its architecture?

Hive is a data warehouse system which is used to analyze structured data. It is built on the top of Hadoop. Initially Hive is developed by Facebook and Amazon, Netflix.

Hive provides the functionality of reading, writing, and managing large datasets residing in distributed storage. It runs SQL like queries called HQL (Hive query language) which gets internally converted to MapReduce jobs.

Using Hive, we can skip the requirement of the traditional approach of writing complex MapReduce programs. Hive supports Data Definition Language (DDL), Data Manipulation Language (DML), and User Defined Functions (UDF).

Features of Hive

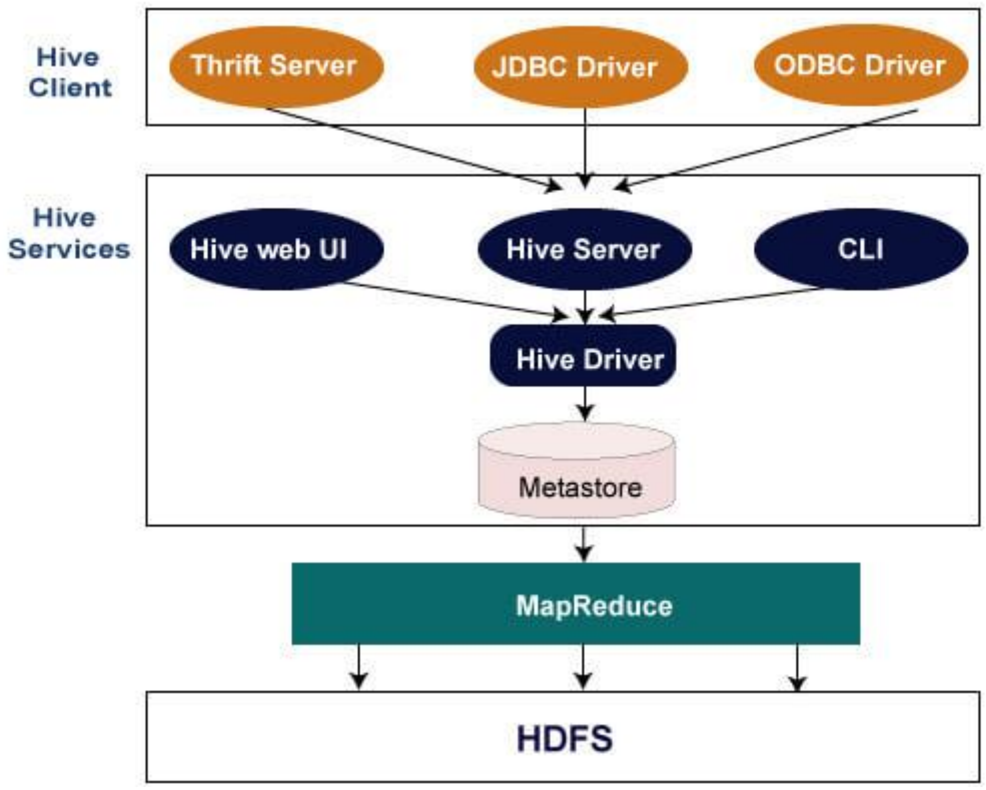
- Hive is fast and scalable.

- It provides SQL-like queries (i.e., HQL) that are implicitly transformed to MapReduce or Spark jobs.
- It is capable of analyzing large datasets stored in HDFS.
- It allows different storage types such as plain text, RCFile, and HBase.
- It uses indexing to accelerate queries.
- It can operate on compressed data stored in the Hadoop ecosystem.
- It supports user-defined functions (UDFs) where user can provide its functionality.

Limitations of Hive

- Hive is not capable of handling real-time data.
- It is not designed for online transaction processing.
- Hive queries contain high latency.

Hive Architecture



Hive Client

Hive allows writing applications in various languages, including Java, Python, and C++. It supports different types of clients such as:-

- Thrift Server - It is a cross-language service provider platform that serves the request from all those programming languages that supports Thrift.
- JDBC Driver - It is used to establish a connection between hive and Java applications. The JDBC Driver is present in the class org.apache.hadoop.hive.jdbc.HiveDriver.
- ODBC Driver - It allows the applications that support the ODBC protocol to connect to Hive.

Hive Services

- Hive CLI - The Hive CLI (Command Line Interface) is a shell where we can execute Hive queries and commands.
- Hive Web User Interface - The Hive Web UI is just an alternative of Hive CLI. It provides a web-based GUI for executing Hive queries and commands.

- Hive MetaStore - It is a central repository that stores all the structure information of various tables and partitions in the warehouse. It also includes metadata of column and its type information, the serializers and deserializers which is used to read and write data and the corresponding HDFS files where the data is stored.
- Hive Server - It is referred to as Apache Thrift Server. It accepts the request from different clients and provides it to Hive Driver.
- Hive Driver - It receives queries from different sources like web UI, CLI, Thrift, and JDBC/ODBC driver. It transfers the queries to the compiler.
- Hive Compiler - The purpose of the compiler is to parse the query and perform semantic analysis on the different query blocks and expressions. It converts HiveQL statements into MapReduce jobs.
- Hive Execution Engine - Optimizer generates the logical plan in the form of DAG of map-reduce tasks and HDFS tasks. In the end, the execution engine executes the incoming tasks in the order of their dependencies.

Different Modes of Hive

Hive operates in two modes depending on the number and size of data nodes. They are:

1. Local Mode - Used when Hadoop has one data node, and the amount of data is small. Here, the processing will be very fast on smaller datasets, which are present in local machines.
2. Mapreduce Mode - Used when the data in Hadoop is spread across multiple data nodes. Processing large datasets can be more efficient using this mode.

9. Explain the process of Hive Installation?

Apache Hive Installation

Pre-requisite

- **Java Installation** - Check whether the Java is installed or not using the following command.
  1. $ java -version
- **Hadoop Installation** - Check whether the Hadoop is installed or not using the following command.
  1. $hadoop version

Steps to install Apache Hive

- Download the Apache Hive tar file.
  http://mirrors.estointernet.in/apache/hive/hive-1.2.2/
- Unzip the downloaded tar file.
  1. tar -xvf apache-hive-1.2.2-bin.tar.gz
- Open the bashrc file.
  1. $ sudo nano ~/.bashrc
- Now, provide the following HIVE_HOME path.
  1. export HIVE_HOME=/home/codegyani/apache-hive-1.2.2-bin
  2. export PATH=$PATH:/home/codegyani/apache-hive-1.2.2-bin/bin
- Update the environment variable.
  1. $ source ~/.bashrc
- Let's start the hive by providing the following command.
  1. $ hive

10) Write a brief note on Data Flow and Data Model in Hive?

Data Flow in Hive

Data flow in the Hive contains the Hive and Hadoop system. Underneath the user interface, we have driver, compiler, execution engine, and metastore. All of that goes into the MapReduce and the Hadoop file system.

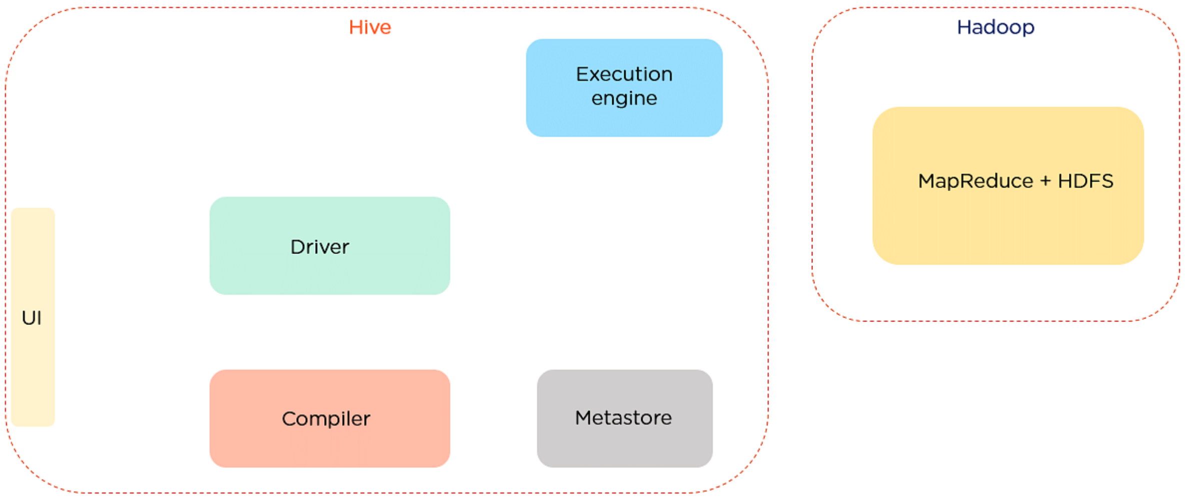


Fig: Data flow in Hive

The data flow in the following sequence:

1. We execute a query, which goes into the driver
2. Then the driver asks for the plan, which refers to the query execution
3. After this, the compiler gets the metadata from the metastore
4. The metastore responds with the metadata
5. The compiler gathers this information and sends the plan back to the driver
6. Now, the driver sends the execution plan to the execution engine
7. The execution engine acts as a bridge between the Hive and Hadoop to process the query
8. In addition to this, the execution engine also communicates bidirectionally with the metastore to perform various operations, such as create and drop tables
9. Finally, we have a bidirectional communication to fetch and send results back to the client

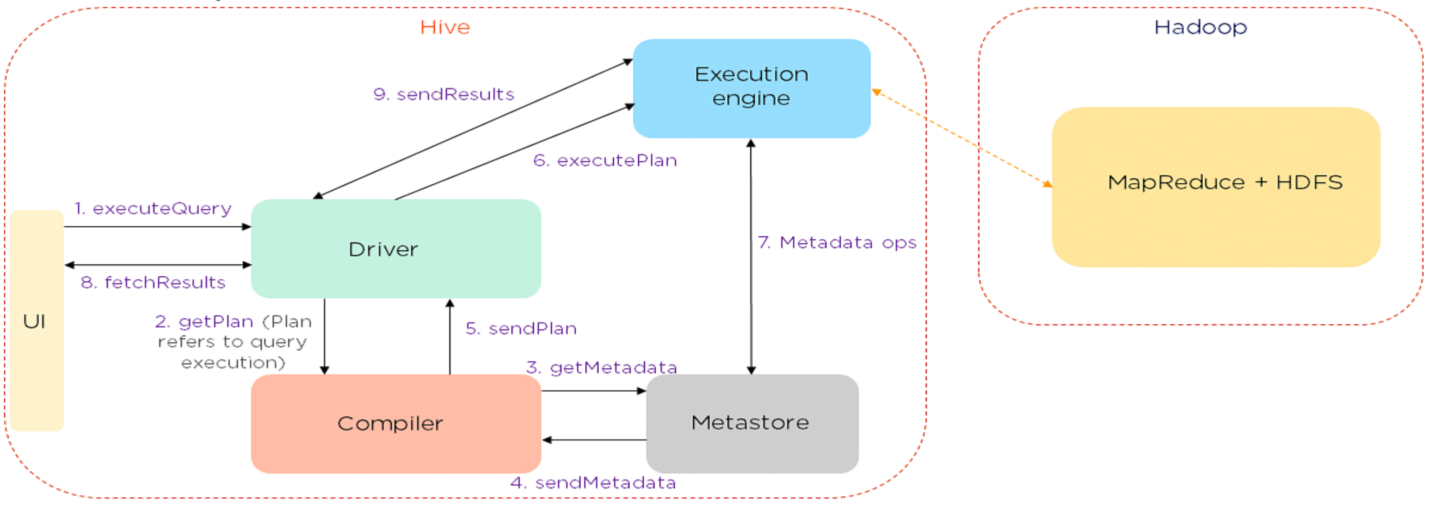


Fig: Data flow in Hive

Hive Data Modeling

1. Tables - Tables in Hive are created the same way it is done in RDBMS
2. Partitions - Here, tables are organized into partitions for grouping similar types of data based on the partition key
3. Buckets - Data present in partitions can be further divided into buckets for efficient querying

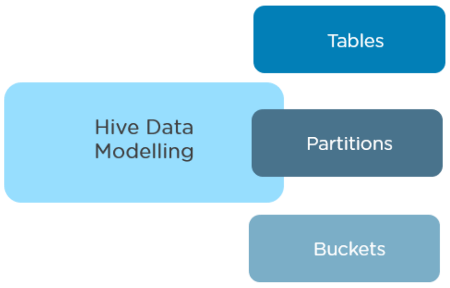


Fig: Hive Data Modelling

11. Compare the Hive the other Databases?

Difference Between Hive and RDBMS

RDBMS, which stands for Relational Database Management System, also works with tables. But what is the difference between Hive and RDBMS?

Hive	RDBMS
• Hive enforces schema on reading • Hive data size is in petabytes • Hive is based on the notion of write once and read many times • Hive resembles a traditional database by supporting SQL, but it is not a database; it is a data warehouse • Easily scalable at low cost	• RDBMS enforces schema on write • Data size is in terabytes • RDBMS is based on the notion of reading and write many times • RDBMS is a type of database management system, which is based on the relational model of data • Not scalable at low cost

Comparison of Hive advantage vs. Pig advantages

The following table compares the advantages of Hive with the advantages of Pig :

Features		
1. Language	Hive uses a declarative language called HiveQL	With Pig Latin, a procedural data flow language is used

2. Schema	Hive supports schema	Creating schema is not required to store data in Pig
3. Data Processing	Hive is used for batch processing	Pig is a high-level data-flow language
4. Partitions	Yes	No. Pig does not support partitions although there is an option for filtering
5. Web interface	Hive has a web interface	Pig does not support web interface
6. User Specification	Data analysts are the primary users	Programmers and researchers use Pig
7. Used for	Reporting	Programming
8. Type of data	Hive works on structured data. Does not work on other types of data	Pig works on structured, semi-structured and unstructured data
9. Operates on	Works on the server-side of the cluster	Works on the client-side of the cluster
10. Avro File Format	Hive does not support Avro	Pig supports Avro
11. Loading Speed	Hive takes time to load but executes quickly	Pig loads data quickly
12. JDBC/ ODBC	Supported, but limited	Unsupported

Fig: Hive vs. Pig Comparison Table

Both Hive and Pig are excellent data analysis tools—one is not necessarily better than the other, but they do have different capabilities and features. Depending on your job role, business requirements, and budget, you can choose either of these Big Data analysis platforms.

12) Write Brief Note on Hive Data Types?

HIVE Data Types

Hive data types are categorized in numeric types, string types, misc types, and complex types. A list of Hive data types is given below.

Integer Types

Type	**Size**	**Range**
TINYINT	1-byte signed integer	-128 to 127
SMALLINT	2-byte signed integer	32,768 to 32,767
INT	4-byte signed integer	2,147,483,648 to 2,147,483,647
BIGINT	8-byte signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Decimal Type

Type	**Size**	**Range**
FLOAT	4-byte	Single precision floating point number
DOUBLE	8-byte	Double precision floating point number

Date/Time Types

TIMESTAMP

- It supports traditional UNIX timestamp with optional nanosecond precision.
- As Integer numeric type, it is interpreted as UNIX timestamp in seconds.
- As Floating point numeric type, it is interpreted as UNIX timestamp in seconds with decimal precision.

- As string, it follows java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff" (9 decimal place precision)

DATES

The Date value is used to specify a particular year, month and day, in the form YYYY--MM--DD. However, it didn't provide the time of the day. The range of Date type lies between 0000--01--01 to 9999--12--31.

String Types

STRING
The string is a sequence of characters. It values can be enclosed within single quotes (') or double quotes (").

Varchar
The varchar is a variable length type whose range lies between 1 and 65535, which specifies that the maximum number of characters allowed in the character string.

CHAR
The char is a fixed-length type whose maximum length is fixed at 255.

Complex Type

Type	Size	Range
Struct	It is similar to C struct or an object where fields are accessed using the "dot" notation.	struct('James','Roy')
Map	It contains the key-value tuples where the fields are accessed using array notation.	map('first','James','last','Roy')
Array	It is a collection of similar type of values that indexable using zero-based integers.	array('James','Roy')

13. What is Hive Query Language (HiveQL)? Give the list of Various HiveQL operators?
Hive Query Language (HiveQL) is a query language in Apache Hive for processing and analyzing structured data. It separates users from the complexity of Map Reduce programming. It reuses common concepts from relational databases, such as tables, rows, columns, and schema, to ease learning. Hive provides a CLI for Hive query writing using Hive Query Language (HiveQL).

Most interactions tend to take place over a command line interface (CLI). Generally, HiveQL syntax is similar to the SQL syntax that most data analysts are familiar with. Hive supports four file formats which are: TEXTFILE, SEQUENCEFILE, ORC and RCFILE (Record Columnar File).

Hive uses derby database for single user metadata storage, and for multiple user Metadata or shared Metadata case, Hive uses MYSQL.

HiveQL Built-in Operators
Hive provides Built-in operators for Data operations to be implemented on the tables present inside Hive warehouse. These operators are used for mathematical operations on operands, and it will return specific value as per the logic applied.

- Relational Operators
- Arithmetic Operators
- Logical Operators
- Operators on Complex types
- Complex type Constructors

Relational Operators in Hive SQL

Built-in Operator	**Description**	**Operand**
X = Y	TRUE if expression X is equivalent to expression Y Otherwise FALSE.	It takes all primitive types
X != Y	TRUE if expression X is not equivalent to expression Y Otherwise FALSE.	It takes all primitive types
X < Y	TRUE if expression X is less than expression Y Otherwise FALSE.	It takes all primitive types
X <= Y	TRUE if expression X is less than or equal to expression Y Otherwise FALSE.	It takes all primitive types
X>Y	TRUE if expression X is greater than expression Y Otherwise FALSE.	It takes all primitive types
X>= Y	TRUE if expression X is greater than or equal to expression Y Otherwise FALSE.	It takes all primitive types
X IS NULL	TRUE if expression X evaluates to NULL otherwise FALSE.	It takes all types
X IS NOT NULL	FALSE If expression X evaluates to NULL otherwise TRUE.	It takes all types
X LIKE Y	TRUE If string pattern X matches to Y otherwise FALSE.	Takes only Strings
X RLIKE Y	NULL if X or Y is NULL, TRUE if any substring of X matches the Java regular expression Y, otherwise FALSE.	Takes only Strings
X REGEXP Y	Same as RLIKE.	Takes only Strings

HiveQL Arithmetic Operators

Built-in Operator	**Description**	**Operand**
X + Y	It will return the output of adding X and Y value.	It takes all number types
X – Y	It will return the output of subtracting Y from X value.	It takes all number types

X * Y	It will return the output of multiplying X and Y values.	It takes all number types
X / Y	It will return the output of dividing Y from X.	It takes all number types
X % Y	It will return the remainder resulting from dividing X by Y.	It takes all number types
X & Y	It will return the output of bitwise AND of X and Y.	It takes all number types
X | Y	It will return the output of bitwise OR of X and Y.	It takes all number types
X ^ Y	It will return the output of bitwise XOR of X and Y.	It takes all number types
~X	It will return the output of bitwise NOT of X.	It takes all number types

Hive QL Logical Operators

Operators	**Description**	**Operands**
X AND Y	TRUE if both X and Y are TRUE, otherwise FALSE.	Boolean types only
X && Y	Same as X AND Y but here we using && symbol	Boolean types only
X OR Y	TRUE if either X or Y or both are TRUE, otherwise FALSE.	Boolean types only
X || Y	Same as X OR Y but here we using || symbol	Boolean types only
NOT X	TRUE if X is FALSE, otherwise FALSE.	Boolean types only
!X	Same as NOT X but here we using! symbol	Boolean types only

Operators on Complex Types

Operators	**Operands**	**Description**
A[n]	A is an Array and n is an integer type	It will return nth element in the array A. The first element has index of 0
M[key]	M is a Map<K, V> and key has type K	It will return the values belongs to the key in the map

14) Explain the Process of Creating Databases, Tables and Querying Tables?
Creating Databases and Tables
In Hive, a table is a collection of data that is sorted according to a specific set of identifiers using a schema.

Step 1: Create a Database

Syntax: CREATE DATABASE IF NOT EXISTS mydatabase;

If a database with the name "mydatabase" doesn't already exist, this statement creates one. The database is only created if it doesn't already exist, thanks to the IF NOT EXISTS condition.

Step 2: Switching to a Database:

Syntax: USE mydatabase;

By switching to the "mydatabase" database using this line, further activities can be carried out in that database.

Step 3: Creating a Table::

Syntax:

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[ROW FORMAT row_format]
[STORED AS file_format]
```

Example 1:

```
CREATE TABLE IF NOT EXISTS employee ( eid int, name String,
salary String, destination String)
COMMENT 'Employee details'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Example 2:

```
CREATE TABLE IF NOT EXISTS employees (
 id INT,
 name STRING,
 age INT
 );
```

The "employees" table is created with this statement. It has three columns: "id" (integer), "name" (string), and "age" (integer). The table is only generated if it doesn't already exist thanks to the IF NOT EXISTS condition.

Step 4: Creating an External Table::

Syntax:

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[ROW FORMAT row_format]
[STORED AS file_format]
```

Example 1:

```
CREATE EXTERNAL TABLE IF NOT EXISTS ext_employee ( eid int, name String,
        salary String, destination String)
        COMMENT 'Employee details'
        ROW FORMAT DELIMITED
        FIELDS TERMINATED BY '\t'
        LINES TERMINATED BY '\n'
        STORED AS TEXTFILE;
```

Example 2:

```
CREATE EXTERNAL TABLE IF NOT EXISTS ext_employees (
          id INT,
          name STRING,
          age INT
         ) LOCATION '/path/to/data';
```

This hadoop hiveql command creates a new external table called "ext_employees." External tables point to data that is kept in a location independent of Hive, preserving the original location of the data. The HDFS path where the data is located is specified by the LOCATION clause.

Loading Data into Tables

- **Load data from HDFS**

  Syntax: LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]

  Example: LOAD DATA LOCAL INPATH '/home/user/sample.txt' OVERWRITE INTO TABLE employee;

Insert data into the table

Example: INSERT INTO TABLE employees VALUES (1, 'John Doe', 30);

The LOAD DATA statement inserts data into the designated table from an HDFS path. The "employees" table receives a specific row of data when the INSERT INTO TABLE query is executed.

Querying Data with HiveQL

One of the core functions of using Apache Hive is data querying with HiveQL. You may obtain, filter, transform, and analyse data stored in Hive tables using HiveQL, which is a language comparable to SQL.

Following are a few typical HiveQL querying operations:

1. Select All Records:

```
SELECT * FROM employees;
```

This hadoop hiveql command retrieves all records from the "employees" table.

2. Filtering: Example: Select employees older than 25

```
SELECT * FROM employees WHERE age > 25;
```

Only those records from the "employees" table that have a "age" greater than 25 are chosen by this.

3. Aggregation: Example: Count the number of employees

```
SELECT COUNT(*) FROM employees;
```

Example: Calculate the average age

```
SELECT AVG(age) FROM employees;
```

These hadoop hiveql queries count the number of employees and determine the average age using aggregation operations on the "employees" table.

4.Sorting:

Example: Sort by age in descending order

```
SELECT * FROM employees ORDER BY age DESC;
```

In order to extract employee names and their related departments, this query connects the "employees" and "departments" databases based on the "department_id" field.

5. Joining Tables:

Example: Join employees and departments based on department_id

```
SELECT e.id, e.name, d.department
FROM employees e
JOIN departments d ON e.department_id = d.id;
```

The "department_id" column is used to link the "employees" and "departments" databases in order to access employee names and their related departments.

6. Grouping and Aggregation:

Example: Count employees in each department

```
SELECT department, COUNT(*) as employee_count
FROM employees
GROUP BY department;
```

This query counts the number of employees in each department and organises employees by department.

7. Limiting Results:

Example: Get the top 10 oldest employees

```
SELECT * FROM employees ORDER BY age DESC LIMIT 10;
```

This search returns the ten oldest employees in order of age.

8. Subqueries: A subquery is a query that is nested inside another query. The SELECT, WHERE, and FROM clauses can all use them.

Example: Determine the typical age of employees in each department using a subquery in the SELECT statement.

```
SELECT department, (
   SELECT AVG(age)
   FROM employees e
   WHERE e.department_id = d.id
) as avg_age
FROM departments d;
```

The average age of employees for each department in the "departments" dataset is determined by this query using a subquery.

9. Correlated Subqueries: An inner query that depends on results from the outer query is referred to as a correlated subquery.

Example: Find employees whose ages are higher than the department's average, for instance.

```
SELECT id, name
FROM employees e
WHERE age > (
   SELECT AVG(age)
   FROM employees
   WHERE department_id = e.department_id
);
```

To locate employees whose ages are higher than the mean ages of employees in the same department, this query uses a correlated subquery.

Drop Database Statement

Drop Database is a statement that drops all the tables and deletes the database. Its syntax is as follows:

DROP DATABASE StatementDROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT|CASCADE];

DROP DATABASE IF EXISTS userdb;

The following query drops the database using **CASCADE**. It means dropping respective tables before dropping the database.

DROP DATABASE IF EXISTS userdb CASCADE;

Alter Table Statement

It is used to alter a table in Hive.

Syntax

The statement takes any of the following syntaxes based on what attributes we wish to modify in a table.

```
ALTER TABLE name RENAME TO new_name
ALTER TABLE name ADD COLUMNS (col_spec[, col_spec ...])
ALTER TABLE name DROP [COLUMN] column_name
ALTER TABLE name CHANGE column_name new_name new_type
ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...])
```

Drop Table Statement

The syntax is as follows:

DROP TABLE [IF EXISTS] table_name;

The following query drops a table named **employee**:

```
hive> DROP TABLE IF EXISTS employee;
```

15) Give the List of various Built-in functions in Hive?

Mathematical Functions in Hive

Return type	Functions	Description
BIGINT	round(num)	It returns the BIGINT for the rounded value of DOUBLE num.
BIGINT	floor(num)	It returns the largest BIGINT that is less than or equal to num.
BIGINT	ceil(num), ceiling(DOUBLE num)	It returns the smallest BIGINT that is greater than or equal to num.
DOUBLE	exp(num)	It returns exponential of num.
DOUBLE	ln(num)	It returns the natural logarithm of num.
DOUBLE	log10(num)	It returns the base-10 logarithm of num.
DOUBLE	sqrt(num)	It returns the square root of num.
DOUBLE	abs(num)	It returns the absolute value of num.
DOUBLE	sin(d)	It returns the sin of num, in radians.
DOUBLE	asin(d)	It returns the arcsin of num, in radians.
DOUBLE	cos(d)	It returns the cosine of num, in radians.
DOUBLE	acos(d)	It returns the arccosine of num, in radians.
DOUBLE	tan(d)	It returns the tangent of num, in radians.
DOUBLE	atan(d)	It returns the arctangent of num, in radians.

Aggregate Functions in Hive

Return Type	Operator	Description
BIGINT	count(*)	It returns the count of the number of rows present in the file.
DOUBLE	sum(col)	It returns the sum of values.
DOUBLE	sum(DISTINCT col)	It returns the sum of distinct values.
DOUBLE	avg(col)	It returns the average of values.
DOUBLE	avg(DISTINCT col)	It returns the average of distinct values.
DOUBLE	min(col)	It compares the values and returns the minimum one form it.
DOUBLE	max(col)	It compares the values and returns the maximum one form it.

Other built-in Functions in Hive

Return Type	Operator	Description
INT	length(str)	It returns the length of the string.
STRING	reverse(str)	It returns the string in reverse order.
STRING	concat(str1, str2, ...)	It returns the concatenation of two or more strings.
STRING	substr(str, start_index)	It returns the substring from the string based on the provided starting index.
STRING	substr(str, int start, int length)	It returns the substring from the string based on the provided starting index and length.
STRING	upper(str)	It returns the string in uppercase.
STRING	lower(str)	It returns the string in lowercase.
STRING	trim(str)	It returns the string by removing whitespaces from both the ends.
STRING	ltrim(str)	It returns the string by removing whitespaces from left-hand side.
TRING	rtrim(str)	It returns the string by removing whitespaces from right-hand side.