Preparing HDFS

In MapReduce mode, Pig reads (loads) data from HDFS and stores the results back in HDFS. Therefore, let us start HDFS and create the following sample data in HDFS.

Student ID	First Name	Last Name	Phone	City
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

The above dataset contains personal details like id, first name, last name, phone number and city, of six students.

Step 1: Verifying Hadoop

First of all, verify the installation using Hadoop version command, as shown below.

```
$ hadoop version
```

If your system contains Hadoop, and if you have set the PATH variable, then you will get the following output −

```
Hadoop 2.6.0
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r
e3496499ecb8d220fba99dc5ed4c99c8f9e33bb1
Compiled by jenkins on 2014-11-13T21:10Z
Compiled with protoc 2.5.0
From source with checksum 18e43357c8f927c0695f1e9522859d6a
This command was run using /home/Hadoop/hadoop/share/hadoop/common/hadoop
common-2.6.0.jar
```

Step 2: Starting HDFS

Browse through the **sbin** directory of Hadoop and start **yarn** and Hadoop dfs (distributed file system) as shown below.

```
cd /$Hadoop_Home/sbin/
$ start-dfs.sh
localhost: starting namenode, logging to
/home/Hadoop/hadoop/logs/hadoopHadoop-namenode-localhost.localdomain.out
localhost: starting datanode, logging to
/home/Hadoop/hadoop/logs/hadoopHadoop-datanode-localhost.localdomain.out
Starting secondary namenodes [0.0.0.0]
starting secondarynamenode, logging to /home/Hadoop/hadoop/logs/hadoop-
Hadoopsecondarynamenode-localhost.localdomain.out

$ start-yarn.sh
starting yarn daemons
```

```
starting resourcemanager, logging to /home/Hadoop/hadoop/logs/yarn-
Hadoopresourcemanager-localhost.localdomain.out
localhost: starting nodemanager, logging to
/home/Hadoop/hadoop/logs/yarnHadoop-nodemanager-localhost.localdomain.out
```

Step 3: Create a Directory in HDFS

In Hadoop DFS, you can create directories using the command **mkdir**. Create a new directory in HDFS with the name **Pig_Data** in the required path as shown below.

```
$cd /$Hadoop_Home/bin/
$ hdfs dfs -mkdir hdfs://localhost:9000/Pig_Data
```

Step 4: Placing the data in HDFS

The input file of Pig contains each tuple/record in individual lines. And the entities of the record are separated by a delimiter (In our example we used **“,”**).

In the local file system, create an input file **student_data.txt** containing data as shown below.

```
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.
```

Now, move the file from the local file system to HDFS using **put** command as shown below. (You can use **copyFromLocal** command as well.)

```
$ cd $HADOOP_HOME/bin
$ hdfs dfs -put /home/Hadoop/Pig/Pig_Data/student_data.txt
dfs://localhost:9000/pig_data/
```

Verifying the file

You can use the **cat** command to verify whether the file has been moved into the HDFS, as shown below.

```
$ cd $HADOOP_HOME/bin
$ hdfs dfs -cat hdfs://localhost:9000/pig_data/student_data.txt
```

Output

You can see the content of the file as shown below.

```
15/10/01 12:16:55 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable

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
```

The Load Operator

You can load data into Apache Pig from the file system (HDFS/ Local) using **LOAD** operator of **Pig Latin**.

Syntax

The load statement consists of two parts divided by the "=" operator. On the left-hand side, we need to mention the name of the relation **where** we want to store the data, and on the right-hand side, we have to define **how** we store the data. Given below is the syntax of the **Load** operator.

```
Relation_name = LOAD 'Input file path' USING function as schema;
```

Where,

- **relation_name** − We have to mention the relation in which we want to store the data.
- **Input file path** − We have to mention the HDFS directory where the file is stored. (In MapReduce mode)
- **function** − We have to choose a function from the set of load functions provided by Apache Pig (**BinStorage, JsonLoader, PigStorage, TextLoader**).
- **Schema** − We have to define the schema of the data. We can define the required schema as follows −

```
(column1 : data type, column2 : data type, column3 : data type);
```

Note − We load the data without specifying the schema. In that case, the columns will be addressed as $01, $02, etc… (check).

Example

As an example, let us load the data in **student_data.txt** in Pig under the schema named **Student** using the **LOAD** command.

Start the Pig Grunt Shell

First of all, open the Linux terminal. Start the Pig Grunt shell in MapReduce mode as shown below.

```
$ Pig –x mapreduce
```

It will start the Pig Grunt shell as shown below.

```
15/10/01 12:33:37 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
15/10/01 12:33:37 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
15/10/01 12:33:37 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the
ExecType
```

```
2015-10-01 12:33:38,080 [main] INFO  org.apache.pig.Main - Apache Pig
version 0.15.0 (r1682971) compiled Jun 01 2015, 11:44:35
2015-10-01 12:33:38,080 [main] INFO  org.apache.pig.Main - Logging error
messages to: /home/Hadoop/pig_1443683018078.log
2015-10-01 12:33:38,242 [main] INFO  org.apache.pig.impl.util.Utils -
Default bootup file /home/Hadoop/.pigbootup not found

2015-10-01 12:33:39,630 [main]
INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine -
Connecting to hadoop file system at: hdfs://localhost:9000

grunt>
```

Execute the Load Statement

Now load the data from the file **student_data.txt** into Pig by executing the following Pig Latin statement in the Grunt shell.

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

Following is the description of the above statement.

Relation name	We have stored the data in the schema **student**.
Input file path	We are reading data from the file **student_data.txt,** which is in the /pig_data/ directory of HDFS.
Storage function	We have used the **PigStorage()** function. It loads and stores data as structured text files. It takes a delimiter using which each entity of a tuple is separated, as a parameter. By default, it takes '\t' as a parameter.
schema	We have stored the data using the following schema. column id firstname lastname phone city datatype int char array char array char array char array

Note − The **load** statement will simply load the data into the specified relation in Pig. To verify the execution of the **Load** statement, you have to use the **Diagnostic Operators** which are discussed in the next chapters.

This chapter explains how to store data in Apache Pig using the **Store** operator.

Syntax

Given below is the syntax of the Store statement.

```
STORE Relation_name INTO ' required_directory_path ' [USING function];
```

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 store the relation in the HDFS directory **"/pig_Output/"** as shown below.

```
grunt> STORE student INTO '/user/cloudera/Pig/pig_Output/ ' USING
PigStorage (',');
```

Output

After executing the **store** statement, you will get the following output. A directory is created with the specified name and the data will be stored in it.

```
2015-10-05 13:05:05,429 [main] INFO
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.
MapReduceLau ncher - 100% complete
2015-10-05 13:05:05,429 [main] INFO
org.apache.pig.tools.pigstats.mapreduce.SimplePigStats -
Script Statistics:

HadoopVersion    PigVersion    UserId    StartedAt             FinishedAt
Features
2.6.0            0.15.0        Hadoop    2015-10-0 13:03:03    2015-10-05
13:05:05    UNKNOWN
Success!
Job Stats (time in seconds):
JobId          Maps    Reduces    MaxMapTime    MinMapTime    AvgMapTime
MedianMapTime
job_14459_06    1        0           n/a           n/a           n/a
n/a
MaxReduceTime    MinReduceTime    AvgReduceTime    MedianReducetime
Alias    Feature
     0                0               0                0
student  MAP_ONLY
OutPut folder
hdfs://localhost:9000/pig_Output/

Input(s): Successfully read 0 records from:
"hdfs://localhost:9000/pig_data/student_data.txt"
Output(s): Successfully stored 0 records in:
"hdfs://localhost:9000/pig_Output"
Counters:
Total records written : 0
Total bytes written : 0
Spillable Memory Manager spill count : 0
```

```
Total bags proactively spilled: 0
Total records proactively spilled: 0

Job DAG: job_1443519499159_0006

2015-10-05 13:06:06,192 [main] INFO
org.apache.pig.backend.hadoop.executionengine
.mapReduceLayer.MapReduceLau ncher - Success!
```

Verification

You can verify the stored data as shown below.

Step 1

First of all, list out the files in the directory named **pig_output** using the **ls** command as shown below.

```
hdfs dfs -ls 'hdfs://localhost:9000/pig_Output/'
Found 2 items
rw-r--r-   1 Hadoop supergroup          0 2015-10-05 13:03
hdfs://localhost:9000/pig_Output/_SUCCESS
rw-r--r-   1 Hadoop supergroup        224 2015-10-05 13:03
hdfs://localhost:9000/pig_Output/part-m-00000
```

You can observe that two files were created after executing the **store** statement.

Step 2

Using **cat** command, list the contents of the file named **part-m-00000** as shown below.

```
$ hdfs dfs -cat 'hdfs://localhost:9000/pig_Output/part-m-00000'
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
```

The **load** statement will simply load the data into the specified relation in Apache Pig. To verify the execution of the **Load** statement, you have to use the **Diagnostic Operators**. Pig Latin provides four different types of diagnostic operators −

- Dump operator
- Describe operator
- Explanation operator
- Illustration operator

In this chapter, we will discuss the Dump operators of Pig Latin.

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

Given below is the 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
```

Once you execute the above **Pig Latin** statement, it will start a MapReduce job to read data from HDFS. It will produce the following output.

```
2015-10-01 15:05:27,642 [main]
INFO
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLaunc
her -
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,

Input(s): Successfully read 0 records from:
"hdfs://localhost:9000/pig_data/
student_data.txt"

Output(s): Successfully stored 0 records in:
"hdfs://localhost:9000/tmp/temp580182027/
tmp757878456"

Counters: Total records written : 0 Total bytes written : 0 Spillable
Memory Manager
spill count : 0Total bags proactively spilled: 0 Total records proactively
spilled: 0

Job DAG: job_1443519499159_0004

2015-10-01 15:06:28,403 [main]
INFO
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLau
ncher - Success!
2015-10-01 15:06:28,441 [main] INFO  org.apache.pig.data.SchemaTupleBackend
-
Key [pig.schematuple] was not set... will not generate code.
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 paths
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)
```

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

Syntax

The syntax of the **describe** operator is as follows −

```
grunt> Describe 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 describe the relation named **student** and verify the schema as shown below.

```
grunt> describe student;
```

Output

Once you execute the above **Pig Latin** statement, it will produce the following output.

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

The **explain** operator is used to display the logical, physical, and MapReduce execution plans of a relation.

Syntax

Given below is the syntax of the **explain** operator.

```
grunt> explain 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 explain the relation named student using the **explain** operator as shown below.

```
grunt> explain student;
```

Output

It will produce the following output.

```
$ explain student;

2015-10-05 11:32:43,660 [main]
2015-10-05 11:32:43,660 [main] INFO
org.apache.pig.newplan.logical.optimizer
.LogicalPlanOptimizer -
{RULES_ENABLED=[AddForEach, ColumnMapKeyPrune, ConstantCalculator,
GroupByConstParallelSetter, LimitOptimizer, LoadTypeCastInserter,
MergeFilter,
MergeForEach, PartitionFilterOptimizer, PredicatePushdownOptimizer,
PushDownForEachFlatten, PushUpFilter, SplitFilter, StreamTypeCastInserter]}
#-----------------------------------------------
# New Logical Plan:
#-----------------------------------------------
student: (Name: LOStore Schema:
id#31:int,firstname#32:chararray,lastname#33:chararray,phone#34:chararray,c
ity#
35:chararray)
|
|---student: (Name: LOForEach Schema:
id#31:int,firstname#32:chararray,lastname#33:chararray,phone#34:chararray,c
ity#
35:chararray)
    |   |
    |   (Name: LOGenerate[false,false,false,false,false] Schema:
id#31:int,firstname#32:chararray,lastname#33:chararray,phone#34:chararray,c
ity#
35:chararray)ColumnPrune:InputUids=[34, 35, 32, 33,
31]ColumnPrune:OutputUids=[34, 35, 32, 33, 31]
    |   |   |
    |   |   (Name: Cast Type: int Uid: 31)
    |   |   |     |   |   |---id:(Name: Project Type: bytearray Uid: 31
Input: 0 Column: (*))
    |   |   |
    |   |   (Name: Cast Type: chararray Uid: 32)
    |   |   |
    |   |   |---firstname:(Name: Project Type: bytearray Uid: 32 Input: 1
Column: (*))
    |   |   |
    |   |   (Name: Cast Type: chararray Uid: 33)
    |   |   |
    |   |   |---lastname:(Name: Project Type: bytearray Uid: 33 Input: 2
         Column: (*))
    |   |   |
    |   |   (Name: Cast Type: chararray Uid: 34)
    |   |   |
```

```
    |   |   |---phone:(Name: Project Type: bytearray Uid: 34 Input: 3
Column:
(*))
    |   |   |
    |   |   (Name: Cast Type: chararray Uid: 35)
    |   |   |
    |   |   |---city:(Name: Project Type: bytearray Uid: 35 Input: 4
Column:
(*))
    |   |
    |   |---(Name: LOInnerLoad[0] Schema: id#31:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[1] Schema: firstname#32:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[2] Schema: lastname#33:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[3] Schema: phone#34:bytearray)
    |   |
    |   |---(Name: LOInnerLoad[4] Schema: city#35:bytearray)
    |
    |---student: (Name: LOLoad Schema:
id#31:bytearray,firstname#32:bytearray,lastname#33:bytearray,phone#34:bytea
rray
,city#35:bytearray)RequiredFields:null
#-----------------------------------------------
# Physical Plan: #-----------------------------------------------
student: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-36
|
|---student: New For Each(false,false,false,false,false)[bag] - scope-35
    |   |
    |   Cast[int] - scope-21
    |   |
    |   |---Project[bytearray][0] - scope-20
    |   |
    |   Cast[chararray] - scope-24
    |   |
    |   |---Project[bytearray][1] - scope-23
    |   |
    |   Cast[chararray] - scope-27
    |   |
    |   |---Project[bytearray][2] - scope-26
    |   |
    |   Cast[chararray] - scope-30
    |   |
    |   |---Project[bytearray][3] - scope-29
    |   |
    |   Cast[chararray] - scope-33
    |   |
    |   |---Project[bytearray][4] - scope-32
    |
    |---student:
Load(hdfs://localhost:9000/pig_data/student_data.txt:PigStorage(',')) -
scope19
2015-10-05 11:32:43,682 [main]
INFO
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MRCompiler -
File concatenation threshold: 100 optimistic? false
2015-10-05 11:32:43,684 [main]
INFO
org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MultiQueryOp
timizer -
```

```
MR plan size before optimization: 1 2015-10-05 11:32:43,685 [main]
INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.
MultiQueryOp timizer - MR plan size after optimization: 1
#-----------------------------------------------
# Map Reduce Plan
#-----------------------------------------------
MapReduce node scope-37
Map Plan
student: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-36
|
|---student: New For Each(false,false,false,false,false)[bag] - scope-35
    |   |
    |   Cast[int] - scope-21
    |   |
    |   |---Project[bytearray][0] - scope-20
    |   |
    |   Cast[chararray] - scope-24
    |   |
    |   |---Project[bytearray][1] - scope-23
    |   |
    |   Cast[chararray] - scope-27
    |   |
    |   |---Project[bytearray][2] - scope-26
    |   |
    |   Cast[chararray] - scope-30
    |   |
    |   |---Project[bytearray][3] - scope-29
    |   |
    |   Cast[chararray] - scope-33
    |   |
    |   |---Project[bytearray][4] - scope-32
    |
    |---student:
Load(hdfs://localhost:9000/pig_data/student_data.txt:PigStorage(',')) -
scope
19-------- Global sort: false
 ----------------
```

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

Syntax

Given below is the syntax of the **illustrate** operator.

```
grunt> illustrate 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 illustrate the relation named student as shown below.

```
grunt> illustrate student;
```

Output

On executing the above statement, you will get the following output.

```
grunt> illustrate student;

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

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

Syntax

Given below is the syntax of the **group** operator.

```
grunt> Group_data = GROUP Relation_name BY age;
```

Example

Assume that we have a file named **student_details.txt** in the HDFS directory **/pig_data/** as shown below.

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
```

And we have loaded this file into Apache Pig with the relation name **student_details** as shown below.

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

Now, let us group the records/tuples in the relation by age as shown below.

```
grunt> group_data = GROUP student_details by age;
```

Verification

Verify the relation **group_data** using the **DUMP** operator as shown below.

```
grunt> Dump group_data;
```

Output

Then you will get output displaying the contents of the relation named **group_data** as shown below. Here you can observe that the resulting schema has two columns −

- One is **age**, by which we have grouped the relation.
- The other is a **bag**, which contains the group of tuples, student records with the respective age.

```
(21,{(4,Preethi,Agarwal,21,9848022330,Pune),(1,Rajiv,Reddy,21,9848022337,Hy
dera bad)})
(22,{(3,Rajesh,Khanna,22,9848022339,Delhi),(2,siddarth,Battacharya,22,98480
2233 8,Kolkata)})
(23,{(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,984802
2336 ,Bhuwaneshwar)})
(24,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,9848022
334, trivendram)})
```

You can see the schema of the table after grouping the data using the **describe** command as shown below.

```
grunt> Describe group_data;

group_data: {group: int,student_details: {(id: int,firstname: chararray,
               lastname: chararray,age: int,phone: chararray,city:
chararray)}}
```

In the same way, you can get the sample illustration of the schema using the **illustrate** command as shown below.

```
$ Illustrate group_data;
```

It will produce the following output −

```
-------------------------------------------------------------------------------
----------------------
|group_data|  group:int |
student_details:bag{:tuple(id:int,firstname:chararray,lastname:chararray,ag
e:int,phone:chararray,city:chararray)}|
-------------------------------------------------------------------------------
----------------------
|          |     21     | { 4, Preethi, Agarwal, 21, 9848022330, Pune), (1,
Rajiv, Reddy, 21, 9848022337, Hyderabad)}|
|          |     2      |
{(2,siddarth,Battacharya,22,9848022338,Kolkata),(003,Rajesh,Khanna,22,98480
22339,Delhi)}|
-------------------------------------------------------------------------------
----------------------
```

Grouping by Multiple Columns

Let us group the relation by age and city as shown below.

```
grunt> group_multiple = GROUP student_details by (age, city);
```

You can verify the content of the relation named **group_multiple** using the Dump operator as shown below.

```
grunt> Dump group_multiple;

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

Group All

You can group a relation by all the columns as shown below.

```
grunt> group_all = GROUP student_details All;
```

Now, verify the content of the relation **group_all** as shown below.

```
grunt> Dump group_all;

(all,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,984802
2334 ,trivendram),
(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,9848022336,
Bhuw aneshwar),
(4,Preethi,Agarwal,21,9848022330,Pune),(3,Rajesh,Khanna,22,9848022339,Delhi
),
```

```
(2,siddarth,Battacharya,22,9848022338,Kolkata),(1,Rajiv,Reddy,21,9848022337
,Hyd erabad)})
```

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/** as shown below.

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
```

And we have loaded these files into Pig with the relation names **student_details** and **employee_details** respectively, as shown below.

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

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

Now, let us group the records/tuples of the relations **student_details** and **employee_details** with the key age, as shown below.

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

Verification

Verify the relation **cogroup_data** using the **DUMP** operator as shown below.

```
grunt> Dump cogroup_data;
```

Output

It will produce the following output, displaying the contents of the relation named **cogroup_data** as shown below.

```
(21,{(4,Preethi,Agarwal,21,9848022330,Pune),
(1,Rajiv,Reddy,21,9848022337,Hyderabad)},
   {    })
(22,{ (3,Rajesh,Khanna,22,9848022339,Delhi),
(2,siddarth,Battacharya,22,9848022338,Kolkata) },
   { (6,Maggy,22,Chennai),(1,Robin,22,newyork) })
(23,{(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,984802
2336 ,Bhuwaneshwar)},
   {(5,David,23,Bhuwaneshwar),(3,Maya,23,Tokyo),(2,BOB,23,Kolkata)})
(24,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,9848022
334, trivendram)},
   { })
(25,{   },
   {(4,Sara,25,London)})
```

The **cogroup** operator groups the tuples from each relation according to age where each group depicts a particular age value.

For example, if we consider the 1st tuple of the result, it is grouped by age 21. And it contains two bags −

- the first bag holds all the tuples from the first relation (**student_details** in this case) having age 21, and
- the second bag contains all the tuples from the second relation (**employee_details** in this case) having age 21.

In case a relation doesn't have tuples having the age value 21, it returns an empty bag.

The **JOIN** operator is used to combine records from two or more relations. While performing a join operation, we declare one (or a group of) tuple(s) from each relation, as keys. When these keys match, the two particular tuples are matched, else the records are dropped. Joins can be of the following types −

- Self-join
- Inner-join
- Outer-join − left join, right join, and full join

This chapter explains with examples how to use the join operator in Pig Latin. Assume that we have two files namely **customers.txt** and **orders.txt** in the **/pig_data/** directory of HDFS as shown below.

customers.txt

```
1,Ramesh,32,Ahmedabad,2000.00
2,Khilan,25,Delhi,1500.00
3,kaushik,23,Kota,2000.00
```

```
4,Chaitali,25,Mumbai,6500.00
5,Hardik,27,Bhopal,8500.00
6,Komal,22,MP,4500.00
7,Muffy,24,Indore,10000.00
```

orders.txt

```
102,2009-10-08 00:00:00,3,3000
100,2009-10-08 00:00:00,3,1500
101,2009-11-20 00:00:00,2,1560
103,2008-05-20 00:00:00,4,2060
```

And we have loaded these two files into Pig with the relations **customers** and **orders** as shown below.

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

grunt> orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING
PigStorage(',')
   as (oid:int, date:chararray, customer_id:int, amount:int);
```

Let us now perform various Join operations on these two relations.

Self - join

Self-join is used to join a table with itself as if the table were two relations, temporarily renaming at least one relation.

Generally, in Apache Pig, to perform self-join, we will load the same data multiple times, under different aliases (names). Therefore let us load the contents of the file **customers.txt** as two tables as shown below.

```
grunt> customers1 = LOAD 'hdfs://localhost:9000/pig_data/customers.txt'
USING PigStorage(',')
   as (id:int, name:chararray, age:int, address:chararray, salary:int);

grunt> customers2 = LOAD 'hdfs://localhost:9000/pig_data/customers.txt'
USING PigStorage(',')
   as (id:int, name:chararray, age:int, address:chararray, salary:int);
```

Syntax

Given below is the syntax of performing **self-join** operation using the **JOIN** operator.

```
grunt> Relation3_name = JOIN Relation1_name BY key, Relation2_name BY key ;
```

Example

Let us perform **self-join** operation on the relation **customers**, by joining the two relations **customers1** and **customers2** as shown below.

```
grunt> customers3 = JOIN customers1 BY id, customers2 BY id;
```

Verification

Verify the relation **customers3** using the **DUMP** operator as shown below.

```
grunt> Dump customers3;
```

Output

It will produce the following output, displaying the contents of the relation **customers**.

```
(1,Ramesh,32,Ahmedabad,2000,1,Ramesh,32,Ahmedabad,2000)
(2,Khilan,25,Delhi,1500,2,Khilan,25,Delhi,1500)
(3,kaushik,23,Kota,2000,3,kaushik,23,Kota,2000)
(4,Chaitali,25,Mumbai,6500,4,Chaitali,25,Mumbai,6500)
(5,Hardik,27,Bhopal,8500,5,Hardik,27,Bhopal,8500)
(6,Komal,22,MP,4500,6,Komal,22,MP,4500)
(7,Muffy,24,Indore,10000,7,Muffy,24,Indore,10000)
```

Inner Join

Inner Join is used quite frequently; it is also referred to as **equijoin**. An inner join returns rows when there is a match in both tables.

It creates a new relation by combining column values of two relations (say A and B) based upon the join-predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, the column values for each matched pair of rows of A and B are combined into a result row.

Syntax

Here is the syntax of performing **inner join** operation using the **JOIN** operator.

```
grunt> result = JOIN relation1 BY columnname, relation2 BY columnname;
```

Example

Let us perform **inner join** operation on the two relations **customers** and **orders** as shown below.

```
grunt> coustomer_orders = JOIN customers BY id, orders BY customer_id;
```

Verification

Verify the relation **coustomer_orders** using the **DUMP** operator as shown below.

```
grunt> Dump coustomer_orders;
```

Output

You will get the following output that will the contents of the relation named **coustomer_orders**.

```
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(3,kaushik,23,Kota,2000,100,2009-10-08 00:00:00,3,1500)
(3,kaushik,23,Kota,2000,102,2009-10-08 00:00:00,3,3000)
(4,Chaitali,25,Mumbai,6500,103,2008-05-20 00:00:00,4,2060)
```

Note −

Outer Join: Unlike inner join, **outer join** returns all the rows from at least one of the relations. An outer join operation is carried out in three ways −

- Left outer join
- Right outer join
- Full outer join

Left Outer Join

The **left outer Join** operation returns all rows from the left table, even if there are no matches in the right relation.

Syntax

Given below is the syntax of performing **left outer join** operation using the **JOIN** operator.

```
grunt> Relation3_name = JOIN Relation1_name BY id LEFT OUTER,
Relation2_name BY customer_id;
```

Example

Let us perform left outer join operation on the two relations customers and orders as shown below.

```
grunt> outer_left = JOIN customers BY id LEFT OUTER, orders BY customer_id;
```

Verification

Verify the relation **outer_left** using the **DUMP** operator as shown below.

```
grunt> Dump outer_left;
```

Output

It will produce the following output, displaying the contents of the relation **outer_left**.

```
(1,Ramesh,32,Ahmedabad,2000,,,,)
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(3,kaushik,23,Kota,2000,100,2009-10-08 00:00:00,3,1500)
(3,kaushik,23,Kota,2000,102,2009-10-08 00:00:00,3,3000)
(4,Chaitali,25,Mumbai,6500,103,2008-05-20 00:00:00,4,2060)
```

```
(5,Hardik,27,Bhopal,8500,,,,)
(6,Komal,22,MP,4500,,,,)
(7,Muffy,24,Indore,10000,,,,)
```

Right Outer Join

The **right outer join** operation returns all rows from the right table, even if there are no matches in the left table.

Syntax

Given below is the syntax of performing **right outer join** operation using the **JOIN** operator.

```
grunt> outer_right = JOIN customers BY id RIGHT, orders BY customer_id;
```

Example

Let us perform **right outer join** operation on the two relations **customers** and **orders** as shown below.

```
grunt> outer_right = JOIN customers BY id RIGHT, orders BY customer_id;
```

Verification

Verify the relation **outer_right** using the **DUMP** operator as shown below.

```
grunt> Dump outer_right
```

Output

It will produce the following output, displaying the contents of the relation **outer_right**.

```
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(3,kaushik,23,Kota,2000,100,2009-10-08 00:00:00,3,1500)
(3,kaushik,23,Kota,2000,102,2009-10-08 00:00:00,3,3000)
(4,Chaitali,25,Mumbai,6500,103,2008-05-20 00:00:00,4,2060)
```

Full Outer Join

The **full outer join** operation returns rows when there is a match in one of the relations.

Syntax

Given below is the syntax of performing **full outer join** using the **JOIN** operator.

```
grunt> outer_full = JOIN customers BY id FULL OUTER, orders BY customer_id;
```

Example

Let us perform **full outer join** operation on the two relations **customers** and **orders** as shown below.

```
grunt> outer_full = JOIN customers BY id FULL OUTER, orders BY customer_id;
```

Verification

Verify the relation **outer_full** using the **DUMP** operator as shown below.

```
grun> Dump outer_full;
```

Output

It will produce the following output, displaying the contents of the relation **outer_full**.

```
(1,Ramesh,32,Ahmedabad,2000,,,,)
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(3,kaushik,23,Kota,2000,100,2009-10-08 00:00:00,3,1500)
(3,kaushik,23,Kota,2000,102,2009-10-08 00:00:00,3,3000)
(4,Chaitali,25,Mumbai,6500,103,2008-05-20 00:00:00,4,2060)
(5,Hardik,27,Bhopal,8500,,,,)
(6,Komal,22,MP,4500,,,,)
(7,Muffy,24,Indore,10000,,,,)
```

Using Multiple Keys

We can perform JOIN operation using multiple keys.

Syntax

Here is how you can perform a JOIN operation on two tables using multiple keys.

```
grunt> Relation3_name = JOIN Relation2_name BY (key1, key2), Relation3_name
BY (key1, key2);
```

Assume that we have two files namely **employee.txt** and **employee_contact.txt** in the **/pig_data/** directory of HDFS as shown below.

employee.txt

```
001,Rajiv,Reddy,21,programmer,003
002,siddarth,Battacharya,22,programmer,003
003,Rajesh,Khanna,22,programmer,003
004,Preethi,Agarwal,21,programmer,003
005,Trupthi,Mohanthy,23,programmer,003
006,Archana,Mishra,23,programmer,003
007,Komal,Nayak,24,teamlead,002
008,Bharathi,Nambiayar,24,manager,001
```

employee_contact.txt

```
001,9848022337,Rajiv@gmail.com,Hyderabad,003
002,9848022338,siddarth@gmail.com,Kolkata,003
```

```
003,9848022339,Rajesh@gmail.com,Delhi,003
004,9848022330,Preethi@gmail.com,Pune,003
005,9848022336,Trupthi@gmail.com,Bhuwaneshwar,003
006,9848022335,Archana@gmail.com,Chennai,003
007,9848022334,Komal@gmail.com,trivendram,002
008,9848022333,Bharathi@gmail.com,Chennai,001
```

And we have loaded these two files into Pig with relations **employee** and **employee_contact** as shown below.

```
grunt> employee = LOAD 'hdfs://localhost:9000/pig_data/employee.txt' USING
PigStorage(',')
   as (id:int, firstname:chararray, lastname:chararray, age:int,
designation:chararray, jobid:int);

grunt> employee_contact = LOAD
'hdfs://localhost:9000/pig_data/employee_contact.txt' USING PigStorage(',')
   as (id:int, phone:chararray, email:chararray, city:chararray,
jobid:int);
```

Now, let us join the contents of these two relations using the **JOIN** operator as shown below.

```
grunt> emp = JOIN employee BY (id,jobid), employee_contact BY (id,jobid);
```

Verification

Verify the relation **emp** using the **DUMP** operator as shown below.

```
grunt> Dump emp;
```

Output

It will produce the following output, displaying the contents of the relation named **emp** as shown below.

```
(1,Rajiv,Reddy,21,programmer,113,1,9848022337,Rajiv@gmail.com,Hyderabad,113
)
(2,siddarth,Battacharya,22,programmer,113,2,9848022338,siddarth@gmail.com,K
olka ta,113)
(3,Rajesh,Khanna,22,programmer,113,3,9848022339,Rajesh@gmail.com,Delhi,113)
(4,Preethi,Agarwal,21,programmer,113,4,9848022330,Preethi@gmail.com,Pune,11
3)
(5,Trupthi,Mohanthy,23,programmer,113,5,9848022336,Trupthi@gmail.com,Bhuwan
eshw ar,113)
(6,Archana,Mishra,23,programmer,113,6,9848022335,Archana@gmail.com,Chennai,
113)
(7,Komal,Nayak,24,teamlead,112,7,9848022334,Komal@gmail.com,trivendram,112)
(8,Bharathi,Nambiayar,24,manager,111,8,9848022333,Bharathi@gmail.com,Chenna
i,111)
```

The **CROSS** operator computes the cross-product of two or more relations. This chapter explains with example how to use the cross operator in Pig Latin.

Syntax

Given below is the syntax of the **CROSS** operator.

```
grunt> Relation3_name = CROSS Relation1_name, Relation2_name;
```

Example

Assume that we have two files namely **customers.txt** and **orders.txt** in the **/pig_data/** directory of HDFS as shown below.

customers.txt

```
1,Ramesh,32,Ahmedabad,2000.00
2,Khilan,25,Delhi,1500.00
3,kaushik,23,Kota,2000.00
4,Chaitali,25,Mumbai,6500.00
5,Hardik,27,Bhopal,8500.00
6,Komal,22,MP,4500.00
7,Muffy,24,Indore,10000.00
```

orders.txt

```
102,2009-10-08 00:00:00,3,3000
100,2009-10-08 00:00:00,3,1500
101,2009-11-20 00:00:00,2,1560
103,2008-05-20 00:00:00,4,2060
```

And we have loaded these two files into Pig with the relations **customers** and **orders** as shown below.

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

grunt> orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING
PigStorage(',')
   as (oid:int, date:chararray, customer_id:int, amount:int);
```

Let us now get the cross-product of these two relations using the **cross** operator on these two relations as shown below.

```
grunt> cross_data = CROSS customers, orders;
```

Verification

Verify the relation **cross_data** using the **DUMP** operator as shown below.

```
grunt> Dump cross_data;
```

Output

It will produce the following output, displaying the contents of the relation **cross_data**.

```
(7,Muffy,24,Indore,10000,103,2008-05-20 00:00:00,4,2060)
```

(7,Muffy,24,Indore,10000,101,2009-11-20 00:00:00,2,1560)
(7,Muffy,24,Indore,10000,100,2009-10-08 00:00:00,3,1500)
(7,Muffy,24,Indore,10000,102,2009-10-08 00:00:00,3,3000)
(6,Komal,22,MP,4500,103,2008-05-20 00:00:00,4,2060)
(6,Komal,22,MP,4500,101,2009-11-20 00:00:00,2,1560)
(6,Komal,22,MP,4500,100,2009-10-08 00:00:00,3,1500)
(6,Komal,22,MP,4500,102,2009-10-08 00:00:00,3,3000)
(5,Hardik,27,Bhopal,8500,103,2008-05-20 00:00:00,4,2060)
(5,Hardik,27,Bhopal,8500,101,2009-11-20 00:00:00,2,1560)
(5,Hardik,27,Bhopal,8500,100,2009-10-08 00:00:00,3,1500)
(5,Hardik,27,Bhopal,8500,102,2009-10-08 00:00:00,3,3000)
(4,Chaitali,25,Mumbai,6500,103,2008-05-20 00:00:00,4,2060)
(4,Chaitali,25,Mumbai,6500,101,2009-20 00:00:00,4,2060)
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(2,Khilan,25,Delhi,1500,100,2009-10-08 00:00:00,3,1500)
(2,Khilan,25,Delhi,1500,102,2009-10-08 00:00:00,3,3000)
(1,Ramesh,32,Ahmedabad,2000,103,2008-05-20 00:00:00,4,2060)
(1,Ramesh,32,Ahmedabad,2000,101,2009-11-20 00:00:00,2,1560)
(1,Ramesh,32,Ahmedabad,2000,100,2009-10-08 00:00:00,3,1500)
(1,Ramesh,32,Ahmedabad,2000,102,2009-10-08 00:00:00,3,3000)-11-20
00:00:00,2,1560)
(4,Chaitali,25,Mumbai,6500,100,2009-10-08 00:00:00,3,1500)
(4,Chaitali,25,Mumbai,6500,102,2009-10-08 00:00:00,3,3000)
(3,kaushik,23,Kota,2000,103,2008-05-20 00:00:00,4,2060)
(3,kaushik,23,Kota,2000,101,2009-11-20 00:00:00,2,1560)
(3,kaushik,23,Kota,2000,100,2009-10-08 00:00:00,3,1500)
(3,kaushik,23,Kota,2000,102,2009-10-08 00:00:00,3,3000)
(2,Khilan,25,Delhi,1500,103,2008-05-20 00:00:00,4,2060)
(2,Khilan,25,Delhi,1500,101,2009-11-20 00:00:00,2,1560)
(2,Khilan,25,Delhi,1500,100,2009-10-08 00:00:00,3,1500)
(2,Khilan,25,Delhi,1500,102,2009-10-08 00:00:00,3,3000)
(1,Ramesh,32,Ahmedabad,2000,103,2008-05-20 00:00:00,4,2060)
(1,Ramesh,32,Ahmedabad,2000,101,2009-11-20 00:00:00,2,1560)
(1,Ramesh,32,Ahmedabad,2000,100,2009-10-08 00:00:00,3,1500)
(1,Ramesh,32,Ahmedabad,2000,102,2009-10-08 00:00:00,3,3000)