University of East London School of Architecture, Computing and Engineering (ACE)

Author: Dr Fahimeh Jafari (f.jafari@uel.ac.uk) 25-26

Tutorial 4: Apache Hive with Cloudera

CN7031 - Big Data Analytics

Dr Fahimeh Jafari (f.jafari@uel.ac.uk)

LEARNING OUTCOMES: After completing this tutorial, you should:

- Have gotten a hands-on experience in deploying Apache Hive
- Create Database/Table/view in Hive
- Create queries using HiveQL
- Join Queries
- Analytic functions in Hive
- Hive Optimisation

Tutorial Submission [OPTIONAL]

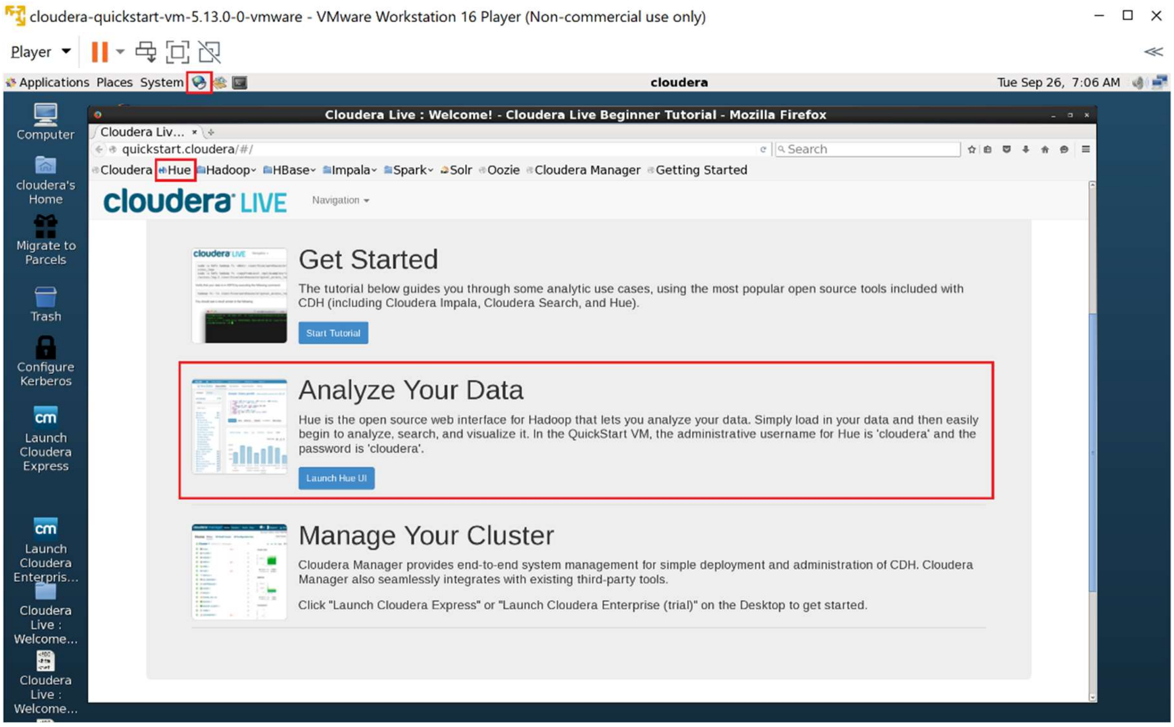
You can submit the results of your work by taking screenshots (wherever pointed), pasting them into a Word file and sending the Word/pdf file through the submission link provided in Moodle.

Exercise 1: Getting Ready for Hue Cloudera

- Launch the “cloudera-quickstart-vm” from VMWare workstation. It takes a couple of minutes to be loaded.
- Like the previous session, click on **Launch Hue UI** icon on the browser and enter Username and Password. To avoid any confusion, the username and password are given below.

  Username: cloudera

  Password: cloudera OR BigDataadmin



cloudera-quickstart-vm-5.13.0-0-vmware - VMware Workstation 16 Player (Non-commercial use only)

Cloudera Live : Welcome! - Cloudera Live Beginner Tutorial - Mozilla Firefox

quickstart.cloudera/#/

Get Started

The tutorial below guides you through some analytic use cases, using the most popular open source tools included with CDH (including Cloudera Impala, Cloudera Search, and Hue).

Start Tutorial

Analyze Your Data

Hue is the open source web interface for Hadoop that lets you analyze your data. Simply load in your data and then easily begin to analyze, search, and visualize it. In the QuickStart VM, the administrative username for Hue is 'cloudera' and the password is 'cloudera'.

Launch Hue UI

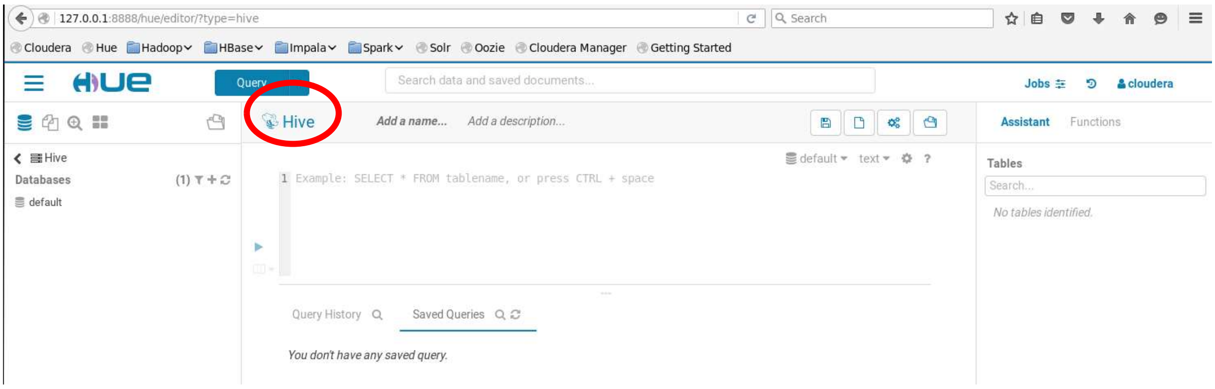
Manage Your Cluster

Cloudera Manager provides end-to-end system management for simple deployment and administration of CDH. Cloudera Manager also seamlessly integrates with existing third-party tools.

Click "Launch Cloudera Express" or "Launch Cloudera Enterprise (trial)" on the Desktop to get started.

Applications Places System
cloudera
Tue Sep 26, 7:06 AM
Computer
cloudera's Home
Migrate to Parcels
Trash
Configure Kerberos
Launch Cloudera Express
Launch Cloudera Enterpris...
Cloudera Live : Welcome...
Cloudera Hue Hadoop HBase Impala Spark Solr Oozie Cloudera Manager Getting Started
cloudera LIVE
Navigation

- Once you are inside the Hue editor, click on **Query → Editors → Hive** to launch the Hive Editor. Finally, you will see the Hive editor as follows:

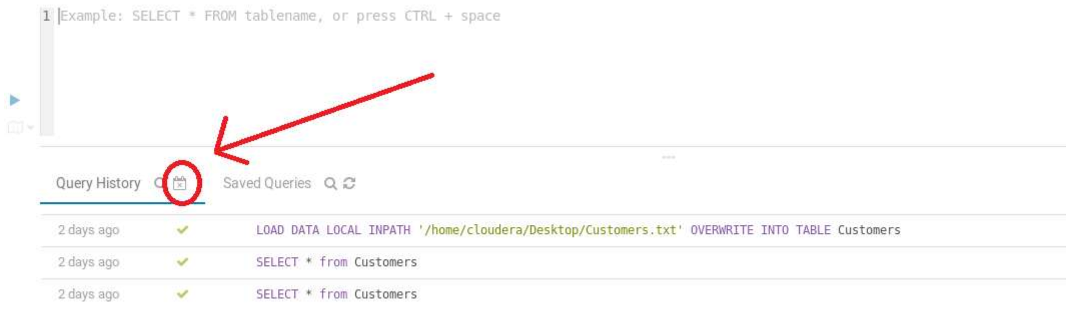


127.0.0.1:8888/hue/editor/?type=hive

```
1 Example: SELECT * FROM tablename, or press CTRL + space
```

You don't have any saved query.

- If you don't want to hold the queries in the history, easily clean it up to keep your work original.



```
1 Example: SELECT * FROM tablename, or press CTRL + space
```

```
2 days ago   ✓   LOAD DATA LOCAL INPATH '/home/cloudera/Desktop/Customers.txt' OVERWRITE INTO TABLE Customers
2 days ago   ✓   SELECT * from Customers
2 days ago   ✓   SELECT * from Customers
```

Exercise 2: Create Database/Table

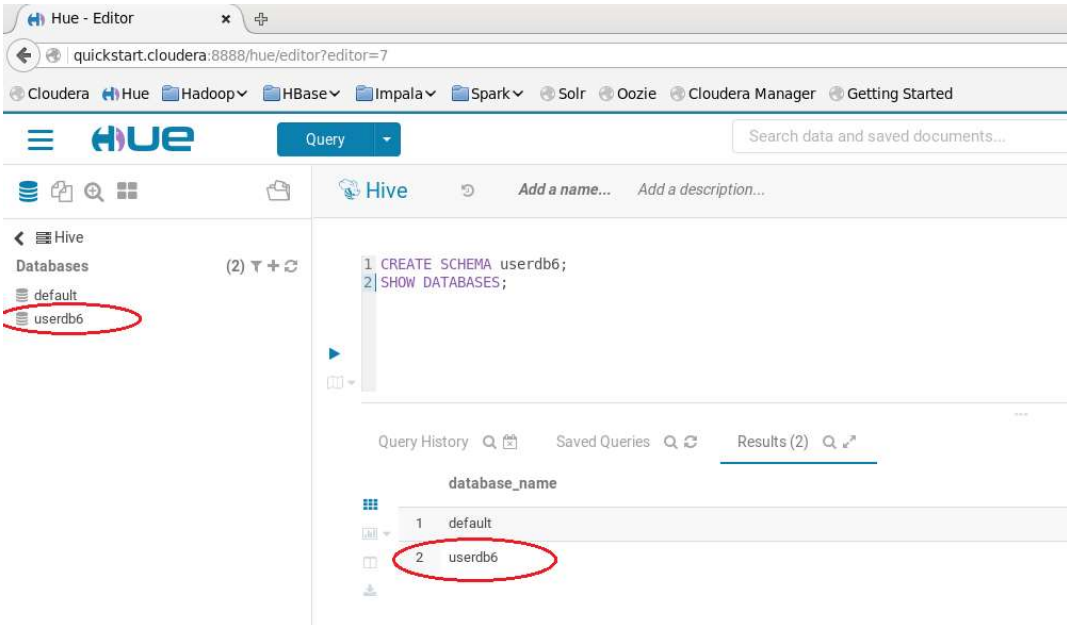
Step 1: Create a database and table

a) Create a database, named `“userdb”+ your lab group number`, by executing the following query in the editor. For example, if your lab group number is 6, create a database with name `userdb6`

```
CREATE SCHEMA userdb6;
```

b) See the created databases from the left side of the editor, or simply by:

```
SHOW DATABASES;
```



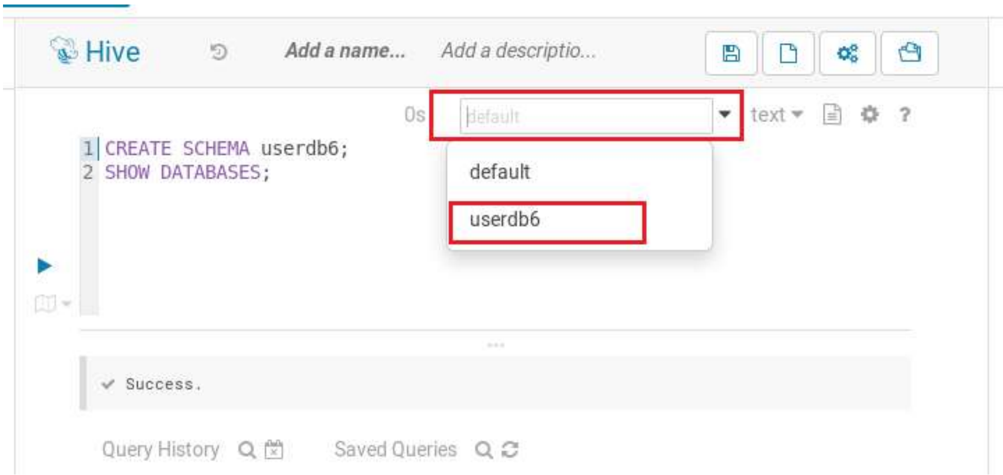
```
CREATE SCHEMA userdb6;
SHOW DATABASES;
```

```
   database_name
1  default
2  userdb6
```

c) Execute the following query to make the next queries in database `userdb6`.

```
USE userdb6;
```

OR you can simply select `userdb6` as shown in the Fig.



```
CREATE SCHEMA userdb6;
SHOW DATABASES;
```

```
Success.
```

d) Create an employee table as follows:

Sr.No	Field Name	Data Type
1	Eid	int
2	Name	String
3	Salary	Float
4	Designation	string

The following query will create an empty table named *employee* and allows us to import a *txt* file with the determined delimiters and line separator in.

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

Step 2: Load Data

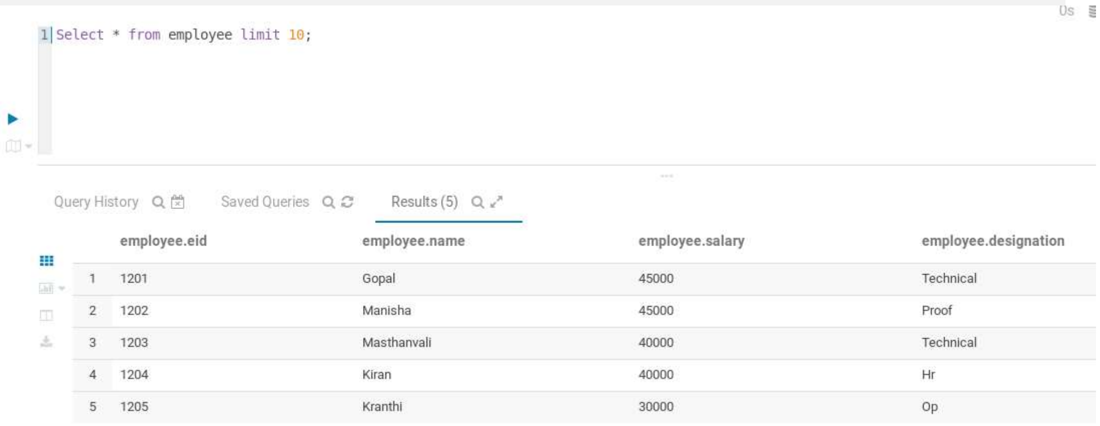
Generally, after creating a table in SQL, we can insert data using the Insert statement. But in Hive, we can insert data using the LOAD DATA statement.

a) Download `sample.txt` file from Moodle and put it in *Desktop*.
b) Now, we import the data in the `sample.txt` file into the `employee` table using the following query:

```
LOAD DATA LOCAL INPATH '/home/cloudera/Desktop/Sample.txt'
OVERWRITE INTO TABLE employee;
```

c) Let's show the content of *employee* table.

```
Select * from employee limit 10;
```



```
Select * from employee limit 10;
```

	employee.eid	employee.name	employee.salary	employee.designation
1	1201	Gopal	45000	Technical
2	1202	Manisha	45000	Proof
3	1203	Masthanvali	40000	Technical
4	1204	Kiran	40000	Hr
5	1205	Kranthi	30000	Op

(Take Screenshot #1)

d) Assume we need to alter *employee* table by adding a new column. Because the `sample.txt` file has 5 columns and we missed the last one.

```
ALTER TABLE employee ADD COLUMNS (dept STRING COMMENT 'Department name');
```

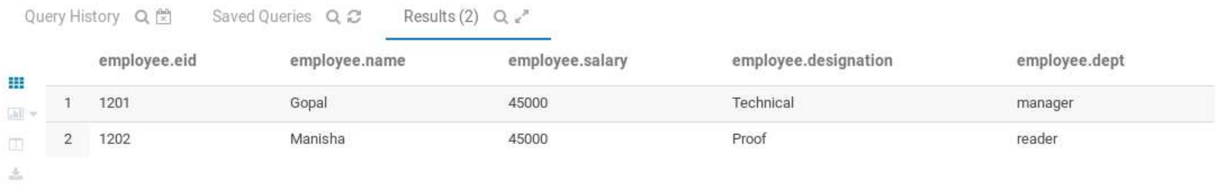
Exercise 3: Working with Hive queries

Step 1: Retrieving information

Let's practice some of retrieving Hive queries:

a) Make a `SELECT … WHERE` query.

```
SELECT * FROM employee WHERE salary > 40000;
```

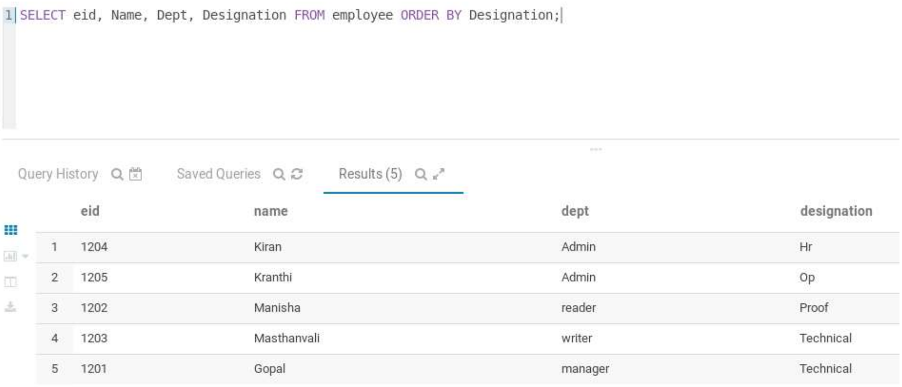


	employee.eid	employee.name	employee.salary	employee.designation	employee.dept
1	1201	Gopal	45000	Technical	manager
2	1202	Manisha	45000	Proof	reader

(Take Screenshot #2)

b) Make a `SELECT … ORDER BY` query.

```
SELECT eid, Name, Dept, Designation FROM employee ORDER BY
Designation;
```



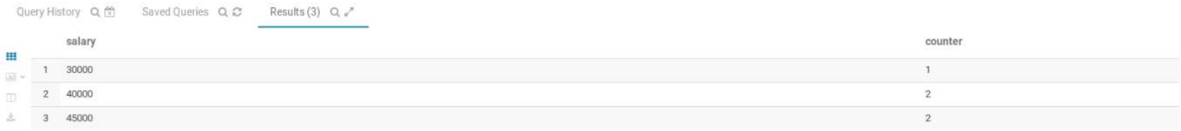
```
SELECT eid, Name, Dept, Designation FROM employee ORDER BY Designation;
```

	eid	name	dept	designation
1	1204	Kiran	Admin	Hr
2	1205	Kranthi	Admin	Op
3	1202	Manisha	reader	Proof
4	1203	Masthanvali	writer	Technical
5	1201	Gopal	manager	Technical

(Take Screenshot #3)

c) Make a `SELECT … GROUP BY` query.

```
SELECT Salary, count(*) as Counter FROM employee GROUP BY
Salary
```



	salary	counter
1	30000	1
2	40000	2
3	45000	2

(Take Screenshot #4)

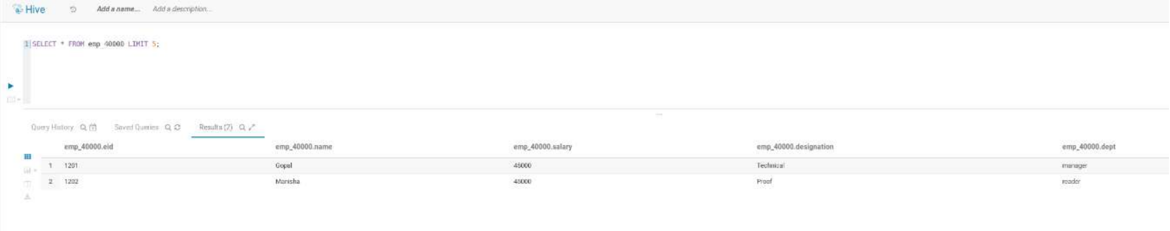
Step 2: Create a view

a) A *view* is a SQL statement that is stored in the database with an associated name as a 'virtual table'. (**check the database lists in the left, to see emp_40000 view**)

```
CREATE VIEW emp_40000 AS SELECT * FROM employee WHERE
salary>40000;
```

b) Then, list the `emp_40000` view by limiting the first 5 records:

```
SELECT * FROM emp_40000 LIMIT 5;
```



```
SELECT * FROM emp_40000 LIMIT 5;
```

	emp_40000.eid	emp_40000.name	emp_40000.salary	emp_40000.designation	emp_40000.dept
1	1201	Gopal	45000	Technical	manager
2	1202	Manisha	45000	Proof	reader

(Take Screenshot #5)

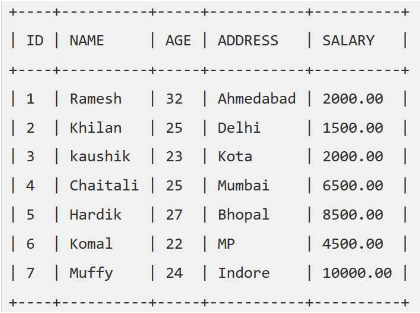
c) Finally, if you wish to drop the view:

```
DROP VIEW emp_40000;
```

Step 3: Join queries

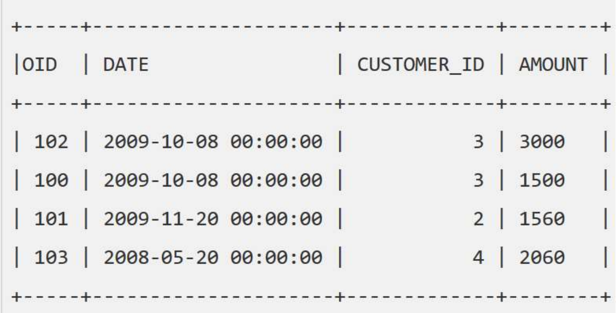
Download `Customesr.txt` and `Orders.txt` from Moodle and put them in the Desktop. Then, create tables CUSTOMERS and ORDERS and load the data into them.

CUSTOMERS:



```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 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:



```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 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   |
+-----+---------------------+-------------+--------+
```

a) Create a table for Customers:

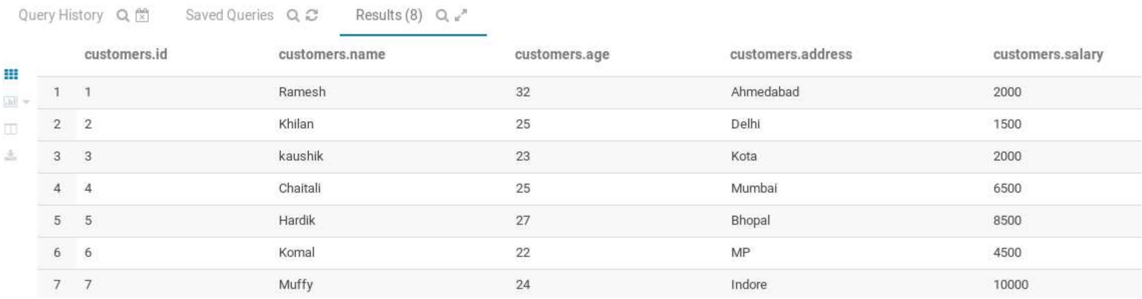
```
CREATE TABLE IF NOT EXISTS Customers (ID int, Name String, Age
int, Address String, Salary int) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' LINES TERMINATED BY '\n' STORED AS TEXTFILE;
```

b) Load the Data:

```
LOAD  DATA  LOCAL  INPATH  '/home/cloudera/Desktop/Customers.txt'
OVERWRITE INTO TABLE Customers;
```

c) List the data:

```
SELECT * from Customers limit 10;
```



	customers.id	customers.name	customers.age	customers.address	customers.salary
1	1	Ramesh	32	Ahmedabad	2000
2	2	Khilan	25	Delhi	1500
3	3	kaushik	23	Kota	2000
4	4	Chaitali	25	Mumbai	6500
5	5	Hardik	27	Bhopal	8500
6	6	Komal	22	MP	4500
7	7	Muffy	24	Indore	10000

(Take Screenshot #6)

d) Create a table for Orders:

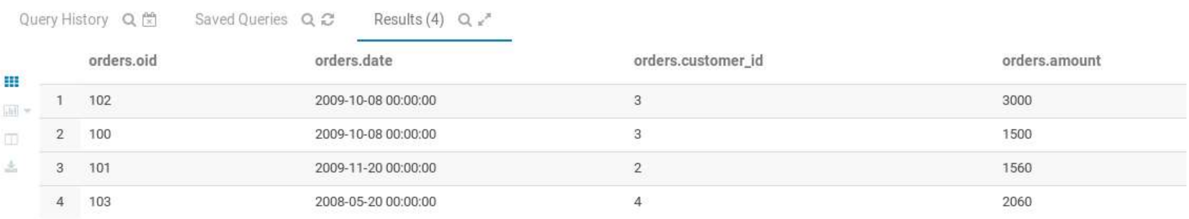
```
CREATE  TABLE  IF  NOT  EXISTS  Orders  (OID  int,  Date  String,
Customer_ID int, Amount int) ROW FORMAT DELIMITED FIELDS TERMINATED
BY '\t' LINES TERMINATED BY '\n' STORED AS TEXTFILE;
```

e) Load the Data:

```
LOAD   DATA   LOCAL   INPATH   '/home/cloudera/Desktop/Orders.txt'
OVERWRITE INTO TABLE Orders;
```

f) List the data:

```
SELECT * from Orders;
```

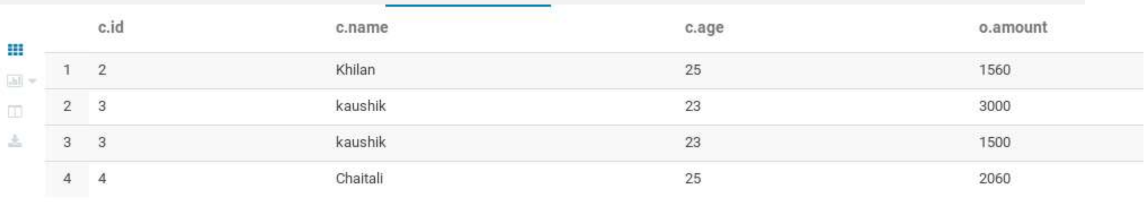


	orders.oid	orders.date	orders.customer_id	orders.amount
1	102	2009-10-08 00:00:00	3	3000
2	100	2009-10-08 00:00:00	3	1500
3	101	2009-11-20 00:00:00	2	1560
4	103	2008-05-20 00:00:00	4	2060

(Take Screenshot #7)

g) *Join* Query:

```
SELECT c.ID, c.NAME, c.AGE, o.AMOUNT FROM CUSTOMERS c JOIN
ORDERS o ON (c.ID = o.CUSTOMER_ID);
```

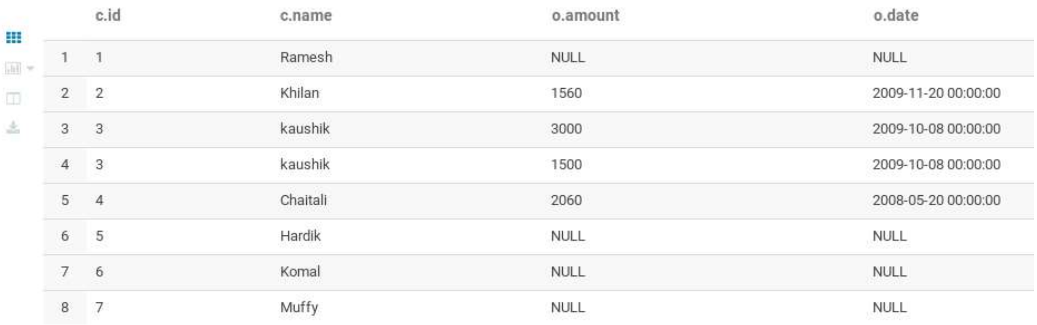


	c.id	c.name	c.age	o.amount
1	2	Khilan	25	1560
2	3	kaushik	23	3000
3	3	kaushik	23	1500
4	4	Chaitali	25	2060

(Take Screenshot #8)

h) Make a LEFT OUTER JOIN between CUSTOMER and ORDER tables:

```
SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c LEFT
OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);
```

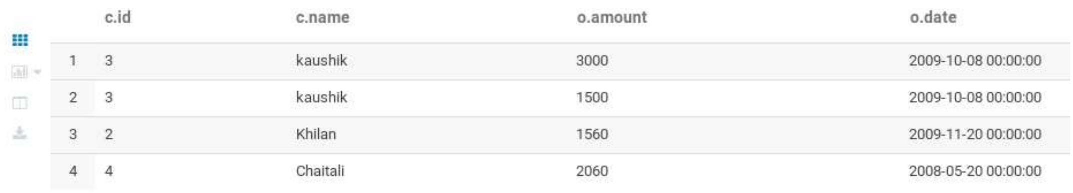


	c.id	c.name	o.amount	o.date
1	1	Ramesh	NULL	NULL
2	2	Khilan	1560	2009-11-20 00:00:00
3	3	kaushik	3000	2009-10-08 00:00:00
4	3	kaushik	1500	2009-10-08 00:00:00
5	4	Chaitali	2060	2008-05-20 00:00:00
6	5	Hardik	NULL	NULL
7	6	Komal	NULL	NULL
8	7	Muffy	NULL	NULL

(Take Screenshot #9)

i) Make a RIGHT OUTER JOIN between CUSTOMER and ORDER tables:

```
SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c RIGHT
OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);
```

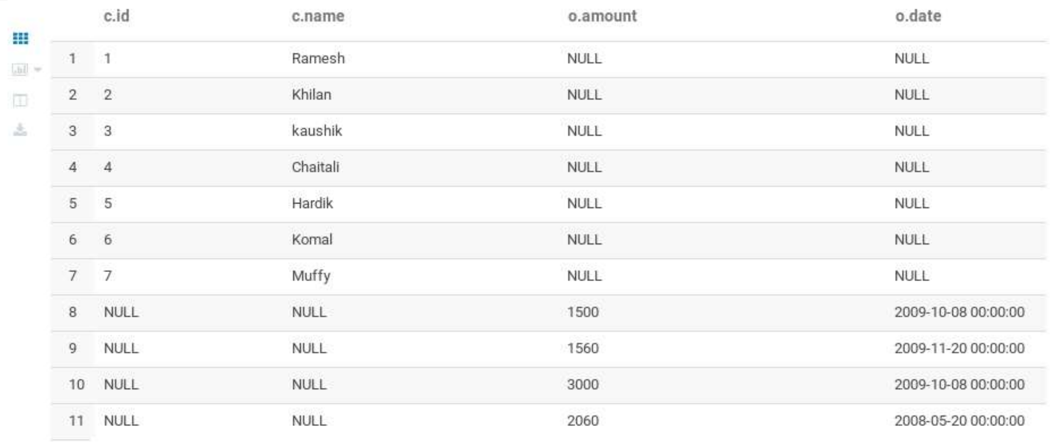


	c.id	c.name	o.amount	o.date
1	3	kaushik	3000	2009-10-08 00:00:00
2	3	kaushik	1500	2009-10-08 00:00:00
3	2	Khilan	1560	2009-11-20 00:00:00
4	4	Chaitali	2060	2008-05-20 00:00:00

(Take Screenshot #10)

j) Make a FULL OUTER JOIN between CUSTOMER and ORDER tables:

```
SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c FULL
OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);
```



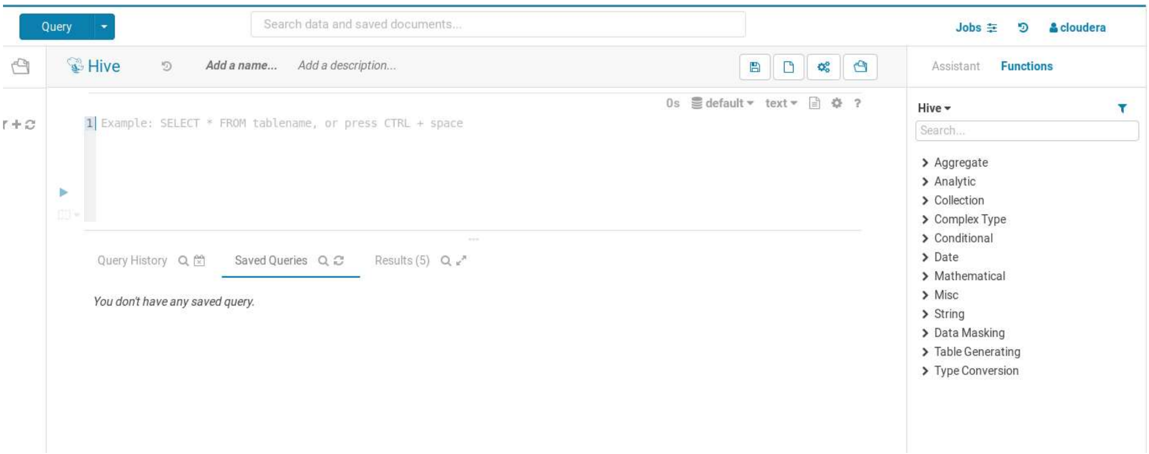
	c.id	c.name	o.amount	o.date
1	1	Ramesh	NULL	NULL
2	2	Khilan	NULL	NULL
3	3	kaushik	NULL	NULL
4	4	Chaitali	NULL	NULL
5	5	Hardik	NULL	NULL
6	6	Komal	NULL	NULL
7	7	Muffy	NULL	NULL
8	NULL	NULL	1500	2009-10-08 00:00:00
9	NULL	NULL	1560	2009-11-20 00:00:00
10	NULL	NULL	3000	2009-10-08 00:00:00
11	NULL	NULL	2060	2008-05-20 00:00:00

(Take Screenshot #11)

More SQL Considerations for Self-Study
(not included in the tutorial recording)

Step 4: Analytic Functions

You can use the “Hive functions” on the right-hand side of the screen.



You don't have any saved query.

Analytic functions are a special group of functions that scan the multiple input rows to compute each output value. Analytic functions are usually used with OVER, PARTITION BY, ORDER BY, and the windowing specification. The analytic functions offer greater flexibility and functionalities than the regular GROUP BY clause and make special aggregations in Hive easier and more powerful.

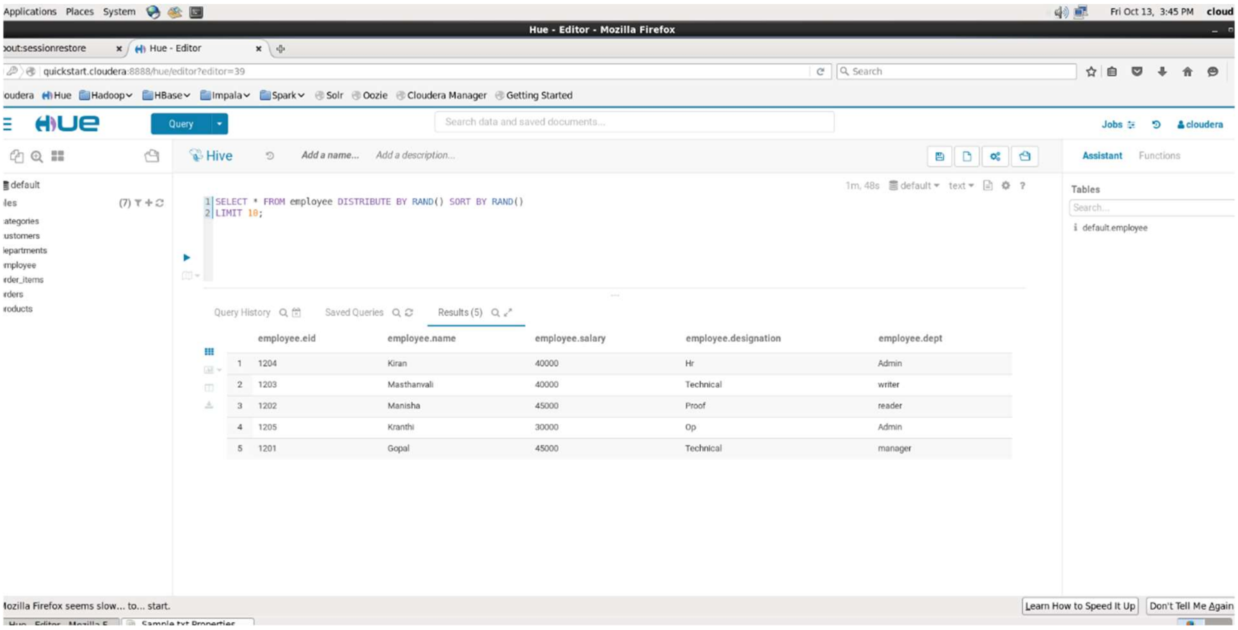
The Analytic functions are:

- RANK
- DENSE_RANK
- ROW_NUMBER
- CUME_DIST
- PERCENT_RANK
- NTILE

RANK ():

The rank function ranks items in a group, such as finding the top N rows for specific conditions. This function is used to assign a rank to the rows based on the column values in an OVER clause. The row with equal values is assigned the same rank with the next rank value skipped.

```
hive> SELECT * FROM employee DISTRIBUTE BY RAND() SORT BY RAND()
LIMIT 10;
```



```
SELECT * FROM employee DISTRIBUTE BY RAND() SORT BY RAND()
LIMIT 10;
```

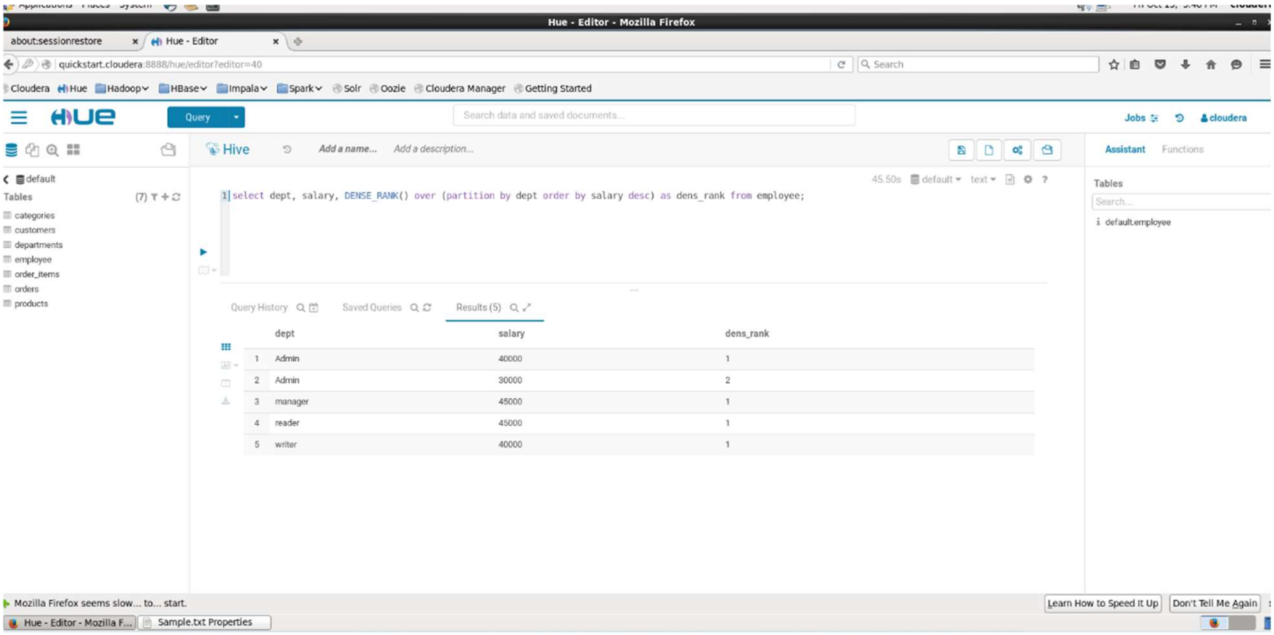
	employee.eid	employee.name	employee.salary	employee.designation	employee.dept
1	1204	Kiran	40000	Hr	Admin
2	1203	Masthanvali	40000	Technical	writer
3	1202	Manisha	45000	Proof	reader
4	1205	Kranthi	30000	Op	Admin
5	1201	Gopal	45000	Technical	manager

(Take Screenshot #12)

DENSE_RANK ():

It is similar to RANK but leaves no gaps in the ranking sequence when there are ties. the rank is assigned in sequential order so that no rank values are skipped. DENSE_RANK() function returns consecutive rank values. Rows in each partition receive the same ranks if they have the same values.

```
hive> select dept, salary, DENSE_RANK() over (partition by dept
order by salary desc) as dens_rank from employee;
```



```
select dept, salary, DENSE_RANK() over (partition by dept order by salary desc) as dens_rank from employee;
```

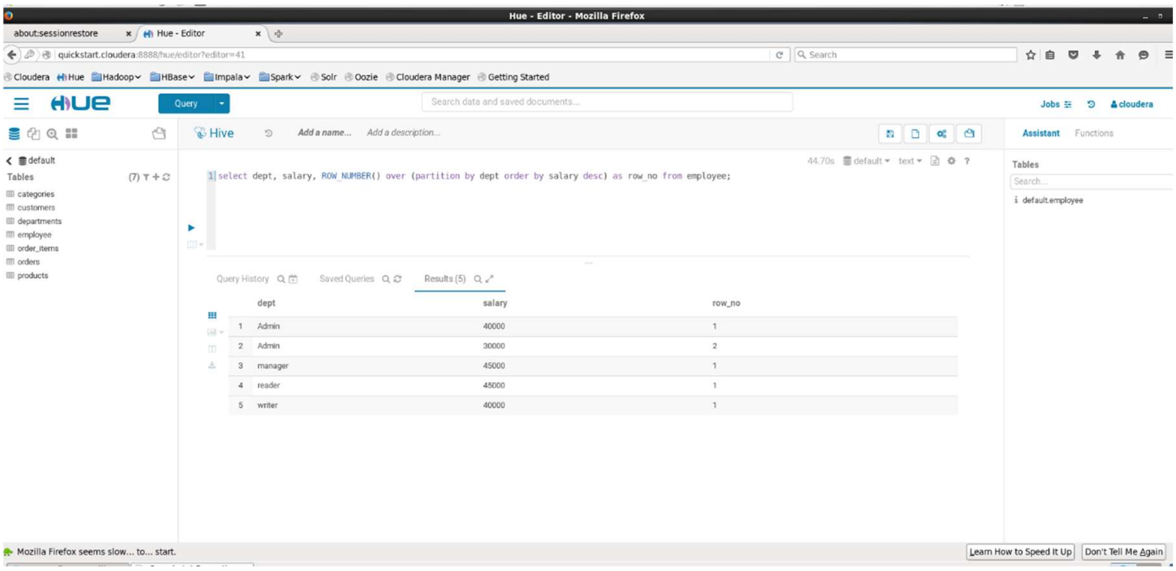
	dept	salary	dens_rank
1	Admin	40000	1
2	Admin	30000	2
3	manager	45000	1
4	reader	45000	1
5	writer	40000	1

(Take Screenshot #13)

ROW_NUMBER():

This function is used to assign a unique sequence number starting from 1 to each row according to the partition and order specification.

```
hive> select dept, salary, ROW_NUMBER() over (partition by dept
order by salary desc) as row_no from employee;
```



```
select dept, salary, ROW_NUMBER() over (partition by dept order by salary desc) as row_no from employee;
```

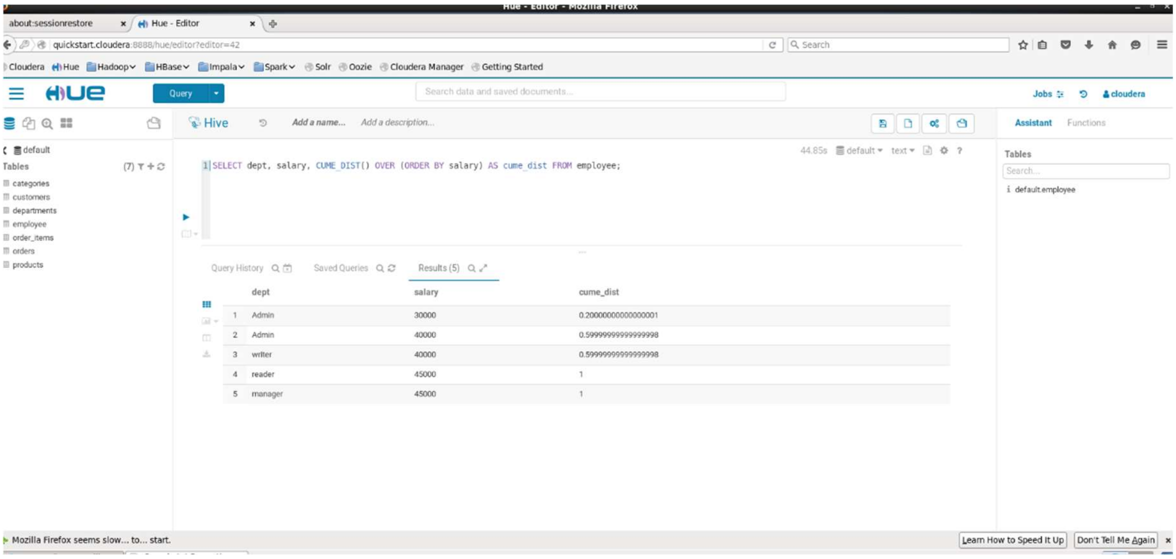
	dept	salary	row_no
1	Admin	40000	1
2	Admin	30000	2
3	manager	45000	1
4	reader	45000	1
5	writer	40000	1

(Take Screenshot #14)

CUME_DIST():

It computes the number of rows whose value is smaller or equal to the value of the total number of rows divided by the current row.it gives the values in the float data type.

```
 hive> SELECT dept, salary, CUME_DIST() OVER (ORDER BY salary) AS
cume_dist FROM employee;
```



```
SELECT dept, salary, CUME_DIST() OVER (ORDER BY salary) AS cume_dist FROM employee;
```

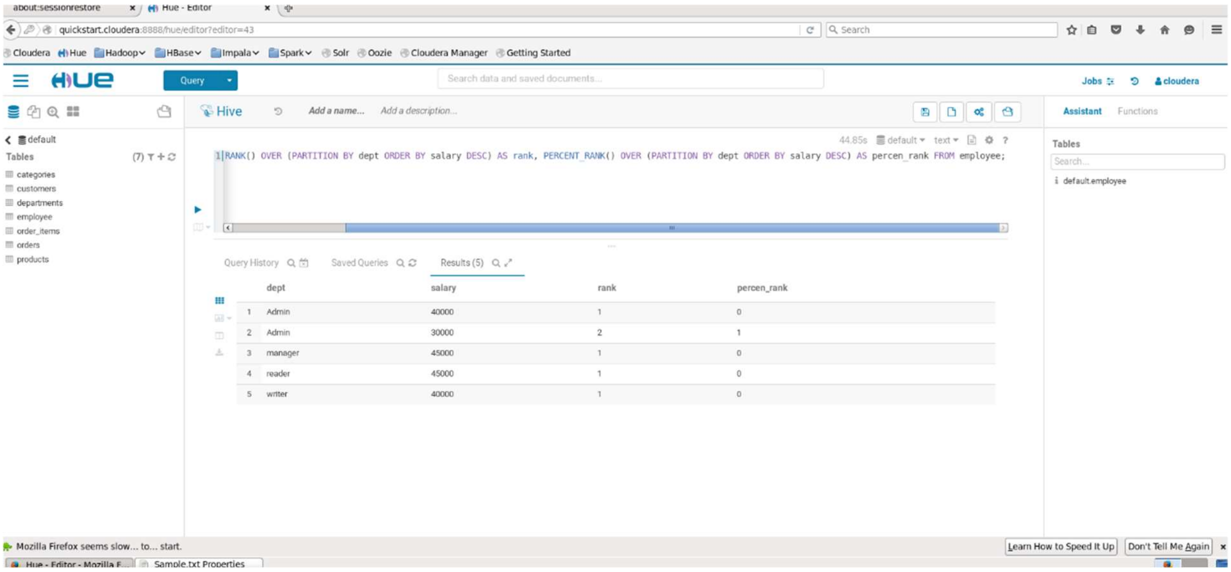
	dept	salary	cume_dist
1	Admin	30000	0.20000000000000001
2	Admin	40000	0.59999999999999998
3	writer	40000	0.59999999999999998
4	reader	45000	1
5	manager	45000	1

(Take Screenshot #15)

PERCENT_RANK():

It is similar to CUME_DIST, but it uses rank values rather than row counts in its numerator as a total number of rows - 1 divided by current rank - 1. Therefore, it returns the percent rank of a value relative to a group of values.

```
hive>SELECT dept, salary, RANK() OVER (PARTITION BY dept ORDER BY
salary DESC) AS rank, PERCENT_RANK() OVER (PARTITION BY dept ORDER
BY salary DESC) AS percen_rank FROM employee;
```



```
RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS rank, PERCENT_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS percen_rank FROM employee;
```

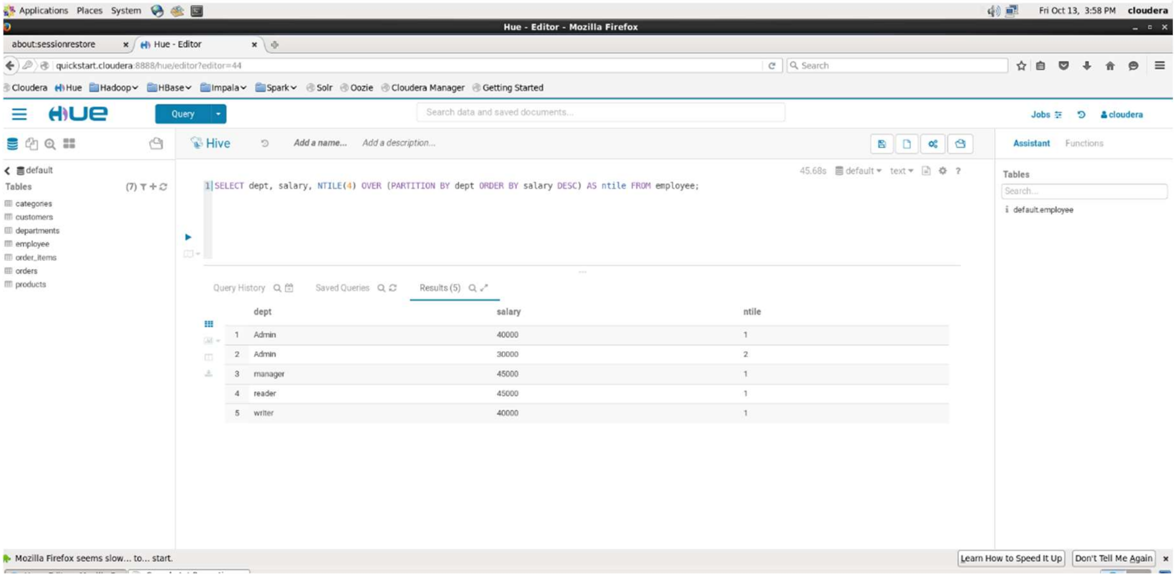
	dept	salary	rank	percen_rank
1	Admin	40000	1	0
2	Admin	30000	2	1
3	manager	45000	1	0
4	reader	45000	1	0
5	writer	40000	1	0

(Take Screenshot #16)

NTILE ():

It divides an ordered dataset into a number of buckets and assigns an appropriate bucket number to each row. It can be used to divide rows into equal sets and assign a number to each row.

```
hive > SELECT dept, salary, NTILE(4) OVER (PARTITION BY dept ORDER
BY salary DESC) AS ntile FROM employee;
```



```
SELECT dept, salary, NTILE(4) OVER (PARTITION BY dept ORDER BY salary DESC) AS ntile FROM employee;
```

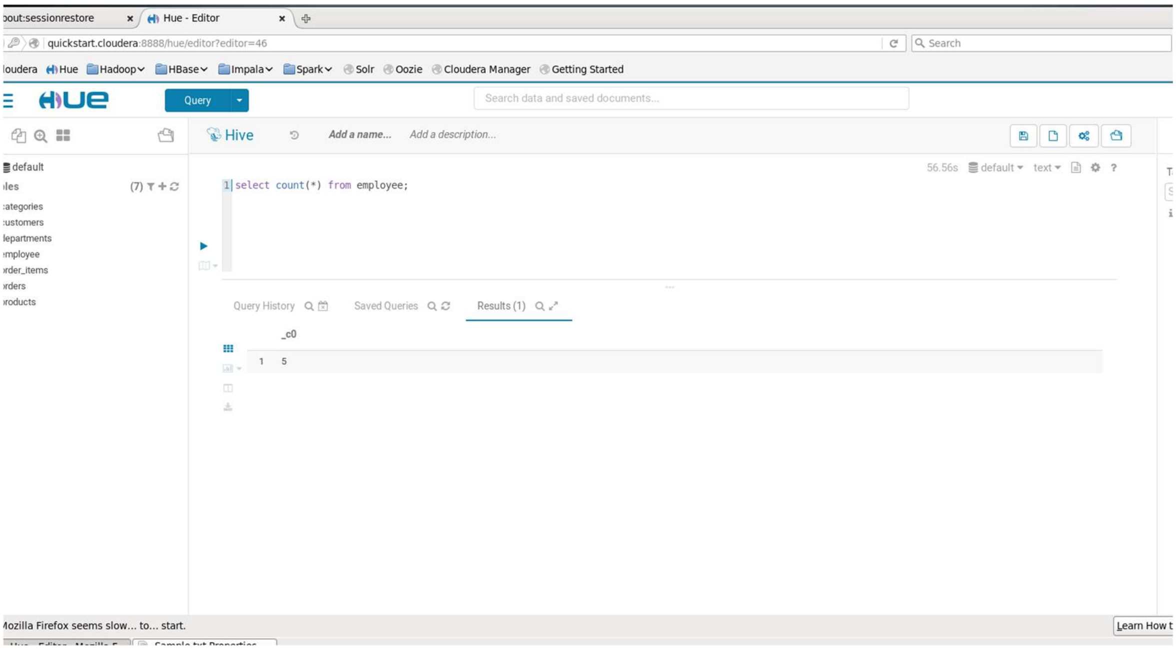
	dept	salary	ntile
1	Admin	40000	1
2	Admin	30000	2
3	manager	45000	1
4	reader	45000	1
5	writer	40000	1

(Take Screenshot #17)

COUNT ():

Returns the count of all rows in a table including rows containing NULL values
When you specify a column as an input, it ignores NULL values in the column for the count.
Also ignores duplicates by using DISTINCT.
Return: BIGINT
`count(*)` – Returns the count of all rows in a table including rows containing NULL values.
`count(expr)` – Returns the total number of rows for expression excluding null.

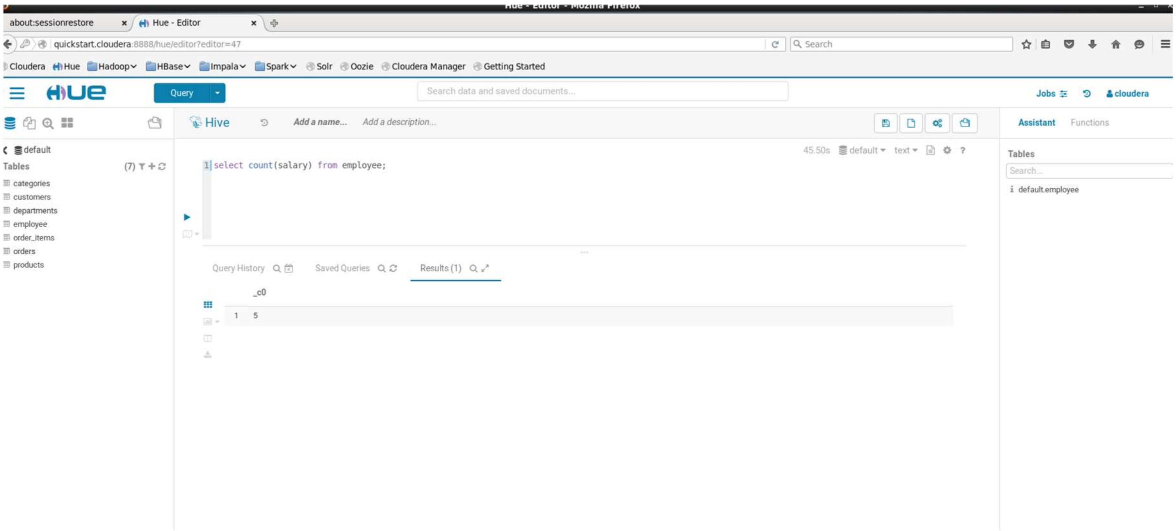
```
Hive> select count(*) from employee;
```



```
select count(*) from employee;
```

```
   _c0
1  5
```

```
Hive> select count(salary) from employee;
```



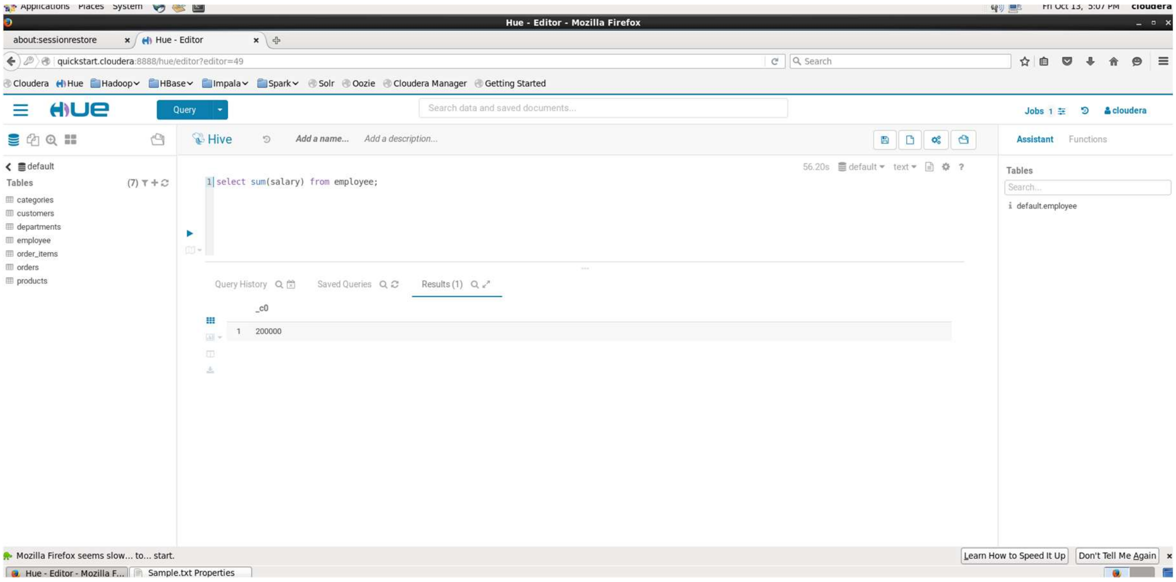
```
select count(salary) from employee;
```

```
   _c0
1  5
```

SUM():

Returns the sum of all values in a column.
When used with a group it returns the sum for each group.
Also ignores duplicates by using DISTINCT.

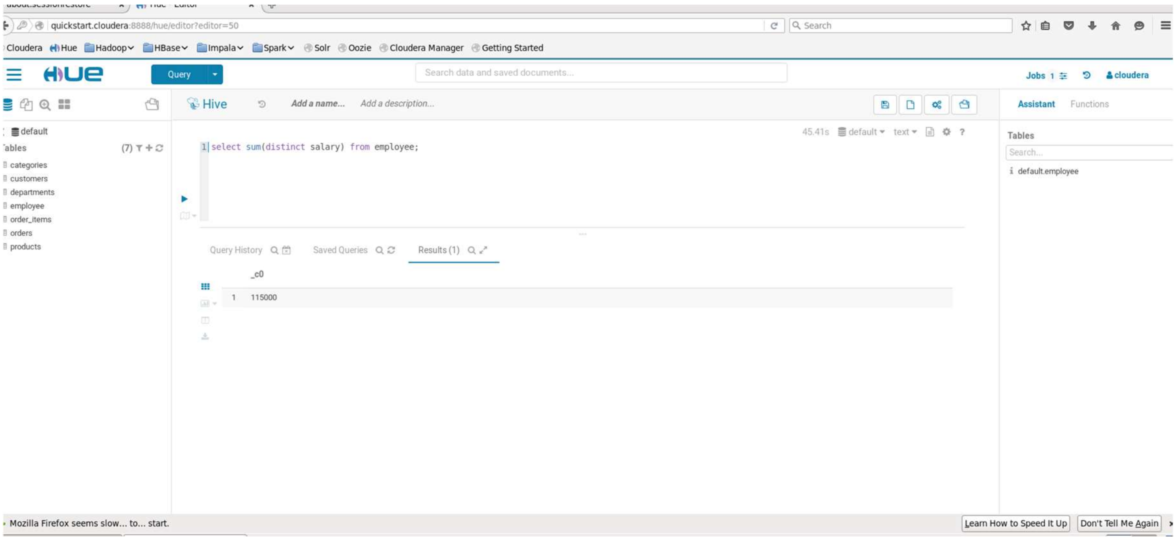
```
Hive> select sum(salary) from employee;
```



```
select sum(salary) from employee;
```

```
   _c0
1  200000
```

```
Hive> select sum(distinct salary) from employee;
```



```
select sum(distinct salary) from employee;
```

```
   _c0
1  115000
```

Step 5: Additional Exercise

For this step, it is recommended that a few more HIVE queries be performed utilising Analytical and mathematical functions.

Examples (you should replace `X, X1, X2` with the desired attributes from the tables):

- `select count(distinct X1), count(distinct X2) from table_X;`
- `select * from table_X where X1 like '%X%';`

- `select * from table_X where LOWER(X1) like '%X%';`
- `select * from table_X where LOWER(X1) like '%X%' or LOWER(X1) NOT LIKE '%X%';`
- `select * from table_X where X1 rlike '.*(X|X).*';`

Exercise 4: Optimising Hive Queries

Step 1: Indexing

- Create an index on the salary column of the employee table to speed up queries filtering by salary.

```
CREATE INDEX idx_salary ON TABLE employee (salary) AS 'COMPACT'
WITH DEFERRED REBUILD;
```

- Rebuild the Index to apply changes.

```
ALTER INDEX idx_salary ON employee REBUILD;
```

- Use the Index in a Query: Run a query that utilizes the index and compare the execution time with a similar query without indexing.

```
SELECT * FROM employee WHERE salary > 50000;
```

Step 2: Partitioning

- Partition the `employee` Table by Department: Create a partitioned table based on the `dept` column.

```
CREATE TABLE employee_partitioned (
    eid INT,
    name STRING,
    salary STRING,
    designation STRING
)
PARTITIONED BY (dept STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

- Before loading the data into the partitioned table, it is important to allow fully dynamic partitioning. By default, Hive operates in **strict mode** for dynamic partitioning, which requires at least one partition column to be statically defined. In this case, we're trying to dynamically partition the entire table without specifying any static partition columns. Switch to non-strict mode to allow fully dynamic partitioning by running the following command:

```
SET hive.exec.dynamic.partition.mode=nonstrict;
```

- Load the data into the partitioned table.

```
INSERT OVERWRITE TABLE employee_partitioned PARTITION (dept)

SELECT eid, name, salary, designation, dept FROM employee;
```

- Query Partitioned Table: Query data from specific partitions to see performance improvements.

```
SELECT * FROM employee_partitioned WHERE dept = 'Admin';
```

(Take Screenshot #18)

Step 3: Bucketing

- Bucket the Partitioned Table: Apply bucketing to the partitioned table based on the eid column

```
CREATE TABLE employee_bucketed (

    eid INT,

    name STRING,

    salary STRING,

    designation STRING

)

PARTITIONED BY (dept STRING)

CLUSTERED BY (eid) INTO 4 BUCKETS

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t'

LINES TERMINATED BY '\n'

STORED AS TEXTFILE;
```

- Insert Data into the Bucketed Table: Populate the bucketed table

```
INSERT OVERWRITE TABLE employee_bucketed PARTITION (dept)

SELECT eid, name, salary, designation, dept FROM
employee_partitioned;
```

- Run Queries on the Bucketed Table: Observe the query execution time improvement when filtering and joining.

```
SELECT * FROM employee_bucketed WHERE eid < 1205;
```

(Take Screenshot #19)

Step 4: Combining Partitioning and Bucketing

- Partition by `dept` and Bucket by `salary`

```
CREATE TABLE employee_optimized (

    eid INT,

    name STRING,

    salary STRING,

    designation STRING

)

PARTITIONED BY (dept STRING)

CLUSTERED BY (salary) INTO 5 BUCKETS

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t'

LINES TERMINATED BY '\n'

STORED AS TEXTFILE;
```

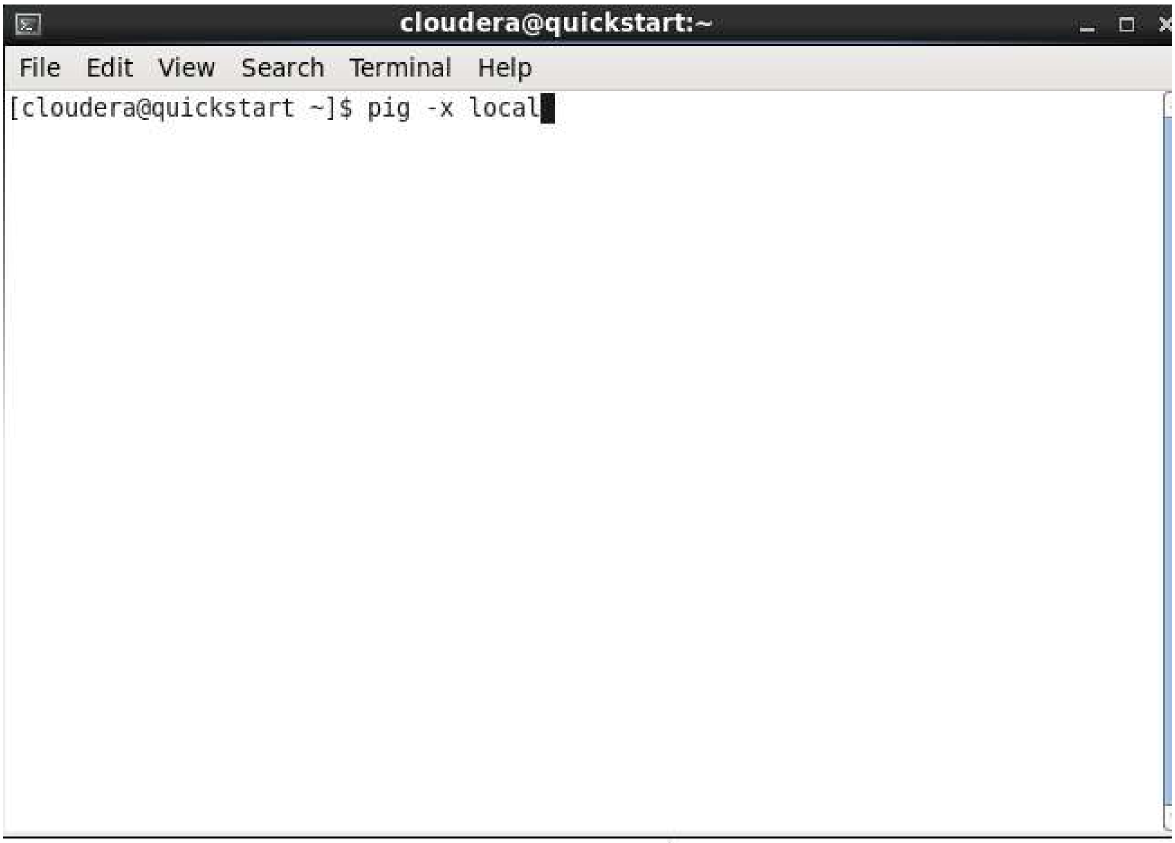
- Compare Query Performance:
  Run similar queries on non-partitioned, partitioned, and bucketed tables to analyse the differences.

  (Write one paragraph description)

Extra Exercise: Using Pig + Grunt Shell

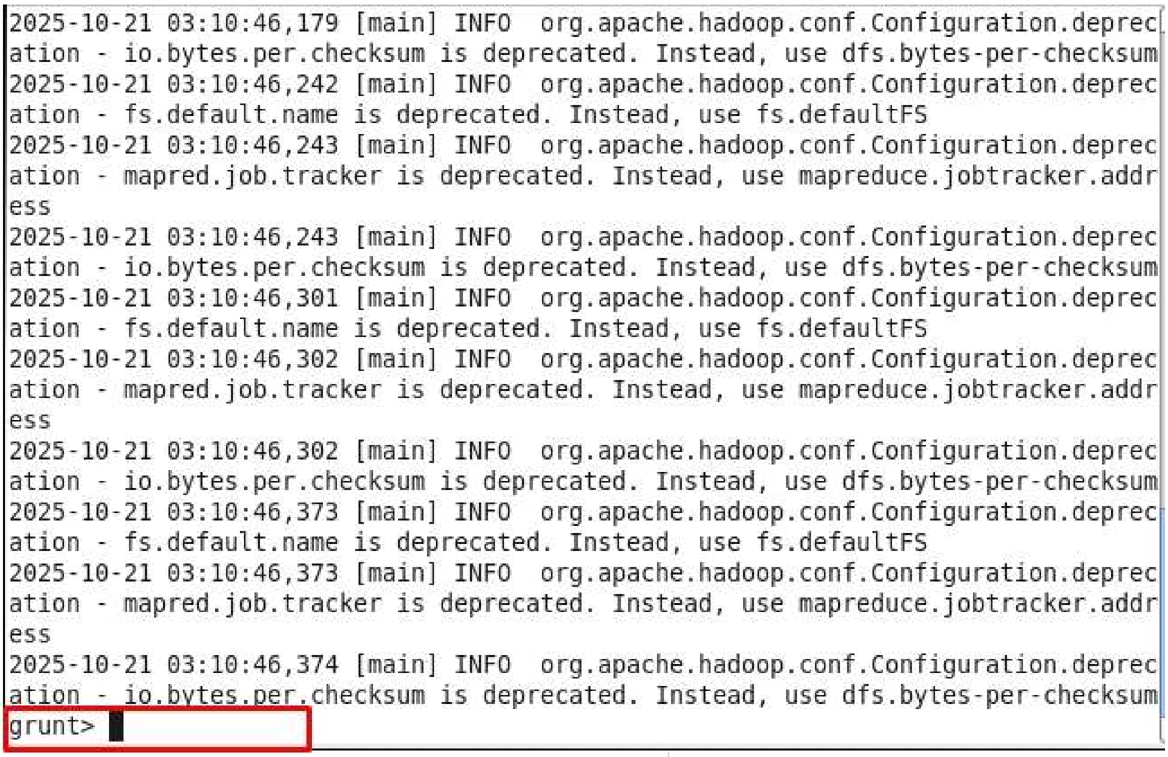
1.1 Launch a new terminal and invoke the following command:

$ pig -x local



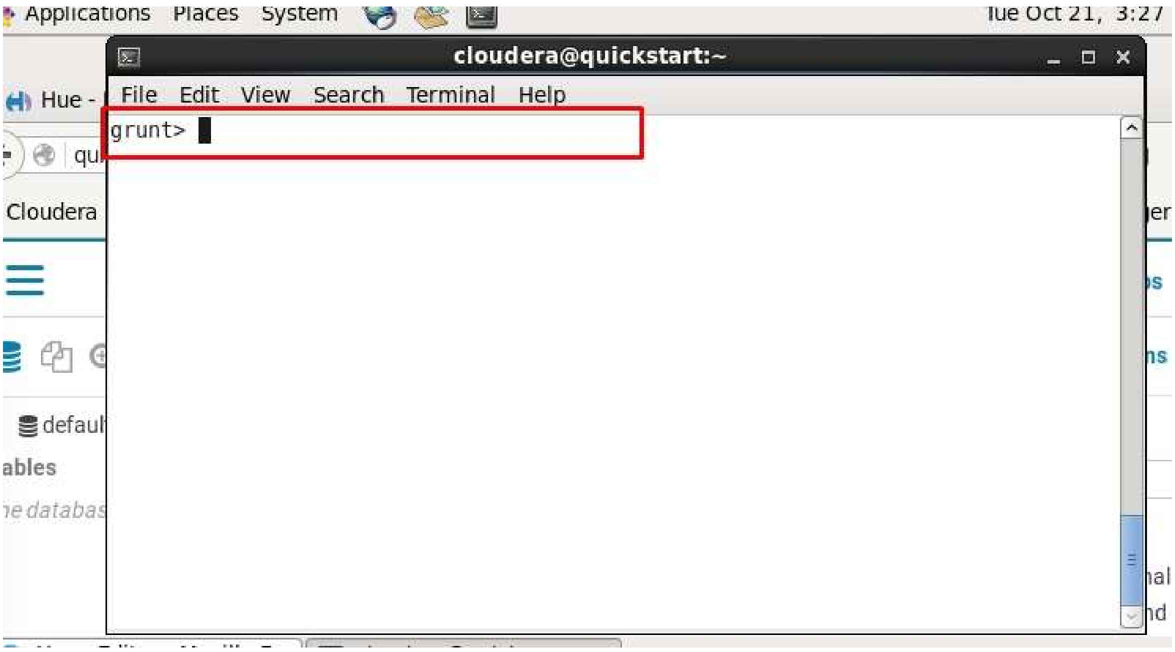
```
[cloudera@quickstart ~]$ pig -x local
```

To verify you're in a grunt shell, you should see the following output; as it's in a new "grunt>" shell.



```
2025-10-21 03:10:46,179 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
2025-10-21 03:10:46,242 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
2025-10-21 03:10:46,243 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.addr
ess
2025-10-21 03:10:46,243 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
2025-10-21 03:10:46,301 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
2025-10-21 03:10:46,302 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.addr
ess
2025-10-21 03:10:46,302 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
2025-10-21 03:10:46,373 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
2025-10-21 03:10:46,373 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.addr
ess
2025-10-21 03:10:46,374 [main] INFO  org.apache.hadoop.conf.Configuration.deprec
ation - io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
grunt> 
```

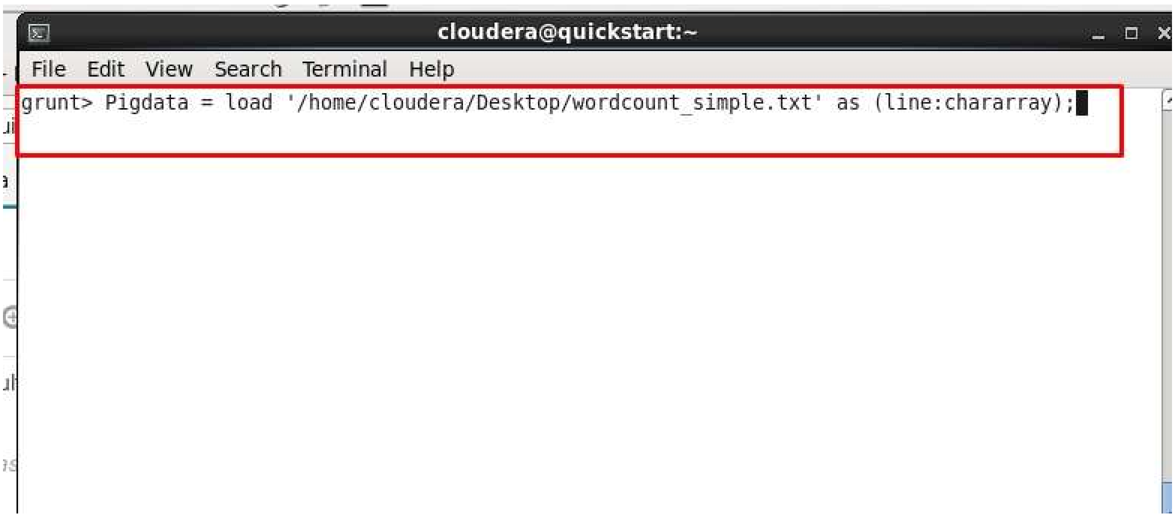
Clean your terminal output by issuing (CTRL + L)



```
grunt> 
```

Download the “wordcount_simple.txt” file that has been shared and drag and drop it into your desktop environment to load it into Pig. After verifying that the file is now in your desktop environment, issue the following command.

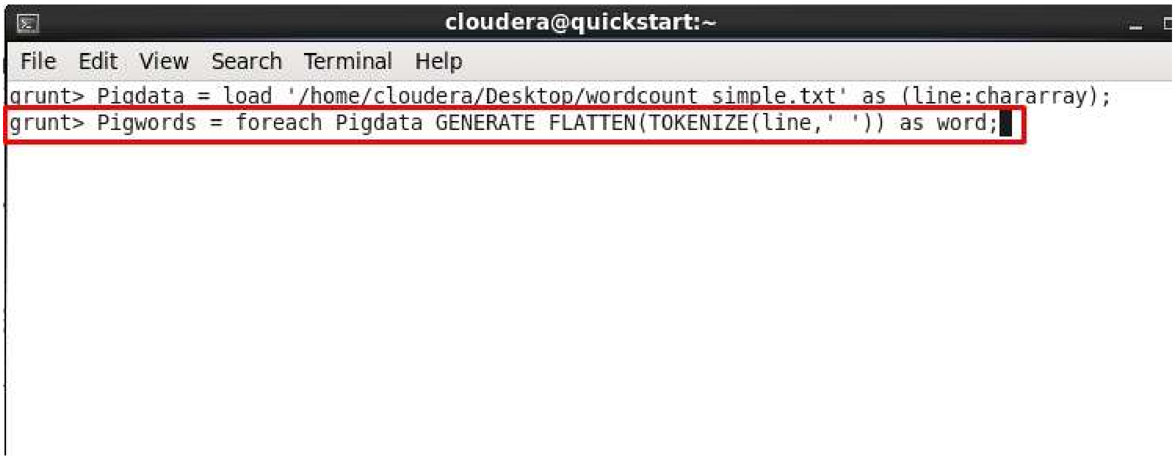
grunt> Pigdata = load '/home/cloudera/Desktop/wordcount_simple.txt' as (line:chararray);



```
grunt> Pigdata = load '/home/cloudera/Desktop/wordcount_simple.txt' as (line:chararray);
```

Now we need to identify the variables that we will be working with, the file contains a few simple sentences that we want to breakdown into individual words and count their frequency of occurrence within that file. To do that, we issue the following command.

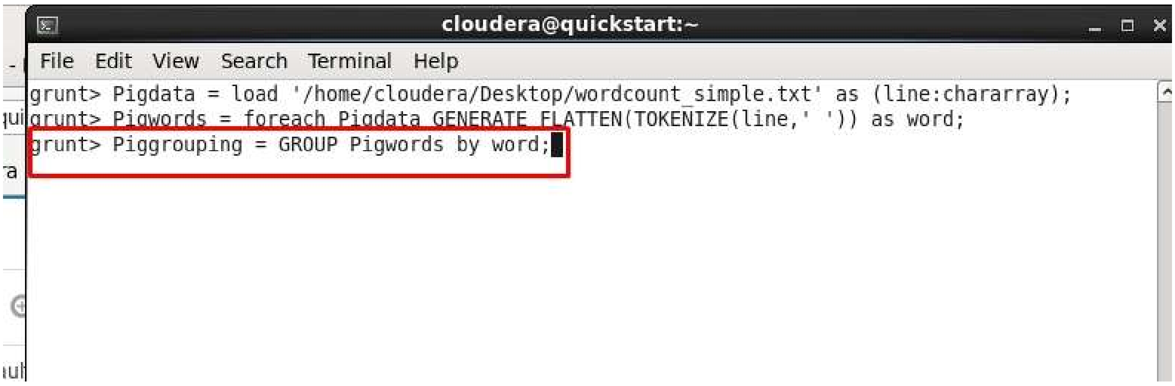
grunt> Pigwords = foreach Pigdata GENERATE FLATTEN(TOKENIZE(line,' ')) as word;



```
grunt> Pigdata = load '/home/cloudera/Desktop/wordcount_simple.txt' as (line:chararray);
grunt> Pigwords = foreach Pigdata GENERATE FLATTEN(TOKENIZE(line,' ')) as word;
```

Now after identifying the input, we need to identify how should Pig process those words, and as mentioned we want to group the output, so for that we issue the following command.

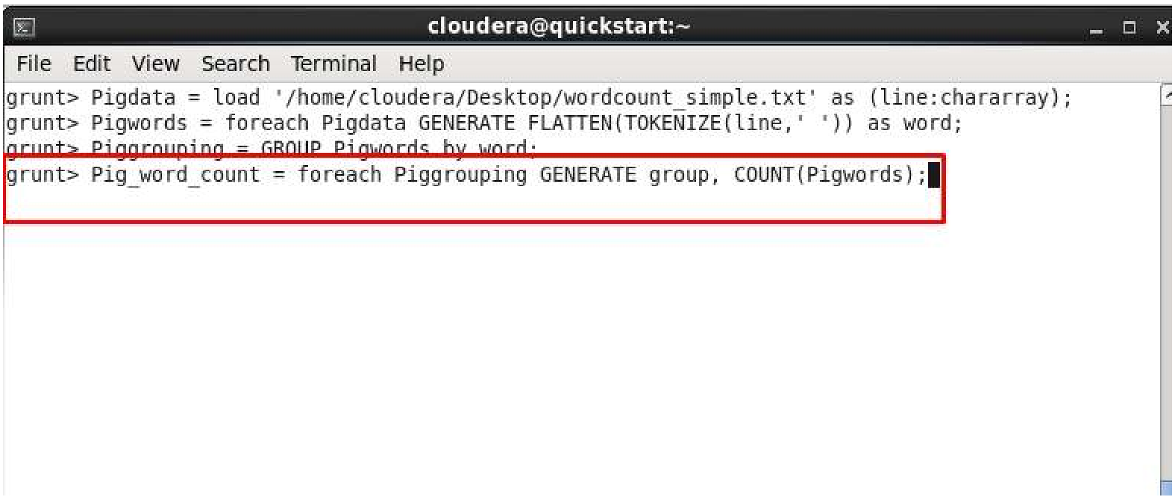
grunt> Piggrouping = GROUP Pigwords by word;



```
grunt> Pigdata = load '/home/cloudera/Desktop/wordcount_simple.txt' as (line:chararray);
grunt> Pigwords = foreach Pigdata GENERATE FLATTEN(TOKENIZE(line,' ')) as word;
grunt> Piggrouping = GROUP Pigwords by word;
```

After executing that command, we want to identify the recall method by issuing the following command and how it should deal with the current set environment variables in play.

grunt> Pig_word_count = foreach Piggrouping GENERATE group, COUNT(Pigwords);

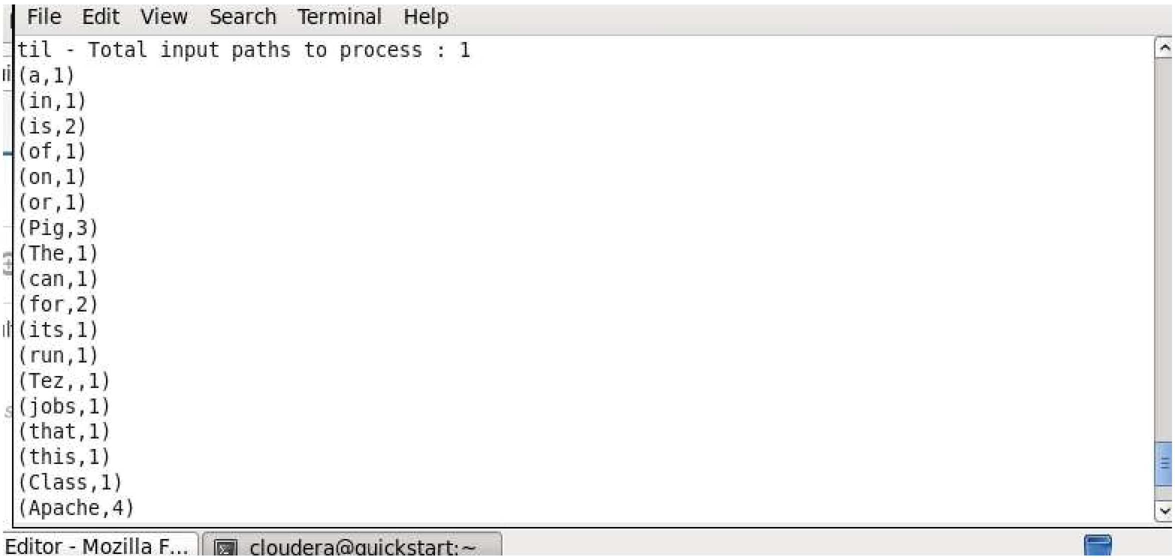


```
grunt> Pigdata = load '/home/cloudera/Desktop/wordcount_simple.txt' as (line:chararray);
grunt> Pigwords = foreach Pigdata GENERATE FLATTEN(TOKENIZE(line,' ')) as word;
grunt> Piggrouping = GROUP Pigwords by word;
grunt> Pig_word_count = foreach Piggrouping GENERATE group, COUNT(Pigwords);
```

Now to print out our results to the terminal, we need to understand that pig uses a different method of recall which is “dump”, and simply just dump the environment variable containing our results. So, issue the following command:

grunt> dump Pig_word_count;

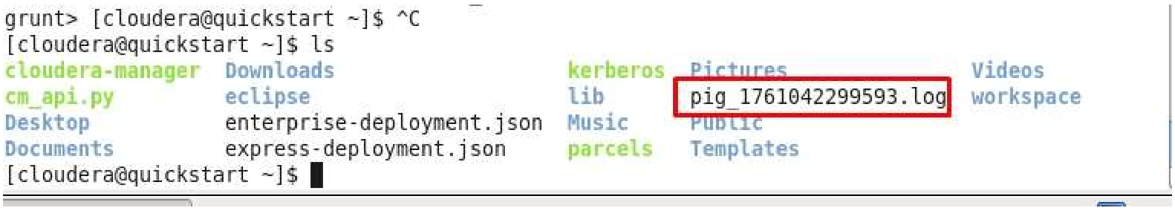
You should see the following output, with counted words and their occurrence frequency.



```
til - Total input paths to process : 1
(a,1)
(in,1)
(is,2)
(of,1)
(on,1)
(or,1)
(Pig,3)
(The,1)
(can,1)
(for,2)
(its,1)
(run,1)
(Tez,,1)
(jobs,1)
(that,1)
(this,1)
(Class,1)
(Apache,4)
```

If you wish to exit Pig, a log file will be generated and added to your home directory.

To quit the Pig terminal, (CTRL + C) will terminate the current Pig terminal and inform you that a log file has been generated. We issue "ls" to view the log file in our home directory.



```
grunt> [cloudera@quickstart ~]$ ^C
[cloudera@quickstart ~]$ ls
cloudera-manager  Downloads                   kerberos  Pictures                 Videos
cm_api.py         eclipse                     lib       pig_1761042299593.log    workspace
Desktop           enterprise-deployment.json  Music     Public
Documents         express-deployment.json     parcels   Templates
[cloudera@quickstart ~]$ 
```