

CN7031 - Big Data Analytics

Student ID: 3059676

Name: Mann Shrestha

Lab-4

Exercise 1: Getting Ready for Hue Cloudera

Exercise 2: Create Database/Table

Step 1: Create a database and table

- “userdb6” Database were created.
- Empty table named “employee” were created with setting up the delimiter field terminated by “\t” and line terminated by “\n”.

Step 2: Load Data

- Loading data named Simple.txt from Cloudera desktop.

The screenshot shows the Hue interface with a query editor and a results table. The query is:

```
5 | SELECT * FROM employee limit 10;
```

The results table has columns: employee.eid, employee.name, employee.salary, and employee.designation. The data is:

	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

- Adding “dept” column in an employee table using ALTER statement.

Exercise 3: Working with Hive queries

Step 1: Retrieving information

a. WHERE query

The screenshot shows the Hue interface with a query editor and a results table. The query is:

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

The results table has columns: employee.eid, employee.name, employee.salary, employee.designation, and employee.dept. The data is:

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

b. Order By query

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

Query History Saved Queries Results (5)

	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

c. Group By query

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

Query History Saved Queries Results (3)

salary	counter
1 30000	1
2 40000	2
3 45000	2

Step 2: Create a view

- Creating a view – virtual table

```
-- retrieving 5 records only
SELECT * FROM emp_40000 LIMIT 5;
```

Query History Saved Queries Results (2)

	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

Step 3: Join queries

- Joining two tables' Customers and Orders table.
- a. Creating Customers table
- b. Load the Customers.txt data from Cloudera home
- c. List 10 records from Customers table.

```
29|-- list 10 records
30|SELECT * FROM Customers LIMIT 10;
31|
```

Query History Saved Queries Results (7)

	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

- d. Create a table for Orders
- e. Load the Orders.txt data from Cloudera home
- f. List the data

```

38 -- list the records
39 SELECT * FROM Orders;

```

Query History Saved Queries Results (4)

	orders.old	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

- g. Joining a tables Customers and Orders

```

41 -- Join Query
42 SELECT c.Id, c.Name, c.Age, o.Amount
43 FROM customers c
44 JOIN Orders o
45 ON (c.ID = o.CUSTOMER_ID);
46

```

Query History Saved Queries Results (4)

	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

- h. Making a LEFT OUTER JOIN between CUSTOMER and ORDER tables

```

47 -- Left outer Join
48 SELECT c.ID, c.NAME, o.AMOUNT, o.DATE
49 FROM customers c
50 LEFT OUTER JOIN Orders o
51 ON (c.ID = o.CUSTOMER_ID);

```

Query History Saved Queries Results (8)

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

i. Making a RIGHT OUTER JOIN between CUSTOMER and ORDER tables

```

53 -- Right outer Join
54 SELECT c.ID, c.NAME, o.AMOUNT, o.DATE
55 FROM customers c
56 RIGHT OUTER JOIN Orders o
57 ON (c.ID = o.CUSTOMER_ID);

```

Query History Saved Queries Results (4)

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

j. Making a FULL OUTER JOIN between CUSTOMER and ORDER tables

```

59 -- FULL outer Join
60 SELECT c.ID, c.NAME, o.AMOUNT, o.DATE
61 FROM customers c
62 FULL OUTER JOIN Orders o
63 ON (c.ID = o.CUSTOMER_ID);
64

```

Query History Saved Queries Results (8)

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	1500	2009-10-08 00:00:00
4 3	kaushik	3000	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

Step 4: Analytic Functions

- RANK()

```

65 -- Analytical Function
66 SELECT * FROM employee DISTRIBUTE BY RAND() SORT BY RAND() LIMIT 10;

```

Query History Saved Queries Results (5)

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

- **DENSE_RANK ()**

```
69| -- DENSE_RANK()
70| SELECT dept, salary, DENSE_RANK() OVER(partition by dept ORDER BY salary desc) as dens_rank from employee;
71|
72|
```

The screenshot shows a database interface with a query editor at the top containing the SQL code for DENSE_RANK(). Below the editor is a results grid titled "Results (5)". The grid has three columns: "dept", "salary", and "dens_rank". The data shows that employees in the same department have the same rank, which is 1 for Admins and Manager, and 2 for Readers and Writers.

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

- **ROW_NUMBER ()**

```
/1
72| -- ROW NUMBER
73| SELECT dept, salary, ROW_NUMBER() OVER(partition by dept ORDER BY salary desc) as row_no from employee;
74|
```

The screenshot shows a database interface with a query editor at the top containing the SQL code for ROW_NUMBER(). Below the editor is a results grid titled "Results (5)". The grid has three columns: "dept", "salary", and "row_no". The data shows that each employee in a department gets a unique rank starting from 1, even if they have the same salary.

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

- **CUME_DIST()**

```
75| -- CUME_DIST
76| SELECT dept, salary, CUME_DIST() OVER(ORDER BY salary) as cume_dist from employee;
```

The screenshot shows a database interface with a query editor at the top containing the SQL code for CUME_DIST(). Below the editor is a results grid titled "Results (5)". The grid has three columns: "dept", "salary", and "cume_dist". The "cume_dist" column shows the cumulative distribution of salaries, where each employee's value is the sum of their own salary and all previous salaries in the department, divided by the total number of employees in the department.

dept	salary	cume_dist
1 Admin	30000	0.2000000000000001
2 Admin	40000	0.5999999999999998
3 writer	40000	0.5999999999999998
4 reader	45000	1
5 manager	45000	1

- PERCENT_RANK ()

```
79 |SELECT dept, salary, RANK() OVER(partition by dept ORDER BY salary desc) as rank, PERCENT_RANK() OVER(partition by dept ORDER BY
80 |
```

Query History Saved Queries Results (5)

dept	salary	rank	percent_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

- NTILE ()

```
81 |-- NTILE
82 |SELECT dept, salary, NTILE(4) OVER(partition by dept ORDER BY salary desc) as ntile from employee;
83 |
```

Query History Saved Queries Results (5)

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

Exercise 4: Optimising Hive Queries

Step 1: Indexing

Step 2: Partitioning

- SET hive.exec.dynamic.partition=true;
- SET hive.exec.dynamic.partition.mode=nonstrict;

```
5 |-- Query Parttitioned Table:
6 |SELECT * FROM employee_partitioned WHERE dept = 'Admin';
7 |
```

INFO : OK

Query History Saved Queries Results (2)

employee_partitioned.eid	employee_partitioned.name	employee_partitioned.salary	employee_partitioned.designation
1 1204	Kiran	40000.0	Hr
2 1205	Kranthi	30000.0	Op

Step 3: Bucketing

The screenshot shows a database interface with a query editor and a results table. The query editor contains:

```
27 -- run wueries on the bucketed table
28 SELECT * FROM employee_bucketed WHERE eid < 1205;
```

The results table has the following data:

	employee_bucketed.eid	employee_bucketed.name	employee_bucketed.salary	employee_bucketed.designation
1	1204	Kiran	40000.0	Hr
2	1201	Gopal	45000.0	Technical
3	1202	Manisha	45000.0	Proof
4	1203	Masthanvali	40000.0	Technical

Step 4: Combining Partitioning and Bucketing

Q. Compare Query Performance:

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

- Based on my observation, non-partitioned table is the slowest because it scans the entire table data for every filter operation (e.g. WHERE statement) which slow down the performance. As for the partition, it divides table into smaller parts based on the categorical column value. In our case, we use filter condition (WHERE) with department, and it fetches the information based on departments. This approach is more efficient, reducing the amount of data scanned, and skips the partition which are not relevant and improve the query performance. On the other hand, bucketing sub-divides the data within each partition into fixed-size clusters. In our case, we use partition based on department and clustered by employee id. I think it is more based on size or numeric function or more logical rather than categorical column and because of that query performance is more efficient. Overall, partition and bucketing are optimized way of improving query performance. However, we must choose each of them based on the requirements.