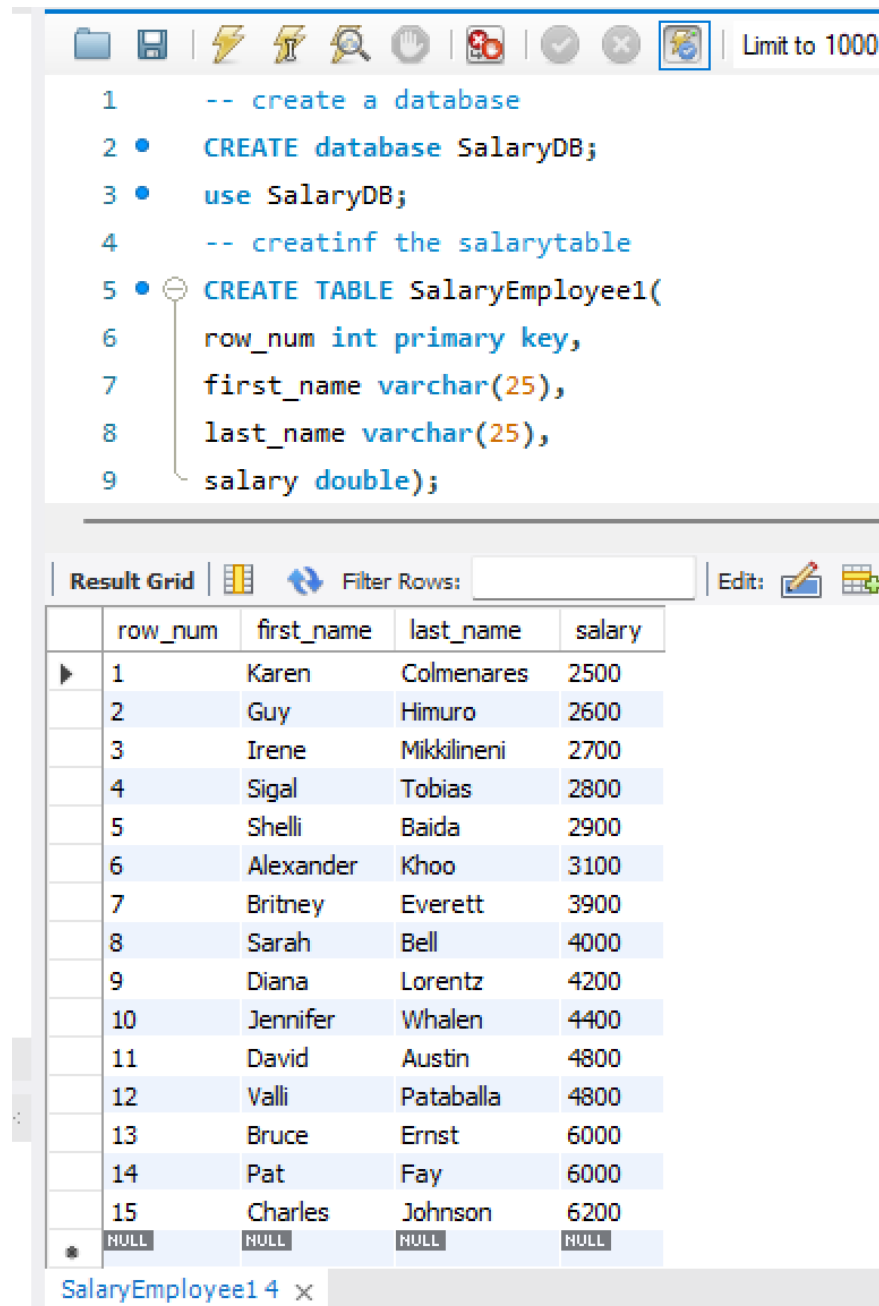


Examination Guide:

SQL Advanced window features: Queries, Output, and Formulas

Using the dataset provided below, write the SQL queries, and expected output in table format for each of the window features supported by MySQL.



The screenshot shows a MySQL IDE interface. The top toolbar includes icons for file operations, execution, and search. The SQL editor contains the following code:

```
1  -- create a database
2  • CREATE database SalaryDB;
3  • use SalaryDB;
4  -- creatinf the salarytable
5  • CREATE TABLE SalaryEmployee1(
6      row_num int primary key,
7      first_name varchar(25),
8      last_name varchar(25),
9      salary double);
```

Below the editor is the 'Result Grid' tab, which displays the output of the query. It includes a 'Filter Rows' input field and an 'Edit' button. The result is a table with 16 rows, including a final row of NULL values.

	row_num	first_name	last_name	salary
▶	1	Karen	Colmenares	2500
	2	Guy	Himuro	2600
	3	Irene	Mikkilineni	2700
	4	Sigal	Tobias	2800
	5	Shelli	Baida	2900
	6	Alexander	Khoo	3100
	7	Britney	Everett	3900
	8	Sarah	Bell	4000
	9	Diana	Lorentz	4200
	10	Jennifer	Whalen	4400
	11	David	Austin	4800
	12	Valli	Pataballa	4800
	13	Bruce	Ernst	6000
	14	Pat	Fay	6000
	15	Charles	Johnson	6200
*	NULL	NULL	NULL	NULL

At the bottom, a tab labeled 'SalaryEmployee1 4' is visible.

1. Compute for the FIRST_VALUE() given the above data in the table.

FIRST_VALUE() – this window feature returns the first value in an ordered set of values. See screenshot for the SQL Command and the table output.

```
40
41      -- usage of FIRST_VALUE()
42 •    SELECT row_num, first_name, last_name, salary,
43      FIRST_VALUE(row_num) OVER(ORDER BY salary) as first_value_answer
44      FROM SalaryEmployee1;
45
```

Result Grid					
		Filter Rows:	Export:		Wrap Cell Content:
	row_num	first_name	last_name	salary	first_value_answer
▶	1	Karen	Colmenares	2500	1
	2	Guy	Himuro	2600	1
	3	Irene	Mikkilineni	2700	1
	4	Sigal	Tobias	2800	1
	5	Shelli	Baida	2900	1
	6	Alexander	Khoo	3100	1
	7	Britney	Everett	3900	1
	8	Sarah	Bell	4000	1
	9	Diana	Lorentz	4200	1
	10	Jennifer	Whalen	4400	1
	11	David	Austin	4800	1
	12	Valli	Pataballa	4800	1
	13	Bruce	Ernst	6000	1
	14	Pat	Fay	6000	1
	15	Charles	Johnson	6200	1

Result 5 ×

2. Compute for the LAST_VALUE() given the above data in the table.

LAST_VALUE() – this window feature returns the first value in an ordered set of values. See screenshot for the SQL Command and the table output.

Note: we use CURRENT ROW | UNBOUNDED PRECEDING | UNBOUNDED FOLLOWING to indicate where the window frame starts(preceding) and ends (following).

```
46 -- usage of LAST_VALUE()
47 • SELECT row_num, first_name, last_name, salary,
48    LAST_VALUE(row_num) OVER(ORDER BY salary
49    ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) as last_value_answer
50    FROM SalaryEmployee1;
51
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	last_value_answer
▶	1	Karen	Colmenares	2500	15
	2	Guy	Himuro	2600	15
	3	Irene	Mikkilineni	2700	15
	4	Sigal	Tobias	2800	15
	5	Shelli	Baida	2900	15
	6	Alexander	Khoo	3100	15
	7	Britney	Everett	3900	15
	8	Sarah	Bell	4000	15
	9	Diana	Lorentz	4200	15
	10	Jennifer	Whalen	4400	15
	11	David	Austin	4800	15
	12	Valli	Pataballa	4800	15
	13	Bruce	Ernst	6000	15
	14	Pat	Fay	6000	15
	15	Charles	Johnson	6200	15

Result 10 x

Output

3. Compute for the LEAD(2) for GUY.

LEAD(n) – this window feature returns the nth NEXT rows from the current row. In this case current row = 2 for guy. Therefore, LEAD(2) for Guy. Answer row num= 4.

See screenshot for the SQL Command and the table output.

```
52
53  -- usage of LEAD(2) FOR Guy
54  SELECT * FROM(
55      SELECT row_num, first_name, last_name, salary,
56      LEAD (row_num, 2) OVER (ORDER BY Salary) lead2_Guy
57      FROM SalaryEmployee1) as table1
58  WHERE first_name ="Guy";
59
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	lead2_Guy
▶	2	Guy	Himuro	2600	4

4. Compute for the LAG(4) for Pat.

LEAD(n) – this window feature returns the nth PREVIOUS row number from the current row. In this case the current row =14 for Pat. Therefore, LAG(4) for Guy. Answer row num= 10.

See screenshot for the SQL Command and the table output.

```
63  -- usage of LAG(4) FOR Pat
64  SELECT * FROM(
65      SELECT row_num, first_name, last_name, salary,
66      LAG (row_num, 4) OVER (ORDER BY Salary) lag4_Pat
67      FROM SalaryEmployee1) as table1
68  WHERE first_name ="Pat";
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	lag4_Pat
▶	14	Pat	Fay	6000	10

5. Compute for the RANK() for Valli.

RANK() – this window feature assigns a rank to each row within a partition of a result set. Note that RANK() – ranks values with gaps. (See DENSE_RANK() which ranks without gaps). Our dataset is ordered by Salary, and the rankings are based on salary. See screenshot for the SQL Command and the table output.

```
60  -- usage of RANK() FOR Valli
61  • SELECT * FROM(
62      SELECT row_num, first_name, last_name, salary,
63      RANK () OVER (ORDER BY Salary) as rank_Valli
64      FROM SalaryEmployee1) as table1
65  WHERE first_name ="Valli";
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	rank_Valli
▶	12	Valli	Pataballa	4800	11

6. Compute the RANK() for Bruce.

RANK() – this window feature assigns a rank to each row within a partition of a result set. Note that RANK() – ranks results with gaps. (See DENSE_RANK() which ranks without gaps). Our dataset is ordered by Salary, and the rankings are based on salary. See screenshot for the SQL Command and the table output.

```
67  -- usage of RANK() FOR Bruce
68  • SELECT * FROM(
69      SELECT row_num, first_name, last_name, salary,
70      RANK () OVER (ORDER BY Salary) as rank_Valli
71      FROM SalaryEmployee1) as table1
72  WHERE first_name ="Bruce";
73
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	rank_Valli
▶	13	Bruce	Ernst	6000	13

7. Compute the DENSE_RANK() for Valli.

DENSE_RANK() – this window feature assigns a rank to each row within a partition of a result set, with no gaps in ranking values. Our dataset is ordered by Salary, and the rankings are based on salary.

See screenshot for the SQL Command and the table output.

```
84  -- usage of DENSE_RANK() FOR Valli
85  SELECT * FROM(
86      SELECT row_num, first_name, last_name, salary,
87      DENSE_RANK() OVER (ORDER BY Salary) as dense_rank_Valli
88      FROM SalaryEmployee1) as table1
89  WHERE first_name ="Valli";
90
```

row_num	first_name	last_name	salary	dense_rank_Valli
12	Valli	Pataballa	4800	11

8. Compute the DENSE_RANK() for Bruce.

DENSE_RANK() – this window feature assigns a rank to each row within a partition of a result set, with no gaps in ranking values. Our dataset is ordered by Salary, and the rankings are based on salary.

See screenshot for the SQL Command and the table output.

```
91  -- usage of DENSE_RANK() FOR Bruce
92  SELECT * FROM(
93      SELECT row_num, first_name, last_name, salary,
94      DENSE_RANK() OVER (ORDER BY Salary) as dense_rank_Bruce
95      FROM SalaryEmployee1) as table1
96  WHERE first_name ="Bruce";
97
```

row_num	first_name	last_name	salary	dense_rank_Bruce
13	Bruce	Ernst	6000	12

9. Write a query computing for RANK() and DENSE_RANK() of the same dataset in the question. Notice the difference in the values -- RANK() Vs. DENSE-RANK()

```
155  -- usage of RANK() values Vs. DENSE_RANK() values of the dataset
156  SELECT row_num, first_name, last_name, salary,
157         RANK() OVER (ORDER BY Salary) as 'Rank',
158         DENSE_RANK() OVER (ORDER BY Salary) as 'Dense Rank'
159  FROM SalaryEmployee1;
160
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	Rank	Dense Rank
▶	1	Karen	Colmenares	2500	1	1
	2	Guy	Himuro	2600	2	2
	3	Irene	Mikkilineni	2700	3	3
	4	Sigal	Tobias	2800	4	4
	5	Shelli	Baida	2900	5	5
	6	Alexander	Khoo	3100	6	6
	7	Britney	Everett	3900	7	7
	8	Sarah	Bell	4000	8	8
	9	Diana	Lorentz	4200	9	9
	10	Jennifer	Whalen	4400	10	10
	11	David	Austin	4800	11	11
	12	Valli	Pataballa	4800	11	11
	13	Bruce	Ernst	6000	13	12
	14	Pat	Fay	6000	13	12
	15	Charles	Johnson	6200	15	13

Result 50 x

10. Compute the ROW_NUMBER() for Valli.

ROW_NUMBER() – this window feature assigns a sequential integer to each row within the partition of a result set. The row number starts with 1 for the first row in each partition. See screenshot for the SQL Command and the table output.

```
97
98  -- usage of ROW_NUMBER() FOR Valli
99  ● ○ SELECT * FROM(
100      SELECT row_num, first_name, last_name, salary,
101             ROW_NUMBER() OVER (ORDER BY Salary) as row_number_Valli
102      FROM SalaryEmployee1) as table1
103  WHERE first_name ="Valli";
104
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	row_number_Valli
▶	12	Valli	Pataballa	4800	12

11. Compute the ROW_NUMBER() for Bruce.

ROW_NUMBER() – this window feature assigns a sequential integer to each row within the partition of a result set. The row number starts with 1 for the first row in each partition. See screenshot for the SQL Command and the table output.

```
105
106      -- usage of ROW_NUMBER() FOR Bruce
107 • SELECT * FROM(
108     SELECT row_num, first_name, last_name, salary,
109     ROW_NUMBER() OVER (ORDER BY Salary) as row_number_Bruce
110     FROM SalaryEmployee1) as table1
111     WHERE first_name ="Bruce";
112
```

Result Grid			Filter Rows:	<input type="text"/>	Export:		Wrap Cell Content:	
	row_num	first_name	last_name	salary	row_number_Bruce			
▶	13	Bruce	Ernst	6000	13			

12. Compute the PERCENT_RANK() for Valli.

PERCENT_RANK() – this window feature calculates the percentile ranking of a row within a partition or result set.

PERCENT_RANK() calculates the rank of that row minus one, divided by 1 less than number of rows in the evaluated partition or query result set: **Formula : ((rank - 1) / (total_rows - 1)) * 100**

See screenshot for the SQL Command and the table output.

```
113 -- usage of PERCENT_RANK() FOR Valli
114 • SELECT row_num, first_name, last_name, salary, ROUND(percent_rank_Valli, 2)*100 as 'Rank(%)' FROM(
115     SELECT row_num, first_name, last_name, salary,
116     PERCENT_RANK() OVER (ORDER BY Salary) as percent_rank_Valli
117     FROM SalaryEmployee1) as table1
118 WHERE first_name ="Valli";
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	Rank(%)
▶	12	Valli	Pataballa	4800	71

13. Compute the PERCENT_RANK() Per row/record.

PERCENT_RANK() – this window feature calculates the percentile ranking of a row within a partition or result set.

PERCENT_RANK() calculates the rank of that row minus one, divided by 1 less than number of rows in the evaluated partition or query result set: **Formula : ((rank - 1) / (total_rows - 1)) * 100**

See screenshot for the SQL Command and the table output.

```
125
126 -- usage of PERCENT_RANK() per row
127 • SELECT row_num, first_name, last_name, salary,
128     RANK() OVER (ORDER BY Salary) as 'Rank',
129     ROUND(ROUND(PERCENT_RANK() OVER (ORDER BY Salary),2) *100,2) as 'PercentRank(%)'
130     FROM SalaryEmployee1;
131
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	Rank	PercentRank(%)
▶	1	Karen	Colmenares	2500	1	0
	2	Guy	Himuro	2600	2	7
	3	Irene	Mikkilineni	2700	3	14
	4	Sigal	Tobias	2800	4	21
	5	Shelli	Baida	2900	5	29
	6	Alexander	Khoo	3100	6	36
	7	Britney	Everett	3900	7	43
	8	Sarah	Bell	4000	8	50
	9	Diana	Lorentz	4200	9	57
	10	Jennifer	Whalen	4400	10	64
	11	David	Austin	4800	11	71
	12	Valli	Pataballa	4800	11	71
	13	Bruce	Ernst	6000	13	86
	14	Pat	Fay	6000	13	86
	15	Charles	Johnson	6200	15	100

Result 42 x

14. Compute the NTILE(n) of the given dataset.

NTILE (n) – this window feature distributes rows of an ordered partition into a specified number of approximately equal groups, or buckets. It assigns each group a bucket number starting from one. For example, NTILE(4) divides the datasets into 4 equal portions by assigning a number to the same bucket or group.

See screenshot for the SQL Command and the table output.

```
133      -- usage of NTILE(4) for our ordered dataset
134 •    SELECT row_num, first_name, last_name, salary,
135      NTILE(4) OVER (ORDER BY Salary) as 'Cluster'
136      FROM SalaryEmployee1;
```

Result Grid					
		Filter Rows:		Export:	Wrap Cell C
	row_num	first_name	last_name	salary	Cluster
▶	1	Karen	Colmenares	2500	1
	2	Guy	Himuro	2600	1
	3	Irene	Mikkilineni	2700	1
	4	Sigal	Tobias	2800	1
	5	Shelli	Baida	2900	2
	6	Alexander	Khoo	3100	2
	7	Britney	Everett	3900	2
	8	Sarah	Bell	4000	2
	9	Diana	Lorentz	4200	3
	10	Jennifer	Whalen	4400	3
	11	David	Austin	4800	3
	12	Valli	Pataballa	4800	3
	13	Bruce	Ernst	6000	4
	14	Pat	Fay	6000	4
	15	Charles	Johnson	6200	4

Result 43 ✕

15. Compute the CUME_DIST() for every record of the given dataset.

CUME_DIST() – this window feature calculates the cumulative distribution of a value within a group of values. It represents the number of rows with values less than or equal to that row's value divided by the total number of rows (value ranges from 0 and 1)

Formula : ROW_NUMBER / total_rows

See screenshot for the SQL Command and the table output.

NOTE: Salary with the same rank MUST have the same CUME_DIST value

```
140 -- usage of CUME_DIST() for every employee
141 • SELECT row_num, first_name, last_name, salary,
142        RANK() OVER (ORDER BY Salary) as 'Rank',
143        ROUND(CUME_DIST() OVER (ORDER BY Salary),2) as 'Cumulative Distribution'
144 FROM SalaryEmployee1;
145
```

Result Grid

Filter Rows:

Export:

Wrap Cell Content:

	row_num	first_name	last_name	salary	Rank	Cummulative Distribution
	1	Karen	Colmenares	2500	1	0.07
	2	Guy	Himuro	2600	2	0.13
	3	Irene	Mikkilineni	2700	3	0.2
	4	Sigal	Tobias	2800	4	0.27
	5	Shelli	Baida	2900	5	0.33
	6	Alexander	Khoo	3100	6	0.4
	7	Britney	Everett	3900	7	0.47
	8	Sarah	Bell	4000	8	0.53
	9	Diana	Lorentz	4200	9	0.6
	10	Jennifer	Whalen	4400	10	0.67
	11	David	Austin	4800	11	0.8
	12	Valli	Pataballa	4800	11	0.8
	13	Bruce	Ernst	6000	13	0.93
	14	Pat	Fay	6000	13	0.93
	15	Charles	Johnson	6200	15	1

Result 46

16. Compute the CUME_DIST() for one employee "Irene".

CUME_DIST() – this window feature calculates the calculate a cumulative distribution of a value within a group of values. . It represents the number of rows with values less than or equal to that row's value divided by the total number of row(value ranges from 0 and 1)

Formula : ROW_NUMBER / total_rows

See screenshot for the SQL Command and the table output.

NOTE: Salary with the same rank MUST have the same CUME_DIST value

```
146
147  -- usage of CUME_DIST() for one employee say "Irene"
148 • SELECT * FROM(
149     SELECT row_num, first_name, last_name, salary,
150     ROUND(CUME_DIST() OVER (ORDER BY Salary),2) as 'Cumulative Distribution'
151     FROM SalaryEmployee1) as table1
152 WHERE first_name ="Irene";
153
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	Cumulative Distribution
▶	3	Irene	Mikkilineni	2700	0.2

17. Compute the CUME_DIST() for one employee "Bruce".

CUME_DIST() – this window feature calculates the calculate a cumulative distribution of a value within a group of values. . It represents the number of rows with values less than or equal to that row's value divided by the total number of row(value ranges from 0 and 1)

Formula : ROW_NUMBER / total_rows

See screenshot for the SQL Command and the table output.

NOTE: Salary with the same rank MUST have the same CUME_DIST value

```
147  -- usage of CUME_DIST() for one employee say "Bruce"
148 • SELECT * FROM(
149     SELECT row_num, first_name, last_name, salary,
150     ROUND(CUME_DIST() OVER (ORDER BY Salary),2) as 'Cumulative Distribution'
151     FROM SalaryEmployee1) as table1
152 WHERE first_name ="Bruce";
153
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: |

	row_num	first_name	last_name	salary	Cumulative Distribution
▶	13	Bruce	Ernst	6000	0.93