Module 2
Databases for Data Science

CO1: Ability to obtain fundamental knowledge on data science.

Dr. Prakash M
School of Computer Science and Engineering,
Vellore Institute of Technology

SQL – Tool for Data Science

- Structured Query Language (SQL) is a language designed for **managing data in a relational database**.
- Since the 1970s, it is the **most common method of accessing data** in databases today.
- SQL has a variety of **built-in** functions that allow its users to **read, write, update, and delete data.**

It is a popular language for data analysts for a few reasons:

- It can **directly access a large amount of data** without copying it in other applications.
- It can be used to **deal with data of almost any shape and massive size**.
- Data analysis done in SQL is **easy to audit and replicate** as compared to an Excel sheet or CSV file. Excel is great with smaller datasets but not useful for large-sized databases.
- **Joining tables, automating, and reusing code** are much easier in SQL than in Excel.
- SQL is **simple and easy to learn** not just for engineers, developers, data analysts, data scientists, but for anyone willing to invest a few days in learning.

Basic Statistics with SQL

Mean Value

- The average of the dataset is calculated by dividing the total sum by the number of values (count) in the dataset.
- This operation can be performed in SQL by using built-in operation Avg.

```
SELECT Avg(ColumnName) as MEAN
FROM TableName
```

- Demo: Open https://sqliteonline.com/ Create and insert record mentioned below to calculate mean

```
-- Create the table
CREATE TABLE employees (
   salary INT,
   name VARCHAR(50)
);

-- Insert data into the table
INSERT INTO employees (salary, name) VALUES (1000, 'Amit');
INSERT INTO employees (salary, name) VALUES (2000, 'Nisha');
INSERT INTO employees (salary, name) VALUES (3000, 'Yogesh');
INSERT INTO employees (salary, name) VALUES (4000, 'Puja');
INSERT INTO employees (salary, name) VALUES (9000, 'Ram');
INSERT INTO employees (salary, name) VALUES (7000, 'Husain');
INSERT INTO employees (salary, name) VALUES (8000, 'Risha');
INSERT INTO employees (salary, name) VALUES (5000, 'Anil');
INSERT INTO employees (salary, name) VALUES (10000, 'Kumar');
INSERT INTO employees (salary, name) VALUES (6000, 'Shiv');

Select * from employees;
SELECT Avg(salary) as MEAN FROM   employees;
```

Mode Value

- The mode is the value that appears most frequently in a series of the given data.
- SQL does not provide any built-in function for this, so we need to write the following queries to calculate it.

```
SELECT TOP 1 ColumnName
FROM TableName
GROUP BY ColumnName
ORDER BY COUNT(*) DESC
```

--Mode from employee table

```
SELECT salary
FROM employees
GROUP BY salary
ORDER BY COUNT(*) DESC
LIMIT 1;
```

Median Value

- In a sorted list (ascending or descending) of numbers, the median value is the middle number and can be more descriptive of that dataset than the average.
- For an odd amount of numeric dataset, the median value is the number that is in the middle.
- For an even amount of numbers in the list, the middle two numbers are added together, and this sum is divided by two to calculate the median value.
- SQL does not have a built-in function to calculate the median value of a column. The following SQL code can be used to calculate the median value

Code to calculate the median of salary column

```
SET @rindex := -1;
SELECT AVG(m.sal) FROM
    (SELECT @rindex:=@rindex + 1 AS rowindex, Emp.salary AS
sal
    FROM Emp
    ORDER BY Emp.salary) AS m
WHERE
m.rowindex IN (FLOOR(@rindex / 2) , CEIL(@rindex / 2));
```

Data Munging with SQL

- Data munging (or wrangling) is the phase of **data transformation**.
- It transforms data into various states so that it is **simpler to work and understand the data**.
- The transformation may lead to **manually convert or merge or update the data manually** in a certain format to generate data, which is ready for processing by the data analysis tools.
- In this process, we actually **transform and map data from one format to another format** to make it more valuable for a variety of analytics tools.

UPPER(): The UPPER() string function is used to convert the lower case to the upper case of the input text data.

Input: SELECT UPPER('welcome to Data Science')
Output: WELCOME TO DATA SCIENCE

LOWER(): The **LOWER()** string function is used to convert the upper case to the lower case of the input text data.

Input: SELECT LOWER('WELCOME TO DATA SCIENCE')

Output: welcome to data science

TRIM(): The TRIM() string function removes blanks from the leading and trailing position of the given input data.

Input: SELECT TRIM(' WELCOME TO DATA SCIENCE ')
Output: ‘WELCOME TO DATA SCIENCE’

LTRIM(): The LTRIM() string function is used to remove the leading blank spaces from a given input string.

Input: SELECT LTRIM('   WELCOME TO DATA SCIENCE')

Output: WELCOME TO DATA SCIENCE’

RTRIM(): The RTRIM() string function is used to remove the trailing blank spaces from a given input string.

Input: SELECT RTRIM('WELCOME TO DATA SCIENCE ')

Output: ‘WELCOME TO DATA SCIENCE’

RIGHT() :The RIGHT() string function is used to return a given number of characters from the right side of the given input string.

Input: SELECT RIGHT('WELCOME TO DATA SCIENCE', 7)

Output: SCIENCE

LEFT() :The LEFT() string function is used to return a given number of characters from the left side of the given input string [4].

Input: SELECT LEFT('WELCOME TO DATA SCIENCE', 7)
Output: WELCOME

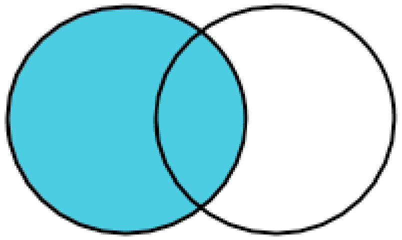
REPLACE() :The REPLACE() string function replaces all the occurrences of a source sub- string with a target substring of a given string.

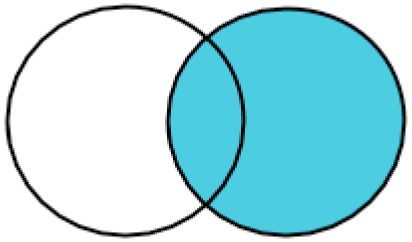
Input: SELECT REPLACE('WELCOME TO DATA SCIENCE', 'DATA', 'INFORMATION')

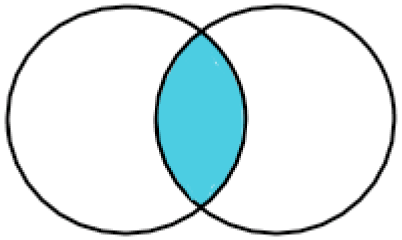
Output: WELCOME TO INFORMATION SCIENCE

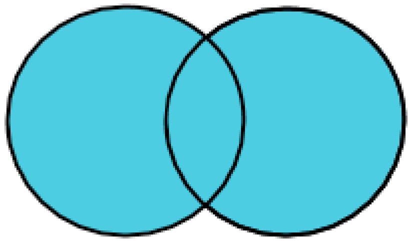
Joins

- In SQL, joins are commands that are used to combine rows from two or more tables.
- These tables are combined based on a related column between those tables. Inner, left, right, and full are four basic types of SQL joins.
- Venn diagram is the easiest way to explain the difference between these four types.
- The result of joining two tables can be represented by the following Venn diagram


Left Join


Right Join


Inner Join

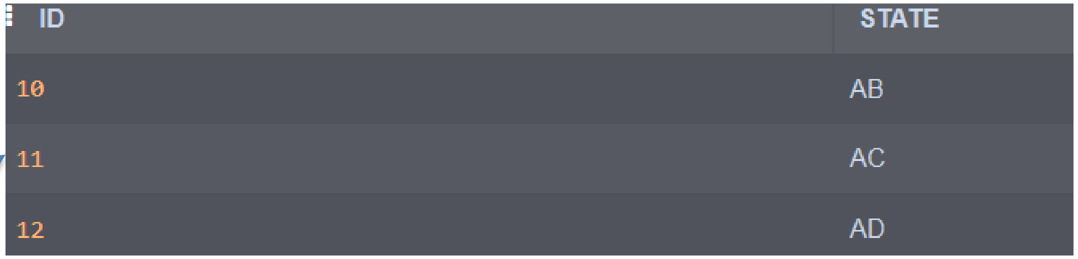

Full Outer Join

Inner Join: The inner join selects all rows from both tables as long as there is a match between the columns.

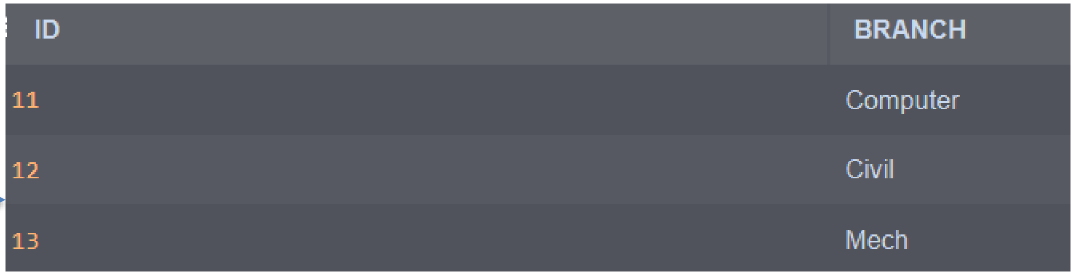
- This join returns those records that have matching values in both tables. So, if you perform an inner join operation between the Emp table and the Dept table, all the records that have matching values in both the tables will be given as output.

SELECT *FROM Emp inner join Dept on Emp.ID = Dept.ID;

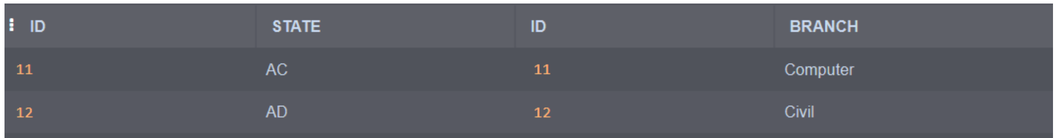
```
--Demo
--Emp Table
CREATE TABLE Emp (
   ID INT PRIMARY KEY,
   STATE VARCHAR(2)
);
--Insert in Emp table
INSERT INTO Emp (ID, STATE) VALUES
(10, 'AB'),
(11, 'AC'),
(12, 'AD');


--Dept Table
CREATE TABLE Dept (
   ID INT PRIMARY KEY,
   BRANCH VARCHAR(20)
);

--Insert into Dept Table
INSERT INTO Dept (ID, BRANCH) VALUES
(11, 'Computer'),
(12, 'Civil'),
(13, 'Mech');

SELECT * from Emp;
SELECT * from Dept;

--Inner Join
SELECT *FROM Emp inner join Dept on Emp.ID = Dept.ID;
```



ID	STATE
10	AB
11	AC
12	AD



ID	BRANCH
11	Computer
12	Civil
13	Mech

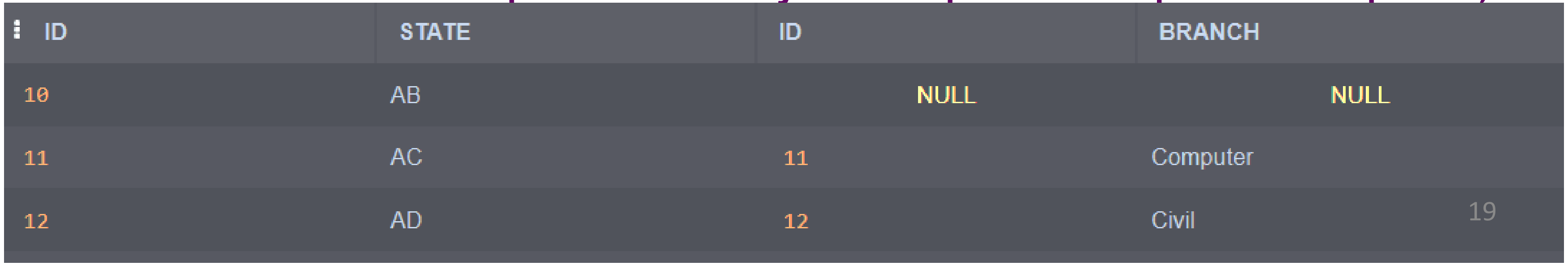


ID	STATE	ID	BRANCH
11	AC	11	Computer
12	AD	12	Civil

Left Outer Join: The left outer join (or left join) returns all the rows of the left side table and matching rows in the right side table of the join.

- The rows for which there is no matching row on the right side, the result will contain *NULL*.
- From the left table (Emp), the left join keyword returns all records, even if there are no matches in the right tables (Dept).

SELECT *FROM Emp left outer join Dept on Emp.ID = Dept.ID;

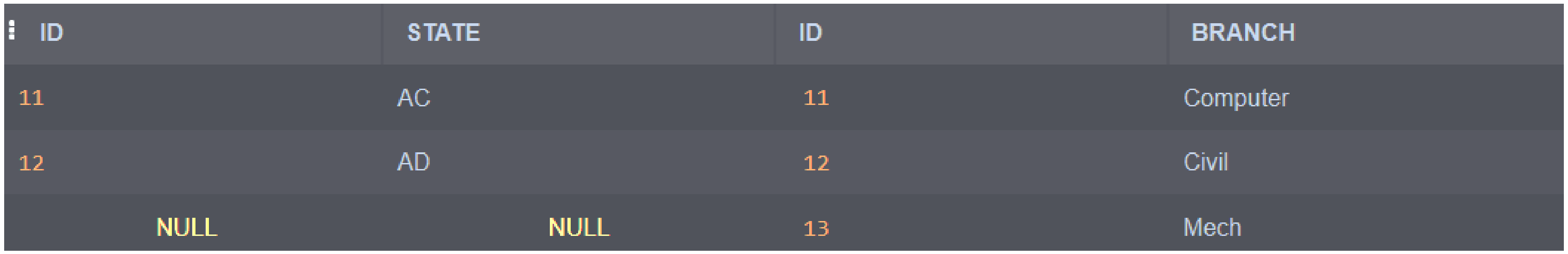


ID	STATE	ID	BRANCH
10	AB	NULL	NULL
11	AC	11	Computer
12	AD	12	Civil

Right Outer Join: The right outer join (or right join) returns all the rows of the right side table and matching rows for the left side table of the join.

- The rows for which there is no matching row on the left side, the result will contain *NULL.*

SELECT *FROM Emp right outer join Dept on Emp.ID = Dept.ID;

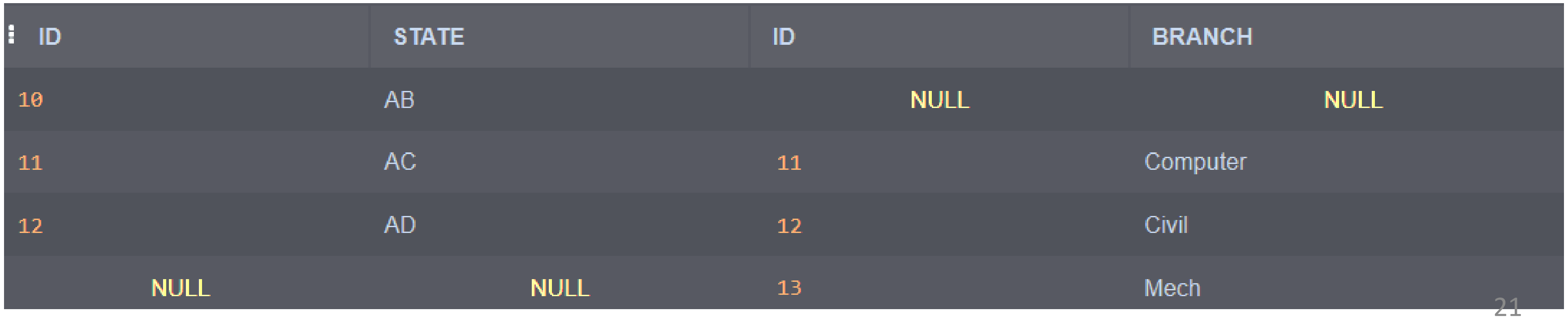


ID	STATE	ID	BRANCH
11	AC	11	Computer
12	AD	12	Civil
NULL	NULL	13	Mech

Full Outer Join: The full outer join (or full join) returns all those records that have a match in either the left table (Emp) or the right table (Dept) table.

- The joined table will contain all records from both the tables, and if there is no matching field on either side, then fill in NULLs.

SELECT *FROM Emp full outer join Dept on Emp.ID = Dept.ID;



ID	STATE	ID	BRANCH
10	AB	NULL	NULL
11	AC	11	Computer
12	AD	12	Civil
NULL	NULL	13	Mech

Aggregation

SQL provides the following built-in functions for aggregating data.

- The COUNT() function returns the number of values in the dataset ("salary" is column in table "Emp").

  ```
  SELECT COUNT(salary) FROM Emp
  ```

- The AVG() function returns the average of a group of selected numeric column values.

  ```
  SELECT AVG(salary)FROM Emp
  ```

- The SUM() function returns the total sum of a numeric column.

  ```
  SELECT SUM(salary)FROM Emp
  ```

- The MIN() function returns the minimum value in the selected column.

  ```
  SELECT MIN(salary)FROM Emp
  ```

- The MAX() function returns the maximum value in the selected column.

  ```
  SELECT MAX(salary)FROM Emp
  ```

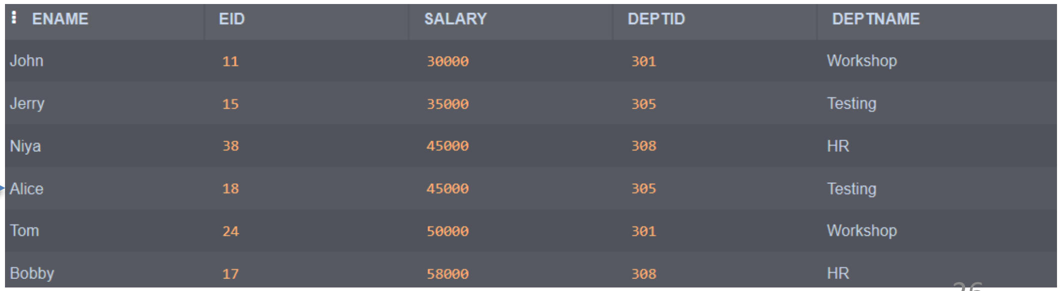
Filtering

- The data generated from the reports of various application software often results in complex and large datasets.
- This **dataset may consist of redundant records or impartial records**. This useless data may confuse the user.
- Filtering this redundant and useless data can also make the dataset more efficient and useful.

- Data filtering is one of the major steps involved in data science due to various reasons, and some are listed below:
  1. During certain situations, we may require a **specific part of the actual data for analysis**.
  2. Sometimes, we may **require reducing the actual retrieved data by removing redundant records** as that may result in wrong analysis.
  3. **Query performance can be greatly enhanced by applying it to refined data**. Also, it can reduce strain on application.

- Data filtering process consists of different strategies for refining and reducing datasets.
- To understand data filtering using SQL, we will use the following dataset throughout further queries

```
--Demo – Open SQLLite online
--worker table
CREATE TABLE workers (
   ENAME VARCHAR(50),
   EID INT PRIMARY KEY,
   SALARY DECIMAL(10, 2),
   DEPTID INT,
   DEPTNAME VARCHAR(50)
);

--insert 6 records into workers table
INSERT INTO workers (ENAME, EID, SALARY, DEPTID, DEPTNAME) VALUES
('John', 11, 30000.00, 301, 'Workshop'),
('Jerry', 15, 35000.00, 305, 'Testing'),
('Niya', 38, 45000.00, 308, 'HR'),
('Alice', 18, 45000.00, 305, 'Testing'),
('Tom', 24, 50000.00, 301, 'Workshop'),
('Bobby', 17, 58000.00, 308, 'HR');

SELECT * from workers;
```



ENAME	EID	SALARY	DEPTID	DEPTNAME
John	11	30000	301	Workshop
Jerry	15	35000	305	Testing
Niya	38	45000	308	HR
Alice	18	45000	305	Testing
Tom	24	50000	301	Workshop
Bobby	17	58000	308	HR

Syntax to filter data using WHERE

```
SELECT *
FROM tablename;
```

- The above query extracts all data from the table. An asterisk (*) in the above simple query indicates that “select all the data” in the table.
- In the above query, when we add WHERE clause with the condition after WHERE, it filters data in the table and returns only those records that satisfy the condition given after WHERE clause.
- WHERE clause can be used as follows:

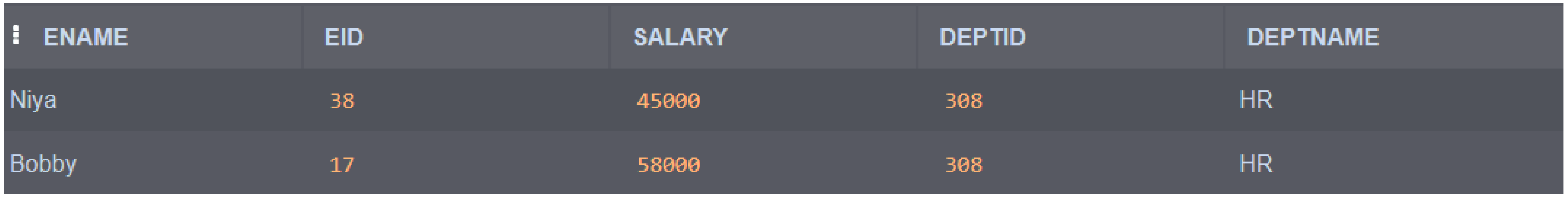
```
SELECT *
FROM tablename
WHERE columnname = expected_value;
```

- Instead of the equal sign (=) operator in the condition statement of **WHERE** clause, we can use the following operators too:

>	(greater than),
<	(lessthan),
>=	(greater than or equal to),
<=	(less than or equal to), and
!=	(not equal to).

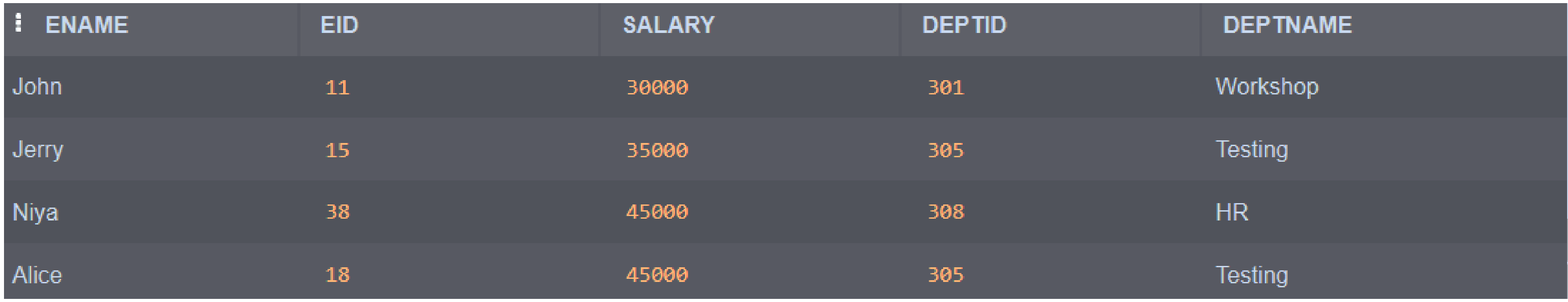
- Suppose we want to extract details of those employees who are working in "HR" department in the workers table, then we can write the query as follows:

```
SELECT * FROM workers WHERE DEPTNAME='HR’;
```



ENAME	EID	SALARY	DEPTID	DEPTNAME
Niya	38	45000	308	HR
Bobby	17	58000	308	HR

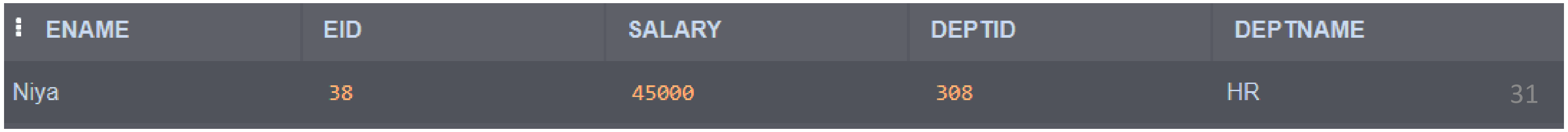
SELECT * FROM workers WHERE SALARY<=47000;



ENAME	EID	SALARY	DEPTID	DEPTNAME
John	11	30000	301	Workshop
Jerry	15	35000	305	Testing
Niya	38	45000	308	HR
Alice	18	45000	305	Testing

- To fetch required data, sometimes, we may require to force **two or more conditions**.
- We can use **AND, OR operators** to achieve this. Only those records that satisfy all the conditions in the query will be retrieved when AND operator is used between two conditions.
- For example, to find **workers in the HR department who have salary more than 47,000**, we can write the query as follows.

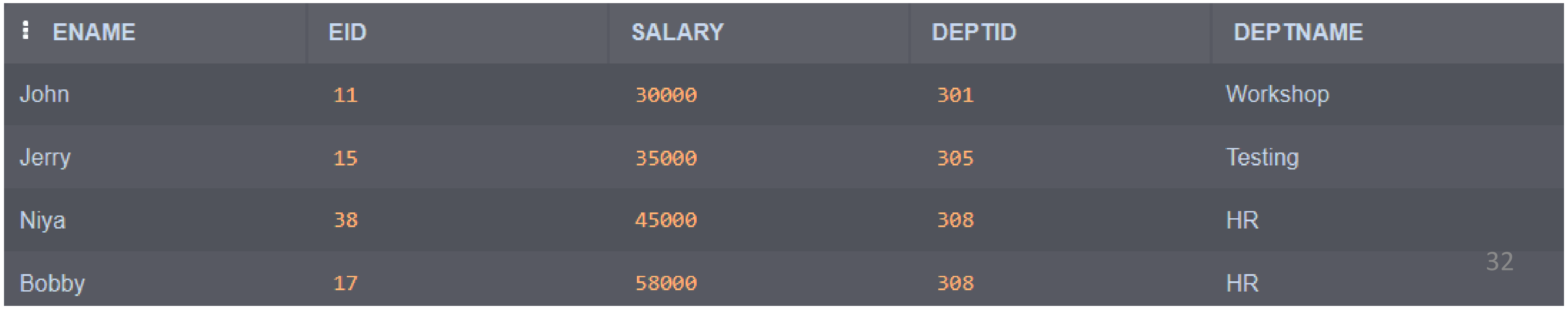
```
SELECT * FROM workers WHERE SALARY<=47000 AND
DEPTNAME='HR';
```



ENAME	EID	SALARY	DEPTID	DEPTNAME
Niya	38	45000	308	HR

- If OR is used between two conditions, then all records that satisfy either condition will get retrieved along with records that satisfy both conditions.
- The following query will fetch the details of the **workers who are working in the HR department or who have a salary less than 36,000.**

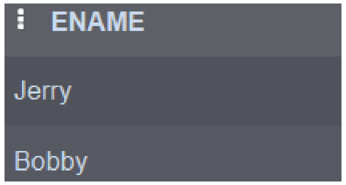
```
SELECT * FROM workers WHERE SALARY<=36000 OR
DEPTNAME='HR';
```



ENAME	EID	SALARY	DEPTID	DEPTNAME
John	11	30000	301	Workshop
Jerry	15	35000	305	Testing
Niya	38	45000	308	HR
Bobby	17	58000	308	HR

- Sometimes, we may want to match a pattern in text data. The **LIKE** clause can be used to specify a pattern matching condition. **Two wildcards, percent sign “%” and underscore “_”,** are used to specify conditions.
- The percent sign is used to represent any string of zero or more characters, and underscore represents a single number or character.
- For example, to retrieve ENAME that ends with character “y” of workers table, we can write the query as follows:

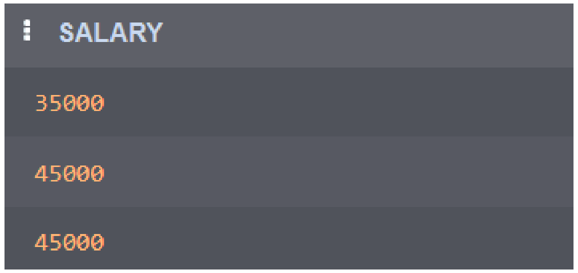
SELECT ENAME FROM workers WHERE ENAME like '%y';



```
ENAME
Jerry
Bobby
```

- The following query retrieves records of salary, which has “5” at second position in workers table.

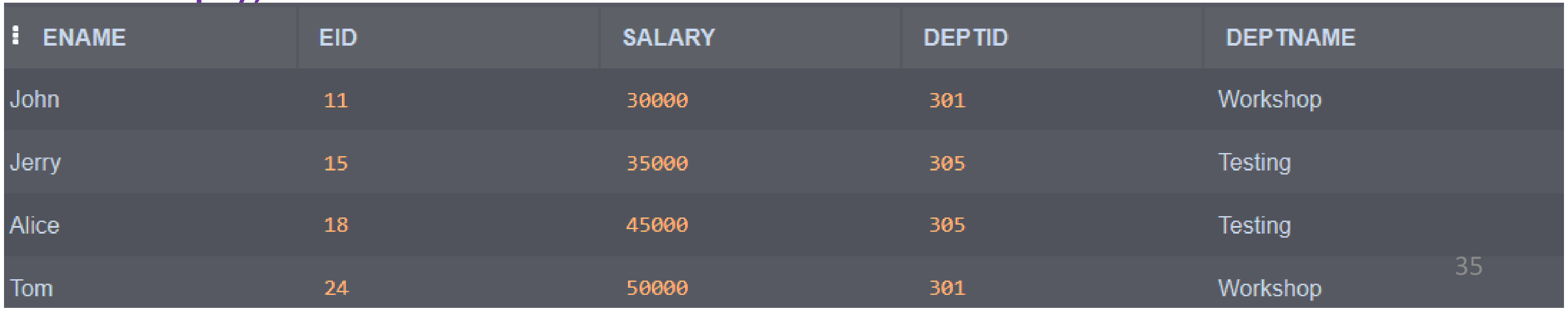
```
SELECT SALARY FROM workers WHERE SALARY like '_5%';
```



SALARY
35000
45000
45000

- Sometimes, we may need to **filter records based on match of multiple values** in a given dataset.
- The SQL IN operator allows you to test if the given expression matches any value in the list of values.
- If the records matched with any one of the values in the list, then it is returned as result. For example,

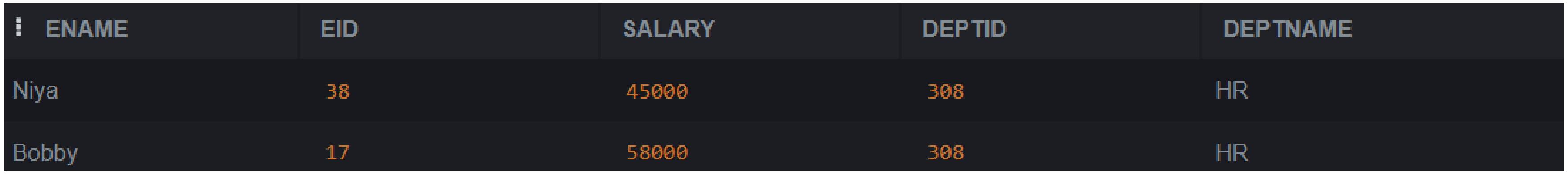
SELECT * FROM workers WHERE DEPTNAME IN('Testing', 'Workshop');



ENAME	EID	SALARY	DEPTID	DEPTNAME
John	11	30000	301	Workshop
Jerry	15	35000	305	Testing
Alice	18	45000	305	Testing
Tom	24	50000	301	Workshop

- Sometimes, we may **want to exclude some values**; we can use NOT keyword in query. The following query returns those workers' details whose DEPTNAME is not "Workshop" or "Testing".

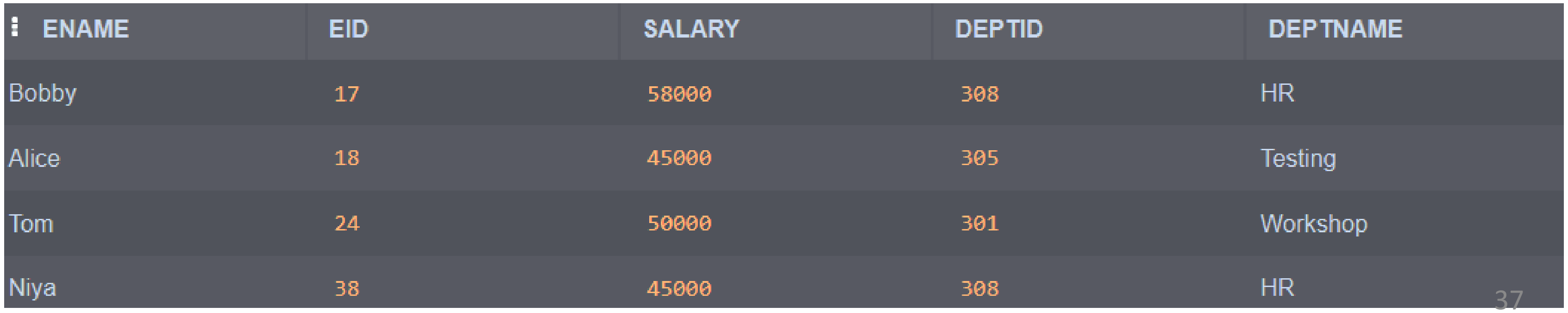
SELECT * FROM workers WHERE DEPTNAME NOT IN ('Testing', 'Workshop');



ENAME	EID	SALARY	DEPTID	DEPTNAME
Niya	38	45000	308	HR
Bobby	17	58000	308	HR

- A subquery is a query within another query. A subquery (called a nested query or subselect) is a SELECT query embedded within the WHERE clause of another query.

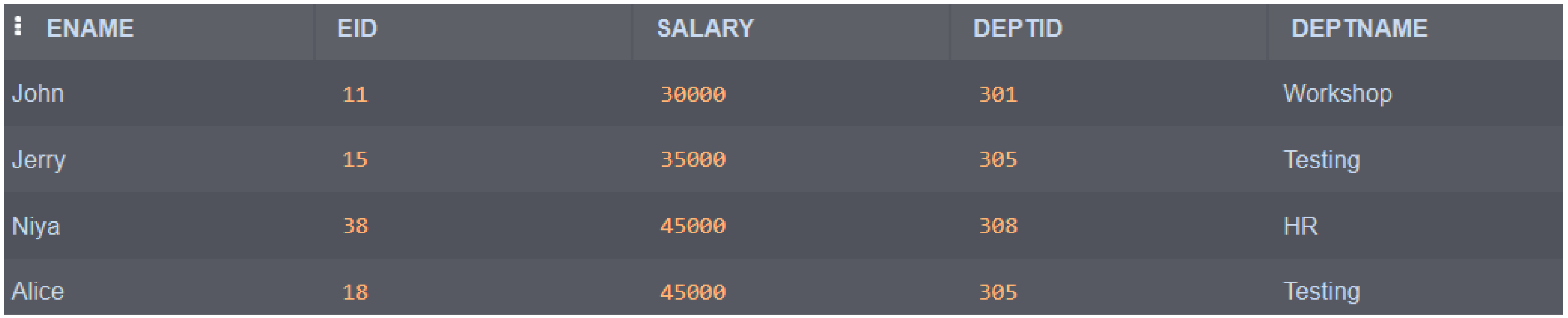
SELECT * FROM workers WHERE EID IN (select EID FROM workers WHERE DEPTNAME='HR' OR SALARY>=40000);



ENAME	EID	SALARY	DEPTID	DEPTNAME
Bobby	17	58000	308	HR
Alice	18	45000	305	Testing
Tom	24	50000	301	Workshop
Niya	38	45000	308	HR

- we may need to **filter records based on match of a large range of values**.
- You can use the **keyword BETWEEN** for this purpose.
- It allows you to specify a start value and an end value of required range. This clause is a shorthand representation for two conditions with >= and <= operators.
- For example, to retrieve details of those workers having salary >= 30,000 and <= 45,000, we can write the query as follows

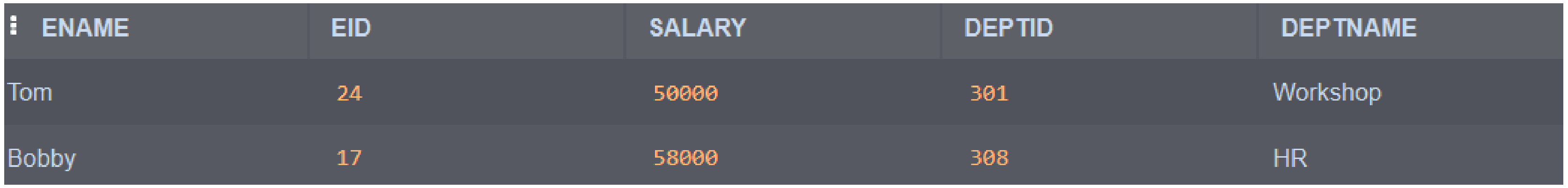
```
SELECT * FROM workers WHERE SALARY BETWEEN 30000 and 45000;
```



ENAME	EID	SALARY	DEPTID	DEPTNAME
John	11	30000	301	Workshop
Jerry	15	35000	305	Testing
Niya	38	45000	308	HR
Alice	18	45000	305	Testing

- To retrieve details of workers whose salary is not in the range of >= 30,000 and <= 45,000, we can write the query using NOT BETWEEN clause as follows:

```
SELECT * FROM workers WHERE SALARY NOT BETWEEN 30000 and 45000;
```



ENAME	EID	SALARY	DEPTID	DEPTNAME
Tom	24	50000	301	Workshop
Bobby	17	58000	308	HR

- Summary – Filtering
  - WHERE CLAUSE – RELATIONAL OPERATORS
  - AND
  - OR
  - LIKE
  - IN
  - NOT IN
  - BETWEEN
  - NOT BETWEEN

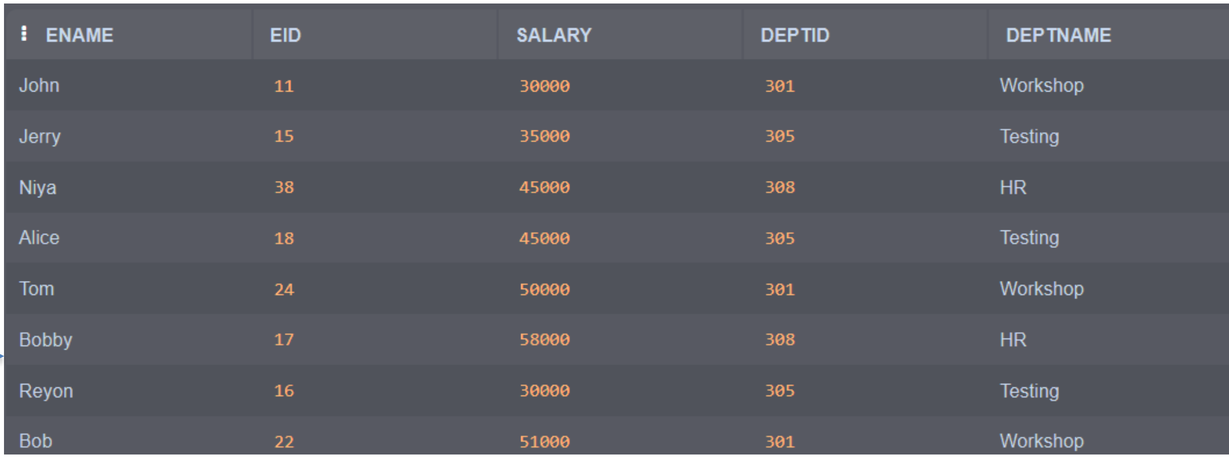
Window Functions and Ordered Data

- SQL window functions **calculate their result based on a set of rows rather than on a single row**.
- In SQL window functions, the word **“window” refers to a set of rows**.
- It is **similar to aggregate functions**, but when we use the **GROUP BY** clause with aggregate functions, it groups the result set based on one or more columns.
- The **aggregate function** is performed on the rows as an **entire group**.
- SQL window functions generate a result with some attributes of an individual row together with the results of the window function

- Window functions can be **called with the SELECT statement or the ORDER BY** clause, but **cannot be called in the WHERE** clause.
- The window function **calculates on a set of rows and returns a value for each row** from the given query.
- It calculates the returned values based on the values of the rows in a window.

- In the window function query syntax, the **window is defined using the OVER()** clause. In a **single query**, we can also include **multiple window** functions.
- The **OVER() clause** has the following **subclauses**:
  - **PARTITION BY** clause to define window partitions to form groups of rows on which window function will be applied.
  - **ORDER BY** clause for logical sorting of rows within a partition.
- To demonstrate SQL window function, we will use the following “workers” table

```
--Demo
-- Step 1: Create the table
CREATE TABLE workers (
    ENAME VARCHAR(50),
    EID INT,
    SALARY INT,
    DEPTID INT,
    DEPTNAME VARCHAR(50)
);

-- Step 2: Insert data into the table
INSERT INTO workers (ENAME, EID, SALARY, DEPTID, DEPTNAME) VALUES
('John', 11, 30000, 301, 'Workshop'),
('Jerry', 15, 35000, 305, 'Testing'),
('Niya', 38, 45000, 308, 'HR'),
('Alice', 18, 45000, 305, 'Testing'),
('Tom', 24, 50000, 301, 'Workshop'),
('Bobby', 17, 58000, 308, 'HR'),
('Reyon', 16, 30000, 305, 'Testing'),
('Bob', 22, 51000, 301, 'Workshop');

SELECT * from workers;
```



ENAME	EID	SALARY	DEPTID	DEPTNAME
John	11	30000	301	Workshop
Jerry	15	35000	305	Testing
Niya	38	45000	308	HR
Alice	18	45000	305	Testing
Tom	24	50000	301	Workshop
Bobby	17	58000	308	HR
Reyon	16	30000	305	Testing
Bob	22	51000	301	Workshop

RANK () Window Function

- The RANK() function returns the **position of any row in the specified partition**.
- The **OVER** and **PARTITION BY** functions are used to **divide the result set into partitions according** to specified criteria.
- Further, **ORDER BY** clause can be used to sort data in **ascending or descending** order based on some attribute.

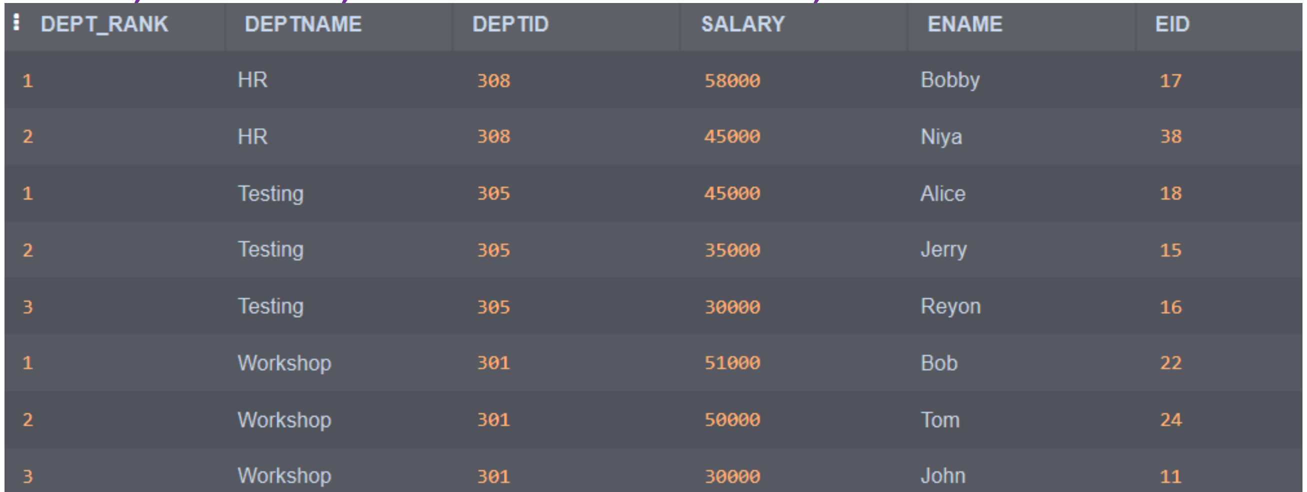
- To rank salaries within departments, we can write the query as follows:

  SELECT RANK()

  OVER (PARTITION BY DEPTNAME ORDER BY SALARY DESC) AS DEPT_RANK,

  DEPTNAME,

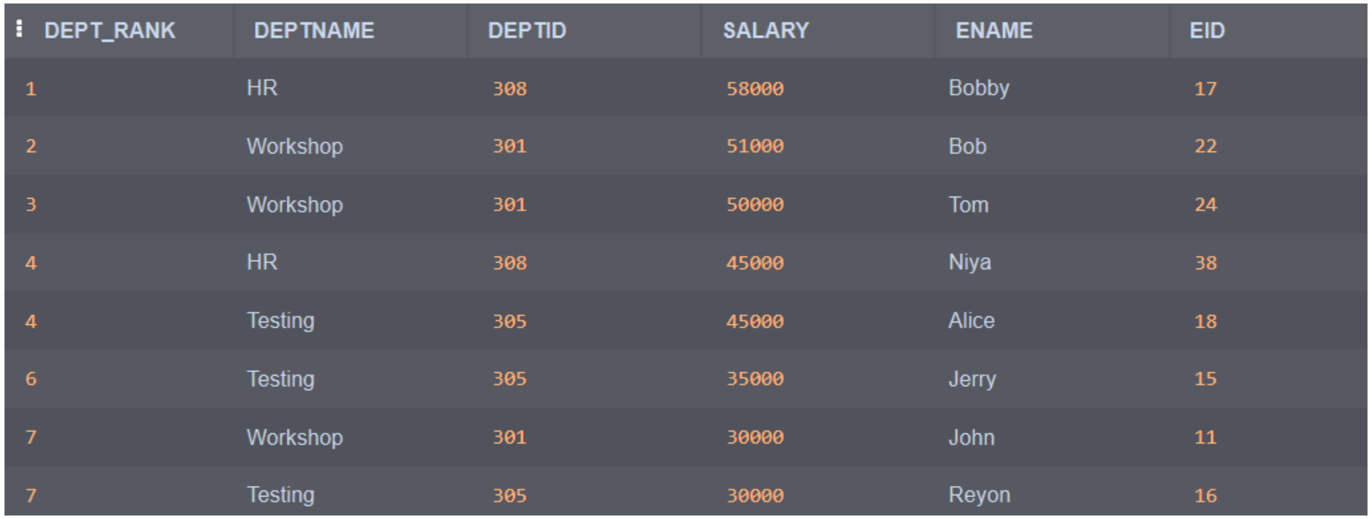
  DEPTID, SALARY, ENAME, EID FROM workers;



DEPT_RANK	DEPTNAME	DEPTID	SALARY	ENAME	EID
1	HR	308	58000	Bobby	17
2	HR	308	45000	Niya	38
1	Testing	305	45000	Alice	18
2	Testing	305	35000	Jerry	15
3	Testing	305	30000	Reyon	16
1	Workshop	301	51000	Bob	22
2	Workshop	301	50000	Tom	24
3	Workshop	301	30000	John	11

- If we exclude PARTITION BY clause from the above query, then we will get the result as follows:

SELECT RANK() OVER (ORDER BY SALARY DESC) AS DEPT_RANK, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;



DEPT_RANK	DEPTNAME	DEPTID	SALARY	ENAME	EID
1	HR	308	58000	Bobby	17
2	Workshop	301	51000	Bob	22
3	Workshop	301	50000	Tom	24
4	HR	308	45000	Niya	38
4	Testing	305	45000	Alice	18
6	Testing	305	35000	Jerry	15
7	Workshop	301	30000	John	11
7	Testing	305	30000	Reyon	16

- Rank 5 and 8 skips

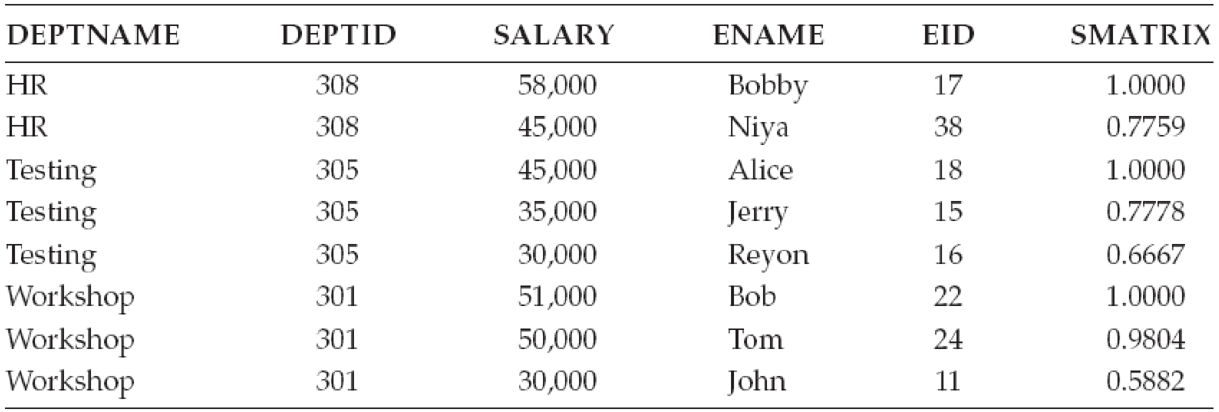
- The RANK() function **skips the rank 5 and rank 8** in the above result because two rows share the fourth rank and two records share the seventh rank.
- The RANK function skips the next k−1 ranks if there is a tie between k previous ranks.

- Suppose we want to find out each **employee's salary ranks in relation to the top salary of their department**. This can be calculated by following math expression:

  Salary / max_salary_in_dept

- The next query will show all employees ordered by the above metric; the **employees with the lowest salary** (relative to their highest departmental salary) **will be listed first**

```
SELECT DEPTNAME, DEPTID, SALARY, ENAME, EID,
SALARY / MAX(SALARY) OVER
(PARTITION BY DEPTNAME ORDER BY SALARY DESC) AS SMATRIX
FROM workers ORDER BY DEPTNAME
```



DEPTNAME	DEPTID	SALARY	ENAME	EID	SMATRIX
HR	308	58,000	Bobby	17	1.0000
HR	308	45,000	Niya	38	0.7759
Testing	305	45,000	Alice	18	1.0000
Testing	305	35,000	Jerry	15	0.7778
Testing	305	30,000	Reyon	16	0.6667
Workshop	301	51,000	Bob	22	1.0000
Workshop	301	50,000	Tom	24	0.9804
Workshop	301	30,000	John	11	0.5882

PERCENT_RANK()

In SQL Server, the **PERCENT_RANK()** function calculates the SQL percentile rank of each row. This **percentile** ranking number may **range from zero to one**.

- For each row, **PERCENT_RANK()** calculates the percentile rank using the following formula:

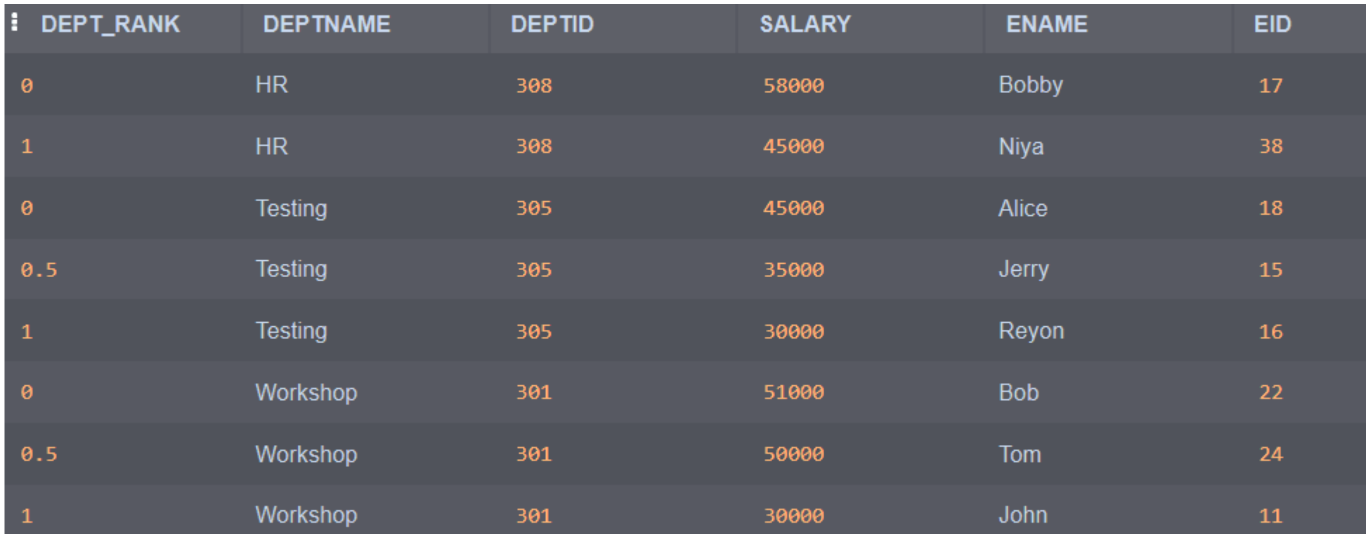
  (Rank – 1) / (total _ number _ rows – 1)

- In this formula, **rank represents the rank of the row**. total_number_rows is the number of rows that are being evaluated. It always **returns the rank of the first row as 0**.

- It divides the rows into partitions by using the **PARTITION BY** clause, and the **ORDER BY** clause logically sorts (in ascending or descending order) rows for each partition. The percentile rank value is calculated for each ordered group independently

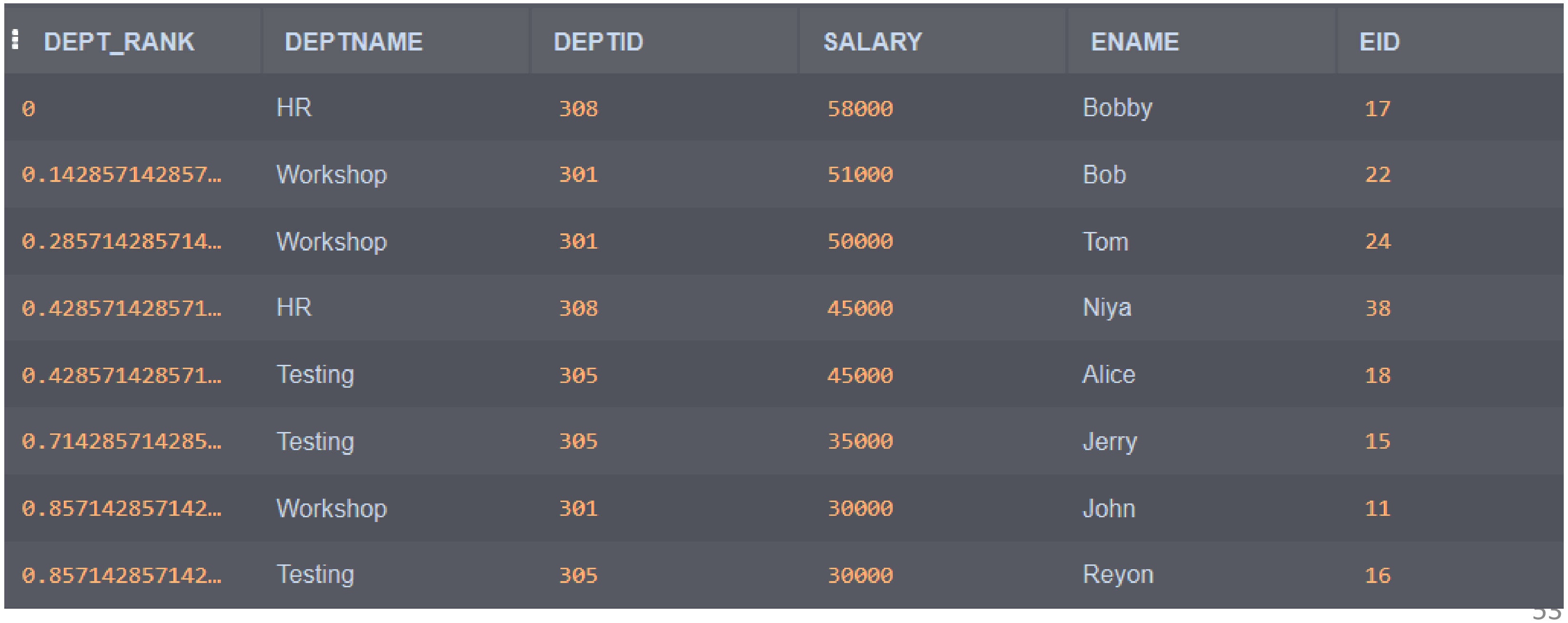
SELECT PERCENT_RANK() OVER

(PARTITION BY DEPTNAME ORDER BY SALARY DESC) AS DEPT_RANK, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;



DEPT_RANK	DEPTNAME	DEPTID	SALARY	ENAME	EID
0	HR	308	58000	Bobby	17
1	HR	308	45000	Niya	38
0	Testing	305	45000	Alice	18
0.5	Testing	305	35000	Jerry	15
1	Testing	305	30000	Reyon	16
0	Workshop	301	51000	Bob	22
0.5	Workshop	301	50000	Tom	24
1	Workshop	301	30000	John	11

```
SELECT PERCENT_RANK() OVER
(ORDER BY SALARY DESC) AS DEPT_RANK,
DEPTNAME, DEPTID, SALARY, ENAME, EID FROM
workers;
```

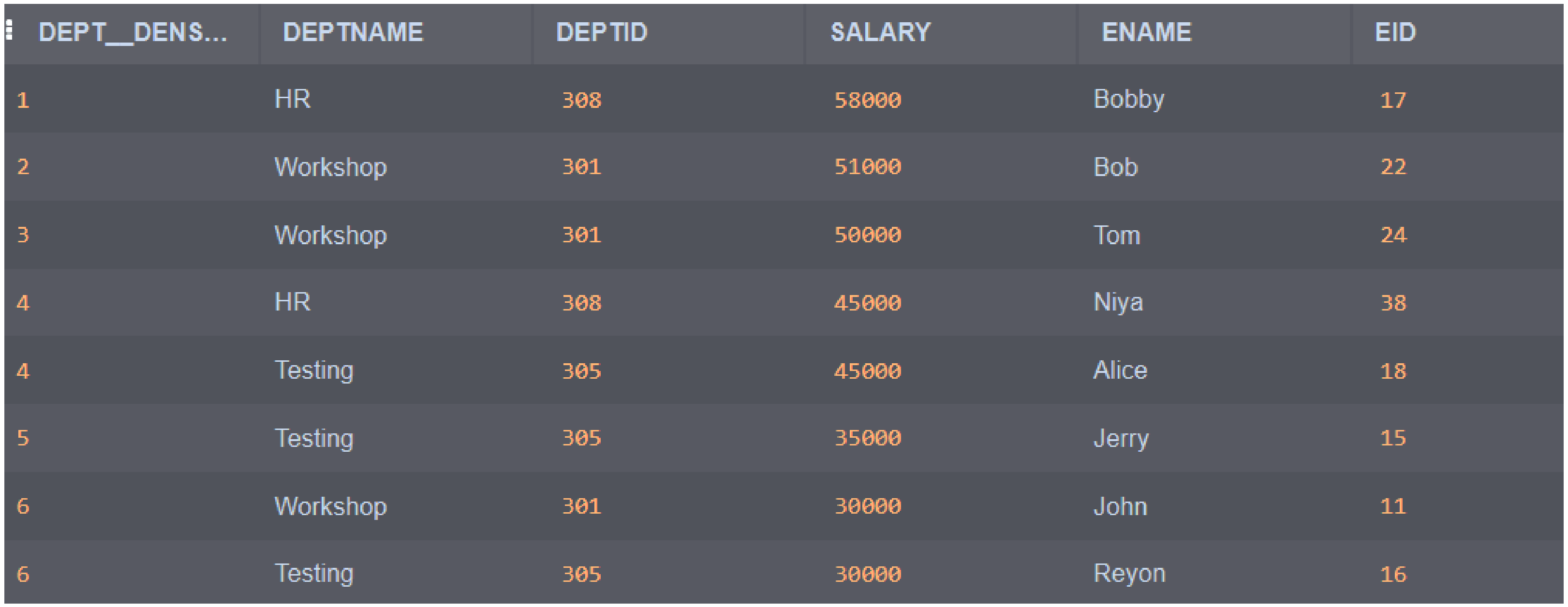


DEPT_RANK	DEPTNAME	DEPTID	SALARY	ENAME	EID
0	HR	308	58000	Bobby	17
0.142857142857…	Workshop	301	51000	Bob	22
0.285714285714…	Workshop	301	50000	Tom	24
0.428571428571…	HR	308	45000	Niya	38
0.428571428571…	Testing	305	45000	Alice	18
0.714285714285…	Testing	305	35000	Jerry	15
0.857142857142…	Workshop	301	30000	John	11
0.857142857142…	Testing	305	30000	Reyon	16

DENSE_RANK ()

- The DENSE_RANK () window function calculates the **rank of value in a group of rows based on the ORDER BY** expression specified in the OVER clause.
- For each partition, rank starts from 1.
- Rows with the same values receive the same rank.
- DENSE_RANK function **does not keep gaps in ranks** if there is a similarity between previous one or more rows ranks. This feature makes it different from RANK() function

SELECT DENSE_RANK() OVER
(ORDER BY SALARY DESC) AS DEPT__DENSE_RANK,
DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;

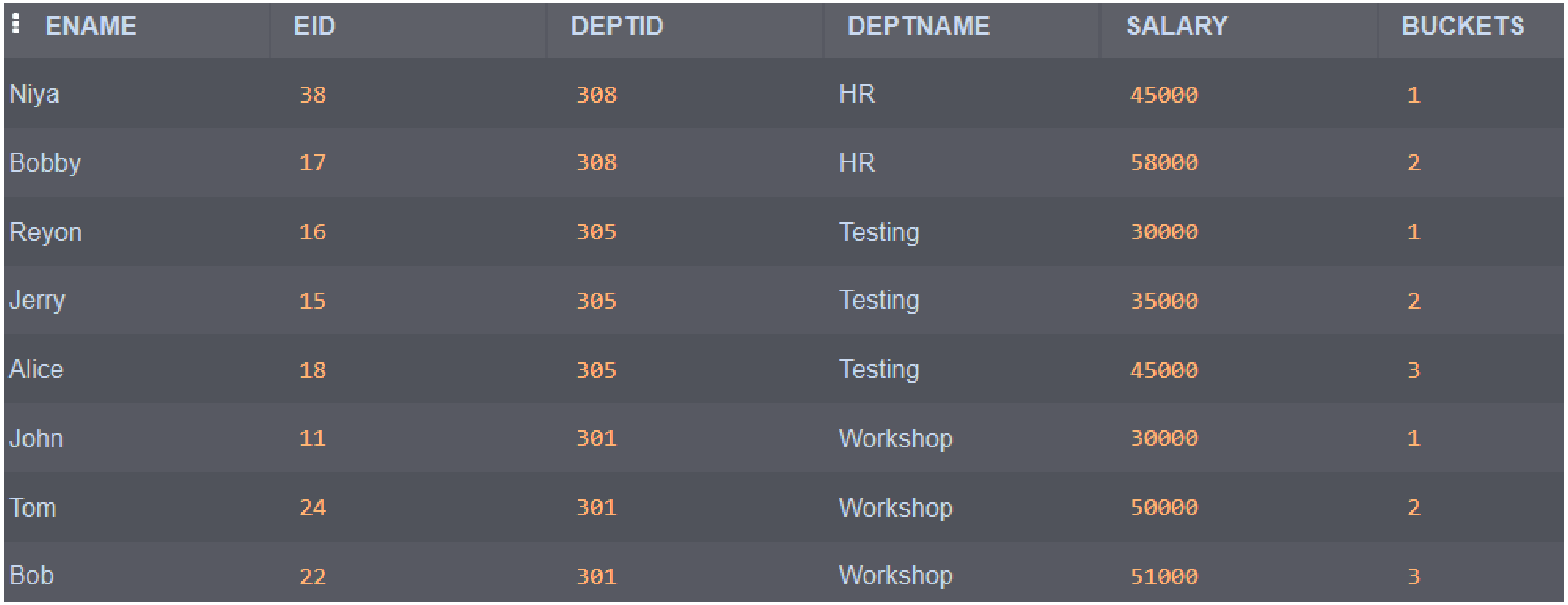


DEPT__DENS...	DEPTNAME	DEPTID	SALARY	ENAME	EID
1	HR	308	58000	Bobby	17
2	Workshop	301	51000	Bob	22
3	Workshop	301	50000	Tom	24
4	HR	308	45000	Niya	38
4	Testing	305	45000	Alice	18
5	Testing	305	35000	Jerry	15
6	Workshop	301	30000	John	11
6	Testing	305	30000	Reyon	16

NTILE()

- The SQL **NTILE() function partitions a logically ordered dataset into a number of buckets** demonstrated by the expression and allocates the **bucket number to each row**.
- The buckets are numbered from 1 through expression where the expression value must result in a positive integer value for each partition.
- For example, the following query will allocate rows to three buckets

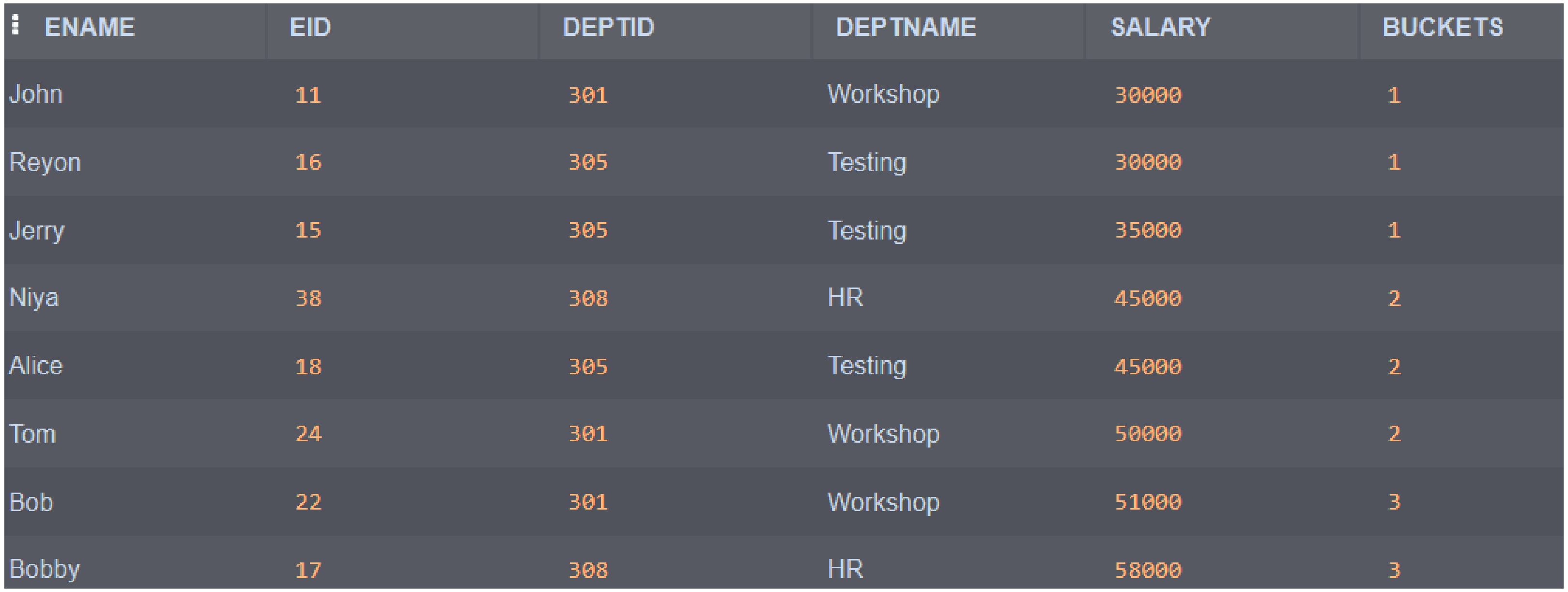
```
SELECT ENAME, EID, DEPTID, DEPTNAME, SALARY, NTILE(3)
OVER (PARTITION BY DEPTNAME ORDER BY SALARY) AS BUCKETS
FROM workers;
```



ENAME	EID	DEPTID	DEPTNAME	SALARY	BUCKETS
Niya	38	308	HR	45000	1
Bobby	17	308	HR	58000	2
Reyon	16	305	Testing	30000	1
Jerry	15	305	Testing	35000	2
Alice	18	305	Testing	45000	3
John	11	301	Workshop	30000	1
Tom	24	301	Workshop	50000	2
Bob	22	301	Workshop	51000	3

- If **PARTITION BY clause is excluded** from the above query, then it will give results as follows

SELECT ENAME, EID, DEPTID, DEPTNAME, SALARY, NTILE(3) OVER (ORDER BY SALARY) AS BUCKETS FROM workers;



ENAME	EID	DEPTID	DEPTNAME	SALARY	BUCKETS
John	11	301	Workshop	30000	1
Reyon	16	305	Testing	30000	1
Jerry	15	305	Testing	35000	1
Niya	38	308	HR	45000	2
Alice	18	305	Testing	45000	2
Tom	24	301	Workshop	50000	2
Bob	22	301	Workshop	51000	3
Bobby	17	308	HR	58000	3

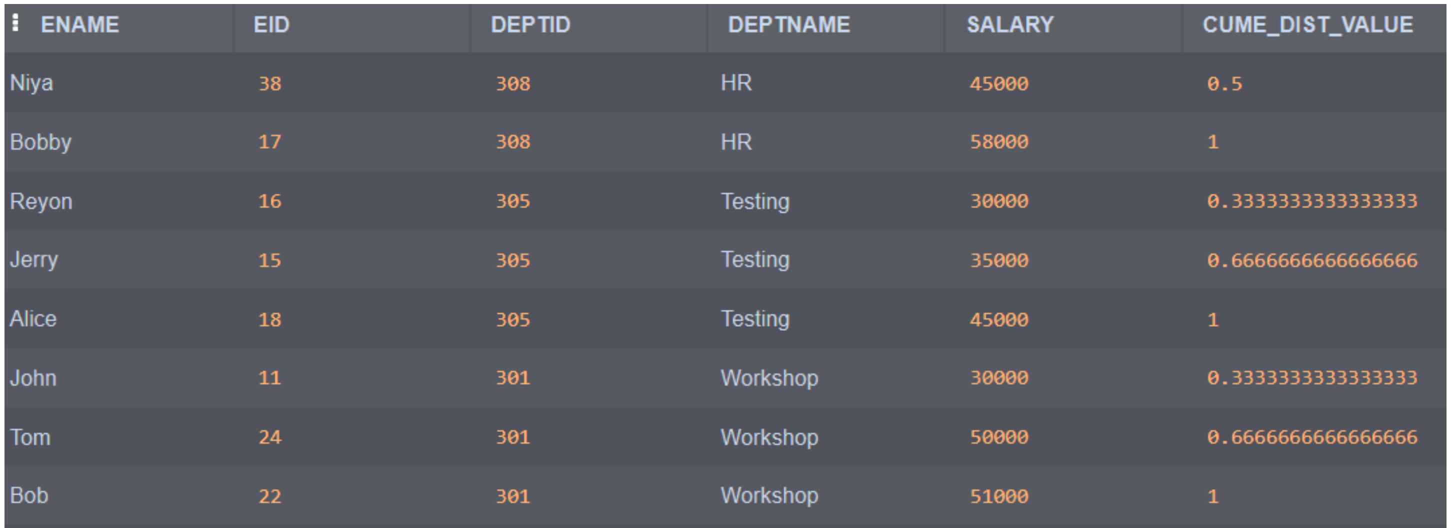
CUME_DIST()

- The SQL window function **CUME_DIST() returns the cumulative distribution** of a value within a partition of values.
- The cumulative distribution of a value calculated by the number of rows with **values less than or equal to (<=) the current row's value** is divided by the total number of rows.

N / totalrows

- where N is the number of rows with the value less than or equal to the current row value and total rows is the number of rows in the group or result set.
- Function returns value having a **range between 0 and 1**

```
SELECT ENAME, EID, DEPTID, DEPTNAME, SALARY, CUME_DIST()
OVER (PARTITION BY DEPTNAME ORDER BY SALARY) AS
CUME_DIST_VALUE FROM workers;
```

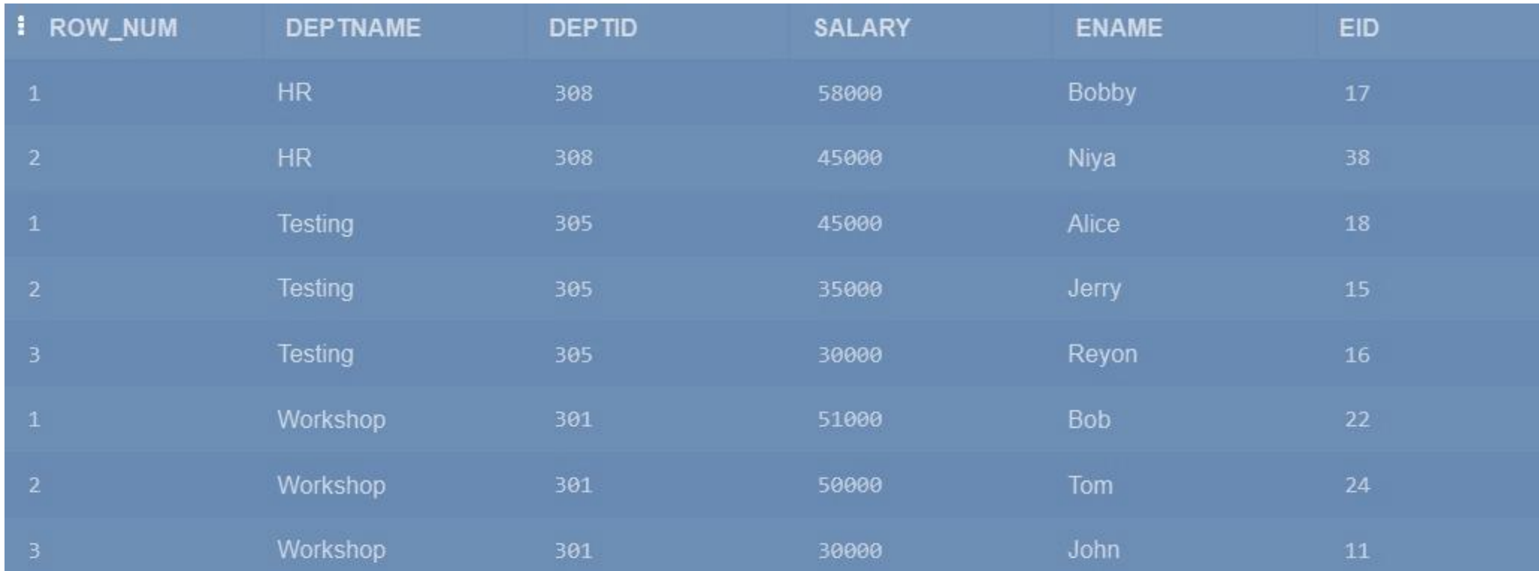


ENAME	EID	DEPTID	DEPTNAME	SALARY	CUME_DIST_VALUE
Niya	38	308	HR	45000	0.5
Bobby	17	308	HR	58000	1
Reyon	16	305	Testing	30000	0.3333333333333333
Jerry	15	305	Testing	35000	0.6666666666666666
Alice	18	305	Testing	45000	1
John	11	301	Workshop	30000	0.3333333333333333
Tom	24	301	Workshop	50000	0.6666666666666666
Bob	22	301	Workshop	51000	1

ROW_NUMBER()

- The SQL window function **ROW_NUMBER()** is used to **display a row number for each row** within a specified partition.

SELECT ROW_NUMBER() OVER (PARTITION BY DEPTNAME ORDER BY SALARY DESC) AS ROW_NUM, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;



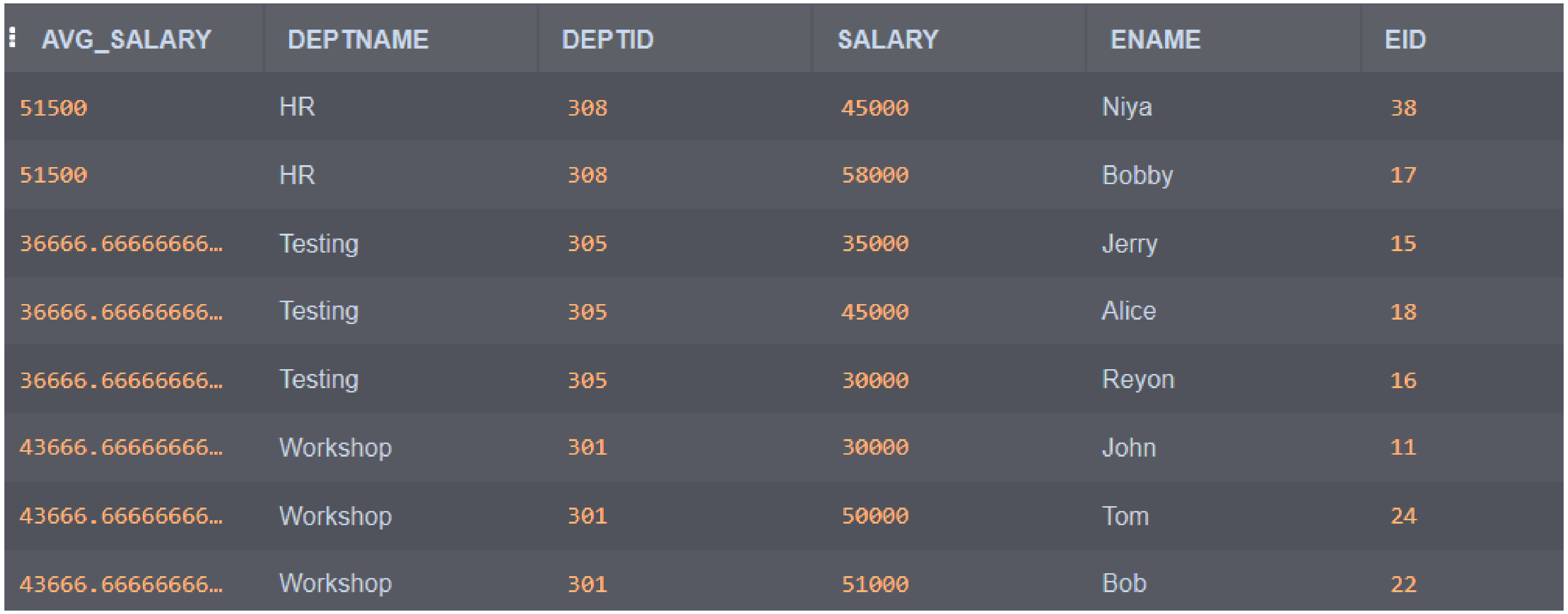
ROW_NUM	DEPTNAME	DEPTID	SALARY	ENAME	EID
1	HR	308	58000	Bobby	17
2	HR	308	45000	Niya	38
1	Testing	305	45000	Alice	18
2	Testing	305	35000	Jerry	15
3	Testing	305	30000	Reyon	16
1	Workshop	301	51000	Bob	22
2	Workshop	301	50000	Tom	24
3	Workshop	301	30000	John	11

AVG()

- A *window function* **applies function across a set of table rows that are related to the current row**.
- The window function does not cause rows to be clustered into a single output row; the rows maintain their separate identities.
- The window function is able to access more than just the current row of the query result.
- To calculate average value of each partition, we can use window function **AVG().**

- To calculate average salary in each department, we can write the query as follows:

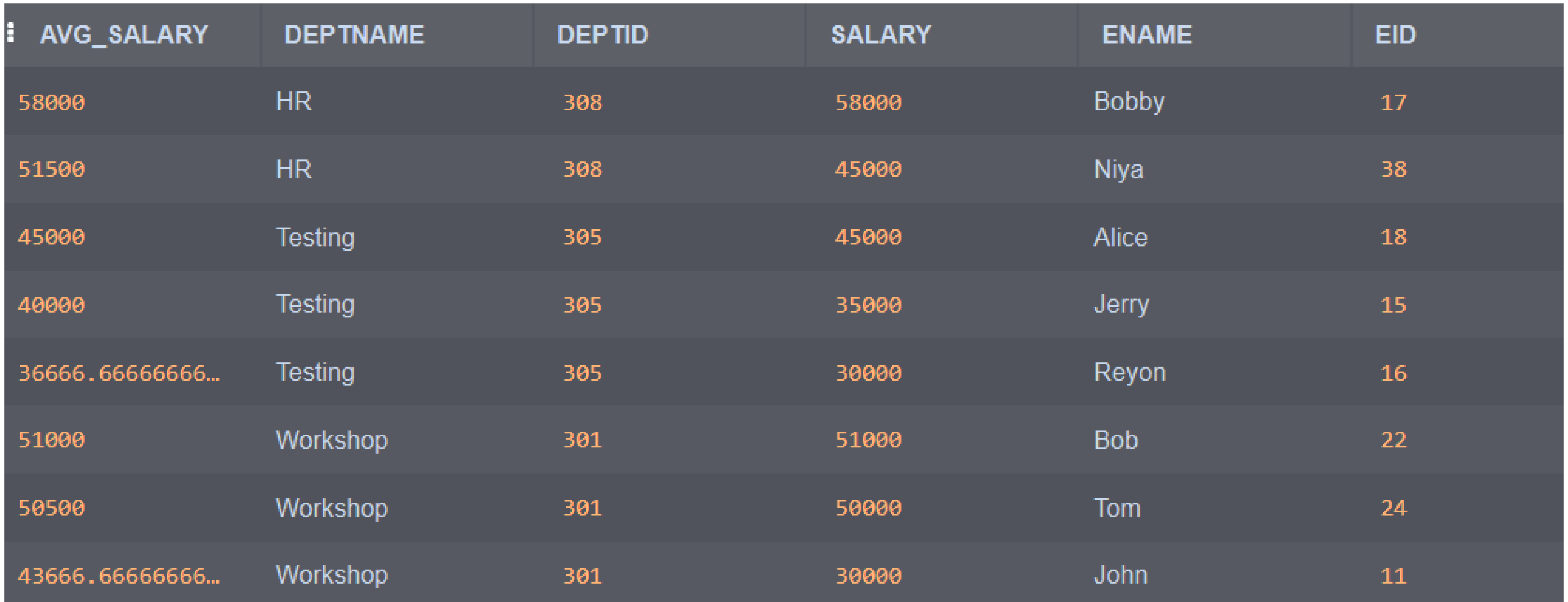
SELECT AVG(SALARY) OVER (PARTITION BY DEPTNAME) AS AVG_SALARY, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;



AVG_SALARY	DEPTNAME	DEPTID	SALARY	ENAME	EID
51500	HR	308	45000	Niya	38
51500	HR	308	58000	Bobby	17
36666.66666666…	Testing	305	35000	Jerry	15
36666.66666666…	Testing	305	45000	Alice	18
36666.66666666…	Testing	305	30000	Reyon	16
43666.66666666…	Workshop	301	30000	John	11
43666.66666666…	Workshop	301	50000	Tom	24
43666.66666666…	Workshop	301	51000	Bob	22

- Also, moving aggregate can be calculated by adding **ORDER BY** clause along with **PARTITION BY** in window function with **AVG().**

SELECT AVG(SALARY) OVER (PARTITION BY DEPTNAME ORDER BY SALARY DESC) AS AVG_SALARY, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;

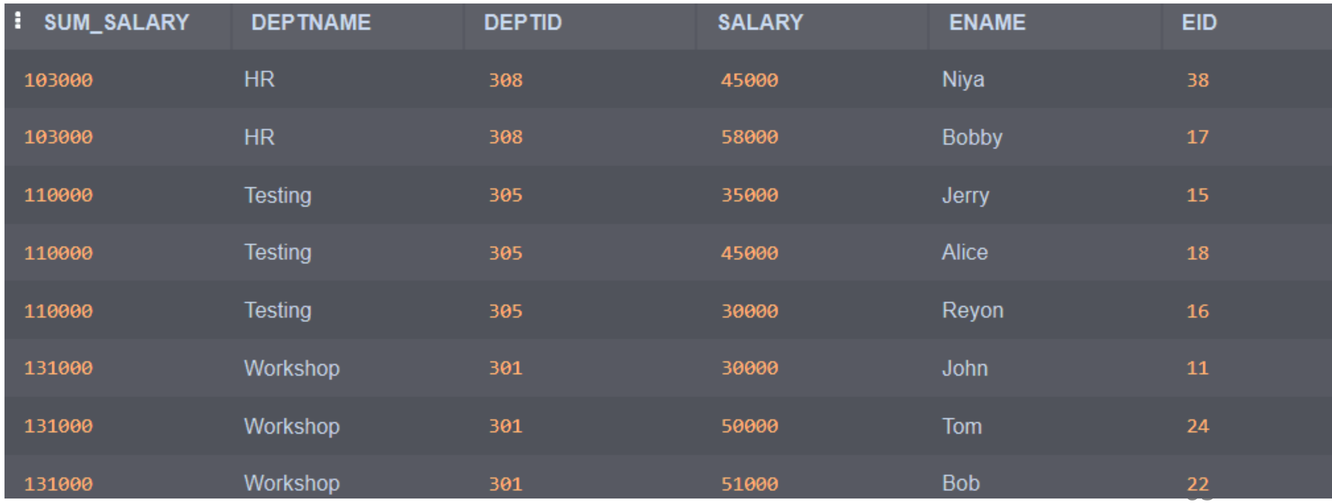


AVG_SALARY	DEPTNAME	DEPTID	SALARY	ENAME	EID
58000	HR	308	58000	Bobby	17
51500	HR	308	45000	Niya	38
45000	Testing	305	45000	Alice	18
40000	Testing	305	35000	Jerry	15
36666.66666666…	Testing	305	30000	Reyon	16
51000	Workshop	301	51000	Bob	22
50500	Workshop	301	50000	Tom	24
43666.66666666…	Workshop	301	30000	John	11

SUM()

- The **SUM()** window function returns the sum of input column or the expression across input values in each partition.
- For example, to calculate sum of salaries of workers in each department, we can write the query as follows:

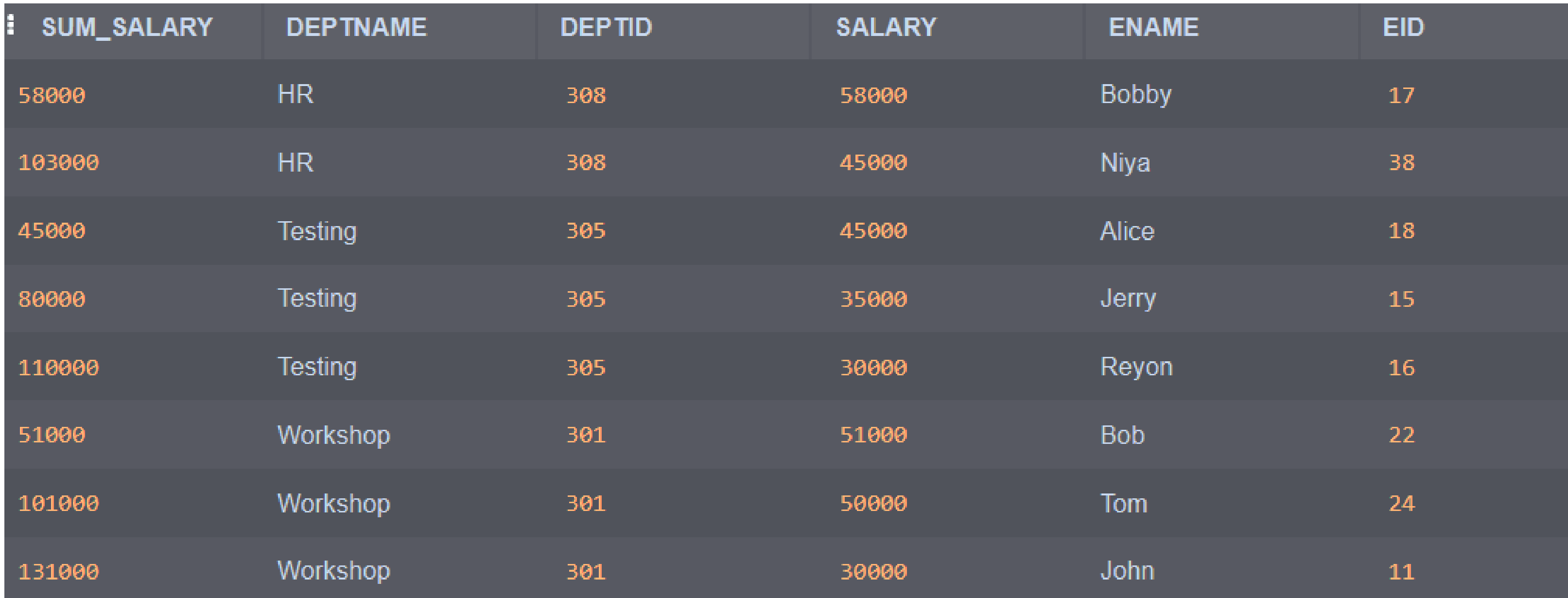
SELECT SUM(SALARY) OVER (PARTITION BY DEPTNAME) AS SUM_SALARY, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;



SUM_SALARY	DEPTNAME	DEPTID	SALARY	ENAME	EID
103000	HR	308	45000	Niya	38
103000	HR	308	58000	Bobby	17
110000	Testing	305	35000	Jerry	15
110000	Testing	305	45000	Alice	18
110000	Testing	305	30000	Reyon	16
131000	Workshop	301	30000	John	11
131000	Workshop	301	50000	Tom	24
131000	Workshop	301	51000	Bob	22

- If we want to calculate moving sum of salaries of each department, then we can add an ORDER BY clause in the above query

SELECT SUM(SALARY) OVER (PARTITION BY DEPTNAME ORDER BY SALARY DESC) AS SUM_SALARY, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;

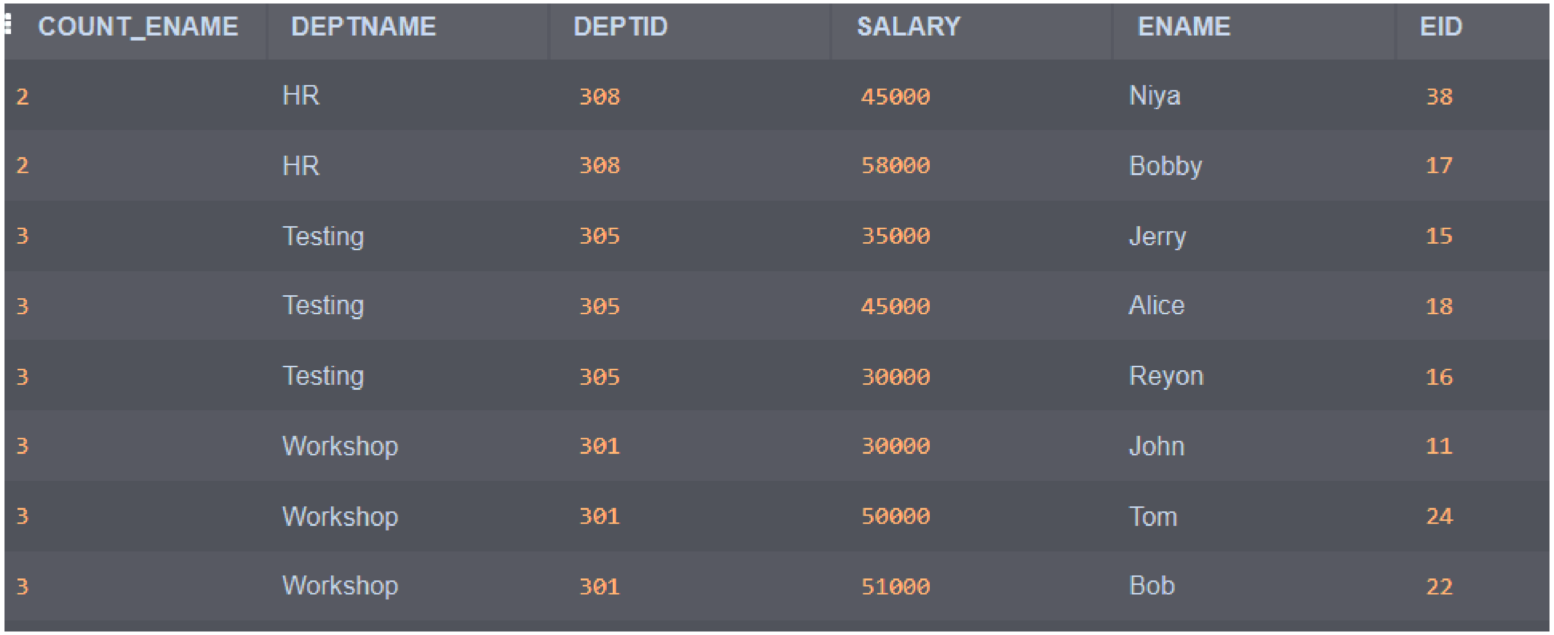


SUM_SALARY	DEPTNAME	DEPTID	SALARY	ENAME	EID
58000	HR	308	58000	Bobby	17
103000	HR	308	45000	Niya	38
45000	Testing	305	45000	Alice	18
80000	Testing	305	35000	Jerry	15
110000	Testing	305	30000	Reyon	16
51000	Workshop	301	51000	Bob	22
101000	Workshop	301	50000	Tom	24
131000	Workshop	301	30000	John	11

COUNT()

- The **COUNT()** window function counts the number of rows defined by the expression in partition. To count employees in each department, we can write the query as follows:

```
SELECT COUNT(ENAME) OVER (PARTITION BY DEPTNAME) AS COUNT_ENAME, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;
```

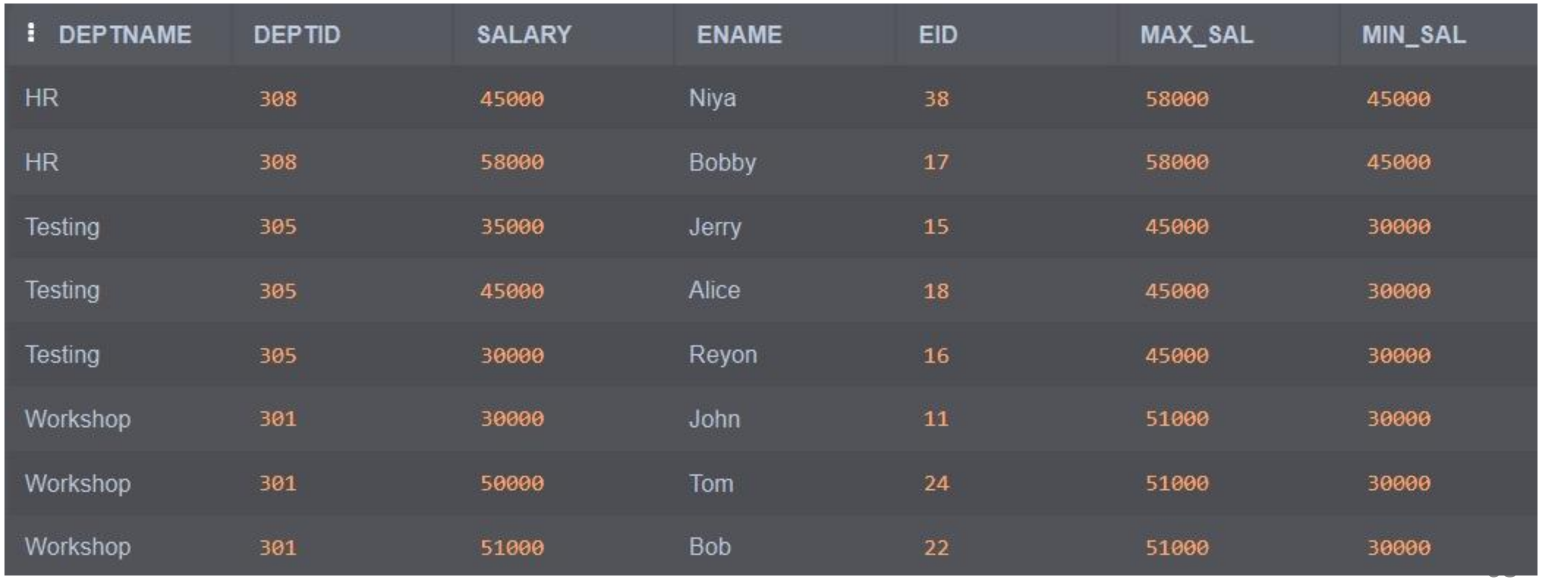


COUNT_ENAME	DEPTNAME	DEPTID	SALARY	ENAME	EID
2	HR	308	45000	Niya	38
2	HR	308	58000	Bobby	17
3	Testing	305	35000	Jerry	15
3	Testing	305	45000	Alice	18
3	Testing	305	30000	Reyon	16
3	Workshop	301	30000	John	11
3	Workshop	301	50000	Tom	24
3	Workshop	301	51000	Bob	22

MIN() and MAX()

- The aggregate window functions **MIN() and MAX()** return the minimum and maximum values of an expression within a specified window. The fol-lowing query will return the maximum and minimum salaries of workers in each department.

SELECT DEPTNAME, DEPTID, SALARY, ENAME, EID, MAX(SALARY) OVER (PARTITION BY DEPTNAME) AS MAX_SAL, MIN(SALARY) OVER (PARTITION BY DEPTNAME) AS MIN_SAL FROM workers;

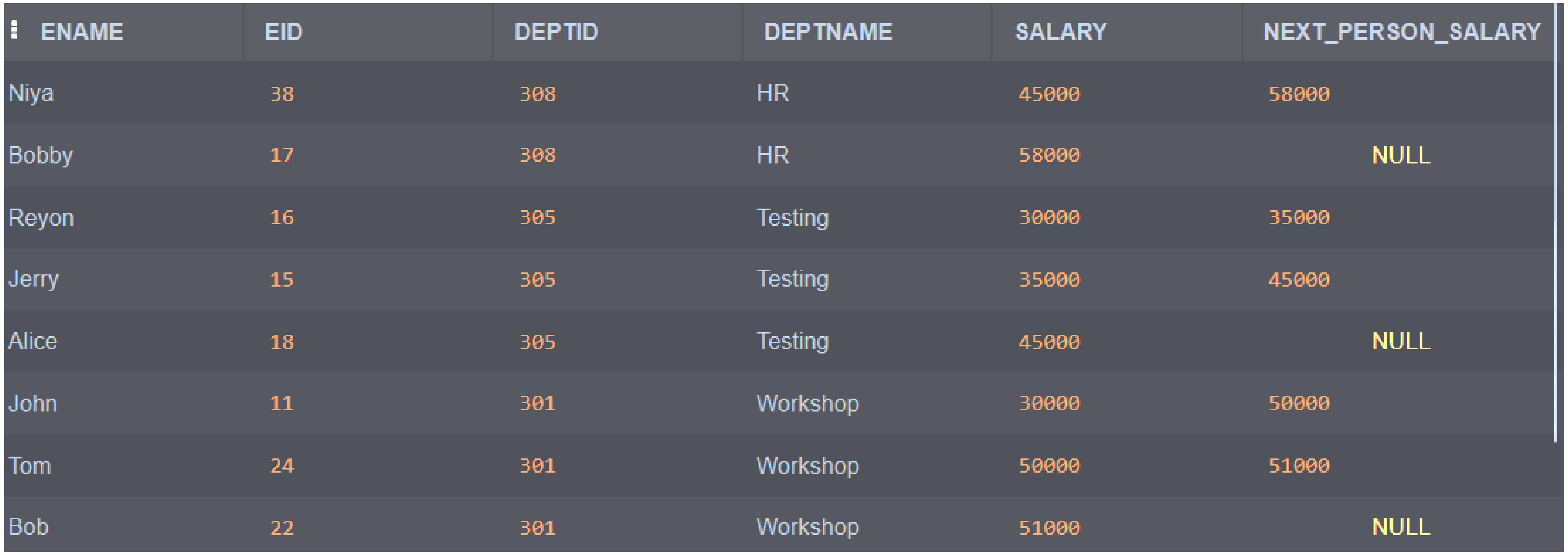


DEPTNAME	DEPTID	SALARY	ENAME	EID	MAX_SAL	MIN_SAL
HR	308	45000	Niya	38	58000	45000
HR	308	58000	Bobby	17	58000	45000
Testing	305	35000	Jerry	15	45000	30000
Testing	305	45000	Alice	18	45000	30000
Testing	305	30000	Reyon	16	45000	30000
Workshop	301	30000	John	11	51000	30000
Workshop	301	50000	Tom	24	51000	30000
Workshop	301	51000	Bob	22	51000	30000

LEAD()

- SQL **LEAD()** function has a capacity that gives admittance to a column at a **predefined actual counter balance which follows the current row**.
- For example, by utilizing the **LEAD()** function, from the current line, you can get information of the following line, or the second line that follows the current line, or the third line that follows the current line, etc.

- SELECT ENAME, EID, DEPTID, DEPTNAME, SALARY, LEAD(SALARY) OVER (PARTITION BY DEPTNAME ORDER BY SALARY) AS NEXT_PERSON_SALARY FROM workers;



ENAME	EID	DEPTID	DEPTNAME	SALARY	NEXT_PERSON_SALARY
Niya	38	308	HR	45000	58000
Bobby	17	308	HR	58000	NULL
Reyon	16	305	Testing	30000	35000
Jerry	15	305	Testing	35000	45000
Alice	18	305	Testing	45000	NULL
John	11	301	Workshop	30000	50000
Tom	24	301	Workshop	50000	51000
Bob	22	301	Workshop	51000	NULL

- The LEAD() function can also be very useful for calculating the difference between the value of the current row and the value of the following row. The following query finds the difference between the salaries of person in the same department.

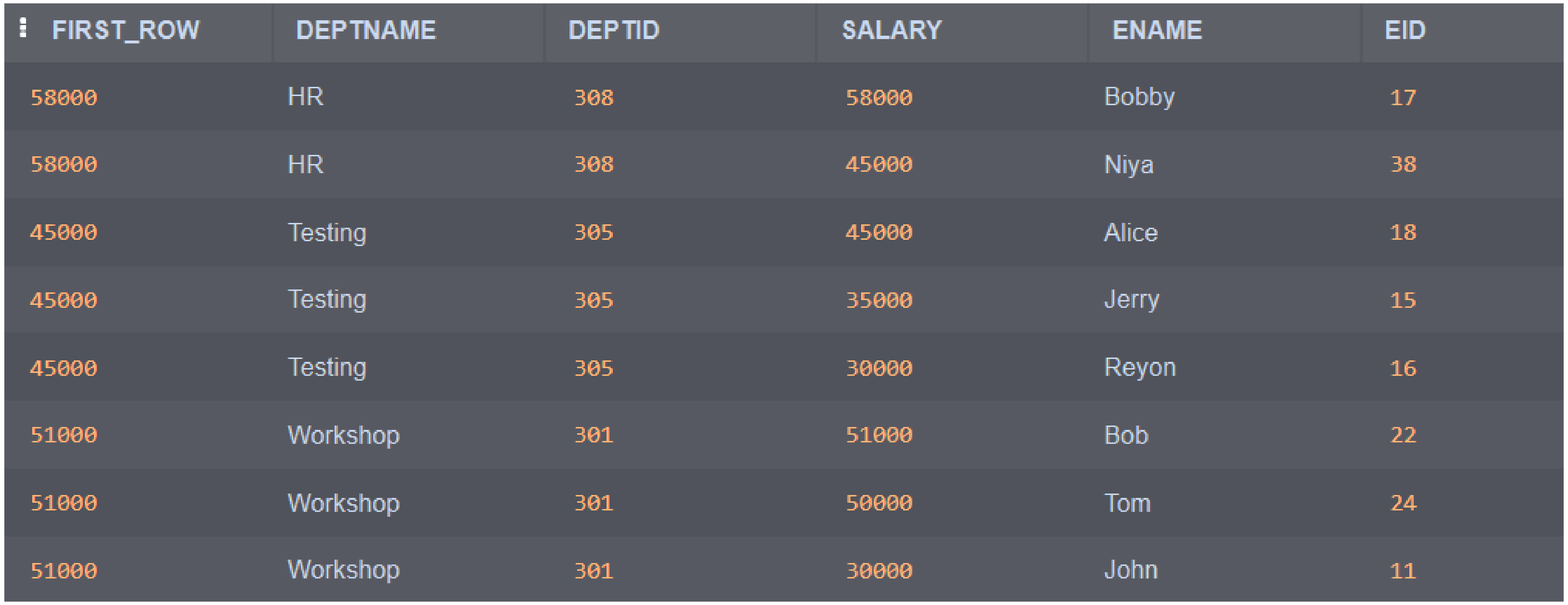
SELECT ENAME, EID, DEPTID, DEPTNAME, SALARY, LEAD(SALARY) OVER (PARTITION BY DEPTNAME ORDER BY SALARY)-SALARY AS SALARY_DIFFERENCE FROM workers;

ENAME	EID	DEPTID	DEPTNAME	SALARY	SALARY_DIFFERENCE
Niya	38	308	HR	45,000	13,000
Bobby	17	308	HR	58,000	NULL
Reyon	16	305	Testing	30,000	5,000
Jerry	15	305	Testing	35,000	10,000
Alice	18	305	Testing	45,000	NULL
John	11	301	Workshop	30,000	20,000
Tom	24	301	Workshop	50,000	1,000
Bob	22	301	Workshop	51,000	NULL

FIRST_VALUE()

- The SQL window function FIRST_VALUE() returns the first value in an ordered group of a result set or window frame. The following query returns the first salary value in each department ordered by salary.

```
SELECT FIRST_VALUE(SALARY) OVER (PARTITION BY DEPTNAME ORDER BY SALARY DESC) AS FIRST_ROW, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;
```

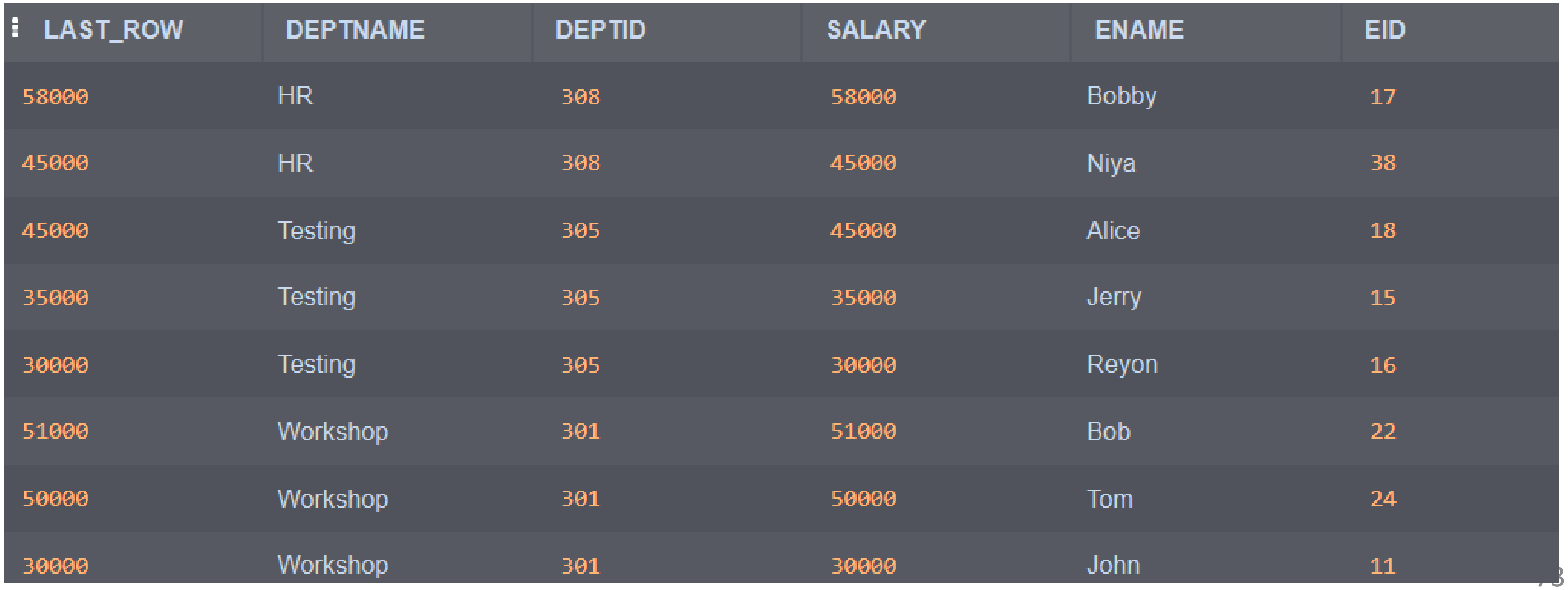


FIRST_ROW	DEPTNAME	DEPTID	SALARY	ENAME	EID
58000	HR	308	58000	Bobby	17
58000	HR	308	45000	Niya	38
45000	Testing	305	45000	Alice	18
45000	Testing	305	35000	Jerry	15
45000	Testing	305	30000	Reyon	16
51000	Workshop	301	51000	Bob	22
51000	Workshop	301	50000	Tom	24
51000	Workshop	301	30000	John	11

LAST_VALUE()

- The SQL window function LAST_VALUE() returns the last value in an ordered group of a result set. The following query returns the last salary value in each department ordered by salary.

SELECT LAST_VALUE(SALARY) OVER (PARTITION BY DEPTNAME ORDER BY SALARY DESC) AS LAST_ROW, DEPTNAME, DEPTID, SALARY, ENAME, EID FROM workers;



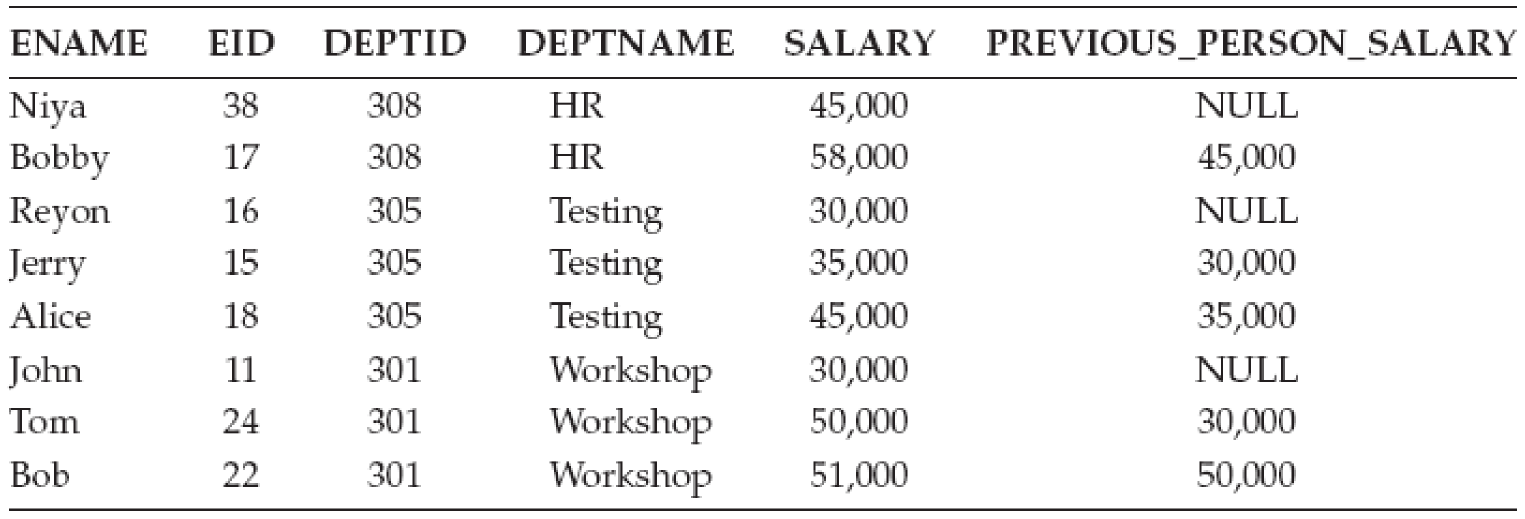
LAST_ROW	DEPTNAME	DEPTID	SALARY	ENAME	EID
58000	HR	308	58000	Bobby	17
45000	HR	308	45000	Niya	38
45000	Testing	305	45000	Alice	18
35000	Testing	305	35000	Jerry	15
30000	Testing	305	30000	Reyon	16
51000	Workshop	301	51000	Bob	22
50000	Workshop	301	50000	Tom	24
30000	Workshop	301	30000	John	11

LAG()

- We can use a SQL window function LAG() to access previous row’s data based on defined offset value.
- It works similar to a LEAD() function.
- In the SQL LEAD() function, we access the values of subsequent rows, but in LAG() function, we access previous row’s data.
- It is useful to compare the current row value from the previous row value.

```
SELECT ENAME, EID, DEPTID, DEPTNAME, SALARY, LAG(SALARY)
OVER (PARTITION BY DEPTNAME ORDER BY SALARY) AS
PREVIOUS_PERSON_SALARY FROM workers;
```

- The above query **finds the salary of the previous person** in each department based on logically sorted salary value.
- As **no previous row** is available for the first row in each department, it returns a **NULL** value.



ENAME	EID	DEPTID	DEPTNAME	SALARY	PREVIOUS_PERSON_SALARY
Niya	38	308	HR	45,000	NULL
Bobby	17	308	HR	58,000	45,000
Reyon	16	305	Testing	30,000	NULL
Jerry	15	305	Testing	35,000	30,000
Alice	18	305	Testing	45,000	35,000
John	11	301	Workshop	30,000	NULL
Tom	24	301	Workshop	50,000	30,000
Bob	22	301	Workshop	51,000	50,000

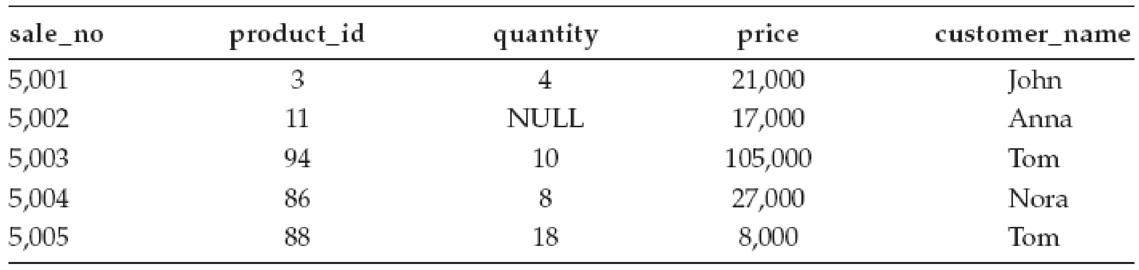
- Summary – Window function and Ordered data
  - R ANK()
  - PERCENT_RANK()
  - DENSE_RANK()
  - NTILE()
  - CUME_DIST()
  - ROW_NUMBER()
  - AVG()
  - SUM()
  - COUNT()
  - MIN()   MAX()
  - LEAD()
  - FIRST_VALUE()
  - LAST_VALUE()
  - LAG()

Preparing Data for Analytics Tool

- One of the **primary steps** performed for data science is the cleaning of the dataset
- The valuable information or patterns extract from dataset are the same class as the data itself, so maximum of the time spent by a data scientist or analyst includes **preparing datasets for use in analysis**.
- SQL can help to speed up this step.
- Various SQL queries can be used to **clean, update, and filter data, by eliminating redundant and unwanted records**. This can be done with the different SQL clauses like **CASE WHEN, COALESCE, NULLIF, LEAST/GREATEST, Casting, and DISTINCT**.

- Demo

```
--sales
CREATE TABLE sales (
   sale_no INT PRIMARY KEY,
   product_id INT,
   quantity INT,
   price INT,
   customer_name VARCHAR(50)
);
--Insert
INSERT INTO sales (sale_no, product_id, quantity, price, customer_name) VALUES
(5001, 3, 4, 21000, 'John'),
(5002, 11, NULL, 17000, 'Anna'),
(5003, 94, 10, 105000, 'Tom'),
(5004, 86, 8, 27000, 'Nora'),
(5005, 88, 18, 8000, 'Tom’);

SELECT * FROM SALES;
```



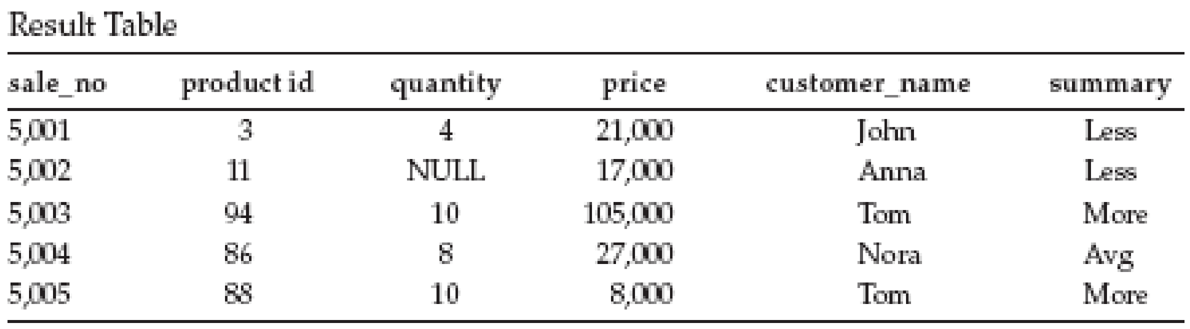
sale_no	product_id	quantity	price	customer_name
5,001	3	4	21,000	John
5,002	11	NULL	17,000	Anna
5,003	94	10	105,000	Tom
5,004	86	8	27,000	Nora
5,005	88	18	8,000	Tom

CASE

- The **CASE** statement goes through **various conditions** specified with **WHEN** clause and **returns a value** when the first condition is met.
- It **works like nested IF-THEN-ELSE** statement.
- Once a condition is true, it will return the value specified after THEN. Value in the ELSE clause is returned, if no conditions are true. It returns NULL when no conditions are true, and no ELSE part is specified in the query.

- Suppose we fetch all data of the sales table and want to **add an extra column** that labels as **summary** which categorizes sales into More, Less, and Avg, this table can be created using a CASE statement as follows:

```
SELECT *,
    CASE     WHEN quantity >= 10 THEN 'More'
             WHEN quantity >= 6 THEN 'Avg'
             ELSE 'Less'
    END AS summary
FROM sales;
```

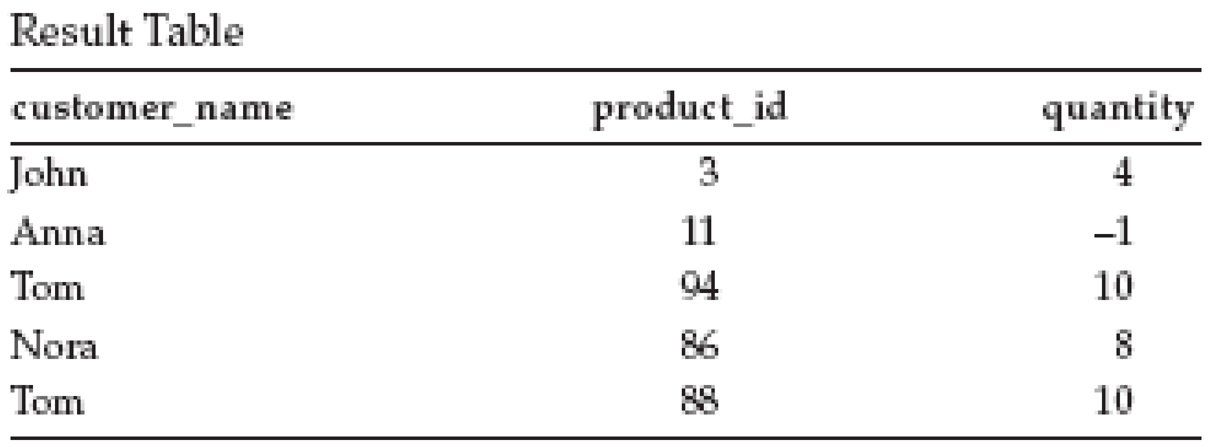

Result Table

sale_no	product id	quantity	price	customer_name	summary
5,001	3	4	21,000	John	Less
5,002	11	NULL	17,000	Anna	Less
5,003	94	10	105,000	Tom	More
5,004	86	8	27,000	Nora	Avg
5,005	88	10	8,000	Tom	More

COALESCE

- Some records of **database may consist of NULL values**, but **while applying statistics** to these datasets, you may **need to replace these NULL values with some other data**.
- This can be **done effectively by the COALESCE** function.
- The first parameter to this function is a column that may consist of NULL, and the second represents value that replaces NULL. E.g. COALESCE(quantity, -1)
- It replaces all NULL values specified in column by the second default value given in the function.
- The following example replaces NULL by −1 in the quantity column.

```
SELECT customer_name, product_id,
COALESCE(quantity, -1) AS quantity
FROM sales;
```



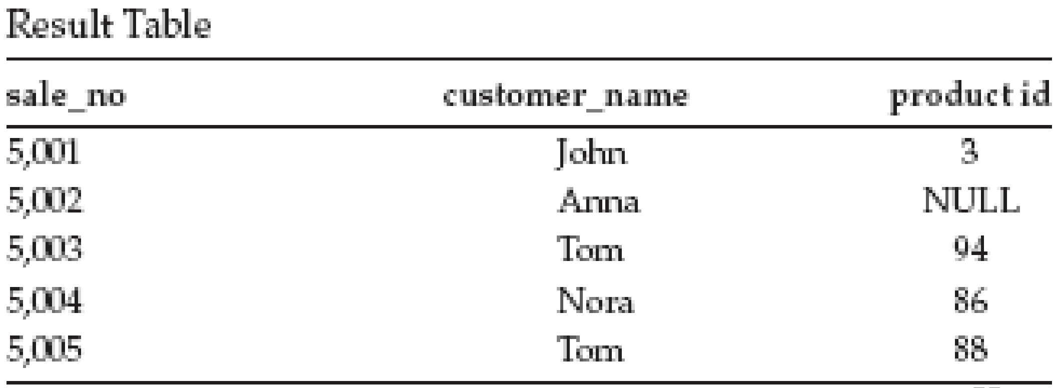
Result Table

customer_name	product_id	quantity
John	3	4
Anna	11	–1
Tom	94	10
Nora	86	8
Tom	88	10

NULLIF

- NULLIF function takes two parameters and will return NULL if the first parameter value equals the second value else returns the first parameter.
- As an example, imagine that we want to replace product_id value 11 by NULL. This could be done with the following query:

```
SELECT sale_no, customer_name,
NULLIF(product_id, 11) AS product_id
FROM sales;
```



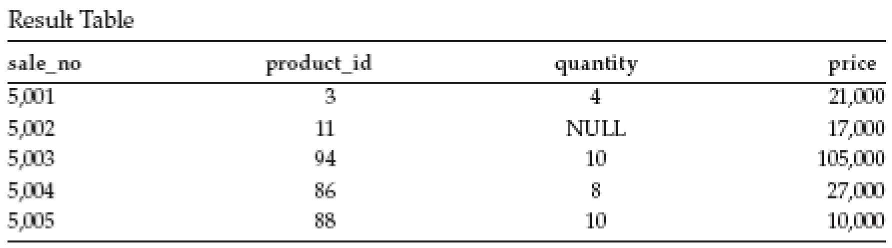
Result Table

sale_no	customer_name	product id
5,001	John	3
5,002	Anna	NULL
5,003	Tom	94
5,004	Nora	86
5,005	Tom	88

LEAST/GREATEST

- The LEAST and GREATEST are frequently used functions for data cleaning.
- These functions return the least and greatest values from the given set of elements, respectively. These functions are useful to replace value in list, especially if it is too high or low.
- For example, minimum price needs to be 10,000 in the above table. This can be done by the following query. Price 8,000 is replaced by value 10,000 in the last row, as 8,000 is less than 10,000, and it replaces it by max value among these two.

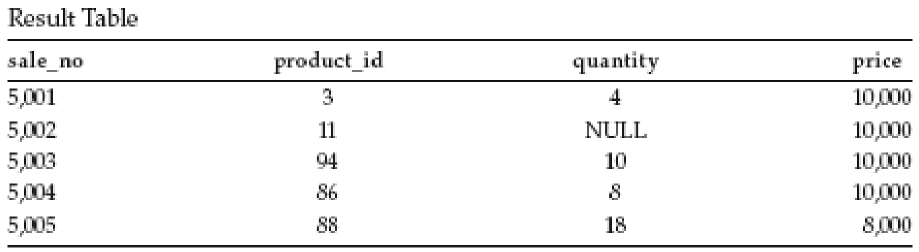
```
SELECT sale_no, product_id, quantity,
GREATEST(10000, price) as price
FROM sales;
```



Result Table

sale_no	product_id	quantity	price
5,001	3	4	21,000
5,002	11	NULL	17,000
5,003	94	10	105,000
5,004	86	8	27,000
5,005	88	10	10,000

```
SELECT sale_no, product_id, quantity,
LEAST(10000, price) as price
FROM sales;
```



Result Table

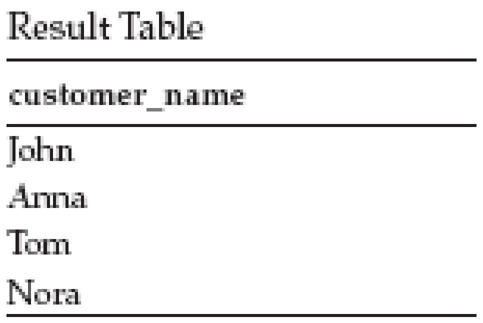
sale_no	product_id	quantity	price
5,001	3	4	10,000
5,002	11	NULL	10,000
5,003	94	10	10,000
5,004	86	8	10,000
5,005	88	18	8,000

DISTINCT

- The DISTINCT keyword returns only distinct values in the specified column value sets.
- For example, to extract all the unique names in the sales table, you could write the following query:

SELECT DISTINCT customer_name FROM sales;

- The above query gives the following result: It removed duplicate names from the customer_name column

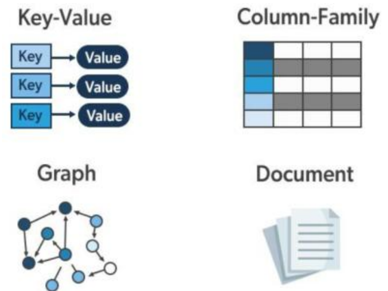


Result Table

customer_name
John
Anna
Tom
Nora

Advanced NoSQL for Data Science

- NoSQL is a database design that can **accommodate various data models**, including key-value, document, columnar, and graph formats.
- NoSQL, which means "**not only SQL**", is an alternative to relational databases in which **data is stored in tables and has a fixed data schema**.
- NoSQL databases are very useful for working with large distributed data.
- The NoSQL databases are built in the **early 2000s to deal with large-scale database clustering in web and cloud applications**.



- NoSQL has a **flexible schema**, unlike the traditional relational database model.
- All **rows can have different structures or attributes**.
- NoSQL databases are found to be very useful for **handling really big data tasks** because it follows the **B**asically **A**vailable, **S**oft State, **E**ventual Consistency (**BASE**) approach **instead of** Atomicity, Consistency, Isolation, and Durability – commonly known as **ACID properties**

- Two major drawbacks of SQL are **rigidity** when adding columns and attributes to tables and **slow performance** when many tables need to be joined and when tables store a large amount of data.
- NoSQL databases tried to overcome these two biggest drawbacks of relational databases.
- **NoSQL** offers a **more flexible, schema-free solution** that can work with **unstructured data.**

Why NoSQL

- NoSQL **supports unstructured data or semi-structured data**.
- In many applications, an attribute usually needs to be added on the fly, for specific rows, but not every row, and may be of different types than attributes in the rows.

Features:

- It is **not using the relational model** to store data.
- NoSQL **running well on clusters**.
- It is mostly **open-source**.
- NoSQL is capable **to handle a large amount** of social media data.
- NoSQL is **schema-less**.

1. Document Databases for Data Science

- Document-based NoSQL databases store the data in the **JSON object format**.
- Each document has **key-value** pairs like structures.
- The document-based NoSQL databases are simple for engineers as they map items as a JSON object.
- JSON is a very common data format truly adaptable by **web developers and permits us to change the structure whenever required**.
- Some examples of document based NoSQL databases are **CouchDB, MongoDB, OrientDB, and BaseX**.

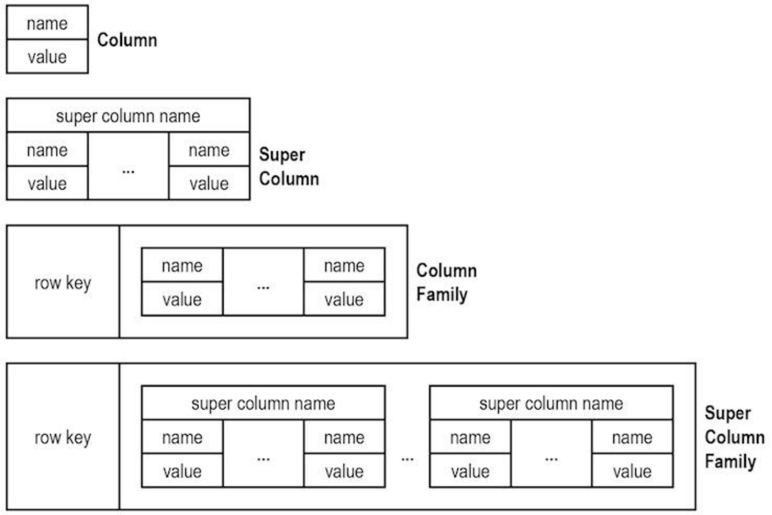
JSON document format:

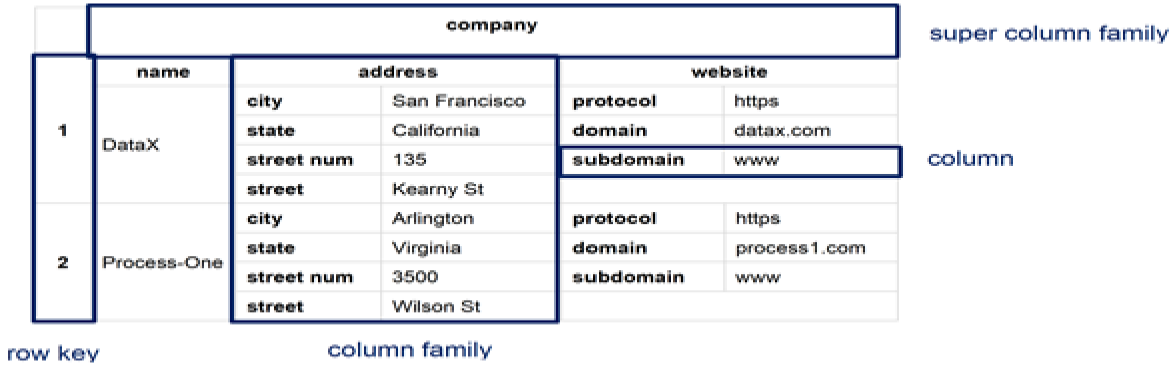
```
{
	"_id": 1,
	"name" : { "first" : "John", "last" : "Backus" },
	"contribs" : [ "Fortran", "ALGOL", "Form", "FP" ],
	"awards" : [
	{
		"award" :"Dowell Award",
		"year" : 1988,
		"by" :"Computer Society"
	},
	{
		"award" :"First Prize",
		"year" : 1993,
		"by" : "National Academy of Engineering"
	}
	]
}
```

2. Wide-Column Databases for Data Science

- Similar to any relational database, this wide-column database stores the data in records, but it can also store very large numbers of dynamic columns.
- It groups the dynamically added columns into column families.
- Instead of having multiple tables like relational databases, we have multiple column families in wide-column databases.
- Examples of wide-column types of databases are Cassandra and HBase

Pattern for wide-column database.



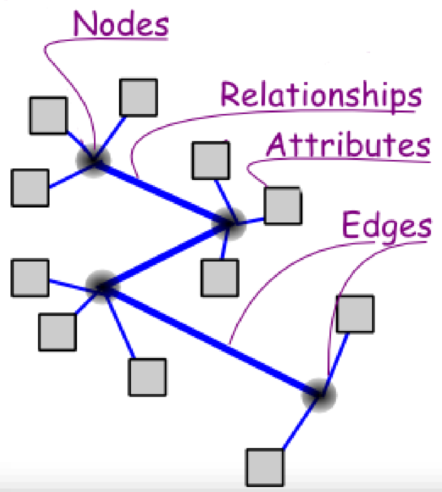


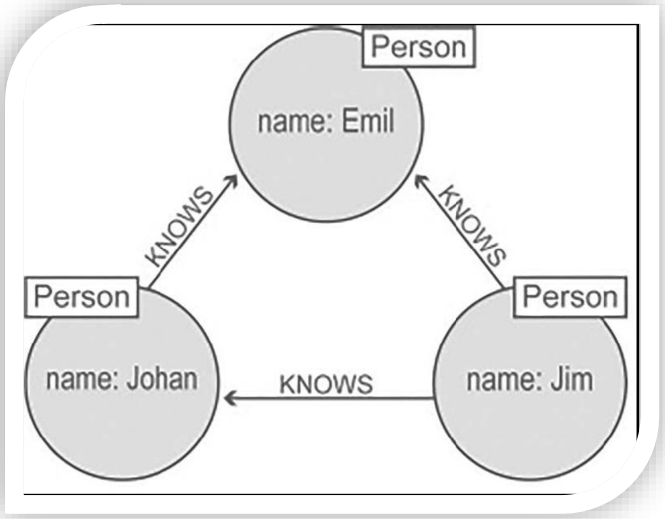
super column family
column
row key
column family

	company				
	name	address		website	
1	DataX	city	San Francisco	protocol	https
		state	California	domain	datax.com
		street num	135	subdomain	www
		street	Kearny St		
2	Process-One	city	Arlington	protocol	https
		state	Virginia	domain	process1.com
		street num	3500	subdomain	www
		street	Wilson St		

3. Graph Databases for Data Science

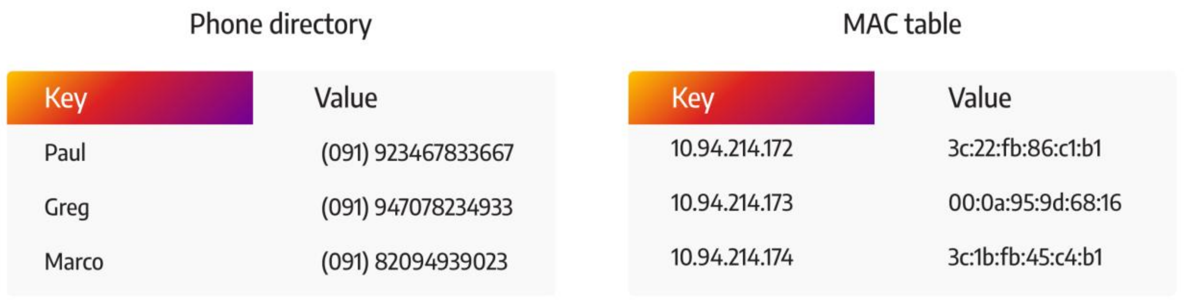
- Graph database stores the data in the form of **nodes and edges**.
- The **node stores information** about the **main entities** like **people, places, and products**, and the **edge stores the relationships between them**.
- Graph database is very useful to find out the **pattern or relationship among data** like a **social network and recommendation engines**.
- Examples of graph databases are Neo4j and Amazon Neptune





4. Key-Value

- Data is stored as key-value pairs, making retrieval extremely fast.
- Optimized for caching and session storage.
- Examples: Redis, Memcached, Amazon DynamoDB
- Perfect for applications requiring session management, real-time data caching, and leaderboards.

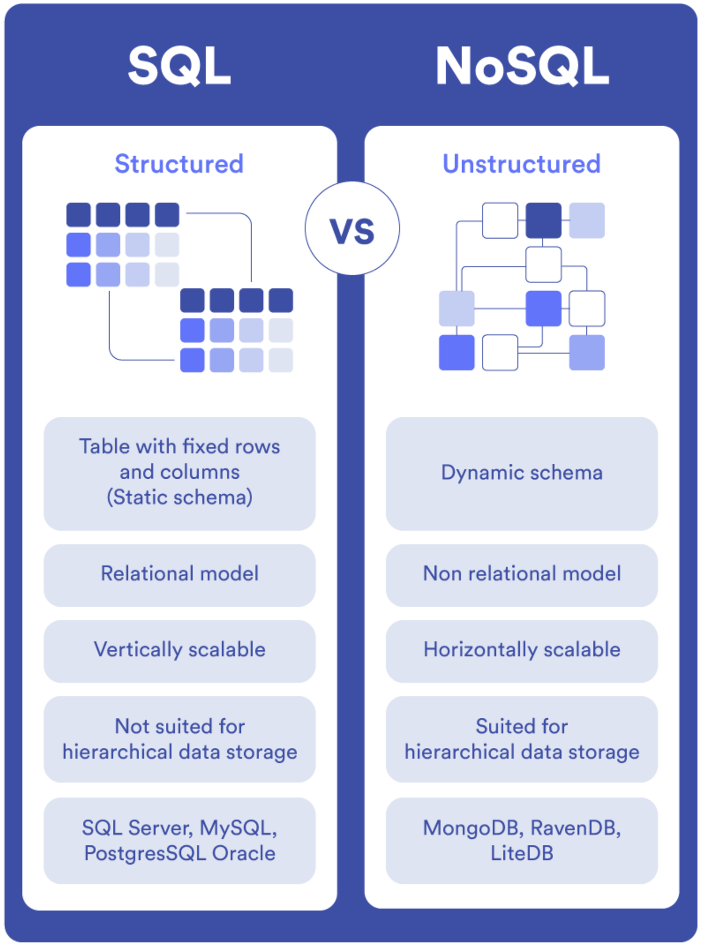


Phone directory

Key	Value
Paul	(091) 923467833667
Greg	(091) 947078234933
Marco	(091) 82094939023

MAC table

Key	Value
10.94.214.172	3c:22:fb:86:c1:b1
10.94.214.173	00:0a:95:9d:68:16
10.94.214.174	3c:1b:fb:45:c4:b1



SQL	NoSQL
Structured	Unstructured
Table with fixed rows and columns (Static schema)	Dynamic schema
Relational model	Non relational model
Vertically scalable	Horizontally scalable
Not suited for hierarchical data storage	Suited for hierarchical data storage
SQL Server, MySQL, PostgresSQL Oracle	MongoDB, RavenDB, LiteDB

STUDENT_ID	FIRST_NAME	LAST_NAME	COURSE	MOBILE_NUMBER	DAD_NUMBER	MUM_NUMBER	GUARDIAN_NUMBER	CGPA
1	Mary	Hopkins	CSE	789123456	456789231	135792468	775544331	7.8
2	Nicholas	Wilde	CSE	321456676	(null)	234567123	(null)	5.5
3	Judy	Hopps	CSE	45361237	(null)	(null)	995566778	3.7
4	Emitt	Otterton	CSE	654321789	908760453	456789234	556790234	8.9
5	Benjamin	Clawhauser	IT	7778067546	(null)	234561789	(null)	2.1
6	Dawn	Bellwether	IT	8977666478	(null)	(null)	667889440	6.5

Consider the above dataset:

1. Display ‘low’ for students with cgpa less than 4, ‘medium’ for students with cgpa between 4 and 7.5 and ‘high’ for other students.
2. Display the first available emergency contact for the students from dad, mom or guardian numbers.
3. List the different courses available for the students
4. Display ‘Yes’ along with the student name if the student belongs to CSE ‘No otherwise.

Solution

1. Display ‘low’ for students with cgpa less than 4, ‘medium’ for students with cgpa between 4 and 7.5 and ‘high’ for other students.

```
SELECT   STUDENT_ID,  FIRST_NAME,  LAST_NAME,  CGPA,
  CASE          WHEN CGPA < 4 THEN 'Low'
                WHEN CGPA >= 4 AND CGPA <= 7.5 THEN 'Medium'
                ELSE 'High’
END AS CGPA_Category FROM students;
```

2. Display the first available emergency contact for the students from dad, mom or guardian numbers.

```
SELECT   STUDENT_ID,  FIRST_NAME,  LAST_NAME,
COALESCE(DAD_NUMBER,  MUM_NUMBER,  GUARDIAN_NUMBER)  AS  Emergency_Contact
FROM students;
```

3. List the different courses available for the students

```
SELECT DISTINCT COURSE FROM students;
```

4. Display ‘Yes’ along with the student name if the student belongs to CSE ‘No otherwise.

```
SELECT   FIRST_NAME,  LAST_NAME,  COURSE,
CASE     WHEN COURSE = 'CSE' THEN 'Yes'
         ELSE 'No’
END AS Is_CSE_Student
FROM students;
```

Module 2

Structured Query Language (SQL): Basic Statistics, Data Munging, Filtering, Joins, Aggregation, Window Functions, Ordered Data, preparing No-SQL: Document Databases, Wide-column Databases and Graphical Databases.