SQL -Structured Query Languages

-storing, manipulating and retrieving data

-Basically info from multiple **related** data bases to give out a specific query.

-A Relational Database Management Systems (**RDBMS**): MySQL, MS, Access, Oracle, postgres ect. -Dif systems can have slightly different language.

-Data base- collection of table

-Tables – rows(Record)- observations of the samples; and columns(Field) characteristics about observations

Data Types-

*-Columns must be same data type*

-String – combo or letters (input it with quotes or will not work)

-Integer- whole number

-Floating point- decimal numbers

- Boolean- True and False

Writing SQL statements (query)

* Not case sensitive. Only MySQL- type table names as they exist in databases
* Keywords cant be split across lines or abbreviated.
* Keywords usually in uppercase – looks nice.
* Clauses usually in sep lines for readability and ease of editing
* Tabs and indents to enhance readably
* Semicolon to end query

Ex:

SELECT \*

FROM (table) Jobs

WHERE Min\_salary > 1000

ORDER BY Job\_title;

Keywords and Clauses- used to find specific data

**DISTINCT**- Brings only distinct values in table. (no dups)

**WHERE**- Filters data from Columns

* Can be combined with **AND**(All need to be TRUE)**, OR**( One needs to be TRUE)**, NOT**(NOT TRUE) operators.
* If using 2 different operators in one query, use ( ) to separate the statements.

**ORDER BY**

* **ASC**- Ascending order. 1,2,3
* **DESC**- Descending order. 3,2,1

TIP: If you type out multiple SELECT values, you can use numbers later on for the position.

SELECT Table\_ID, Table\_date, Table\_item

FROM random

ORDER BY 2 Asc;

* “2” will be Table\_date

**IS NULL** -field with no value

-Use **IS NULL** in **WHERE** clause; **IS NOT NULL**- gives you everything with a value

SELECT Min\_salary

FROM (table) Jobs

WHERE Min\_salary **IS NULL**

**ROWNUM**- in oracle

**LIMIT-** In other SQL systems

-Both specify the number of records to return

SELECT \*

FROM countries

WHERE ROWNUM < 6; *OR LIMIT 6*

**MIN** -returns the smallest value of the selected field

SELECT MIN(salary)

FROM employees;

**MAX-** returns the largest value of the selected field.

SELECT MAX(salary)

FROM employees;

**COUNT**- returns the number of records that matches a specified criteria.

SELECT COUNT(\*)

FROM employees;

-For unique, use COUNT(DISTINCT)

SELECT COUNT(DISTINCT first\_name)

FROM employees;

**AVG**- returns the average value of a numeric field (number)

SELECT AVG(salary)

FROM employees;

**SUM**- total sum of a numeric field (number)- must specify what you’re summing.

SELECT SUM(salary)

FROM employees;

Ex: Report:

* **Calculate number of employees, minimum, maximum and average salary from employees table**

SELECT COUNT(employee\_id), MAX(salary), MIN(salary), AVG(salary)

FROM employees;

**ROUND-** returns a numeric value, rounded to the specified length or precision.

SELECT ROUND(AVG(salary), 2)

FROM employees;

-“2” being the amount of decimal points

Arithmetic Operations

* **Calculate the sum of maximum and minimum salary in employees table.**

SELECT MAX(salary)+MIN(salary)

FROM employees;

* **Calculate the average salary in employees table**

SELECT SUM(salary)/COUNT(employee\_id)

FROM employees;

* **Calculate 10 times average salary of employees in employee table**

SELECT AVG(salary)\*10

FROM employees;

**Like -**Used with **WHERE**

**-** search for a specified pattern/string/number in a field.

- “%” - rep 0,1, multi numbers

- “\_”- rep 1 character

|  |  |
| --- | --- |
| WHERE first\_name LIKE 'a%' | Finds any names that start with "a" |
| WHERE first\_name LIKE '%a' | Finds any names that end with "a" |
| WHERE first\_name LIKE '%ar%' | Finds any names that have "a" in any position |
| WHERE first\_name LIKE '\_a%' | Finds any names that have "a" in the second position |
| WHERE first\_name LIKE 'a\_%' | Finds any names that start with "a" and are at least 2 characters in length |
| WHERE first\_name LIKE 'a\_\_%' | Finds any names that start with "a" and are at least 3 characters in length |
| WHERE first\_name LIKE 'a%b' | Finds any names that start with "a" and ends with "b" |

**Select all employee records whose first name start with ‘S’.**

SELECT \*

FROM employees

WHERE first\_name LIKE 'S%';

* **Select all employee records whose first name contains with ‘er’.**

SELECT \*

FROM employees

WHERE first\_name LIKE '%er%';

**IN-** allows you to specify multiple values in **WHERE** clause. It  is a shorthand for multiple **OR** conditions.

* Can use **NOT IN**
* **Find all the records where the employees first names are ‘Peter’ or ‘Steven’.**

SELECT \*

FROM employees

WHERE first\_name IN ('Peter', 'Steven');

**BETWEEN**- selects values within a specified range. -Inclusive of all data type. **NOT BETWEEN-** for op

* **Find all the employee records where employees salaries are between 11000 and 13000**

SELECT \*

FROM employees

WHERE salary BETWEEN 11000 AND 13000;

Aliases -to make column names more readable (only for fields)

**-AS**

-Only exists for the duration of the query

SELECT country\_id AS cid, country\_name, region\_id

FROM countries

SELECT C.\*

FROM countries  C

SELECT C.country\_id AS cid, C.country\_name, C.region\_id

FROM countries  C

Comments- Single line comments start with –

* Multi line starts with /\* and ends with \*/

SELECT country\_id--, country\_name, region\_id

FROM countries;

SELECT country\_id

/\*

This is my second comment

And I love it.

\*/

FROM countries;

Concatenation – It concatenates columns or character strings to other columns. It is represented by two vertical bars ( **||** ) - *(Shift \\)- above “enter”*.

SELECT first\_name || last\_name AS full\_name

FROM employees;

-**put ‘ ‘ for space b/w names**

SELECT first\_name || '  ' || last\_name AS full\_name

FROM employees;

SELECT first\_name || '  ' || last\_name AS "Full Name"

FROM employees;

**JOIN**: clause is used to combine records from two or more tables in a database

Types:

* (INNER) JOIN: Returns records that have matching values in both tables
* LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
* RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
* FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table

1. **INNER JOIN** returns records that have matching values in both tables.

SELECT table1.column, table2.column

FROM table1

INNER JOIN table2

ON table1.column\_name = table2.column\_name;

Ex:

SELECT employees.first\_name, employees.employee\_id, job\_history.start\_date, job\_history.end\_date

FROM employees

INNER JOIN job\_history

ON employees.employee\_id = job\_history.employee\_id;

1. **LEFT JOIN** returns all records from the left table, and the matched records from the right table.

SELECT table1.column, table2.column

FROM table1

LEFT JOIN table2

ON table1.column\_name = table2.column\_name;

Ex:

SELECT E.first\_name, E.employee\_id, J.start\_date, J.end\_date

FROM employees  E

LEFT JOIN job\_history  J

ON E.employee\_id = J.employee\_id;

1. **RIGHT JOIN** - returns all records from the right table, and the matched records from the left table

SELECT table1.column, table2.column

FROM table1

RIGHT JOIN table2

ON table1.column\_name = table2.column\_name;

Ex:

SELECT d.manager\_id, d.department\_name, D.manager\_id, E.first\_name

FROM departments  D

RIGHT JOIN employees  E

ON D.manager\_id = E.manager\_id;

1. **FULL OUTER JOIN**-returns all records when there is a match in either left or right table.

SELECT table1.column, table2.column

FROM table1

FULL OUTER JOIN table2

ON table1.column\_name = table2.column\_name;

Ex:

SELECT d.department\_id, d.department\_name, D.manager\_id, E.first\_name

FROM departments  D

FULL OUTER JOIN employees  E

ON D.department\_id = E.department\_id;

1. **SELF JOIN**- join where a table is joined with itself.

* To join a table itself means that each row of the table is combined with itself and with every other row of the table.
* The self join can be viewed as a join of two copies of the same table

SELECT A.column, B.column

FROM table1 A, table1 B

WHERE A.common\_filed = A.common\_field;

Ex:

* **Find who works for who in employees table**

SELECT E1.first\_name ||' works for '|| E2.first\_name "Employees and Their Managers"

FROM employees E1, employees E2

WHERE E1.manager\_id = E2.employee\_id;

**UNION-** operator is used to combine the results of two or more queries . It returns only distinct records.

* Every query must have the same number of columns
* The columns must have similar data types
* The columns must be in the same order

SELECT column(s)

FROM table1

UNION

SELECT column(s)

FROM table2

Ex: **Combine distinct employees information from two select statement.**

SELECT \*

FROM employees

WHERE employee\_id > 200

UNION

SELECT \*

FROM employees

WHERE employee\_id < 220

* **UNION ALL**- to get all duplicates

Ex: **Combine all employees information from two select statement.**

SELECT \*

FROM employees

WHERE employee\_id > 200

UNION ALL

SELECT \*

FROM employees

WHERE employee\_id < 220

**INTERSECT**- compares the result of two queries  and returns the distinct rows that are output by both queries.

* Every query must have the same number of columns
* The columns must have similar data types
* The columns must be in the same order

SELECT column(s)

FROM table1

INTERSECT

SELECT column(s)

FROM table2

* Ex: **Return common employee records from two queries.**

SELECT \*

FROM employees

WHERE employee\_id > 200

INTERSECT

SELECT \*

FROM employees

WHERE employee\_id < 220

**MINUS-**operator is used to combine two queries and returns rows from the first query that are not returned by the second query. The MINUS operator is the same as EXCEPT operator in other database such as MySQL.

* Every query must have the same number of columns
* The columns must have similar data types
* The columns must be in the same order

SELECT column(s)

FROM table1

EXCEPT

SELECT column(s)

FROM table2

* Ex: **Return employee records which returned by first query but not available in second query.**

SELECT \*

FROM employees

WHERE employee\_id > 200

MINUS

SELECT \*

FROM employees

WHERE employee\_id < 220

Grouping Data

**GROUP BY**- sed to collect data across multiple records and group the results by one or more columns.

* The **GROUP BY** clause returns one row per group.
* The **GROUP BY** clause is often used with aggregate functions such as AVG(). COUNT(), MAX(), MIN() and SUM(). In this case, the aggregate function returns the summary information per group.
* To sort the groups, you add the **ORDER BY** clause after the **GROUP BY** clause.

Ex:

**This GROUP BY example uses the MIN function to return the name of each job id and the minimum salary for the job.**

SELECT job\_id, MIN(salary) AS "Lowest salary"

FROM employees

GROUP BY job\_id;

* **Retrieve the name of each job id and the minimum salary for the job.**

SELECT department\_id

, MIN(salary) AS "Lowest salary"

, MAX(salary) AS "Highest salary"

, AVG(salary) AS "Average salary"

, SUM(salary) AS "Total Salary"

, COUNT(employee\_id) AS "Employee Count"

FROM employees

GROUP BY department\_id;

* **Find the headcount of each department.**

SELECT department\_id, COUNT(employee\_id) AS headcount

FROM employees

GROUP BY department\_id;

GROUP BY and ORDER BY with Joins

* **Retrieve department id, department name and employee count of that department**

SELECT E.department\_id

, D.department\_name

, COUNT(E.employee\_id) AS headcount

FROM employees E

INNER JOIN departments D ON D.department\_id = E.department\_id

GROUP BY E.department\_id, D.department\_name

ORDER BY 3

**HAVING**- clause to filter groups returned by the GROUP BY clause

* The HAVING clause appears immediately after the GROUP BY clause.

Note that the HAVING clause filters groups of rows while the WHERE clause filters rows. This is a main difference between the HAVING and WHERE clauses.

Ex:

* **The following query will return only those jobs whose maximum salary is greater than $10,000**

SELECT job\_id, MAX(salary) AS "Highest salary"

FROM employees

GROUP BY job\_id

HAVING MAX(salary) > 10000;

HAVING with WHERE clause

Ex:

SELECT department\_id, COUNT(\*) AS headcount

FROM employees

WHERE salary < 13000

GROUP BY department\_id

HAVING COUNT(\*) > 5;

Ex:

* **The following query will return only those departments whose total salary is between $20,000 and $30,000.**

SELECT  department\_id, SUM(salary)

FROM employees

GROUP BY department\_id

HAVING SUM(salary) BETWEEN 20000 AND 30000

ORDER BY SUM(salary);

**CASE -**allows you to evaluate a list of conditions and returns one of the possible results. In other words, it goes through conditions and returns a value when the first condition is met. If no conditions is met, then it will return the value (default value) in the ELSE statement.

Syntax:

* The CASE statement is used grouping, filtering and aggregating.
* The syntax is as follows:
* CASE WHEN expression\_1 THEN result\_1
* WHEN expression\_2 THEN result\_2
* WHEN expression\_3 THEN result\_3
* ELSE else\_result
* END

Ex:

SELECT first\_name, salary

, CASE WHEN Salary < 2000 THEN 'Junior Level'

       WHEN (Salary >= 2000 and Salary < 5000) THEN 'Middle Level'

       WHEN (Salary >= 5000 and Salary < 10000) THEN 'Senior Level'

       ELSE 'Managers'

       END AS employee\_level

FROM employees;

Ex:

SELECT first\_name, last\_name, hire\_date, EXTRACT(YEAR FROM hire\_date)

, CASE (2010 - EXTRACT(YEAR FROM hire\_date))

      WHEN 1 THEN '1 year'

       WHEN 2 THEN '2 years'

        WHEN 3 THEN '3 years'

        WHEN 4 THEN '4 years'

        WHEN 5 THEN '5 years'

        ELSE '5+ years'

        END aniversary

FROM employees

ORDER BY 3;

**Aggregating with CASE:** MIN(), MAX(), SUM(), COUNT() and AVG()

Ex:

Let’s say we would like to know how many employees get commission and how many do not get any commission in the employees table based on the reported commision\_pct field

SELECT job\_id

, COUNT(CASE WHEN commission\_pct IS NULL THEN 'no\_commission' END) AS no\_commision\_count

, COUNT(CASE WHEN commission\_pct IS NOT NULL THEN 'get\_commission'  END) AS get\_commision\_count

FROM employees

GROUP BY job\_id;

* **Let’s consider $5,000 is minimum wage for the HR database. Write a sql query that returns department id and how many employees earn more than and less than equal to the minimum wage for each department.**

SELECT department\_id

, COUNT(CASE WHEN salary > 5000 THEN 'more\_than\_mw' END) AS more\_than\_mw\_count

, COUNT(CASE WHEN salary <= 5000 THEN 'less\_than\_mw'  END) AS less\_than\_mw\_count

FROM employees

GROUP BY department\_id;

**SUBQUERRIES-** Subquery or Inner query or a Nested query is a query within another SQL query. You can create subqueries within your SQL statements. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

* A subquery is a query within another query. The outer query is called as **main query** and inner query is called as **subquery**.
* A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
* Subqueries can be used with the SELECT, FROM, WHERE, HAVING statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
* Subqueries must be enclosed within parentheses.
* Ex: **Write a sql query which returns all employees who locate in the location with the id 1700**

SELECT employee\_id, first\_name, last\_name

FROM employees

WHERE department\_id IN (

SELECT department\_id

FROM departments

WHERE location\_id = 1700);

* Ex: **Write a SQL query to find those departments, located in the city ‘Munich’. Return department ID, department name**

SELECT department\_id, department\_name

FROM departments

WHERE location\_id = (SELECT location\_id FROM locations WHERE city = ‘Munich’);

* Ex: **Write a SQL query to find all those employees who earn more than an employee whose last name is 'Ozer'. Sort the result in ascending order by last name. Return first name, last name and salary.**

SELECT first\_name, last\_name, salary

FROM employees

WHERE salary > (SELECT salary FROM employees WHERE last\_name='Ozer')

ORDER BY  last\_name;

**IN and AVG**

* **Ex: write a SQL query to find those employees who earn more than average salary and who work in any of the ‘IT’ departments. Return first name and last name.**

SELECT first\_name,  last\_name

FROM employees

WHERE department\_id IN

(SELECT department\_id  FROM departments  WHERE department\_name LIKE 'IT%')

AND salary > (SELECT AVG(salary) FROM employees);

**WHERE and IN**

* **Ex: Write a SQL query to find those employees who work in a department where the employee’s first name contains a letter 'R'. Return employee ID, first name and last name.**

SELECT employee\_id, first\_name, last\_name

FROM employees

WHERE department\_id IN

( SELECT department\_id

FROM employees

WHERE first\_name LIKE '%R%' );

**WHERE and ALL**

* **Ex: Write a SQL query to find those employees who earn more than the maximum salary of a department of ID 30. Return first name, last name and department ID.**

SELECT first\_name, last\_name, salary, department\_id

FROM employees

WHERE salary > ALL

 (SELECT salary

 FROM employees

 WHERE department\_id = 30);

**WHERE and ANY**

* **Ex: Write a SQL query to find those employees whose salary is lower than any salary of those employees whose job title is 'IT\_PROG' . Return employee ID, first name, last name, job ID.**

SELECT employee\_id,first\_name,last\_name,job\_id

FROM employees

WHERE salary < ANY

( SELECT salary

FROM employees

WHERE job\_id = 'IT\_PROG' );

**WINDOWS FUNCTIONS=** allow us to apply functions like COUNT(), AVG(), COUNT(), MAX(), and MIN() on a group of records while still leaving the individual records accessible.

\* The [aggregate functions](https://www.sqltutorial.org/sql-aggregate-functions/) perform calculations across a set of rows and return a single output row.

Remember that **GROUP BY** collapses the individual records into group. You cannot refer to any individual field after using **GROUP BY** because it is collapsed.  
How will you create a report with employee names, salaries, and the maximum salary of all employees? You cannot do it with **GROUP BY**. To generate this kind of report, you need to use Window Functions.\*

Dif b/w ag function vs window function

A picture containing diagram

Description automatically generated

* Window functions applies aggregate and ranking functions over a particular window (set of rows).
* **OVER** clause is used with window functions to define that window.
* **PARTITION BY** is used to split the result set into partitions on which window function is applied.

The syntax is as follows:

SELECT coulmn\_name1

, window\_function(cloumn\_name2),OVER([PARTITION BY column\_name1] [ORDER BY column\_name3])

FROM table\_name;

* **window\_function=** any aggregate, ranking or value function
* **column\_name1**= column to be selected
* **column\_name2 =** column on which window function is to be applied
* **column\_name3**= column on whose basis partition of rows is to be done

**ORDER BY** orders rows within those partitions into a particular order.

* SUM()
* COUNT()
* MAX()
* MIN()
* AVG()
* MEDIAN
* STDEV()
* VARIANCE()

**Ex: The following query uses the MAX() as a window function. It returns the maximum salary of all employees along with the salary of each individual employee**

SELECT first\_name, last\_name, salary

, MAX(salary) OVER() AS max\_salary

FROM employees;

**Ex: The following query uses the MAX() as a window function and it returns a list of employee names, salaries, and the maximum salary in their departments.**

SELECT first\_name, last\_name, department\_id, salary

, MAX(salary) OVER(Partition BY department\_id) AS max\_salary

FROM employees;

**Ex: Write a SQL query which return employees first names, manager id, department id and minimum, maximum, total and average salary, and  employee headcount for each department from the window functions.**

SELECT first\_name, manager\_id, department\_id, salary

, MIN(salary) OVER( PARTITION BY department\_id) AS min\_salary

, MAX(salary) OVER( PARTITION BY department\_id) AS max\_salary

, SUM(salary) OVER( PARTITION BY department\_id ) AS sum\_Salary

, COUNT(\*) OVER( PARTITION BY department\_id) AS count\_emp

, AVG(salary) OVER( PARTITION BY department\_id ) AS avg\_salary

FROM employees;

When you need to use **ORDER BY** after **partition by** use:

**RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING**) As

-Considers the department\_Id as 1 partition

**RANK WINDOWS FUNCTIONS**

* **RANK()**:  It assigns rank to each row within every partition. For the next rank after two same rank values, one rank value will be skipped.
* **DENSE\_RANK()**: It assigns rank to each row within every partition. For the next rank after two same rank values, consecutive integer is used, no rank will be skipped (MOST USED)
* **ROW\_NUMBER()**: It assigns consecutive integers to each row within every partition. Within a partition, no two rows can have same row number.

**Ex: Calculate row number, rank, dense rank of employees in the employees table according to salary within each department.**

SELECT first\_name, department\_id, salary

, ROW\_NUMBER() OVER(PARTITION BY department\_id ORDER BY salary DESC) AS emp\_row\_no

, RANK() OVER(PARTITION BY department\_id ORDER BY salary DESC) AS emp\_rank

, DENSE\_RANK() OVER(PARTITION BY department\_id ORDER BY salary DESC) AS emp\_dense\_rank

FROM employees;

**VALUE WINDOW FUNCTIONS**

* **FIRST\_VALUE()**: It assigns the first value to each row within every partition.
* **LAST\_VALUE()**: It assigns the last value to each row within every partition.
* **LEAD()**: It returns an offset (incrementally increased) value of an argument column. The offset amount can be defined in the query; its default amount is 1. The syntax is LEAD(column\_name, offset, default value (optional) ). It retrieves the following rows.
* **LAG()**: It is the opposite of **LEAD()**. It retrieves the preceding (previous) rows.

**Ex:Write a sql query which returns first and last names, their department ids, salary and the employees who have the lowest salary in their each department.**

SELECT first\_name, last\_name, department\_id, salary

, FIRST\_VALUE(first\_name) OVER(PARTITION BY department\_id ORDER BY salary) AS lowest\_salary

FROM employees

**Ex: Write a sql query which returns first and last names, their department ids, salary and the employees who have the highest salary in their each department.**

SELECT first\_name, last\_name, department\_id, salary

, LAST\_VALUE(first\_name) OVER(PARTITION BY department\_id ORDER BY salary RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS highest\_salary

FROM employees;

**Ex: Write a sql query which returns first and last names, their department ids, hire date, the hire date of the employee hired just after in the company.**

SELECT first\_name, last\_name, department\_id, hire\_date

, LEAD(hire\_date, 1) OVER (ORDER BY hire\_date) AS next\_hire\_date

FROM employees;

* **Ex: Write a sql query which returns first and last names, their department ids, hire date, the hire date of the employee hired just after in the same department.**

SELECT first\_name, last\_name, department\_id, hire\_date

, LEAD(hire\_date, 1) OVER (PARTITION BY department\_id ORDER BY hire\_date) AS next\_hire\_date

FROM employees;

\*change number when you want a certain position of person hire date) Ex: , LEAD(hire\_date, 2)

**Ex: Write a sql query which returns first and last names, their department ids, hire date, the hire date of the employee hired just before in the company.**

SELECT first\_name, last\_name, department\_id, hire\_date

, LAG(hire\_date, 1) OVER (ORDER BY hire\_date) AS previous\_hire\_date

FROM employees;

**Ex: Try the following query**

SELECT first\_name, last\_name, department\_id, hire\_date

, LAG(hire\_date, 1) OVER ( ORDER BY hire\_date) AS previous\_hire\_date

, LAG(first\_name, 1) OVER ( ORDER BY hire\_date) AS previous\_hired\_name

FROM employees;

Common Table Expression (CTEs)

A Common Table Expression (CTE) is a named temporary result set that exists within the scope of a single query that can be referred later within that sql query. It can be though as a temporary table.

Here are the steps to create and use a CTE:

* It starts with **WITH** keyword and is followed by the expression name.
* Reference the CTE later in your main query.

**Syntax**

WITH CTE\_NAME AS

(

CTE QUERY

)

**Parameters**

* CTE\_NAME: It refers to the name we want to give the common table expression
* CTE query: It actually refers to the query or the statement.

**CTE using WITH ClauseWITH CTE AS**

(SELECT first\_name, last\_name, department\_id, salary

, RANK() OVER (PARTITION BY department\_id ORDER BY salary) salary\_rank

FROM employees

)

SELECT \* FROM CTE;

**Multiple CTE using WITH clause**

Ex: WITH CTE1 AS (

SELECT first\_name, last\_name, department\_id, salary

, RANK() OVER (PARTITION BY department\_id ORDER BY salary) salary\_rank

FROM employees

)

, CTE2 AS (

SELECT department\_id, department\_name

FROM departments

 )

SELECT first\_name, last\_name, department\_name, salary

FROM CTE1 C1

INNER JOIN CTE2 C2

ON C1.department\_id = C2.department\_id;

Ex:

WITH CTE1 AS (

SELECT first\_name, last\_name, department\_id, salary

, RANK() OVER (PARTITION BY department\_id ORDER BY salary) salary\_rank

FROM employees

)

, CTE2 AS (

SELECT department\_id, department\_name

FROM departments

 )

SELECT first\_name, last\_name, department\_name, salary

FROM CTE1 C1

INNER JOIN CTE2 C2

ON C1.department\_id = C2.department\_id

WHERE C1.salary\_rank = 2;