# 📃 Snowflake SQL Notes: CTEs, Recursive CTEs & Window Functions

## ✅ Audience

Freshers and professionals new to Snowflake and advanced SQL

## 📂 Data Setup Summary

Ensure the following tables exist (see script provided earlier):

* EMPLOYEES (EMP\_ID, EMP\_NAME, DEPARTMENT, SALARY, DOJ)
* PROJECTS (PROJECT\_ID, EMP\_ID, PROJECT\_NAME, START\_DATE, END\_DATE)

## 1. ❆ Common Table Expressions (CTEs) — WITH

### 🧠 Concept:

CTEs (Common Table Expressions) simplify complex queries by allowing you to define temporary result sets that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement. They’re useful for breaking down logic into readable blocks and are defined using the WITH keyword. CTEs only exist during the execution of the SQL query and are not stored as database objects.

CTEs are particularly useful for:

* Simplifying subqueries
* Improving readability
* Referencing the same logic multiple times
* Creating modular SQL scripts

### 🧪 Example 1: Get highest salary per department

WITH DeptMax AS (  
 SELECT DEPARTMENT, MAX(SALARY) AS MAX\_SALARY  
 FROM EMPLOYEES  
 GROUP BY DEPARTMENT  
)  
SELECT \* FROM DeptMax;

### 🧪 Example 2: Count employees in each department

WITH DeptCounts AS (  
 SELECT DEPARTMENT, COUNT(\*) AS EMP\_COUNT  
 FROM EMPLOYEES  
 GROUP BY DEPARTMENT  
)  
SELECT \* FROM DeptCounts;

### 🧪 Example 3 : Get all employees who joined after 2021

WITH RecentJoins AS (  
 SELECT \* FROM EMPLOYEES  
 WHERE DOJ >= '2021-01-01'  
)  
SELECT \* FROM RecentJoins;

### 🧪 Example 4 : Top 3 earners overall

WITH Ranked AS (  
 SELECT \*, RANK() OVER (ORDER BY SALARY DESC) AS RANKING  
 FROM EMPLOYEES  
)  
SELECT \* FROM Ranked WHERE RANKING <= 3;

### 🧪 Example 5 : Join CTE with aggregation and filtering

WITH DeptAgg AS (  
 SELECT DEPARTMENT, COUNT(\*) AS CNT, AVG(SALARY) AS AVG\_SAL  
 FROM EMPLOYEES  
 GROUP BY DEPARTMENT  
)  
SELECT \* FROM DeptAgg WHERE AVG\_SAL > 60000;

## 2. 🔄 Recursive CTEs

### 🧠 Concept:

Recursive CTEs are used to solve problems that require iterative logic or hierarchical traversal, such as org charts or bill-of-material structures. A recursive CTE has two parts:

1. **Anchor Member**: the starting point
2. **Recursive Member**: repeatedly executed until no new rows are returned

Syntax involves the WITH RECURSIVE clause and a UNION ALL between anchor and recursive members.

Use cases:

* Traversing employee hierarchies
* Generating series
* Simulating iterative conditions

### 🧪 Example 1 : Simulate levels of growth (salary-based tiers)

WITH RECURSIVE SalaryTiers(EMP\_NAME, SALARY, TIER) AS (  
 SELECT EMP\_NAME, SALARY, 1 AS TIER  
 FROM EMPLOYEES  
  
 UNION ALL  
  
 SELECT EMP\_NAME, SALARY, TIER + 1  
 FROM SalaryTiers  
 WHERE SALARY > TIER \* 30000  
)  
SELECT \* FROM SalaryTiers ORDER BY EMP\_NAME, TIER;

### 🧪 Example 2: Generate 5 numbers

WITH RECURSIVE Numbers(n) AS (  
 SELECT 1  
 UNION ALL  
 SELECT n + 1 FROM Numbers WHERE n < 5  
)  
SELECT \* FROM Numbers;

### 🧪 Example 3 : Generate year-wise employee count

WITH RECURSIVE Years(y) AS (  
 SELECT 2018  
 UNION ALL  
 SELECT y + 1 FROM Years WHERE y < 2023  
)  
SELECT y, COUNT(\*) AS EMP\_COUNT  
FROM Years y  
LEFT JOIN EMPLOYEES e ON YEAR(e.DOJ) = y  
GROUP BY y ORDER BY y;

### 🧪 Example 4: Flatten department salary tree (stepwise check)

WITH RECURSIVE Tree(DEPARTMENT, AVG\_SAL, STEP) AS (  
 SELECT DEPARTMENT, AVG(SALARY), 1 FROM EMPLOYEES GROUP BY DEPARTMENT  
 UNION ALL  
 SELECT DEPARTMENT, AVG\_SAL + 1000, STEP + 1  
 FROM Tree WHERE STEP < 3  
)  
SELECT \* FROM Tree;

### 🧪 Example 5: Simulate bonus accumulation

WITH RECURSIVE BonusTracker(EMP\_ID, BONUS, ROUND) AS (  
 SELECT EMP\_ID, 1000, 1 FROM EMPLOYEES  
 UNION ALL  
 SELECT EMP\_ID, BONUS + 500, ROUND + 1 FROM BonusTracker WHERE ROUND < 3  
)  
SELECT \* FROM BonusTracker ORDER BY EMP\_ID, ROUND;

## 3. 🔢 Window Functions

### 🧠 Concept:

Window functions perform calculations across a set of table rows related to the current row. Unlike aggregate functions, they do not group the result into a single output row.

Each window function uses the OVER() clause and can be partitioned and ordered. Common categories:

* Ranking: RANK(), DENSE\_RANK(), ROW\_NUMBER()
* Navigation: LEAD(), LAG()
* Aggregates: SUM() OVER(), AVG() OVER()

Use cases:

* Running totals
* Trend analysis
* Time-based comparison
* Row numbering within partitions

### 🧪 Example 1: RANK within department

SELECT \*, RANK() OVER (PARTITION BY DEPARTMENT ORDER BY SALARY DESC) AS SAL\_RANK  
FROM EMPLOYEES;

### 🧪 Example 2: DENSE\_RANK across all

SELECT \*, DENSE\_RANK() OVER (ORDER BY SALARY DESC) AS DENSE\_SAL\_RANK  
FROM EMPLOYEES;

### 🧪 Example 3: ROW\_NUMBER across employees

SELECT \*, ROW\_NUMBER() OVER (ORDER BY DOJ ASC) AS JOIN\_ORDER  
FROM EMPLOYEES;

### 🧪 Example 4: LEAD/LAG salary

SELECT EMP\_NAME, SALARY,  
 LAG(SALARY) OVER (ORDER BY DOJ) AS PREV\_SALARY,  
 LEAD(SALARY) OVER (ORDER BY DOJ) AS NEXT\_SALARY  
FROM EMPLOYEES;

### 🧪 Example 5: Running total salary by department

SELECT EMP\_NAME, DEPARTMENT, SALARY,  
 SUM(SALARY) OVER (PARTITION BY DEPARTMENT ORDER BY DOJ ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS CUM\_SAL  
FROM EMPLOYEES;

## 📄 Summary Cheat Sheet

* WITH CTE\_NAME AS (...) → use for query modularity
* RECURSIVE CTE → self-join logic for hierarchies or simulations
* RANK() → skips ranks when ties
* DENSE\_RANK() → no skipping
* ROW\_NUMBER() → unique per row
* LEAD()/LAG() → future/past values
* SUM() OVER() → running/partition totals

Ready to practice? Use these on the sample dataset and visualize results clearly with row numbers.