4 Common Table Expressions and Recursive Queries

In SQL-99 and later versions, Common Table Expressions (CTEs) and recursive queries can be defined.

Example:
Employee (empNr, deptNr, mgrNr, salary, bonus) with mgrNr referring to empNr as a foreign key. mgrNr represents the immediate supervisor of the employee empNr.

You can create this table by entering the following SQL code into the DBMS:

```
CREATE TABLE Employee (
empNr INT PRIMARY KEY,
deptNr INT,
mgrNr INT,
salary FLOAT,
bonus FLOAT);
```

As an example, you can populate the table with the following tuples:

```
INSERT INTO Employee VALUES
(9, 101, NULL, 180, 20),
(8, 101, 9, 110, 10),
(7, 101, 9, 70, 5),
(6, 101, 7, 120, 15),
(5, 101, 7, 50, 5),
(4, 101, 6, 150, 15),
(3, 101, 5, 90, 10),
(2, 101, 5, 50, 5),
(1, 101, 3, 110, 10);
```

Query:
Find the department that has the highest sum of salaries.

First approach:

```
CREATE VIEW SumList (deptNr, salarySum) AS
                SELECT deptNr, SUM(salary) + SUM(bonus)
                FROM Employee
                GROUP BY deptNr;

SELECT deptNr, salarySum
FROM SumList
WHERE salarySum = (SELECT MAX(salarySum)
                              FROM SumList);
```

Answer from the system:

```
+--------+-----------+
| deptNr | salarySum |
+--------+-----------+
|    101 |      1025 |
+--------+-----------+
```

Problem:

- The method is awkward since only for this query, a new view is created and afterwards perhaps deleted again.

Next approach:

```
SELECT deptNr, salarySum
FROM (SELECT deptNr, SUM(salary) + SUM(bonus) AS salarySum
          FROM Employee
          GROUP BY deptNr) AS SumList_1
WHERE salarySum =
          (SELECT MAX(salarySum)
           FROM (SELECT deptNr, SUM(salary) + SUM(bonus) AS salarySum
                     FROM Employee
                     GROUP BY deptNr) AS SumList_2);
```

The answer from the system is the same as above:

```
+--------+-----------+
| deptNr | salarySum |
+--------+-----------+
|    101 |      1025 |
+--------+-----------+
```

Note:
Every derived table must have its own alias (here: SumList_1 and SumList_2).
Problem:

- The same table expression is evaluated more than once in a single query
  => Unnecessary work, takes longer.
  => Potential inconsistencies depending on the isolation level set in the system. Unrepeatable reads may be possible. (Isolation levels "read uncommitted" and "read committed" allow unrepeatable reads).

Solution:
Common Table Expressions (CTE) are built as a table expression and represent temporary, named resultsets. In a query, multiple references to the same CTE are possible without having to evaluate it more than once.

Syntax for CTE:
WITH cteName { (columnName, ...) } AS (SelectStatement)
This can be followed by a query referring to cteName once or several times.

New approach for the query above:

```
WITH SumList (deptNr, salarySum) AS
        (SELECT deptNr, SUM(salary) + SUM(bonus)
         FROM Employee
         GROUP BY deptNr)
SELECT deptNr, salarySum
FROM SumList
WHERE salarySum =
        (SELECT MAX(salarySum)
          FROM SumList);
```

The answer from the system is still the same as above:

```
+--------+-----------+
| deptNr | salarySum |
+--------+-----------+
|    101 |      1025 |
+--------+-----------+
```

Now:
The view SumList is evaluated only once. The evaluation is optimized by the internal query optimizer.

Recursion:

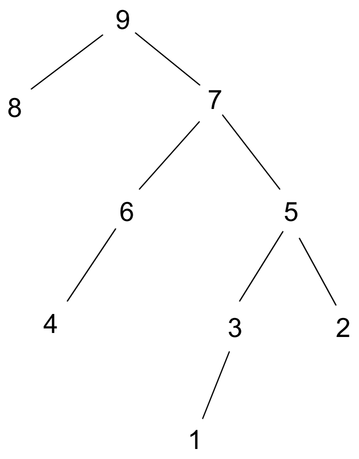
A common table expression is recursive if the cteName it defines is also referred to within the WITH clause.

This construction is useful for queries that recursively refer to the same table, for instance traversal of trees and networks.

The relation Employee represents the job hierarchy as a tree, for example, the following persons could be represented (see the tuples specified before):

Employee		
empNr	**mgrNr**	**salary**
9	-	180
8	9	110
7	9	70
6	7	120

Employee		
empNr	**mgrNr**	**salary**
5	7	50
4	6	150
3	5	90
2	5	50
1	3	110



Query:
Find all employees who have the direct supervisor 7 and whose salary is greater than 100.

```
SELECT empNr, salary
FROM Employee
WHERE mgrNr = 7
AND salary > 100;
```

System answer:

```
+-------+--------+
| empNr | salary |
+-------+--------+
|     6 |    120 |
+-------+--------+
```

Query:
Find all employees, to whom 7 is supervisor (either direct or higher up in the hierarchy) and whose salary is greater than 100.

Solution strategy:
First build a table with all supervisees, direct and indirect by recursively building a table:
- Start with the direct supervisees of 7
- Recursively add the supervisees of the supervisees.
- Termination when no more supervisees can be added.

Then query this table expression.

```
WITH RECURSIVE Supervisees (empNr, salary) AS
        (  (SELECT empNr, salary
             FROM Employee
             WHERE mgrNr = 7)
           UNION ALL
           (SELECT Employee.empNr, Employee.salary
             FROM Supervisees, Employee
             WHERE Employee.mgrNr = Supervisees.empNr) )
```

SELECT empNr, salary
FROM Supervisees
WHERE salary > 100;

Now the system answer includes recursively all emloyees further down the hierarchy tree:

```
+-------+--------+
| empNr | salary |
+-------+--------+
|     6 |    120 |
|     4 |    150 |
|     1 |    110 |
+-------+--------+
```

Evaluation:

Employee		
empNr	**mgrNr**	**salary**
9	-	180
8	9	110
7	9	70
6	7	120
5	7	50
4	6	150
3	5	90
2	5	50
1	3	110

Supervisees	
empNr	**salary**
6	120
5	50
4	150
3	90
2	50
1	110

Query Result
empNr
6
4
1

The general syntax for such a query is:

WITH RECURSIVE recursiontable (attribute list) AS

```
(
  query expression
)
[traversal clause] [cycle clause]
query expression;
```

Note:

- The UNION ALL clause is mandatory in the syntax of recursive SQL. It removes no duplicates, so the result grows monotonously. If the result was allowed to also shrink or tuples to be changed during the evaluation, some steps might be performed again and again, causing an infinite loop. (In the given example this would not happen, though).
- The tree traversal is breadth first. By adding the traversal clause SEARCH DEPTH FIRST or SEARCH BREADTH FIRST, one can explicitly specify the sequence of the search results.
- A key question with recursion is whether the algorithm terminates.
  In the above example with the management hierarchy, the algorithm will terminate because in each step of building Supervisees, only a finite number of tuples is added, and only a finite number of steps is executed. This is because the structure is a tree (hence no cycles) with a finite depth.
  The cycle clause in the WITH statement serves to exclude cycles by specifying a maximum depth down to which the recursion proceeds before stopping. In systems that don’t implement this, one can define an additional column for the CTE with a numerical value that is increased after each recursive step. The recursive query part can in the WHERE part restrict this parameter to a maximum value, meaning that the recursion will be executed only up to a certain depth. (For more details see for instance Can Türker: SQL:1999 & SQL:2003).