Advanced SQL

A. Common Table Expression

a. Common Table Expression (CTE) - specifies a temporary named result set.

- ➢ Can be defined using the **WITH** operator
- ➢ Allows users to define tables that can be used in a particular query
- ➢ Can be referenced within another SELECT, INSERT, UPDATE, or DELETE statement
- ➢ Clauses like ORDER BY, INTO, and OPTION cannot be used in CTE queries.

o **Types of CTE**

a. Non-Recursive CTE

- ▪ Doesn't use repeated procedural loops/recursion
- ▪ Easier to understand

b. Recursive CTE

- ▪ Uses recursion
- ▪ Useful when working with hierarchical data because the CTE continues to execute until the query returns the entire hierarchy

Assume that we have a table (see table 1) named **Orders** having a columns *OrderID, OrderDate, Cust_ID, ItemID.*

Next, using **non-recursive** queries, we will select the data of customers (excluding *ItemID* column) who ordered in 2017.

```
-- Non-recursive
;WITH Simple_CTE (OrderID, Cust_ID, OrderDate)
AS
(
  SELECT O.OrderID, O.CustomerID, O.OrderDate
  FROM Orders O
)

SELECT * FROM Simple_CTE WHERE YEAR(Orderdate) = 2017
```

OrderID	OrderDate	ItemID	CustomerID
OR-1	2019-08-14	IT-5	Cust-3
OR-2	2019-12-04	IT-2	Cust-6
OR-3	2017-08-20	IT-7	Cust-2
OR-4	2018-06-12	IT-10	Cust-4
OR-5	2017-02-28	IT-5	Cust-9
OR-6	2019-05-21	IT-9	Cust-10
OR-7	2017-12-27	IT-2	Cust-1
OR-8	2017-02-10	IT-4	Cust-5
OR-9	2018-09-12	IT-10	Cust-3
OR-10	2018-01-17	IT-3	Cust-4

***Table 1.** Orders*

Output:

OrderID	CustomerID	OrderDate
OR-3	Cust-2	2017-08-20
OR-5	Cust-9	2017-02-28
OR-7	Cust-1	2017-12-27
OR-8	Cust-5	2017-02-10

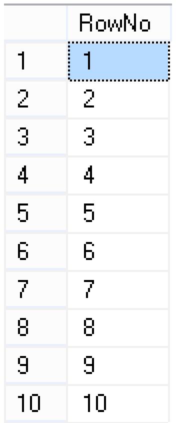
Explanation:

1. We defined *Simple_CTE* as the name of common table expression. The CTE returns a result set that consists of three columns: *OrderID, Cust_ID, OrderDate*.
2. We constructed a query that retrieves all the data from table **Orders,** excluding *ItemID* column.
   (**Note:** *We can give a temporary name or alias to a specific column or table without using the "AS" keyword.*)
3. We get our dataset from *Simple_CTE* and select only the rows whose year is *2017*.

Now, we will create a simple **recursive query** to display the row number from 1 to 10

```
-- Recursive
;WITH Recursive_CTE
AS
(
        SELECT 1 AS RowNo      --Anchor part
        UNION ALL
        SELECT RowNo + 1       --Recursive part
        FROM Recursive_CTE
        WHERE RowNo < 10       --terminator
)
SELECT * FROM Recursive_CTE     --Statement using CTE
```

Output:



```
    RowNo
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
9   9
10  10
```

Explanation:

1. We defined *Recursive_CTE* as the name of common table expression.
2. We constructed a query that will define a temporary column named *RowNo* and set its value as 1.
3. In the anchor part, we display the base value of *RowNo*.
4. By using UNION ALL, we execute the recursive part repeatedly until the row satisfied the condition in the WHERE clause.
5. Using the SELECT statement, we retrieve the data from Recursive_CTE.

Another example of Recursive CTE:
Assume that we have a table (see figure 2) named **ClassOfficers** having columns *OfficerID, OfficerName, Position, and Reporting.*

In the column *Reporting*, each row represents to whom *OfficerID* they are reporting. Since Chin was the president, her reporting value is null because she got the highest-class position.

OfficerID	OfficerName	Position	Reporting
201	Chin	President	NULL
207	Jeremiah	Vice President	201
209	Baldo	Secretary	201
212	JunJun	Treasurer	209
223	Bernadette	Spokesperson	209

Table 2. *Officers*

Next, using recursive queries, we will display the reporting structure of all class officers.

```
;WITH ReportingStructure(Reporting, OfficerID, OfficerName,
Position, PosLevel) AS
(
  -- Anchor part
  SELECT Reporting, OfficerID, OfficerName, Position, 0
AS PosLevel
  FROM Officers
  WHERE Reporting IS NULL
  UNION ALL
  -- Recursive part
  SELECT O.Reporting, O.OfficerID, O.OfficerName,
      O.Position, R.PosLevel + 1
  FROM Officers O
  INNER JOIN ReportingStructure R
  ON  R.OfficerID = O.Reporting
)
SELECT Reporting, OfficerID, OfficerName,
    Position, PosLevel
FROM ReportingStructure;
```

Output:

Reporting	OfficerID	OfficerName	Position	PosLevel
NULL	201	Chin	President	0
201	207	Jeremiah	Vice President	1
201	209	Baldo	Secretary	1
209	212	JunJun	Treasurer	2
209	223	Bernadette	Spokesperson	2

Explanation:

1. The ReportingStructure CTE returns a result set that consists of five (5) columns: *Reporting, OfficerID, OfficerName, Position, PosLevel*.
2. In the anchor part, we retrieve the four columns from the *Officers* table and returns the base result set of *PosLevel* as 0. This is the highest-ranking officer as she does not report to a higher position.
3. In the recursive query, it returns the other officers in the anchor part result set. This self-referring is achieved by a *JOIN* statement between *Officers* table and *ReportingStructure* CTE.
4. Then, we increment the *PosLeve*l every time the statement loops through the hierarchy.
5. The termination condition is met through a *JOIN* statement where it stops its recursion when all rows are returned from *Officers* table.
6. After defining the WITH clause, we create a SELECT statement that retrieves the data from CTE.

B. Subqueries – A query (SELECT statement) inside another query.

- Expressed inside parentheses
- The first query in the SQL statement is known as the outer query.
- Query inside the SQL statement is known as the inner query.
- The inner query is executed first.
- The output of an inner query is used as the input for the outer query.

a. **In WHERE clause –** work as part of the row selection process.
  - A subquery often found in the WHERE clause
  - Also called nested subqueries.

b. **In FROM clause –** returns a temporary or virtual table.
  - Useful in data warehousing application
  - Also called an inline view or derived table

c. **In SELECT clause –** a subquery that is nested in the list of another SELECT statement.

d. **In IN operator –** allows users to match one item from any of those in the list.

e. **In ALL and ANY operator**
  - **ANY** – returns true if any of the subquery values satisfy the condition.
  - **ALL** – returns true if all the subquery values meet the condition.

f. **Correlated Subqueries –** are used to select data from a table referenced in the outer query.
  - Cannot be executed independently as a simple subquery
  - A correlated subquery is executed repeatedly, once for each row evaluated by the outer query.
  - Also known as a repeating subquery

g. **In EXISTS operator –** used to check whether a subquery produces any rows of query results.
  - Commonly used with correlated subqueries

Subqueries Examples:

Assume that we have three (3) tables named *Customers, Items, and Orders* (see table 3, 4, and 5).

CustomerID	Cust_Name	Address
Cust-1	Badang	Taytay
Cust-2	Hanzo	Angono
Cust-3	Lilia	Cainta
Cust-4	Layla	Angono
Cust-5	Lesley	Taytay
Cust-6	Balmond	Cainta
Cust-7	Chou	Pililia
Cust-8	Eudora	Cardona
Cust-9	Miya	Pililia
Cust-10	Cecilion	Cardona

Table 3. *Customers*

ItemID	ItemName	Price
IT-1	Samsung S9	32000.00
IT-2	Samsung S10	40000.00
IT-3	Huawei P30	29000.00
IT-4	Huawei Mate30	31000.00
IT-5	Realme 6	9999.99
IT-6	Realme 6 Pro	11000.00

IT-7	Xiaomi Note 9	8000.00
IT-8	Xiaomi Note 10	11000.00
IT-9	LG V40	15000.00
IT-10	LG V50	23000.00

Table 4. *Items*

OrderID	OrderDate	ItemID	CustomerID
OR-1	2019-08-14	IT-5	Cust-3
OR-2	2019-12-04	IT-2	Cust-6
OR-3	2017-08-20	IT-7	Cust-2
OR-4	2018-06-12	IT-10	Cust-4
OR-5	2017-02-28	IT-5	Cust-9
OR-6	2019-05-21	IT-9	Cust-10
OR-7	2017-12-27	IT-2	Cust-1
OR-8	2017-02-10	IT-4	Cust-5
OR-9	2018-09-12	IT-10	Cust-3
OR-10	2018-01-17	IT-3	Cust-4

Table 5. *Orders*

a. WHERE clause

- Display the items that have a price lower than or equal to LG V40.

```
SELECT *
FROM Items
WHERE Price < (SELECT Price FROM Items
               WHERE ItemName = 'LG V40')
```

Output:

ItemID	ItemName	Price
IT-5	Realme 6	9,999.99
IT-6	Realme 6 Pro	11,000.00
IT-7	Xiaomi Note 9	8,000.00
IT-8	Xiaomi Note 10	11,000.00

b. FROM clause

- Display the items that have a price higher than LG V40.

```
SELECT I.ItemID, I.ItemName, I.Price
FROM (SELECT Price FROM Items WHERE ItemName = 'LG
        V40') AS TEMP_table, Items I
WHERE I.Price > TEMP_table.Price
```

Output:

ItemID	ItemName	Price
IT-1	Samsung S9	32,000.00
IT-2	Samsung S10	40,000.00
IT-3	Huawei P30	29,000.00
IT-4	Huawei Mate30	31,000.00
IT-10	LG V50	23,000.00

c. SELECT clause

- Using CONCAT and SUBSTRING function, change the structure of Customer ID and display them as "NewID-##".

```
SELECT TOP 5 (SELECT CONCAT('NewID',
                (SELECT SUBSTRING(CustomerID, 5, 3))))
                 AS 'New ID Format', Cust_Name
FROM Customers
–TOP keyword limits the returning rows
```

Output:

New ID Format	Cust_Name
NewID-1	Badang
NewID-2	Hanzo
NewID-3	Lilia
NewID-4	Layla
NewID-5	Lesley

d. IN operator

- Display the customers who purchased a gadget with a brand name of Huawei and Samsung.

```
SELECT      C.Cust_Name,      C.Address,      I.ItemName,
YEAR(O.OrderDate) AS 'Year of sales'

FROM Orders O
JOIN Customers C ON O.CustomerID = C.CustomerID
JOIN Items I ON I.ItemID = O.ItemID

WHERE I.ItemID
          IN (SELECT I.ItemID FROM Items
                    WHERE  I.ItemName  LIKE '%SAMSUNG%' OR
I.ItemName LIKE '%HUAWEI%')
                    AND YEAR(O.OrderDate) >= 2018
```

Output:

Cust_Name	Address	ItemName	Year of sales
Layla	Angono	Huawei P30	2018
Balmond	Cainta	Samsung S10	2019

e. ALL and ANY

- Using **ANY** operator, display the items that have a price higher than LG V40 **OR** Huawei Mate30.

```
SELECT ItemName, Price
FROM Items
WHERE Price > ANY
  (SELECT Price FROM Items WHERE ItemName IN ('LG V40',
    'HUAWEI MATE30'))
GROUP BY ItemName, Price
```

Output:

ItemName	Price
Huawei Mate30	31000.00
Huawei P30	29000.00
LG V50	23000.00
Samsung S10	40000.00
Samsung S9	32000.00

- Using **ALL** operator, display the items that have a price higher than LG V40 **AND** Huawei Mate30.

```
SELECT ItemName, Price
FROM Items
WHERE Price > ALL
  (SELECT Price FROM Items WHERE ItemName IN ('LG V40',
    'HUAWEI MATE30'))
GROUP BY ItemName, Price
```

Output:

ItemName	Price
Samsung S10	40000.00
Samsung S9	32000.00

f. Correlated Subquery

- Display the list of customers who bought gadgets in 2018.

```
SELECT (SELECT Cust_Name FROM Customers C
WHERE C.CustomerID = O.CustomerID) AS Cust_Name,
YEAR(O.OrderDate) AS 'Year bought'
FROM Orders O
WHERE YEAR(O.OrderDate) = 2018
```

Output:

Cust_Name	Year bought
Layla	2018
Lilia	2018
Layla	2018

g. EXISTS operator

- Display the list of customers who bought gadgets in 2017.

```
SELECT C.CustomerID, C.Cust_Name
FROM Customers C
WHERE EXISTS
   (SELECT O.CustomerID FROM Orders O
    WHERE O.CustomerID = C.CustomerID AND
    YEAR(O.OrderDate) = 2017)
```

Output:

CustomerID	Cust_Name
Cust-1	Badang
Cust-2	Hanzo
Cust-5	Lesley
Cust-9	Miya

C. Views

- **Views –** a virtual table that is constructed from other tables or views and saved as an object in the database
  - ➢ Has no data of its own, but obtains data from tables or other views
  - ➢ Cannot include the following:
    1. ORDER BY clause
    2. A reference to a temporary table or a table variable.

Using tables Customers, Orders, Items (see table 3, 4, and 5). We will **create** a View that retrieves specific columns from three tables.

```
CREATE VIEW orders_report
AS
   SELECT o.OrderID, c.Cust_Name, c.Address, i.ItemName ,O.
OrderDate
   FROM Orders O
   JOIN Customers C ON C.CustomerID = O.CustomerID
   JOIN Items I ON I.ItemID = O.ItemID
```

From the newly created view, we will select the rows whose customer's address is Taytay and Angono.

```
SELECT * FROM sales_report WHERE Address IN
('Taytay','Angono')
```

Output:

OrderID	Cust_Name	Address	ItemName	OrderDate
OR-7	Badang	Taytay	Samsung S10	2017-12-27
OR-10	Layla	Angono	Huawei P30	2018-01-17
OR-8	Lesley	Taytay	Huawei Mate30	2017-02-10
OR-3	Hanzo	Angono	Xiaomi Note 9	2017-08-20
OR-4	Layla	Angono	LG V50	2018-06-12

For **updating** the view. See the example below.

```
ALTER VIEW orders_report
AS
   SELECT o.OrderID, O.OrderDate
FROM Orders O
```

For **deleting** a view. See the example below.

```
DROP VIEW orders_report
```

D. Index

- **Index –** used to speed up searches/queries, resulting in high performance
  - o Factors to consider creating an index:
    1. Frequency of search – creating an index to a particular column that is frequently searched can give performance benefits.
    2. Size of table – putting an index on a relatively large table that contains a great number of rows can improve performance.
    3. Number of updates – a database that is frequently updated should have fewer indexes as it slows the performance of inserts, updates, and deletes.
    4. Space considerations – create an index only if necessary, because indexes take up spaces within the database.

Example:

- **Single-Column Indexes –** based on only one table column.
  Ex.

```
CREATE INDEX ix_CustomerID
ON Customers (CustomerID)
```

- **Unique Indexes** – does not allow any duplicate values to be inserted into the table.
  Ex.

```
CREATE UNIQUE INDEX ix_ItemID
ON Items (ItemID)
```

- **Composite Indexes** – based on two or more columns of a table.
  Ex.

```
CREATE INDEX ix_OrderRecords
ON Orders (OrderID, OrderDate, CustomerID, ItemID)
```

- **Dropping Index** – deleting an index can be done using the DROP command.
  Ex.

```
DROP INDEX Orders.ix_OrderRecords
```

REFERENCES

Coronel, C. and Morris, S. (2018). *Database systems design, implementation, & management (13th ed.)*. Cengage Learning.

Elmasri, R. & Navathe, S. (2016). *Fundamentals of database systems (7th ed.)*. Pearson Higher Education.

Kroenke, D. & Auer, D. *Database processing: Fundamentals, design, and implementation (12th ed.)*. Pearson Higher Education.

Silberschatz A., Korth H.F., & Sudarshan, S. (2019). *Database system concepts (7th ed.)*. McGraw-Hill Education.

EssentialSQL. (n.d.). *Recursive CTEs explained*. Retrieved from https://www.essentialsql.com/recursive-ctes-explained/

MariaDB. (n.d.). *Subqueries in a FROM clause.* Retrieved from https://mariadb.com/kb/en/subqueries-in-a-from-clause/

Microsoft. (n.d.). *CREATE VIEW (Transact-SQL).* Retrieved from https://docs.microsoft.com/en-us/sql/t-sql/statements/create-view-transact-sql?view=sql-server-ver15

SQLservertutorial.net (n.d.). *SQL server subquery.* Retrieved from https://www.sqlservertutorial.net/sql-server-basics/sql-server-subquery/

Techonthenet (n.d.). *SQL Server: Subqueries.* Retrieved from https://www.techonthenet.com/sql_server/subqueries.php

W3resource (n.d.). *SQL subqueries.* Retrieved from https://www.w3resource.com/sql/subqueries/understanding-sql-subqueries.php