05

Subqueries, CTE and Intermediate Practices

Enhance your queries with common table expression and subqueries

MAZ Học Data

Writing Subqueries

A **subquery** is a nested query—a query within a query. One reason to use a subquery is to find the rows in one table that match the rows in another table without actually joining the second table

Scalar or Multi-Valued Subqueries

Scalar subquery returns single value to outer query

- Can be used anywhere single-valued expression is used: **SELECT, WHERE, and FROM**

```
SELECT SalesOrderID, ProductID, OrderQty
FROM SalesLT.SalesOrderDetail
WHERE SalesOrderID =
  ( SELECT MAX(SalesOrderID)
    FROM SalesLT.SalesOrderHeader )
```

Multi-valued subquery returns multiple values as a single column set to the outer query

- Used with IN predicate

```
SELECT CustomerID, SalesOrderID
FROM SalesLT.SalesOrderHeader
WHERE CustomerID IN (
  SELECT CustomerID
  FROM SalesLT.Customer
  WHERE  MiddleName is NULL )
```

Practice: Writing Subqueries

Using Common Table Expressions (CTE)

Microsoft introduced the **CTE** feature with SQL Server 2005. This gives developers another way to separate out the logic of part of the query. When writing a **CTE**, you define one or more queries upfront, which you can then immediately use.

- Here is the syntax:

```
WITH <new table name> AS (
    SELECT <select list FROM <table>
)
SELECT * FROM  < new table name >
```

- Example:

Practice: CTE

06

Grouping, summarizing data and discovering analytics functions

Use advanced analytics queries with
GROUP BY and WINDOW FUNCTION

Aggregate Functions

These functions operate on sets of values from multiple rows all at once.

- **COUNT**: Counts the number of rows or the number of non-NULL values in a column.
- **SUM**: Adds up the values in numeric or money data.
- **AVG**: Calculates the average in numeric or money data.
- **MIN**: Finds the lowest value in the set of values. This can be used on string data as well as numeric, money, or date data.
- **MAX**: Finds the highest value in the set of values. This can be used on string data as well as numeric, money, or date data.

```
SELECT COUNT(*) AS CountOfRows,
        MAX ( TotalDue ) AS MaxTotal,
        MIN ( TotalDue ) AS MinTotal,
        SUM ( TotalDue ) AS SumOfTotal,
        AVG ( TotalDue ) AS AvgTotal
FROM Sales.SalesOrderHeader
```

The GROUP BY Clause

You can use the **GROUP BY** clause to group data so the aggregate functions apply to groups of values instead of the entire result set.

One big difference you will notice once the query contains a GROUP BY clause is that additional **nonaggregated columns** may be included in the SELECT list. Once nonaggregated columns are in the SELECT list, you must add the GROUP BY clause and include all the nonaggregated columns.

- Here is the syntax:

```
SELECT  <col1>
        , <aggregate function>(<col2>)
FROM    <table>
GROUP BY  <col1>
```

- Example:

```
SELECT TerritoryID
        , AVG (TotalDue) AS AveragePerTerritory
FROM Sales.SalesOrderHeader
GROUP BY TerritoryID
```

The HAVING Clause

To eliminate rows based on an aggregate expression, use the **HAVING** clause. The HAVING clause may contain aggregate expressions that may or may not appear in the SELECT list

- Example:

```
SELECT CustomerID
      ,SUM(TotalDue) AS TotalPerCustomer
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
HAVING SUM(TotalDue) > 5000;
```

```
SELECT CustomerID
      ,SUM(TotalDue) AS TotalPerCustomer
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
HAVING COUNT(*) = 10 AND SUM(TotalDue) > 5000;
```

Summary

If you follow the steps outlined in the preceding sections, you will be able to write aggregate queries. With practice, you will become proficient in doing this. Keep the following rules in mind when writing an aggregate query:

- Any column not contained in an aggregate function in the SELECT list or ORDER BY clause must be part of the GROUP BY clause.
- Once an aggregate function, the GROUP BY clause, or the HAVING clause appears in a query, it is an aggregate query.
- Use the WHERE clause to filter out rows before the grouping and aggregates are applied.
- The WHERE clause doesn't allow aggregate functions.
- Use the HAVING clause to filter out rows using aggregate functions.
- **Don't include anything in the SELECT list or ORDER BY clause that you don't want as a grouping level**

Windowing Functions

What Is a Windowing Function?

Windowing functions operate on the set of the data that is returned to the client. They might perform a calculation like a SUM over all the rows without losing the details, rank the data, or pull a value from a different row without doing a self-join.

Windowing functions are allowed only in the SELECT and ORDER BY clauses.

- **Ranking functions**: This type of function adds a ranking for each row or divides the rows into buckets.
- **Window aggregates**: This function allows you to calculate summary values in a nonaggregated query.
- **Accumulating aggregates**: Enables the calculation of running totals.
- **Analytic functions**: Several new scalar functions, four of which are almost magical!

Windowing Functions

Defining the Window by OVER which then determines how the function is applied to the data.

```
SELECT column 1
        , column 2
        , Ranking function/Aggregate function OVER( PARTITION BY column 3 ORDER BY column 4 ASC/DESC ) AS new_column
FROM  Table_name ;
```

Function Category	Description
Scalar	Operate on a single row, return a single value
Logical	Compare multiple values to determine a single output
Ranking	**Operate on a partition (set) of rows**
Aggregate	**Take one or more input values, return a single summarizing value**

Aggregate Functions

An aggregate function performs a calculation on a set of values, and returns a single value. **Except for COUNT**, **aggregate functions ignore null values**. Aggregate functions are often used with the GROUP BY clause of the SELECT statement

MIN(): Return the smallest value of the selected column.

```
SELECT MIN (colum_1) FROM table_name
```

MAX(): Return the largest value of the selected column.

```
SELECT MAX (colum_1) FROM table_name
```

COUNT(): Return the number of rows that matches a specified crieria.

```
SELECT COUNT (colum_1) FROM table_name
```

AVG(): Return the average value of a numeric column.

```
SELECT AVG (colum_1) FROM table_name
```

SUM() Return the total sum of a numeric column.

```
SELECT SUM (colum_1) FROM table_name
```

Ranking Functions

ROW_NUMBER, RANK, DENSE_RANK, NTILE

The ranking functions — **ROW_NUMBER, RANK, DENSE_RANK**, and **NTILE—were** added to SQL Server as part of SQL Server 2005. The first three assign a ranking number to each row in the result set. The NTILE function divides a set of rows into buckets.

```
SELECT Customer_name, order_id, Order_Date,
ROW_NUMBER() OVER( ORDER BY Order_Date ASC) AS
[ROW_NUMBER],
RANK() OVER( ORDER BY Order_Date ASC ) AS [RANK],
DENSE_RANK() OVER( ORDER BY Order_Date ASC) AS [DENSE_RANK]
FROM Orders
WHERE Customer_name = 'Tamara Chand';
```

Results | Messages

	Customer_name	order_id	Order_Date	ROW_NUMBER	RANK	DENSE_RANK
1	Tamara Chand	57415	2009-03-24 00:00:00.000	1	1	1
2	Tamara Chand	6592	2009-12-01 00:00:00.000	2	2	2
3	Tamara Chand	6592	2009-12-01 00:00:00.000	3	2	2
4	Tamara Chand	640	2010-01-22 00:00:00.000	4	4	3
5	Tamara Chand	640	2010-01-22 00:00:00.000	5	4	3
6	Tamara Chand	44960	2011-05-03 00:00:00.000	6	6	4
7	Tamara Chand	44960	2011-05-03 00:00:00.000	7	6	4
8	Tamara Chand	44960	2011-05-03 00:00:00.000	8	6	4

Windowing Functions

Table name: Product

ProductID	Name	Price	Category	SubCategory
1	Martin 110	1,000	Vehicle	bike
2	Honda wave	1,200	Vehicle	motobike
3	T-shirt	200	Clothing	shirt
4	Mmen's pants	100	Clothing	trousers
5	Candy	50	Grocery	Confectionery
6	Sandwitch	20	Grocery	Cake
7	Bread	70	Grocery	Cake
8	Cookie	110	Grocery	Cake

ProductID	Name	Price	Category	SubCategory	# product by category	Rank product by price	Rank product by price per category
1	Martin 110	1,000	Vehicle	bike	2	7	1
2	Honda wave	1,200	Vehicle	motobike	2	8	2
3	T-shirt	200	Clothing	shirt	2	6	2
4	Mmen's pants	100	Clothing	trousers	2	4	1
5	Candy	50	Grocery	Confectionery	4	2	2
6	Sandwitch	20	Grocery	Cake	4	1	1
7	Bread	70	Grocery	Cake	4	3	3
8	Cookie	110	Grocery	Cake	4	5	4

```
SELECT *
        , COUNT(ProductID) OVER( PARTTION BY categoy ) AS '# product by category'
        , RANK() OVER( ORDER BY ProductID DESC) AS 'rank product by price'
        , RANK() OVER( PARTITON BY category ORDER BY ProductID DESC) AS 'rank product by price per category'
FROM Product
```

Calculating Running Total

Table name: Product

ProductID	Name	Price	Category	SubCategory
1	Martin 110	1,000	Vehicle	bike
2	Honda wave	1,200	Vehicle	motobike
3	T-shirt	200	Clothing	shirt
4	Mmen's pants	100	Clothing	trousers
5	Candy	50	Grocery	Confectionery
6	Sandwitch	20	Grocery	Cake
7	Bread	70	Grocery	Cake
8	Cookie	110	Grocery	Cake

ProductID	Name	Price	Category	SubCategory	Total price	Price running total
1	Martin 110	1,000	Vehicle	bike	2,750	**1,000**
2	Honda wave	1,200	Vehicle	motobike	2,750	**2,200**
3	T-shirt	200	Clothing	shirt	2,750	**2,400**
4	Mmen's pants	100	Clothing	trousers	2,750	**2,500**
5	Candy	50	Grocery	Confectionery	2,750	**2,550**
6	Sandwitch	20	Grocery	Cake	2,750	**2,570**
7	Bread	70	Grocery	Cake	2,750	**2,640**
8	Cookie	110	Grocery	Cake	2,750	**2,750**

```
SELECT *
        , SUM(Price) OVER() AS 'total price'
        , SUM(Price) OVER( ORDER BY ProductID ASC) AS 'price running total'
FROM Product
```

Using Window Analytic Functions *LAG and LEAD*

The two new functions **LAG** and **LEAD** are simply amazing. These functions allow you to "take a peek" at a different row. The **LAG** function lets you pull any column from a previous row. The **LEAD** function allows you to pull any column from a following row.

	transaction_id	customer_id	transaction_time	previous_time	next_time
1	1869	102	2019-01-03 23:50:12.0820000	2019-01-03 23:50:12.0820000	2019-01-04 13:31:55.8750000
2	2152	102	2019-01-04 13:31:55.8750000	2019-01-03 23:50:12.0820000	2019-01-05 14:47:08.3930000
3	2845	102	2019-01-05 14:47:08.3930000	2019-01-04 13:31:55.8750000	2019-01-06 17:26:51.5900000
4	3540	102	2019-01-06 17:26:51.5900000	2019-01-05 14:47:08.3930000	2019-01-07 16:22:20.5390000
5	4057	102	2019-01-07 16:22:20.5390000	2019-01-06 17:26:51.5900000	2019-01-07 16:52:55.3700000
6	4063	102	2019-01-07 16:52:55.3700000	2019-01-07 16:22:20.5390000	2019-01-07 16:52:15.2520000
7	4083	102	2019-01-07 16:52:15.2520000	2019-01-07 16:52:55.3700000	2019-01-08 19:43:29.1820000
8	4785	102	2019-01-08 19:43:29.1820000	2019-01-07 16:52:15.2520000	2019-01-10 23:11:20.0850000
9	6482	102	2019-01-10 23:11:20.0850000	2019-01-08 19:43:29.1820000	2019-01-11 22:40:45.2320000
10	7152	102	2019-01-11 22:40:45.2320000	2019-01-10 23:11:20.0850000	2019-01-18 17:22:24.3710000

```
SELECT transaction_id, customer_id, transaction_time
   , LAG (transaction_time, 1, transaction_time) OVER ( PARTITION BY customer_id ORDER BY transaction_id) AS previous_time
   , LEAD (transaction_time, 1, transaction_time) OVER ( PARTITION BY customer_id ORDER BY transaction_id) AS next_time
FROM fact_transaction_2019
WHERE transaction_time < '2019-02-01'
ORDER BY customer_id
```

Practice