**Instructor Notes**

Let’s look at this again. We have broken down the syntax to explain LAG and LEAD functions separately.

### **LAG function**

**Purpose**

It returns the value from a previous row to the current row in the table.

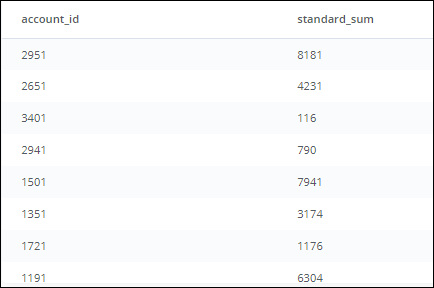
**Step 1:**

Let’s first look at the **inner query** and see what this creates.

SELECT account\_id, SUM(standard\_qty) AS standard\_sum  
FROM orders  
GROUP BY 1

**What you see after running this SQL code:**

1. The query sums the standard\_qty amounts for each account\_id to give the standard paper each account has purchased over all time. E.g., account\_id 2951 has purchased 8181 units of standard paper.
2. Notice that the results are not ordered by account\_id or standard\_qty.

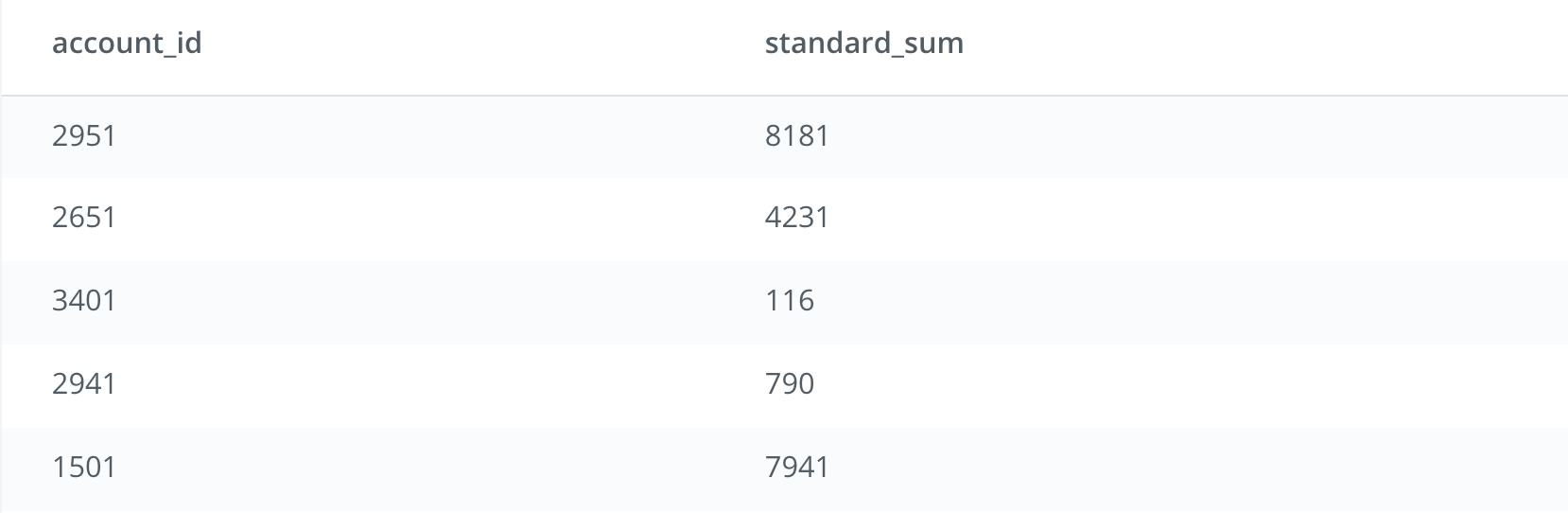


**Step 2:**

We start building the **outer query**, and name the inner query as sub.

SELECT account\_id, standard\_sum   
FROM (  
 SELECT account\_id, SUM(standard\_qty) AS standard\_sum  
 FROM orders  
 GROUP BY 1  
 ) sub

This still returns the same table you see above, which is also shown below.



**Step 3 (Part A):**

We add the Window Function OVER (ORDER BY standard\_sum) in the outer query that will create a result set in ascending order based on the *standard\_sum* column.

SELECT account\_id,   
 standard\_sum,  
 LAG(standard\_sum) OVER (ORDER BY standard\_sum) AS lag  
FROM (  
 SELECT account\_id, SUM(standard\_qty) AS standard\_sum  
 FROM orders  
 GROUP BY 1  
 ) sub

This ordered column will set us up for the other part of the Window Function (see below).

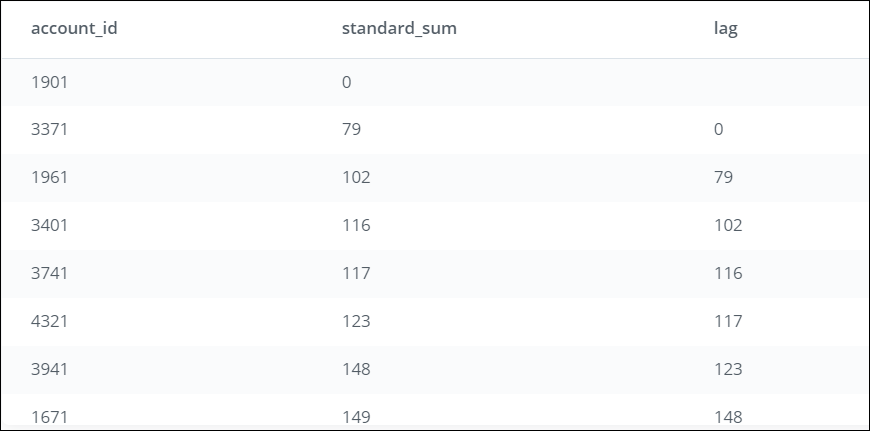
**Step 3 (Part B):**

The LAG function creates a new column called *lag* as part of the **outer query**: LAG(standard\_sum) OVER (ORDER BY standard\_sum) AS lag. This new column named *lag* uses the values from the ordered *standard\_sum* (Part A within Step 3).

SELECT account\_id,  
 standard\_sum,  
 LAG(standard\_sum) OVER (ORDER BY standard\_sum) AS lag  
FROM (  
 SELECT account\_id,  
 SUM(standard\_qty) AS standard\_sum  
 FROM demo.orders  
 GROUP BY 1  
 ) sub

Each row’s value in *lag* is pulled from the previous row. E.g., for account\_id 1901, the value in *lag* will come from the previous row. However, since there is no previous row to pull from, the value in *lag* for account\_id 1901 will be NULL. For account\_id 3371, the value in *lag* will be pulled from the previous row (i.e., account\_id 1901), which will be 0. This goes on for each row in the table.

**What you see after running this SQL code:**

****

**Step 4:**

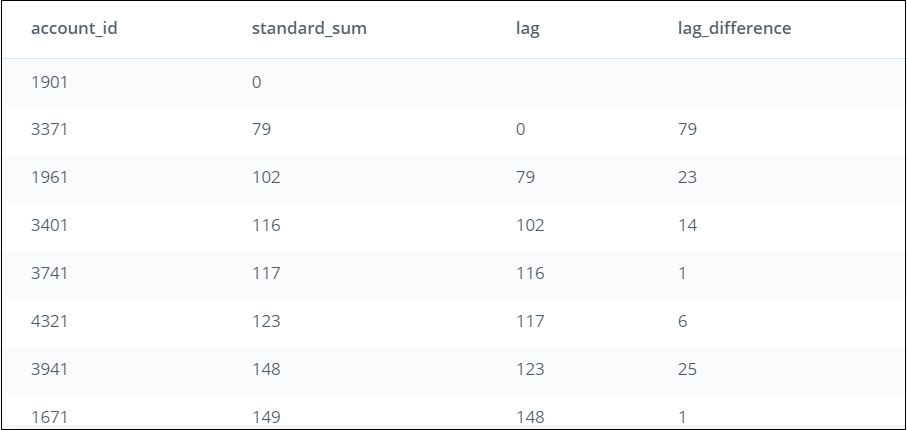
To compare the values between the rows, we need to use both columns (*standard\_sum* and *lag*). We add a new column named lag\_difference, which subtracts the *lag* value from the value in *standard\_sum* for each row in the table:

standard\_sum - LAG(standard\_sum) OVER (ORDER BY standard\_sum) AS lag\_difference

SELECT account\_id,  
 standard\_sum,  
 LAG(standard\_sum) OVER (ORDER BY standard\_sum) AS lag,  
 standard\_sum - LAG(standard\_sum) OVER (ORDER BY standard\_sum) AS lag\_difference  
FROM (  
 SELECT account\_id,  
 SUM(standard\_qty) AS standard\_sum  
 FROM orders   
 GROUP BY 1  
 ) sub

Each value in *lag\_difference* is comparing the row values between the 2 columns (*standard\_sum* and *lag*). E.g., since the value for *lag* in the case of account\_id 1901 is NULL, the value in *lag\_difference* for account\_id 1901 will be NULL. However, for account\_id 3371, the value in *lag\_difference* will compare the value 79 (*standard\_sum* for account\_id 3371) with 0 (*lag* for account\_id 3371) resulting in 79. This goes on for each row in the table.

**What you see after running this SQL code:**

****

**Now let’s look at the LEAD function.**

### **LEAD function**

**Purpose**:

Return the value from the row following the current row in the table.

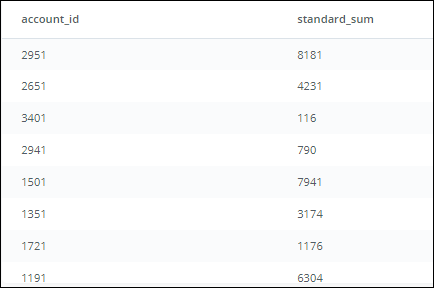
**Step 1:**

Let’s first look at the **inner query** and see what this creates.

SELECT account\_id,  
 SUM(standard\_qty) AS standard\_sum  
FROM demo.orders  
GROUP BY 1

**What you see after running this SQL code:**

1. The query sums the standard\_qty amounts for each account\_id to give the standard paper each account has purchased over all time. E.g., account\_id 2951 has purchased 8181 units of standard paper.
2. Notice that the results are not ordered by account\_id or standard\_qty.

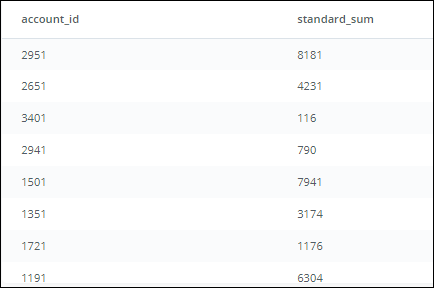


**Step 2:**

We start building the **outer query**, and name the inner query as sub.

SELECT account\_id,  
 standard\_sum   
FROM (  
 SELECT account\_id,  
 SUM(standard\_qty) AS standard\_sum  
 FROM demo.orders  
 GROUP BY 1  
 ) sub

This will produce the same table as above, but sets us up for the next part.



**Step 3 (Part A):**

We add the Window Function (OVER BY standard\_sum) in the outer query that will create a result set ordered in ascending order of the *standard\_sum* column.

SELECT account\_id,  
 standard\_sum,  
 LEAD(standard\_sum) OVER (ORDER BY standard\_sum) AS lead  
FROM (  
 SELECT account\_id,  
 SUM(standard\_qty) AS standard\_sum  
 FROM demo.orders  
 GROUP BY 1  
 ) sub

This ordered column will set us up for the other part of the Window Function (see below).

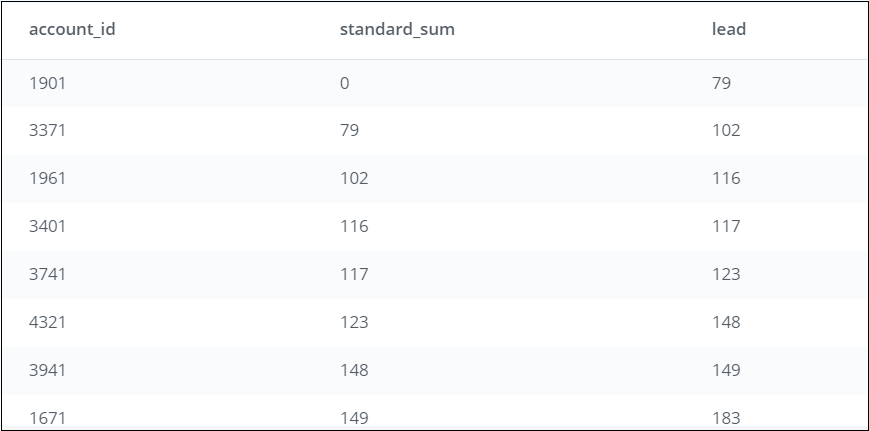
**Step 3 (Part B):**

The LEAD function in the Window Function statement creates a new column called *lead* as part of the outer query: LEAD(standard\_sum) OVER (ORDER BY standard\_sum) AS lead

This new column named *lead* uses the values from *standard\_sum* (in the ordered table from Step 3 (Part A)). Each row’s value in *lead* is pulled from the row after it. E.g., for account\_id 1901, the value in *lead* will come from the row following it (i.e., for account\_id 3371). Since the value is 79, the value in *lead* for account\_id 1901 will be 79. For account\_id 3371, the value in *lead* will be pulled from the following row (i.e., account\_id 1961), which will be 102. This goes on for each row in the table.

SELECT account\_id,  
 standard\_sum,  
 LEAD(standard\_sum) OVER (ORDER BY standard\_sum) AS lead  
FROM (  
 SELECT account\_id,  
 SUM(standard\_qty) AS standard\_sum  
 FROM demo.orders  
 GROUP BY 1  
 ) sub

**What you see after running this SQL code:**

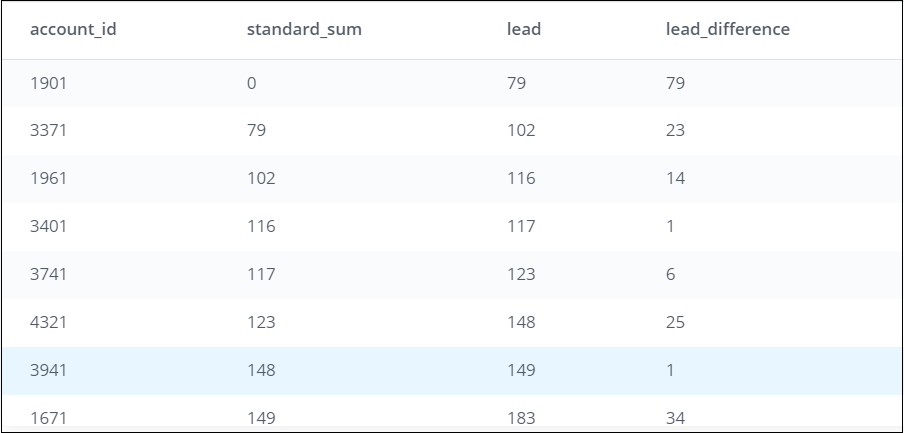
****

**Step 4:** To compare the values between the rows, we need to use both columns (*standard\_sum* and *lag*). We add a column named *lead\_difference*, which subtracts the value in *standard\_sum* from *lead* for each row in the table: LEAD(standard\_sum) OVER (ORDER BY standard\_sum) - standard\_sum AS lead\_difference

SELECT account\_id,  
 standard\_sum,  
 LEAD(standard\_sum) OVER (ORDER BY standard\_sum) AS lead,  
 LEAD(standard\_sum) OVER (ORDER BY standard\_sum) - standard\_sum AS lead\_difference  
FROM (  
SELECT account\_id,  
 SUM(standard\_qty) AS standard\_sum  
 FROM orders   
 GROUP BY 1  
 ) sub

Each value in *lead\_difference* is comparing the row values between the 2 columns (*standard\_sum* and *lead*). E.g., for account\_id 1901, the value in *lead\_difference* will compare the value 0 (*standard\_sum* for account\_id 1901) with 79 (*lead* for account\_id 1901) resulting in 79. This goes on for each row in the table.

**What you see after running this SQL code:**

****

### **Scenarios for using LAG and LEAD functions**

You can use LAG and LEAD functions whenever you are trying to compare the values in adjacent rows or rows that are offset by a certain number.

*Example 1:* You have a sales dataset with the following data and need to compare how the market segments fare against each other on profits earned.

|  |  |
| --- | --- |
| **Market Segment** | **Profits earned by each market segment** |
| A | $550 |
| B | $500 |
| C | $670 |
| D | $730 |
| E | $982 |

*Example 2:* You have an inventory dataset with the following data and need to compare the number of days elapsed between each subsequent order placed for Item A.

|  |  |  |
| --- | --- | --- |
| **Inventory** | **Order\_id** | **Dates when orders were placed** |
| Item A | 001 | 11/2/2017 |
| Item A | 002 | 11/5/2017 |
| Item A | 003 | 11/8/2017 |
| Item A | 004 | 11/15/2017 |
| Item A | 005 | 11/28/2017 |

As you can see, these are useful data analysis tools that you can use for more complex analysis!