ARTICLE 5

Further Notes on SQL Window Functions

Sippo Rossi
Hanken School of Economics

Window functions were briefly presented in Chapter 5 of *Database Systems: Introduction to Databases and Data Warehouses (Edition 3.0)* with a set of examples illustrating how they function. This article provides a more detailed visual illustration of window functions as well as a broader description of different types of window functions.

Recall that window functions are used for calculation across multiple rows that are related to the current row by using the keyword OVER to define a portion of rows (i.e., window) that should be considered as inputs to produce an output. Moreover, it is possible to display multiple different outputs based on the grouping of the rows, instead of applying one calculation to all rows, by using the keyword PARTITION BY.

Consider Query 1 (same as Query 47a in Chapter 5 of *Database Systems: Introduction to Databases and Data Warehouses [Edition 3.0]*) shown here, containing a basic window function.

Query 1:

```
SELECT      productid, productname, productprice,
            vendorid,
            AVG(productprice) OVER() AS avg_p_price
FROM        product
ORDER BY    vendorid, productid;
```

ProductID	ProductName	ProductPrice	VendorID	Avg_P_Price
2X2	Easy Boot	70	MK	112.50
3X3	Cosy Sock	15	MK	112.50
5X5	Tiny Tent	150	MK	112.50
6X6	Biggy Tent	250	MK	112.50
1X1	Zzz Bag	100	PG	112.50
4X4	Dura Boot	90	PG	112.50

Figure 1a Result of Query 1.

Visual demonstration of the functioning of a window function in Query 1, is shown in Figure 1b.

In the top part of Figure 1b we show the result of a query using only an aggregate function (avg(productprice)), which results in an output consisting of a simple row (with one value: 112.5). If we want to see the result of an aggregate function applied over multiple

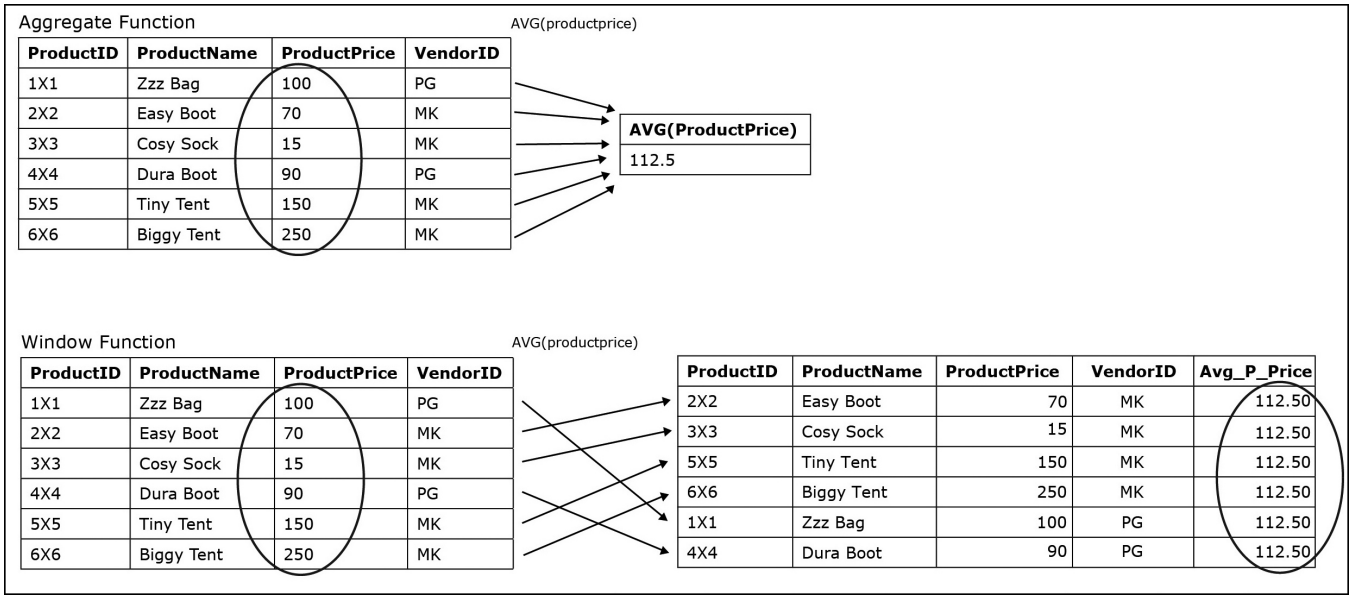


Figure 1b Aggregate functions and window functions.

rows, a window function is a convenient way to do so without needing to use a subquery, as illustrated by the bottom part of Figure 1b.

It should be noted that in window functions the value applied using OVER() does not have to be created using an aggregate function, as we will show later in this article.

Next, we will illustrate the use of PARTITION BY in window functions. Consider Query 2 (same as Query 48 in Chapter 5 of *Database Systems: Introduction to Databases and Data Warehouses [Edition 3.0]*), which calculates the average price, but instead of calculating it for all products, we use the average price of products that share the same vendor as specified by PARTITION BY.

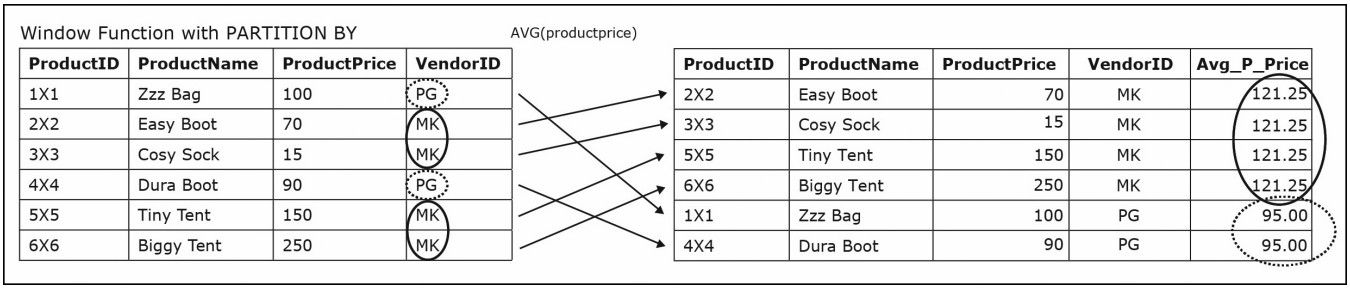
Query 2:

```
SELECT      productid, productname, productprice,
            vendorid,
            AVG(productprice) OVER (PARTITION BY
            vendorid) AS avg_p_price_per_vendor
FROM        product
ORDER BY    vendorid, productid;
```

ProductID	ProductName	ProductPrice	VendorID	Avg_P_Price_Per_Vendor
2X2	Easy Boot	70	MK	121.25
3X3	Cosy Sock	15	MK	121.25
5X5	Tiny Tent	150	MK	121.25
6X6	Biggy Tent	250	MK	121.25
1X1	Zzz Bag	100	PG	95.00
4X4	Dura Boot	90	PG	95.00

Figure 2a Result of Query 2.

Visual demonstration of the functioning of a window function in Query 2 is shown in Figure 2b, which illustrates how the AVG() function is aggregating based on VendorID, resulting in the Avg_P_Price_Per_Vendor derived column having multiple different values.



ProductID	ProductName	ProductPrice	VendorID
1X1	Zzz Bag	100	PG
2X2	Easy Boot	70	MK
3X3	Cosy Sock	15	MK
4X4	Dura Boot	90	PG
5X5	Tiny Tent	150	MK
6X6	Biggy Tent	250	MK

ProductID	ProductName	ProductPrice	VendorID	Avg_P_Price
2X2	Easy Boot	70	MK	121.25
3X3	Cosy Sock	15	MK	121.25
5X5	Tiny Tent	150	MK	121.25
6X6	Biggy Tent	250	MK	121.25
1X1	Zzz Bag	100	PG	95.00
4X4	Dura Boot	90	PG	95.00

Figure 2b Window function with PARTITION BY.

Ranking Window Functions

In Chapter 5 of *Database Systems: Introduction to Databases and Data Warehouses (Edition 3.0)*, we introduced the RANK function. An example was given (Query 49) demonstrating a way to calculate and show a rank for products based on product price from low to high. In this article, we will present an additional ranking window function, which is the ROW_NUMBER function. Other ranking window functions, such as PERCENT_RANK, NTILE, DENSE_RANKE, and CUME_DIST functions, are not covered in this article. Readers are encouraged to study more about using the various resources available online as well as in the documentation of the DBMS of their choice.

The ROW_NUMBER window function is used to assign an integer value that indicates the row number for each row. The row number is calculated by partition, thus resulting in each partition starting from row number 1 as seen in Query 3 and Figure 3.

Query 3 text: *For each product, retrieve the product ID, name of the product, price of the product, and vendor ID, and include a row number so that for each vendor the numbering starts from one.*

Query 3:

```
SELECT      productid, productname, productprice,
            vendorid,
            ROW_NUMBER() OVER (PARTITION BY vendorid)
AS          row_number_grouped_by_vendorid
FROM        product;
```

ProductID	ProductName	ProductPrice	VendorID	Row_Number_Grouped_By_VendorID
2X2	Easy Boot	70	MK	1
3X3	Cosy Sock	15	MK	2
5X5	Tiny Tent	150	MK	3
6X6	Biggy Tent	250	MK	4
1X1	Zzz Bag	100	PG	1
4X4	Dura Boot	90	PG	2

Figure 3 Result of Query 3.

Instead of using PARTITION BY, it would also be possible to use the ORDER BY followed by specifying a column, and then the numbering would be done based on the column with all rows included without partitioning. This is demonstrated in Query 4 and Figure.4.

Query 4 text: *For each product, retrieve the product ID, name of the product, price of the product, and vendor ID, and include a row number ordered by vendor ID.*

Query 4:

```
SELECT      productid, productname, productprice,
            vendorid,
            ROW_NUMBER() OVER (ORDER BY vendorid) AS
            row_number
FROM        product;
```

ProductID	ProductName	ProductPrice	VendorID	Row_Number
2X2	Easy Boot	70	MK	1
3X3	Cosy Sock	15	MK	2
5X5	Tiny Tent	150	MK	3
6X6	Biggy Tent	250	MK	4
1X1	Zzz Bag	100	PG	5
4X4	Dura Boot	90	PG	6

Figure 4 Result of Query 4.

Lastly, it is possible to combine both the PARTITION and ORDER BY command in one query, as is illustrated in Query 5 and Figure 5.

Query 5 text: *For each product, retrieve the product ID, name of the product, price of the product, and vendor ID, and include a row number ordered by product price separately for each vendor.*

Query 5:

```
SELECT      productid, productname, productprice,
            vendorid,
            ROW_NUMBER()
            OVER (PARTITION BY vendorid ORDER BY
            productprice)
            AS item_number_by_vendor
FROM        product;
```

ProductID	ProductName	ProductPrice	VendorID	Item_Number_By_Vendor
3X3	Cosy Sock	15	MK	1
2X2	Easy Boot	70	MK	2
5X5	Tiny Tent	150	MK	3
6X6	Biggy Tent	250	MK	4
4X4	Dura Boot	90	PG	1
1X1	Zzz Bag	100	PG	2

Figure 5 Result of Query 5.

Value Window Functions

Value window function can be used to compare the current row to the value of another row, such as the value in the first, last or nth row in a table, using the keywords FIRST_VALUE, LAST_VALUE or NTH_VALUE. Comparing to the first or last value makes sense when the table is ordered in a sensible manner. For instance, if a table is ordered by total sales volume by customer, with value window functions, it is easy to compare the difference between any individual customer and the customer who has spent the most (or least). Another possible use case is when wanting to compare the current product to the most expensive product within the same product category. This latter example is demonstrated in Query 6 and Figure 6.

Query 6 text: *For each product, retrieve the product ID, name of the product, category ID, price of the product, and difference between the price of this product and the most expensive product within the same product category.*

Query 6:

```
SELECT      productid, productname, categoryid,
            productprice,
            productprice - FIRST_VALUE(productprice)
            OVER (PARTITION BY categoryid ORDER BY
            productprice DESC)
            AS dif_to_most_expensive_in_category
FROM        product;
```

ProductID	ProductName	CategoryID	ProductPrice	Dif_to_Most_Expensive_in_Category
6X6	Biggy Tent	CP	250	0
5X5	Tiny Tent	CP	150	-100
1X1	Zzz Bag	CP	100	-150
4X4	Dura Boot	FW	90	0
2X2	Easy Boot	FW	70	-20
3X3	Cosy Sock	FW	15	-75

Figure 6 Result of Query 6.

LAG and LEAD Window Functions

As the last part of this article, we will discuss the **LAG** and **LEAD** functions. These functions can be used to create a column that is calculated from another column, using rows before or after the current row. It is important to note that when using LAG and LEAD, the rows must represent data that is ordered, for example by time or by some other reasonable attribute. One example of a situation where this functionality is particularly useful is when the goal is to calculate differences between rows. Depending on which function is used, the first or last element in each partition will be null, since there is no row to compare the value to.

The use of LAG is demonstrated in Query 7, and the result of the query is shown in Figure 7. Here we calculate a new column with information on the difference between the price of the product and the next most expensive product in the same category.

Query 7 text: *For each product, retrieve the product name, category ID, product price and difference between this product's price and the next most expensive product's price within the same product category. Within the category ID, sort by product price, in descending order.*

Query 7:

```
SELECT      productname, categoryid,
            productprice, productprice -
            LAG(productprice, 1)
            OVER (PARTITION BY categoryid ORDER BY
            productprice)
            AS dif_to_next_cheaper_prod_in_category
FROM        product
ORDER BY    categoryid, productprice DESC
```

ProductName	CategoryID	ProductPrice	Dif_to_Next_Cheaper_Prod_in_Category
Biggy Tent	CP	250	100
Tiny Tent	CP	150	50
Zzz Bag	CP	100	null
Dura Boot	FW	90	20
Easy Boot	FW	70	55
Cosy Sock	FW	15	null

Figure 7 Result of Query 7.

Conversely, if we were to use the LEAD function instead of LAG, with minor adjustments to the SQL query it is possible to compare each row to a row above. Moreover, it is possible to create multiple new columns with window functions for example by simply using LEAD more than once within the same query. This is demonstrated in Query 8 and Figure 8.

Query 8 text: *For each product, retrieve the product name, product price, price of the product that is one above the current product in price, and difference between these two prices, considering products within the same product category. Within the category ID, sort by product price.*

Query 8:

```
SELECT      productname, productprice, categoryid,
            LEAD(productprice)OVER
            (PARTITION BY categoryid ORDER BY productprice)
            AS price_of_next_more_expensive_product_in_category,
            LEAD(productprice) OVER
            (PARTITION BY categoryid ORDER BY productprice) - productprice
            AS dif_to_next_more_expensive_prod_in_category
FROM        product
ORDER BY    categoryid,productprice;
```

ProductName	ProductPrice	CategoryID	Price_of_Next_More_Expensive_Prod_in_Category	Dif_to_Next_More_Expensive_Prod_in_Category
Zzz Bag	100	CP	150	50
Tiny Tent	150	CP	250	100
Biggy Tent	250	CP	null	null
Cosy Sock	15	FW	70	55
Easy Boot	70	FW	90	20
Dura Boot	90	FW	null	null

Figure 8 Result of Query 8.

There are two additional parameters for LAG and LEAD that can be adjusted: the offset and default value. The former is used to determine how many rows are skipped if one wishes to compare a row that is more than one step above or below the current row. With the default value parameter, it is possible to set a different value instead of null when there is no element with which to compare.

In conclusion, window functions provide a powerful set of tools for calculations across rows in relation to the current row. They are particularly valuable when the goal is to perform calculations, such as those in aggregate functions, while maintaining outputs at a row-level. Moreover, windows functions provide a convenient way to produce columns that allow comparing or viewing data in a ranked format when using the value window functions or LAG and LEAD clauses.