**Analytic Functions**

Analytic functions often appear in the SELECT clause and are typically the last to be processed

PARTITION BY

We can use the PARTITION BY clause with the OVER clause to specify the *column* on which we need to perform aggregation

This completes aggregation for each row in a specified column

* PARTITION BY divides a query’s result set into *partitions*
* The aggregate is operated on each partition *separately* and *recalculated* for each partition

If you want the analytic function to go over the entire table, then can do:

COUNT (\*) OVER ( ) AS “Total Count”

It is worth understanding the difference between GROUP BY and PARTITION BY:

|  |  |
| --- | --- |
| **GROUP BY** | **PARTITION BY** |
| * Often used in conjunction with an aggregate function * Reduces the number of rows returned by rolling them up and calculating the sums or averages for each group * **Aggregate** | * Does not reduce the number of rows * Divides the result set into partitions and changes how the window function is calculated * **Analytic** |

PARTITION BY also provides the benefit of not having to process a JOIN to achieve the same information, namely when creating an aggregate

A close up of a white background

Description automatically generatedThis counts the number of employees that have participated in an order, and uses two tables without having to do a JOIN

A close-up of a computer screen

Description automatically generatedThis query calculates the sum of units on order for each category, a join is required as the aggregate function and PARTITION BY require different tables

ORDER BY

Tells the data how to order it within the PARTITION BY clause

* Input the column that you want to be ordered, and whether this is ASC or DESC

ROW NUMBER, RANK, & DENSE\_RANK

ROW NUMBER () returns the row number when data ordered according to ORDER BY statement

* ORDER BY can be ascending or descending
* Identical values ordered arbitrarily
  + RANK() and DENSE\_RANK() will assign the same number for ties
  + RANK() skips after ties
  + DENSE\_RANK() doesn’t skip
* Independent of the ORDER BY clause in the *main query*

A close up of a logo

Description automatically generatedThis returns the row number with the data being ordered by UnitPrice in DESC order, highest price at the top

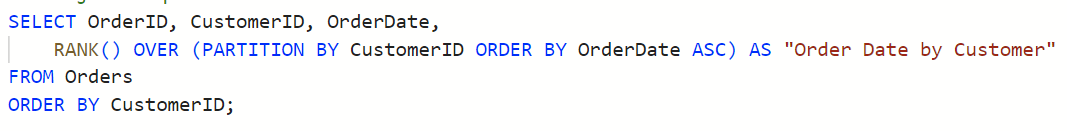
A screenshot of a computer

Description automatically generated

PARTITION BY and ORDER BY can be used in *conjunction*

* ORDER BY determines the *order of processing* within each partition

The analytic being applied will restart whenever a partition boundary is reached

A screenshot of a computer

Description automatically generatedThis query looks at the date orders of customers in ascending order, the order date is checked for each customer until it reaches the next one in which the rank returns to 1

LEAD & LAG

**LEAD( )** Looks ahead a certain number of rows

It takes 3 arguments:

1. Column to return
2. Number of rows to *look forward*
3. Default value if nothing can be returned

**LAG ( )** works in the same way, but looks *behind*

*A screen shot of a computer

Description automatically generated*This code looks 1 ahead, as specified in the second argument, the default value returned if nothing can be is set to NULL

Remember LEAD( ), FIRST\_VALUE( ), ROW\_NUMBER( ) , etc. are separate keywords and so the brackets should be closed before reaching the OVER clause

*A close-up of a sign

Description automatically generated*This looks at the freight for a previous order as it uses the LAG( ) keyword

FIRST & LAST\_VALUE

**FIRST\_VALUE( )and LAST\_VALUE**  will return the first row based on the ORDER BY, for each partition

* It takes *one argument*, the column *name* to return from the first or last row
* FIRST\_VALUE returns the first value, whilst LAST\_VALUE returns the last value

A close-up of text

Description automatically generatedThis outputs the amount of orders for each supplier as it is PARITIONED BY SupplierID

* Remember, the calculation restarts after it moves on to the next supplier

A computer screen shot of text

Description automatically generatedThis query calculates the difference between the customers first order and the current date by using DATEDIFF

* See that all the code in the DATEDIFF clause is within two parentheses
* The first date is obtained using the FIRST\_VALUE keyword which is PARTITIONED BY CustomerID
* It takes the FIRST\_VALUE of the OrderDate of a CustomerID and outputs the difference between that and the GETDATE( )

WINDOW CLAUSE

Analytical functions include a “**Window Clause’**, which needs to be specified

The default value is:

* Rows between UNBOUNDED PROCEDING and current row

There are several terms that of use here:

* UNBOUNDED PROCEDING 🡪 all rows *before* the current row = default
* UNBOUNDED FOLLOWING 🡪 all rows *after* the current row = default
* X PRECEDING 🡪 x rows *before* the current row
* Y FOLLOWING 🡪 y rows *after* the current row

Simply put, it is a “frame” or “range” clause of window functions

This can be useful for creating “Running totals” or “Moving averages”

Along with the PARTITION BY and ORDER BY clauses, there is also ROW and RANGE

* The default means ‘run the analytic function on all rows between UNBOUND PROCEDING and the current row
* So, by default, LAST\_VALUE only looks between the start of the partition and the current row

A row of rows and rows

Description automatically generatedThe window must be specified, it follows this format:

When specified, the analytical function now considers the *entire partition* before returning the LAST\_VALUE**A close-up of a white background

Description automatically generated**

**A close-up of a computer screen

Description automatically generated**This new code resolves the previous problem as the rows that the query is calculating has been specified

* ROWS BETWEEN UNBOUNDED PROCEDING AND UNBOUNDED FOLLOWING simply means that it looks at all rows *before* and *after* the current row
* The previous code did not work because it only looked at all the previous rows and the current row, and so could not obtain the last order placed by an employee

By specifying the number of preceding and following rows, we can calculate a **MOVING AVERAGE**

A moving average is calculated by taking the average of a set of values *over a specific period of time*

* The type of moving average is depending on the period e.g., a seven-day moving average, 3 month moving average, etc.
* You may also refer to it as a 3-point moving average i.e., averages from three points of time

A close up of text

Description automatically generatedThis is a 3-point moving average because the window clause specifies that the query looks all rows between the prior row and the following row – three points

STATISTICAL FUNCTIONS

* **CUME\_DIST() OVER (ORDER BY x)**🡪 shows cumulative relative distance – the numbers of rows with values *less than* and *equal* to the current row’s *value*, divided by the *total number of rows*
* **PERCENT\_RANK() OVER (ORDER BY x)**🡪 computes the rank as a proportion of the *total number of rows*
* **PERCENTILE\_DISC(A) WITHIN GROUP (ORDER BY x) OVER ( )**🡪 gives the value matching (or higher than if none match) the specified *percentile* the inverse of CUME\_DIST. Only returns values that exist
* **PERCENTILE\_CONT(a) WITHIN GROUP (ORDER BY x) OVER ( )** 🡪 performs a PERCENTILE\_DISC but will interpolate if the specified percentile can’t be matched exactly

**X = Column(s)**

**A = Decimal between 0 and 1**

WITH

Makes SQL queries neater

* Prevent repeated execution of subqueries
  + Define *once*, then use *multiple times*
* Known as **Common Table Expressions (CTE) in SQL Server**
  + Subquery Factoring Clause in Oracle
* Create temporary tables to be used in the rest of the query
* Removed after query execution

WITH clauses define subqueries *before* the main query

Subqueries can be chained:

* WITH cte1 AS (subquery)

Subqueries can reference previous subqueries in the WITH clause, allowing a chain of sophisticated transformations

CTEs are stored in *temporary workspace* until the query has finished executing, then they are removed

A screenshot of a computer

Description automatically generatedIn this example the WITH clause is creating a temporary table with the average unit price per category for only those above 30

This is then later referenced in the main query by using the table alias that was created, a JOIN is created between the two tables

TEMPORARY TABLES

If you want a temporary table to exist *after* query execution, **do not use CTE**

* Temporary tables always begin with #
* They are stored in tempdb
* Only accessible to the connection that created them
  + Unless made global with ##
* Automatically deleted when connection closed
  + Or using DROP TABLE

**A screenshot of a computer

Description automatically generated**

* The first query *defines* and *stores* them temp table
* The second query *uses* it
* If we wanted to make it globally available, we’d use ##MeanAbove30
* It will be dropped when we disconnect or call:
  + DROP TABLE #MeanAbove30

The way to create a temporary table is to use the INTO clause after the SELECT clause, this is where you name your table and make sure to put a #

A close-up of a computer screen

Description automatically generated

UNPIVOT & PIVOT

UNPIVOT Moves data from a *wide format to a long format*

It requires an aggregate function, in case more than one value is present

A close up of text

Description automatically generatedThis query means that rather than phone and fax being in separate columns (thus wide format), they are in the same column with another column that outputs the phone/fax number itself

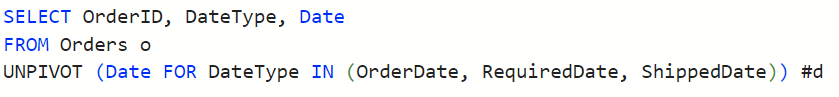
* The UNPIVOT must have an alias at the end

A screenshot of a computer

Description automatically generatedPIVOT does the opposite, moves data from a *long format* to a *wide format*

This also requires an aggregate function, in case more than one value is present

The format for the PIVOT/UNPIVOT clause is:

The columns are defined in the SELECT clause and then the data that it is in them is defined in the IN clause of the UNPIVOT keyword

A screen shot of a computer

Description automatically generatedThis example shows a table being unpivoted (long 🡪 wide) and then pivoted (wide 🡪 long), PIVOT is the opposite of UNPIVOT and the code follows this

The breakdown of PIVOT functions:

1. SELECT 🡪 SELECT what the new columns will be called
2. FROM 🡪 the table the data is coming from + will be put into
3. UNPIVOT/PIVOT **🡪** this slightly differs for both:
   1. **PIVOT** 🡪 (column1 FOR column2 IN (data1, data2, etc)) table alias
   2. **UNPIVOT 🡪** (MAX(data1) FOR data2 IN (column1, column2, etc.)) table alias