**A Little ROW\_NUMBER Magic**

by [Ben Taylor](https://www.sswug.org/author/bentaylor/)

FROM: <https://www.sswug.org/bentaylor/editorials/a-little-row_number-magic/>

**[](https://www.sswug.org/membership-account/membership-levels/)**

**[](https://www.surveymonkey.com/r/K5ZJ8VH)**

I really like the ROW\_NUMBER() function in SQL Server as a means of getting the first or last instance of a record for each specific group within record set containing more than one group. Ok, let me break that down a little bit.

It’s pretty easy to get the last record of the line items for a single purchase order. But, what if you wanted to get the last line item for all purchase orders? You can; they are all in the same table. But, without using ROW\_NUMBER() the task is a little more complicated. Since this editorial is about ROW\_NUMBER, I won’t take the time to show you how we used to do it.

For my example today I am using the EmployeePayHistory table found in the AdventureWorks2012 database, because it demonstrates the problem, and I have a copy of that database handy. The table has BusinessEntityID, RateChangeDate, Rate, PayFrequency and ModifiedDate columns. The first thing I wanted to do was to determine if there were any groups within that table having multiple records. To find that out, I did a GROUP BY query on BusinessEntityID.

SELECT BusinessEntityID, Count(1) as Records

FROM AdventureWorks2012.HumanResources.EmployeePayHistory

GROUP BY BusinessentityId

HAVING count(1) > 1

This query returned 13 BusinessEntityID records, each having 3 unique entries. I wanted to see what those specific records looked like, so I used the results of the previous input in a new query, joining back to the detail table to get the records for only those 13 BusinessEntities.

WITH MyEntities

AS

(

SELECT BusinessentityId, Count(1) as Records

FROM AdventureWorks2012.HumanResources.EmployeePayHistory

GROUP BY BusinessentityId

HAVING count(1) > 1

)

SELECT eph.\*

FROM MyEntities e

JOIN AdventureWorks2012.HumanResources.EmployeePayHistory eph

ON e.BusinessEntityID = eph.BusinessEntityID

ORDER BY eph.BusinessEntityID, eph.RateChangeDate

Following is a table with the first BusinessEntityID

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BusinessEntityID** | **RateChangeDate** | **Rate** | **PayFrequency** | **ModifiedDate** |
| 4 | 2002-01-05 | 8.62 | 2 | 2001-12-22 |
| 4 | 2004-07-01 | 23.72 | 2 | 2004-06-16 |
| 4 | 2006-01-15 | 29.8462 | 2 | 2006-01-01 |

The same kind of history was demonstrated for the other 12 Business Entities. I now want to write a query that returns only one record for every Business Entity in the detail table, regardless of how many records may be present for a single Business Entity. Using ROW\_NUMBER() it is easy to get the first or last record for each individual Business Entity. In my query I want to get the last record for each BusinessEntityID based on the RateChangeDate. From the table above, I want to get the third record, having the last RateChangeDate.

WITH MyPay

AS

(

SELECT BusinessentityId

,RateChangeDate

,Rate

,PayFrequency

,ModifiedDate

,ROW\_NUMBER() OVER

(

PARTITION BY BusinessEntityID

ORDER BY RateChangeDate DESC

) as RowId

FROM AdventureWorks2012.HumanResources.EmployeePayHistory

)

SELECT BusinessentityId

,RateChangeDate

,Rate

,PayFrequency

,ModifiedDate

FROM MyPay

WHERE RowId = 1

This query returns the last record for each Business Entity, based on the RateChangeDate. I enforce this behavior by Partitioning by each BusinessEntityID. Using this partition causes the ROW\_NUMBER function to start counting from 1 each time the BusinessEntityID changes. In the ROW\_NUMBER function I also sort the records with the ORDER BY RateChangeDate DESC clause, so the data is sorted by the latest to the oldest RateChangeDate. I wrap all of that magic in a CTE I call MyPay, which I may then use to write my query, returning only the records I want with the where clause WHERE RowId = 1, the value of my ROW\_NUMBER function in the CTE.

Your final results are now 290 records in my database, 26 less than the total number of records in the table.

There you have it, a way to get max out of a group within a set of groups.

Cheers,

Ben

**ROW\_NUBER vs. GROUP BY**

by [Ben Taylor](https://www.sswug.org/author/bentaylor/)

FROM: <https://www.sswug.org/bentaylor/editorials/row_nuber-vs-group-by/>

**[](https://sqllive360.com/ECG/live360events/Events/Orlando-2016/Information/SQLLive.aspx?utm_source=AttendeeMktg&utm_medium=Banner%20Ad&utm_campaign=UGOR14)**

**[](https://www.surveymonkey.com/r/K5ZJ8VH)**

**[](http://www.sswug.org/what-is-a-pro-membership/)**

Last Thursday I published an editorial demonstrating one use of the ROW\_NUMBER function as a technique to get the last value for each group contained within a set of groups, [**A Little ROW\_NUMBER Magic**](https://www.sswug.org/bentaylor/editorials/a-little-row_number-magic/). See the example below in figure 1 as executed against my instance of AdventureWorks2012.

Figure 1 – ROW\_NUMBER Grouping Query

WITH MyPay

AS

(

SELECT BusinessentityId

,RateChangeDate

,Rate

,PayFrequency

,ModifiedDate

,ROW\_NUMBER() OVER

(

PARTITION BY BusinessEntityID

ORDER BY ModifiedDate DESC

) as RowId

FROM AdventureWorks2012.HumanResources.EmployeePayHistory

)

SELECT \*

FROM MyPay

WHERE RowId = 1

Eilenblogger posted a comment with a different solution, which was the method we used before ROW\_NUMBER was made available in SQL Server 2005. I modified his example slightly so that the difference is easily visible. I’ll call this the GROUP BY example for clarification.

Figure 2 – GROUP BY Query

WITH MyPay

AS

(

SELECT BusinessentityId

,Max(RateChangeDate) as RateChangeDate

FROM AdventureWorks2012.HumanResources.EmployeePayHistory

GROUP BY BusinessEntityID

)

SELECT \*

FROM MyPay

JOIN AdventureWorks2012.HumanResources.EmployeePayHistory eph

ON MyPay.BusinessEntityID = eph.BusinessEntityID

AND MyPay.RateChangeDate = eph.RateChangeDate

We both reviewed the query plan for either query, and it made sense that the group by method should be the faster in performance. I had done that comparison back in 2005 and found that ROW\_NUMBER performed best for the scenarios I was testing. As promised in the comments, I did go back and do a comparison for performance between the two implementations. This editorial contains the results of that reseach. Because of limited space, I am summarizing the results.

Even though the query plans were considerably different, I was not able to get any measurable difference between the two implementations. I compared CPU execution time, total duration, and read count by turning on the advanced options in SSMS for SET STATISTICS TIME and SET STATISTICS IO on. Running multiple executions of both queries were essentially the same.

I then tried the same thing by changing the Max Degrees of Parallelism and testing performance multiple times. Again, both queries were essentially comparable. This left me still wanting more. So, I added some records to the table being queried. Originally the table had only 324 rows, and returned only 290 rows when our query returned only one row per BusinessEntityID. I added records to the table so that the number of unique BusinessEntitityID values was still 290, but had nearly 2 million records. That meant that the query then needed to find the last record per BusinessEntityID out of a list of nearly 2 million records, returning only 290 records.

With these additional records, I finally had something that I could measure. The GROUP BY implementation was far superior, regardless of Max Degrees of Parallelism. It consistently ran at about 500ms. The ROW\_NUMBER implementation ran at 4,000ms+. Changing Parallelism did reduce performance time to 3,500ms.

Another thing you might look into for something like this would be indexing. In this case, the best index was already part of the primary key of the table. The table had a unique primary key of BusinessEntityID and RateChangeDate. So, it seemed like there may be some unfair advantage to the GROUP BY implementation. To test this theory I modified both queries to use the ModifiedDate column instead of RateChangeDate, because my table didn’t have any index for ModifiedDate. Even with 2 million records, the results were virtually the same.

Here are some things I take away from this exercise

* The SQL Server engine has changes from one version to the next. Things that may have been true in one version may not be true in another version
* The size of tables (columns and number of rows), indexes available, and parallelism can make a difference in performance
* If you have a query that runs good now, it may not run well later if the data changes dramatically
* Don’t be afraid to challenge things that you have proven in the past

Cheers,

Ben