

Aggregate Product Function extends SQL

Solution extends the capability of standard SQL by adding aggregate Product Function

By Dr. Alexander Bell

Disclaimer:

This article and all supplemental materials are provided on AS IS basis without warranty of any kind.



Technology: Access 2000 /XP /2003 /2007, SQL



Article contains: 4 Mathematical formulas
6 Code Listings (SQL statements)
4 Tables
2 Figures (screen shots)



Downloads: Microsoft Access database file with sample data Tables and Queries

Article demonstrates how the entire "Pure SQL" solution could be encapsulated into a single MS Access Query, portable to many other SQL-backed RDMB. Several approaches, which differ in terms of their performance-universality, are presented and discussed in details. Practical implementation and "real-life" samples are included in the supplemental Microsoft Access Database file.

TABLE OF CONTENTS

- [1. Key concept and definitions](#)
- [2. Programming Technique: Math-to-SQL translation](#)
- [3. Real-life example: investment performance calculation](#)
- [4. SQL Aggregate Product function: Universal solution](#)

1. KEY CONCEPT AND DEFINITIONS

In a modern data-centric, Service-Oriented (SOA) applications central Databases typically host a great deal of business logic, written in SQL. Pure SQL solutions have many practical advantages: they are native to any Database and as such, they are highly optimized and extremely effective in dealing with regular data sets, they are portable between the varieties of commercially available databases and are mostly independent on the OS platform.

As SQL was in place for several decades, then its security vulnerability issues were pretty much addressed. Alternative to data-centric solutions, the ones which involve in addition to SQL some other programming language (like C/C++, C#, VB, Java, etc.) could potentially raise the portability, performance, maintainability and application security issues; many of them are platform dependent, some of them are less efficient (especially those which do not produce the native compiled code) and any of them will inevitably add more security concerns and limitations because of their own (known and unknown) security vulnerabilities.

It's also relevant to mention that the application maintenance and upgrade are less difficult when its core solution is mostly based on a single highly standardized and portable language such as SQL instead of being programmed with extended set of different (and, in many cases, platform dependent) programming languages. Having all these considerations listed above, SQL is recommended to be the first choice in building the core of any primarily data-centric application.

Standard SQL, either Jet or T-SQL, contains several aggregate functions (Sum, Count, Min, Max, etc) with pretty noticeable lack of aggregate Product. Just as a reminder, a Product function \prod of multiple arguments (X_1, X_2, \dots, X_N) is defined as:

$$\prod_{i=1}^N (X_i) = X_1 * X_2 * \dots * X_N \dots \dots \dots (1)$$

The alternative to SQL query solution could use VBA and Recordset (DAO or ADO) object, but that approach put additional programmatic overhead to the task and raise a portability issue because VBA is not an option in RDBM other than Microsoft Access. For portability reason it is much more desirable to find the solution which allows to stay just within the SQL as it is almost universally presented in the most popular Databases currently existing on the market.

Database engine cannot perform the aggregate product calculation directly, but it can calculate sums. Simple mathematical transforms, demonstrated in Formula (1) provides a workaround: converting values into Logs, adding them and applying exponential function Exp to the result will allow product calculation by using standard SQL built in Jet database engine. Simple mathematical transforms provide a workaround enabling to compute the product \prod using the standard built-in mathematical Log(), Exp() and SQL aggregated Sum() functions; the core technique is based on the mathematical transforms, shown in (2) and (3):

$$\text{Log}(X_1 * X_2 * \dots * X_N) = \text{Log}(X_1) + \text{Log}(X_2) + \dots + \text{Log}(X_N) \dots\dots\dots(2)$$

$$\prod_{i=1}^N = \text{Exp} \left(\sum_{i=1}^N (\text{Log}(X_i)) \right) \dots\dots\dots(3)$$

Formula (3) could be programmatically translated into equivalent SQL statement in a rather straightforward manner, enabling the calculation of aggregate Product by means of standard built-in SQL features.

There is a certain practical limitation of using formulas (2) and (3): they provide an accurate solution applicable to any set of positive non-zero values (X_i), but this solution cannot be applied directly to the negative values and zero (mathematical 0). The issue is rooted in the properties of a Log(X_i) function, which does not provide a real solution for negative numbers, plus, theoretically it should return a negative infinite value for zero. While negative infinite is a quite common mathematical abstraction, it presents a programming challenge when translating into machine code. In order to extend formula (3) to cover the full range of positive, negative and zero-equal values, couple additional mathematical transforms must apply, namely:

- All negatives must be converted to positive numbers: standard Abs(X_i) function can do the job;
- The sign of the final expression (aggregate Product) must be defined. It could be done, for example, by counting of all negatives (negative values); then the assignment should be made based on a simple rule: “+” sign will be assigned to the final expression for even and “-“ for odd count of negatives.
- The solution should implement a special handling of zeros. Quite ironically, though this case has rather simple logical solution (no computation actually needed: resulting Product must be 0 regardless of other row values) it causes some programmatic overhead, added to the SQL statement. There are couple solutions to this issue, which are discussed in greater details later in the article: one, though rather simple will cause additional methodical error rooted in the difference between “true negative infinite” in a pure mathematical sense and its approximate machine representation. Second solution will provide more accurate result by using the aggregate Min() function to detect zeros and, in case there any – to assign a zero value to the Product function output.

Further in the article you will find detailed discussion of various SQL solutions, which differ in terms of their performance and applicable area, accompanied by several practical examples of aggregate Product calculation.

2. PROGRAMMING TECHNIQUE: MATH-TO-SQL TRANSLATION

2.1. Simplified Aggregate Product of all positive numbers

This simple yet practical example will demonstrate the SQL programming technique enabling to calculate the Product of all positive numbers, stored in a Microsoft Access Table1 (see Fig.1). Table1 has a single column Num and 5 rows containing all positive values.

Table1. Sample set of positive numbers in a single Column "Num"	
Num	
2	
4	
5	
7	
8	

Based on the precondition that there are no any negative values in the Table1 rows, rather simple SQL query can do the job of calculating Product (see corresponding SQL statement shown in the Listing 1):

LISTING

1

SELECT Exp(Sum(Log([Num]))) AS P FROM Table1

Just run this query to get the accurate result of 2240 ($2 \times 4 \times 5 \times 7 \times 8 = 2240$).

The statement shown in Listing 1 is a direct SQL equivalent of the formula (3); it does not include any exception handling caused by negatives or zeros, so any of such values entered in Table1 will cause the SQL procedure to fail with error message displayed as shown on Figure 1 (the explanation for this issue was already given in a previous discussion dedicated to the property of Log() function)



Figure 1. Error message appears as a result of an attempt to calculate the Log(0) value in SQL query

The statement could be modified with **Iif()** conditional operator added in order to handle zeros (see Listing 2):

LISTING

2

SELECT Exp(Sum(Log(Iif([Num]=0,10^-306,[Num]))) AS P FROM Table1

The logic behind this statement is as follows: any time zero value is found the system instead of trying to calculate the Log(0) will replace zero value with extremely small positive number, for e.g. $10^{(-306)}$, close to machine 0. This simple approach has certain deficiency due to methodical error; the aggregate Product output is not exactly 0, moreover the actual result depends on other values in the rows. In many

practical cases this error is relatively small and could fit the acceptable error budget. For more accurate calculation there is another solution with improved zero-value handling achieved through the use of conditional IIf() operator (see the corresponding SQL statement shown in the Listing 3):

LISTING

3

```
SELECT Exp( Sum(IIf( [Num]=0,0,Log( [Num] ))) * IIf( Min( [Num] )=0,0,1) AS P  
FROM Table1
```

Additional multiplier IIf(Min([Num])=0,0,1) is added to the SQL statement in order to accurately “nullify” (set to 0) the return value of aggregate Product function in case at least one zero value was detected in any row. The first conditional operator is also modified, excluding 0 values from further processing (otherwise they will be converted to a small number and then passed to a Log() function). This measure allows increasing the overall computational efficiency; performance boost could be quite noticeable in case there are multiple rows in the Table1 containing zeros.

3. REAL-LIFE EXAMPLE: INVESTMENT PERFORMANCE CALCULATION

This simple example related to the investment performance calculation will serve both practical and didactical purposes, demonstrating the application of the SQL aggregate Product programming technique, discussed in the previous chapter to the real-world mathematical problem. Based on the commonly accepted practice, the compound annual performance (ANNUAL_PFM) of the financial investment portfolio could be calculated on the basis of its monthly performance array PFM_i (i=1...12) as following:

$$\text{ANNUAL_PFM} = (1+\text{PFM}_1) * (1+\text{PFM}_2) * \dots * (1+\text{PFM}_{12}) - 1 \dots \dots \dots (4)$$

Similar approach is applicable to the Quarterly Performance (Q_PFM), Semi-Annual, 3-, 5- and 10 years performance calculation, etc. Programmatically, all this type of compound investment performance calculations could use the same Aggregate Product function and differ only by CROUP BY clause. Following two examples will demonstrate this technique in regards to Annual and Quarterly performance calculation. It's relevant to mention that similar technique is also applicable to the accrued compound interest calculation (with reinvestment) and many other fixed income instruments analytics.

Sample Table 2 shown contains some hypothetical XYZ portfolio monthly performance data entered for two consequent years, 2004 and 2005 (please notice, that all data in this article and corresponding downloads is shown for demonstration purpose only and does not represent any actual financial information).

Table 2. Virtual “XYZ” Investment Portfolio monthly Performance		
Year	Month	PFM
2004	1	0.03
2004	2	0.02
2004	3	0.04
2004	4	0.01
2004	5	-0.01
2004	6	-0.02
2004	7	-0.07
2004	8	-0.01
2004	9	0.03
2004	10	0.04

2004	11	0.02
2004	12	0.01
2005	1	0.02
2005	2	-0.04
2005	3	-0.02
2005	4	0.01
2005	5	0.02
2005	6	0.01
2005	7	0.03
2005	8	0.03
2005	9	0.05
2005	10	0.07
2005	11	0.05
2005	12	0.02

Invested performance is typically presented in a percent; here the numbers shown are stored as a relative values, provided that the conversion to percent could be done later on a different layers (mid-tier or presentation). In a business sense, positive numbers reflect monthly capital gain, negative numbers, correspondingly, capital losses. This example is chosen because of one rather important application specific feature: typically, monthly losses for the investment portfolios do not exceed 100%, which mathematically means that $(PFM_i) \geq -1$ and correspondingly: $(1+PFM_i) \geq 0$. It is quite a distant probability, that monthly losses could reach 100%, which means a financial disaster (all invested capital is lost), but please be aware, that theoretically monthly losses could reach or even exceed 100%; for example, in case when investment portfolio contains short positions or some type of exotic high-risk derivatives. Anyway, this type of discussion belongs to the rather different subject matter (finance and investment) and goes far beyond the boundary and original intention of this article, which is intend to be focused primarily on the mathematical and programmatic technique of SQL aggregate Product function. Therefore, the following solution based on the pre-condition that $(PFM_i) \geq -1$ will have SQL statement as shown in Listing 4 with corresponding result in Table 3:

LISTING 4

```
SELECT Year, Exp(Sum(Iif([PFM]+1=0,0,Log([PFM]+1))))*Iif(Min([PFM]+1)=0,0,1)-1 AS Annual_PFM
FROM Table2
GROUP BY Year;
```

Table 3. Compound Annual Performance, %

Year	Annual_PFM
2004	8.78%
2005	27.46%

It is easy to see that suggested SQL solution applies the mathematical technique demonstrated above in the equation (3) to the annual performance computation formula (4). By adding another Group By clause as shown in Listing 5 the query could return Quarterly Performance calculation (see Table 4).

LISTING 5

```
SELECT Year, Round((([Month]+1)/3) AS Q,
Exp(Sum(Iif([PFM]+1=0,0,Log([PFM]+1))))*Iif(Min([PFM]+1)=0,0,1)-1 AS
Annual_PFM
```

FROM Table2**GROUP BY Year, Round(([Month]+1)/3);**

Please notice the expression: Round(([Month]+1)/3), which provides an easy way to return a Quarter value corresponding to the Month of the Year.

Year	Q	Q_PFM
2004	1	9.26%
2004	2	-2.01%
2004	3	-5.17%
2004	4	7.14%
2005	1	-4.04%
2005	2	4.05%
2005	3	11.39%
2005	4	14.60%

Up to this point all discussed SQL solutions were able to calculate aggregate Product for positive numbers and zero. In the next section the universal solution will be introduced, extending the applicable range to cover negatives as well.

4. SQL AGGREGATE PRODUCT FUNCTION: UNIVERSAL SOLUTION

Universal SQL aggregate Product function, capable of handling any positive, negative numbers and 0 (in other words, any real numbers) is shown in the Listing 6;

LISTING 6

SELECT

Exp(Sum(IIf(Abs([Num])=0,0,Log(Abs([Num])))))*IIf(Min(Abs([Num]))=0,0,1)*(1-2*(Sum(IIf([Num]>=0,0,1)) Mod 2)) AS P

FROM Table1

This universal solution, encapsulated into this single SQL statement, contains three parts (multipliers). The first and the second are recognizable ones, which already have been discussed early in article (see Listing 3); they are capable of handling positive numbers and zeros. Third multiplier is added to take care of the sign by counting negative numbers, then dividing it by Modulo 2 and subtracting the doubled result value from 1. In simple words, it returns 1 for even count and -1 for odd count of all negatives, if any.

4.1. Special handling of Null

The solutions for aggregate Product function discussed above, including the universal one, do not implement the implicit Null-to-value conversion, so any row containing Null will cause the procedure to fail, displaying the error message as shown on Fig.2.



Figure 2. Exception happens when trying to open SQL query (Listing 6), caused by one or more rows in Table 1 containing Null

This issue could be easily resolved by adding Null handling procedure, but the resolution could be application specific. Variable scenarios are listed below.

- ❖ Restrict rows from having Null values; for e.g., by adding constrains to the Table schema definition and/or Field level Validation object.
- ❖ Add Null-to-Zero conversion: `NZ([Num])` function can do the job. In this case any row containing Null will cause the aggregate Product function to return 0 regardless of the value in other rows.
- ❖ Add Null-to-1 conversion; for example, use the expression: `IIF ([Num] Is Null, 1, [Num])`. In this case the aggregate Product function will return the product of all Non-Null row values.

4.2. Performance-Universality consideration

In a context of this section performance means a speed of computation; not to be confused with investment performance discussed above. As the programmatic complexity of the universal solution (Listing 6) is higher than the simplified versions (Listing 1 and 3), then the performance of the universal SQL query will be lower comparing to that simplified ones. This is a classical example of performance-universality dilemma; decision should be made based on the compromise between these two competing metrics when choosing a solution to be adequate to the particular application.

SUMMARY

Suggested approach extends the capability of standard SQL by adding aggregate Product function. The whole solution is encapsulated into a single query, portable to many other SQL backed RDMB, for example, Microsoft SQL server; some syntactical differences could apply. Three versions, which differ in terms of their performance-universality matrix, are presented and discussed in details.