

SQL Titbits for the Inexperienced

Erland Sommarskog

Data Platform MVP



www.DataPlatformGeeks.com | www.SQLServerGeeks.com

[@TheDataGeeks](https://twitter.com/TheDataGeeks) | [@SQLServerGeeks](https://twitter.com/SQLServerGeeks) | admin@DataPlatformGeeks.com

Scope for this Presentation

- Based on questions, mistakes and misconceptions I frequently see in SQL forums.
- Intended for persons who have done SQL left-handedly for a year or so.
 - Maybe more directed to developers than business users.
- We'll start with very basic things – maybe too basic for some.
 - The later titbits will be a little more advanced.
- For all titbits I start with demo and then sum up on slides.



Erland Sommarskog
Independent consultant based in Stockholm
SQL Server MVP since 2001
<http://www.sommarskog.se>
esquel@sommarskog.se

Slides and scripts are available on
<http://www.sommarskog.se/present>

Thanks to Knowledge Partners



actifio

stellar



Video Courses
...by SQLMaestros

cd data

erwin®
the data governance company

These are the Titbits

- GO is not an SQL statement.
- NULL values and the NOT IN trap.
- The CASE expression.
- Some on Data Types in SQL Server.
- UNION and UNION ALL.
- CTEs and Derived Tables.
- Temp Tables and Table Variables.

GO Is Not an SQL Statement

[00_go.sql](#)

- GO is not an SQL statement – it's a *batch separator*.
- GO is never seen by SQL Server, it is an instruction to the query tool to split up the script in batches.
- The tool sends the text up to the first GO to SQL Server and waits for response.
- When the first batch has completed, the tool sends the next batch and so on.

SSMS

```
DECLARE @d datetime2(3) =  
sysdatetime();  
SELECT @d AS now;  
WAITFOR DELAY '00:00:02';  
  
GO  
  
DECLARE @d datetime2(3) =  
sysdatetime();  
SELECT @d AS now;  
WAITFOR DELAY '00:00:02';  
  
GO
```

SQL Server

~~2020-02-16 17:12:35.456~~

More Notes on GO

- GO is not understood by a client API.
 - You need to parse it out yourself.
- CREATE PROCEDURE / FUNCTION / VIEW / TRIGGER must be in a batch of their own.
- Thus, a GO always marks the end of a stored procedure!

NULL values

[02_NULLs.sql](#)

- NULL represents an *unknown* value.
- SQL has three truth values: TRUE, FALSE and UNKNOWN.
- The operators =, <>, >= etc yield UNKNOWN with NULL.
 - The NULL may or may not be the same as the value we are comparing to. Since it is unknown, we don't know.
- WHERE, ON etc only include a row if the condition is TRUE.
- Use IS NULL and IS NOT NULL to check for NULL in a column or variable.

Truth Table for SQL

		AND	OR
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN
TRUE	UNKNOWN	UNKNOWN	TRUE
UNKNOWN	TRUE	UNKNOWN	TRUE
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

NOT UNKNOWN => UNKNOWN

Watch Out for NOT IN

[03 NOTIN trap.sql](#)

```
SELECT *  
FROM    dbo.CustomerResponsibles  
WHERE   CustRespID NOT IN  
        (SELECT CustRespID FROM dbo.Customers);
```

- This query will not return any row if CustRespID is NULL for *any* Customer
 - Since NULL is an unknown value, it may or may not be equal to the CustRespID in CustomerResponsibles.

EXISTS / NOT EXISTS

```
SELECT * FROM dbo.CustomerResponsibles CR
WHERE NOT EXISTS (SELECT * FROM dbo.Customers C
                  WHERE C.CustRespID = CR.CustRespID);
```

- The subquery with [NOT] EXISTS is *logically* evaluated for every row in the outer table.
- If the subquery returns at least one row, EXISTS returns TRUE, else FALSE. (But never UNKNOWN!)
- [NOT] EXISTS is great for multi-column conditions.
- The WHERE clause for an EXISTS subquery should always have at least one condition relating to the outer query.

The CASE Expression

[04 CASE.sql](#)

- There is no CASE statement in SQL. There is a CASE *expression*.
- The result of a CASE is the THEN of the first WHEN condition that evaluates to TRUE.
- If no WHEN evaluates to TRUE, the result of CASE is the value for ELSE or NULL if there is no ELSE.
- THEN/ELSE must be followed by an expression that evaluates to a *single* value – even if it is a subquery.
- And, no, you cannot call stored procedures in CASE.

Data Types in SQL Server

[05_Datatypes.sql](#)

- SQL is a *statically* typed language.
 - But with more implicit conversions than in other languages.
- An expression always returns the one and the same data type.
- If two types meet in an expression, the one with lower precedence is converted to the type with higher.
 - If an implicit conversion exists, that is. Else, there is a compilation error.
- Better to convert explicitly for clarity.

Simplified Precedence List

5. Lowest: (var)binary.
4. Strings.
3. Numbers.
 - 3c. Integer numbers.
 - 3b. Decimal.
 - 3a. Floating-point.
2. Date and time.
1. sql_variant.

Reference:

Complete Precedence List

33. binary (lowest)	24. uniqueidentifier	18. tinyint	9. time
32. varbinary	23. timestamp	17. smallint	8. date
31. varbinary(MAX)	22. image	16. int	7. smalldatetime
30. char	21. text	15. bigint	6. datetime
29. varchar	20. ntext	14. smallmoney	5. datetime2
28. varchar(MAX)	19. bit	13. money	4. datetimeoffset
27. nchar		12. decimal	3. xml
26. nvarchar		11. float	2. sql_variant
25. nvarchar (MAX)		10. real	1. CLR types (highest)

Comparing Strings to Other Types

- Since strings have low precedence, you often get errors if you compare strings to numbers or datetime values:

```
WHERE stringcol = 467  
WHERE stringcol = @dateval
```

Not all values in the string column may be numbers/dates.

- For the computer `467` and `'467'` are very different.
- Convert number/date variables to string explicitly.
- Or convert the string value with `try_convert` which returns NULL if conversion is not possible.

UNION and UNION ALL

[06 UNION.sql](#)

- UNION and UNION ALL combine the result sets of two or more queries into a single result set.
- Column names are taken from the first query.
- If the data type is different for the same column in the individual queries, there will be conversion according to the precedence rules, just as for CASE.
- Better to convert explicitly for clarity.
- ORDER BY can only be at the end; applies to the full query.

More on UNION [ALL]

- UNION removes duplicate rows from the result sets, *including* duplicates within the sets.
- UNION ALL retains duplicates.
- Tip: Never use UNION, always UNION ALL. This is what you want 95 % of the time anyway.
- If you want distinct values, wrap UNION ALL in an outer SELECT DISTINCT – makes it clearer what you are doing.
- Checking for duplicates comes with a cost in performance.

Derived Tables

[07_CTE temptables.sql](#)

- A *derived table* is an independent subquery that you can use in FROM or JOIN.
 - It cannot refer to anything in the outer query.
 - It *must always* have an alias.
- It is a *logical* construct, and may not be computed as such, if the optimizer recasts the computation order.
- It is a handy way to reuse a smaller query in a bigger query.

Common Table Expressions (CTE)

[07_CTE temptables.sql](#)

- On the top of your query, you can define one or more CTEs.
- The first CTE is introduced by WITH.
 - Previous statement must be terminated by a semicolon.
 - Alternatively, put a semicolon before WITH.
- CTE2 and CTE3 can be refinements of CTE1– but they can also be entirely unrelated.

```
; WITH CTE1 AS (  
    SELECT ...  
    FROM ...  
) , CTE2 AS (  
    SELECT ...  
    FROM ...  
) , CTE3 AS (  
    ...
```


CTEs, cont'd

- A CTE is just like a derived table – but it has a name.
- You can use that name in the rest of the query just like a table.
- For each occurrence of the name, the text of the CTE is expanded into the query – it's a macro.
- It is a *logical* construct and optimizer may recast computation order, and a CTE may never be computed as such.
- CTEs are great to structure complex queries.
- The scope of a CTE is a single query; it cannot be used in queries that follow.

Temp Tables and Table Variables

[07_CTE temptables.sql](#)

- Temp tables: CREATE TABLE, name starts with a single #.
- Table variables: DECLARE @tvar TABLE (...).
- Both serve the same purpose: a working area that is private to the process and not visible to other processes.
- Two processes can create a temp table or declare a table variable with the same name at the same time without conflict.

Temp Tables and Table Vars, cont'd

- When defined in an SP they go away when procedure exits.
- A temp table created on top level goes away when process exits, or you say DROP TABLE. A table variable only lives for a single batch.
- Temp tables can be accessed by inner procedures. Table variables cannot.
- Both live in tempdb.
- Myth has it that table variables are memory-only. Not true!

Temp Tables vs Table Vars, Performance

- Table variables are known to be prone to bad performance.
- Tip: if you run into a performance issue with a query that uses a table variable, try replacing it with a temp table.
- But the full story is a lot more complicated and sometimes a table variable gives better performance.
- Table variables are OK if you only have a handful of rows, but if you expect more, use a temp table.

CTE vs Temp Tables

- Using a temp table (or table var) for an intermediate result means that it is always computed as such and materialised.
- Thus, a CTE can be expected to be faster than a temp table.
- With complex or convoluted CTEs, it may be difficult for the optimizer to get estimates right.
- In that case, you can help the optimizer by storing partial results in a temp table.
- Using a temp table can also make your debugging easier.

Temp Tables and Name Resolution

[08_tmptbl_TV.sql](#)

- When you create a procedure that creates a temp table, the table typically does not exist at that time.
 - Therefore, SQL Server suppresses errors about missing tables.
- While helpful, this can serve to mask errors with other tables in a query with a temp table.
- Tip: attempt to give your temp tables specific names, and do not use generic names like #temp, #t etc.
 - This can save you from nasty surprises with nested procedures.
- Table variables do not have these problems.
 - Use while developing and change to temp tables when you are done?

Summary I

- GO is not an SQL statement – it's a batch separator.
- NULL is an unknown value, all comparisons with NULL yield UNKNOWN.
- Use IS [NOT] NULL to check for NULL.
- NOT IN will not give any results with NULL values. Use NOT EXISTS to avoid this trap.
- [NOT] EXISTS is also good for multi-column conditions.
- There is no CASE statement in SQL – there is a CASE expression.

Summary II

- SQL is a statically typed language. Expressions and columns in a result set have a static data type.
- Many implicit conversions are permitted. They follow a precedence list that defines which type “wins”.
- UNION [ALL] permits you to combine two or more result sets into a single one.
- UNION removes duplicates, UNION ALL does not.
- Always use UNION ALL. Wrap in DISTINCT to make it clear that this what you want!

Summary III

- CTEs and derived tables are logical tools to build complex queries.
- They are necessarily not computed as such.
- Temp tables and table variables are private work areas.
- Temp tables often give better performance than table variables, particularly with larger amounts of data.
- CTEs and derived tables are generally faster than temp tables – but exceptions are commonplace.

Summary IV

- With temp tables, errors due misspellings etc may not be uncovered until run-time.
- Table variables do not have this issue.
- Attempt to give your temp table distinct names!

The Final Titbit

Erland Sommarskog – esquel@sommarskog.se.

Slides and scripts on <http://www.sommarskog.se/present>.

SQLTitbits database: [SQLtitbits db.sql](#).

Northgale: [instnwnd.sql](#) + [Northgale.sql](#).