Martin's Microsoft SQL Cheat Sheet - v20161226

I use this page myself. When one deals with so many different dialect and languages, sometimes even I get confused or forgetful what is correct in which case.

Also, there are many useful script snippets that I might need once every few months. Unless you do ONLY SQL on a DAILY basis, you cannot be expected to keep these snippets at your immediate memory's grasp.

- Items to still document:
  - Comparing Strings
  - Type Conversions
  - Finding Duplicates
  - Finding PK issues
  - Creating Objects: Database
  - Creating Objects: Tables
  - Creating Objects: Views
  - Creating Objects: Index
  - Creating Objects: Stored Procedures
  - Creating Objects: User Defined Functions, SCALAR
  - Creating Objects: User Defined Functions, TABLE
  - Creating Objects: User Defined Functions, INLINE TABLE
  - TRY CATCH
  - Transactions
  - SQL Best Practices
  - etc

GO?

The word GO is used to seperate SQL batches in the standard Microsoft SQL Server Management Studio. Each seperated SQL batch has its own scope - so variables defined in one batch will not be accessable in other batches.

```
GO
```

Comments

Comments are useful in documenting your code. The downside is, sometimes someones changes code, but not the comments, so the comments may say one thing, but the code does something else. So never 100% trust comments - they might be just rumours...

```
-- SINGLE LINE COMMENT
GO

        SET @a = 1; -- END OF LINE COMMENT
GO

/*
        Block
        Comment
        Note, you can have:
```

```
                -- Single Line Comments in Block Comments.
        But block comments can not have block comments...
*/

GO
```

Variables

Variables are the way T-SQL can keep a temporary state - they are only valid within an SQL batch, and are not persisted in any way.

```
/*
        Basic Variable as defined with a DECLARE statement:
*/
        DECLARE @myNumber AS int;
        SET @myNumber = 1;
GO
/* Alternative short form for SQL 2008 and up. */
        DECLARE @myNumber AS int = 1;
GO
/* Setting a variable as a result from a query. */
        DECLARE @myNumber AS int = 0;
        SELECT @myNumber = COUNT(*) FROM myTable;
GO
/* Attempting to use a variable from a previous batch. */
        SELECT @myNumber; -- This will ERROR.
GO
```

Strings

```
/*
        Strings are denoted between two single quotes:
*/
        SELECT 'This is my String!';
/*
        To include a single quote in a string, it has to be escaped with a single quote.
*/
        SELECT 'Yep, that''s an escaped single quote you see here!';
/*
        Various Basic, Non-UNICODE Strings Types
        These use one (1) BYTE per charachter.
*/
        DECLARE @myNumber AS char(10) = '1234567890';
        DECLARE @myNumber AS varchar(10) = '1234567890';
/*
        Various WideChar / UNICODE Strings Types
        Recommended to use where international data may have to be stored.
        Each UNICODE uses two (2) BYTES per charachter.
*/
        DECLARE @myNumber AS char(10) = '1234567890';
        DECLARE @myNumber AS varchar(10) = '1234567890';
GO
```

Trimming Strings

Many languages just have one TRIM() function that fully trims whitespace from a string, both pre-fixed and post-fixed. In T-SQL, trimming only removes the SPACE character. To remove basic whitespace, you will need a bit more code - but better would be to trim results on the applciation layer, anyway, for performance reasons.

In T-SQL, due to engine specific considerations regarding indexes (ie: trying to keep things sargable, there are two versions of TRIM().
LTRIM() for left trim (not sargable) and RTRIM() for right trim (still sargable).

Because Microsoft SQL follows ANSI/ISO SQL-92, when comparing two strings, RTRIM() is not normally needed, as both strings being compared will be internally padded with SPACE to make them the same length. This does not apply to LIKE, though.

On the other hand, if you do want to compare two strings taking any suffixed SPACES, into consideration, you will need to do a bit of work - especially if you want to stay sargable. When I find a good method, I will update this.

Performance note: Using LTRIM() on a column, any index on that column will be ignored. This can significantly slow down statements, so should be avoided.

```
/* Basic TRIM examples - I want to return a SPACE trimmed string: */
        SELECT LTRIM(RTRIM(myTextColumn)) AS myAlias FROM myTable;
        -- Performance Note: Do this on the application layer.
GO
/* No RTRIM() required here, still sargable: */
        SELECT * FROM myTable where myColumn = 'Boo    ';
GO
/* Not sargable, but sometimes necessary if you have post-fixed SPACES: */
        SELECT * FROM myTable where LTRIM(myColumn) = 'Boo    ';
GO
```

Replacing Strings

T-SQL provides a simple REPLACE({original text},{search text},{new text}) command. Note that case sensitivity is dependent on collation.

```
/* Using default collation: */
        DECLARE @myString as nvarchar(max) = 'Run Run Slow, or slow.';
        SET @myString = REPLACE(@myString,'Slow','Fast');
        -- This will replace both "Slow" and "slow".
GO
/*
        Forcing a specific collation using the COLLATE keyword,
        in this example both CASE and ACCENT sensitive:
*/
        DECLARE @myString as nvarchar(max) = 'Run Run Slow, or slow.';
        SET @myString = REPLACE(@myString COLLATE Latin1_General_CS_AS,'Slow','Fast');
        -- This will only replace the first capitolized "Slow".
        GO
```

Concating Strings

```
/* Concating with a + sign: */
        DECLARE @myString AS nvarchar(max) = 'ABC';
        SET @myString = '[' + @myString + ']';
        SELECT @myString;
GO
/* Concating using CONCAT() : */
        DECLARE @myString AS nvarchar(max) = 'ABC';
        SET @myString = CONCAT('[',@myString,']');
        SELECT @myString;
GO
```

Padding Strings

There are a couple of choices to pad strings.
If you just want to pad a SPACE, use SPACE({repeat count}).
A handy one for UNICODE strings that can pad any charachter (or even a set of charachters) is REPLICATE({charachter(s)},{repeat count});

```
/* Padding strings with SPACE(): */
        SELECT SPACE(4)+myColumn FROM myTable;
GO
/* Padding strings with REPLICATE(): */
        SELECT REPLICATE(' ',4)+myColumn FROM myTable;
GO
```

Numbers

Numbers come in various precisions and bases, and in two flavours, exact and approximate.

Exact Numbers

- BIT
- TINYINT
- SMALLINT
- INT
- BIGINT
- DECIMAL (aka NUMERIC)
- SMALLMONEY
- MONEY

Approximate Numbers

- FLOAT
- REAL

As you can imagine, the use of approximate numbers should be avoided where you do not want rounding errors. This should be especially avoided in financial and ERP systems.

```
/*
        There are no calculation shortcut to assignments:
*/
        DECLARE @myInt AS int = 3;
        SET @myInt = @myInt + 1; -- Increment by one. += or ++ does not exist.
GO
```

Dates

Dates are frankly PITA because most people do not use the sanest Hungarian format (also known as an "invarient" format in some texts):

```
/*
        YYYYMMDD
                => Full Year, Zero Leading Month, Zero Leading Day
                => Example: 20161225 = December 25, 2016.
                => Example: 19070311 = July 11, 1907.
*/
```

Which BTW, I've been told Microsoft SQL server uses internally. Whoever designed MS-SQL was very sane. (That said, future MS-SQL Product Owners may vear off sanity and use other, obviously inferior formats.)

Why is YYYYMMDD superiour, you ask? First off, try sorting any other format - you will quickly find it a lot of work. YYYYMMDD logically is made up of most significant to least significant. You also do not format time as mm:ss:hh, do you?

Also, bonus: YYYYMMDD can be easily converted to an INT, and no-one needs to painfully remember if the valid delimiter is a minus, slash, dot, or what have you. Neither do you need to decide what number part represents a day, a month, or even an year for any number below 13.

TEST: Here a typical confusing date format. Quick! Tell me what date this is:

```
/*
        12/4/5 = ?
*/
```

Nope. Wrong.

```
/*
        Some DATE related facts and tricks:
*/
        DECLARE @myDate as DATE = '20170101'; -- Jan 1, 2017.
        SELECT @myDate; -- Returns format set by the SQL server's collation.
GO
```

TOP

DateTime

```
/*TODO*/
```

TOP

Time

```
/*TODO*/
```

TOP

Query Conditional : Simple CASE

```
/*
        Simple CASE uses the form:
        CASE {expression}
                WHEN {value} THEN {result}
                ELSE {default result}
        END

        Note 1:
                ELSE is optional.
                If no ELSE specified, and no matches found, NULL returned.

        Note 2:
                As soon as a WHEN is TRUE, all other WHEN and ELSE are
                no longer processed.
*/
        SELECT
                CASE myColumn
                        WHEN 1 THEN 'I got One!'
                        WHEN 2 THEN 'I got Two!'
                        ELSE 'I got... Unknown?!?'
                END
        FROM
                myTable;
GO
```

TOP

Query Conditional : Searched CASE

```
/*
Searched CASE Uses the form:
        CASE
                WHEN {condition} THEN {result}
                ELSE {default result}
        END
```

```
        Note 1:
                ELSE is optional.
                If no ELSE specified, and no matches found, NULL returned.

        Note 2:
                As soon as a WHEN is TRUE, all other WHEN and ELSE are
                no longer processed.
*/
        SELECT
                CASE
                        WHEN myColumn = 1 THEN 'I got One!'
                        WHEN myColumn in (2,3,4) THEN 'I got 2, 3 or maybe 4!'
                        WHEN myColumn between 5 and 8 THEN 'I got something between 5 and 8!
                        WHEN myColumn = 9 or myOtherThing = 9 THEN 'I got a 9, somehow...'
                        WHEN myColumn+10 > 10 THEN 'More than 10, that is for sure!'
                        ELSE 'I got... Unknown?!?'
                END
        FROM
                myTable;
GO
```

TOP

Query Conditional : Nested CASE

```
/*
        CASE NESTING.

        Simple and Searched CASE can be nested in any combination.

        Note:
                T-SQL up to v2016 only allows up to 10 nests.
                Check your version for the current limit.
*/
        SELECT
                CASE
                        WHEN myColumn in (1,3,5,7,9) THEN
                                CASE myColumn
                                        WHEN 1 then 'I got One!'
                                        ELSE 'I got an Odd Number!'
                                END
                        ELSE 'I got either an EVEN number between 2 and 8, or something unkn
                END
        FROM
                myTable;
GO
```

TOP

Conditional: IF ELSE ELSEIF

```
/*
        Format:
                IF {first conditional}
```

```
                        {action if first conditional is TRUE}
                ELSE IF {second condtional, only checked when first conditional was FALSE}
                        {action if second conditional is TRUE}
                -- Repeat ELSE IF as needed...
                ELSE
                        {action if all above conditionals were FALSE}

        Note:
                Highly reommend using the BEGIN ... END format as per below example.

                This will keep future code changes clean.

*/
        DECLARE @myNumber AS int = 0;
        IF @myNumber = 999
        BEGIN
                PRINT 'Yo, you got 999 things! Nice!';
        END
        ELSE IF @myNumber % 2 = 0
        BEGIN
                PRINT 'Yo, you got yourself a sweet, sweet EVEN number there! RAD!';
        END
        ELSE
        BEGIN
                PRINT 'Yo, I have NO IDEA what you got. Hope it is useful, though...';
        END
GO
/*
        Sloppy Programmer Shortcut IF...

        SRSLY. Do not do this! You will kick yourself.

        (And yeah, MSDN has this SLOPPY form as an example. Ugh.)
*/
        DECLARE @myNumber AS int = 0;
        IF @myNumber = 999
                PRINT 'Sloppy. But you got 999.'
        ELSE
                PRINT 'Still sloppy, and I don''t want to talk to you anymore.'
GO
```

TOP

Loops: While

```
/* ... */
        DECLARE @myLooper AS int = 0
        WHILE @myLooper < 10
        BEGIN
                SET @myLooper = @myLooper + 1;
                PRINT @myLooper;
                IF @myLooper >= 10
                BEGIN
                        BREAK;
                END
                ELSE
                BEGIN
                        CONTINUE;
                END
                PRINT 'Because of the CONTINUE above, you will never see this. Sniff.';
        END
```

```
        PRINT 'Looping all done...';
GO
```

Renaming Objects

DBA Note: Make sure no-one is connected when renaming objects. Otherwise unwanted "hilarity" will ensue.

DBA Note 2: Renaming may break code. Make sure you follow through any renaming task all the way downstream.

DBA Note 3: Do not directly use the following methods to rname objects if database is running any of these - a bit more work will be required:

- SQL Replication;
- Always On;
- Any sort of CDC.

```
/*
        Renamimg a Database - SQL 2005 and up:
*/
        ALTER DATABASE oldDBName MODIFY NAME = newDBName;
GO
/*
        Renamimg a Database - SQL 2008 and up:
*/
        EXECUTE sp_rename N'myOldDB', N'myNewDB', N'DATABASE';
GO
/*
        How to rename a table - no type needed:
*/
        EXECUTE sp_rename N'myOldTableName', N'myNewTableName';
GO
/*
        How to rename a column in a given table:
*/
        EXECUTE sp_rename N'mySchema.myTable.OldColumnName', N'NewColumnName', N'COLUMN';
GO
/*
        How to rename an index of a given table:
*/
        EXECUTE sp_rename N'mySchema.myTable.OldIndexName', N'NewIndexName', N'INDEX';
GO
```

Cursors

```
/*

        Cursors.

        Just... don't.

*/
```

Ok, first off a confession.

Yes, I use cursors.

But only in adhoc and DBA type situations, where I (mis?)use T-SQL as my application layer. Something you do not want to do in a proffesional application though.

In the right situation, cursors can be amazingly powerful.

BUT:

Cursors are performance killers if used without very, very carefully understanding all the implications of how they work and the various side effects associated with the many configuration options.

Rule of thumb: If you can, move the functionality of what a cursor supplies out to your application layer. Or at least look into ways to run cursorless.

So... if you REALLY need to use cursors, well, here you go, and may the tears of overworked DBA soak your slippers.

Basic Cursor

This is the basic, default cursor type. There are a lot of options available for T-SQL cursors that you should really look into if you _have_ to use cursors.

(We are sure you do not _want_ to use cursors - no sane person would.)

This basic cursor in not very optimized, and it is updateable. That is, you can update and even delete the current row.

We need variables to hold all the data from the current row we are on during the cursor looping.

We also need to define what our cursor needs to do - that is the SELECT. And yes, you can do an ORDER BY if you want to retrieve rows in a defined way.

NOTE: The cursor variable does NOT have the @ as a pre-fix!

Finally, if you want to get truly ugly, you can even nest cursors.

Ugh - now I have to go wash my mouth out with soap...

```
/*
        Setup - do all the needed declarations.
*/
        DECLARE @myCol1 AS int;
        DECLARE @myCol2 AS nvarchar(30);
        DECLARE myCursor CURSOR FOR
        SELECT
                myColumn1
                , myColumn2
        FROM
                mySchema.myTable;
```

```
/*
        Open a Cursor:
*/
        OPEN myCursor;

/*
        Fetch the very first row into our variables we declared above:
        NOTE: We can only do a NEXT for the first row with the default cursor type.
*/
        FETCH NEXT FROM myCursor INTO @myCol1, @myCol2;

/*
        Loop through.
        The internal variable @@FETCH_STATUS will stay ZERO as long as
        current row has data.
*/
        WHILE @@FETCH_STATUS = 0
        BEGIN
                -- DO SOMETHING HERE.
                PRINT 'Something or other is done here...'
                -- And as the last line, get the next row of data.
                FETCH NEXT FROM myCursor INTO @myCol1, @myCol2;
        END

/*
        Vital - the cleanup!
        If you do not cleanup, you will get zombies.
        No one likes zombies.
        Not even Carl.
*/

        CLOSE myCursor; -- Close the current row.
        DEALLOCATE myCursor; -- and then free up the cursor resources.

GO
```