

# Introduction to Regular Expressions in SQL Server

Erland Sommarskog  
Data Platform MVP



# Erland Sommarskog

Independent consultant based in Stockholm

SQL Server MVP since 2001

[esquel@sommarskog.se](mailto:esquel@sommarskog.se)

<http://www.sommarskog.se>

Slides and scripts available on

<http://www.sommarskog.se/present>



# Agenda

- Introduction and overview.
- Lessons – demos in SSMS and key points repeated on slides.
  - One: Comparison with LIKE.
  - Two: Brackets.
  - Three: Alternates.
  - Four: Quantifiers.
  - Five: REGEXP\_REPLACE.
  - Six: Escape Sequences.
  - Seven: Letters and words.

# Regular Expressions? What? Why?

- Versatile search patterns for string data that can be used for advanced string manipulation.
- Example: replace multiple spaces with a single:  
`SELECT regexp_replace(  
 'A text with extra space', ' +', ' ')`
- Returns `A text with extra space`.
- Doing this in SQL 2022 is very difficult!

# Caveat about Implementations

- About every environment with regular expressions has its own implementation.
- => Just like SQL, there are differences between environments.
- SQL Server's implementation is based on the [RE2](#) library from Google.
- RE2 is fairly basic – still powerful enough.
- There are some irritating deviations.

# Overview of the RegExp Functions

- **REGEXP\_LIKE** – Does *string* match *regexp*?
- **REGEXP\_REPLACE** – Like the REPLACE function, but the search string can be a regular expression.
- **REGEXP\_COUNT** – How many matches of *regexp* are there in *string*?
- **REGEXP\_SUBSTR** – Returns one match of *regexp* in *string* – by default the first, but you can get others.
- **REGEXP\_INSTR** – Returns starting/ending position of the match.



# Two Table-Valued Functions

- REGEXP\_MATCHES – Returns a table with all matches of *regexp* in *string*.
- REGEXP\_SPLIT\_TO\_TABLE – Returns a table with the fragments of *string* outside matches of *regexp*.
  - This is STRING\_SPLIT on steroids!

# Notes on Compatibility Level

- REGEXP\_LIKE and the two table-valued functions require compatibility level **170**.
- The other functions – REGEXP\_REPLACE, REGEXP\_COUNT, REGEXP\_SUBSTR and REGEXP\_INSTR – work on any compatibility level.
- On lower compatibility levels, REGEXP\_COUNT can be a decent substitute for REGEXP\_LIKE.



# Regex functions and LOB support

- For (n)varchar(MAX) types, input is restricted to 2 MB.
- You need CU5 or later, for all regexp functions to support MAX types.
  - Prior to CU5, REGEXP\_REPLACE and the table-valued functions did not accept MAX.
- The old types **text** and **ntext** are not supported.

# Lesson One – Comparison with LIKE

[regexdemos.sql](#)

- REGEXP\_LIKE is a boolean function.
  - Cannot be used in a SELECT list directly, wrap in IIF or similar.
- With LIKE, the search pattern must describe the full search string.
- A regular expression matches anywhere in the search string.



# Lesson One – Case Sensitivity

- LIKE respects and understands collations.
- Regular expressions *do not* and are by default case-sensitive.
- Set the **flags** parameter to **i** for case-insensitive.
- All regexp functions accept **flags** – position varies.

# Lesson One – Metacharacters

- In regexps, a number of punctuation characters serve as *metacharacters* with special meaning.

First few metacharacters:

- ^ – Matches at the beginning of string.
- \$ – Matches at the end of string.
- . (dot) – Matches any character exactly once.
  - Exception: newline.



# Lesson One – Backslash

Backslash (\) is an escape character.

- Before an ASCII punctuation character it means “take next character at face value”.
- Best practice: To match ASCII punctuation chars, always escape with backslash to avoid surprises.
- Backslash + ASCII alphanumeric may have a special meaning, as we will look at later.
  - If there is no special meaning, you get an explicit error.

# Lesson Two – Brackets

[regexdemos.sql](#)

- Both with LIKE and regular expressions you can match a group of characters with help of brackets.
- **[abc]** – Matches any of “a”, “b” and “c”.
- **[0-9]** – Matches all characters in the range.
  - With regexps, the range is defined from character code points.
- **[^abc0-9]** – The ^ negates the set; matches anything *but* “a”, “b”, “c” and digits.



# Lesson Three – Alternates

[regexdemos.sql](#)

- The metacharacter `|` (pipe) separates alternate regular expressions that qualify for a match.
- E.g. `abc|jkl|xyz` = any of *abc*, *jkl* and *xyz* match.
- Use parentheses, `()`, to change precedence order.
- `|` has the lowest precedence.
  - `^abc|def` = starts with *abc* or contains *def* anywhere in string.
  - `^(abc|def)` = starts with *abc* or *def*.

# Lesson Four – Quantifiers

[regexdemos.sql](https://regexdemos.sql)

- Quantifiers is the core of why regular expressions are so powerful!
- A quantifier specifies how many times the previous regular expression needs to appear for a match.

The most commonly used:

- `*` (star) – 0 or more times.
- `+` (plus) – 1 or more times.
- `?` (question mark) – 0 or 1 time.

# Lesson Four – Quantifiers, cont'd

The general ones:

- $\{m\}$  – Match exactly  $m$  times.
- $\{m,n\}$  – Match at least  $m$  and at most  $n$  times.
- $\{m,\}$  – Match  $m$  or more times.



# Lesson Five – REGEXP\_REPLACE

[regexdemos.sql](#)

- In the replacement string, use `\&` to have the entire match inserted into the return value.
  - Useful if you want to wrap a match with something.
- Parentheses in the regexp define *capture groups*, which you can refer to in the replacement string as `\1`, `\2` etc.
  - Very useful when you want to replace something in a context and retain the context.
- Keep in mind that nested parentheses also define capture groups.

# Lesson Five – Greedy Quantifiers

- Quantifiers are by default “greedy”.
  - They match as long as they can, and only then the RE engine starts backtracking to find matches for the rest of the regexp.
- Override this behaviour by putting a ? after the quantifier.
  - Now it will stop when the rest of the regexp matches.
- Problems with greed often appear with .\* and .+.
- Tip: Write your regexp without any extra ?, but try adding ? if wrong result seems to be due to greed.

# Lesson Six – Escape Sequences

[regexdemos.sql](#)

- Shortcuts to specify control characters in regexps:
  - `\t` = Tab.
  - `\r` = Carriage return.
  - `\n` = Line feed (a.k.a newline).
  - `\f` = Form feed.
- To specify any character, use `\x{nnnnnnnn}`, where *nnnnnnnn* is the Unicode code point in hex.
  - Leading zeroes can be left out.
  - Braces not needed for values  $\leq$  FF.



# Lesson Six – Character Classes

- `\s` – Matches white space: Space, tab, CR, LF and FF – but not hard space.
  - To also match hard space, use `[\s\xA0]`.
- `\S` – Matches everything that `\s` does not match.
  - I.e., shortcut for `[^\s]`.
- `\d` – Matches digits 0-9. I.e., the same as `[0-9]`.
- `\D` – Inverse of `\d`, that is `[^0-9]`.

# Lesson Seven – Letters and Words

[regexdemos.sql](https://regexdemos.sql)

- **\pL** matches characters classified as letters in Unicode.
  - **\p{Lu}** matches uppercase letters.
  - **\p{Ll}** matches lowercase letters.
- Use **\pL** rather than something like **[A-Za-z]**, so that you can match *résumé*, *naïve*, *άλφα* etc without hassle.
- **\PL** matches everything that is not a letter. Useful to match an entire word in a text without worry about punctuation and beginning/end of line.
  - Example: **(^|\PL)word(\PL|\$)**.

# Lesson Seven – Unicode Categories

- More generally, `\pX` matches a Unicode category, and `\p{Xx}` a subcategory.
- Examples:
  - `\pP` – Punctuation.
  - `\p{Nd}` – Decimal numbers.
- In total, there are 30+ of categories/subcategories.
- Full list in RE2 documentation, link on last slide.



# Lesson Seven – The Ugly Ones

- **\w** – Matches word characters, i.e. digits, letters and underscore, but is ASCII only. **DO NOT USE!**
- **\W** – The inverse of **\w**. **DO NOT USE!**
- **\b** – Word boundary. ASCII-crippled. **DO NOT USE!**
- **[[:class:]]** – Posix classes. ASCII only. **DO NOT USE!**
- These restrictions are specific to SQL Server (and RE2). These operators work correctly on other platforms.

# Conclusion

- With regular expressions you can perform advanced string operations that previously were impossible or very cumbersome to write in T-SQL.
- Complex regular expressions can certainly be deterrently hard to read. You will get used to it. :-)
- Be warned: You will still hit limits. Not all parsing is fit for regular expressions.
- It still a giant leap forwards.

# The Last Slide

- Regular Expressions Overview:  
<https://learn.microsoft.com/en-us/sql/relational-databases/regular-expressions/overview>.
- Syntax Google RE2:  
<https://github.com/google/re2/wiki/Syntax>.
- Slides and scripts:  
<https://www.sommarskog.se/present>.
- Email: [esquel@sommarskog.se](mailto:esquel@sommarskog.se).