QA Framework

SQL syntax used in QA tests

Esri Schweiz AG

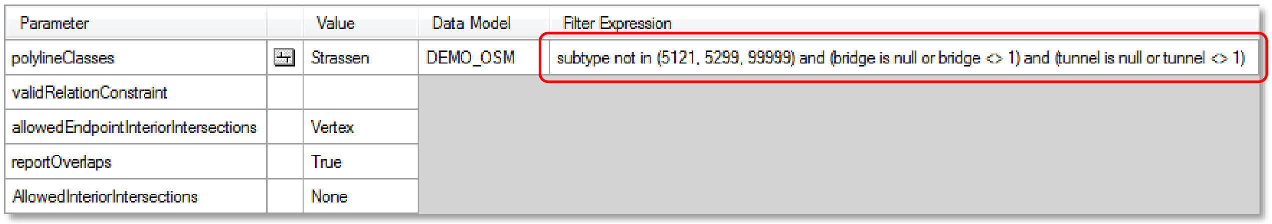
Josefstrasse 218
CH-8005 Zürich
Phone +41 58 267 18 00
http://esri.ch

Table of Contents

1 Introduction

The QA framework uses SQL expressions for several purposes:

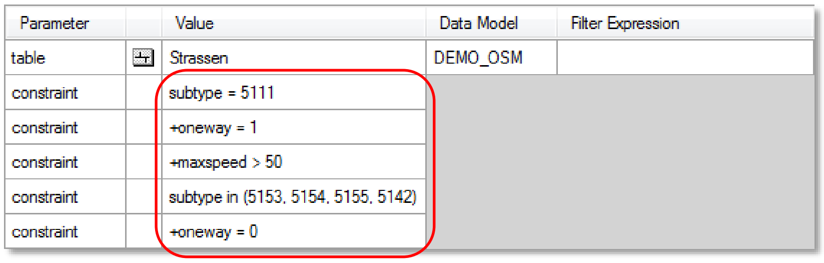
- As filter expressions for dataset parameters of quality conditions, to restrict the set of objects that are to be processed by a test.



Parameter	Value	Data Model	Filter Expression
polylineClasses	Strassen	DEMO_OSM	subtype not in (5121, 5299, 99999) and (bridge is null or bridge <> 1) and (tunnel is null or tunnel <> 1)
validRelationConstraint			
allowedEndpointInteriorIntersections	Vertex		
reportOverlaps	True		
AllowedInteriorIntersections	None		

  The above figure shows a quality condition based on the test **QaLineConnection**, stating that intersections of roads of certain types (as defined by the filter expression) should only occur at feature end points.

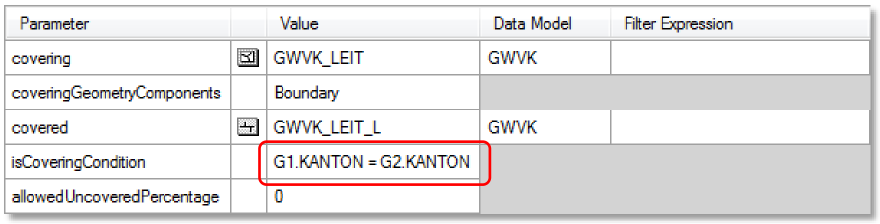
- As attribute constraints for valid objects.



Parameter	Value	Data Model	Filter Expression
table	Strassen	DEMO_OSM	
constraint	subtype = 5111		
constraint	+oneway = 1		
constraint	+maxspeed > 50		
constraint	subtype in (5153, 5154, 5155, 5142)		
constraint	+oneway = 0		

  The above figure shows a quality condition based on the test **QaDatasetConstraintFactory**, which allows the definition of hierarchical constraints, with nested constraints prefixed by a `+` character. The expressions at the lowest level are evaluated as constraints applicable to selections of objects defined by their parent expressions.

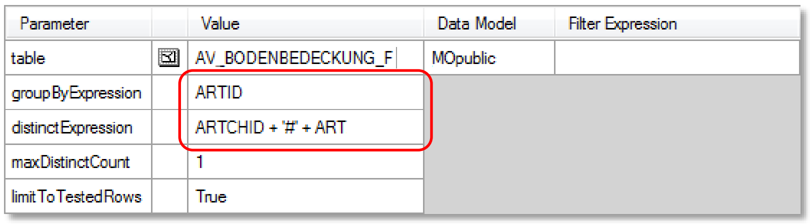
- For the comparison of attributes of two objects, to apply attribute-based on conditions on object pairs involved in a certain spatial relationship.



Parameter	Value	Data Model	Filter Expression
covering	GWVK_LEIT	GWVK	
coveringGeometryComponents	Boundary		
covered	GWVK_LEIT_L	GWVK	
isCoveringCondition	G1.KANTON = G2.KANTON		
allowedUncoveredPercentage	0		

  The above figure shows a quality condition based on the test **QaIsCoveredByOther**, stating that certain line features must be covered by the boundary of polygon features which have the same value for the attribute 'KANTON' as the covered line. In the **isCoveringCondition** parameter, the covering feature can be addressed with the alias `G1`, and the covered feature with the alias `G2` (the available alias names are documented for each test parameter that allows to specify a comparison expression).

- To derive a calculated value from attributes of an object (e.g. grouping criteria, concatenation of URL strings, calculation of M values at calibration points etc.).



Parameter		Value	Data Model	Filter Expression
table		AV_BODENBEDECKUNG_F	MOpublic	
groupByExpression		ARTID		
distinctExpression		ARTCHID + '#' + ART		
maxDistinctCount		1		
limitToTestedRows		True		

The above figure shows a quality condition based on the test **QaGroupConstraints**, which allows to limit the number of distinct values resulting from a SQL expression which exist within groups that are also defined by the results of a SQL expression. The example uses a single field as grouping criterion, and counts the number of distinct values resulting from a concatenation of two fields within those groups.

With very few exceptions[1] these SQL expressions are evaluated by a SQL engine within the QA framework, and not by the underlying geodatabase. This has several advantages:

- A common SQL syntax can be used for all data source types (personal geodatabase, file geodatabase, ArcSDE geodatabases in all supported database products and Shapefiles). This is important for using the same quality specifications on different data sources (e.g. using geoprocessing tools or when verifying child geodatabases in ArcMap).
- The expressions can be evaluated for features that are cached by the QA framework during the verification. This capability reduces round-trips to the database and increases the effectiveness of a common feature cache, and is a key factor for ensuring the outstanding performance of the verification process in the QA framework.
- Expressions can be used on columns that are not directly available in database tables. Examples are special columns like the number of dangling ends of a polyline (“`_DangleCount`” in **QaDangleCount**) or an indication of the direction of a polyline at a line junction (“`_StartsIn`” in **QaLineConnection** and **QaLineConnectionFieldValues**). The comparison of object pairs also relies on this by allowing to address the field values for the two objects using a table-like alias name (e.g. “`G1.TYPE <> G2.TYPE`”).

The syntax used by this SQL engine follows standard SQL for the most common syntax elements. The majority of expressions will not be different from corresponding expressions written for and evaluated by a specific data source.

This document documents syntax of the SQL engine used by the QA framework.

2 Column names

The names of columns can be used directly in the SQL expressions:

```
ID = 10   -- column name
```

Only if a column name corresponds to a reserved word (one of `And`, `Between`, `Child`, `False`, `In`, `Is`, `Like`, `Not`, `Null`, `Or`, `Parent`, `True`) it must be wrapped either in square brackets or ``` ` ``` (grave accent) quotes:

[1] mainly filter expressions for a few special tests, see section 11

```
[Parent] = 999    -- column name matching reserved word, escaped with []
`Parent` = 999    -- alternative escape character "`"
```

The same applies to column names that contain any of these special characters:

`~` `(` `)` `#` `\` `/` `=` `>` `<` `+` `-` `*` `%` `&` `|` `^` `'` `"` `[` `]`

However these characters are not valid for geodatabase column names either, so this situation should not occur.

To address columns from joined tables, the column names must be qualified with the source table name:

```
LANDUSE.Type = 2  -- qualified column name, to be used for joined view
```

2.1 Row property alias names

The field names for the length and area of the geometry (shape) of a feature can be different for different types of geodatabases. This is inconvenient when the same filter conditions or constraints referring to these fields should be applied in different geodatabases. To facilitate the use of the same quality conditions in different geodatabase types, these properties can be addressed using the alias names `$ShapeArea` and `$ShapeLength`. Similar alias names exist for other shape properties and special fields.

The following table lists all aliases for properties of features or table rows that can be used in SQL expressions:

Alias	Description	Data type	Applicable for
$ObjectID	The Object ID for the feature or table row	Integer	All tables and feature classes
$ShapeArea	The area of the polygon (in the units of the spatial reference)	Double	Polygon and MultiPatch feature classes
$ShapeLength	The polyline length or polygon perimeter length (in the units of the spatial reference)	Double	Polygon and polyline feature classes
$ShapePartCount	The number of rings/paths in the shape	Integer	Polygon, polyline and MultiPatch feature classes
$ShapeVertexCount	The number of points/vertices in the shape	Integer	All feature classes

$ShapeInteriorRingCount	The number of interior rings (holes) in the polygon. `NULL` for non-simple (incorrectly structured) polygons.	Integer	Polygon feature classes
$ShapeExteriorRingCount	The number of exterior rings in the polygons. `NULL` for non-simple (incorrectly structured) polygons.	Integer	Polygon feature classes
$ShapeMMax	The maximum M value for the feature. `NULL` for feature classes without M values, and for features without *any* assigned M values.	Double	All feature classes
$ShapeMMin	The minimum M value for the feature. `NULL` for feature classes without M values, and for features without *any* assigned M values.	Double	All feature classes
$ShapeXMax	The maximum X (or easting) value for the feature. `NULL` for features with no or empty geometries.	Double	All feature classes
$ShapeXMin	The minimum X (or easting) value for the feature. `NULL` for features with no or empty geometries.	Double	All feature classes
$ShapeYMax	The maximum Y (or northing) value for the feature. `NULL` for features with no or empty geometries.	Double	All feature classes
$ShapeYMin	The minimum Y (or northing) value for the feature. `NULL` for features with no or empty geometries.	Double	All feature classes
$ShapeZMax	The maximum Z value for the feature. `NULL` for feature classes without Z values.	Double	All feature classes
$ShapeZMin	The minimum Z value for the feature. `NULL` for feature classes without Z values.	Double	All feature classes
$TableName	The unqualified name of the table/feature class	String	All tables and feature classes
$QualifiedTableName	The qualified name of the table/feature class: • `<owner>.<table>` for Oracle • `<database>.<schema>.<table>` for Microsoft SQL Server and PostgreSQL • `<table>` for File Geodatabases, Shapefiles etc.	String	All tables and feature classes

3 Literal values

String values are enclosed within single quotes `'` `'`. If the string contains single quote `'`, the quote must be doubled.

```
Name = 'John'        -- string value
Name = 'John ''A'''  -- string with single quotes "John 'A'"
```

Number values are not enclosed within any characters. Floating point numbers must use a dot `.` as decimal separator.

```
Year = 2008          -- integer value
Price = 1199.9       -- floating point value
```

Date values are enclosed within sharp characters `#` `#`. For the date part, both the `month/day/year` format and the `year-month-day` format are supported. For the (optional) time part, the format `hours:minutes:seconds` is used.

```
DateOfChange = #12/31/2008#          -- date value (time is 00:00:00)
DateOfChange = #2008-12-31#          -- this format is also supported
DateOfChange = #12/31/2008 16:44:58# -- date and time value
```

Boolean values are `true` and `false`. They are not quoted.

4 Comparison operators

The **Equal, not equal, less, greater** operators are used to include only values that suit to a comparison expression. These operators can be used:

=	Equal
<>	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

The operator **`IN`** is used to only include values from a list. You can use the operator for all data types, such as numbers or strings.

```
Id IN (1, 2, 3)                     -- integer values
Price IN (1.0, 9.9, 11.5)           -- float values
Name IN ('John', 'Jim', 'Tom')      -- string values
Date IN (#12/31/2008#, #1/1/2009#)  -- date time values
"Id NOT IN (1, 2, 3)"               -- values not from the list
```

The operator **`LIKE`** is used to include only values that match a pattern with wildcards. **Wildcard** characters are `*` or `%`, they can be at the beginning of a pattern `'*value'`, at the end `'value*'`, or at both `'*value*'`. A wildcard in the middle of a pattern `'va*lue'` is **not allowed**.

```
Name LIKE 'j*'       -- values that start with 'j'
Name LIKE '%jo%'     -- values that contain 'jo'
Name NOT LIKE 'j*'   -- values that don't start with 'j'
```

If a pattern in a `LIKE` clause contains any of these special characters `*` `%` `[` `]`, those characters must be escaped in brackets `[ ]` like this `[*]`, `[%]`, `[[]` or `[]]`.

```
Name LIKE '[*]*'     -- values that starts with '*'
Name LIKE '[[]*'     -- values that starts with '['
```

5 Boolean operators

Boolean operators **`AND`**, **`OR`** and **`NOT`** are used to concatenate expressions. Operator `NOT` has precedence over `AND` and `OR`. The `AND` operator has precedence over `OR` operator. Parentheses can be used to group clauses and force precedence.

```
-- operator AND has precedence over OR operator, parenthesis are needed
City = 'Tokyo' AND (Age < 20 OR Age > 60)

-- the following examples do the same
City <> 'Tokyo' AND City <> 'Paris'
NOT City = 'Tokyo' AND NOT City = 'Paris'
NOT (City = 'Tokyo' OR City = 'Paris')
City NOT IN ('Tokyo', 'Paris')
```

6 Arithmetic operators

Arithmetic operators are addition **`+`**, subtraction **`–`**, multiplication **`*`**, division **`/`** and modulus `%`.

```
MotherAge - Age < 20"    -- people with young mother
Age % 10 = 0"            -- people with decennial birthday
```

7 String operators

Strings can be concatenated using the **`+`** character.

8 Functions

The following functions are supported:

8.1 CONVERT

Description	Converts particular expression to a specified .NET Framework Type.
Syntax	`Convert(`*`expression`*`, `*`type`*`)`
Arguments	*`expression`* -- The expression to convert. *`type`* -- The .NET Framework type to which the value will be converted. The type name must be enclosed in single quotes `'` `'`.

```
Convert(TOTAL, 'System.Int32') -- convert numeric value stored as text to an integer
```

The most common types for conversion are:

- Integer: `'System.Int32'`
- Double: `'System.Double'`
- String: `'System.String'`
- Date/Time: `'System.DateTime'`

All conversions are valid, with the following exceptions:

- **Boolean** can be coerced to and from **Byte**, **SByte**, **Int16**, **Int32**, **Int64**, **UInt16**, **UInt32**, **UInt64**, **String** and itself only.
- **Char** can be coerced to and from **Int32**, **UInt32**, **String**, and itself only.
- **DateTime** can be coerced to and from **String** and itself only.
- **TimeSpan** can be coerced to and from **String** and itself only.

8.2 LEN

Description	Gets the length of a string
Syntax	`LEN(`*`expression`*`)`
Arguments	*`expression`* -- The string to be evaluated.

```
Len(NAME) > 60 – matches names with more than 60 characters
```

8.3 ISNULL

Description	Checks an expression and either returns the checked expression or a replacement value.
Syntax	`ISNULL(`*`expression`*`, `*`replacementvalue`*`)`
Arguments	*`expression`* -- The expression to check. *`replacementvalue`* -- If expression is **`null`**, *replacementvalue* is returned.

```
IsNull(PRICE, 0) < 1000 -- use 0 for NULL values in column PRICE
```

8.4 IIF

Description	Gets one of two values depending on the result of a logical expression.
Syntax	`IIF(`*`expr`*`, `*`truepart`*`, `*`falsepart`*`)`
Arguments	*`expr`* -- The expression to evaluate. *`truepart`* -- The value to return if the expression is true. *`falsepart`* -- The value to return if the expression is false.

```
-- assign rows to groups based on column TOTAL
--   Group1: values > 100
--   Group2: values > 50 and <= 100
--   Group3: values <= 50
IIF(TOTAL > 100, 'Group1', IIF(TOTAL > 50, 'Group2', 'Group3'))
```

8.5 TRIM

Description	Removes all leading and trailing blank characters like \r, \n, \t, ' '
Syntax	`TRIM(`*`expression`*`)`
Arguments	*`expression`* -- The expression to trim.

8.6 SUBSTRING

Description	Gets a sub-string of a specified length, starting at a specified point in the string.
Syntax	`SUBSTRING(`*`expression`*`, `*`start`*`, `*`length`*`)`
Arguments	*`expression`* -- The source string for the substring. *`start`* -- Integer that specifies where the substring starts (1 indicates the first character) *`length`* -- Integer that specifies the length of the substring.

```
Substring(LASTNAME, 1, 2) = 'Aa' -- matches last names that start with 'Aa'
```

9 A note on NULL values

With some expressions, the existence of NULL values in fields can lead to unexpected results. Here is an example:

```
TYPE <> 'A'                        -- evaluates to NULL (→ not true) if TYPE is NULL
```

In this example, `NULL` is certainly not equal to `'A'`, therefore a result of `true` would be expected. The reason for this behavior is that most expressions and predicates that involve a `NULL` value evaluate to `NULL`, which in turn is treated as `'`not true`'`. For an explicit treatment of `NULL` values, the `NULL` predicates **`IS NULL`** and **`IS NOT NULL`** can be used:

```
TYPE IS NULL OR TYPE <> 'A'   -- evaluates to true if TYPE is NULL
```

Alternatively, the `IsNull()` function (see 8.3 above) is available to supply a replacement value for `NULL` values.

As illustrated by the above example, predicates involving a negation (such as `<>` or `NOT IN (…)`) are particularly affected by these unexpected results due to the presence of NULL values. Another common mistake is the explicit comparison with `NULL`, which returns `NULL` (and not true) when TYPE contains a `NULL` value:

```
TYPE = NULL                    -- incorrect, return NULL if TYPE is NULL
```

The correct version would use the `IS NULL` predicate:

```
TYPE IS NULL                   -- correct, returns true if TYPE is NULL
```

10 Example expressions

The following examples are adapted from real-world quality conditions.

10.1 Filter expression using IN combined with interval condition

The following expression matches rows with ARTID values 0-7, 32 and 33.

```
ARTID IN (32, 33) OR (ARTID >= 0 AND ARTID <= 7)
```

Note that the `BETWEEN` predicate is not supported, therefore interval conditions should be defined as *`COLUMN_NAME`* `>=` *`lower_limit`* `AND` *`COLUMN_NAME`* `<=` *`upper_limit`*.

10.2 Filter expression using IS NOT NULL and NOT IN ()

The following expression matches rows with RTE values that are not `NULL, 'UNK',` or `'N_A'`.

```
RTE IS NOT NULL AND RTE NOT IN ('UNK','N_A')
```

10.3 Filter expression using LIKE

This expression matches rows with NBIDENT values that start with `'ZH'`.

```
NBIDENT LIKE 'ZH%'
```

10.4 Filter expression using Len(), Trim(), IsNull()

This expression matches rows which have a not-NULL, non-empty NAME value:

```
Len(Trim(IsNull(NAME, ''))) > 0
```

10.5 Filter expression based on several columns

This expression matches features for specific values for STATUS and TYPE, which also have a defined value for COSTUNIT.

```
STATUS IN (1,6,8) AND TYPE = 1 AND COSTUNIT IS NOT NULL AND Len(Trim(COSTUNIT)) > 0
```

Note that the condition on COSTUNIT is equivalent to

```
Len(Trim(IsNull(COSTUNIT, ''))) > 0
```

The separate `IS NULL` predicate avoids another nested function and may make the condition easier to read.

10.6 Comparison expression using IsNull()

Comparison expression used for Parameter **validRelationConstraint** of Test **QaTouchesOther**, with use of **`IsNull()`** function to treat `NULL` as equivalent to an empty string:

```
IsNull(G1.OPERATION_GROUP, '') = IsNull(G2.OPERATION_GROUP, '')
```

Note that a direct comparison of `G1.OPERATION_GROUP = G2.OPERATION_GROUP` would return NULL (not true) if both features had a NULL value in field `OPERATION_GROUP`.

10.7 Hierarchical constraint on structured ID

This constraint is used in **QaDatasetConstraintFactory**, which allows the definition of hierarchical constraints, with nested constraints prefixed by a `+` character. The expressions at the lowest level are evaluated as constraints applicable to selections of objects defined by their parent expressions.

In this example, the constraint prefixed with `+` is only applied to rows having a value for FUNCTIONAL_LOCATION with a minimum length of 6 characters. The constraint requires that the first 6 characters of FUNCTIONAL_LOCATION are numeric and are equal to the PLANT_ID + 700000.

```
Len(FUNCTIONAL_LOCATION) >= 6
+Convert(Substring(FUNCTIONAL_LOCATION, 1, 6), 'System.Int32') = 700000 + PLANT_ID
```

10.8 Hierarchical constraint on ID that must start with a year value

In this example of a hierarchical constraint used in **QaDatasetConstraintFactory**, the constraint prefixed with `+` is only applied to rows having a value of ORDER_ID that is at least 4 characters long and which have a not-NULL value for ORDER_YEAR (numeric column). For these rows, the first 4 characters of ORDER_ID should correspond to the value of ORDER_YEAR.

```
ORDER_ID IS NOT NULL AND Len(ORDERID) >= 4 AND ORDER_YEAR IS NOT NULL
+Substring(ORDER_ID, 1, 4) = Convert(ORDER_YEAR, 'System.String')
```

10.9 Concatenated grouping expression

The following expression is used as **groupByExpression** for test **QaGroupConstraints**. The quality conditions states that a RouteID field for watercourse features must be unique for the combination of watercourse name and alias name. The expression produces concatenated strings in a readable format, facilitating the interpretation of reported errors (which include the result of the **groupByExpression** in the error description).

```
WATERCOURSE.NAME + ' (alias: ' + IsNull(WATERCOURSE.ALIASNAME, '<no alias>') + ')'
```

11 Tests processing SQL expressions in the underlying database

A small fraction of the available QA tests executes direct queries to the underlying database. The following table lists these tests.

Test	Notes
QaForeignKey	The **filterExpression** values of both tables must conform to the SQL syntax of the *underlying database*.
QaGraphicConflict	The **filterExpression** values of both feature classes must conform to the SQL syntax of the *underlying database*.
QaGroupConstraints	The **filterExpression** of the table must conform to *both* to the SQL syntax of the underlying database *and* the SQL syntax of the QA framework. **groupByExpressions** and **distinctExpressions** must conform to the QA framework SQL syntax.
QaNonEmptyGeometry	The **filterExpression** of the feature class must conform to the SQL syntax of the *underlying database*.
QaRouteMeasuresUnique	The **filterExpression** values of all feature classes must conform to the SQL syntax of the *underlying database*.
QaRowCount	The **filterExpression** values of the tables must conform to the SQL syntax of the *underlying database*.
QaUnique	The **filterExpression** of the table must conform to *both* to the SQL syntax of the underlying database *and* the SQL syntax of the QA framework.
QaUnreferencedRows	The **filterExpression** values of all tables must conform to the SQL syntax of the *underlying database*.
QaValidNonLinearSegments	The **filterExpression** of the feature class must conform to the SQL syntax of the *underlying database*.

Additionally, filter expressions for other tests are also executed in the underlying database in the following situation:

- The test uses *only* standalone tables and *no* spatial datasets (feature classes, terrain datasets, etc.), and
- the standalone tables are *not* filtered based on a spatial filter to related features, either because
    - There are no relationship classes between the table and any feature class, or

- There is a relationship class between the table and a feature class, but it is configured in the data dictionary to not be used as the basis for derived table geometries (property “Not used for derived Geometry” on association items in a data model), or
- The verification is executed without a verification perimeter (i.e. without an applicable spatial filter)

It is recommended that filter expressions used for quality conditions which *exclusively* involve standalone tables generally conform to both the SQL syntax of the underlying database, and the QA framework SQL syntax. Note that this only applies to filter expressions. Other uses of SQL expressions in tests (constraints, comparison expressions, value calculation) always use the QA framework SQL syntax as described in this document.