AMQP Filter Expressions Version 1.0

Committee Specification Draft 01

17 March 2021

This stage:
https://docs.oasis-open.org/amqp/filtex/v1.0/csd01/filtex-v1.0-csd01.docx (Authoritative)
https://docs.oasis-open.org/amqp/filtex/v1.0/csd01/filtex-v1.0-csd01.html
https://docs.oasis-open.org/amqp/filtex/v1.0/csd01/filtex-v1.0-csd01.pdf

Previous stage of Version 1.0:
N/A

Latest stage of Version 1.0:
https://docs.oasis-open.org/amqp/filtex/v1.0/filtex-v1.0.docx (Authoritative)
https://docs.oasis-open.org/amqp/filtex/v1.0/filtex-v1.0.html
https://docs.oasis-open.org/amqp/filtex/v1.0/filtex-v1.0.pdf

Technical Committee:
OASIS Advanced Message Queuing Protocol (AMQP) TC

Chairs:
Rob Godfrey (rgodfrey@redhat.com), Red Hat
Clemens Vasters (clemensv@microsoft.com), Microsoft

Editor:
Clemens Vasters (clemensv@microsoft.com), Microsoft

Related work:
This document is related to:

- *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0 Part 0: Overview*. Edited by Robert Godfrey, David Ingham, and Rafael Schloming. 29 October 2012. OASIS Standard. http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html.

Abstract:
The AMQP Filter Expressions specification describes a syntax for expressions consisting of property selectors, functions, and operators that can be used for conditional transfer operations and for configuring a messaging infrastructure to conditionally distribute, route, or retain messages.

Status:
This document was last revised or approved by the OASIS Advanced Message Queuing Protocol (AMQP) TC on the above date. The level of approval is also listed above. Check the "Latest stage" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=amqp#technical.

TC members should send comments on this document to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at https://www.oasis-open.org/committees/amqp/.

This specification is provided under the RF on RAND Terms Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing

Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:
When referencing this document, the following citation format should be used:

[Filter-Expressions-v1.0]

AMQP Filter Expressions Version 1.0. Edited by Clemens Vasters. 17 March 2021. OASIS Committee Specification Draft 01. https://docs.oasis-open.org/amqp/filtex/v1.0/csd01/filtex-v1.0-csd01.html. Latest stage: https://docs.oasis-open.org/amqp/filtex/v1.0/filtex-v1.0.html.

Notices:

Table of Contents

1 Introduction

This specification introduces AMQP type definitions for message filter expressions. Filter expressions are logical statements, implemented as AMQP “filter” archetypes, that can be evaluated against the contents of an AMQP message or against the result of other AMQP filters, and yield a Boolean “true” or “false” result.

Filter expressions are designed for use in other specifications or by AMQP applications that require durable or ephemeral configuration of message filters.

An example for such a configuration is the creation of a durable subscription on a message topic similar to the equivalent Java Message Service (JMS) construct.

This specification only defines the data types and related syntax definitions for filters and does not prescribe specific application scenarios.

- Property filter expressions allow for simple matching of metadata carried in a message against reference data held in the filter expression, and is specifically optimized for high-performance scenarios, because it does not require text parsing.
- SQL filter expressions allow for complex filter conditions. Their definition leans on the SQL-92 standard just as the familiar SQL selector syntax used by Java Message Service (JMS) specification does.
- Group filter expressions allow grouping AMQP filters logically, allowing composite expressions where the result is “true” when all, any, or none of the grouped expressions match. Group filter expressions apply to instances of any AMQP “filter” archetype, not just instances of the filter expression types defined here.

1.1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119] and [RFC8174] when, and only when, they appear in all capitals, as shown here.

When used in this specification and unless explicitly stated otherwise, the term “message” always refers to an AMQP message using the default message format of [AMQP 1.0, 3.2.16]

1.2 Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC5646] Phillips, A., Ed., and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <https://www.rfc-editor.org/info/rfc5646>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <http://www.rfc-editor.org/info/rfc8174>.

[AMQP 1.0] *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0 Part 0: Overview*. Edited by Robert Godfrey, David Ingham, and Rafael Schloming. 29 October 2012. OASIS Standard. http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html**.**

2 Filter Expressions

Filter expressions are logical statements that refer to elements of an AMQP 1.0 message using the default AMQP message format as defined in AMQP 1.0, Section 3.2.16.

Implementers of alternate message formats might reuse these filter expressions, but the behavior with other message formats is formally undefined.

2.1 Overview

Filter expressions are evaluated by an AMQP container against messages flowing out of the AMQP container. Successful evaluation of a filter expression against a message will result in a `boolean` result: `true` or `false`. Where expression evaluation fails (for instance through an attempt to compare between values of types which cannot be compared), the evaluation is neither `true` nor `false` but instead can be considered to be unknowable - that is `null`.

Property filter expressions and SQL filter expressions are evaluated referring to the fields contained within the “header”, “delivery-annotations”, “message-annotations”, “properties”, “application-properties”, and “footer” [AMQP 1.0] sections of a message.

The body (data, amqp-value, or amqp-sequence sections) may be large, unstructured and intentionally opaque to intermediary containers. For these reasons, filters against the body sections are intentionally out of scope for this specification.

2.2 Connection and Link Capabilities

On connection establishment, a partner MUST indicate whether and which filter expressions it supports through the exchange of connection capabilities (see Section 2.7.1 [AMQP 1.0]).

The capabilities allow for the partner to detect whether filter expressions as defined in this specification have been implemented and can be used with the partner container, at all. A partner MAY indicate support and yet it MAY still refuse to accept certain filters or combination of filters for some scenarios.

Each type-group of filters MUST be negotiated separately.

Capability Name	Definition
AMQP_FILTEX_SQL_V1_0	If present in the `offered-capabilities` field of the `open` or the `attach` frame, the sender of the `open` or `attach` supports the use of SQL filter expressions as defined in this specification.
AMQP_FILTEX_PROP_V1_0	If present in the `offered-capabilities` field of the `open` or the `attach` frame, the sender of the `open` or `attach` supports the use of property filter expressions as defined in this specification.
AMQP_FILTEX_GROUP_V1_0	If present in the `offered-capabilities` field of the `open` or the `attach` frame, the sender of the `open` or `attach` supports the use of group filter expressions as defined in this specification.

2.3 Error Handling

There are two kinds of errors that may occur when filters are being handled: Definitional errors and evaluation errors.

Definitional errors occur when the filter is defined, for instance when it is applied as a source filter in an attach frame, and the filter validation encounters a syntax error.

If a container establishes sending link on receipt of an unsolicited attach by its partner, and the attach from its partner contains a filter on the source, the container SHOULD validate the filter before sending its own attach with the same filter defined at its source.

When validation is performed, and the validation yields an error, or if the indicated filter is not supported, the source-side MUST detach the link with an appropriate error code, typically with the invalid-field code.

Evaluation errors occur when the filter is used at runtime and the evaluation cannot be completed, for instance by an arithmetic division by zero where the divisor stems from a message property. In this case, the message causing the error shall not be eligible for delivery. The implementation MAY choose to move the message to a different node for diagnostic purposes.

3 Grouping filter expressions

Grouping filter expressions allows logical combinations of instances of any AMQP “filter” archetype as filters.

A grouping filter expression is of the AMQP archetype “filter” as a restriction of the “list” base type and enumerates one or more objects of archetype “filter”.

Because grouping filters are filters they can themselves be grouped. Implementations supporting grouping expressions MAY restrict the depth of nesting permitted.

If any of the filter expressions contained by a grouping filter expression contains evaluates to NULL due to a failure, the entire grouping expression evaluates to NULL and the error condition of the first failing filter is propagated as the error code and reason.

3.1 all filter

This filter type groups one of multiple filters and evaluates to true if all filters it contains evaluate to true when matched against the candidate message. This is a logical “and” operation.

```
<type name="all-filter" class="restricted" source="list" provides="filter">
   <descriptor name="amqp:all-filter " code="0x00000000:0x00000100"/>
</type>
```

The filter-type is “amqp:all-filter”, with type-code 0x00000000:0x00000100

3.2 any filter

This filter type groups one of multiple filters and evaluates to true if any of the filters it contains evaluate to true when matched against the candidate message. This is a logical “or” operation.

```
<type name="any-filter" class="restricted" source="list" provides="filter">
   <descriptor name="amqp:any-filter" code="0x00000000:0x00000101"/>
</type>
```

The filter-type is “amqp:any-filter”, with type-code 0x00000000:0x00000101

3.3 not filter

This filter type groups one of multiple filters and evaluates to true if none of the filters it contains evaluate to true when matched against the candidate message.

```
<type name="not-filter" class="restricted" source="list" provides="filter">
   <descriptor name="amqp:not-filter" code="0x00000000:0x00000102"/>
</type>
```

The filter-type is “amqp:not-filter”, with type-code 0x00000000:0x00000102

4 Property Filter Expressions

Property filters expressions allow matching of message metadata against reference metadata.

A property filter expression is of the AMQP archetype “filter”. This section defines property filter types for each of these AMQP message sections: “header”, “message-annotations”, “properties”, “application-properties”, and “footer”.

Each of the property filter expression types is a map.

For the “header” and “properties” message sections that are made up of well-known fields, the map MUST correspond (name, type, and any further restrictions) to the respective message metadata section type’s field definitions.

If invalid entries are present for these expressions, validation and evaluation MUST fail.

For instance, the “properties-filter” reference expression type is a map, whereby the permissible entries of the map correspond to the fields of the AMQP “properties” type and all respective type and value restrictions apply.

Any entry referenced by the filter expression that is not present in the section is treated as if the entry was present and its value were either the defined default value for the corresponding well-known field, if applicable, or NULL otherwise. That is the case both for well-known fields as well as for references to application-defined fields.

A property filter matches the corresponding AMQP message section and is evaluated as “true” if each element of the reference data in the property filter matches the corresponding element in the AMQP message section (logical “AND”). For logical “OR” operations, combine multiple property filter expressions with an “any” grouping filter expression.

4.1 Value matching rules

The following rules apply for matching the contents of the message sections.

The term “field value” refers to the value of an entry in a message metadata section. A “reference field value” is the value in the filter expression whose key matches the name of the message metadata field.

4.1.1 Simple type matching

A reference field value in a property filter expression matches its corresponding message metadata field value if:

The reference field value is NULL, or

the reference field value is of the same type and value as the corresponding message metadata field, or

the reference field value is of the same type and value as the corresponding message metadata field when applying the [[simplified interpretation of AMQP types]], or

the reference field value is of an integer number type and the message metadata field is of a different integer number type, the reference value and the metadata field value are within the value range of both types, and the values are equal,

at least one field is a floating point value, and the reference field value is of a floating-point or integer number type and the message metadata field is of a different floating-point or integer number type, the reference value and the metadata field value are within the value range of both types, and the values are equal when treated as a floating-point, the reference field and message metadata field values are of type string or of type symbol (any combination is permitted) and the reference field value’s and the message metadata field length is identical and ordinal values of the characters in the sequence match exactly (case sensitive), or

the reference field value is of type string and that string is prefixed with a modifier (‘&’+{operator}+’:’), and the message metadata field satisfies the modifier’s matching rule:

Modifier Prefix	**Description**
&s:*suffix*	Suffix. The message metadata field matches the expression if the ordinal values of the characters of the *suffix* expression equal the ordinal values of the same number of characters trailing the message metadata field value.
&p:prefix	Prefix. The message metadata field matches the expression if the ordinal values of the characters of the *prefix* expression equal the ordinal values of the same number of characters leading the message metadata field value.
&&remaining-string	Escape prefix for case-sensitive matching of a string starting with ‘&’

All text comparisons in filter expressions are case-sensitive.

4.1.2 Map-type matching

A map in a property filter expression matches its message metadata counterpart if:

every entry in the reference map that is not NULL-valued is present (key lookup match) in the message metadata map, AND

the value of the reference map entry matches the corresponding message metadata map value following the field matching rules defined above.

The order of map entries is not significant.

4.1.3 List-type matching

A list in a property filter matches its message metadata counterpart if:

every entry in the reference list is present (positional match) in the message metadata list, AND

the value of the reference list entry matches the corresponding message metadata list value following the field matching rules defined above.

4.1.4 Array-type matching

An array in a property filter matches its message metadata counterpart if:

the reference array and the message metadata array have the same size, AND

the value of each reference array entry matches the corresponding (same position) message metadata array value following the field matching rules defined above.

4.1.5 Described type matching

AMQP described types (AMQP 1.1.2) are either restrictions or compositions of native AMQP types. Composite types are generally represented as AMQP lists (AMQP 1.3.2).

For restricted or composite types, the matching rules for the base type as described above apply. Any restriction rules defined in the described type MAY be ignored, because they are generally not available for evaluation at runtime.

4.2 Filter definitions

4.2.1 header filter

This filter type applies to the AMQP header section (AMQP 3.2.1).

The filter evaluates to *true* if all header fields enclosed in the filter expression map match the respective header field values in the message. Absent or NULL-valued header fields match the default values of the corresponding header fields for which default values are defined.

```
<type name="header-filter" class="restricted" source="map" provides="filter">
<descriptor name="amqp:header-filter" code="0x00000000:0x00000170"/>
</type>
```

The filter-type is “amqp:header-filter”, with type-code 0x00000000:0x00000170.

4.2.2 delivery-annotations filter

This filter type applies to the AMQP delivery-annotations section (AMQP 3.2.2).

The filter evaluates to *true* if all annotations enclosed in the filter expression match the respective delivery-annotation map entries in the message.

```
<type name="delivery-annotations-filter" class="restricted" source="map"
provides="filter">
<descriptor name="amqp:delivery-annotations-filter"
code="0x00000000:0x00000171"/>
</type>
```

The filter-type is “amqp:delivery-annotations-filter”, with type-code 0x00000000:0x00000171

4.2.3 message-annotations filter

This filter type applies to the AMQP message-annotations section (AMQP 3.2.3).

The filter evaluates to *true* if all annotations enclosed in the filter expression match the respective message-annotation map entries in the message.

```
<type name="message-annotations-filter" class="restricted" source="map"
provides="filter">
<descriptor name="amqp:message-annotations-filter"
code="0x00000000:0x00000172"/>
</type>
```

The filter-type is “amqp:message-annotations-filter”, with type-code 0x00000000:0x00000172

4.2.4 properties filters

This filter type applies to the immutable AMQP properties of the message (AMQP 3.2.4).

The filter evaluates to *true* if all properties enclosed in the filter expression match the respective properties in the message.

```
<type name="properties-filter" class="restricted" source="map"
provides="filter">
<descriptor name="amqp:properties-filter" code="0x00000000:0x00000173"/>
</type>
```

The filter-type is “amqp:properties-filter”, with type-code 0x00000000:0x00000173.

4.2.5 application-properties filter

This filter type applies to the AMQP application-properties section (AMQP 3.2.5).

The filter evaluates to *true* if all properties enclosed in the filter expression match the respective entries in the application-properties section in the message.

```
<type name="application-properties-filter" class="restricted" source="map"
provides="filter">
<descriptor name="amqp:application-properties-filter"
code="0x00000000:0x00000174"/>
</type>
```

The filter-type is “amqp:application-properties-filter”, with type-code 0x00000000:0x00000174.

4.3 footer filter

This filter type applies to the AMQP footer section (AMQP 3.2.9).

The filter evaluates to *true* if all footer fields enclosed in the filter expression match the respective footer section map entries in the message.

```
<type name="footer-filter" class="restricted" source="footer"
provides="filter">
<descriptor name="amqp:footer-filter" code="0x00000000:0x00000178"/>
</type>
```

The filter-type is “amqp:footer-filter”, with type-code 0x00000000:0x00000178.

5 Constant filter expressions

Constant filter expressions are provided for cases where a filter is expected, but where the evaluation shall always return true or false.

5.1 True Filter

The true filter is a constant filter that always returns evaluates to true.

```
<type name="true-filter" class="restricted" source="boolean"
provides="filter">
   <descriptor name="amqp:true-filter" code="0x00000000:0x00000110"/>
</type>
```

The filter-type is “amqp:true-filter”, with type-code 0x00000000:0x00000110

5.2 False Filter

The false filter is a constant filter that always returns false.

```
<type name="false-filter" class="restricted" source="boolean"
provides="filter">
   <descriptor name="amqp:false-filter" code="0x00000000:0x00000111"/>
</type>
```

The filter-type is “amqp:false-filter”, with type-code 0x00000000:0x00000111

6 SQL Filter Expressions

SQL filter expressions allow matching of message metadata against complex expressions that lean on the syntax of Structured Query Language (SQL) WHERE clauses.

Using SQL-derived expressions for message filtering is in widespread implementation use because the Java Message Service (JMS) message selector syntax also leans on SQL. Neither the SQL standard (ISO 9075) nor the JMS standard are used as a normative foundation or constrain the expression syntax defined in this specification, but the syntax is informed by them.

A SQL filter is of the AMQP archetype "filter" and restricts the string type to holding a text expression that follows the grammar rules described in this section. The grammar allows access to all metadata sections of any candidate message. SQL filters cannot access any of the body sections.

```
<type name="sql-filter" class="restricted" source="string" provides="filter">
<descriptor name="amqp:sql-filter" code="0x00000000:0x00000120"/>
</type>
```

The filter-type is "amqp:sql-filter", with type-code 0x00000000:0x00000120

6.1 Overview

The filter grammar defined below allows to evaluate all metadata aspects of an AMQP message.

In one of the simplest cases, a filter can check whether the 'application-properties' section contains a field 'color' with the value 'blue' like this:

```
color = 'blue'
```

because the *application-properties* section does not require qualification of its fields by default. Using the qualified notation for the application-properties section, the expression text would look like this:

```
application-properties.color = 'blue'
```

and with the shorthand notation like this:

```
a.color = 'blue'
```

The content of all other AMQP section is accessed using the qualified notation.

Probing whether the 'to' field from the 'properties' section contains a particular value is expressed like this:

`p.to = 'test'` (or: `properties.to = 'test'`)

If the developer also wants to check whether the 'content-type' field is of a particular media type class, you can combine two conditions:

```
p.to = 'test'
   AND ( p.contentType LIKE 'application/json%' OR
         p.contentType LIKE '%+json%')
```

References from multiple section can also be combined:

```
p.to = 'test'
   AND ( p.contentType LIKE 'application/json%' OR
         p.contentType LIKE '%+json%')
   AND ( a.color = 'blue' OR a.color = 'red' )
```

6.2 Data types in SQL expressions

The base type system for all filter expressions is the AMQP type system.

SQL expressions are formulated as strings, and therefore all constant values are expressed as strings. The grammar defines how integer, decimal, floating point (approximate number), string, and boolean constants are expressed, and provides a helper function DATE() to turn a properly formatted ISO8601 string constant into a timestamp value.

Constant values in SQL expressions map to the smallest corresponding AMQP type whose value space fits the constant expression's value:

SQL constant inferred type	AMQP type
Integer	Best-fit signed integer
Approximate Number	Float or double
Decimal	Decimal32, Decimal64, Decimal128
String	String
Binary	Binary
DATE()	Timestamp

6.3 Implicit Conversions and Expansions

There are no explicit type-casts or conversions in AMQP SQL expressions except for the DATE() function.

All other type conversions are implicit. A type conversion is required, for instance, when two operands of an arithmetic operation are of different types. In that case, the type of the left operand is converted into the type of the right operand if that conversion is supported, otherwise the conversion is attempted the other way around. If no implicit conversion is supported in either direction, the expression evaluation MUST fail.

An implicit expansion is required if an arithmetic operation yields a value outside of the value space of the operand(s) type(s) and a type with value space is available that can accurately represent the result.

6.3.1 Numeric Conversions and Expansions

The following implicit conversions MUST be supported for numerical types:

- All AMQP integer types (ubyte, ushort, uint, ulong, byte, short, int, long) MAY be implicitly converted to other integer types with a greater value space. An AMQP integer type MUST NOT be converted to an integer type with a smaller value space.
- All AMQP integer types MAY be implicitly converted to floating-point types with the same or greater value space for the integral portion of the floating-point number.
- All AMQP integer types MAY be implicitly converted to decimal types with the same or greater value space for the integral portion of the decimal number.
- All AMQP floating-point types (float, double) MAY be implicitly converted to other floating-point types with a greater value space. An AMQP floating-point type MUST NOT be converted to a floating-point type with a smaller value space.
- All AMQP floating-point types MAY be implicitly converted to decimal types with the same or greater value space.
- All AMQP decimal types (decimal32, decimal64, decimal128) MAY be implicitly converted to other decimal types with a greater value space. An AMQP integer type MUST NOT be converted to a decimal type with a smaller value space.

Expansions yield the next largest type that can provides a value space large enough to fit the result of an arithmetic operation. When an unsigned integer is negated using the unary '-' operator, the result is stored in a signed integer type whose value space can fit the result of the operation. When two values are multiplied and the result exceeds the value space of the operation type, the result type is expanded to fit the result of the operation.

Values that are the result of arithmetic operations and that exceed the value space of long, double, and decimal128 and therefore do not permit any further expansion cannot be handled. Such overflows are handled as the result being treated as “not a number”. Overflows MUST NOT fail the filter evaluation.

6.3.2 Timestamp conversions

AMQP defines the *timestamp* type as a 64-bit wide signed-integer UNIX epoch (IEEE1003). The following implicit conversions MUST be supported for all comparison operations where one of the operands is of type *timestamp*:

- Any AMQP integer types (ubyte, ushort, uint, ulong, byte, short, int, long) MAY be implicitly converted to a *timestamp* value.
- A *timestamp* type value MAY be implicitly converted to a *long* value.
- An AMQP string type MAY be implicitly converted IF its content is a valid ISO8601 date-expression.
  - An ISO8601 date-time expression is converted into a *timestamp* value
  - An ISO8601 duration expression is converted into a *long* value representing the milliseconds of the given duration.

The *DATE()* function (see 6.4.4.3.7) exists to allow for *timestamp* values to be defined as literals without being able to rely on an implicit conversion, for instance when a expression wants to force a date comparison operation against a property that might contain an ISO8601 string, but the operation shall be performed using date arithmetic rather than a string comparison.

6.4 Grammar Elements

This section defines the grammar elements for the text of SQL filter expressions. The notation is EBNF, interspersed with definitions.

6.4.1 Predicates

A predicate is a logical condition evaluating to Boolean “true” or “false” that is applied to expressions or other predicates. A predicate can be

- a Boolean-valued expression
- formed from the comparison of two expressions using a comparison-operator
- formed from testing for whether an AMQP field value is null-valued
- formed from testing whether an expression matches another expression inside a given set
- formed from a string pattern match
- formed from a testing whether an AMQP map entry exists
- modified using a unary logical operator
- composed with another predicate using a binary operator

```
predicate ::=
      <boolean_constant>
      | <expression> <comparison-operator> <expression>
      | <is-predicate>
      | <in-predicate>
      | <like-predicate>
      | <exists-predicate>
      | <unary-logical-operator> <predicate>
      | <predicate> <binary-logical-operator> <predicate>
      | ‘(‘ <predicate> ‘)’
```

6.4.2 Logical operators

Operators evaluate a combination of predicates with a rule.

6.4.2.1 Unary logical operators

Unary logical operators are applied as prefix to another predicate:

- the ‘NOT’ operator logically negates the predicate it prefixes

```
unary-logical-operator ::= ‘NOT’
```

6.4.2.2 Binary logical operators

Binary logical operators compose two predicates:

- The ‘AND’ operator combines the left and the right operand such that the result is ‘true’ if and only if both operands are ‘true’.
- The ‘OR’ operator combines the left and the right operand such that the result is ‘true’ if either of both operands is ‘true’.

```
binary-logical-operator ::= ‘AND’ | ‘OR’
```

6.4.2.3 Comparison operators

Comparison operators compare expressions

- The ‘=’ operator evaluates to ‘true’ if the left and the right operand expressions are of equal value.
- The ‘!=’ operator evaluates to ‘true’ if the left and the right operand expressions are not of equal value.
- The ‘<>’ operator is synonymous to the ‘!=’ operator.
- The ‘>’ operator evaluates to ‘true’ if the left operand is of greater value than the right operand.
- The ‘>=’ operator evaluates to ‘true’ if the left operand is of greater value than or of equal value to the right operand.
- The ‘<’ operator evaluates to ‘true’ if the left operand is of lesser value than the right operand.
- The ‘<=’ operator evaluates to ‘true’ if the left operand is of lesser value than or of equal value to the right operand.

Two operands are of equal value, if:

- neither operand is NULL or “not a number”
- the left operand is of the same type and value as the right operand, or
- the left operand is of an integer number type and the right operand is of a different integer number type, both operands are within the value range of both types, and the values are equal,
- if at least one of the operands is a floating point type, and the left operand is of a floating-point type or integer number type and the right operand is of a different floating-point or integer number type, and the values are equal after being both have been converted to a double floating point value,
- if at least one of the operands is a decimal point type, the left operand is of a decimal type or floating-point type or integer number type and the right operand is of a different decimal type or floating-point or integer number type, and the values are equal after being both have been converted to a decimal value,
- both operands are of type string or of type symbol (any combination is permitted) and the length is identical and ordinal values of the characters in the sequence match exactly (case sensitive).

The left operand is of greater value than the right operand if:

- neither operand is NULL or “not a number”
- the left operand is of the same type as the right operand and the value is greater, or

- the left operand is of an integer number type and the right operand is of a different integer number type, both operands are within the value range of both types, and the value of the left operand is greater than that of the right operand,
- if at least one of the operands is a floating point type, and the left operand is of a floating-point type or integer number type and the right operand is of a different floating-point or integer number type, and the value of the left operand is greater than that of the right operand after being both have been converted to a double floating point value,
- if at least one of the operands is a decimal point type, the left operand is of a decimal type or floating-point type or integer number type and the right operand is of a different decimal type or floating-point or integer number type, and the value of the left operand is greater than that of the right operand after being both have been converted to a decimal value,
- both operands are of type string or of type symbol (any combination is permitted) and the lexicographical rank of the left operand is greater than the lexicographical rank of the right operand.

The left operand is of lesser value than the right operand if:

- neither operand is NULL or “not a number”
- the left operand is neither equal to nor greater than the right operand

```
comparison-operator ::= ‘=’ | ‘<>’ | ‘!=’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’
```

6.4.3 Other logical predicates

6.4.3.1 IS NULL and IS NOT NULL predicate

The ‘IS NULL’ operator tests whether an expression is null-valued. The ‘IS NOT NULL’ operator tests whether an expression is null-valued.

Expressions cannot be tested for equality with NULL.

```
is_predicate ::= <expression> ‘IS’ [‘NOT’] ‘NULL’
```

6.4.3.2 LIKE predicate

The LIKE operator tests whether the left operand string expression is matches the pattern expression on the right.

The pattern expression is evaluated as a string. It can contain the following wildcard characters:

- %: Any string of zero or more characters.
- _: Any single character.

in order to allow for literal comparisons with the two wildcard characters, an escape character can be defined for the expression. The escape_char expression must be a string of length 1. There is no default escape character.

For example,

> *expression* LIKE 'ABC\%' ESCAPE '\'

matches ABC% literally, rather than a string that starts with ABC.

The strings match if the either the left operand and the right operand are equal under the comparison rules, or

- if the pattern expression contains the ’_’ character and that character is not prefixed by the <escape_char> value, the pattern expression matches any character in the left operand at the current character position.
- if the pattern expression contains the ’%’ character and that character is not prefixed by the <escape_char> value, the pattern expression matches any sequence of characters in the left

operand starting at the current character position. The wildcard matching MUST consume as few characters as possible. If the pattern expression extends past the ‘%’ wildcard, the wildcard matching must stop once the remaining string at the current position of the left operand expression matches the remaining pattern expression or matches a prefix of the remaining pattern expression up to a subsequent wildcard character.

```
like_predicate ::= <expression> [‘NOT’] ‘LIKE’ <pattern> [‘ESCAPE’ <escape_char>]
pattern ::= <expression>
escape_char ::= <expression>
```

6.4.3.3 IN predicate

The ‘IN’ operator tests whether the left operand is equal to at least one of the expressions in the right operand, which MUST be a set expression. Set expressions are only defined for use with the ‘IN’ operator.

```
in-predicate ::= <expression> [‘NOT’] ‘IN’ <set-expression>
set-expression ::= ( <expression> {‘,’ <expression>} )
```

6.4.3.4 EXISTS predicate

The unary ‘EXISTS’ operator tests whether the referenced field exists in the probed message. For message sections that are composite types, the operator evaluates to true if and only if there is a value at the field position. For message sections that maps, the operator evaluates to true if and only if a map entry with the respective key exists. The general rule that an omitted field is treated as if present and NULL-itvalued does not apply for the EXISTS operator. The value of the field or map entry is irrelevant.

```
exists-predicate ::= ‘EXISTS’ ‘(‘ <field> ‘)’
```

6.4.4 Expressions

Expressions are either constants, functions, references to values of elements of a message, or results of operations over other expressions.

```
<expression> ::=
      <constant>
      | <function>
      | <field_value>
      | <expression> ‘+’ | ‘-‘| ‘*’ | ‘/’ | ‘%’ <expression>
      | ‘+’ | ‘-‘ <expression>
      | ‘(‘ <expression> ‘)’
```

6.4.4.1 Expression operators

6.4.4.1.1 Unary expression operators

The following unary expression operators are defined for numerical values:

- The ‘-’ operator negates the sign of the numerical expression
- The ‘+’ operator has no effect and exists for consistency.

6.4.4.1.2 Binary expression operators

The following binary expression operators are defined for cases where the left and right operand are both numerical values (timestamps, integers, or floating point). With exception of with the ‘%’ operator, If either of the operands is a floating point expression, the result is a floating point expression.

- The ‘+’ operator adds the values of the left and the right operand expression.
- The ‘-’ operator subtracts the value of the right operand from that of the left operand expression.

- The ‘*’ operator multiplies the values of the left operand by the right operand expression.
- The ‘/‘ operator divides the value of the left operand by that of the right operand expression. If the right operand is zero, the result MUST be “not a number”.
- The ‘%’ operator yields the integer division remainder of dividing the value of the left operand by that of the right operand expression. The right operand MUST be an integer expression and the result is an integer expression.

Some numerical operations are invalid. Any numerical overflow and any division by zero (‘/’ and ‘%’ operators) MUST yield a float-typed not-a-number value, independent of the input types.

The following binary expression operators are defined for string and symbol values.

- The ‘+’ operator concatenates the left and right operand

6.4.4.2 Constant Expressions

The following expressions are available for constants. Timestamps are expressed as string constants and can be explicitly converted into *timestamp* with the DATE() function.

```
<constant> ::=
      <integer_constant> |
      <decimal_constant> |
      <approximate_number_constant> |
      <boolean_constant> |
      <string_constant> |
      <binary_constant> |
      NULL
```

6.4.4.2.1 Integer constants

An integer constant is a string of numbers that are not enclosed in quotation marks and do not contain decimal points. Negative integer constants are formally an application of the unary ‘-‘ operator to an integer constant expression. The maximum value range from zero to 2^{64}-1.

```
integer_constant ::= <digit> {<digit>}
```

6.4.4.2.2 Decimal constants

A decimal constant is a string of numbers that are not enclosed in quotation marks, and contain a decimal point. Negative decimal constants are formally an application of the unary ‘-‘ operator to a decimal constant expression. The maximum value range is equivalent to IEEE 754, $\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$.

```
decimal_constant ::= <digit> {<digit>} ‘.’ <digit> {<digit>}
```

6.4.4.2.3 Approximate number constants

An approximate constant is a string of numbers that are not enclosed in quotation marks, and use scientific notation, with the coefficient being a decimal constant and the exponent being a integer constant, separated by the literal ‘E’. Negative constants are formally an application of the unary ‘-‘ operator to a constant expression. The maximum value range is equivalent to IEEE 754, $\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$.

The literal ‘INF’ represents the IEEE 754 infinity value. The literal ‘NAN’ expression represents the IEEE 754 not-a-number value.

```
approximate_number_constant ::= <digit> {<digit>} ‘.’ <digit> {<digit>} ‘E‘ <digit>
{<digit>} | INF | NAN
```

6.4.4.2.4 Boolean constants

A Boolean constant is a string of letters that are not enclosed in quotation marks and either spell out TRUE or FALSE.

```
boolean_constant :=  TRUE | FALSE
```

6.4.4.2.5 String constants

A string constant is a string of arbitrary text consisting of any valid printable Unicode characters surrounded by single or double quotation marks. A quotation mark inside the string is represented by two consecutive quotation marks.

```
string_constant ::= { ‘ | “ } <any> [<any>] { ‘ | “ }
```

6.4.4.2.6 Binary constants

A binary constant is a string of pairs of hexadecimal digits prefixed by ‘0x’ that are not enclosed in quotation marks.

```
decimal_constant ::= ‘0x’ <hex-digit><hex-digit> {<hex-digit><hex-digit>}
```

6.4.4.3 Functions

A function is an operation that cannot be expressed with the operators defined herein. A function accepts a comma-separated list of expressions as arguments and yields an expression as a return value.

Implementers MAY define their own functions; such functions MUST be prefixed with a vendor prefix. Functions defined herein do not carry a prefix.

If a function is not understood by an implementation, it MUST be evaluated as NULL.

```
function ::= <function_name> ‘(‘ <expression’> [ ‘,’ <expression> ] ‘)’

function_name ::= <vendor_prefix> ‘:’ <identifier> | UPPER | LOWER | UTC | DATE

vendor_prefix ::= <identifier>
```

6.4.4.3.1 LOWER function

The LOWER(string [,lang_tag]) function converts a string into its corresponding lowercase representation, following Unicode case folding rules. The function exists to facilitate case-insensitive comparison of strings.

It accepts a string and an optional language tag identifier, compliant with [RFC5646]. if the language tag identifier is omitted, the system default locale language is used for the case folding operation.

6.4.4.3.2 UPPER function

The UPPER(string [,lang_tag]) function converts a string into its corresponding uppercase representation, following Unicode case folding rules. The function exists to facilitate case-insensitive comparison of strings.

It accepts a string and an optional language tag identifier, compliant with [RFC5646]. if the language tag identifier is omitted, the system default locale language is used for the case folding operation.

6.4.4.3.3 LEFT function

The LEFT(string, count) function yields the left-side prefix of the given string, up to *count* characters. Count MUST be a positive integer; a negative value MUST cause an error. The function succeeds if the input string has fewer than *count* characters and then just returns the input string.

6.4.4.3.4 RIGHT function

The RIGHT(string, count) function yields the right-side suffix of the given string, up to *count* characters. Count MUST be a positive integer; a negative value MUST cause an error. The function succeeds if the input string has fewer than *count* characters and then just returns the input string.

6.4.4.3.5 SUBSTRING function

The SUBSTRING(string, start, count) function yields the substring of the given string that starts at *start* (counting starts at 1) with up to *count* characters. *Start* and *count* MUST be positive integers; a negative value MUST cause an error. The function succeeds if the input string has fewer than *count* characters and then just returns the input string from the *start* index onwards. If the *start* index points past the end of the string, the function returns an empty string.

6.4.4.3.6 UTC function

The UTC function yields the current UTC time as a timestamp value. The function has no parameters. The function exists to, for instance, facilitate filtering of messages based on expiration times contained in nonstandard locations.

The UTC function is OPTIONAL for AMQP source filter expressions as an implementation might require filters to always yield the same result for each message, and comparing a time value inside the message against the ‘current’ time might run afoul of this requirement.

6.4.4.3.7 DATE function

The DATE function accepts an ISO 8601 string expression and yields the corresponding timestamp value. Any unqualified time expressions MUST assume UTC as the local time zone.

6.4.4.4 Field References and Values

All values in the sections of a message MUST be referenced using field_value expressions.

The field_value expression yields the value of the referenced field or a referenced element of said value.

Since operators can only work with simple types and strings, field_value expressions not only allow referring to a field by section and name, but also allow directly referencing a particular element of an array held by the field or a particular sub-field or map entry held by the field. This is detailed in the next section.

The evaluation result of a field_value expression MUST be a simple type, string, symbol, or NULL.

```
field_value :=
       <field>
       | <field>‘.’<composite_type_reference>
       | <field><array_element_reference>
```

The field element refers to a field inside a message section. When the section is omitted, the assumed section is ‘application-properties’.

```
field ::= [<section> ’.’]<field_name>
field_name ::=  <identifier> | <section_field_name>
```

The section is a qualifier for the section to which the field reference applies. The section qualifier ‘header’, with shorthand ‘h’, refers to the respective section in the AMQP message, and the other qualifiers refer to the same-named sections equivalently.

```
section ::=

       {‘header’ | ‘h’ } |
       { ‘footer’ | ‘f’ } |
       { ‘properties’ | ‘p’ } |
       { ‘application_properties’ | ‘a’ } |
       { ‘delivery_annotations’ | ‘d’ } |
       { ‘message_annotations’ | ‘m’ }
```

The header and properties section of the AMQP message are described types, with the respective field names known only at compilation time. “section_field_name” refers to the well-known names in the respective sections, e.g. to “reply-to” in “properties”.

```
section_field_name ::= <identifier>
```

For section field names it is a syntax error if the name is not a well-known field name in the respective section, and both validation and evaluation of the SQ filter expression MUST fail.

6.4.4.5 Composite Type and Array Value References

Values inside fields that are map-typed can be referenced with a map_reference. The entry_name refers to an entry in the map. If respective entry is not present, the expression MUST yield NULL. Named references to fields of described types cannot be expressed.

```
map_reference :=
       <entry_name>
       | <entry_name> ’.’ <map_reference>
       | <entry_name> <array_reference>
<entry_name> ::=  <identifier>
```

Values inside fields that are list- or array-typed can be referenced with a positional_reference subexpression. The expression inside the square brackets MUST be an integer number and SHOULD be within the range of the list or array. Access outside the array or list range MUST yield NULL.

```
<positional_reference> := ‘[‘ <expression> ‘]’
```

6.4.4.6 Identifier Syntax

The following syntax applies to identifiers.

‘Delimited identifiers’ allow using identifiers with special characters, including whitespace and reserved words, inside of SQL expressions that would otherwise cause parsing errors. Control characters are not permitted. If needed, the ‘[‘ and ‘]’ characters are used as their own escape character, meaning ‘]]’ represents a literal ‘]’ and ‘[[‘ represents a literal ‘[‘.

```
identifier ::=  <regular_identifier> | <delimited_identifier>
regular_identifier ::= <letter> {<letter> | <underscore> | <digit> }
delimited_identifier ::= ‘[‘ {<letter> | <symbol> | <digit> } ‘]’


letter ::= Unicode Letter
symbol ::= Unicode Symbol | Unicode Punctuation | Unicode Separator
digit ::= Unicode Numeric Decimal
hex-digit ::= Unicode Numeric Decimal | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’
```

7 Security Considerations

7.1 Complex expressions

Very complex expressions might consume significant compute resources, especially when an attacker sets them as a source filter and then crafts messages such that the messages exercise the full complexity of the expression.

Also consider that source filters are configured as filter sets, meaning multiple filters might be evaluated for every message.

An implementation SHOULD guard against excessive filter expression complexity. This might be accomplished by a combination of the following and further measures:

- Impose a limit on elements permissible in a filter set
- Impose a limit on the complexity of each filter expression. This might limit the count of filters in a grouping filter expression, the depth of nesting of grouping filter expressions, the text length of a SQL filter expression, or the number of conditions in a SQL filter expression,
- Impose a limit on the execution time of a filter expression.

When such security-motivated limits are hit, the error condition SHOULD be logged in the administrator's log and SHOULD be masked as a generic server error in response to the remote client.

7.2 Inducing errors

Messages might be sent with knowledge of an existing filter and with the intent to cause evaluation errors or exploit known evaluation defects and therefore flood log files. An implementation SHOULD guard against induced errors by a combination of the following and further measures:

- Keep track of the source of messages to determine which sender causes excessive error rates during filter evaluation. The sender might be brought to the attention of an operator, or disconnected, or even banned.
- Keep logging information terse.

7.3 Injection

SQL filter expression strings SHOULD NOT be formed whole or in part from raw end-user input strings. A raw user input expression might for instance use alter the shape of a predefined SQL filter expression when the user input is applied to a string:

```
color='%s'
```

If the inserted string contains a closing quote character, the input could expand the expression to

```
color='' or 1=1/0 or other=''
```

Applications SHOULD therefore always sanitize user input that needs to be included into SQL expressions.

References to AMQP message properties cannot cause this issue.

8 Conformance

All filter expressions defined in this specification MAY be implemented singly, meaning that a complete implementation of all expression types is not required for conformance. It is RECOMMENDED to implement all filter expression types.

To ensure interoperability, each filter expression type that is selected for implementation MUST be implemented completely and as specified here for the implementation to be conformant.

None of the filter expression types defined herein has optional features that can be omitted.

Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

Alan Conway, Red Hat
Rob Godfrey, Red Hat
Keith Wall, Red Hat
Robbie Gemmell, Red Hat
Justin Ross, Red Hat
Ted Ross, Red Hat
Oleksandr Rudyy, JP Morgan
Xin Chen, Microsoft

Appendix B. Revision History

Revision	Date	Editor	Changes Made
[Rev number]	[Rev Date]	[Modified By]	[Summary of Changes]

Appendix C. Notices