Data Manipulation Language (DML)

In this chapter are the additions and improvements that have been added to the SQL data manipulation language subset in Firebird 2.5.

Quick links

RegEx search support using SIMILAR TO

Adriano dos Santos Fernandes

Tracker reference CORE-769.

A new SIMILAR TO predicate is introduced to support regular expressions. The predicate's function is to verify whether a given SQL-standard regular expression matches a string argument. It is valid in any context that accepts Boolean expressions, such as WHERE clauses, CHECK constraints and PSQL IF() tests.

Syntax patterns

```
<similar predicate> ::=
  <value> [ NOT ] SIMILAR TO <similar pattern> [ ESCAPE <escape character> ]
<similar pattern> ::= <character value expression: regular expression>

<regular expression> ::=
     <regular term>
   | <regular expression> <vertical bar> <regular term>

<regular term> ::=
     <regular factor>
   | <regular term> <regular factor>

<regular factor> ::=
     <regular primary>
   | <regular primary> <asterisk>
   | <regular primary> <plus sign>
   | <regular primary> <question mark>
```

```
    | <regular primary> <repeat factor>

<repeat factor> ::=
   <left brace> <low value> [ <upper limit> ] <right brace>

<upper limit> ::= <comma> [ <high value> ]

<low value> ::= <unsigned integer>

<high value> ::= <unsigned integer>

<regular primary> ::=
     <character specifier>
   | <percent>
   | <regular character set>
   | <left paren> <regular expression> <right paren>

<character specifier> ::=
     <non-escaped character>
   | <escaped character>

<regular character set> ::=
     <underscore>
   | <left bracket> <character enumeration>... <right bracket>
   | <left bracket> <circumflex> <character enumeration>... <right bracket>
   | <left bracket> <character enumeration include>... <circumflex>
<character enumeration exclude>...  <right bracket>

<character enumeration include> ::= <character enumeration>

<character enumeration exclude> ::= <character enumeration>

<character enumeration> ::=
     <character specifier>
   | <character specifier> <minus sign> <character specifier>
   | <left bracket> <colon> <character class identifier> <colon> <right
bracket>

<character specifier> ::=
     <non-escaped character>
   | <escaped character>

<character class identifier> ::=
     ALPHA
   | UPPER
   | LOWER
   | DIGIT
   | SPACE
   | WHITESPACE
   | ALNUM
```

Note:

1. **<non-escaped character>** is any character except <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace> and <escape character>.

2. **<escaped character>** is the <escape character> succeeded by one of <left bracket>, <right bracket>, <left paren>, <right paren>, <vertical bar>, <circumflex>, <minus sign>, <plus sign>, <asterisk>, <underscore>, <percent>, <question mark>, <left brace> or <escape character>.

back to top of page

Table 10.1. Character class identifiers

Identifier	Description	Note
ALPHA	All characters that are simple latin letters (a-z, A-Z).	Includes latin letters with accents when using accent-insensitive collation.
UPPER	All characters that are simple latin uppercase letters (A-Z).	Includes lowercase latters when using case-insensitive collation.
LOWER	All characters that are simple latin lowercase letters (a-z).	Includes uppercase latters when using case-insensitive collation.
DIGIT	All characters that are numeric digits (0-9).	
SPACE	All characters that are the space character (ASCII 32).	
WHITESPACE	All characters that are whitespaces (vertical tab (9), newline (10), horizontal tab (11), carriage return (13), formfeed (12), space (32)).	
ALNUM	All characters that are simple latin letters (ALPHA) or numeric digits (DIGIT).	

back to top of page

Usage Guide

Return true for a string that matches <regular expression> or <regular term>:

```
<regular expression> <vertical bar> <regular term>

'ab' SIMILAR TO 'ab|cd|efg'    -- true
'efg' SIMILAR TO 'ab|cd|efg'   -- true
'a' SIMILAR TO 'ab|cd|efg'     -- false
```

Match zero or more occurrences of <regular primary>: <regular primary> <asterisk>

```
'' SIMILAR TO 'a*'             -- true
'a' SIMILAR TO 'a*'            -- true
'aaa' SIMILAR TO 'a*'          -- true
```

Match one or more occurrences of <regular primary>: <regular primary> <plus sign>

```
'' SIMILAR TO 'a+'              -- false
'a' SIMILAR TO 'a+'             -- true
'aaa' SIMILAR TO 'a+'           -- true
```

Match zero or one occurrence of <regular primary>: <regular primary> <question mark>

```
'' SIMILAR TO 'a?'              -- true
'a' SIMILAR TO 'a?'             -- true
'aaa' SIMILAR TO 'a?'           -- false
```

Match exact <low value> occurrences of <regular primary>: <regular primary> <left brace> <low value> <right brace>

```
'' SIMILAR TO 'a{2}'            -- false
'a' SIMILAR TO 'a{2}'           -- false
'aa' SIMILAR TO 'a{2}'          -- true
'aaa' SIMILAR TO 'a{2}'         -- false
```

Match <low value> or more occurrences of <regular primary>: <regular primary> <left brace> <low value> <comma> <right brace>

```
'' SIMILAR TO 'a{2,}'           -- false
'a' SIMILAR TO 'a{2,}'          -- false
'aa' SIMILAR TO 'a{2,}'         -- true
'aaa' SIMILAR TO 'a{2,}'        -- true
```

Match <low value> to <high value> occurrences of <regular primary> <regular primary> <left brace> <low value> <comma> <high value> <right brace>:

```
'' SIMILAR TO 'a{2,4}'          -- false
'a' SIMILAR TO 'a{2,4}'         -- false
'aa' SIMILAR TO 'a{2,4}'        -- true
'aaa' SIMILAR TO 'a{2,4}'       -- true
'aaaa' SIMILAR TO 'a{2,4}'      -- true
'aaaaa' SIMILAR TO 'a{2,4}'     -- false
```

Match any (non-empty) character: <underscore>

```
'' SIMILAR TO '_'               -- false
'a' SIMILAR TO '_'              -- true
'1' SIMILAR TO '_'              -- true
'a1' SIMILAR TO '_'             -- false
```

Match a string of any length (including empty strings): <percent>

```
'' SIMILAR TO '%'               -- true
'az' SIMILAR TO 'a%z'           -- true
'a123z' SIMILAR TO 'a%z'        -- true
```

```
'azx' SIMILAR TO 'a%z'          -- false
```

Group a complete <regular expression> to use as one single <regular primary> as a sub-expression: <left paren> <regular expression> <right paren>

```
'ab' SIMILAR TO '(ab){2}'       -- false
'aabb' SIMILAR TO '(ab){2}'     -- false
'abab' SIMILAR TO '(ab){2}'     -- true
```

Match a character identical to one of <character enumeration>: <left bracket> <character enumeration>... <right bracket>

```
'b' SIMILAR TO '[abc]'          -- true
'd' SIMILAR TO '[abc]'          -- false
'9' SIMILAR TO '[0-9]'          -- true
'9' SIMILAR TO '[0-8]'          -- false
```

Match a character not identical to one of <character enumeration>: <left bracket> <circumflex> <character enumeration>... <right bracket>

'b' SIMILAR TO '[^abc]' – false 'd' SIMILAR TO '[^abc]' – true Match a character identical to one of <character enumeration include> but not identical to one of <character enumeration exclude>: <left bracket> <character enumeration include>... <circumflex> <character enumeration exclude>...

```
'3' SIMILAR TO '[[:DIGIT:]^3]' -- false
'4' SIMILAR TO '[[:DIGIT:]^3]' -- true
```

Match a character identical to one character included in <character class identifier>. Refer to the table of Character class identifiers, above. May be used with <circumflex> to invert the logic (see above): <left bracket> <colon> <character class identifier> <colon> <right bracket>

```
'4' SIMILAR TO '[[:DIGIT:]]'    -- true
'a' SIMILAR TO '[[:DIGIT:]]'    -- false
'4' SIMILAR TO '[^[:DIGIT:]]'   -- false
'a' SIMILAR TO '[^[:DIGIT:]]'   -- true
```

back to top of page

Examples

```
create table department (
  number numeric(3) not null,
  name varchar(25) not null,
  phone varchar(14)
  check (phone similar to '\([0-9]{3}\) [0-9]{3}\-[0-9]{4}' escape '\')
);

insert into department
  values ('000', 'Corporate Headquarters', '(408) 555-1234');
```

```
insert into department
  values ('100', 'Sales and Marketing', '(415) 555-1234');
insert into department
  values ('140', 'Field Office: Canada', '(416) 677-1000');

insert into department
  values ('600', 'Engineering', '(408) 555-123'); -- check constraint
violation

select * from department
  where phone not similar to '\([0-9]{3}\) 555\-%' escape '\';
```

back to top of page

Hex literal support

Bill Oliver

Adriano dos Santos Fernandes

Tracker reference CORE-1760.

Support for hexadecimal numeric and binary string literals has been introduced.

Syntax patterns

```
<numeric hex literal> ::=
    { 0x | 0X } <hexit> [ <hexit>... ]

<binary string literal> ::=
    { x | X } <quote> [ { <hexit> <hexit> }... ] <quote>

<digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<hexit> ::=
    <digit> | A | B | C | D | E | F | a | b | c | d | e | f
```

back to top of page

Numeric hex literals

- The number of <hexit> in the string cannot exceed 16.
- If the number of <hexit> is greater than eight, the constant data type is a signed BIGINT. If it is eight or less, the datatype is a signed INTEGER.

Tip: That means 0xF0000000 is -268435456 and 0x0F0000000 is 4026531840.

Binary string literals

The resulting string is defined as a CHAR(n/2) CHARACTER SET OCTETS, where n is the number of <hexit>.

Examples

```
select 0x10, cast('0x0F0000000' as bigint)
  from rdb$database;
select x'deadbeef'
  from rdb$database;
```

back to top of page

Important change to GEN_UUID() function

Adriano dos Santos Fernandes

Prior to Firebird 2.5.2, the built-in function GEN_UUID() was returning completely random strings, making it non-compliant with the RFC-4122 (UUID specification). From Firebird 2.5.2 forward, GEN_UUID() returns a compliant UUID version 4 string, where some bits are reserved and the others are random.

Note: The string format of a compliant UUID is

```
XXXXXXXX-XXXX-4XXX-YXXX-XXXXXXXXXXXX
```

where 4 is fixed (version) and Y is 8, 9, A or B.

See Tracker item CORE-3238.

back to top of page

New UUID conversion functions

Adriano dos Santos Fernandes

Tracker references CORE-1656 and CORE-1682.

Two new built-in functions, UUID_TO_CHAR and CHAR_TO_UUID, enable conversion between a UUID in the form of a CHAR(16) OCTETS string and the RFC4122-compliant form.

Important for big-Endian servers:

It was discovered that CHAR_TO_UUID and UUID_TO_CHAR worked incorrectly in Firebird 2.5 and 2.5.1 on big-Endian servers, where bytes or characters were swapped and went into the wrong positions when converting. The bug was fixed in versions 2.5.2 and above: see Tracker item CORE-3887. However, it means that, from v.2.5.2 onward, CHAR_TO_UUID and UUID_TO_CHAR return different values than in the earlier versions, for the same input parameter.

CHAR_TO_UUID()

The function CHAR_TO_UUID() converts the CHAR(32) ASCII representation of a UUID (XXXXXXXXXXXX-XXXX-XXXX-XXXXXXXXXXXX) to the CHAR(16) OCTETS representation, optimized for storage.

Syntax model

```
CHAR_TO_UUID( <string> )
```

Example

```
select char_to_uuid('93519227-8D50-4E47-81AA-8F6678C096A1')
  from rdb$database;
```

UUID_TO_CHAR()

The function UUID_TO_CHAR() converts a CHAR(16) OCTETS UUID (as returned by the GEN_UUID() function) to the CHAR(32) ASCII representation (XXXXXXXX-XXXX-XXXX-XXXXXXXXXXXXXXXX).

Syntax model

```
UUID_TO_CHAR( <string> )
```

Example

```
select uuid_to_char(gen_uuid())
  from rdb$database;
```

back to top of page

SOME_COL = ? OR ? IS NULL predication

Adriano dos Santos Fernandes

Tracker reference CORE-2298.

By popular request, particularly from Delphi programmers, support has been implemented for a predication that is able to OR test both the equivalence between a column and a parameter and whether the value passed to the parameter is NULL. This construct is often desired as a way to avoid the need to prepare one query to request a filtered result set and another for the same query without the filter.

Users of Delphi and other programming interfaces that apply client-side object names to parameters wanted the ability for the DSQL engine to recognise a usage like the following:

```
WHERE col1 = :param1 OR :param1 IS NULL
```

At the API level, the language interface translates the query to

```
WHERE col1 = ? OR ? IS NULL
```

That presented two problems:

1. While the programmer treated the parameter :param1 as though it were a single variable with two references, the API could not: it is presented with two parameters.

2. The second parameter is of an unknown data type and the program has no way to assign to it.

What was needed to solve this problem was to introduce a new data type to handle the ? is NULL condition and teach Firebird to do the right thing when it received such a request.

The implementation works like this. To resolve the first problem, the request must supply two parameters (for our Delphi example):

```
WHERE col1 = :param1 OR :param2 IS NULL
```

- If param1 is not NULL, the language interface is required to assign the correct value for the first parameter, set the XSQLVAR.sqlind to NOT NULL and leave XSQLVAR.sqldata NULL.
- If param2 is NULL, the language interface is required to set the XSQLVAR.sqlind of both parameters to NULL and leave the XSQLVAR.sqldata NULL.

In other words, for the parameter (?) in ? IS NULL:

- XSQLVAR.sqlind should be set in accordance with NULL/NON-NULL state of the parameter. This is the type of parameter that is described by the new constant SQL_NULL.
- The XSQLVAR.sqldata of a SQL_NULL type of parameter should always be passed by the client application as a NULL pointer and should never be accessed.

NULL specified in the output set: When NULL is specified as an output constant (select NULL from ...), it continues to be described as CHAR(1), rather than by SQL_NULL. That may change in a future version.

back to top of page

Extension to LIST() function

Adriano dos Santos Fernandes

Tracker reference CORE-1453.

A string expression is now allowed as the delimiter argument of the LIST() function.

Example

```
SELECT
  DISCUSSION_ID,
  LIST(COMMMENT, ASCII_CHAR(13))
```

```
FROM COMMENTS
  GROUP BY DISCUSSION_ID;
```

back to top of page

Extension to DATEADD and DATEDIFF() functions

Adriano dos Santos Fernandes

The WEEK unit was introduced for functions DATEADD and DATEDIFF.

MILLISECOND, SECOND, MINUTE and HOUR units are no longer invalid units to use with DATE **arguments**.

back to top of page

BIN_NOT() function added

Adriano dos Santos Fernandes

Completing the set of built-in binary functions added in v.2.1, new function BIN_NOT() returns the result of a bitwise NOT operation performed on its argument.

Syntax pattern

```
BIN_NOT( <number> )
```

Example

```
select bin_not(flags) from x;
```

back to top of page

Write to temporary tables in a read-only database

Vladyslav Khorsun

(V.2.5.1) Write operations to global temporary tables in a read-only database are enabled.

Tracker reference CORE-3399.

back to top of page

Optimizer improvements

Changes in optimizer logic that address recognised problems include:

CROSS JOIN logic (D. Yemanov)

When a CROSS JOIN involved an empty table, the optimizer had no special logic to detect that the query was futile and return the empty set immediately. That shortcut logic has now been implemented.

Tracker reference CORE-2200.

Note: The same change was implemented in v.2.1.2.

Derived tables (A. dos Santos Fernandes)

The limit on the number of contexts available when using derived tables has been raised.

Tracker reference CORE-2029.

Timing of DEFAULT evaluation (A. dos Santos Fernandes)

Under rare conditions, the early evaluation of a DEFAULT value or expression defined for a column might give rise to a confused situation regarding the evaluation of an input value supplied for that column. The possibility was addressed by deferring the evaluation of DEFAULT and not actually performing the evaluation at all unless it was actually necessary.

Tracker reference CORE-1842.

Index use for NOT IN searches (A. dos Santos Fernandes)

Better performance has been achieved for the NOT IN predicate by enabling the use of an index.

Tracker reference CORE-1137.

Undo log memory consumption (D. Yemanov)

Excessive memory consumption by the *Undo* log after a lengthy series of updates in a single transaction has been avoided.

Tracker reference CORE-1477.

back to top of page

Other improvements

Other changes to smooth out the little annoyances include:

FREE_IT error detection (A. dos Santos Fernandes)

Previously, a UDF declared with FREE_IT would crash if the pointer returned had not been allocated by the ib_util_malloc() function. Now, such a condition is detected, an exception is thrown and the pointer remains in its allocated state.

Tracker reference CORE-1937.

Expression evaluation not supported message improved (C. Valderrama)

A number of secondary GDS codes were introduced to provide more details about an operation that fails with an Expression evaluation not supported exception, for example:

```
'Argument for @1 in dialect 1 must be string or numeric'
'Strings cannot be added to or subtracted from DATE or TIME types'
'Invalid data type for subtraction involving DATE, TIME or TIMESTAMP types'
  etc.
```

These detailed messages follow the GDS code for the isc_expression_eval_err (expression evaluation not supported) error in the status vector.

Tracker reference CORE-1799.