Data Manipulation Language (DML)

In this chapter are the additions and improvements that have been added to the SQL data manipulation language subset in the Firebird 2 development cycle. Those marked as introduced in v.2.1 are available only to ODS 11.1 and higher databases.

Important: A new configuration parameter, named RelaxedAliasChecking was added to the firebird.conf in Firebird 2.1 to permit a slight relaxation of the Firebird 2.0.x restrictions on mixing relation aliases and table names in a query (see DSQL parsing of table names is stricter, below).

This parameter will not be a permanent fixture in Firebird but is intended as a migration aid for those needing time to adjust existing code. More information under RelaxedAliasChecking in the chapter New configuration parameters and changes.

Common table expressions

Vlad Khorsun

Based on work by Paul Ruizendaal for the Fyracle project.

(**v.2.1**) A common table expression (CTE) is like a view that is defined locally within a main query. The engine treats a CTE like a derived table and no intermediate materialisation of the data is performed.

Benefits of CTEs

Using CTEs allows you to specify dynamic queries that are recursive:

- The engine begins execution from a non-recursive member.
- For each row evaluated, it starts executing each recursive member one-by-one, using the current values from the outer row as parameters.
- If the currently executing instance of a recursive member produces no rows, execution loops back one level and gets the next row from the outer result set.

The memory and CPU overhead of a recursive CTE is much less than that of an equivalent recursive stored procedure.

Recursion limit

Currently the recursion depth is limited to a hard-coded value of 1024.

Syntax and rules for CTEs

```
select :
```

```
  select_expr for_update_clause lock_clause
select_expr :
  with_clause select_expr_body order_clause rows_clause
            | select_expr_body order_clause rows_clause
with_clause :
  WITH RECURSIVE with_list | WITH with_list
with_list :
  with_item | with_item ',' with_list
with_item :
 symbol_table_alias_name derived_column_list
    AS '(' select_expr ')'
select_expr_body :
  query_term
    | select_expr_body UNION distinct_noise query_term
                | select_expr_body UNION ALL query_term
```

A less formal representation:

```
WITH [RECURSIVE]
  CTE_A [(a1, a2, …)]
  AS ( SELECT … ),

CTE_B [(b1, b2, …)]
  AS ( SELECT … ),
...
SELECT ...
  FROM CTE_A, CTE_B, TAB1, TAB2 ...
 WHERE ...
```

Rules for non-recursive CTEs

- Multiple table expressions can be defined in one query.
- Any clause legal in a SELECT specification is legal in table expressions.
- Table expressions can reference one another.
- References between expressions should not have loops.
- Table expressions can be used within any part of the main query or another table expression.
- The same table expression can be used more than once in the main query.
- Table expressions (as subqueries) can be used in INSERT, UPDATE and DELETE statements.
- Table expressions are legal in PSQL code.
- WITH statements can not be nested.

Example of a non-recursive CTE

```
WITH
  DEPT_YEAR_BUDGET AS (
    SELECT FISCAL_YEAR, DEPT_NO,
        SUM(PROJECTED_BUDGET) AS BUDGET
      FROM PROJ_DEPT_BUDGET
    GROUP BY FISCAL_YEAR, DEPT_NO
```

```
   )
SELECT D.DEPT_NO, D.DEPARTMENT,
  B_1993.BUDGET AS B_1993, B_1994.BUDGET AS B_1994,
       B_1995.BUDGET AS B_1995, B_1996.BUDGET AS B_1996
FROM DEPARTMENT D
  LEFT JOIN DEPT_YEAR_BUDGET B_1993
    ON D.DEPT_NO = B_1993.DEPT_NO
    AND B_1993.FISCAL_YEAR = 1993
   LEFT JOIN DEPT_YEAR_BUDGET B_1994
    ON D.DEPT_NO = B_1994.DEPT_NO
    AND B_1994.FISCAL_YEAR = 1994
  LEFT JOIN DEPT_YEAR_BUDGET B_1995
    ON D.DEPT_NO = B_1995.DEPT_NO
     AND B_1995.FISCAL_YEAR = 1995
  LEFT JOIN DEPT_YEAR_BUDGET B_1996
    ON D.DEPT_NO = B_1996.DEPT_NO
    AND B_1996.FISCAL_YEAR = 1996

WHERE EXISTS (
  SELECT * FROM PROJ_DEPT_BUDGET B
  WHERE D.DEPT_NO = B.DEPT_NO)
```

Rules for recursive CTEs

- A recursive CTE is self-referencing (has a reference to itself).
- A recursive CTE is a UNION of recursive and non-recursive members:
  - At least one non-recursive member (anchor) must be present.
  - Non-recursive members are placed first in the UNION.
  - Recursive members are separated from anchor members and from one another with UNION ALL clauses, i.e.,

```
non-recursive member (anchor)
UNION [ALL | DISTINCT]
non-recursive member (anchor)
UNION [ALL | DISTINCT]
non-recursive member (anchor)
UNION ALL
recursive member
UNION ALL
recursive member
```

- References between CTEs should not have loops.
- Aggregates (DISTINCT, GROUP BY, HAVING) and aggregate functions (SUM, COUNT, MAX etc.) are not allowed in recursive members.
- A recursive member can have only one reference to itself and only in a FROM clause.
- A recursive reference cannot participate in an outer join.

Example of a recursive CTE

```
WITH RECURSIVE
```

```
  DEPT_YEAR_BUDGET AS
  (
    SELECT FISCAL_YEAR, DEPT_NO,
      SUM(PROJECTED_BUDGET) AS BUDGET
    FROM PROJ_DEPT_BUDGET
  GROUP BY FISCAL_YEAR, DEPT_NO
),

DEPT_TREE AS
(
  SELECT DEPT_NO, HEAD_DEPT, DEPARTMENT,
      CAST('' AS VARCHAR(255)) AS INDENT
    FROM DEPARTMENT
  WHERE HEAD_DEPT IS NULL

  UNION ALL

  SELECT D.DEPT_NO, D.HEAD_DEPT, D.DEPARTMENT,
  H.INDENT || ' '
    FROM DEPARTMENT D
    JOIN DEPT_TREE H
      ON D.HEAD_DEPT = H.DEPT_NO
  )

  SELECT D.DEPT_NO,
D.INDENT || D.DEPARTMENT AS DEPARTMENT,
B_1993.BUDGET AS B_1993,
B_1994.BUDGET AS B_1994,
B_1995.BUDGET AS B_1995,
B_1996.BUDGET AS B_1996

  FROM DEPT_TREE D
    LEFT JOIN DEPT_YEAR_BUDGET B_1993
      ON D.DEPT_NO = B_1993.DEPT_NO
      AND B_1993.FISCAL_YEAR = 1993
    LEFT JOIN DEPT_YEAR_BUDGET B_1994
      ON D.DEPT_NO = B_1994.DEPT_NO
      AND B_1994.FISCAL_YEAR = 1994
    LEFT JOIN DEPT_YEAR_BUDGET B_1995
      ON D.DEPT_NO = B_1995.DEPT_NO
      AND B_1995.FISCAL_YEAR = 1995
    LEFT JOIN DEPT_YEAR_BUDGET B_1996
      ON D.DEPT_NO = B_1996.DEPT_NO
      AND B_1996.FISCAL_YEAR = 1996
```

back to top of page

The LIST function

Oleg Loa

Dmitry Yemanov

(**v.2.1**) This function returns a string result with the concatenated non-NULL values from a group. It returns NULL if there are no non-NULL values.

Format

```
<list function> ::=
  LIST '(' [ {ALL | DISTINCT} ] <value expression> [',' <delimiter value>
   ] ')'

<delimiter value> ::=
    { <string literal> | <parameter> | <variable> }
```

Syntax rules

- If neither ALL nor DISTINCT is specified, ALL is implied.
- If <delimiter value> is omitted, a comma is used to separate the concatenated values.

Other notes

- Numeric and date/time values are implicitly converted to strings during evaluation.
- The result value is of type BLOB with SUB_TYPE TEXT for all cases except list of BLOB with different subtype.
- Ordering of values within a group is implementation-defined.

Examples

```
/* A */
 SELECT LIST(ID, ':')
 FROM MY_TABLE

/* B */
 SELECT TAG_TYPE, LIST(TAG_VALUE)
 FROM TAGS
 GROUP BY TAG_TYPE
```

back to top of page

The RETURNING clause

Dmitry Yemanov

Adriano dos Santos Fernandes

(**v.2.1**) The purpose of this SQL enhancement is to enable the column values stored into a table as a result of the INSERT, UPDATE OR INSERT, UPDATE and DELETE statements to be returned to the client.

The most likely usage is for retrieving the value generated for a primary key inside a BEFORE-trigger. The RETURNING clause is optional and is available in both DSQL and PSQL, although the rules differ slightly.

In DSQL, the execution of the operation itself and the return of the set occur in a single protocol round trip.

Because the RETURNING clause is designed to return a singleton set in response to completing an operation on a single record, it is not valid to specify the clause in a statement that inserts, updates or deletes multiple records.

Note: In DSQL, the statement always returns the set, even if the operation has no effect on any record. Hence, at this stage of implementation, the potential exists to return an “empty” set. (This may be changed in the future.)

In PSQL, if no row was affected by the statement, nothing is returned and values of the receiving variables are unchanged.

Support for this feature in Embedded SQL (ESQL) was added in v.2.1.6.

Syntax patterns

```
INSERT INTO ... VALUES (...)
    [RETURNING <column_list> [INTO <variable_list>]]

INSERT INTO ... SELECT ...
    [RETURNING <column_list> [INTO <variable_list>]]

UPDATE OR INSERT INTO ... VALUES (...) ...
   [RETURNING <column_list> [INTO <variable_list>]]

UPDATE ... [RETURNING <column_list> [INTO <variable_list>]]

DELETE FROM ...
    [RETURNING <column_list> [INTO <variable_list>]]
```

Rules for using a RETURNING clause

1. The INTO part (i.e. the variable list) is allowed in PSQL only, for assigning the output set to local variables. It is rejected in DSQL.
2. The presence of the RETURNING clause causes an INSERT statement to be described by the API as isc_info_sql_stmt_exec_procedure rather than isc_info_sql_stmt_insert. Existing connectivity drivers should already be capable of supporting this feature without special alterations.

3. The RETURNING clause ignores any explicit record change (update or delete) that occurs as a result of the execution of an AFTER trigger.
4. OLD and NEW context variables can be used in the RETURNING clause of UPDATE and INSERT OR UPDATE statements.
5. In UPDATE and INSERT OR UPDATE statements, field references that are unqualified or qualified by table name or relation alias are resolved to the value of the corresponding NEW context variable.

Examples

1.

```
INSERT INTO T1 (F1, F2)
    VALUES (:F1, :F2)
      RETURNING F1, F2 INTO :V1, :V2;
```

2.

```
INSERT INTO T2 (F1, F2)
    VALUES (1, 2)
      RETURNING ID INTO :PK;
```

3.

```
DELETE FROM T1
    WHERE F1 = 1
      RETURNING F2;
```

4.

```
UPDATE T1
    SET F2 = F2 * 10
       RETURNING OLD.F2, NEW.F2;
```

back to top of page

UPDATE OR INSERT statement

Adriano dos Santos Fernandes

(**v.2.1**) This syntax has been introduced to enable a record to be either updated or inserted, according to whether or not it already exists (checked with IS NOT DISTINCT). The statement is available in both DSQL and PSQL.

Support for this feature in Embedded SQL (ESQL) was added in v.2.1.6.

Syntax pattern

```
UPDATE OR INSERT INTO <table or view> [(<column_list>)]
    VALUES (<value_list>)
    [MATCHING <column_list>]
    [RETURNING <column_list> [INTO <variable_list>]]
```

Examples

1.

```
UPDATE OR INSERT INTO T1 (F1, F2)
    VALUES (:F1, :F2);
```

2.

```
UPDATE OR INSERT INTO EMPLOYEE (ID, NAME)
    VALUES (:ID, :NAME)
    RETURNING ID;
```

3.

```
UPDATE OR INSERT INTO T1 (F1, F2)
    VALUES (:F1, :F2)
    MATCHING (F1);
```

4.

```
UPDATE OR INSERT INTO EMPLOYEE (ID, NAME)
    VALUES (:ID, :NAME)
    RETURNING OLD.NAME;
```

Usage notes

1. When MATCHING is omitted, the existence of a primary key is required.
2. INSERT and UPDATE permissions are needed on <table or view>.
3. If the RETURNING clause is present, then the statement is described as isc_info_sql_stmt_exec_procedure by the API; otherwise, it is described as isc_info_sql_stmt_insert.

Note: A *multiple rows* in singleton select error will be raised if the RETURNING clause is present and more than one record matches the search condition.

back to top of page

MERGE statement

Adriano dos Santos Fernandes

(**v.2.1**) This syntax has been introduced to enable a record to be either updated or inserted, according to whether or not a stated condition is met. The statement is available in both DSQL and PSQL.

Syntax pattern

```
<merge statement> ::=
MERGE
  INTO <table or view> [ [AS] <correlation name> ]
  USING <table or view or derived table> [ [AS] <correlation name> ]
    ON <condition>
    [ <merge when matched> ]
    [ <merge when not matched> ]

<merge when matched> ::=
  WHEN MATCHED THEN
    UPDATE SET <assignment list>

<merge when not matched> ::=
  WHEN NOT MATCHED THEN
    INSERT [ <left paren> <column list> <right paren> ]
    VALUES <left paren> <value list> <right paren>
```

Rules for MERGE

1. At least one of <merge when matched> and <merge when not matched> should be specified.
2. Neither should be specified more than once.

Note: A right join is made between the INTO and USING tables using the condition. UPDATE is called when a matching record exists in the left (INTO) table, otherwise INSERT is called.

If no record is returned from the join, INSERT is not called.

Example

```
MERGE INTO customers c
  USING (SELECT * FROM customers_delta WHERE id > 10) cd
  ON (c.id = cd.id)
    WHEN MATCHED THEN
      UPDATE SET
       name = cd.name
    WHEN NOT MATCHED THEN
      INSERT (id, name)
      VALUES (cd.id, cd.name)
```

See also:

- MERGE

back to top of page

New JOIN types

Adriano dos Santos Fernandes

(**v.2.1**) Two new JOIN types are introduced: the NAMED COLUMNS join and its close relative, the NATURAL join.

Syntax and rules

```
<named columns join> ::=
  <table reference> <join type> JOIN <table reference>
    USING ( <column list> )

<natural join> ::=
  <table reference> NATURAL <join type> JOIN <table primary>
```

Named columns join

1. All columns specified in <column list> should exist in the tables at both sides.
2. An equi-join (<left table>.<column> = <right table>.<column>) is automatically created for all columns (ANDed).
3. The USING columns can be accessed without qualifiers - in this case, the result is equivalent to COALESCE(<left table>.<column>, <right table>.<column>).
4. In "SELECT *", USING columns are expanded once, using the above rule.

Natural join

1. A "named columns join" is automatically created with all columns common to the left and right tables.
2. If there is no common column, a CROSS JOIN is created.

Examples

```
/* 1 */
select * from employee
  join department
  using (dept_no);

/* 2 */
select * from employee_project
  natural join employee
  natural join project;
```

CROSS JOIN

D. Yemanov

(**v.2.0.x**) CROSS JOIN is now supported. Logically, this syntax pattern:

```
A CROSS JOIN B
```

is equivalent to either of the following:

```
A INNER JOIN B ON 1 = 1
```

or, simply:

```
FROM A, B
```

Performance improvement at v.2.1.2

D. Yemanov

In the rare case where a cross join of three or more tables involved table[s] that contained no records, extremely slow performance was reported (CORE-2200). A performance improvement was gained by teaching the optimizer not to waste time and effort on walking through populated tables in an attempt to find matches in empty tables.

See also:

- JOIN
- Firebird 2.0 Language Reference Update: JOIN

back to top of page

INSERT with defaults

D. Yemanov

Feature request

(**v.2.1**) It is now possible to INSERT without supplying values, if Before Insert triggers and/or declared defaults are available for every column and none is dependent on the presence of any supplied 'NEW' value.

Example

```
INSERT INTO <table>
  DEFAULT VALUES
  [RETURNING <values>]
```

back to top of page

BLOB subtype 1 compatibility with VARCHAR

A. dos Santos Fernandes

(**v.2.1**) At various levels of evaluation, the engine now treats text BLOBs that are within the 32,765-byte string size limit as though they were VARCHARs. Operations that now allow text BLOBs to behave like strings are:

- Assignments, conversions and concatenations (|| operator).
- Operators =, <>, >, <, >=, ⇐, BETWEEN, IS [NOT] DISTINCT FROM.
- Functions CAST, BIT_LENGTH, CHAR[ACTER]_LENGTH, OCTET_LENGTH, LEFT, RIGHT, HASH, LOWER, UPPER, LPAD, RPAD, TRIM, OVERLAY, REPLACE, POSITION, REVERSE, MINVALUE, MAXVALUE, SUBSTRING.

Note Carefully!: SUBSTRING(), when applied to a text BLOB, now returns a text BLOB as its result, instead of the VARCHAR result that was implemented previously. This change has the potential to break expressions in existing client and PSQL code.

If the FOR argument is absent, the BLOB returned will be no greater than 32,767 bytes, even if the end of the string was not reached.

- Existential predicators IN, ANY/SOME, ALL.
- Search predicators CONTAINING, STARTING [WITH], LIKE.

Important: The predicating expression must not resolve to more than 32,767 bytes!

- A LIST expression. Note that, prior to v.2.1.4, the last part of the result may be truncated, an effect that applies to native VARCHAR columns also.

Full equality comparisons between BLOBs

(**v.2.0.x**) Comparison can be performed on the entire content of a text BLOB.

back to top of page

RDB$DB_KEY returns NULL in outer joins

A. dos Santos Fernandes

Feature request CORE-979

(**v.2.1**) By some anomaly, the physical RDB$DB_KEY has always returned a value on every output row

when specified in an outer join, thereby making a test predicated on the assumption that a non-match returns NULL in all fields return False when it ought to return True. Now, RDB$DB_KEY returns NULL when it should do so.

Sorting on BLOB and ARRAY columns is restored

Dmitry Yemanov

(**v.2.1**) In earlier pre-release versions of Firebird 2.1, changes were introduced to reject sorts (ORDER BY, GROUP BY and SELECT DISTINCT operations) at prepare time if the sort clause implicitly or explicitly involved sorting on a BLOB or ARRAY column.

That change was reversed in the RC2 pre-release version, not because it was wrong but because so many users complained that it broke the behaviour of legacy applications.

Important: This reversion to “bad old behaviour” does not in any way imply that such queries will magically return correct results. A BLOB cannot be converted to a sortable type and so, as previously, DISTINCT sortings and ORDER BY arguments that involve BLOBs, will use the BLOB_ID. GROUP BY arguments that are BLOB types will prepare successfully, but the aggregation will be non-existent.

back to top of page

Built-in functions

(**v.2.1**) Some existing built-in functions have been enhanced, while a large number of new ones has been added.

New built-in functions

Adriano dos Santos Fernandes, Oleg Loa, Alexey Karyakin

A number of built-in functions has been implemented in v.2.1 to replace common UDFs with the same names. The built-in functions will not be used if the UDF of the same name is declared in the database.

Note: The choice between UDF and built-in function is decided when compiling the statement. If the statement is compiled in a PSQL module whilst the UDF is available in the database, then the module will continue to require the UDF declaration to be present until it is next recompiled.

The new built-in function DECODE() does not have an equivalent UDF in the libraries that are distributed with Firebird.

The functions are detailed in Appendix A.

Note: Several of these built-in functions were already available in Firebird 2/ODS 11, viz., LOWER(), TRIM(), BIT_LENGTH(), CHAR_LENGTH() and OCTET_LENGTH().

back to top of page

Enhancements to functions

A. dos Santos Fernandes

EXTRACT(WEEK FROM DATE)

Feature request CORE-663

The EXTRACT() function is extended to support the ISO-8601 ordinal week numbers. For example:

```
EXTRACT (WEEK FROM date '30.09.2007')
```

returns 39.

```
alter table xyz
add WeekOfTheYear
computed by (
  case
     when (extract(month from CertainDate) = 12)
    and (extract(week from CertainDate) = 1)
    then
      'Week '||extract (WEEK from CertainDate)||' of year '
      || (1 + (extract( year from CertainDate)))
    else 'Week '||extract (WEEK from CertainDate)||' of year '
      ||extract( year from CertainDate)
    end )
```

Specify the scale for TRUNC()

Feature request CORE-1340

In Beta 1 the implementation of the TRUNC() function supported only one argument, the value to be truncated. From Beta 2, an optional second argument can be supplied to specify the scale of the truncation. For example:

```
select
  trunc(987.65, 1),
  trunc(987.65, -1)
  from rdb$database;
```

returns 987.60, 980.00.

For other examples of using TRUNC() with and without the optional scale argument, refer to the alphabetical listing of functions in Appendix A.

Milliseconds handling for EXTRACT(), DATEADD() and DATEDIFF()

Feature request CORE-1387

From v.2.1 Beta 2, EXTRACT(), DATEADD() and DATEDIFF() can operate with milliseconds (represented as an integer number). For example:

```
EXTRACT ( MILLISECOND FROM timestamp '01.01.2000 01:00:00.1234' )
```

returns 123.

```
DATEADD ( MILLISECOND, 100, timestamp '01.01.2000 01:00:00.0000' )
DATEDIFF ( MILLISECOND, timestamp '01.01.2000 02:00:00.0000', timestamp
'01.01.2000 01:00:00.0000'
```

For more explanatory examples of using DATEADD() and DATEDIFF(), refer to the alphabetical listing of functions in Appendix A.

back to top of page

Functions enhanced in v.2.0.x

Some function enhancements were already available in the V.2.0.x releases:

IIF() expression

O. Loa

(**v.2.0.x**) An IIF() expression can be used as a shortcut for a CASE expression that tests exactly two conditions. It returns the value of the first sub-expression if the given search condition evaluates to TRUE, otherwise it returns a value of the second sub-expression.

```
IIF (<search_condition>, <value1>, <value2>)
```

is implemented as a shortcut for

```
CASE
  WHEN <search_condition> THEN <value1>
  ELSE <value2>
END
```

Example

```
SELECT IIF(VAL > 0, VAL, -VAL) FROM OPERATION
```

Improvement in CAST() behaviour

D. Yemanov

(**v.2.0.x**) The infamous *Datatype unknown* error (SF Bug #1371274) when attempting some castings has been eliminated. It is now possible to use CAST to advise the engine about the data type of a parameter.

Example

```
SELECT CAST(? AS INT) FROM RDB$DATABASE
```

CAST(x as <domain-name>)

A. dos Santos Fernandes

(v.2.1.x) Casting of compatible values or expressions can now be made to a domain, after the manner of variable declarations in procedural SQL.

Syntax pattern

```
CAST (<value> | <expression> AS <builtin-data-type> | <domain-name> | TYPE
OF <domain-name>)
```

Examples

```
CREATE DOMAIN DOM AS INTEGER;
...
SELECT CAST (10.44 AS TYPE OF DOM) AN_INTEGER
  FROM RDB$DATABASE;

AN_INTEGER
----------
       10

...
SELECT CAST (3.142/2 AS DOM) AN_INTEGER
  FROM RDB$DATABASE;

AN_INTEGER
----------
         2
```

Note: Directly casting to <domain-name> applies any default or constraint defined for the domain. TYPE OF <domain-name> gets only the datatype of the domain and ignores any other attributes.

Expression arguments for SUBSTRING()

O. Loa

D. Yemanov

(**v.2.0.x**) The built-in function SUBSTRING() can now take arbitrary expressions in its parameters.

Formerly, the inbuilt SUBSTRING() function accepted only constants as its second and third arguments (start position and length, respectively). Now, the arguments can be anything that resolves to a value, including host parameters, function results, expressions, subqueries, etc.

Tip: If your attempts to use this feature fail with “invalid token” errors, bear in mind that expression arguments often need to be enclosed in brackets!

Changes to results returned from SUBSTRING()

(**v.2.1.x**) To conform with standards, the character length of the result of applying SUBSTRING() to a VARCHAR or CHAR is now a VARCHAR of the same character length declared for or deduced from the value in the first argument.

In Firebird 2.0 and 1.5, the returned value was a CHAR with the same character length as the declared or implied value of the first argument, too. That implementation could become a bug in Firebird 2.0 under conditions where the input string was a CHAR and the FOR argument was presented as an expression whose result was not known at the point where memory was prepared to receive the result string. The v.2.1 change addresses that.

It is not necessary to redefine any PSQL variables you have declared to receive the results from SUBSTRING(). It is still correct to declare its size just big enough to accommodate the actual data returned. Just be sure that any FOR argument that is an expression cannot resolve to an integer that is larger than the number of characters declared for your variable.

GOTCHA for BLOBs

Clearly, a text BLOB, being of indeterminate character length, cannot fit into a paradigm that populates a string of known maximum dimension. Therefore, the result returned from applying SUBSTRING() to a text BLOB is not a VARCHAR() as previously, but a text BLOB.

This change can break existing PSQL and expression code.

- Watch out for overflows! Take particular care with CASTs and concatenations.
- In v.2.1.x sub-releases prior to v.2.1.4, pay attention to memory usage when assigning to temporary BLOBs in loops! The engine formerly allocated a minimum of one database page of memory for each temporary BLOB, regardless of its actual size. The implementation was improved in v.2.1.4 (see tracker ticket CORE-1658).

back to top of page

DSQL parsing of table names is stricter

A. Brinkman

Alias handling and ambiguous field detecting have been improved. In summary:

1. When a table alias is provided for a table, either that alias, or no alias, must be used. It is no longer valid to supply only the table name.
2. Ambiguity checking now checks first for ambiguity at the current level of scope, making it valid in some conditions for columns to be used without qualifiers at a higher scope level.

Examples

1. When an alias is present it must be used; or no alias at all is allowed.

a) This query was allowed in Firebird 1.5 and earlier versions:

```
SELECT
  RDB$RELATIONS.RDB$RELATION_NAME
FROM
  RDB$RELATIONS R
```

but will now correctly report an error that the field RDB$RELATIONS.RDB$RELATION_NAME could not be found.

Use this (preferred):

```
SELECT
  R.RDB$RELATION_NAME
FROM
  RDB$RELATIONS R
```

or this statement:

```
SELECT
  RDB$RELATION_NAME
FROM
  RDB$RELATIONS R
```

b) The statement below will now correctly use the FieldID from the subquery and from the updating table:

```
UPDATE
  TableA
SET
  FieldA = (SELECT SUM(A.FieldB) FROM TableA A
    WHERE A.FieldID = TableA.FieldID)
```

Note: In Firebird it is possible to provide an alias in an update statement. Although many other

database vendors do not support it, this capability should help those developers who have requested it to make Firebird's SQL more interchangeable with SQL database products that do support it.

c) This example did not run correctly in Firebird 1.5 and earlier:

```
SELECT
  RDB$RELATIONS.RDB$RELATION_NAME,
  R2.RDB$RELATION_NAME
FROM
  RDB$RELATIONS
  JOIN RDB$RELATIONS R2 ON
   (R2.RDB$RELATION_NAME = RDB$RELATIONS.RDB$RELATION_NAME)
```

If RDB$RELATIONS contained 90 records, it would return 90 * 90 = 8100 records, but in Firebird 2 it will correctly return 90 records.

2. a) This would except with a syntax error in Firebird 1.5, but is possible in Firebird 2:

```
SELECT
  (SELECT RDB$RELATION_NAME FROM RDB$DATABASE)
FROM
  RDB$RELATIONS
```

b) Ambiguity checking in subqueries: the query below would run in Firebird 1.5 without reporting an ambiguity, but will report it in Firebird 2:

```
SELECT
  (SELECT
    FIRST 1 RDB$RELATION_NAME
  FROM
    RDB$RELATIONS R1
    JOIN RDB$RELATIONS R2 ON
      (R2.RDB$RELATION_NAME = R1.RDB$RELATION_NAME))
FROM
  RDB$DATABASE
```

back to top of page

EXECUTE BLOCK statement

V. Khorsun

The SQL language extension EXECUTE BLOCK makes “dynamic PSQL” available to SELECT specifications. It has the effect of allowing a self-contained block of PSQL code to be executed in dynamic SQL as if it were a stored procedure.

Syntax pattern

```
EXECUTE BLOCK [ (param datatype = ?, param datatype = ?, ...) ]
  [ RETURNS (param datatype, param datatype, ...) ]
AS
[DECLARE VARIABLE var datatype; ...]
BEGIN
  ...
END
```

For the client, the call isc_dsql_sql_info with the parameter isc_info_sql_stmt_type returns

- isc_info_sql_stmt_select if the block has output parameters. The semantics of a call is similar to a SELECT query: the client has a cursor open, can fetch data from it, and must close it after use.
- isc_info_sql_stmt_exec_procedure if the block has no output parameters. The semantics of a call is similar to an EXECUTE query: the client has no cursor and execution continues until it reaches the end of the block or is terminated by a SUSPEND.

The client should preprocess only the head of the SQL statement or use '?' instead of ':' as the parameter indicator because, in the body of the block, there may be references to local variables or arguments with a colon prefixed.

Example

The user SQL is

```
EXECUTE BLOCK (X INTEGER = :X)
  RETURNS (Y VARCHAR)
AS
DECLARE V INTEGER;
BEGIN
  INSERT INTO T(...) VALUES (... :X ...);
  SELECT ... FROM T INTO :Y;
  SUSPEND;
END
```

The preprocessed SQL is

```
EXECUTE BLOCK (X INTEGER = ?)
  RETURNS (Y VARCHAR)
AS
DECLARE V INTEGER;
BEGIN
  INSERT INTO T(...) VALUES (... :X ...);
  SELECT ... FROM T INTO :Y;
  SUSPEND;
END
```

back to top of page

Derived tables

A. Brinkman

Implemented support for derived tables in DSQL (subqueries in FROM clause) as defined by SQL200X. A derived table is a set, derived from a dynamic SELECT statement. Derived tables can be nested, if required, to build complex queries and they can be involved in joins as though they were normal tables or views.

Syntax pattern

```
SELECT
  <select list>
FROM
  <table reference list>

<table reference list> ::= <table reference> [{<comma> <table
reference>}...]

<table reference> ::=
    <table primary>
  | <joined table>

<table primary> ::=
    <table> [[AS] <correlation name>]
  | <derived table>

<derived table> ::=
    <query expression> [[AS] <correlation name>]
      [<left paren> <derived column list> <right paren>]

<derived column list> ::= <column name> [{<comma> <column name>}...]
```

Examples

a) Simple derived table:

```
SELECT
  *
FROM
  (SELECT
    RDB$RELATION_NAME, RDB$RELATION_ID
  FROM
    RDB$RELATIONS) AS R (RELATION_NAME, RELATION_ID)
```

b) Aggregate on a derived table which also contains an aggregate:

```
SELECT
  DT.FIELDS,
```

```
  Count(*)
FROM
  (SELECT
    R.RDB$RELATION_NAME,
    Count(*)
  FROM
    RDB$RELATIONS R
    JOIN RDB$RELATION_FIELDS RF ON (RF.RDB$RELATION_NAME =
R.RDB$RELATION_NAME)
  GROUP BY
    R.RDB$RELATION_NAME) AS DT (RELATION_NAME, FIELDS)
GROUP BY
  DT.FIELDS
```

c) UNION and ORDER BY example:

```
SELECT
  DT.*
FROM
  (SELECT
    R.RDB$RELATION_NAME,
    R.RDB$RELATION_ID
  FROM
    RDB$RELATIONS R
  UNION ALL
   SELECT
   R.RDB$OWNER_NAME,
     R.RDB$RELATION_ID
  FROM
     RDB$RELATIONS R
  ORDER BY
    2) AS DT
WHERE
  DT.RDB$RELATION_ID <= 4
```

Points to note

- Every column in the derived table must have a name. Unnamed expressions like constants should be added with an alias or the column list should be used.
- The number of columns in the column list should be the same as the number of columns from the query expression.
- The optimizer can handle a derived table very efficiently. However, if the derived table is involved in an inner join and contains a subquery, then no join order can be established and performance will suffer.

back to top of page

ROLLBACK RETAIN syntax

D. Yemanov

The ROLLBACK RETAIN statement is now supported in DSQL.

A rollback retaining feature was introduced in InterBase 6.0, but this rollback mode could be used only via an API call to isc_rollback_retaining(). By contrast, commit retaining could be used either via an API call to isc_commit_retaining() or by using a DSQL COMMIT RETAIN statement.

Firebird 2.0 adds an optional RETAIN clause to the DSQL ROLLBACK statement to make it consistent with COMMIT [RETAIN].

Syntax pattern: follows that of COMMIT RETAIN.

back to top of page

ROWS syntax

D. Yemanov

ROWS syntax is used to limit the number of rows retrieved from a select expression. For an uppermost-level select statement, it could specify the number of rows to be returned to the host program. A more understandable alternative to the FIRST/SKIP clauses, the ROWS syntax accords with the latest SQL standard and brings some extra benefits. It can be used in unions, any kind of subquery and in UPDATE or DELETE statements.

It is available in both DSQL and PSQL.

Syntax pattern

```
SELECT ...
  [ORDER BY <expr_list>]
  ROWS <expr1> [TO <expr2>]
```

Examples

1.

```
SELECT * FROM T1
  UNION ALL
SELECT * FROM T2
  ORDER BY COL
  ROWS 10 TO 100
```

2.

```
SELECT COL1, COL2,
  ( SELECT COL3 FROM T3 ORDER BY COL4 DESC ROWS 1 )
FROM T4
```

3.

```
DELETE FROM T5
  ORDER BY COL5
  ROWS 1
```

Points to note

1. When <expr2> is omitted, then ROWS <expr1> is semantically equivalent to FIRST <expr1>. When both <expr1> and <expr2> are used, then ROWS <expr1> TO <expr2> means the same as FIRST (<expr2> - <expr1> + 1) SKIP (<expr1> - 1)
2. There is nothing that is semantically equivalent to a SKIP clause used without a FIRST clause.

back to top of page

Enhancements to UNION handling

The rules for UNION queries have been improved as follows:

UNION DISTINCT keyword implementation

D. Yemanov

UNION DISTINCT is now allowed as a synonym for simple UNION, in accordance with the SQL-99 specification. It is a minor change: DISTINCT is the default mode, according to the standard. Formerly, Firebird did not support the explicit inclusion of the optional keyword DISTINCT.

Syntax pattern

```
UNION [{DISTINCT | ALL}]
```

Improved type coercion in UNIONs

A. Brinkman

Automatic resolution of the data type of the result of an aggregation over values of compatible data types, such as case expressions and columns at the same position in a union query expression, now uses smarter rules.

Syntax rules

Let DTS be the set of data types over which we must determine the final result data type.

1. All of the data types in DTS shall be comparable.
2. Case:
    1. If any of the data types in DTS is character string, then:
        1. If any of the data types in DTS is variable-length character string, then the result data type is variable-length character string with maximum length in characters equal to the largest maximum amongst the data types in DTS.
        2. Otherwise, the result data type is fixed-length character string with length in characters equal to the maximum of the lengths in characters of the data types in DTS.
        3. The character set/collation is used from the first character string data type in DTS.
3. If all of the data types in DTS are exact numeric, then the result data type is exact numeric with scale equal to the maximum of the scales of the data types in DTS and the maximum precision of all data types in DTS. Note: *Checking for precision overflows is done at run-time only. The developer should take measures to avoid the aggregation resolving to a precision overflow.*
4. If any data type in DTS is approximate numeric, then each data type in DTS shall be numeric else an error is thrown.
5. If some data type in DTS is a date/time data type, then every data type in DTS shall be a date/time data type having the same date/time type.
6. If any data type in DTS is BLOB, then each data type in DTS shall be BLOB and all with the same subtype.

UNIONs allowed in ANY/ALL/IN subqueries

D. Yemanov

The subquery element of an ANY, ALL or IN search may now be a UNION query.

back to top of page

Enhancements to NULL logic

The following features involving NULL in DSQL have been implemented:

New [NOT] DISTINCT test treats two NULL operands as equal

O. Loa, D. Yemanov

A new equivalence predicate behaves exactly like the equality/inequality predicates, but, instead of testing for equality, it tests whether one operand is distinct from the other.

Thus, IS NOT DISTINCT treats (NULL equals NULL) as if it were true, since one NULL (or expression resolving to NULL) is not distinct from another. It is available in both DSQL and PSQL.

Syntax pattern

```
<value> IS [NOT] DISTINCT FROM <value>
```

Examples

1.

```
SELECT * FROM T1
  JOIN T2
    ON T1.NAME IS NOT DISTINCT FROM T2.NAME;
```

2.

```
SELECT * FROM T
  WHERE T.MARK IS DISTINCT FROM 'test';
```

Points to note

1. Because the DISTINCT predicate considers that two NULL values are not distinct, it never evaluates to the truth value UNKNOWN. Like the IS [NOT] NULL predicate, it can only be True or False.

Read more about NULL: For more understanding of the way NULL comparisons are evaluated, please read the Firebird Null Guide, available through the Documentation Index at the Firebird website.

2. The NOT DISTINCT predicate can be optimized using an index, if one is available.

NULL comparison rule relaxed

D. Yemanov

A NULL literal can now be treated as a value in all expressions without returning a syntax error. You may now specify expressions such as

```
A = NULL
B > NULL
A + NULL
B || NULL
```

Note: All such expressions evaluate to NULL. The change does not alter nullability-aware semantics of the engine, it simply relaxes the syntax restrictions a little.

NULLs ordering changed to comply with standard

N. Samofatov

Placement of NULLs in an ordered set has been changed to accord with the SQL standard that NULL

ordering be consistent, i.e. if ASC[ENDING] order puts them at the bottom, then DESC[ENDING] puts them at the top; or vice-versa. This applies only to databases created under the new on-disk structure, since it needs to use the index changes in order to work.

Important: If you override the default NULLs placement, no index can be used for sorting. That is, no index will be used for an ASCENDING sort if NULLS LAST is specified, nor for a DESCENDING sort if NULLS FIRST is specified.

Examples

```
Database: proc.fdb
SQL> create table gnull(a int);
SQL> insert into gnull values(null);
SQL> insert into gnull values(1);
SQL> select a from gnull order by a;

          A
============
     <null>
          1

SQL> select a from gnull order by a asc;

          A
============
     <null>
          1

SQL> select a from gnull order by a desc;

          A
============
          1
     <null>

SQL> select a from gnull order by a asc nulls first;

          A
============
     <null>
          1

SQL> select a from gnull order by a asc nulls last;

          A
============
          1
     <null>

SQL> select a from gnull order by a desc nulls last;
```

```
           A
============
           1
      <null>

SQL> select a from gnull order by a desc nulls first;

           A
============
      <null>
           1
```

See also:

- ORDER BY
- Firebird 2.0.4 Release Notes: Improvements in sorting
- Firebird 2.0 SQL Language Reference Update: NULLs placement

back to top of page

Subqueries and INSERT statements can now accept UNION sets

D. Yemanov

SELECT specifications used in subqueries and in INSERT INTO <insert-specification> SELECT.. statements can now specify a UNION set.

New extensions to UPDATE and DELETE syntaxes

O. Loa

ROWS specifications and PLAN and ORDER BY clauses can now be used in UPDATE and DELETE statements.

Users can now specify explicit plans for UPDATE/DELETE statements in order to optimize them manually. It is also possible to limit the number of affected rows with a ROWS clause, optionally used in combination with an ORDER BY clause to have a sorted record set.

Syntax pattern

```
UPDATE ... SET ... WHERE ...
[PLAN <plan items>]
```

```
[ORDER BY <value list>]
[ROWS <value> [TO <value>]]
```

or

```
DELETE ... FROM ...
[PLAN <plan items>]
[ORDER BY <value list>]
[ROWS <value> [TO <value>]]
```

See also:

- UPDATE
- DELETE

back to top of page

Extended context variables

A number of new facilities have been added to extend the context information that can be retrieved:

Sub-second values enabled for Time and DateTime variables

D. Yemanov

CURRENT_TIMESTAMP, 'NOW' now return milliseconds

The context variable CURRENT_TIMESTAMP and the date/time literal 'NOW' will now return the sub-second time part in milliseconds.

Seconds precision enabled for CURRENT_TIME and CURRENT_TIMESTAMP

CURRENT_TIME and CURRENT_TIMESTAMP now optionally allow seconds precision.

The feature is available in both DSQL and PSQL.

Syntax pattern

```
CURRENT_TIME [(<seconds precision>)]
CURRENT_TIMESTAMP [(<seconds precision>)]
```

Examples

1. SELECT CURRENT_TIME FROM RDB$DATABASE;
2. SELECT CURRENT_TIME(3) FROM RDB$DATABASE;

3. SELECT CURRENT_TIMESTAMP(3) FROM RDB$DATABASE;

Note:

1. The maximum possible precision is 3 which means accuracy of 1/1000 second (one millisecond). This accuracy may be improved in future versions.
2. If no seconds precision is specified, the following values are implicit:
    1. 0 for CURRENT_TIME
    2. 3 for CURRENT_TIMESTAMP

New system functions to retrieve context variables

N. Samofatov

Values of context variables can now be obtained using the system functions RDB$GET_CONTEXT and RDB$SET_CONTEXT. These new built-in functions give access through SQL to some information about the current connection and current transaction. They also provide a mechanism to retrieve user context data and associate it with the transaction or connection.

Syntax pattern

```
RDB$SET_CONTEXT( <namespace>, <variable>, <value> )
RDB$GET_CONTEXT( <namespace>, <variable> )
```

These functions are really a form of external function that exists inside the database intead of being called from a dynamically loaded library. The following declarations are made automatically by the engine at database creation time:

Declaration

```
DECLARE EXTERNAL FUNCTION RDB$GET_CONTEXT
    VARCHAR(80),
    VARCHAR(80)
RETURNS VARCHAR(255) FREE_IT;

DECLARE EXTERNAL FUNCTION RDB$SET_CONTEXT
    VARCHAR(80),
    VARCHAR(80),
    VARCHAR(255)
RETURNS INTEGER BY VALUE;
```

Usage

RDB$SET_CONTEXT and RDB$GET_CONTEXT set and retrieve the current value of a context variable. Groups of context variables with similar properties are identified by namespace identifiers. The namespace determines the usage rules, such as whether the variables may be read and written to, and by whom.

Note: Namespace and variable names are case-sensitive.

- RDB$GET_CONTEXT retrieves current value of a variable. If the variable does not exist in namespace, the function returns NULL.
- RDB$SET_CONTEXT sets a value for specific variable, if it is writable. The function returns a value of 1 if the variable existed before the call and 0 otherwise.
- To delete a variable from a context, set its value to NULL.

Pre-defined namespaces

A fixed number of pre-defined namespaces is available:

USER_SESSION

Offers access to session-specific user-defined variables. You can define and set values for variables with any name in this context.

USER_TRANSACTION

Offers similar possibilities for individual transactions.

SYSTEM

Provides read-only access to the following variables:

NETWORK_PROTOCOL: The network protocol used by client to connect. Currently used values: “TCPv4”, “WNET”, “XNET” and NULL. **CLIENT_ADDRESS:** The wire protocol address of the remote client, represented as a string. The value is an IP address in form “xxx.xxx.xxx.xxx” for TCPv4 protocol; the local process ID for XNET protocol; and NULL for any other protocol. **DB_NAME:** Canonical name of the current database. It is either the alias name (if connection via file names is disallowed DatabaseAccess = NONE) or, otherwise, the fully expanded database file name. **ISOLATION_LEVEL:** The isolation level of the current transaction. The returned value will be one of READ COMMITTED, SNAPSHOT, CONSISTENCY. **TRANSACTION_ID:** The numeric ID of the current transaction. The returned value is the same as would be returned by the CURRENT_TRANSACTION pseudo-variable. **SESSION_ID:** The numeric ID of the current session. The returned value is the same as would be returned by the CURRENT_CONNECTION pseudo-variable. **CURRENT_USER:** The current user. The returned value is the same as would be returned by the CURRENT_USER pseudo-variable or the predefined variable USER. **CURRENT_ROLE:** Current role for the connection. Returns the same value as the CURRENT_ROLE pseudo-variable.

Notes

To avoid DoS attacks against the Firebird Server, the number of variables stored for each transaction or session context is limited to 1000.

Example of use

```
set term ^;
create procedure set_context(User_ID varchar(40), Trn_ID integer) as
begin
```

```
    RDB$SET_CONTEXT('USER_TRANSACTION', 'Trn_ID', Trn_ID);
    RDB$SET_CONTEXT('USER_TRANSACTION', 'User_ID', User_ID);
  end ^

  create table journal (
     jrn_id integer not null primary key,
     jrn_lastuser varchar(40),
     jrn_lastaddr varchar(255),
     jrn_lasttransaction integer
  )^

CREATE TRIGGER UI_JOURNAL FOR JOURNAL BEFORE INSERT OR UPDATE
  as
  begin
    new.jrn_lastuser = rdb$get_context('USER_TRANSACTION', 'User_ID');
    new.jrn_lastaddr = rdb$get_context('SYSTEM', 'CLIENT_ADDRESS');
    new.jrn_lasttransaction = rdb$get_context('USER_TRANSACTION', 'Trn_ID');
  end ^
  commit ^
  execute procedure set_context('skidder', 1) ^

  insert into journal(jrn_id) values(0) ^
  set term ;^
```

Since rdb$set_context returns 1 or zero, it can be made to work with a simple SELECT statement.

Example

```
SQL> select rdb$set_context('USER_SESSION', 'Nickolay', 'ru')
CNT> from rdb$database;

RDB$SET_CONTEXT
===============
            0
```

0 means not defined already; we have set it to ru

```
SQL> select rdb$set_context('USER_SESSION', 'Nickolay', 'ca')
CNT> from rdb$database;

RDB$SET_CONTEXT
===============
             1
```

1 means it was defined already; we have changed it to ca

```
SQL> select rdb$set_context('USER_SESSION', 'Nickolay', NULL)
CNT> from rdb$database;

RDB$SET_CONTEXT
```

```
===============
              1
```

1 says it existed before; we have changed it to NULL, i.e. undefined it.

```
SQL> select rdb$set_context('USER_SESSION', 'Nickolay', NULL)
CNT> from rdb$database;

RDB$SET_CONTEXT
===============
              0
```

0, since nothing actually happened this time: it was already undefined.

back to top of page

Improvements in handling user-specified query plans

D. Yemanov

1. Plan fragments are propagated to nested levels of joins, enabling manual optimization of complex .
2. A user-supplied plan will be checked for correctness in outer joins.
3. Short-circuit optimization for user-supplied plans has been added.
4. A user-specified access path can be supplied for any SELECT-based statement or clause.

Syntax rules

The following schema describing the syntax rules should be helpful when composing plans:

```
PLAN ( { <stream_retrieval> | <sorted_streams> | <joined_streams> } )

<stream_retrieval> ::= { <natural_scan> | <indexed_retrieval> |
    <navigational_scan> }

<natural_scan> ::= <stream_alias> NATURAL

<indexed_retrieval> ::= <stream_alias> INDEX ( <index_name>
    [, <index_name> ...] )

<navigational_scan> ::= <stream_alias> ORDER <index_name>
    [ INDEX ( <index_name> [, <index_name> ...] ) ]

<sorted_streams> ::= SORT ( <stream_retrieval> )

<joined_streams> ::= JOIN ( <stream_retrieval>, <stream_retrieval>
    [, <stream_retrieval> ...] )
```

```
    | [SORT] MERGE ( <sorted_streams>, <sorted_streams> )
```

Details

Natural scan means that all rows are fetched in their natural storage order. Thus, all pages must be read before search criteria are validated.

Indexed retrieval uses an index range scan to find row ids that match the given search criteria. The found matches are combined in a sparse bitmap which is sorted by page numbers, so every data page will be read only once. After that the table pages are read and required rows are fetched from them.

Navigational scan uses an index to return rows in the given order, if such an operation is appropriate:

- The index b-tree is walked from the leftmost node to the rightmost one.
- If any search criterion is used on a column specified in an ORDER BY clause, the navigation is limited to some subtree path, depending on a predicate.
- If any search criterion is used on other columns which are indexed, then a range index scan is performed in advance and every fetched key has its row id validated against the resulting bitmap. Then a data page is read and the required row is fetched.

Note: Note that a navigational scan incurs random page I/O, as reads are not optimized.

A *sort operation* performs an external sort of the given stream retrieval.

A *join* can be performed either via the nested loops algorithm (JOIN plan) or via the sort merge algorithm (MERGE plan):

- An *inner nested loop join* may contain as many streams as are required to be joined. All of them are equivalent.
- An *outer nested loop join* always operates with two streams, so you'll see nested JOIN clauses in the case of 3 or more outer streams joined.

A *sort merge* operates with two input streams which are sorted beforehand, then merged in a single run.

Examples

```
SELECT RDB$RELATION_NAME
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME LIKE 'RDB$%'
PLAN (RDB$RELATIONS NATURAL)
ORDER BY RDB$RELATION_NAME

SELECT R.RDB$RELATION_NAME, RF.RDB$FIELD_NAME
FROM RDB$RELATIONS R
  JOIN RDB$RELATION_FIELDS RF
  ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
PLAN MERGE (SORT (R NATURAL), SORT (RF NATURAL))
```

Notes

- A PLAN clause may be used in all select expressions, including subqueries, derived tables and view definitions. It can be also used in UPDATE and DELETE statements, because they're implicitly based on select expressions.
- If a PLAN clause contains some invalid retrieval description, then either an error will be returned or this bad clause will be silently ignored, depending on severity of the issue.
- ORDER <navigational_index> INDEX (<filter_indices>) kind of plan is reported by the engine and can be used in the user-supplied plans starting with FB 2.0.

back to top of page

Improvements in sorting

A. Brinkman

Some useful improvements have been made to SQL sorting operations:

ORDER BY or GROUP BY <alias-name>

Column aliases are now allowed in both these clauses.

Examples

1. ORDER BY

```
SELECT RDB$RELATION_ID AS ID
FROM RDB$RELATIONS
ORDER BY ID
```

2. GROUP BY

```
SELECT RDB$RELATION_NAME AS ID, COUNT(*)
FROM RDB$RELATION_FIELDS
GROUP BY ID
```

GROUP BY arbitrary expressions

A GROUP BY condition can now be any valid expression.

Example

```
...
   GROUP BY
   SUBSTRING(CAST((A * B) / 2 AS VARCHAR(15)) FROM 1 FOR 2)
```

Order SELECT * sets by degree number

ORDER BY degree (ordinal column position) now works on a select * list.

Example

```
SELECT *
  FROM RDB$RELATIONS
  ORDER BY 9
```

Parameters and ordinal sorts - a "Gotcha"

According to grammar rules, since v.1.5, ORDER BY <value_expression> is allowed and <value_expression> could be a variable or a parameter. It is tempting to assume that ORDER BY <degree_number> could thus be validly represented as a replaceable input parameter, or an expression containing a parameter.

However, while the DSQL parser does not reject the parameterised ORDER BY clause expression if it resolves to an integer, the optimizer requires an absolute, constant value in order to identify the *position in the output list* of the ordering column or derived field. If a parameter is accepted by the parser, the output will undergo a “dummy sort” and the returned set will be unsorted.

back to top of page

NEXT VALUE FOR expression

D. Yemanov

Added SQL-99 compliant NEXT VALUE FOR <sequence_name> expression as a synonym for GEN_ID(<generator-name>, 1), complementing the introduction of CREATE SEQUENCE syntax as the SQL standard equivalent of CREATE GENERATOR.

Examples

1.

```
SELECT GEN_ID(S_EMPLOYEE, 1) FROM RDB$DATABASE;
```

2.

```
INSERT INTO EMPLOYEE (ID, NAME)
VALUES (NEXT VALUE FOR S_EMPLOYEE, 'John Smith');
```

Note:

- Currently, increment (“step”) values not equal to 1 (one) can be used only by calling the GEN_ID function. Future versions are expected to provide full support for SQL-99 sequence generators, which allows the required increment values to be specified at the DDL level. Unless there is a

vital need to use a step value that is not 1, use of a NEXT VALUE FOR value expression instead of the GEN_ID function is recommended.
- GEN_ID(<name>, 0) allows you to retrieve the current sequence value, but it should never be used in insert/update statements, as it produces a high risk of uniqueness violations in a concurrent environment.

back to top of page

Articles

SELECT statement & expression syntax

Dmitry Yemanov

About the semantics

- A select statement is used to return data to the caller (PSQL module or the client program).
- Select expressions retrieve parts of data that construct columns that can be in either the final result set or in any of the intermediate sets. Select expressions are also known as subqueries.

Syntax rules

```
<select statement> ::=
  <select expression> [FOR UPDATE] [WITH LOCK]

<select expression> ::=
  <query specification> [UNION [{ALL | DISTINCT}] <query specification>]

<query specification> ::=
  SELECT [FIRST <value>] [SKIP <value>] <select list>
  FROM <table expression list>
  WHERE <search condition>
  GROUP BY <group value list>
  HAVING <group condition>
  PLAN <plan item list>
  ORDER BY <sort value list>
  ROWS <value> [TO <value>]

<table expression> ::=
  <table name> | <joined table> | <derived table>

<joined table> ::=
  {<cross join> | <qualified join>}

<cross join> ::=
  <table expression> CROSS JOIN <table expression>

<qualified join> ::=
```

```
  <table expression> [{INNER | {LEFT | RIGHT | FULL} [OUTER]}] JOIN <table
expression>
  ON <join condition>

<derived table> ::=
  '(' <select expression> ')'
```

Conclusions

- FOR UPDATE mode and row locking can only be performed for a final dataset, they cannot be applied to a subquery.
- Unions are allowed inside any subquery.
- Clauses FIRST, SKIP, PLAN, ORDER BY, ROWS are allowed for any subquery.

Notes

- Either FIRST/SKIP or ROWS is allowed, but a syntax error is thrown if you try to mix the syntaxes.
- An INSERT statement accepts a select expression to define a set to be inserted into a table. Its SELECT part supports all the features defined for select statements/expressions.
- UPDATE and DELETE statements are always based on an implicit cursor iterating through its target table and limited with the WHERE clause. You may also specify the final parts of the select expression syntax to limit the number of affected rows or optimize the statement.

Clauses allowed at the end of UPDATE/DELETE statements are PLAN, ORDER BY and ROWS.

back to top of page

Data type of an aggregation result

Arno Brinkman

When aggregations, CASE evaluations and UNIONs for output columns are performed over a mix of comparable data types, the engine has to choose one data type for the result. The developer often has to prepare a variable or buffer for such results and is mystified when a request returns a data type exception. The rules followed by the engine in determining the data type for an output column under these conditions are explained here.

1. Let DTS be the set of data types over which we must determine the final result data type.
2. All of the data types in DTS shall be comparable.
3. In the case that
    1. Any of the data types in DTS is a character string
        1. If all data types in DTS are fixed-length character strings, then the result is also a fixed-length character string; otherwise the result is a variable-length character string. The resulting string length, in characters, is equal to the maximum of the lengths, in characters, of the data types in DTS.
        2. The character set and collation used are taken from the data type of the first character string in DTS.
    2. All of the data types in DTS are exact numeric

The result data type is exact numeric with scale equal to the maximum of the scales of the data types in DTS and precision equal to the maximum precision of all data types in DTS.

c. Any data type in DTS is approximate numeric

Each data type in DTS must be numeric, otherwise an error is thrown.

d. Any data type in DTS is a date/time data type

Every data type in DTS must be a date/time type having the same date/time type, otherwise an error is thrown.

e. Any data type in DTS is a BLOB

Each data type in DTS must be BLOB and all with the same subtype.

back to top of page

A useful trick with date literals

H. Borrie

In days gone by, before the advent of context variables like CURRENT_DATE, CURRENT_TIMESTAMP, et al., we had *predefined date literals*, such as 'NOW', 'TODAY', 'YESTERDAY' and so on. These predefined date literals survive in Firebird's SQL language set and are still useful.

In InterBase 5.x and lower, the following statement was “legal” and returned a DATE value (remembering that the DATE type then was what is now TIMESTAMP):

```
select 'NOW' from rdb$database /* returns system date and time */
```

In a database of ODS 10 or higher, that statement returns the string 'NOW'. We have had to learn to cast the date literal to get the result we want:

```
select cast('NOW' as TIMESTAMP) from rdb$database
```

For a long time - probably since IB 6 - there has been an undocumented “short expression syntax” for casting not just the predefined date/time literals but any date literals. Actually, it is defined in the standard. Most of us were just not aware that it was available. It takes the form <data type> <date literal>. Taking the CAST example above, the short syntax would be as follows:

```
select TIMESTAMP 'NOW' from rdb$database
```

This short syntax can participate in other expressions. The following example illustrates a date/time arithmetic operation on a predefined literal:

```
update mytable
  set OVERDUE = 'T'
  where DATE 'YESTERDAY' - DATE_DUE > 10
```