Data Manipulation Language (DML)

New and extended DSQL syntaxes

In this section are details of DML language statements or constructs that have been added to the DSQL language set in Firebird 2.0.

EXECUTE BLOCK statement

V. Horsun

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
```

See also:

EXECUTE IBEBLOCK

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 made.

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.

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 would 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

- 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).
- 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 type coercion logic between subsets of a union is now more intelligent. 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 characterset/collation is used from the first character string data type in DTS.
    2. 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.

c. If any data type in DTS is approximate numeric, then each data type in DTS shall be numeric else an error is thrown.

d. 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.

e. If any data type in DTS is BLOB, then each data type in DTS shall be BLOB and all with the same sub-type.

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

IIF expression syntax added

O. Loa

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

is implemented as a shortcut for

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

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.

Example

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

See also:

- IIF
- ibec_IIF

back to top of page

CAST() behaviour improved

D. Yemanov

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
```

See also:

CAST

back to top of page

Built-in function SUBSTRING() enhanced

O. Loa, D. Yemanov

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.

Note: The length of the resulting column is the same as the length of the first argument. This means that, in the following

```
x = varchar(50);
substring(x from 1 for 1);
```

the new column has a length of 50, not 1. (Thank the SQL standards committee!)

See also:

SUBSTRING() * SUBSTRING()

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';
```

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.
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:

NULL

back to top of page

CROSS JOIN is now supported

D. Yemanov

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.2.0.6

(v.2.0.6) 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. (Backported from v.2.1.2.)

See also:

JOIN

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 recordset.

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>]]
```

back to top of page

New 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:
   - 0 for CURRENT_TIME
   - 3 for CURRENT_TIMESTAMP

back to top of page

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$GET_CONTEXT retrieves the current value of a variable. If the variable does not exist in namespace, the function returns NULL.
- RDB$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 the 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 0, 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. undefine 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 outer joins.
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 loops 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:

1. 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.
2. 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.
3. 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)
```

back to top of page

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 syntax

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:

1. 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.
2. 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

RETURNING clause for insert statements

D. Yemanov

The RETURNING clause syntax has been implemented for the INSERT statement, enabling the return of a result set from the INSERT statement. The set contains the column values actually stored. Most common usage would be for retrieving the value of the primary key generated inside a BEFORE-trigger.

Available in DSQL and PSQL.

Syntax pattern

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

Example(s)

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;
```

Note:

1. The INTO part (i.e. the variable list) is allowed in PSQL only (to assign local variables) and rejected in DSQL.
2. In DSQL, values are being returned within the same protocol roundtrip as the INSERT itself is executed.
3. If the RETURNING clause is present, then the statement is described as isc_info_sql_stmt_exec_procedure by the API (instead of isc_info_sql_stmt_insert), so the existing connectivity drivers should support this feature automatically.
4. Any explicit record change (update or delete) performed by AFTER-triggers is ignored by the RETURNING clause.
5. Cursor based inserts (INSERT INTO ...? SELECT ... RETURNING ...) are not supported.
6. This clause can return table column values or arbitrary expressions.

back to top of page

DSQL parsing of table aliases 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 FB1.5 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, but many other database vendors do not support it. These SQL statements will improve the interchangeability of Firebird's SQL with other SQL database products.

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 failed 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

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.