New SQL Features in Firebird

Whats new in Firebird 3

- **Common SQL**
  - Full syntax of **MERGE** statement (per SQL 2008)
  - **MERGE ... RETURNING**
  - Window (analytical) functions
  - **SUBSTRING** with regular expressions
  - **BOOLEAN** data type
  - New **RDB$RECORD_VERSION** pseudo-column
  - Cursor stability with data modification queries
  - Global temporary tables are improved

Whats new in Firebird 3

- **Procedural SQL**
  - **SQL functions**
  - **Sub-routines**
  - **External functions, procedures and triggers on C\C++\Pascal\Java etc.**
  - **Packages**
  - **Exceptions with parameters : EXCEPTION ... USING (...)**
  - **SQLSTATE in WHEN handler**
  - **CONTINUE in loops**

Whats new in Firebird 3

- **DDL**
  - **Manage nullability of column**
    - **ALTER DOMAIN ... {NULL | NOT NULL}**
    - **ALTER COLUMN ... {NULL | NOT NULL}**
  - **ALTER DATABASE ... SET DEFAULT CHARACTER SET**
  - **IDENTITY columns**
  - **RECREATE SEQUENCE, RECREATE GENERATOR**
  - **DDL triggers**

Whats new in Firebird 3

- **Security**
  - New virtual table SEC$USERS
  - Support for database encryption
  - GRANT CREATE | ALTER | DROP <object> TO <user> | <role>
  - GRANT ROLE TO ROLE
- **Monitoring**
  - Extended statistics, queries plan's, ...
- **To be continued :-)**

Common SQL : MERGE

Full SQL 2008 syntax

```
MERGE INTO <table>
   USING <table_or_join>
      ON <search_condition>

   [WHEN MATCHED [AND <search_condition>] THEN
      UPDATE SET col1 = val1, ..., colN = valN
     |
      DELETE]

   [WHEN NOT MATCHED [AND <search_condition>] THEN
      INSERT [(col1, ..., colN)] VALUES (val1, ..., valN)]
```


Common SQL : MERGE

DELETE substatement and multiply WHEN clauses

```
MERGE INTO TABLE
   USING LOG
      ON TABLE.PK = LOG.PK

   WHEN MATCHED AND LOG.ACTION = 'D' THEN
      DELETE

   WHEN MATCHED THEN                    -- second WHEN MATCHED clause
      UPDATE SET col1 = LOG.val1, ...

   WHEN NOT MATCHED THEN
      INSERT (col1, ...) VALUES (LOG.val1, ...)
```


Common SQL : MERGE

RETURNING clause

```
MERGE INTO <table>
   USING <table_or_join>
      ON <search_condition>

   [WHEN MATCHED [AND <search_condition>] THEN
      UPDATE SET col1 = val1, ..., colN = valN
     |
      DELETE]

   [WHEN NOT MATCHED [AND <search_condition>] THEN
      INSERT [(col1, ..., colN)] VALUES (val1, ..., valN)]

   [RETURNING ... [INTO ...]]
```


Common SQL : WINDOW FUNCTIONS

Syntax

```
<window function> ::=
    <window function type> OVER (<window specification>)

<window function type> ::=
     <aggregate function>                -- aggregate
   | <rank function type>                -- ranking
   | ROW_NUMBER                          -- yes, it is row number ;)
   | <lead or lag function>              -- navigational
   | <first or last value function>
   | <nth value function>
```


Common SQL : WINDOW FUNCTIONS

Syntax

```
<aggregate function> ::=
    AVG | MAX | MIN | SUM | COUNT | LIST

<rank function type> ::=
    RANK | DENSE_RANK

<lead or lag function> ::=
    LEAD | LAG

<first or last value function> ::=
    {FIRST_VALUE | LAST_VALUE} (<value>)

<nth value function> ::=
    NTH_VALUE (<value>, <nth row>)
```


Common SQL : WINDOW FUNCTIONS

Syntax

```
<window function> ::=
    <window function type> OVER (<window specification>)

<window specification> ::=
    [PARTITION BY column1, ...]
    [ORDER BY column1 [ASC|DESC] [NULLS {FIRST|LAST}], ... ]
```


Common SQL : WINDOW FUNCTIONS

Example

```
SELECT A, B, C,
       SUM(C) OVER(),
       SUM(C) OVER(ORDER BY A, B),
       SUM(C) OVER(PARTITION BY A),
       SUM(C) OVER(PARTITION BY A ORDER BY B)
```

A	B	C	SUM	SUM1	SUM2	SUM3
1	1	30	141	30	60	30
1	2	20	141	50	60	50
1	3	10	141	60	60	60
2	1	25	141	85	40	25
2	2	15	141	100	40	40
3	1	41	141	141	41	41

Common SQL : substring search using REGEXP

Syntax

```
SUBSTRING(<string> SIMILAR <pattern> ESCAPE <char>)
```

Pattern rules

- **Pattern consists from three parts :**

  R = <R1> <E> " <R2> <E> " <R3>

  - **R1 prefix**
  - **R2 matching**
  - **R3 suffix**
  - **E escape character, mandatory**

Common SQL : substring search using REGEXP

Syntax

```
SUBSTRING(<string> SIMILAR <pattern> ESCAPE <char>)
```

Search steps

S = <S1> <S2> <S3>

```
1.<S>      SIMILAR TO <R1> <R2> <R3> ESCAPE <E>

2.<S1>     SIMILAR TO <R1>          ESCAPE <E> AND
  <S2> <S3>   SIMILAR TO <R2> <R3> ESCAPE <E>

3.<S3>     SIMILAR TO <R3>          ESCAPE <E> AND
  <S2>     SIMILAR TO <R2>          ESCAPE <E>

4.<S2> is result
```


Common SQL : substring search using REGEXP

Syntax

```
SUBSTRING(<string> SIMILAR <pattern> ESCAPE <char>)
```

Example

```
SUBSTRING('abc-12b34xyz' SIMILAR '%\"[\+\-]?[0-9]+\"%' ESCAPE '\')

R1 = %
R2 = [\+\-]?[0-9]+
R3 = %
```

1) `'abc-12b34xyz' SIMILAR TO '%[\+\-]?[0-9]+%' ESCAPE '\'`

2)
```
'abc'          SIMILAR TO '%'               ESCAPE '\' AND
'-12b34xyz'    SIMILAR TO '[\+\-]?[0-9]+%'  ESCAPE '\'
```

3)
```
'-12'          SIMILAR TO '[\+\-]?[0-9]+'   ESCAPE '\' AND
'b34xyz'       SIMILAR TO '%'               ESCAPE '\'
```

Result

```
'-12'
```


Common SQL : BOOLEAN data type

Syntax

```
<data_type> ::= BOOLEAN

<boolean_literal> ::= TRUE | FALSE | UNKNOWN
```

Storage

```
1 byte
```

Client support (XSQLDA)

```
#define SQL_BOOLEAN 32764
```


Common SQL : BOOLEAN data type

Truth tables

AND	True	False	Unknown
True	*True*	*False*	*Unknown*
False	*False*	*False*	*False*
Unknown	*Unknown*	*False*	*Unknown*

OR	True	False	Unknown
True	*True*	*True*	*True*
False	*True*	*False*	*Unknown*
Unknown	*True*	*Unknown*	*Unknown*

IS	True	False	Unknown
True	*True*	*False*	*False*
False	*False*	*True*	*False*
Unknown	*False*	*False*	*True*

NOT	
True	*False*
False	*True*
Unknown	*Unknown*

Common SQL : BOOLEAN data type

Examples

```
CREATE TABLE TBOOL (ID INT, BVAL BOOLEAN);
COMMIT;

INSERT INTO TBOOL VALUES (1, TRUE);
INSERT INTO TBOOL VALUES (2, 2 = 4);
INSERT INTO TBOOL VALUES (3, NULL = 1);
COMMIT;
```

```
SELECT * FROM TBOOL

            ID     BVAL
============= =======
             1 <true>
             2 <false>
             3 <null>
```


Common SQL : BOOLEAN data type

Examples

1. Test for TRUE value

```
SELECT * FROM TBOOL
 WHERE BVAL

            ID     BVAL
============ =======
             1 <true>
```

2. Test for FALSE value

```
SELECT * FROM TBOOL
 WHERE BVAL IS FALSE

            ID     BVAL
============ =======
             2 <false>
```

3. Tests for UNKNOWN value

```
SELECT * FROM TBOOL
 WHERE BVAL IS UNKNOWN

            ID     BVAL
============ =======
             3 <null>


SELECT * FROM TBOOL
 WHERE BVAL = UNKNOWN


SELECT * FROM TBOOL
 WHERE BVAL <> UNKNOWN
```


Common SQL : BOOLEAN data type

Examples

4. Boolean values in SELECT list

```
SELECT ID, BVAL, BVAL AND ID < 2
  FROM TBOOL

           ID     BVAL
============ ======= =======
           1 <true>  <true>
           2 <false> <false>
           3 <null>  <false>
```


Common SQL : cursor stability

The issue

- Famous infinite insertion circle (CORE-92)

```
INSERT INTO T
SELECT * FROM T
```

- DELETE more rows than expected (CORE-634)

```
DELETE FROM T
 WHERE ID IN (SELECT FIRST 1 ID FROM T)
```

- All DML statements is affected (INSERT, UPDATE, DELETE, MERGE)
- Common ticket at tracker CORE-3362

Common SQL : cursor stability

The reason of the issue

- DML statements used implicit cursors :
  - `INSERT INTO T SELECT ... FROM T`
    works as

```
FOR SELECT <values> FROM T INTO <tmp_vars>
  DO INSERT INTO T VALUES (<tmp_vars>)
```

  - `UPDATE T SET <fields> = <values> WHERE <conditions>`
    works as

```
FOR SELECT <values> FROM T WHERE <conditions> INTO <tmp_vars>
    AS CURSOR <cursor>
DO UPDATE T SET <fields> = <tmp_vars>
       WHERE CURRENT OF <cursor>
```

  - `DELETE` works like `UPDATE`

Common SQL : cursor stability

The “standard” way

- Rows to be inserted\updated\deleted should be marked first
- Marked rows is inserted\updated\deleted then
- Pros
  - rowset is stable and is not affected by DML statement itself
- Cons
  - Marks should be saved somewhere and rows will be visited again, or
  - Whole marked rows should be saved somewhere and this store will be visited again
- Note : this could be reached in Firebird using (well known) workaround: ***force query to have SORT in PLAN - it will materialize implicit cursor and make it stable***

Common SQL : cursor stability

The Firebird 3 way

- Use undo-log to see if record was already modified by current cursor
  - if record was inserted - ignore it
  - if record was updated or deleted - read backversion
- Pros
  - No additional bookkeeping required
  - No additional storage required
  - Relatively easy to implement
- Cons
  - Inserted records could be visited (but ignored, of course)
  - Backversions of updated\deleted records should be read
  - Not works with SUSPEND in PSQL

Common SQL : cursor stability

PSQL notes

- **PSQL cursors with SUSPEND inside still not stable !**

  This query still produced infinite circle

```
FOR SELECT ID FROM T INTO :ID
  DO BEGIN
    INSERT INTO T (ID) VALUES (:ID);
    SUSPEND;
  END
```


Common SQL : cursor stability

PSQL notes

- **PSQL cursors without SUSPEND inside is stable, this could change old behavior**

```
FOR SELECT ID FROM T WHERE VAL IS NULL INTO :ID
  DO BEGIN
    UPDATE T SET VAL = 1
     WHERE ID = :ID;
  END
```


Common SQL : improvements in GTT

- Global temporary tables is writable even in read-only transactions
  - Read-only transaction in read-write database
    - Both GTT ON COMMIT PRESERVE ROWS and COMMIT DELETE ROWS
  - Read-only transaction in read-only database
    - GTT ON COMMIT DELETE ROWS only
- Faster rollback for GTT ON COMMIT DELETE ROWS
  - No need to backout records on rollback
- Garbage collection in GTT is not delayed by active transactions of another connections
- All this improvements is backported into v2.5.1 too

PSQL : SQL functions

Syntax

```
{CREATE [OR ALTER] | ALTER | RECREATE} FUNCTION <name>
    [(param1 [, ...])]
    RETURNS <type>
AS
BEGIN
    ...
END
```

Example

```
CREATE FUNCTION F(X INT) RETURNS INT
AS
BEGIN
  RETURN X+1;
END;

SELECT F(5) FROM RDB$DATABASE;
```


PSQL : SQL sub-routines

Syntax

- **Sub-procedures**

```
DECLARE PROCEDURE <name> [(param1 [, ...])]
    [RETURNS (param1 [, ...])]
AS
...
```

- **Sub-functions**

```
DECLARE FUNCTION <name> [(param1 [, ...])]
    RETURNS <type>
AS
...
```


PSQL : SQL sub-routines

Example

```
EXECUTE BLOCK RETURNS (N INT)
AS
  DECLARE FUNCTION F(X INT) RETURNS INT
  AS
  BEGIN
    RETURN X+1;
  END;
BEGIN
  N = F(5);
  SUSPEND;
END
```


PSQL : Packages

```
-- package header, declarations only

CREATE OR ALTER PACKAGE TEST
AS
BEGIN
  PROCEDURE P(I INT) RETURNS (O INT);   -- public procedure
  FUNCTION  F(I INT) RETURNS INT;       -- public function
...
END
```


PSQL : Packages

```
-- package body, implementation
RECREATE PACKAGE BODY TEST
AS
BEGIN
  FUNCTION F1(I INT) RETURNS INT;        -- private function

  PROCEDURE P(I INT) RETURNS (O INT)
  AS
  BEGIN
  END;

  FUNCTION F(I INT) RETURNS INT
  AS
  BEGIN
  END;

  FUNCTION F1(I INT) RETURNS INT
  AS
  BEGIN
  END;
END
```


PSQL : Packages

```
-- Usage of the packaged procedure

EXECUTE PROCEDURE TEST.P(1);

SELECT TEST.F(x) FROM FOO;
```


PSQL : EXCEPTION with parameters

Syntax

a) Exception text could contain parameters markers (@1, @2, ...)

```
CREATE EXCEPTION EX_WITH_PARAMS 'Error @1 : @2';
```

b) New clause **USING** allows to set parameters values when exception raised:

```
EXCEPTION EX_WITH_PARAMS USING (1, 'You can not do it');
```


DDL : IDENTITY columns

Syntax

```
<column definition> ::=
   <name> <type> GENERATED BY DEFAULT AS IDENTITY <constraints>
```

Rules

- Column type – INTEGER or NUMERIC(P, 0)
- Implicit NOT NULL
- Not guarantees uniqueness
- Can not have DEFAULT clause
- Can not be COMPUTED BY

DDL : DDL triggers

Syntax

```
<ddl-trigger> ::=

    {CREATE | RECREATE | CREATE OR ALTER} TRIGGER <name>
        [ACTIVE | INACTIVE] {BEFORE | AFTER} <ddl event>
        [POSITION <n>]

<ddl event> ::=

    ANY DDL STATEMENT
  | <ddl event item> [{OR <ddl event item>}...]

<ddl event item> ::=

    {CREATE | ALTER | DROP}

    {TABLE | PROCEDURE | FUNCTION | TRIGGER | EXCEPTION |
     VIEW  | DOMAIN    | SEQUENCE | INDEX   | ROLE      |
     USER  | COLLATION | PACKAGE  | PACKAGE BODY        |
     CHARACTER SET }
```


DDL : DDL triggers

New context variables (RDB$GET_CONTEXT)

- Namespace *DDL_TRIGGER*
- Defined inside DDL trigger only
- Read only
- Predefined variables:
  - *DDL_EVENT* - kind of DDL event
  - *OBJECT_NAME* – name of metadata object
  - *SQL_TEXT* - text of SQL query

DDL : DDL triggers

Example

```
CREATE EXCEPTION EX_BAD_SP_NAME
    'Name of procedures must start with ''@1'' : ''@2''';

CREATE TRIGGER TRG_SP_CREATE BEFORE CREATE PROCEDURE
AS
DECLARE SP_NAME VARCHAR(255);
BEGIN
  SP_NAME = RDB$GET_CONTEXT('DDL_TRIGGER', 'OBJECT_NAME');

  IF (SP_NAME NOT STARTING 'SP_')
  THEN EXCEPTION EX_BAD_SP_NAME USING ('SP_', SP_NAME);
END;
```


Security

- **New virtual table `SEC$USERS`**

```
CREATE DOMAIN RDB$USER AS CHAR(31) CHARACTER SET UNICODE_FSS;
CREATE DOMAIN SEC$UID AS INTEGER;
CREATE DOMAIN SEC$NAME_PART AS CHAR(31) CHARACTER SET
    UNICODE_FSS;

CREATE TABLE SEC$USERS
(
    SEC$USER_NAME     RDB$USER,
    SEC$GROUP_NAME    RDB$USER,
    SEC$UID           SEC$UID,
    SEC$GID           SEC$GID,
    SEC$FIRST_NAME    SEC$NAME_PART,
    SEC$MIDDLE_NAME   SEC$NAME_PART,
    SEC$LAST_NAME     SEC$NAME_PART
);
```


Security

- **Support for database encryption**
  - `ALTER DATABASE ENCRYPT WITH <plugin name>`
  - `ALTER DATABASE DECRYPT`
- **Features**
  - Based on plugins (sample crypto-plugin code in examples)
  - Background and restartable (allow to stop and start database during encryption work in progress)
  - Encryption key could be supplied by client application or by crypto-plugin itself

THANK YOU FOR ATTENTION

Questions ?

Firebird official web site

Firebird tracker

hvlad@users.sf.net