New SQL Features in Firebird 3
Sponsors!

Firebird Conference 2014

24-25 October

Prague
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
What's 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
  - Cursors could be references as record variable
  - Bi-directional cursors
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
  - Database LINGER
What's new in Firebird 3

- **Security**
  - User management
  - Support for database encryption
  - Enhanced object privileges
  - DDL privileges
  - Database-level privileges
  - `SET ROLE TO <role>`

- **Monitoring**
  - Extended statistics, queries plan's, ...
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

```sql
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>

>window specification> ::= 

[PARTITION BY column1, ...]

[ORDER BY column1 [ASC|DESC] [NULLS {FIRST|LAST}], ... ]
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>) [FROM_FIRST | FROM_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)
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>SUM</th>
<th>SUM1</th>
<th>SUM2</th>
<th>SUM3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>30</td>
<td>141</td>
<td>30</td>
<td>60</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>20</td>
<td>141</td>
<td>50</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
<td>141</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>25</td>
<td>141</td>
<td>85</td>
<td>40</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>15</td>
<td>141</td>
<td>100</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>41</td>
<td>141</td>
<td>141</td>
<td>41</td>
<td>41</td>
</tr>
</tbody>
</table>
Common SQL: substring search using REGEXP

Syntax

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

Rules

- Pattern
  - \( R = \langle R1 \rangle \langle E \rangle " \langle R2 \rangle \langle E \rangle " \langle R3 \rangle \)

- Search
  - \( S = \langle S1 \rangle \langle S2 \rangle \langle S3 \rangle \)

  1) \( \langle S \rangle \) SIMILAR TO \( \langle R1 \rangle \langle R2 \rangle \langle R3 \rangle \) ESCAPE \( \langle E \rangle \)

  2) \( \langle S1 \rangle \) SIMILAR TO \( \langle R1 \rangle \) ESCAPE \( \langle E \rangle \) AND \( \langle S2 \rangle \langle S3 \rangle \) SIMILAR TO \( \langle R2 \rangle \langle R3 \rangle \) ESCAPE \( \langle E \rangle \)

  3) \( \langle S2 \rangle \) SIMILAR TO \( \langle R2 \rangle \) ESCAPE \( \langle E \rangle \) AND \( \langle S3 \rangle \) SIMILAR TO \( \langle R3 \rangle \) ESCAPE \( \langle E \rangle \)

- Result
  - \( S2 \)
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 'b34xyz' 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

\[ <\text{data\_type}> ::= \text{BOOLEAN} \]
\[ <\text{boolean\_literal}> ::= \text{TRUE} \mid \text{FALSE} \mid \text{UNKNOWN} \]

Storage

1 byte

Client support (XSQLDA)

#define SQL_BOOLEAN 32764
# Common SQL: BOOLEAN data type

## Truth tables

<table>
<thead>
<tr>
<th>AND</th>
<th>True</th>
<th>False</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>Unknown</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
<td>False</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>True</th>
<th>False</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>True</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IS</th>
<th>True</th>
<th>False</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Unknown</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th>True</th>
<th>False</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
<td></td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>ID</th>
<th>BVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;true&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;false&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>
```
Common SQL : BOOLEAN data type

Examples

1. Test for TRUE value

```
SELECT * FROM TBOOL
WHERE BVAL
```

<table>
<thead>
<tr>
<th>ID</th>
<th>BVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;true&gt;</td>
</tr>
</tbody>
</table>

2. Test for FALSE value

```
SELECT * FROM TBOOL
WHERE BVAL IS FALSE
```

<table>
<thead>
<tr>
<th>ID</th>
<th>BVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>&lt;false&gt;</td>
</tr>
</tbody>
</table>

3. Tests for UNKNOWN value

```
SELECT * FROM TBOOL
WHERE BVAL IS UNKNOWN
```

<table>
<thead>
<tr>
<th>ID</th>
<th>BVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

```
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
```

<table>
<thead>
<tr>
<th>ID</th>
<th>BVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;true&gt;</td>
<td>&lt;true&gt;</td>
</tr>
<tr>
<td>2 &lt;false&gt;</td>
<td>&lt;false&gt;</td>
</tr>
<tr>
<td>3 &lt;null&gt;</td>
<td>&lt;false&gt;</td>
</tr>
</tbody>
</table>
Common SQL : cursor stability

The issue

- Famous infinite insertion circle (CORE-92)
  
  ```sql
  INSERT INTO T
  SELECT * FROM T
  ```

- DELETE more rows than expected (CORE-634)
  
  ```sql
  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 \textit{SORT} in \textit{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

  ```sql
  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

```sql
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

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

Example

```sql
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  
  ...  
```
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 P1(I INT) RETURNS (O INT);  -- public procedure
END

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

  PROCEDURE P1(I INT) RETURNS (O INT)
  AS
  BEGIN
  END;

  FUNCTION F1(I INT) RETURNS INT
  AS
  BEGIN
    RETURN 0;
  END;
END
PSQL : EXCEPTION with parameters

Syntax

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

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

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

```sql
EXCEPTION EX_WITH_PARAMS USING (1, 'You can not do it');
```
PSQL: Cursors references as record variable

FOR SELECT A, B, C FROM ...
    AS CURSOR C1                       // no INTO clause
DO
BEGIN
    ...
    INSERT INTO ...
END
DDL : IDENTITY columns

Syntax

\<column \ definition> ::= \<name> \<type>

    GENERATED BY DEFAULT AS IDENTITY [STARTS WITH \<number\>]

\<constraints>\

\<alter \ column \ definition> ::= \<name>

    RESTART [WITH \<number\>]
DDL: IDENTITY columns

**Rules**

- Column type – INTEGER or NUMERIC(P, 0)
- Implicit NOT NULL
- Not guarantees uniqueness
- Can not have DEFAULT clause
- Can not be COMPUTED BY
- Can not be altered to become non-identity and vice versa
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 }

Prague 2014
Whats new in Firebird SQL
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
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;
DDL : Database LINGER

• **Linger purposes**
  • Allow engine to keep in memory some database related structures after last disconnect:
    • Page cache
    • Background GC thread continue to work
  • For Shared Cache only
  • LINGER is switched off for current session, if
    • database shutdown
    • gfix -nolinger

• **Syntax**
  • `ALTER DATABASE SET LINGER TO <seconds>`
  • `ALTER DATABASE DROP LINGER`
Security : user management

Syntax

CREATE [OR ALTER] USER <string>
[SET] <option> [, ...]

ALTER CURRENT USER
[SET] <option> [, ...]

<option> ::=

PASSWORD <string>
FIRSTNAME <string>
MIDDLENAME <string>
LASTNAME <string>

{GRANT | REVOKE} ADMIN ROLE
{ACTIVE | INACTIVE}
TAGS (<list>)


Security : user management

Example

```
CREATE USER Vlad PASSWORD pass INACTIVE;
ALTER USER Vlad ACTIVE;
ALTER USER Vlad SET TAGS (id = 1, x = 'abcd');
ALTER USER Vlad SET TAGS (x = 'xyz');
ALTER USER Vlad SET TAGS (DROP x);
```
Security: user management

New virtual tables

CREATE TABLE SEC$USERS
(
    SEC$USER_NAME RDB$USER,
    SEC$FIRST_NAME SEC$NAME_PART,
    SEC$MIDDLE_NAME SEC$NAME_PART,
    SEC$LAST_NAME SEC$NAME_PART,
    SEC$ACTIVE RDB$BOOLEAN,
    SEC$ADMIN RDB$BOOLEAN,
    SEC$DESCRIPTION RDB$DESCRIPTION
);

CREATE TABLE SEC$USER_ATTRIBUTES
(
    SEC$USER_NAME RDB$USER,
    SEC$KEY SEC$KEY,
    SEC$VALUE SEC$VALUE
);

What's new in Firebird SQL
Security : EXECUTE privilege

**Syntax**

```sql
GRANT EXECUTE ON <object_type> <object_name>
    TO <grantee> [WITH GRANT OPTION]

REVOKE [GRANT OPTION FOR]
    EXECUTE ON <object_type> <object_name>
    FROM <grantee> [GRANTED BY <grantor>]
```

object_type ::= PROCEDURE
   | FUNCTION
   | PACKAGE
Security : USAGE privilege

Syntax

GRANT USAGE ON <object_type> <object_name>
  TO <grantee> [WITH GRANT OPTION]

REVOKE [GRANT OPTION FOR]
  USAGE ON <object_type> <object_name>
  FROM <grantee> [GRANTED BY <grantor>]

object_type ::= {DOMAIN | EXCEPTION | GENERATOR | SEQUENCE |
                   CHARACTER SET | COLLATION}
Security : DDL privileges

Syntax

GRANT <ddl_privileges> <object>
   TO <grantee> [WITH GRANT OPTION]

REVOKE [GRANT OPTION FOR] <ddl_privileges> <object>
   FROM <grantee> [GRANTED BY <grantor>]

ddl_privileges ::= ALL [PRIVILEGES]
   | ddl_privilege[, ddl_privilege]

ddl_privilege ::= {CREATE | ALTER ANY | DROP ANY}

object ::= {TABLE | VIEW | PROCEDURE | FUNCTION | PACKAGE |
   GENERATOR | SEQUENCE | DOMAIN | EXCEPTION | ROLE |
   CHARACTER SET | COLLATION | FILTER}
Security: database privileges

Syntax

REVOKE [GRANT OPTION FOR] <ddl_privileges> <object_name>
FROM <grantee> [GRANTED BY <grantor>]

ddl_privileges ::= ALL [PRIVILEGES]
| ddl_privilege[, ddl_privilege]

ddl_privilege ::= {CREATE | ALTER ANY | DROP ANY}
Security : SET ROLE

**Syntax**

- `SET ROLE <name>`
- `SET TRUSTED ROLE`
THANK YOU FOR ATTENTION

Questions ?

Firebird official web site

Firebird tracker

hvlad@users.sf.net