ABOUT ME

STORED PROCEDURES

Session HOWTO-P207-R

Firebird Conference • Praha 2005 • Firebird Conference

Lucas Franzen

- FRRED Software GmbH
    - Logistics software for transport, export
      Add-On for ERP-Systems (SAP, BAAN, Navision, ProAlpha, etc.)
    - approx. 30 installations all over Germany
    - all installations with **Firebird**
    - average 5 users, average database size 100 MB
    - Database with ~300 tables, ~300 stored procedures
- Working with InterBase since v4 (Delphi 1)
- Working with Firebird since the very beginning (Version 0.9)
- Member of the **Firebird Foundation** since the very beginning

FirebirdSQL
Foundation
FULL MEMBER
Supporting
Firebird development

WHAT IS THIS SESSION **ABOUT**?

Stored Procedures

This session is about STORED PROCEDURES.

- What are Stored Procedures, how and when to use them.
- What are the advantages of using them.
- How to write, maintain and optimize them.
- What are the pitfalls, what to obey in general?

For this purpose there are

- Code examples
- An example database

WHAT ARE STORED PROCEDURES?

In short

Stored Procedures are pre-compiled functions that are stored within the database and executed at server side.

They do give a wide variety of possible enhancements and functionality and also may help increasing the speed.

They can be used as well for doing multiple DML operations as they can be used as virtual tables for building complex queries.

All DML statements can be used within Stored Procedures, plus the extensions that are introduced by the Procedural SQL.

BUT neither dynamic SQL nor DDL can be used
(exceptions to this rule to follow).

WHAT ARE THEY **GOOD FOR**?

- The complete work is done at server-side
  - increasing speed
  - less network traffic
- Flexibility by "SQL extensions"
  - Procedural Structured Query Language – PSQL is enhancing SQL language.
- Execute multiple statements within a **single** call
- Use as a Black Box
- Extending privileges within the database
- Centralizing of Business Rules
- Posting of Events
- Stored Procedures are executable from TRIGGERS.

RESTRICTIONS

- Following Restrictions are given
  - Parameters and Variables can't be Domains!
    - *Domains might include a Check-Constraints, so changing field types would imply to re-compiled automatically all procedures.*
  - **DDL** Statements are invalid:
    - CREATE / ALTER / DROP
  - No Cursor declaration
    - declare cursor
    - Fetch
  - No dynamical SQL-Statements
    - Example: A tablename can't be replaced by the value of a variable
    - Exception to this rule: **EXECUTE STATEMENT**

(from Firebird 1.5 on)

INSIDE THE DATABASE

Stored Procedures

Stored Procedures are compiled and stored in the database in this compiled form.
Having a look at the System Tables:

SYSTEM TABLE: RDB$PROCEDURES:

```
CREATE TABLE RDB$PROCEDURES (
  RDB$PROCEDURE_NAME                 CHAR (31),  ← Procedurename
  RDB$PROCEDURE_ID                   SMALLINT,   ← Unique ID
  RDB$PROCEDURE_INPUTS               SMALLINT,   ← Count of Input-Parameters
  RDB$PROCEDURE_OUTPUTS              SMALLINT,   ← Count of Output-Parameters
  RDB$DESCRIPTION                    BLOB,       ← Description (use as you like)
  RDB$PROCEDURE_SOURCE               BLOB,       ← Source (just for the curious!)
  RDB$PROCEDURE_BLR                  BLOB,       ← compiled Code in BLR
  RDB$SECURITY_CLASS                 CHAR (31),
  RDB$OWNER_NAME                     CHAR (31),  ← Creator of this procedure
  RDB$RUNTIME                        BLOB,
  RDB$SYSTEM_FLAG                    SMALLINT    ← 1 = SYSTEM GENERATED
);
```

the compiled source is stored as BLR (**B**inary **L**anguage **R**epresentation) in the field RDB$PROCEDURE_BLR.
The source code itself has just *informational character* and can be hidden or replaced by any dummy entry.

```
UPDATE RDB$PROCEDURES SET
  RDB$PROCEDURE_SRC = NULL
WHERE RDB$PROCEDURE_NAME = <PROCEDURE_NAME>

  (AND RDB$SYSTEM_FLAG <> 1 AND RDB$SYSTEM_FLAG  IS NOT NULL)
```

STORED PROCEDURE **LANGUAGE** - PSQL

PSQL = Procedural Structured Query Language

PSQL is SQL + additional language elements, as

- declare variable
- if ( ) then ... else ...
- While ( ) do ...
- when ...
- for select
  do begin
  end
- suspend
- leave(*), break(**), exit

(*) available from FB 1.5 on, deprecates break

(**) available from FB 1.0 on

STORED PROCEDURE **LANGUAGE** – PSQL (continued)

- ROW_COUNT
  - ROW_COUNT can be used to retrieve the count of affected rows of the last DML-Statement.
    Example:

```
UPDATE TABLE1 SET FIELD1 = 0 WHERE ID = :ID;
IF (ROW_COUNT = 0) THEN
  INSERT INTO TABLE1 (ID, FIELD1) VALUES (:ID, 0);
```

- LEAVE / BREAK
  - Exit loops with LEAVE / BREAK (Leave deprecates break).
    Execution of the code will be continued after the end that's encapsulating the statement.

- GDSCODE / SQLCODE
  - Can be used within WHEN statements.
    - GDSCODE holds the ISC-Errorcode
    - SQLCODE holds the error number

EXECUTE STATEMENT

Since FB1.5 there's the possibility to use dynamic statements and even DDL from within procedures by **EXECUTE STATEMENT**

EXECUTE STATEMENT may execute any SQL operation that returns:

1. **NOTHING**
   i.e. INSERT, UPDATE, DELETE, EXECUTE PROCEDURE or any other DDL statement, except CREATE/DROP DATABASE.
   Example: **EXECUTE STATEMENT <string>**;

2. **A SINGLE RECORD**
   Only singleton SELECT operations may be executed this way.
   Example: **EXECUTE STATEMENT <string> INTO :var1, [..., :var^{n}] ;**

4. **ANY NUMBER OF RECORDS**
   Example: **FOR EXECUTE STATEMENT <string>**
   INTO :var^{1}, ..., :var^{n} DO
   <compound-statement>;

DRAWBACKS AND **RESTRICTIONS** OF EXECUTE STATEMENT

- The "EXECUTE STATEMENT" DSQL-String can't hold parameters!
  All Variables inside the static part have to be set before the execution.
- Embedded statements can't be checked syntactically.
- There's no check of the system tables if the involved objects are still part of the database!
- Operations are slow – the embedded statements have to be prepared before EACH execution.
- Datatypes of return values are strictly checked, to circumvent unexpectable typecasting errors.
  Example: The string "1234" could be converted to the Integer 1234, "abc" however would not and cause a typecast exception.
- If the stored procedure has special privileges on some objects, the dynamic statement will **NOT** inherit these.
  Privileges are restrict to the current user or role which is executing the procedure.

This feature was meant for **careful use!**
When using, take all factors into account.

PRINCIPLE

EXECUTE STATEMENT should only be used when

- there are no other means
- there are other means but they do perform much worse.

HOW TO **WRITE** A STORED PROCEDURE

Stored Procedures

- **CREATE** PROCEDURE <NAME>
  <List of Input-Params>
  <List of Output-Params>
  AS
    <declare variable <name> <datatype>;>
  BEGIN
    <code>
  END

- Altering / recompiling procedures
  - **ALTER** PROCEDURE <NAME>
  - **RECREATE** PROCEDURE <NAME>
  - **CREATE OR ALTER** PROCEDURE <NAME>

- Dropping procedures
  - **DROP** PROCEDURE <NAME>
    Only the Owner of the procedure or SYSDBA might drop a procedure!

What happens if...	USING **RECREATE**	USING **CREATE OR ALTER**
PROCEDURE <u>DOES</u> EXIST	**DROP PROCEDURE** **! ERROR IF DEPENDENCIES!** **CREATE PROCEDURE**	**ALTER PROCEDURE**
PROCEDURE <u>DOES NOT</u> EXIST	**CREATE PROCEDURE**	**CREATE PROCEDURE**

WRITING A STORED PROCEDURE

Example:

```
SET TERM #;
CREATE PROCEDURE SP_SUM_TWO_INTS (
  VALUE_1  INTEGER,
  VALUE_2  INTEGER,
)
RETURNS ( A_SUM INTEGER )
AS BEGIN
 A_SUM = VALUE_1 + VALUE_ 2;
END #
SET TERM ;#
```

Working with variables

Working with table values

Example:

```
SET TERM #;
CREATE PROCEDURE SP_GET_COUNTRY_DATA ( COUNRY_ID   INTEGER )
RETURNS (
  ISO_CODE            CHAR(2),
  COUNTRY_NAME        VARCHAR(40),
  COUNTRY_POPULATION  INTEGER
)
AS BEGIN
  SELECT COUNTRY_ISO_CODE, COUNTRY_NAME, COUNTRY_POPULATION
  FROM COUNTRY
  WHERE COUNTRY_ID = :COUNTRY_ID
  INTO :ISO_CODE, :COUNTRY_NAME, :COUNTRY_POPULATION
END #
SET TERM ;#
```

The **COLON** ":"

- In which cases the COLON has to precede the VARIABLE?
  This one of the major misunderstandings.

- Generally:
  The colon has to precede the variable when the variable will be used within a SQL statement
  (and otherwise would be misinterpreted as begin a part of the DML and not the value it holds).

CASES TO USE IT **- WHEN POPULATING OR USING WITHIN DML**	**DON'T USE** **- WHEN USED OUTSIDE DML**
SELECT .. FROM .. **INTO** :VAR1, :VAR2	IF (VAR1 = AVALUE) THEN ... (Comparing values)

INSERT INTO ..
VALUES (:VAR1, :VAR2)
	VAR1 = VAR2
	(Assigning values in code that's not DML)

UPDATE ... SET
FIELD1 = :VAR1 ...	

... WHERE FIELD1 = :VAR1	

Compiling: The semicolon and **SET TERM**

(I) SQL statements (including DDL) do use the semicolon for termination (in fact this is rather a client app problem)

(II) Stored procedure need to use the semicolon to separate and terminate different (P)SQL statements within the code

(III) Compiling procedures is an SQL statement so there's a problem arising from the contradiction of (I) and (II)

By bracketing the procedure with a **SET TERM** this problem will be solved.

Example

```
SET TERM #;                  ← the new termination symbol is # now
CREATE PROCEDURE ...
   CODE
END #              ← # = end of procedure
SET TERM ;#                  ← the new termination symbol is ; again
```

Watch out!
Some database tools add this on their own!

ErrorMessage: *(-104: Unexpected end of command*) the termination is missing.

ErrorMessage (*-104: Token unknown - line n, char 5. TERM*) the termination was already added by the tool.

EXECUTABLE or SELECTABLE?

- Stored Procedures can be invoked in two different ways.

1. **EXECUTE <PROCEDURE_NAME>** (<INPUT_PARAMS>)
   An executable Procedure always returns **one** "set" of parameters.

2. **SELECT** <FIELDS>
   FROM <PROCEDURE_NAME>
   <WHERE>
   <ORDER BY>
   <GROUP BY>

   A selectable procedure returns between **0 and <n>** records. The return parameters can be treated as fields
   → a selectable procedure can be treated as a regular table.

   The resultset can be further queried, for example by WHERE clauses.
   This will slow down the query, but in some cases it might allow working with very complex queries.

EXECUTABLE or SELECTABLE?

- Executable Procedure

```
SET TERM #;
CREATE PROCEDURE SP_SUM_UP (
  VALUE_1  INTEGER,
  VALUE_2  INTEGER,
)
RETURNS ( A_SUM INTEGER )
AS BEGIN
  A_SUM = VALUE_1 + VALUE_2;
END #
SET TERM ;#
```

- Selectable Procedure

```
SET TERM #;_
CREATE PROCEDURE SEL_CUSTOMER_NAMES ( CITY  VARCHAR(40) )
RETURNS (
  CUSTOMER_NAME   VARCHAR(40),
  CUSTOMER_NUMBER VARCHAR(40)
)
AS BEGIN
  FOR SELECT CUSTOMER_NAME, CUSTOMER_NUMBER
  FROM CUSTOMER
  WHERE CUSTOMER_CITY = :SITY
  INTO         : CUSTOMER_NAME, :CUSTOMER_NUMBER
  DO BEGIN
    SUSPEND;
  END
END #
SET TERM ;#
```

SUSPEND

- When working with SELECTABLE procedures SUSPEND has to be used.

  How SUSPEND does work:
  - execution of the procedure is suspended until the row is fetched from the client
  - After that code execution continues
  - When there's no more data to be retrieved SQL code 100 (end of data) is sent

- Suspend might be used in executable procedures (EXIT) **but** it isn't recommended!

- **Do use** SUSPEND in **selectable** procedures
- **Don't use** it in **executable** procedures

STORED PROCEDURES AT **CLIENTSIDE**

- When using stored procedures from a client application, the difference between EXECUTABLE and SELECTABLE Procedures is important!

  Which components to use?

EXECUTABLE PROCEDURE	**SELECTABLE PROCEDURE**
Stored-Procedure-components	Query / Cursor-Components
DSQL-components	Buffered- / Non-buffered-datasets

NAMING CONVENTIONS

There are no specials ones, but the ones that are given by Firebird

BUT:
Naming executable and selectable procedures differently, will make life easier!

For example:

- SP_GET_CUSTOMER_ID
  for EXECUTABLE Procedures
- SEL_CUSTOMER_DATA
  for Selectable Procedures.

When working with a bunch of stored procedure and using them internally it's a big advantage being able to judge from the name what and how it might do ...

SQL WITHIN STORED PROCEDURES

The **complete** SQL can be used in Stored Procedures!

- Not just
  - SELECT
  - UPDATE
  - DELETE, ... !

- also
  - LIKE,
  - CONTAINING,
  - STARTING WITH
  - CASE
  - etc.

can be used and not just within querying the data,
also comparing values is possible.

Examples →

EXAMPLE: STRING LENGTH

```
SET TERM #;
CREATE PROCEDURE SP_GET_STRINGLEN ( A_STRING VARCHAR(1024) )
RETURNS ( A_LENGTH INTEGER )
AS
 DECLARE VARIABLE CV VARCHAR (1024);  /* Compare value */
 DECLARE VARIABLE II INTEGER;          /* Iteration */
BEGIN
  CV = '';  /* COMPARE STARTS WITH AN EMPTY STRING */
  II = 0;
  A_LENGTH = -1;
  WHILE ( II <= 1024 ) DO
  BEGIN
    IF ( A_STRING LIKE CV ) THEN    /* does the compare value equal the given string? */
    BEGIN
      A_LENGTH = II;
      II = 1024;
    END
    II = II + 1;
    CV = CV || '_';     /* extend the Compare Value by any character, in SQL this is '_'*/
  END
END #
SET TERM ;#
```

EXAMPLE: SUBSTRING

```
CREATE PROCEDURE SP_GET_SUBSTRING (
  SRC VARCHAR (255), START_AT INTEGER, NLEN INTEGER )
RETURNS ( RESULT VARCHAR(255) )
AS
  DECLARE VARIABLE II    INTEGER;
  DECLARE VARIABLE CV    VARCHAR(255);
  DECLARE VARIABLE PFX   VARCHAR(255);
  DECLARE VARIABLE C     CHAR(1);
BEGIN
  CV = '';  RESULT = ''; PFX = '';
  IF ( START_AT > 1 ) THEN   /* FILL FIRST <START_AT> CHARACTERS WITH '_' */
  BEGIN
    II = 1;
    WHILE ( II < START_AT ) DO
    BEGIN
      PFX = PFX || '_';
      II = II + 1;
    END
  END
  II = START_AT;
  WHILE ( II < NLEN + START_AT ) DO
  BEGIN
    C = ' ';
    /* CHECK THE NEXT CHARACTER */
    IF ( SRC LIKE PFX || 'A%' ) THEN C = 'A';
    ELSE IF ( SRC LIKE PFX || 'B%' ) THEN C = 'B';
    ELSE IF ( SRC LIKE PFX || 'C%' ) THEN C = 'C';
    ELSE IF ( SRC LIKE PFX || 'D%' ) THEN C = 'D';
    ... etc ..
    RESULT = RESULT || :C;
    PFX = PFX || '_';
  END
```

EXAMPLE: SQR (Square Root)

```
CREATE PROCEDURE SQR ( NUMBER        DOUBLE PRECISION )
RETURNS              ( SQUARE_ROOT   DOUBLE PRECISION )
AS
  DECLARE VARIABLE AVALUE     DOUBLE PRECISION;
  DECLARE VARIABLE DIFF       DOUBLE PRECISION;
  DECLARE VARIABLE OK         INTEGER;
  DECLARE VARIABLE II         INTEGER;
  DECLARE VARIABLE DIFF_OK    DOUBLE PRECISION;

BEGIN
  /* Newton Iteration Method */
  AVALUE = 1;
  II = 0; OK = 0;  DIFF_OK = 0.00001;
  IF ( NUMBER < 0 ) SQUARE_ROOT = -1; /* SQUARE_ROOT of negative numbers ... */
  ELSE BEGIN
    WHILE ( OK = 0 ) DO
    BEGIN
    AVALUE = ( AVALUE + ( NUMBER / AVALUE ) ) / 2.00000;
      II = II + 1;
      DIFF = ( NUMBER - ( AVALUE * AVALUE ) );
      IF ( DIFF < 0.000000 ) THEN DIFF = DIFF * -1.000000;  /* ABS */
      IF ( DIFF <= DIFF_OK ) THEN
      BEGIN
        SQUARE_ROOT = AVALUE; /* RESULT is in range that was declared valid */
        OK = 1;
      END
      ELSE IF ( I > 100 ) THEN
      BEGIN
        OK = 1; /* if not ended after 100 trials, end now - or write another code */
        SQUARE_ROOT = -2;
      END
    END
  END
  SUSPEND;
END
```

PITFALLS - Not **initializing** variables

- Always initialize variables (mainly within loops)!

  Remark:
  Since FB1.5 variables can be initialized within the declaration
  DECLARE VARIABLE ABC VARCHAR(3) = 'ABC';

  When overloading variables within LOOPS (FOR SELECT ... DO , WHILE (...) DO ...)
  always initialize variables within the loop!

  If a select will not return any row the result won't be NULL –it's **nothing** - which won't change the value of the variable.

EXAMPLE

```
    BEGIN
      WHILE ( II < 100 ) DO
      BEGIN
         CUSTOMER = 'NOT ASSIGNED'; /* INITIALIZATION */
         SELECT CUSTOMER_NAME FROM CUSTOMERS
         WHERE CUST_ID = :II
         INTO :CUSTOMER;
         SUSPEND;
         II = II + 1;
      END
```

DATA

CUST_ID	CUSTOMER_NAME
1	ARTHUR DENT
2	JOHN DOE
4	MARVIN

RESULT

VAL **II**	WITH INITIALIZATION	WITHOUT INITIALIZATION
1	ARTHUR DENT	ARTHUR DENT
2	JOHN DOE	JOHN DOE
3	***NOT ASSIGNED***	***JOHN DOE***
4	MARVIN	MARVIN

PITFALLS - WRONG VARIABLE **DECLARATION**

- Error: Arithmetic exception string truncation or overflow

EXAMPLE:

```
CREATE TABLE TABLE_SELECT (
  TABLE_SELECT_ID       INTEGER NOT NULL,
  TABLE_SELECT_FIELD1   VARCHAR(40),          ← Field being declared as VARCHAR(40)
  TABLE_SELECT_FIELD2   VARCHAR(40),
  CONSTRAINT PK_TABLE_SELECT PRIMARY KEY (TABLE_SELECT_ID)
);


CREATE PROCEDURE SEL_TABLE_SELECT
RETURNS ( FIELD_1 VARCHAR(30) )               ← Returnfield being declared as VARCHAR(30)!
AS BEGIN

  FOR SELECT    TABLE_SELECT_FIELD1
  FROM          TABLE_SELECT
  INTO          :FIELD_1
  DO BEGIN
    SUSPEND;
  END
END #
SET TERM ;#
```

this will work as long as no value of TABLE_SELECT_FIELD1 will hold more than 30 characters.

➔ a VARCHAR(40) field should be filled with max. 40 characters!!! 🡐

WHAT SHOULD BE **OBEYED**

- Thoroughly checking and testing of stored procedures does help.

  A stored procedure should always be tested under real-time conditions in advance.

- Most common errors in stored procedures are:
  - Arithmetic exception string truncation or overflow
    General error, the cause is not easy to find.

    But the most common cause can be easily excluded by
    filling the fields / variables to their maximum size
    (→ see previous page)

  - Multiple rows in singleton select
    If a SELECT FROM .. INTO statement will return **more than one** record → working on unique data only.
    (!NULL within an unique index is allowed!).

WHAT STORED PROCEDURES ARE **PERFECT FOR**

Stored procedures are perfect for situations like

- Creating own functions without using UDFs
- having multiple statements
  - multiple insert / updates on different tables within one transaction context and one piece of code
- Statements beyond Standard-SQL
  - Queries that can't be expressed in Standard-SQL
  - Complex data retrieval for reports
- Complex calculations
- Extending privileges
- Black Boxes

EXTENDING **PRIVILEGES**

Stored Procedures

Stored procedures can be used for extending privileges...

EXAMPLE:

```
CREATE TABLE TABLE_SELECT (
  TABLE_SELECT_ID      INTEGER NOT NULL,
  TABLE_SELECT_FIELD1  VARCHAR(40),
  TABLE_SELECT_FIELD2  VARCHAR(40),
  CONSTRAINT PK_TABLE_SELECT PRIMARY KEY (TABLE_SELECT_ID)
);

USER_A has NO Access on this table!

SET TERM #;
CREATE PROCEDURE SEL_TABLE_SELECT
RETURNS ( FIELD_1 VARCHAR(40) )
AS BEGIN

  FOR SELECT    TABLE_SELECT_FIELD1
  FROM          TABLE_SELECT
  INTO          :FIELD_1
  DO BEGIN
    SUSPEND;
  END
END #
SET TERM ;#

GRANT EXECUTE ON PROCEDURE SEL_TABLE_SELECT TO USER_A;
USER_A HAS the right to execute the procedure!

GRANT SELECT ON TABLE_SELECT TO PROCEDURE SEL_TABEL_SELECT;
The procedure itself has the right to do a SELECT on the
table → USER_A has via the procedure Access to the table
(but only on the fields the procedure accesses).
```

→ ALLOWING RESTRICTED TABLE ACCESS ←

EXECUTING **PROCEDURES FROM WITHIN PROCEDURES**

- Stored procedures might execute / select from other stored procedures.
  - The difference between EXECUTABLE and SELECTABLE procedures is important!
    - Executable Procedure
      EXECUTE PROCEDURE <PROC_NAME>
      RETURNING VALUES
      :VAR1, :VAR2, ..., :VARn

    - Selectable Procedure
      [FOR] SELECT <FIELD_LIST>
      FROM <PROC_NAME>
      INTO <VARIABLE_LIST>

  - An important difference:
    Executable procedures **have to** (if at least one return parameter will be used) define a variable for each return parameter!

    Selectable procedures do have to supply a return parameter (field) just for the fields they do select!

RECURSIVE PROCEDURES

- Recursion is possible, since stored procedures can use other stored procedures, including **themselves.** (Recursion depth is approx. 1.000)

- Recursive procedures are perfect for reading tree structures.

EXAMPLE: ***MANAGING PRIVILEGES***

```
/* Table for usergroup privileges */
CREATE TABLE PRIVILEGE_POOL (
  PRIVILEGE_NAME                    T_STRING70,
  PRIVILEGE_PARENT                  T_STRING70,
  CONSTRAINT PK_PRIVILEGE_POOL PRIMARY KEY (PRIVILEGE_NAME )
);



/* Table for privilege assignments for each Usergroup */
CREATE TABLE GROUP_PRIVILEGES (
  GP_ID                             T_PRIMARYKEY NOT NULL,
  GROUP_ID                          T_FOREIGNKEY,
  PRIVILEGE                         T_STRING70,
  GP_GIVEN                          T_BOOL_NO,
  GP_LEVEL                          T_INTEGER,
  CONSTRAINT PK_GRUPPENRECHTE PRIMARY KEY ( GR_ID )
);
```

RECURSIVE PROCEDURES – A PRIVILEGE TREE

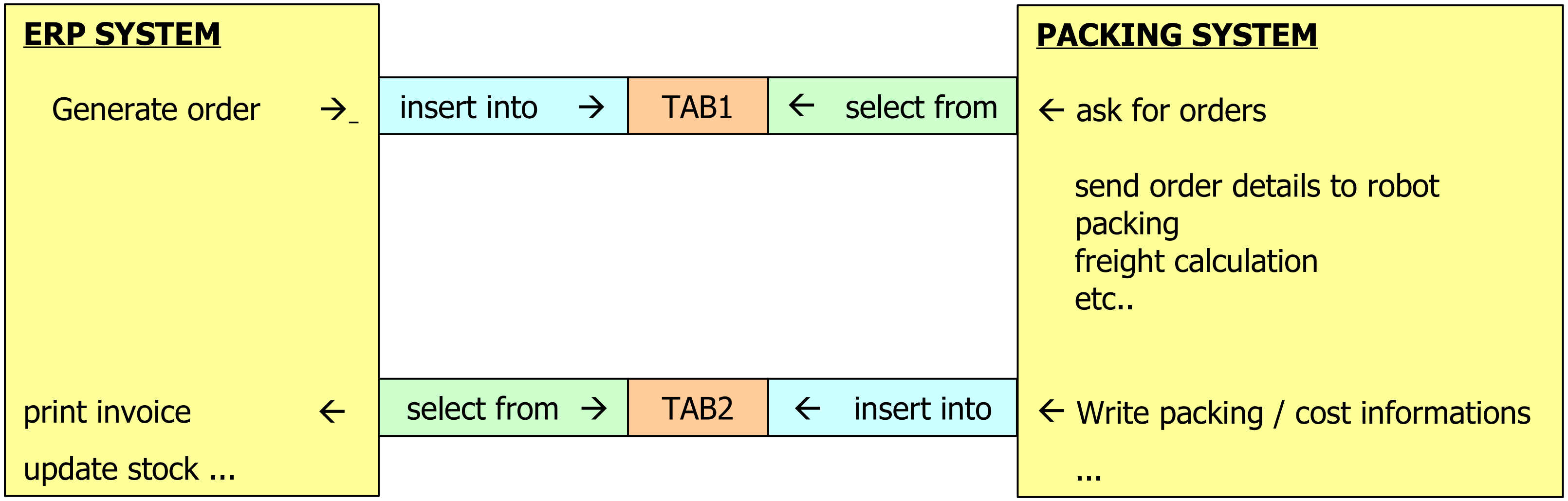
EXAMPLE:

```
CREATE PROCEDURE SEL_GROUP_PRIVS (
  START_PRIVILEGE                 VARCHAR (70),
  STARTLEVEL                      INTEGER,
  GROUP_ID                        INTEGER
  )
RETURNS (
  PRIVILEGE_NAME                  VARCHAR (70),
  PRIVILEGE_PARENT                VARCHAR (70),
  A_LEVEL                         INTEGER,
  GIVEN                           CHAR (1),
  GP_ID                           INTEGER
)
AS BEGIN
  /* read the first level, as given in the input param */
  FOR SELECT       P.PRIVILEGE_NAME, P.PRIVILEGE _PARENT, G.GP_ID, G.GP_GIVEN
  FROM             PRIVILEGE_POOL P
  LEFT JOIN        GROUP_PRIVILEGES  G ON P.PRIVILEGE_NAME = G.PRIVILEGE
  WHERE            P.PRIVILEGE_PARENT = :START_ PRIVILEGE   AND
                   G.GRP_ID           = :GROUP_ID
  INTO            :PRIVILEGE_NAME, :PRIVILEGE_PARENT, :GP_ID, :GIVEN
  DO BEGIN
    A_LEVEL = START_LEVEL + 1;
    SUSPEND;
    /* procedure calls itself, retrieving the next level (which will retrieve the next level...) */
    FOR SELECT * FROM SEL_GROUP_PRIVS ( :PRIVILEGE_NAME, :A_LEVEL, :GROUP_ID )
    INTO : PRIVILEGE_NAME, :PRIVILEGE_PARENT, :A_LEVEL, :GIVEN, :GP_ID
    DO BEGIN
      SUSPEND;
    END
  END
END
```

STORED PROCEDURES AS **BLACK BOX**

Stored Procedures

- SPs might work as black boxes.
  - As an interface between different databases
  - Between different "parts" of the database which separate teams working on
- Real life example:
  - ERP and our packing system; both running with Firebird



ERP SYSTEM

Generate order →

insert into → | TAB1 | ← select from

print invoice ←

update stock ...

select from → | TAB2 | ← insert into

PACKING SYSTEM

← ask for orders

send order details to robot
packing
freight calculation
etc..

← Write packing / cost informations

...

TRANSACTIONS

- Stored Procedures are ALWAYS within the transaction context of the executing component!
- Stored Procedures can't hold neither COMMIT nor ROLLBACK within their source.

What happens in case of an error?

- WITHOUT EXCEPTION HANDLING
  Any changes will be lost
- WITH EXCEPTION HANDLING
  All changes might be committed or rollbacked

EXCEPTION HANDLING

- Using **WHEN** allows to catch and handle Exceptions.
- A **WHEN**-clause has **ALWAYS** to be located at the end of the code.
- **WHEN ANY** does catch **ALL** exceptions
- **WHEN <ERRORNO>** just catches the exception with the matching errorcode
- The internal values of GDSCODE and SQLCODE can be used with **WHEN**.

Remark:
<WHEN GDSCODE> does work with the identifiers, **not** the errornumbers.

Download the list of Identifiers:
http://www.ibobjects.com/docs/fb_1_5_errorcodes.zip

EXAMPLE: EXCEPTION HANDLING

Example:

```
CREATE TABLE TABLE_SELECT (
  TABLE_SELECT_ID      INTEGER NOT NULL,
  TABLE_SELECT_FIELD1  VARCHAR(40),
  CONSTRAINT PK_TABLE_SELECT PRIMARY KEY (TABLE_SELECT_ID) );

CREATE EXCEPTION  E_UNKNOWN  'UNKNOWN ERROR';
CREATE EXCEPTION  E_EXISTS   'VALUE DOES ALREADY EXIST';
CREATE EXCEPTION  E_MISMATCH 'PARAMETER MISMATCH';

CREATE PROCEDURE SP_INSERT_TABLE_SELECT (
  NEW_ID   INTEGER,
  VALUE_1  VARCHAR(60) )
RETURNS ( ERRORCODE INTEGER )
AS BEGIN
  ERRORCODE = 0;
  INSERT INTO TABLE_SELECT (
    TABLE_SELECT_ID, TABLE_SELECT_FIELD1 )
  VALUES ( :NEW_ID, :VALUE_1 );

 WHEN SQLCODE -803 DO  /* attempt to store dupl. value */
 BEGIN
   EXCEPTION E_EXISTS;
 END
 WHEN ANY DO  /* any other error */
 BEGIN
   EXCEPTION E_UNKNOWN;
 END
END
```

DEBUGGING

Stored Procedures

- Tools including a debugger
  - Database Workbench
  - IB Expert
  - ...
  - But keep in mind:
    Tools do just emulate the procedure, most of the times they do help,
    sometimes they do fail!
- Use the database itself for locating the problem
  - Catch the exception by **WHEN ANY**
  - Use a return-variable for locating the problem
  - once located: get details by inserting information into a logtable

Example:

```
SET TERM #;
CREATE PROCEDURE SEL_PROCEDURE
RETURNS (
  FIELD_1 VARCHAR(40),
  ...
  ERROR_AT
)
AS BEGIN
  ERROR_AT = 0;
  .. CODE FOR STEP 1 ...

  ERROR_AT = 1;
  ... CODE ...

  ERROR_AT = 2;
  ... CODE FOR STEP 2 ...

  /* end */

  WHEN ANY DO
  BEGIN
    SUSPEND;
  END
END #
SET TERM ;#
```

ANY exception will jump to the **WHEN ANY** block:
➔ locate the problem, investigate it, remove it.

HELP! ALTERING PROCEDURES THAT DO HAVE DEPENDENCIES

- **ATTENTION** when altering procedures that are used by other procedures or triggers!

  ALWAYS obey these rules:
  - When altering the **Inputparams**:
    Either DROP or ALTER any procedure that is using it.
  - When altering the **Returnparams**:
    again DROP or ALTER any procedure that is using it AND
    - the procedure is invoked as an executable procedure and the return params are defined by **RETURNING_VALUES**
    - the procedure is invoked as an selectable procedure and SELECT * FROM ... INTO... is used.
- This way the database can't come into a state where metadata might become inconsistent.

HELP! STORED PROCEDURE WON'T RECOMPILE

- In case a stored procedure can't be altered or dropped, a gfix or backup and restore won't help (or work) nor firebird.support will be of any quick help:
  - Delete the procedure-source from the database!

SYSTEM TABLE: RDB$PROCEDURES:

```
CREATE TABLE RDB$PROCEDURES (
  RDB$PROCEDURE_NAME                  CHAR (31),   ← Procedurename
  RDB$PROCEDURE_ID                    SMALLINT,    ← Unique ID
  RDB$PROCEDURE_INPUTS                SMALLINT,    ← Count of Input-Parameters
  RDB$PROCEDURE_OUTPUTS               SMALLINT,    ← Count of Output-Parameters
  RDB$DESCRIPTION                     BLOB,        ← Description (use as you like)
  RDB$PROCEDURE_SOURCE                BLOB,        ← Source (just for the curious!)
  RDB$PROCEDURE_BLR                   BLOB,        ← compiled Code in BLR
  RDB$SECURITY_CLASS                  CHAR (31),
  RDB$OWNER_NAME                      CHAR (31),   ← Creator of this procedure
  RDB$RUNTIME                         BLOB,
  RDB$SYSTEM_FLAG                     SMALLINT     ← 1 = SYSTEM GENERATED
);
```

```
UPDATE RDB$PROCEDURES SET
  RDB$PROCEDURE_BLR = NULL
WHERE RDB$PROCEDURE_NAME = <PROCEDURE_NAME>
```

ALMOST THE END

Using **STORED PROCEDURES** will help by

- increasing speed
- Solving complex tasks by *easy* means
- introducing / maintaining / extending the Business Logic.

- Usually the (database-) Server is the better place for operations to be done in a classic C/S-environment:
  Deploying a job to the server will also decrease network traffic.

- Extending the possibilities of standard SQL.

DON'T BE AFRAID OF – YOU'LL LOVE THEM!

FINALLY

ANY MORE QUESTIONS ???

... and later on ...

Contact:

Lucas Franzen
c/o Frred Software GmbH
Wilhelmstr. 24a
D-79098 Freiburg
Germany

Tel.: +0049 (0)761 / 76 777 95

Email lucas.franzen@frred.de
Internet www.frred.de

FirebirdSQL
Foundation
FULL MEMBER
Supporting
Firebird development

and don't forget to **JOIN**!