Introduction to Procedural Extensions of SQL

Stored Routines in Transactional Context

Martti Laiho, Fritz Laux and Ilia Petrov

Parts of application logic can be stored at database server-side as “stored routines”, that is as stored procedures, user-defined-functions, and triggers. The client-side application logic makes use of these routines in the context of database transactions, except that the application does not have control over triggers, but triggers control the data access actions of the application. The tutorial introduces the procedural extension of the SQL language, the concepts of stored routines in SQL/PSM of ISO SQL standard, and the implementations in some mainstream DBMS products available for hands-on experimentations in the free virtual database laboratory of DBTechNet.org.

© Martti Laiho 2014
Introduction to Procedural Extensions of SQL

- in Transactional Context

This is a companion tutorial to the “SQL Transactions – Theory and hands-on labs” of DBTech VET project available at www.dbtechnet.org/download/SQL-Transactions_handbook_EN.pdf . For more information or corrections please email to martti.laiho(at)gmail.com. You can contact us also via the LinkedIn group DBTechNet.

Disclaimer

Don’t believe what you read 😊. This tutorial may provide you some valuable technology details, but you need to verify everything by yourself! Things may change in every new software version. The writers don’t accept any responsibilities on damages resulting from this material. The trademarks of mentioned products belong to the trademark owners. Permission is granted to copy this material as long as the original authors are mentioned.

Acknowledgements

This material is not funded by EU LLP programme, but without the EU LLP projects and the inspiring cooperation with the DBTechNet community and friends this tutorial would never have been written. Special thanks to Ken North on raising the topics to us, Peter Gulutzan and Trudy Pelzer for their exciting books and presentations at the SQL 2012 workshop in Helia, to Riitta Blomster, Timo Leppänen, and to Dr. Malcolm Crowe for their contributions to this paper.

Preface

The SQL Transactions handbook was aimed for basic training on database transactions at the vocational education level and for self-study by application developers. The book was limited to Client/Server data access using SQL only, and more advanced topics were left out. The purpose of this tutorial is to fill the gap and present these topics for interested learners. With database transactions in mind, we move to the server-side topics of stored procedures, user-defined functions, and triggers extending the built-in constraints of SQL. We look at what ISO SQL says, and experiment with the SQL dialects of free DBMS products which we have in our free virtual database laboratory available from www.dbtechnet.org/download/DebianDBVM06.zip.

The appendices extend the introduction to even more advanced topics and proceed to client-side application development calling the stored procedures from Java/JDBC. Our examples are minimalistic, and the coverage of the topics is only on introductory level. However, we hope that learners get at least an overview of these exciting topics and get interested in more information on these rich topics. The learners are encouraged to get the product manuals available on the websites of the product vendors and experiment with their products. All examples have been tested in our virtual database lab, but don’t trust on what you read – verify things yourself.

© Martti Laiho 2014
## Contents

**Part 1** ISO SQL/PSM ................................................................................................................................. 1  
1.1 Stored Procedures ................................................................................................................................. 2  
  SQL/PSM procedure language .................................................................................................................. 6  
  Exceptions and Condition handlers ........................................................................................................ 7  
  Benefits of stored procedures ................................................................................................................. 10  
1.2 Stored Functions ................................................................................................................................... 11  
1.3 Triggers ................................................................................................................................................ 14  

**Part 2** Implementations of Stored Routines ............................................................................................... 18  
2.1 DB2 SQL PL .......................................................................................................................................... 20  
  Simple myProc and myFun tests ............................................................................................................. 22  
  Exception handling and BankTransfer examples .................................................................................... 25  
  Triggers .................................................................................................................................................... 27  
  RVV example on optimistic locking ........................................................................................................ 27  
2.2 Oracle PL/SQL ....................................................................................................................................... 28  
  Simple myProc and myFun tests ............................................................................................................. 30  
  Exception handling and BankTransfer examples .................................................................................... 33  
  On Oracle Triggers .................................................................................................................................. 36  
  RVV example on optimistic locking ........................................................................................................ 36  
2.3 SQL Server Transact-SQL ..................................................................................................................... 38  
  Simple myProc and myFun tests ............................................................................................................. 38  
  Exception handling and BankTransfer examples .................................................................................... 40  
  Triggers and RVV example on optimistic locking .................................................................................. 41  
2.4 MySQL/InnoDB ...................................................................................................................................... 42  
  Simple myProc and myFun tests ............................................................................................................. 42  
  Trigger workaround for the missing support of CHECK constraints: ..................................................... 43  
  Exception handling and BankTransfer examples .................................................................................... 44  
  Triggers .................................................................................................................................................... 45  
  RVV example on optimistic locking ........................................................................................................ 45  
2.5 PostgreSQL PL/pgSQL .......................................................................................................................... 46  
  Simple myProc and myFun tests ............................................................................................................. 46  
  Exception handling and BankTransfer examples .................................................................................... 47  
  Triggers and RVV example on optimistic locking .................................................................................. 48  
2.6 Pyrrho DBMS ......................................................................................................................................... 48
Simple myProc and myFun tests ................................................................. 49
Exception handling and BankTransfer examples ........................................ 50
Triggers and RVV example on optimistic locking ........................................ 51

Part 3 Concurrency Problems due to Side-effects ........................................ 53

3.1 Side-effects due to Foreign Keys ............................................................ 53
SQL Server Transact-SQL ............................................................................. 54
DB2 SQL PL .................................................................................................. 55
Oracle PL/SQL ............................................................................................... 56
MySQL/InnoDB ............................................................................................ 57
PostgreSQL PL/pgSQL .................................................................................. 59
PyrrhoDB 5.2 ................................................................................................ 60

3.2 Concurrency Side-effects due to Triggers .............................................. 61
DB2 SQL PL .................................................................................................. 62
Oracle PL/SQL ............................................................................................... 64
SQL Server Transact-SQL ............................................................................. 65
MySQL/InnoDB ............................................................................................ 66
PostgreSQL PL/pgSQL .................................................................................. 67
PyrrhoDB (to be tested) ................................................................................ 68

3.3 Referential Integrity Rules and Triggers ................................................ 69

Part 4 Comparing SQL/PSM and Implementations ...................................... 79

References .................................................................................................. 82

Appendix 1 Result Set Processing by Cursor Mechanism ............................. 83
Appendix 2 Nested Transactions and Transactions with Savepoints ............. 89
Appendix 3 Autonomous Transactions ......................................................... 95
Appendix 4 Calling Stored Procedures from Java Code ................................. 97
Appendix 5 External Routines ...................................................................... 108
Index ........................................................................................................... 109
Comments of Readers .................................................................................. 111
Part 1  ISO SQL/PSM

The SQL language is based on the ideal of relational model and relational algebra, processing of sets. It can be seen as nonprocedural in nature. With SQL we basically specify what data result we expect, but not how this has to be done. It provides productive and powerful means for designing database solutions and accessing data from applications in most cases. However, some data processing tasks needed in applications would need “acrobatic”, not-that-productive SQL programming, and for practical purposes procedural extensions to the language have been implemented by most vendors, early before the definition in the SQL standard in 1996 of Persistent Stored Modules SQL/PSM (Melton 1998). One of the first procedural SQL extensions was Oracle’s PL/SQL used for scripting, stored procedures, functions, triggers, and which has had influence in the SQL/PSM standard. In the standard the modules consist of stored routines defined as procedures and functions. In the SQL:1999 standard also triggers were added to the SQL/PSM part of the standard. The extended language used in these routines is generally called stored procedure language (SPL or SQL PL). Triggers were already part of the experimental System R’s SEQUEL language (Astrahan et al 1976), but were left out from the early RDBMS products, and also from the early SQL standard.

Figure 1 presents SQL routines stored in the database. Stored routines allow parts of application logic to be stored in the database for security and performance reasons. The same routines can be available for multi-use, in different transactions or by multiple programs. Stored functions, called user defined functions (UDF), can be created and used to extend the implemented SQL language. According to SQL
standard, special UDFs can also be created as methods of user defined type (UDT) objects, and these are implemented for example in DB2. The methods technology is out of the scope of this tutorial.

Triggers supplement the SQL constraints in enforcing data integrity and implementing business rules (North 1999).

The application logic is responsible for controlling all transactions, even if the whole transaction with its retry logic is implemented in a stored procedure, and a best practice is that the transaction is started and ended in the same module on the client-side. However, the application logic on the client-side needs always be informed and prepared to manage the cases when the transaction is rolled back automatically by the database server during the execution of server-side routines. Since stored functions can only be invoked as part of some SQL statement, common sense says that these should not contain any transaction demarcation. Same applies to triggers, since the application logic is not aware of these, and can only notice some exceptions raised by the triggers.

What was said above may not be true for every DBMS product as, for example, in PostgreSQL stored procedures and functions are the same concepts. In Appendix 3 we will learn about autonomous transactions in stored procedures, which can be also invoked from functions or triggers, typically for tracing purposes, and independent of the current transaction initiated by the application.

Before the extended SQL language implementations of SQL/PSM, the routine bodies used to be implemented registering the routine’s action part from external function libraries coded in 3GL languages, such as COBOL, PL/I, C/C++, or even Java, and accessing the host database using embedded SQL (ESQL), Call Level Interfaces (CLI), or JDBC.

Even if SQL/PSM standard was a late-comer, it has now been implemented for example in DB2, Mimer, MySQL, Pyrrho, and optionally in PostgreSQL. Its predecessor PL/SQL still dominates the market as the native procedural language of Oracle, and as an optional PL SQL for PostgreSQL and DB2. Another mainstream procedural language is Transact SQL (T-SQL) of Microsoft SQL Server, but this is not available in our DBTechLab.

### 1.1 Stored Procedures

SQL stored procedures increase productivity and manageability in application development by providing means for

a) storing parts of application logic, encapsulated in the database, and to be used by multiple programs and multiple programmers, allowing also logic maintenance centrally in a single place. Stored routines should be written by experienced developers because they may be used in different transactions which make it difficult to develop and test.

b) security as the used data access privileges are needed only from the professional creators (with create procedure privilege) of the procedures, and only execution privilege need to be granted per procedure to proper user groups. Also SQL injection possibilities are eliminated on part of stored procedures.

c) performance benefits in minimizing package preparation (parsing and binding) work, and reducing network traffic from remote clients, but also decreasing context switching, when several SQL statements can be included in a package.
d) productivity in data access programming by extending the SQL language with procedural flow of control and sub-programming techniques familiar from the traditional block structured programming languages.

Using parameters, data of SQL data types can be passed in and out from the procedure’s “routine body”. The routine may contain simply deterministic data processing, such as calculations, or may include accessing of the database data contents. The routine body can be written also in various programming languages, but in this tutorial we are interested only in the extended SQL. A stored procedure can be called directly as subprogram from the application programming languages or from SQL code, other stored procedures, functions or triggers.

The following simple example applies the MySQL implementation of SQL/PSM and presents the basic idea of stored procedures

```
DELIMITER !
CREATE PROCEDURE balanceCalc ( IN interestRate INT,
                               INOUT balance INT,
                               OUT interest INT)
DETERMINISTIC
BEGIN
  SET interest = interestRate * balance / 100;
  SET balance = balance + interest;
END !
DELIMITER ;
```

The procedure is created by the CREATE PROCEDURE command which contains SQL statements ending with semicolons (;), so for the temporary delimiter of the whole command we have chosen exclamation mark (!). The procedure can be called using local MySQL variables as follows

```
mysql> SET @balance = 10000;
Query OK, 0 rows affected (0.00 sec)
mysql> SET @interest = 0;
Query OK, 0 rows affected (0.00 sec)
mysql> CALL balanceCalc (10, @balance, @interest);
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @balance, @interest;
+----------+----------+
| @balance | @interest |
|----------+----------|
| 11000    | 1000     |
+----------+----------+
1 row in set (0.00 sec)
```

Just like in programming languages, IN parameters are used to pass values to the procedure and OUT parameters return values to the invoking (calling) application. INOUT parameters serve passing values in both directions.

The following procedure provides a simple example of accessing the contents in the database

```
CREATE PROCEDURE getBalance (IN acctNo INT, OUT o_balance INT)
LANGUAGE SQL
SELECT balance INTO o_balance FROM Accounts
WHERE acctID = acctNo ;
```
An overview of SQL/PSM syntax for stored procedures is presented in Figure 2.

```
CREATE PROCEDURE [[<catalog>.] <schema>.] <procedure name> 
( [[<parameter mode>.] <parameter name> <data type> [, ...] ] ) 
[ < routine characteristic> [, …] ] 
{ < SQL routine body>
| EXTERNAL NAME
| {<name> | <char string literal> } 
| PARAMETER STYLE
| <parameter style> ] 
| EXTERNAL SECURITY
| { DEFINER
| | INVOKER
| | IMPLEMENTATION DEFINED
| }
| }
}
```

Figure 2. ISO SQL syntax overview of CREATE PROCEDURE
(Melton 2003)

The rich details presented in the syntax are beyond the scope of this tutorial, and for more details we refer to SQL/PSM or the books by Melton and by Gulutzan & Pelzer. The listed programming languages refer to external routines written in those languages, to be registered in system tables in the database, for invocation by the server when called by the application. Information about the actual programming language is needed for proper mapping of the parameter values which depend on the language. The external routines as well as the .NET CLR classes stored as routines in databases are out of scope of this tutorial, since we focus on SQL only.

According to the standard, stored routines are polymorphic, which means that there can be multiple routines with the same name distinguished only by the parameter list. To simplify the management of such routines a unique SPECIFIC name should be given to a routine, for example for dropping the specific routine.

The routine characteristic DETERMINISTIC means that for given input parameter values the routine always generates the same output or result, independently of the server environment or contents of the database. Usually this applies to user defined functions, which extend the SQL language, such as calculations or transformations.

NO SQL declares that the procedure cannot contain any executable SQL statements.

CONTAINS SQL declares that the procedure cannot contain any SQL DML statements (INSERT, UPDATE, DELETE, SELECT) nor COMMIT, but may contain statements like CALL, ROLLBACK, etc.

READS SQL DATA adds SELECT and cursor programming statements to statements allowed by CONTAINS SQL.
MODIFIES SQL DATA adds all DML statements, CREATE, ALTER, DROP, GRANT, REVOKE, and SAVEPOINT programming to those allowed by READS SQL DATA.

For accessing database objects, the creator of the stored routine has to have all the needed privileges, whereas users of the routine just need to have EXECUTE privilege to the routine granted by the creator. Some products support also the security alternative that for an execution of the routine, the invoker need to have all the privileges needed to accessing the database objects, independently of the privileges of the routine creator.

In addition to passing results via parameters like 3GL programming languages, the stored procedures may pass results as a sequence of result sets to the calling application. In the application the result sets are processed by cursors if the programming language cannot handle sets. The “DYNAMIC RESULT SETS <n>” clause specifies the maximum number n of the result sets to be returned by the procedure.

**Result Set Cursors**

The following minimalistic example executed in DB2 demonstrates the use of DYNAMIC RESULT SETS passed to the caller by result set cursor (Melton & Simon, 2002, p 462), a cursor defined as “WITH RETURN”. For our example we create the following simple test table

```sql
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(10), i SMALLINT);
INSERT INTO T (id, s, i) VALUES (1, 'first', 1);
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'fourth', 2);
COMMIT;
```

and we create the procedure

```sql
CREATE PROCEDURE mySet (IN ind INT)
DYNAMIC RESULT SETS 1
BEGIN
DECLARE myCur CURSOR WITH RETURN
  FOR SELECT id, s FROM T WHERE i = ind;
OPEN myCur;
END @
COMMIT @
```

For execution of the CREATE command we have temporarily defined the at-sign (@) as command terminator to allow semicolons as statement terminators in the procedure body. This can be done with the –td@ command line option when starting the db2 command line processor program CLP.

While calling the procedure from a SQL client, such as DB2 CLP utility, the virtual table opened by the cursor appears as result set like for a SELECT command, as in the following tests:

```sql
db2 => call mySet(2)
Result set 1
----------
ID       S
----------
2        second
4        fourth
2 record(s) selected.
```

© Martti Laiho 2014
db2 => call mySet(1)

Result set 1
--------------
ID          S
----------- -------
1 first     third
2 record(s) selected.

The SQL client program processes the result set using a cursor ‘behind the scene’. If the caller is a 3GL application program, it must use an explicit cursor or a result set object which wraps the cursor processing.

**SQL/PSM procedure language**

The SQL procedure language allows programming stored routines which contain SQL statements with elements known from traditional block structured 3GL programming languages including:

- Polymorphism with SQL parameters of IN, OUT, INOUT modes
- Procedure Call statements
- Function invocations and RETURN statement
- BEGIN [ATOMIC] – END blocks
- Declared local SQL variables (of SQL data types)
- Assignment statements of the form “SET variable = expression”
- Conditional control structures IF and CASE
- Labelling of statements with ITERATE and LEAVE
- Looping structures LOOP, WHILE, UNTIL and FOR
- SQL cursor processing
- Error signalling and exception handling

(Melton 1998).

SQL-clients can call a procedure passing IN or INOUT parameters to the procedure and the procedure can return results by OUT or INOUT parameters. Beside returning a single result via parameters, a procedure can return one or more result sets to the caller, which should then handle these using SQL cursor processing.

The procedure body can be a single SQL statement, but usually consists of a compound statement, a BEGIN – END block, and inside the SQL block we can have nested SQL blocks. An ATOMIC compound statement means that if a statement in the series of statements fails, then the whole block will be rolled back. In a sense the beginning of ATOMIC block is an implicit savepoint\(^1\), and an error generates implicit rollback to that savepoint and exiting from the block.

Inside a compound statement local SQL variables can be declared based on SQL data types. The variables can have default values, but variables are not transactional, which means that possible ROLLBACKs have no effect on variable values (Melton 1998). The variable is visible in the block where it was declared, and in its nested sub-blocks. The compound statements can be labelled, and in case of nested blocks the

[\(^1\) We will explain the savepoint concept in Appendix 2]
label can be used to qualify the variables. The label name of the compound statement can also be used for jumping out from the block, as ITERATE <label> returns to the beginning of the labelled block whereas LEAVE <label> will exit immediately from the labelled block.

Procedures are like 3GL subprograms and encapsulate application logic in database. The SQL control structures IF – END IF, CASE – END CASE, WHILE – END WHILE, REPEAT – UNTIL – END REPEAT, based on logical condition evaluations of SQL, are easy to learn if you are familiar with programming languages, but control structures LOOP – END LOOP and FOR – END FOR are not that intuitive. LOOP control structure can be explained by the following syntax definition

\[
\begin{align*}
&[<\text{label}> : ] \text{ LOOP} \\
&\quad <\text{SQL statement}> [, .. ] \\
&\text{END LOOP}[<\text{label}>]
\end{align*}
\]

where the optional label names in the beginning and at the end need to be identical. The included list of SQL statements is executed sequentially over and over again until at some step of the execution the loop will be left by an explicit LEAVE <label> statement or the execution will be stopped by some implicit exception condition or an exception generated by an explicit SIGNAL statement.

The FOR – END FOR structure is a very SQL specific control structure, which simplifies the use of SQL cursor processing. An SQL cursor is a language mechanism to solve the so called “impedance mismatch” between the set oriented SQL language and a record-at-a-time processing by typical programming languages. The basic idea in cursor processing is that the rows of a query result set can be fetched for application processing sequentially one row at a time until the end of the result set is reached. For more details of cursor processing see Appendix 1. The FOR control structure can be explained by the following syntax diagram

\[
\begin{align*}
&[<\text{label}> : ] \text{ FOR <loop name> AS} \\
&\quad [<\text{cursor name}] \ [\text{INSENSITIVE} \ \text{CURSOR FOR}] \\
&\quad <\text{SELECT statement}> \\
&\quad \text{DO} \ <\text{SQL statement}> [, .. ] \\
&\text{END FOR}[<\text{label}>]
\end{align*}
\]

The required <loop name> need to be a unique identifier in the whole FOR statement. It can be used for cursor processing as qualifier of the column names in the result set. The cursor opening step and building of the result set is done implicitly. INSENSITIVE option defines that the result set will be a snapshot of the cursor opening step. Fetching the rows from the result set is done implicitly, and every fetched row is processed by the SQL statement list (a compound statement) of the DO clause. At the end of the result set the cursor is closed implicitly and control returns from the FOR structure.

**Exceptions and Condition handlers**

Every SQL statement is atomic in the sense that if it fails all effects of the whole statement is rolled back from the database. As we discussed in the SQL Transactions Handbook, usually this does not mean that the whole current SQL transaction would be rolled back. The basic diagnostic information of the success, failure, or warning after every statement can be sorted out from the SQLCODE\(^2\) integer values or the five

\(^2\) Only values 0 for successful execution and 100 for NOT FOUND have been standardized for SQLCODE, and it has been announced as deprecated in the standard. However, the products continue to support it, since the product
character SQLSTATE indicator as defined in SQL-92 standard. The value of SQLSTATE consists of 2 character condition class followed by 3 character subclass code for more precise information. The major SQLSTATE classes consist of “00” for successful execution, “01” for success with SQLWARNING, “02” for success of SELECT but with NOT FOUND results, and all other classes indicating some sort of failed execution (SQLEXCEPTION).

SQL-92 defined also more detailed diagnostic information available using GET DIAGNOSTICS statements. For more information on these, please refer to our “SQL Transactions” handbook. Frequent use of GET DIAGNOSTICS after execution of every step in a compound statement complicates the procedural SQL code, and to rationalize the exception handling in stored routines the concept of condition handlers has been introduced in the SQL/PSM standard. The idea is not new, earlier implementation was the WHENEVER “error trapping” directive defined in SQL-89 standard for embedded SQL. However, the new condition handlers are much more powerful providing also a SQL statement as handler action (which can also be a compound statement block) for handling the condition, such as generic SQL exceptions, specified SQLSTATE values, warnings or application defined conditions, declared as follows

```
DECLARE { CONTINUE | EXIT | UNDO }
HANDLER FOR
{ SQLSTATE <value>
 |<condition name>
 | SQLEXCEPTION
 | SQLWARNING
 | NOT FOUND }
<SQL statement> ;
```

In addition to the system defined conditions SQLEXCEPTION, SQLWARNING, and NOT FOUND, the application can declare named conditions for certain SQLSTATE values as follows

```
DECLARE <condition name> CONDITION
[ FOR SQLSTATE <value> ] ;
```

or mere condition names, which the application can activate using a SIGNAL statement as follows

```
SIGNAL <condition name> ;
```

The application can also raise conditions with SQLSTATE values without the actual SQL exception, just by signaling it as follows

```
SIGNAL SQLSTATE <value> ;
```

In case of handler type CONTINUE, the diagnostics are captured for the handler action, and after the handler action terminates the condition is considered as handled.
In case of handler type EXIT the diagnostics are captured for the handler, and after the handler action the condition is considered as handled, and the routine is exited.
In case of handler type UNDO all statements of the compound statement including triggered actions will be rolled back, the handler action is executed and the condition is considered as handled.
If the handler action raises an exception, then the condition will be implicitly re-signalled.

dependent code values give more accurate diagnostics (Chong et al, 2010), and for example Oracle does not even support SQLSTATE directly in PL/SQL.

© Martti Laiho 2014
The following example demonstrates the use of SQL/PSM condition handlers in a stored procedure implementing the BankTransfer transaction with which we have been experimenting in the SQL Transactions handbook examples using the following table:

```
CREATE TABLE Accounts (
    acctID INTEGER NOT NULL PRIMARY KEY,
    balance INTEGER NOT NULL,
    CONSTRAINT unloanable_account CHECK (balance >= 0)
);
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;
```

And our BankTransfer procedure with condition handlers is the following:

```
CREATE PROCEDURE BankTransfer (IN fromAcct INT,
    IN toAcct   INT,
    IN amount   INT,
    OUT msg     VARCHAR(100))
LANGUAGE SQL MODIFIES SQL DATA
P1: BEGIN
    DECLARE acct INT;
    DECLARE EXIT HANDLER FOR NOT FOUND
    BEGIN ROLLBACK;
    SET msg = CONCAT('missing account ', CAST(acct AS VARCHAR(10)));
    END;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN ROLLBACK;
    SET msg = CONCAT('negative balance (?) in ', fromAcct);
    END;
    SET acct = fromAcct;
    SELECT acctID INTO acct FROM Accounts WHERE acctID = fromAcct;
    UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
    SET acct = toAcct;
    SELECT acctID INTO acct FROM Accounts WHERE acctID = toAcct;
    UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;
    COMMIT;
    SET msg = 'committed';
END P1
```
Benefits of stored procedures

Procedures are written for improved performance, security, and centralized maintainability of parts of the application logic as listed in the beginning of this chapter.

When multiple SQL statements are encapsulated in a procedure, the network traffic i.e. “round trips” between the client and server are reduced. In addition, only the final result of the procedure will be transmitted to the client. Intermediate data is processed on the server. This reduces network traffic, too.

Performance gets improved also since the SQL statements are parsed and optimized when the procedure is created, and the execution plans are packaged, stored in the database and cached for repeated use. The optimized machine code runs much faster than interpreted SQL-code.

Security is improved, since procedures should be created only by competent developers (definers) who are granted privileges to create procedures and who need to have all the privileges that are required for the embedded SQL statements, and users (invokers) who have execution privilege for the procedure do not need to have all those privileges that the developer needs.

The external stored routines are written in various programming languages, compiled and the executable code is registered in system tables by the CREATE command referring to the code library. This means that procedures can be delivered without the source code. If the routines were executed directly by the server process, the whole server might be vulnerable for programming errors. Therefore, in DB2, to protect the server against aborting in these cases, the external routines are run by default FENCED. This means, the routines are run in separate son processes of the server.

Improved maintainability means that procedures can be invoked by multiple programs, and if the procedure logic needs to be updated, it can be done “on the fly” as an atomic operation.

Stored routines ensure data access consistency and maintainability, since when some objects to be accessed are changed or deleted the execution plan of the routine is invalidated automatically.

As a benefit of stored routines, the DB2 manual “Developing User-defined Routines” mentions “interoperability of logic implemented in different programming languages”. Data access APIs are available for calling SQL stored procedures from various programming languages. The SQL interface (signature) of stored routines used for accessing the external stored routines, perhaps written in different programming languages, such as Java, even extends the interoperability.

Challenges for stored procedures include their involvement and synchronization in transactions, complicated exception handling, and need for extended documentation.

Implementing whole database transactions as stored procedures requires and promotes a strict transaction programming discipline. Developing stored procedures in IDE workbenches, such as IBM Data Studio or Oracle SQL Developer, before deployment into database provide means for code debugging and Unit Testing of transactions.

3 Possibly re-optimized for new parameter values.
1.2 Stored Functions

In addition to the aggregate, arithmetic, temporal and string functions defined in SQL standard, SQL dialects of DBMS products contain various built-in functions. SQL/PSM standard and the dialect implementations provide also means for extending the language by user-defined functions (UDF) also called Stored Functions, which can be general-purpose or application dependent. Stored functions are usually invoked from SQL statements, but that depends on the product.

Figure 3 presents the ISO SQL syntax overview of CREATE FUNCTION. With stored functions developers can extend and implement customized features in the available SQL language.

```
CREATE FUNCTION [[<catalog>.] <schema>.] <function name>
  ( [ [parameter mode] <parameter name> <data type> [RESULT] [, ... ] ] )
  RETURNS <data type> [ CAST FROM <data type> | AS <LOCATOR> ]
  [ < routine characteristic>
    [, ... ] ]
  [ STATIC DISPATCH ]
  [ < SQL routine body>
    | EXTERNAL NAME
      {<name> | <char string literal> }
    | PARAMETER STYLE
      <parameter style>
    | EXTERNAL SECURITY
      { DEFINER
        | INVOKER
        | IMPLEMENTATION DEFINED
      }
  ]
)
```

Figure 3. ISO SQL syntax overview of CREATE FUNCTION (according to Melton 2003)

According to ISO SQL standard stored functions allow only IN mode parameters, although some products allow also INOUT and OUT mode parameters. Compared with the stored procedures, a stored function returns only one SQL value (which in some implementations may also be a result set in form of table or opened cursor) the type of which is defined by the RETURNS clause. The result can be a scalar SQL data type value, a virtual table, or a row of SQL data type columns. Only one parameter can have the RESULT keyword, in which case the parameter has to be of the same user-defined data type as defined in RETURNS clause for the result of the function.

RETURNS NULL ON NULL INPUT clause defines that if NULL is passed to the function in some parameter, then without wasting resources in processing the function immediately returns NULL value. The opposite clause CALL ON NULL INPUT defines that in spite of NULL values in parameters, the function will be processed.

© Martti Laiho 2014
Figure 3 lists the languages which could be used to implement the SQL routine body. We will focus on the scalar type functions, written in SQL language, but we will also discuss table-valued functions. In this paper we will not cover the externally stored functions.

Since every SQL statement is atomic and functions are invoked from SQL statements, also functions need to be atomic, that is the SQL routine body can contain only a single SQL statement, or the routine body consists of an atomic compound statement of form BEGIN ATOMIC .. END.

Some implementations don’t allow maintenance of database contents or transaction controlling by the stored functions. An interesting implementation of stored routines is in PostgreSQL in which stored procedure and function are the same.

An example of a DETERMINISTIC general-purpose function is the following function which calculates factorial value $n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$ for the given integer $n$

```
CREATE FUNCTION factorial(n INT)
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE f INT DEFAULT 1;
    WHILE n > 0 DO
        SET f = n * f;
        SET n = n - 1;
    END WHILE;
    RETURN f;
END
```

which can be invoked in MySQL implementation as follows

```
mysql> SELECT factorial(3);
+----------------+
<table>
<thead>
<tr>
<th>factorial(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
</tbody>
</table>
+----------------+
1 row in set (0.00 sec)

mysql> SELECT factorial(0);
+----------------+
<table>
<thead>
<tr>
<th>factorial(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
+----------------+
1 row in set (0.00 sec)
```

Note: In DB2 Express-C the factorial $13!$ and above 13 will generate arithmetic overflow and in MySQL version 5.6 erroneous values.

Note: Factorial function can also be defined as recursive form, but for example MySQL does not accept recursive stored functions.
Just like for a stored procedure, the creator of a stored function needs to have the privileges for all data access done in the function, but all other users need only EXECUTE privilege to the function for using the function.

In the chapter “Implementations ..” we will present variants of our minimalistic scalar stored function “myFun” which accesses database contents using SQL statements and which we have tuned for most of the DBMS products in our database laboratory.

Table-valued Functions

Functions returning a virtual table are called table-valued functions. The following minimalistic example tested with DB2 Express-C and using the same table as our stored procedure example on result set cursors presents the basic idea of table-valued function. The following example shows a function that returns a parameterized SQL view:

```
CREATE FUNCTION myView (ind INT)
RETURNS TABLE (id  INT,
                str VARCHAR(10)
            )
RETURN SELECT id, s FROM T WHERE i = ind
```

More powerful processing could be built into table-valued function in which the routine body consists of atomic compound statement like in the following, which would allow also many more statements

```
CREATE FUNCTION myView (ind INT)
RETURNS TABLE (id  INT,
                str VARCHAR(10)
            )
BEGIN ATOMIC
RETURN
SELECT id, s FROM T WHERE i = ind ;
END @
```

The function can be used where a SQL view could be used, but only as argument of SQL TABLE function which need to be made available for the statement by alias name (Chong et al 2010), for which we use name V in the following:

```
db2 => SELECT V.* FROM TABLE(myView(1)) AS V
```

<table>
<thead>
<tr>
<th>ID</th>
<th>STR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>first</td>
</tr>
<tr>
<td>3</td>
<td>third</td>
</tr>
</tbody>
</table>

2 record(s) selected.

Above we have demonstrated only the basic technique, but the table-valued functions can provide much more powerful capabilities for server-side programming.
1.3 Triggers

Triggers cannot be called by the application, but these are “fired” i.e. activated by certain type of SQL statements modifying the table which the trigger is controlling. Triggers can be seen as active agents in the database, used for business rule control and as extended constraints. Triggers can monitor the INSERT, UPDATE or DELETE events (<trigger event>) affecting a certain table.

```
CREATE TRIGGER <trigger name>
{BEFORE | AFTER | INSTEAD OF }<trigger event> ON <table name>
[REFERENCING OLD AS <old alias> ]
[REFERENCING NEW AS <new alias> ]
<triggered action>

Where
<trigger event> ::= INSERT | DELETE | UPDATE [OF <column list>]
<triggered action> ::=  
[FOR EACH {ROW | STATEMENT} ] ←------ Action Granularity
[ WHEN ( <SEARCH CONDITION>) ] ←------ Action Condition
{<SQL statement> |  
   BEGIN ATOMIC 
      {<SQL statement>; }...
   END 
}
```

Figure 4. ISO SQL syntax for CREATE TRIGGER

Figure 4 presents the ISO SQL syntax overview of CREATE TRIGGER.

An UPDATE trigger can be set to be fired only in case of changes in some of the listed columns only.

Note that SELECT is not a triggering event!

A BEFORE trigger takes action before the actual event action and AFTER trigger after the event action, whereas INSTEAD OF trigger prevents the action of the firing statement and replacing it by the triggered action. INSTEAD OF triggers have been invented in DBMS products earlier and included in the ISO SQL standard just recently.

FOR EACH clause defines the action granularity i.e. if the trigger will be activated for each selected row (row-level trigger) or only on statement-level before or after the trigger event. Depending on the action granularity the REFERENCING clauses refer to old and new copies of the row in the triggered action or the whole table. Figure 5 presents what the old and new copies of the row are in cases of row-level triggers depending on the trigger event. So the trigger can see versions of the same columns and do some operations or decisions based on the value changes.

The optional WHEN clause can be used to decide if the row changes are of interest for the trigger.

The action body of a trigger may consist of a single statement, for example a procedure call, or an atomic compound statement, which either succeeds or will be cancelled as a whole.
Figure 5 demonstrates the visibility of the old and new version of the row for BEFORE and AFTER triggers in case of row-level INSERT, UPDATE and DELETE events.

Figure 6 demonstrates some possible actions of a trigger. The action body (trigger body) consists of one SQL statement or an ATOMIC compound statement.
The trigger body of a row-level BEFORE trigger may make its own changes in the content of the NEW row on INSERT or UPDATE events. BEFORE triggers may not contain INSERT, UPDATE nor DELETE statements (Chong et al).

A BEFORE or AFTERUPDATE row-level trigger sees both OLD and NEW versions of the row and can compare old and new values of the same column.

A trigger may execute actions to some other tables, for example for

- tracing (logging) of events
- update some derived (denormalized) columns for better performance of some queries
- stamping the updated version of the row (see our RVV trigger examples)
- validating the data against rows in other tables or rows in the same table.

The trigger may find some unacceptable condition and can RAISE an error exception, which means that the SQL statement which activated the trigger will be rolled back.

The SQL:2003 standard did not define INSTEAD OF triggers (Kline et all 2009), but this has been implemented by most vendors, so we mention it here: An INSTEAD OF trigger catches the event statement against the table, then prevents the action of the SQL statement itself and instead of it executes the action defined in the trigger body.

A table can have multiple triggers, but only one trigger for the same event for BEFORE and one for AFTER on row-level and on statement level. The firing order of the triggers for the same event is following (Connolly & Begg 2010)

Statement-level BEFORE trigger
For every row in turn affected by the statement:
  Row-level BEFORE trigger for the row in turn
  <execution of the statement with its immediate constraints>
  Row-level AFTER trigger for the row in turn
Statement-level AFTER trigger.

The transaction context of a trigger is the transaction of the firing statement. This means that in principle the isolation level of the trigger cannot be changed (unless the implementation allows it). This may be a challenge, and may cause mysterious blocking.

Triggers don’t come for free. They add extra overhead in processing. Generally creation of triggers does not suit for novice developers, but should be the responsibility of database administrator group.

A trigger can be created only to the same database schema as the table to be controlled by the trigger and the creator needs to have TRIGGER privilege to the table. However, users accessing the table do not need any EXECUTE privilege to the trigger. Triggers can also be a security threat, if the TRIGGER privilege is granted to unreliable persons.

While writing to a table a trigger may fire some other triggers. This is called nesting of triggers. When a trigger’s action fires the same trigger, the trigger is called recursive. Ending the recursion can be challenging. Nesting and especially recursion of triggers may become a performance killer, and to protect against this some products offer an option to prevent cascading of triggers.
In the chapter “When Not to Use Triggers” Avi Silberschatz et all (2011) agrees that there are many good uses for triggers, but developers should first consider alternative available technologies, such as update/delete rules of foreign keys, materialized views, and modern replication facilities instead of overusing triggers, - and when used “Triggers should be written with great care”. Detecting trigger errors at runtime can be a really challenging task.

Connolly & Begg (2010) list as disadvantages of triggers the following
- complexity in database design, implementation and administration
- hidden functionality and unplanned side effects. This may cause difficulties to application developers and end-users.
- performance overhead, as discussed above.

Database administrators can enable or disable triggers. For example, when dealing with bulk loading data, triggers may need to be disabled by the administrator, and enabled afterwards.

Based on practical needs in applications trigger implementations in DBMS products have evolved to include various features, for example multi-event DML triggers, means of controlling cascading DML triggers and order of firing, DDL triggers, etc. Triggers can also be implemented for events on other objects than just tables, for example on logins or system/database-level. In this tutorial we will focus only on DML triggers.
Part 2  Implementations of Stored Routines

Since the SQL standard on SQL/PSM was published too late (in 1996), most vendors already had implemented procedural extension of their own, such as Oracle’s PL/SQL and the Transact SQL of Sybase and Microsoft. So the Implementations and syntaxes vary. Before SQL/PSM procedural language implementations, for example in DB2 the stored routines were based on external stored routines. The concept of external stored routines was adapted also in SQL/PSM as we have seen. Now both internal and externally stored routines can co-exist in a database, as illustrated below:

Only recent procedural language (PL) implementations are based on SQL/PSM, but may have some additional features, as can be seen in Table 2.1 which presents an overview of control structures of the implementations compared with the SQL/PSM. Some of the procedural language implementations are so rich that they deserve a book of their own, such as the book of Harrison and Feuerstein of MySQL’s implementation.

Table 2.1 Overview of control structures of PL implementations compared with the SQL/PSM

<table>
<thead>
<tr>
<th>Control statement</th>
<th>ANSI/ISO SQL</th>
<th>DB2</th>
<th>Oracle</th>
<th>SQL Server</th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>Pytho</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE WHEN</td>
<td>CASE WHEN</td>
<td>CASE WHEN</td>
<td>CASE WHEN</td>
<td>CASE WHEN</td>
<td>CASE WHEN</td>
<td>CASE WHEN</td>
<td>CASE WHEN</td>
</tr>
<tr>
<td>F. THEN</td>
<td>F. THEN</td>
<td>F. ELSE</td>
<td>F. THEN</td>
<td>F. THEN</td>
<td>F. THEN</td>
<td>F. THEN</td>
<td>F. THEN</td>
</tr>
<tr>
<td>DO UNTIL</td>
<td>DO UNTIL</td>
<td>DO UNTIL</td>
<td>DO UNTIL</td>
<td>DO UNTIL</td>
<td>DO UNTIL</td>
<td>DO UNTIL</td>
<td>DO UNTIL</td>
</tr>
<tr>
<td>ITERATE</td>
<td>ITERATE</td>
<td>ITERATE</td>
<td>ITERATE</td>
<td>ITERATE</td>
<td>ITERATE</td>
<td>ITERATE</td>
<td>ITERATE</td>
</tr>
<tr>
<td>LEAVE</td>
<td>LEAVE</td>
<td>LEAVE</td>
<td>LEAVE</td>
<td>LEAVE</td>
<td>LEAVE</td>
<td>LEAVE</td>
<td>LEAVE</td>
</tr>
<tr>
<td>BREAK</td>
<td>BREAK</td>
<td>BREAK</td>
<td>BREAK</td>
<td>BREAK</td>
<td>BREAK</td>
<td>BREAK</td>
<td>BREAK</td>
</tr>
<tr>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
<td>LOOP</td>
</tr>
<tr>
<td>LOOP END LOOP</td>
<td>LOOP END LOOP</td>
<td>LOOP END LOOP</td>
<td>LOOP END LOOP</td>
<td>LOOP END LOOP</td>
<td>LOOP END LOOP</td>
<td>LOOP END LOOP</td>
<td>LOOP END LOOP</td>
</tr>
<tr>
<td>WHILE</td>
<td>WHILE DO</td>
<td>WHILE LOOP</td>
<td>WHILE</td>
<td>WHILE LOOP</td>
<td>WHILE</td>
<td>WHILE LOOP</td>
<td>WHILE DO</td>
</tr>
<tr>
<td>DO UNTIL (Repetition)</td>
<td>DO UNTIL (Repetition)</td>
<td>DO UNTIL (Repetition)</td>
<td>DO UNTIL (Repetition)</td>
<td>DO UNTIL (Repetition)</td>
<td>DO UNTIL (Repetition)</td>
<td>DO UNTIL (Repetition)</td>
<td></td>
</tr>
<tr>
<td>FOR</td>
<td>FOR LOOP</td>
<td>FOR LOOP</td>
<td>FOR LOOP</td>
<td>FOR LOOP</td>
<td>FOR LOOP</td>
<td>FOR LOOP</td>
<td>FOR LOOP</td>
</tr>
<tr>
<td>OOTO</td>
<td>OOTO</td>
<td>OOTO</td>
<td>OOTO</td>
<td>OOTO</td>
<td>OOTO</td>
<td>OOTO</td>
<td>OOTO</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
</tr>
<tr>
<td>RARE APPLICATION ERROR</td>
<td>RARE ERROR</td>
<td>RARE ERROR</td>
<td>RARE ERROR</td>
<td>RARE ERROR</td>
<td>RARE ERROR</td>
<td>RARE ERROR</td>
<td>RARE ERROR</td>
</tr>
<tr>
<td>BASERERROR</td>
<td>BASERERROR</td>
<td>BASERERROR</td>
<td>BASERERROR</td>
<td>BASERERROR</td>
<td>BASERERROR</td>
<td>BASERERROR</td>
<td>BASERERROR</td>
</tr>
<tr>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
<td>SIGNAL</td>
</tr>
<tr>
<td>RARE EXCEPTION</td>
<td>RARE EXCEPTION</td>
<td>RARE EXCEPTION</td>
<td>RARE EXCEPTION</td>
<td>RARE EXCEPTION</td>
<td>RARE EXCEPTION</td>
<td>RARE EXCEPTION</td>
<td>RARE EXCEPTION</td>
</tr>
<tr>
<td>DECODE HANDLER</td>
<td>DECODE HANDLER</td>
<td>DECODE HANDLER</td>
<td>DECODE HANDLER</td>
<td>DECODE HANDLER</td>
<td>DECODE HANDLER</td>
<td>DECODE HANDLER</td>
<td>DECODE HANDLER</td>
</tr>
<tr>
<td>GET DIAGNOSTICS</td>
<td>GET DIAGNOSTICS</td>
<td>GET DIAGNOSTICS</td>
<td>GET DIAGNOSTICS</td>
<td>GET DIAGNOSTICS</td>
<td>GET DIAGNOSTICS</td>
<td>GET DIAGNOSTICS</td>
<td>GET DIAGNOSTICS</td>
</tr>
<tr>
<td>EXCEPTION WHEN</td>
<td>EXCEPTION WHEN</td>
<td>EXCEPTION WHEN</td>
<td>EXCEPTION WHEN</td>
<td>EXCEPTION WHEN</td>
<td>EXCEPTION WHEN</td>
<td>EXCEPTION WHEN</td>
<td>EXCEPTION WHEN</td>
</tr>
</tbody>
</table>

In the following we compare the procedural language implementations using only minimalistic examples, which we hope will help you to get started. Scripts for experimenting with all our examples, and more, can be found at [www.dbtechnet.org/download/StoredRoutines.zip](http://www.dbtechnet.org/download/StoredRoutines.zip). For some of the DBMS products in our DBTechLab (DebianDB) we present “Hello world” like examples of stored procedures and functions and basic models of calling them as follows.

© Martti Laiho 2014
First we create a tracing table

```
CREATE TABLE myTrace (  
t_no     INT,  
t_user   CHAR(20),  
t_date   DATE,  
t_time   TIME,  
t_proc   VARCHAR(16),  
t_what   VARCHAR(30)  
);  
INSERT INTO myTrace (t_no) VALUES (2);  
```

and then create the stored procedure myProc like following

```
CREATE PROCEDURE myProc (IN p_no INT, IN p_in VARCHAR(30), OUT p_out VARCHAR(30))  
LANGUAGE SQL  
BEGIN  
    SET p_out = p_in;  
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)  
    VALUES (p_no, user, current date, current time, 'myProc', p_in);  
    IF (p_no = 1) THEN COMMIT; ELSE ROLLBACK; END IF;  
END;  
```  

to be called as follows

```
CALL myProc (1, 'Hello ISO SQL/PSM', ?);  
SELECT * FROM myTrace;  
ROLLBACK;  
```  

and then the stored function myFun

```
CREATE FUNCTION myFun (IN p_no INT, IN p_in VARCHAR(30))  
RETURNS VARCHAR(30)  
LANGUAGE SQL  
BEGIN  
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)  
    VALUES (p_no, user, current date, current time, 'myProc', p_in);  
    IF (p_no = 1) THEN COMMIT; ELSE ROLLBACK; END IF;  
    RETURN p_in;  
END;  
```  

to be tested as follows

```
SELECT myFun (1, 'Hello ISO SQL/PSM') FROM myTrace;  
SELECT * FROM myTrace;  
ROLLBACK;  
```  

Using this format we can test which products allow COMMIT and ROLLBACK statements in stored routines. Just as Hello World programs are perhaps the most important programs for education, these examples try to present useful details to be applied in real application development.

As basic examples we use the bank transfer example of the “SQL Transactions” handbook of DBTechNet, creating the table Accounts as follows

```
CREATE TABLE Accounts ( acctID  INTEGER NOT NULL PRIMARY KEY,  balance INTEGER NOT NULL,  CONSTRAINT unloanable_account CHECK (balance >= 0) );  
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);  
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);  
COMMIT;  
```
Finally we will discuss on the triggers and as an example the various implementations of our Row Version Verifying (RVV) stampings for optimistic locking. Most implementations are based on row-level triggers, but some products can provide a native data type for the version stamping column. A typical use of triggers is implementing business rules beyond the built-in constraints. In Part 3 we will experiment with some side-effects of triggers and foreign keys, and implementing some referential integrity rules by triggers.

2.1 DB2 SQL PL

Even if the first ANSI SQL standard was based on DB2 SQL, the stored procedures in DB2 were first implemented only as external routines written in various programming languages with embedded SQL for data access, and were only registered as routines available for the DB2 server engine. The SQL stored routines were implemented only some ten years ago as SQL Procedural Language based on the SQL/PSM standard.

In DB2 SQL PL the SPECIFIC clause can be used for CREATE procedure, function or method to define an alternate unique name for the routine. The specific name can be used in ALTER and DROP commands, for example as follows

```
DROP SPECIFIC PROCEDURE <specific name>
```

but polymorphism / overloading of stored routine names based on the number of parameters has been implemented only for functions. To verify this and use of specific names on functions we create the following routines:

```sql
db2 -td/
CONNECT TO testdb /

CREATE FUNCTION F (p1 INT)
RETURNS VARCHAR(30)
SPECIFIC myF1
RETURN 'p1' /

CREATE FUNCTION F (p1 INT, IN p2 INT)
RETURNS VARCHAR(30)
SPECIFIC myF2
RETURN 'p1, p2' /

CREATE FUNCTION F (p1 INT, p2 INT, p3 INT)
RETURNS VARCHAR(30)
SPECIFIC myF3
RETURN 'p1, p2, p3' /
```

DB2 has a table called SYSIBM.SYSDUMM1 which can be used for reading any scalar values, but let’s first create our own single row table like the DUAL table of Oracle as follows:

```sql
CREATE TABLE DUAL (X CHAR(1) NOT NULL PRIMARY KEY,
                    CHECK (X IN 'X')) /
INSERT INTO DUAL VALUES ('X') /
COMMIT /
```
Now we can test the overloading functionality as follows:

```sql
db2 => SELECT F(1,2) FROM DUAL /
1
------------------------------
p1, p2
1 record(s) selected.

db2 => SELECT F(1,2,3) FROM DUAL /
1
------------------------------
p1, p2, p3
1 record(s) selected.
```

Using the SPECIFIC names we can drop the functions as follows:

```sql
db2 => DROP SPECIFIC FUNCTION myF /
DB20000I  The SQL command completed successfully.

db2 => DROP SPECIFIC FUNCTION myF2 /
DB20000I  The SQL command completed successfully.

db2 => DROP SPECIFIC FUNCTION myF3 /
DB20000I  The SQL command completed successfully.
```

We continue testing the polymorphism / overloading based on data types of parameters:

```sql
CREATE FUNCTION pmf (p1 INT)
RETURNS VARCHAR(10)
SPECIFIC myPMF1
RETURN 'p1 INT' /

CREATE FUNCTION pmf (p1 REAL)
RETURNS VARCHAR(10)
SPECIFIC myPMF2
RETURN 'p1 REAL' /
```

and running the following commands:

```sql
db2 => SELECT pmf (1) FROM DUAL /
1
------------------------------
p1 INT
1 record(s) selected.

db2 => SELECT pmf (1.1) FROM DUAL /
1
------------------------------
p1 REAL
1 record(s) selected.
```

© Martti Laiho 2014
Since SQL statements inside stored routines are ended by a semicolon, so to control the end of CREATE commands some other termination character need to be defined, for example “/” by starting the Command Line Processor program CLP using command line option –td/ as shown above.

An easy-to-read introduction to DB2 SQL PL is available in “DB2 Application Development” eBook by Raul F. Chong et al at the DB2 on Campus Book Series. An extensive free eBook on DB2 Stored Procedures, Triggers, and User-Defined Functions is available at the IBM Redbooks website. We will cover the topics starting with our minimalistic examples:

Simple myProc and myFun tests

-- Using Command Editor:
CONNECT TO testdb @

CREATE PROCEDURE myProc (IN p_no INT, IN p_in VARCHAR(30), OUT p_out VARCHAR(30))
LANGUAGE SQL
BEGIN
  SET p_out = p_in;
  INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
  VALUES (p_no, user, current date, current time, 'myProc', p_in);
  IF (p_no = 1) THEN COMMIT; ELSE ROLLBACK; END IF;
END @

-- Test using CLP:
db2 -c
db2 => CONNECT TO testdb
db2 => CALL myProc (1, 'Hello DB2', ?)

Value of output parameters
----------------------------------
Parameter Name : P_OUT
Parameter Value : Hello DB2

Return Status = 0

db2 => SELECT * FROM myTrace
SQL Stored Routines

Draft 2015-05-12, page 23

<table>
<thead>
<tr>
<th>T_NO</th>
<th>T_USER</th>
<th>T_DATE</th>
<th>T_TIME</th>
<th>T_PROC</th>
<th>T_WHAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-</td>
<td>04/04/2014</td>
<td>15:23:15</td>
<td>myProc</td>
<td>Hello DB2</td>
</tr>
</tbody>
</table>

2 record(s) selected.

db2 => ROLLBACK

DB20000I The SQL command completed successfully.

-- Command Editor:
CREATE FUNCTION myFun (IN p_no INT, IN p_in VARCHAR(30))
RETURNS VARCHAR(30)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
    VALUES (p_no, user, current date, current time, 'myProc', p_in);
    IF (p_no = 1) THEN COMMIT; ELSE ROLLBACK; END IF;
    RETURN p_in;
END @

-- Test using CLP:
db2 -c
db2 => CONNECT TO testdb
db2 => SELECT myFun (1, 'Hello DB2 fun') FROM myTrace

SQL0740N Routine "STUDENT.MYFUN" (specific name "SQL140404153324600") is defined with the MODIFIES SQL DATA option, which is not valid in the context where the routine is invoked. SQLSTATE=51034

DB2 doesn’t allow data modifying SQL statements in scalar SQL functions.

-- Plan B:
CREATE FUNCTION myFun (IN p_no INT, IN p_in VARCHAR(30))
RETURNS VARCHAR(30)
LANGUAGE SQL
BEGIN
    RETURN p_in;
END @

-- Test using CLP:

db2 -c

db2 => CONNECT TO testdb

db2 => SELECT myFun (1, 'Hello DB2 fun') FROM myTrace

1

Hello DB2 fun

1 record(s) selected.

Even if DB2 does not accept data modifying SQL statements in scalar user-defined functions, these are allowed according to Chong et al in table-valued functions.
Exception handling and BankTransfer examples

Using GET DIAGNOSTICS

```sql
CREATE PROCEDURE BankTransfer (IN fromAcct INT,
                                IN toAcct INT,
                                IN amount INT,
                                OUT msg VARCHAR(100))

LANGUAGE SQL
P1: BEGIN
    DECLARE rowcount INT;
    UPDATE Accounts SET balance = balance - amount
    WHERE acctID = fromAcct;
    GET DIAGNOSTICS rowcount = ROW_COUNT;
    IF rowcount = 0 THEN
        ROLLBACK;
        SET msg = CONCAT('missing account or negative balance in ', fromAcct);
    ELSE
        UPDATE Accounts SET balance = balance + amount
        WHERE acctID = toAcct;
        GET DIAGNOSTICS rowcount = ROW_COUNT;
        IF rowcount = 0 THEN
            ROLLBACK;
            SET msg = CONCAT('rolled back because of missing account ', toAcct);
        ELSE
            COMMIT;
            SET msg = 'committed';
        END IF;
    END IF;
END P1;

-- test
CALL BankTransfer (101, 202, 100, ?);
```

Value of output parameters

Parameter Name : MSG
Parameter Value : committed

Return Status = 0

```sql
CALL BankTransfer (101, 202, 1000, ?);
```

SQL0545N  The requested operation is not allowed because a row does not satisfy the check constraint "STUDENT.ACCOUNTS.UNLOANABLE_ACCOUNT".

SQLSTATE=23513

Then BankTransfer using Condition Handlers
CREATE PROCEDURE BankTransfer (IN fromAcct INT, 
    IN toAcct INT, 
    IN amount INT, 
    OUT msg VARCHAR(100))

LANGUAGE SQL MODIFIES SQL DATA
P1: BEGIN
    DECLARE acct INT;
    DECLARE EXIT HANDLER FOR NOT FOUND 
    BEGIN ROLLBACK;
        SET msg = CONCAT('missing account ', CAST(acct AS VARCHAR(10)));
    END;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION 
    BEGIN ROLLBACK;
        SET msg = CONCAT('negative balance (?) in ', fromAcct);
    END;
    SET acct = fromAcct;
    SELECT acctID INTO acct FROM Accounts WHERE acctID = fromAcct ;
    UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
    SET acct = toAcct;
    SELECT acctID INTO acct FROM Accounts WHERE acctID = toAcct ;
    UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;
    COMMIT;
    SET msg = 'committed';
END P1 @

CALL BankTransfer (101, 202, 100, ?)

db2 => CALL BankTransfer (101, 202, 100, ?)

Value of output parameters
----------------------------
Parameter Name : MSG
Parameter Value : committed

Return Status = 0

CALL BankTransfer (101, 202, 3000, ?)

db2 => CALL BankTransfer (101, 202, 3000, ?)

Value of output parameters
----------------------------
Parameter Name : MSG
Parameter Value : negative balance (?) in 101

Return Status = 0

CALL BankTransfer (101, 999, 100, ?)

db2 => CALL BankTransfer (101, 999, 100, ?)

Value of output parameters
----------------------------
Parameter Name : MSG
Parameter Value : missing account 999

Return Status = 0
More support on stored procedure and function development, deployment to database, and even interactive debugging facilities are available using the IBM Data Studio workbench (based on Eclipse) installed in our virtual database lab. For more details on this, please see Chapter 8 in the ebook “Getting Started with IBM Data Studio” downloadable at https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Big+Data+University/page/FREE+ebook++Getting+Started+with+IBM+Data+Studio+for+DB2

Triggers

Triggers in DB2 SQL are implemented according to the ISO SQL standard, but an important extension is the NO CASCADE clause of BEFORE triggers, which we apply in the next example.

Another extension is the multi-event DML triggers, which means that the same trigger can react on multiple events defined for example as follows:

```
CREATE TRIGGER <triggername> <timing> INSERT OR UPDATE OR DELETE ON <tablename>...
```

and the actual firing event can be detected in the trigger body by event predicates INSERTING, UPDATING, and DELETING implemented like in Oracle PL/SQL multi-event triggers which we will present later.

RVV example on optimistic locking

RVV stands for Row-Version Verification, for which we use server-side stamping of row whenever the row is updated. Then in any SELECT – UPDATE sequence during the same transaction, or sequence of transaction in the same connection, or even separate connection of the same client process, the version stamp is read in SELECT action to the client, and verification of version stamp value is included in search condition of the following UPDATE to ensure that no concurrent client has not updated the row meanwhile.

In DB2 we can explicitly add a new column rv to be used as the version stamp, and we will create a row-version stamping trigger as shown below, changing "@" as the statement terminator character:

```
db2 -td@
CONNECT TO <database> @
ALTER TABLE Accounts ADD COLUMN rv INT NOT NULL DEFAULT 0 @
COMMIT @

DROP TRIGGER Accounts_RvvTrg @
CREATE TRIGGER Accounts_RvvTrg
NO CASCADE BEFORE UPDATE ON Accounts
REFERENCING NEW AS new_row OLD AS old_row
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
IF (old_row.rv = 2147483647) THEN
```

4Beside DB2, NO CASCADE clause has been implemented also in Apache DerbyDB
SET new_row.rv = -2147483648;
ELSE
    SET new_row.rv = old_row.rv + 1;
END IF;
END @
COMMIT @

UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101 @
COMMIT @
SELECT * FROM Accounts @

Note: As a new alternative to our trigger solution DB2 now supports also timestamping of rows, as we present in our RVV Paper.

A subset of DB2 SQL PL, called Inline SQL PL, can be used also in compound SQL inlined statement scripts. This has several limitations, for example cursors, condition handlers, and COMMIT and ROLLBACK statements are not supported.

2.2 Oracle PL/SQL

A database server architecture typically consists of two layers: the SQL engine which takes care of the SQL language and optimizes the execution plan, which is passed for execution to the lower level database engine. The Oracle server consists of an additional PL engine on top of the stack as described in Figure 7. The PL engine interprets the Ada based Procedural Language of Oracle which is known as PL/SQL in which we typically have “embedded” SQL commands (Stürner, 1995). The SQL commands based on the SQL language of Oracle (which is defined in a separate language manual) will then be passed for processing to the lower levels.

![Figure 7. Oracle server’s stack of engines and languages](image)
The PL/SQL blocks can be included in stored procedures, functions, package methods, or used as independent anonymous blocks, as described in figures 7b, 7c, and 7d.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Function</th>
<th>Anonymous</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE myProc</td>
<td>FUNCTION myFunc</td>
<td>[DECLARE</td>
</tr>
<tr>
<td>IS</td>
<td>RETURN someType</td>
<td>-- declarations</td>
</tr>
<tr>
<td>[-- declarations]</td>
<td>IS</td>
<td>BEGIN</td>
</tr>
<tr>
<td>BEGIN</td>
<td>[-- declarations]</td>
<td>-- statements</td>
</tr>
<tr>
<td>[EXCEPTION</td>
<td>[EXCEPTION</td>
<td>[EXCEPTION</td>
</tr>
<tr>
<td>-- error handling]</td>
<td>-- statements</td>
<td>-- error handling</td>
</tr>
<tr>
<td>END;</td>
<td>END;</td>
<td>END;</td>
</tr>
</tbody>
</table>

**Figure 7b.** PL/SQL blocks (as presented by Timo Leppänen / Oracle Finland)

The PL/SQL anonymous blocks are especially useful in external PL/SQL scripts for automation of database administration tasks and as testing tools. Scripting use is out of scope of this tutorial, but we use anonymous blocks for testing our stored routine examples, like in the anonymous block for calling the del_emp procedure with parameters in Figure 7c.

```sql
CREATE OR REPLACE PROCEDURE del_emp(
    p_id IN employees.employees_id%TYPE,
    p_ok OUT BOOLEAN)
IS BEGIN
    DELETE FROM employees
    WHERE employees_id = p_id;
    IF SQLROWCOUNT = 0 THEN
        p_ok := FALSE;
    ELSE
        p_ok := TRUE;
    END IF;
END del_emp;
...
DECLARE
    v_ok BOOLEAN;
BEGIN
    del_emp(1234, p_ok => v_ok);
END;
```

**Figure 7c.** Create and use examples on PL/SQL procedures and functions (as presented by Timo Leppänen / Oracle Finland)

An important concept in PL/SQL is package, described in Figure 7d. PL/SQL packages can seen as singular objects, with public attributes and methods presented in the interface part, and private implementations of the methods as procedures or functions, and possible hidden procedures and functions in the separately created package body.
With Oracle server comes also a large collection of built-in packages, which extend the PL/SQL language to enable, for example, messaging, extended locking, scheduling, accessing LOBs, and files. Later, in our examples we make use DBMS_LOCK and DBMS_OUTPUT packages.

For more detailed introduction to PL/SQL see Chapter 8 Advanced SQL in the “Database Systems” textbook of Connolly and Begg.

**Simple myProc and myFun tests**

Let's assume that the SYSTEM user of the Oracle instance has created a virtual user USER1 as follows

```sql
CREATE USER user1 IDENTIFIED BY sql;
GRANT CONNECT, RESOURCE, CREATE VIEW TO user1;
```

We will now login to the database in Terminal window as USER1 for the following experiments:

```sql
sqlplus user1/sql
```

Oracle’s data type DATE contains both date and time, and there is no separate TIME type, so we need to change the PL/SQL structure of table myTrace as follows, and we can create a view extracting the date and time parts of the DATE data type

```sql
CREATE TABLE myTrace (  
t_no INT NOT NULL,  
t_user VARCHAR(20),  
t_date DATE,  
t_proc VARCHAR(10),
```
Now let's verify how it looks as a native Oracle SQL table:

```sql
SQL> describe myTrace;
Name Null?    Type
-------------------------------
T_NO  NOT NULL NUMBER(38)
T_USER VARCHAR2(20)
T_DATE DATE
T_PROC VARCHAR2(10)
T_WHAT VARCHAR2(30)

SQL> describe myTraceV;
Name Null?    Type
-------------------------------
T_NO  NOT NULL NUMBER(38)
T_USER CHAR(20)
T_DATE VARCHAR2(10)
T_TIME VARCHAR2(8)
T_PROC VARCHAR2(10)
T_WHAT VARCHAR2(30)
```

The myProc procedure in PL/SQL looks as follows:

```sql
CREATE OR REPLACE PROCEDURE myProc (p_no IN INT,p_in IN VARCHAR, p_out OUT VARCHAR)
IS
BEGIN
  p_out := p_in;
  INSERT INTO myTrace (t_no, t_user, t_date, t_proc, t_what)
  VALUES (p_no, SUBSTR(USER,1,20), SYSDATE, 'myProc', p_in);
  IF (p_no = 1) THEN COMMIT; ELSE ROLLBACK; END IF;
END;
/
```

Interesting feature in Oracle is that parameters of VARCHAR data type cannot have length value.

For testing the procedure we can use following PL/SQL anonymous block:
```
DECLARE
  p_out VARCHAR(30) := ' '
BEGIN
  myProc (1, 'Hello Oracle',p_out) ;
  dbms_output.put_line(p_out);
END;
/
```

and run the test block as follows, setting first the serveroutput for displaying the DBMS_OUTPUT messages:
```
SQL> set serveroutput on ;
SQL> DECLARE
    p_out VARCHAR(30) := ' ';
BEGIN
  myProc (1, 'Hello Oracle',p_out) ;
  dbms_output.put_line(p_out);
END;
/
END;
/
Hello Oracle

PL/SQL procedure successfully completed.

```
SELECT * FROM myTrace;
T_NO T_USER T_DATE T_TIME T_PROC T_WHAT
----------------- ----------- ----------- -------------- ------- -------------------
1 USER1 25-APR-14 22:40:31 myProc Hello Oracle
```

How about rollback?

```
DELETE FROM myTrace WHERE t_no < 2;
COMMIT;
DECLARE out VARCHAR(30) ;
BEGIN
   myProc (0, 'Hello Oracle', out) ;
   dbms_output.put_line(out);
END;
/
Hello Oracle
```

```
SELECT * FROM myTrace;
SQL> no rows selected
```

Now let's see how it would work as a PL/SQL function

```
CREATE OR REPLACE FUNCTION myFun (p_no INT, p_in VARCHAR)
RETURN VARCHAR IS
BEGIN
   INSERT INTO myTrace (t_no, t_user, t_date, t_proc, t_what)
   VALUES (p_no, user, SYSDATE, 'myFun', p_in);
   RETURN p_in;
END;
/
```

In the following we test the function invoking it from a SELECT command:

```
SQL> SELECT myFun (1, 'Hello Oracle ') FROM DUAL ;
SELECT myFun (1, 'Hello Oracle ') FROM DUAL *
```

```
ERROR at line 1:
ORA-14551: cannot perform a DML operation inside a query
ORA-06512: at "USER1.MYFUN", line 4
```

But instead of SELECT statements, a function with DML operations can be invoked where an expression can be used, for example as follows:

```
set serveroutput on;
DECLARE r VARCHAR(30);
BEGIN r := myFun(2, 'Hello again!');
   DBMS_OUTPUT.PUT_LINE(r);
END;
/
```
Exception handling and BankTransfer examples

PL/SQL was one of the first procedural extensions to SQL, and has influenced to SQL/PSM and some competitors. However the syntax and procedure structure in SQL/PSM differs a bit from the following structure used in PL/SQL:

```
CREATE [OR REPLACE] PROCEDURE <name> [(parameter [IN |OUT | INOUT ] <data type> [, .. ] ) ]
IS
   [ <local variable declarations> ]
   [ <condition declarations> ]
BEGIN
   <SQL statement> [, .. ]
 [EXCEPTION
   <exception handler> [, .. ] ]
END [<name>] ;
/
```

Instead of SQL/PSM way of declaring condition handlers before the actual SQL statement executions, in PL/SQL blocks, all exception handlers are gathered into the optional EXCEPTION part of the block, and defined as follows

```
WHEN <exception name> THEN <SQL statement> ; [ .. ]
[ .. ]
[WHEN OTHERS THEN <SQL statement> ; [ .. ] ]
```

Interesting observation is that exceptions are diagnosed using SQLCODE values and mnemonics, whereas Oracle does not support native SQLSTATE values in PL/SQL.

The following example demonstrates some differences between PL/SQL and SQL/PSM using the stored procedure BankTransfer we presented above. For pause in the PL/SQL concurrency testing, we use built-in package DBMS_LOCK and we need allow its use by the following:

```
sqlplus / as SYSDBA
GRANT EXECUTE ON DBMS_LOCK TO user1;
EXIT;
```

The SELECT statements in the procedure check that the accounts exist, and the FOR UPDATE clause locks the accessed row. If an account doesn’t exist, then NO_DATA_FOUND exception is raised automatically by the SELECT.
CREATE OR REPLACE PROCEDURE BankTransfer
    (fromAcct IN INT,
     toAcct IN INT,
     amount IN INT,
     msg OUT VARCHAR )
IS
    acct INT;
    ernum NUMBER;
    mesg VARCHAR2(200);
BEGIN
    acct := fromAcct;
    SELECT acctID INTO acct FROM Accounts
    WHERE acctID = fromAcct FOR UPDATE;
    UPDATE Accounts SET balance = balance - amount
    WHERE acctID = fromAcct;
    dbms_lock.sleep(20); -- pause for deadlock testing
    acct := toAcct;
    SELECT acctID INTO acct FROM Accounts
    WHERE acctID = acct FOR UPDATE;
    UPDATE Accounts SET balance = balance + amount
    WHERE acctID = toAcct;
    COMMIT;
    mesg := 'committed';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        ROLLBACK;
        mesg := 'missing account ' || TO_CHAR(acct);
    WHEN OTHERS THEN
        ROLLBACK;
        ernum := SQLCODE;
        mesg := SUBSTR(SQLERRM, 1, 200);
        mesg := mesg || ' SQLcode=' || TO_CHAR(ernum);
END;
/
show errors

In the following we will run concurrency test of the UPDATE-UPDATE scenario using PL/SQL blocks in opposite orders. Note the format of invoking PL/SQL procedures and how OUT parameters are handled.

clear screen
-- session A
set serveroutput on;
DECLARE mesg VARCHAR2(200) := ' ';
BEGIN
    BankTransfer (101,202,100, mesg);
    DBMS_OUTPUT.PUT_LINE(mesg);
END;
/
SELECT * FROM Accounts;
COMMIT;
-------------------
-----------
In an other Terminal window
sqlplus user1/sql
-- concurrent session B
set serveroutput on;
DECLARE mesg VARCHAR2(200) := ' ';
BEGIN
    BankTransfer (202,101,100, mesg);
    DBMS_OUTPUT.PUT_LINE(mesg);
END;
Following PL/SQL anonymous block tests the case when the other account is missing

```sql
DECLARE mesg VARCHAR2(200) := ' '; 
BEGIN 
    BankTransfer (101,999,100, mesg); 
    DBMS_OUTPUT.PUT_LINE(mesg); 
END;
/
SELECT * FROM Accounts; 
COMMIT;
```

And the following PL/SQL block tests functionality of the CHECK constraint

```sql
DECLARE mesg VARCHAR2(200) := ' '; 
BEGIN 
    BankTransfer (101,202,3000, mesg); 
    DBMS_OUTPUT.PUT_LINE(mesg); 
END;
/
SELECT * FROM Accounts; 
COMMIT;
```
On Oracle Triggers

Beside the usual DML triggers, Oracle supports triggers also on DDL command, login/logout and server events.

An interesting feature in PL/SQL triggers is multi-event DML triggers, where the events can be caught by INSERTING, UPDATING, or DELETING conditions, as explained by the following pseudocode:

```
CREATE OR REPLACE TRIGGER <trigger name>
  {AFTER | BEFORE} INSERT OR UPDATE OR DELETE ON <table name>
  FOR EACH ROW
DECLARE <local variables>;
BEGIN
  IF INSERTING THEN <handling of insert case>;
  IF UPDATING THEN <handling of the update case>;
  IF DELETING THEN <handling of the delete case>;
EXCEPTION
  <exception handlers>;
END;
```

For better performance Oracle has also implemented extension of statement-level multi-event DML triggers controlling actions on statement and row levels, which they call compound triggers and which are explained in the PL/SQL Language Reference manual. A compound trigger may contain sections on all possible timing-points: before statement, before each row, after each row, and after statement. Any of these sections may contain conditional executions on inserting, updating, deleting and applying. The topic of compound triggers goes far beyond the scope of our introductory level tutorial, but in Part 3 we will present referential integrity rule triggers which we have adapted from the PL/SQL Language Reference manual and a simple example on use of compound triggers.

RVV example on optimistic locking

In PL/SQL session we can use local variables only inside compound commands, so in the following we will have 2 transactions in the same block of client A and 20 seconds pause between them. In the first transaction (step 1) A reads account 101 and updates the same account in the second transaction (step 3). During the pause client B (in step 2) updates the account 101. Client A can update account 101 only if no other client has meanwhile updated the account.

For the pause in PL/SQL we use the built-in package DBMS_LOCK and we need to allow it use to user1 (unless granted before) by following:

```
sqlplus / as SYSDBA
GRANT EXECUTE ON DBMS_LOCK TO user1;
EXIT;
```

We will first initialize the table Accounts and create the RVV trigger

```
sqlplus user1/sql
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
```
COMMIT;
DESCRIBE Accounts;
-- if column rv is missing
ALTER TABLE Accounts
  ADD COLUMN rv INT DEFAULT 0; --

CREATE OR REPLACE TRIGGER Accounts_RvvTrg
BEFORE UPDATE ON Accounts
FOR EACH ROW
BEGIN
  IF (:OLD.rv = 2147483647) THEN
    :NEW.rv := -2147483648; ELSE
    :NEW.rv := :OLD.rv + 1;
  END IF;
END;
/ COMMIT;

Using the concurrent sqlplus sessions we can now test if the trigger works

-- step 1 Client A
SELECT * FROM Accounts;
DECLARE  p_balance INT;
  p_rv INT;
BEGIN
  SELECT balance, rv INTO p_balance, p_rv FROM Accounts WHERE acctID = 101;
  COMMIT;
  DBMS_LOCK.SLEEP(20);
-- step 3 after client B
  UPDATE Accounts SET balance = balance - 500
    WHERE acctID = 101 AND rv = p_rv;
END;
/
SELECT * FROM Accounts;
ROLLBACK;

-- step 2 Client B while A is sleeping
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;

Note the assignment operator “:=” of Ada and colons “:” in front of the row qualifiers OLD and NEW.
For the Oracle 10.2 and later versions (excluding the XE edition) we don’t need the RVV trigger whereas we can use the ROWSCN pseudo column as row version indicator, as explained in the “RVV Paper” at www.dbtechnet.org/papers/RVV_Paper.pdf

Notes: According to Oracle manuals the INSTEAD OF triggers can be created only for views. (To be tested with other products too).

2.3 SQL Server Transact-SQL

Microsoft SQL Server’s Transact-SQL (T-SQL) language is a procedural language with integrated SQL dialect. The language was originally developed by Sybase already in 80’s, so the stored routines don’t follow the SQL/PSM. In the following we have myProc in T-SQL

Simple myProc and myFun tests

```sql
CREATE PROCEDURE myProc @p_no INT, @p_in VARCHAR(30), @p_out VARCHAR(30) OUT
AS BEGIN
    SET NOCOUNT ON
    SET @p_out = @p_in;
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
    VALUES (@p_no, CURRENT_USER, GETDATE(), CAST(GETDATE() AS time(0)), 'myProc', @p_in);
    IF (@p_no = 1) COMMIT;
ELSE ROLLBACK;
END
GO
```

© Martti Laiho 2014
Following some test result

```sql
SET IMPLICIT_TRANSACTIONS ON
DECLARE @p_out VARCHAR(30)
DECLARE @p_in VARCHAR(30) = 'Hello T-SQL' DECLARE @p_no INT = 1
EXEC myProc @p_no, @p_in, @p_out OUTPUT
SELECT @p_out AS p_out
                   -----------
Hello T-SQL
                   (1 row(s) affected)

SELECT * FROM myTrace t_no t_user t_date t_time t_proc t_what
           -----------
2           NULL           NULL    NULL    NULL    NULL
1           dbo            2014-04-04 21:56:56.0000000 myProc  Hello T-SQL
(2 row(s) affected)
```

Then let's look at adapting the myFun function in T-SQL:

```sql
CREATE FUNCTION myFun (@p_no INT, @p_in VARCHAR(30))
RETURNS VARCHAR(30)
BEGIN
    SET @p_in = 'count(*) = ' + CAST((SELECT COUNT(*) FROM myTrace) AS VARCHAR);
    RETURN @p_in;
END;
GO
```

=> DML statements in T-SQL function can only access a local table inside the function

-- Plan B:
-- Note: SELECT statements inside a function cannot return data to a client, so we have a bit more tricky example using SELECT.

```sql
DROP FUNCTION myFun;
GO
CREATE FUNCTION myFun (@p_no INT, @p_in VARCHAR(30))
RETURNS VARCHAR(30)
BEGIN
    SET @p_in = 'count(*) = ' + CAST((SELECT COUNT(*) FROM myTrace) AS VARCHAR);
    RETURN @p_in;
END;
GO
```
SELECT dbo.myFun (1, 'Hello T-SQL fun') AS p_out

p_out
------------------------------
count(*) = 1
(1 row(s) affected)

So SELECT statements, but no other DML statements, are allowed in T-SQL functions.

Exception handling and BankTransfer examples

Then let’s test the exception handling in T-SQL procedures implementing our BankTransfer transaction

DROP PROCEDURE BankTransfer;
GO
CREATE PROCEDURE BankTransfer @fromAcct INT, @toAcct INT, @amount INT, @msg VARCHAR(100) OUTPUT
AS BEGIN
    DECLARE @acct VARCHAR(10);
    BEGIN TRY
        BEGIN TRANSACTION;
        UPDATE Accounts SET balance = balance - @amount WHERE acctID = @fromAcct;
        IF @@ROWCOUNT < 1 BEGIN
            SET @acct = @fromAcct;
            SET @msg = 'Missing account ' + @acct;
            RAISERROR (@msg, 11, 1);
        END;
        UPDATE Accounts SET balance = balance + @amount WHERE acctID = @toAcct;
        IF @@ROWCOUNT < 1 BEGIN
            SET @acct = @toAcct;
            SET @msg = 'Missing account ' + @acct;
            RAISERROR (@msg, 11, 1);
        END;
        COMMIT;
        SET @msg = 'Transaction committed';
    END TRY
    BEGIN CATCH
        SELECT -- only for testing
            ERROR_NUMBER() AS ErrorNumber,
            ERROR_SEVERITY() AS ErrorSeverity,
            ERROR_STATE() AS ErrorState,
            ERROR_PROCEDURE() AS ErrorProcedure,
            ERROR_LINE() AS ErrorLine,
            ERROR_MESSAGE() AS ErrorMessage;
        SET @msg = ERROR_MESSAGE();
        ROLLBACK;
    END CATCH;
END;}

Let’s now test invoking the procedure. Note the use of the OUTPUT parameter

DECLARE @msg VARCHAR(100);
SET @msg = 'This test run should be OK';
EXEC BankTransfer 101, 202, 100, @msg OUTPUT;
SELECT @msg AS Msg;
DECLARE @msg VARCHAR(100);
SET @msg = 'Missing account';
EXEC BankTransfer 101, 999, 100, @msg OUTPUT;
SELECT @msg AS Msg;

<table>
<thead>
<tr>
<th>ErrorNumber</th>
<th>ErrorSeverity</th>
<th>ErrorState</th>
<th>ErrorProcedure</th>
<th>ErrorLine</th>
<th>ErrorMessage</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>11</td>
<td>1</td>
<td>BankTransfer</td>
<td>19</td>
<td>Missing account 999</td>
</tr>
</tbody>
</table>

Missing account 999

DECLARE @msg VARCHAR(100);
SET @msg = 'Testing CHECK constraint by our triggers';
EXEC BankTransfer 101, 202, 3000, @msg OUTPUT;
SELECT @msg AS Msg;

<table>
<thead>
<tr>
<th>ErrorNumber</th>
<th>ErrorSeverity</th>
<th>ErrorState</th>
<th>ErrorProcedure</th>
<th>ErrorLine</th>
<th>ErrorMessage</th>
</tr>
</thead>
<tbody>
<tr>
<td>E47</td>
<td>16</td>
<td>0</td>
<td>BankTransfer</td>
<td>9</td>
<td>The UPDATE statement conflicted with the CHECK c...</td>
</tr>
</tbody>
</table>

The UPDATE statement conflicted with the CHECK constraint "uniqueable_account". The conflict occurred

Triggers and RVV example on optimistic locking

T-SQL language doesn’t have BEFORE triggers nor row-level triggers. Row-level processing can be constructed in AFTER UPDATE triggers inspecting the following temporary tables: table “DELETED” which contains the before images of the updated rows, and table “INSERTED” which contains the corresponding after images of the updated rows. However, the performance of this row-level processing is not efficient, as shown in our RVV Paper.

Microsoft has its own strategy for triggers extending the concept far beyond the DML events, including logon, schema, and database events.

Like for Oracle we don’t need the RVV trigger in T-SQL language of SQL server, since we can create a rv column of ROWVERSION data type as follows

ALTER TABLE Accounts ADD rv ROWVERSION;

which automatically gets stamped on every UPDATE command, as explained in our “RVV Paper” at www.dbtechnet.org/papers/RVV_Paper.pdf
2.4 MySQL/InnoDB

The language for stored routines in MySQL was implemented as late as 2005 in version 5, so the language is based on SQL/PSM.

Simple myProc and myFun tests

```
-- myProc:
delimiter $$
DROP PROCEDURE if exists myProc $$
CREATE PROCEDURE myProc (IN p_no INT,IN p_in VARCHAR(30), OUT p_out VARCHAR(30))
BEGIN
    SET p_out = p_in;
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
    VALUES (p_no, current_user(), current_date, current_timestamp, 'myProc', p_in);
    IF (p_no = 1) THEN COMMIT; ELSE ROLLBACK; END IF;
END; $$
delimiter

-- to be tested by following
SET AUTOCOMMIT = 0;
SET @out = '';
CALL myProc (1, 'Hello MySQL', @out);
SELECT @out;
SELECT * FROM myTrace;
ROLLBACK;
CALL myProc (0, 'Hello MySQL', @out);
SELECT @out;
SELECT * FROM myTrace;
ROLLBACK;

-- myFun:
delimiter $$
DROP FUNCTION if exists myFun $$
CREATE FUNCTION myFun (p_no INT, p_in VARCHAR(30))
RETURNS VARCHAR(30)
BEGIN
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
    VALUES (p_no, current_user(), current_date, current_timestamp, 'myFun', p_in);
    IF (p_no = 1) THEN COMMIT; ELSE ROLLBACK; END IF;
    RETURN p_in;
END $$
delimiter

COMMIT;
SELECT myFun (1, 'Hello MySQL') FROM myDummy;
SELECT * FROM myTrace;
delete from myTrace;
COMMIT;
```
On CREATE FUNCTION we got the error message

ERROR 1422 (HY000): Explicit or implicit commit is not allowed in stored function or trigger.
Indicating that both COMMIT and ROLLBACK statements are not allowed in an SQL function, so we have commented these out.

Trigger workaround for the missing support of CHECK constraints:

As workaround for the missing support of CHECK constraints we can setup row-level triggers as follows. We start by creating the table and contents for the test.

```sql
DROP TABLE Accounts;
CREATE TABLE Accounts (
  acctID  INTEGER NOT NULL PRIMARY KEY,
  balance INTEGER NOT NULL,
  CONSTRAINT unloanable_account  CHECK (balance >= 0)
);
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;

CREATE TRIGGER Accounts_upd_trg
BEFORE UPDATE ON Accounts
FOR EACH ROW
BEGIN
  IF NEW.balance < 0 THEN
    SIGNAL SQLSTATE '23513'
      SET MESSAGE_TEXT = 'Negative balance not allowed';
  END IF;
END;

CREATE TRIGGER Accounts_ins_trg
BEFORE INSERT ON Accounts
FOR EACH ROW
BEGIN
  IF NEW.balance < 0 THEN
    SIGNAL SQLSTATE '23513'
      SET MESSAGE_TEXT = 'Negative balance not allowed';
  END IF;
END;

--testing the triggers
INSERT INTO Accounts VALUES (1, -1);
GET DIAGNOSTICS @rowcount = ROW_COUNT;
GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE, @sqlcode = MYSQL_ERRNO;
SELECT @sqlstate, @sqlcode, @rowcount;
UPDATE Accounts SET balance = -100 WHERE acctID = 2;
GET DIAGNOSTICS @rowcount = ROW_COUNT;
```

© Martti Laiho 2014
GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE, @sqlcode = MYSQL_ERRNO;
SELECT @sqlstate, @sqlcode, @rowcount;
DELETE FROM Accounts WHERE acctID = 2; COMMIT;

Exception handling and BankTransfer examples

First a version using **DIAGNOSTICS**

```sql
CREATE PROCEDURE BankTransfer (IN fromAcct INT,
IN toAcct   INT,
IN amount   INT,
OUT msg     VARCHAR(100))
P1: BEGIN
    UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
    GET DIAGNOSTICS @rowcount = ROW_COUNT;
    IF @rowcount = 0 THEN
        ROLLBACK;
        SET msg = CONCAT('missing account or negative balance in ', fromAcct);
    ELSE
        UPDATE Accounts SET balance = balance + amount
        WHERE acctID = toAcct;
        GET DIAGNOSTICS @rowcount = ROW_COUNT;
        IF @rowcount = 0 THEN
            ROLLBACK;
            SET msg = CONCAT('rolled back because of missing account ', toAcct);
        ELSE
            COMMIT;
            SET msg = 'committed';
        END IF;
    END IF;
END IF;
END P1 !
```

Then another version using **condition handlers**

```sql
delimiter !
CREATE PROCEDURE BankTransfer (IN fromAcct INT,
IN toAcct   INT,
IN amount   INT,
OUT msg     VARCHAR(100))
LANGUAGE SQL MODIFIES SQL DATA
P1: BEGIN
    DECLARE acct INT;
    DECLARE EXIT HANDLER FOR NOT FOUND
    BEGIN ROLLBACK;
        SET msg = CONCAT('missing account ', CAST(acct AS CHAR));
    END;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN ROLLBACK;
        SET msg = CONCAT('negative balance (?) in ', fromAcct);
    END;
    SET acct = fromAcct;
    SELECT acctID INTO acct FROM Accounts WHERE acctID = fromAcct;
```
UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
SET acct = toAcct;
SELECT acctID INTO acct FROM Accounts WHERE acctID = toAcct;
UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;
COMMIT;
SET msg = 'committed';
END P1 !
delimiter ;

and test scripts:

-- 'This test run should be OK'
CALL BankTransfer (101, 202, 100, @msg);
SELECT @msg;

-- 'Missing account'
CALL BankTransfer (101, 999, 100, @msg);
SELECT @msg;

-- 'Testing CHECK constraint by our triggers';
CALL BankTransfer (101, 202, 3000, @msg);
SELECT @msg;

Triggers

MySQL 5.6 does not yet support INSTEAD OF triggers, nor triggers on views, and not triggers ON UPDATE OF <column> events.

RVV example on optimistic locking

ALTER TABLE Accounts
    ADD COLUMN rv INT DEFAULT 0;
delimiter #
CREATE TRIGGER Accounts_RvvTrg
BEFORE UPDATE ON Accounts
FOR EACH ROW
BEGIN
IF (old.rv = 2147483647) THEN
    SET new.rv = -2147483648;
ELSE
    SET new.rv = old.rv + 1;
END IF;
FAIL
#
delimiter ;

COMMIT;

-- Concurrency test by 2 clients
--
-- step 1 Client A
SET AUTOCOMMIT = 0;
SET @balanceB = 0;  -- init value
SET @rv = 0;  -- init value
SELECT balance, rv INTO @balance, @rv FROM Accounts WHERE acctID = 101;
SELECT @balance, @rv;
COMMIT;

-- step 2 Client B
SET AUTOCOMMIT = 0;
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
SELECT acctID, balance FROM Accounts WHERE acctID = 101; COMMIT;

-- step 3 Client A
UPDATE Accounts SET balance = balance - 500
WHERE acctID = 101 AND rv = @rv;
GET DIAGNOSTICS @rowcount = ROW_COUNT;
SELECT @rowcount;
SELECT acctID, balance, rv FROM Accounts WHERE acctID = 101;
ROLLBACK;

2.5 PostgreSQL PL/pgSQL

For the procedural extensions of PostgreSQL there are multiple choices. Originally the server has only the two basic engine layers, and the Procedural engine need to be added to the server processes per database using the following command in a session of the “postgres” user

    CREATE LANGUAGE plpgsql;

which makes the Procedural Language/PostgreSQL (partly based on Oracle PL/SQL) available in SQL-sessions accessing the database.

In PLPGSQL language procedures are written as functions which return NULL. There is no other essential difference.

Simple myProc and myFun tests

    CREATE OR REPLACE FUNCTION myProc (p_no INT, p_in VARCHAR)
    RETURNS VARCHAR
    AS $$
    BEGIN
        INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
        VALUES (p_no, CURRENT_USER, CURRENT_DATE, CURRENT_TIME, 'myProc', p_in);
        IF (p_no = 1) THEN
            COMMIT;
        ELSE ROLLBACK;
        END IF;
        RETURN p_in;
    END;
    $$
    LANGUAGE plpgsql;

    SELECT myProc (1, 'Hello PostgreSQL')
ERROR:  cannot begin/end transactions in PL/pgSQL
HINT:  Use a BEGIN block with an EXCEPTION clause instead.
CONTEXT: PL/pgSQL function myproc(integer, character varying) line 6 at SQL statement

CREATE OR REPLACE FUNCTION myProc (p_no INT, p_in VARCHAR) RETURNS VARCHAR
AS $$
BEGIN
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
    VALUES (p_no, CURRENT_USER, CURRENT_DATE, CURRENT_TIME, 'myProc', p_in);
    RETURN p_in;
END;
$$
LANGUAGE plpgsql;

SELECT myProc (1, 'Hello PostgreSQL')

Exception handling and BankTransfer examples

Following is the implementation of the BankTransfer example using the PL/SQL like plpgsql language

CREATE OR REPLACE FUNCTION BankTransfer
(IN fromAcct INT,
 IN toAcct   INT,
 IN amount   INT)
RETURNS INT AS $$
BEGIN
    UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
    IF (NOT FOUND) THEN
        RAISE EXCEPTION USING MESSAGE = '* Unknown from account ' || fromAcct;
    ELSE
        UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;
        IF (NOT FOUND) THEN
            RAISE EXCEPTION USING MESSAGE = '* Unknown from account ' || toAcct;
        END IF;
    END IF;
    RETURN 1;
END;
$$
WHEN raise_exception THEN
  RAISE; RETURN -1;
WHEN check_violation THEN
  RAISE; RETURN -1;
WHEN OTHERS THEN
  RAISE NOTICE '** SQLexception %', SQLSTATE; RAISE; RETURN -1;
END;
$$ LANGUAGE plpgsql;

Triggers and RVV example on optimistic locking

For triggers the trigger body needs to be created first as a stored function returning “trigger” as follows in the RVV example tuned for PostgreSQL

```
ALTER TABLE Accounts
ADD COLUMN rv INT NOT NULL DEFAULT 0;

CREATE OR REPLACE FUNCTION Accounts_Rvv()
RETURNS trigger AS $BODY$
BEGIN
  IF (OLD.rv = 2147483647) THEN
    NEW.rv := -2147483648;
  ELSE
    NEW.rv := OLD.rv + 1;
  END IF;
  RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql VOLATILE;

DROP TRIGGER Accounts_RvvTrg ON Accounts;
CREATE TRIGGER Accounts_RvvTrg BEFORE UPDATE ON Accounts
FOR EACH ROW
EXECUTE PROCEDURE Accounts_Rvv();
```

PostgreSQL has also a SQL/PSM based procedural language library available.

2.6 Pyrrho DBMS

Pyrrho is a compact and efficient relational database management system for the .NET framework. It is available at http://www.pyrrhodb.org. It is developed at University of the West of Scotland by Dr. Malcolm Crowe. It is based on a subset of ISO SQL:2011 standard. It uses the real optimistic concurrency control model, which is more suited for Internet use than the traditional database products.

Like in CLP client of DB2, the PyrrhoCMD client tool uses single line commands. Usually a command doesn’t need the terminating character “;”. If we want to use multi-line command, for example in creating a table or stored function, we need to enclose it in brackets [ ] as demonstrated below.
Simple myProc and myFun tests

For the myTrace table we need some syntax tuning as follows:

```sql
[CREATE TABLE myTrace (
    t_no     INT,
    t_user   CHAR,
    t_date   DATE,
    t_time   TIME,
    t_proc   CHAR,
    t_what   CHAR
) ;]
```

Pyrrho does not support the LANGUAGE SQL clause and COMMIT/ROLLBACK statements in procedures so we leave them out. Also for the time being OUT parameters need some more testing, so the following myProc version succeeds:

```sql
[CREATE PROCEDURE myProc (IN p_no INT, IN p_in CHAR, OUT p_out CHAR)
BEGIN
    SET p_out = p_in;
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
        VALUES (p_no, current_user, current_date, current_time, 'myProc', p_in);
END; ]
```

And the following procedure invoking

```
SQL> CALL myProc (1, 'Hello Pyrrho');
SQL> SELECT * FROM myTrace;
```

The following function experiment with myFun implemented in Pyrrho proves that we can execute DML statements in Pyrrho functions. First we create the function:

```sql
[CREATE FUNCTION myFun (IN p_no INT, IN p_in CHAR) RETURNS CHAR
BEGIN
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
        VALUES (p_no, current_user, current_date, current_time, 'myProc', p_in);
RETURN p_in;
END; ]
```

And then experiment with invoking the function:

```
SQL> delete from myTrace;
SQL> CREATE TABLE single (id INT NOT NULL PRIMARY KEY);
SQL> INSERT INTO single VALUES (1);
SQL> SELECT myFun (1, 'Hello Pyrrho fun') FROM single;
```

© Martti Laiho 2014
Exception handling and BankTransfer examples

Following is a test run of the BankTransfer

```sql
SQL> BEGIN TRANSACTION;
SQL-T> [CREATE TABLE Accounts (acctID  INTEGER NOT NULL PRIMARY KEY,
> balance INTEGER NOT NULL
> CONSTRAINT uncreditable_account CHECK (balance >= 0)
> )]
SQL-T> INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
SQL-T> INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SQL-T> SELECT * FROM Accounts;
<table>
<thead>
<tr>
<th>ACCTID</th>
<th>BALANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1000</td>
</tr>
<tr>
<td>202</td>
<td>2000</td>
</tr>
</tbody>
</table>
SQL-T> COMMIT;
SQL> [CREATE FUNCTION BankTransfer (IN fromAcct INT,
>     IN toAcct   INT,
>     IN amount   INT)
>     RETURNS VARCHAR(100)
> BEGIN
>     DECLARE msg VARCHAR(100) ;
>     DECLARE NOT_FOUND BOOLEAN DEFAULT FALSE;
>     DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET NOT_FOUND=TRUE;
>     SET msg = 'ok';
>     UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
>     IF (NOT_FOUND) THEN
>         SET msg = '* Unknown from account ' || CAST(fromAcct AS VARCHAR(10))
>     ELSE
>     BEGIN
>         UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;
>         IF (NOT_FOUND) THEN
>             SET msg = '* Unknown from account ' || CAST(toAcct AS VARCHAR(10))
>         ELSE
>             SET msg = 'OK'
>         END IF
>     END;
>     RETURN msg;
> END;]
SQL> BEGIN TRANSACTION;
SQL-T>SELECT BankTransfer (101, 202, 100) FROM DUAL;
```

© Martti Laiho 2014
Triggers and RVV example on optimistic locking

Just like in PostgreSQL, the trigger body need to be defined as a separate function in case it consists of multiple statements.

Just to give an example of creating a trigger to Pyrrho database, we first implement the RVV versioning for the Accounts table we adding the versioning column rv as follows

```sql
ALTER TABLE Accounts ADD rv INT DEFAULT 0 ;
```

and create the following function and trigger pair for the table

```sql
[CREATE FUNCTION Accounts_Rvv(old_rv INT) RETURNS INT
  IF (old_rv = 2147483647) THEN
    RETURN -2147483648
  ELSE RETURN (old_rv + 1)
  END IF ]

[CREATE TRIGGER Accounts_RvvTrg BEFORE UPDATE ON Accounts
  REFERENCING OLD ROW AS old_row NEW ROW AS new_row
  FOR EACH ROW
  SET new_row.rv = Accounts_Rvv(old_row.rv) ]
```

Now, let’s verify the RVV versioning

```sql
SQL> UPDATE Accounts SET balance = balance - 100 WHERE acctid = 101;
SQL> SELECT * FROM Accounts;
```

<table>
<thead>
<tr>
<th>ACCTID</th>
<th>BALANCE</th>
<th>RV</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>900</td>
<td>1</td>
</tr>
<tr>
<td>202</td>
<td>2000</td>
<td>0</td>
</tr>
</tbody>
</table>

More examples on Pyrrho triggers can be found at [http://pyrrhodb.uws.ac.uk/triggers.htm](http://pyrrhodb.uws.ac.uk/triggers.htm)

However, starting from Pyrrho 5.2 we don’t need triggers for RVV versioning since this update stamping of rows is provided automatically for every table by Pyyrhos’s new advanced pseudocolumn CHECK consisting of components PARTITION, POSITION and VERSIONING where PARTITION refers to the transaction log file, POSITION refers to the row position, and VERSIONING is integer referring to the row version (Malcolm Crowe 2015). The VERSIONING part can be accessed also directly as numeric pseudocolumn serving as version stamp of the row like the RV column we used above. So we can drop the trigger and the RV column, and get the same version stamping functionality, shown below:

```sql
SQL> [CREATE TABLE Accounts (  
> acctID INTEGER NOT NULL PRIMARY KEY,  
> balance INTEGER NOT NULL,  
> CONSTRAINT unloanable_account CHECK (balance >= 0)  
> ); ]
SQL> INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
SQL> INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
SQL> SELECT acctid, balance, check, versioning FROM Accounts;
```

<table>
<thead>
<tr>
<th>ACCTID</th>
<th>BALANCE</th>
<th>CHECK</th>
<th>VERSIONING</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1000</td>
<td>testdb.osp:228:228</td>
<td>228</td>
</tr>
<tr>
<td>202</td>
<td>2000</td>
<td>testdb.osp:264:264</td>
<td>264</td>
</tr>
</tbody>
</table>

© Martti Laiho 2014
SQL> UPDATE Accounts SET balance = balance - 100 WHERE acctid = 101;
1 records affected

SQL> SELECT acctid, balance, check, versioning FROM Accounts;

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCTID</td>
<td>BALANCE</td>
<td>CHECK</td>
<td>VERSIONING</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>------------------</td>
<td>-------</td>
</tr>
<tr>
<td>101</td>
<td>900</td>
<td>testdb.osp:228:301</td>
<td>301</td>
</tr>
<tr>
<td>202</td>
<td>2000</td>
<td>testdb.osp:264:264</td>
<td>264</td>
</tr>
</tbody>
</table>

SQL>

SQL> SELECT acctid, balance, VERSIONING AS RV FROM Accounts;

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCTID</td>
<td>BALANCE</td>
<td>RV</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>101</td>
<td>900</td>
<td>301</td>
</tr>
<tr>
<td>202</td>
<td>2000</td>
<td>264</td>
</tr>
</tbody>
</table>

except that the sequence of RV values per row are not increased by 1, but this is not relevant. Critical functionality for the row version verifying (RVV) protocol is that differing VERSIONING value in database on comparison with the earlier retrieved RV value reveals if the row has been updated meanwhile by some other process. Let’s see, if some other session updates the same row, for example by

SQL> UPDATE Accounts SET balance = balance - 100 WHERE acctid = 101;
1 records affected

and our session wants to update the version 301 seen before, as follows

SQL> [UPDATE Accounts SET balance = 800
> WHERE acctid = 101 AND VERSIONING = 301;]
0 records affected

the VERSIONING test reveals us that our information of the database content is outdated and we are not allowed blindly write over the latest content. However, into an UPDATE sensitive to the current content, like the following

UPDATE Accounts SET balance = balance - 200 WHERE acctid = 101;

we shall not include the VERSIONING test.
Part 3  Concurrency Problems due to Side-effects

The isolation levels defined in the ISO SQL standard can be explained according to the S-locking of the MGL (Multi-Granular Locking) concurrency control, explained in the “SQL Transactions handbook”. So they seem to apply only to SELECT operations protected by S-locks, which may delay write operations of competing transactions, but having no affect in write-only transactions. Even if the transaction itself contains only write operations like INSERT, UPDATE or DELETE commands, other operations may occur in the background. In the following we will study effects of Referential Integrity (i.e. Foreign Key) checks and triggered operations, which are not seen by the application code, but may generate concurrency problems. Sometimes also isolation level and/or the concurrency control mechanism may have impact in these problems.

Let’s create the following minimalistic table pair:

```sql
CREATE TABLE Parent ( pid INT NOT NULL CONSTRAINT Parent_PK PRIMARY KEY,
                      ps VARCHAR(20),
                      psum INT DEFAULT 0 );

CREATE TABLE Child ( cid INT NOT NULL CONSTRAINT Child_PK PRIMARY KEY,
                     pid INT,
                     cs VARCHAR(20),
                     csum INT,
                     CONSTRAINT Child_Parent_FK FOREIGN KEY (pid)
                     REFERENCES Parent (pid) );
```

and insert into these some contents:

```sql
INSERT INTO Parent (pid, ps) VALUES (1,'pa1');
INSERT INTO Parent (pid, ps) VALUES (2,'pa2');
INSERT INTO Child (cid, pid, cs, csum) VALUES (1, 1,'kid1',10);
INSERT INTO Child (cid, pid, cs, csum) VALUES (2, 1,'kid2',20);
INSERT INTO Child (cid, pid, cs, csum) VALUES (3,NULL,'orp3', 30);
SELECT * FROM Parent;
SELECT * FROM Child;
```

3.1 Side-effects due to Foreign Keys

*Note: This chapter is rewritten entirely in May 2015*

In this chapter we will temporarily step aside from the topics procedural SQL and stored routines. The purpose of this chapter is to prepare the reader for the next chapter on concurrency side-effects of triggers. This also extends the discussion on concurrency topics we have presented in our tutorial “On SQL Concurrency Technologies” at www.dbtechnet.org/papers/SQL_ConcurrencyTechnologies.pdf

Based on FOREIGN KEY constraints the DBMS protects the referential integrity (RI) between child rows of the table and their logical parent rows in the parent table (or in the same table). This protection does not come for free, but requires read operations by the DBMS, thus influencing the concurrency control. In the following we test how SQL Server and the DBMS products in our database laboratory sort out the concurrency in case two concurrent clients, A and B, updating different rows in the Parent table and then inserting a child row for the parent row which the other has updated. The updated rows are locked with
X-lock, and if the INSERT consistency rule of the Child table is checked using read operation protected by S-lock, then this would lead to concurrency conflict, i.e. deadlock.

**SQL Server Transact-SQL**

We start the tests using SQL Server 2012 in which we can easily verify the assumed locking behavior on foreign key lookups. We run the 2 concurrent sessions in SQL Server Management Studio, having process 54 for Client A and process 55 for Client B, as follows:

Right after the INSERT command of process 55 in step 2 we look at the locking status by a third session (process 58) using the `sp_lock` procedure of administrator:

Right after the process 54 enters its INSERT command in step 3, the server detects the deadlock status and selects process 55 as the victim aborting it by rollback operation and raising the following exception.
Msg 1205, Level 13, State 51, Line 2
Transaction (Process ID 54) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

Our tutorial “On SQL Concurrency Technologies” explains also how to interpret the columns of the sp_lock report.

**DB2 SQL PL**

We will now proceed to test behavior of DB2 Express-C using the following test scenario:

```sql
SELECT * FROM Parent;
SELECT * FROM Child;

1. Client A
SET ISOLATION = RS;
UPDATE Parent SET psum = 100 WHERE pid = 1;

2. Client B
SET ISOLATION = RS;
SET CURRENT LOCK TIMEOUT = 20;
UPDATE Parent SET psum = 200 WHERE pid = 2;
INSERT INTO Child (cid,pid,cs,csum) VALUES (4,1,'kid4',40);

3. Client A
SET CURRENT LOCK TIMEOUT = 20;
INSERT INTO Child (cid,pid,cs,csum) VALUES (5,2,'kid5',10);
```

The isolation level RS of DB2 corresponds the REPEATABLE READ isolation of ISO SQL and would protect reads by S-locks which will be kept up to end of transaction.

By default DB2 Express-C seem to use locksize PAGE, so even the first Parent row update by client A would block client in step 2, but after we changed the locksize to ROW our test proceeded as follows:

```sql
db2 => connect to testdb;
Database Connection Information
| Database server | DB2/LINUX 9.7.2 |
| SQL authorization ID | STUDENT |
| Local database alias | TESTDB |

db2 => 1. Client A
SET ISOLATION = RS;
db2 => DEED0001 The SQL command completed successfully.
db2 => UPDATE Parent SET psum = 100 WHERE pid = 1;
DB200001 The SQL command completed successfully.

db2 => 2. Client B
SET ISOLATION = RS;
SET CURRENT LOCK TIMEOUT = 20;
db2 => DEED0001 The SQL command completed successfully.
db2 => UPDATE Parent SET psum = 200 WHERE pid = 2;
DB200001 The SQL command completed successfully.
db2 => INSERT INTO Child (cid,pid,cs,csum) VALUES (4,1,'kid4',40);
DB200001 The SQL command completed successfully.
db2 => SELECT * FROM Parent;

<table>
<thead>
<tr>
<th>PID</th>
<th>PS</th>
<th>PSLM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

2 record(s) selected.

db2 => 1. Client A
SET CURRENT LOCK TIMEOUT = 20;
db2 => INSERT INTO Child (cid,pid,cs,csum) VALUES (5,2,'kid5',10);
DB200001 The SQL command completed successfully.

db2 => SELECT * FROM Parent;

<table>
<thead>
<tr>
<th>PID</th>
<th>PS</th>
<th>PSLM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

2 record(s) selected.

db2 => 2. Client B
SET ISOLATION = RS;

db2 => DEED0001 The SQL command completed successfully.
db2 => SET CURRENT LOCK TIMEOUT = 20;
DB200001 The SQL command completed successfully.
db2 => UPDATE Parent SET psum = 200 WHERE pid = 2;
DB200001 The SQL command completed successfully.
db2 => INSERT INTO Child (cid,pid,cs,csum) VALUES (4,1,'kid4',40);
DB200001 The SQL command completed successfully.
db2 => SELECT * FROM Parent;
SQL0911N The current transaction has been rolled back because of a deadlock or timeout. Reason code = '58'. SIGSTATE=40001

The INSERT of client A did not raise exception, so we can conclude that the foreign key lookup to the referred table is not done by lock protected reads. The explicit read operations by SELECT commands in our test finally lead to deadlock and raising the abort transaction exception to client B.
Oracle PL/SQL

Oracle does not use S-locks for read protection. The default isolation level READ COMMITTED means that if the row to be read is locked then Oracle reads the latest committed version of the row. The isolation level SERIALIZABLE means that read operations see the latest committed row versions at the start time of the transaction. Note that if the row’s record at the ROWID location is not locked, then it is the first node in the history of committed versions.

Let’s now verify by the following experiment that the foreign key lookups for INSERT commands don’t lead to concurrency conflict in Oracle:

```sql
-- 1. Client A
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE Child SET pid = 1 WHERE c16 = 2;
INSERT INTO Child (c10, pid, c7, c5, csum) VALUES (4, 1, 'kid4', 40);
SELECT * FROM Child;

-- 2. Client B
SELECT * FROM Parent;
...
```

From the results we can see that no concurrency conflict was detected, and at the end of transactions the clients see different row versions in the database.

Let’s now test if Oracle performs the lookup read of foreign key integrity against the committed versions inserting a temporary parent in a transaction, and then after deleting it by inserting a child row for it:
This proves that the foreign key integrity is verified against the existence of the row instead of its version history.

MySQL/InnoDB

We will now proceed testing the behavior of MySQL/InnoDB on foreign key RI lookup. For the test we create the tables and the initial contents as we have done with the other DBMS products, and observe the results of the concurrent client pair, A and B, which don’t touch the same rows in following test scenario:

```sql
mysql> -- Client A
mysql> SELECT * FROM Parent;
+----+----+-----+
| pid| ps | psum|
+----+----+-----+
|   1| pa1|    0|
|   2| pa2|    0|
+----+----+-----+
2 rows in set (0.00 sec)

mysql> SELECT * FROM Child;
+----+----+----+-----+
| cid| pid| cs | csum|
+----+----+----+-----+
|   1|   1| kid1|   10|
|   2|   1| kid2|   20|
|   3| NULL| orp3|   30|
+----+----+----+-----+
3 rows in set (0.01 sec)
```
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> -- 1. Client A
mysql> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE Parent SET ps = 'new' WHERE pid = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

-- --------------------------------------

mysql> -- 2. Client B
mysql> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> set lock_wait_timeout = 300;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE Parent SET ps = 'new' WHERE pid = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> INSERT INTO Child (cid, pid, cs, csum) VALUES (4, 1, 'kid4', 40);
The INSERT command of Client B is blocked waiting for read lock of the foreign key RI lookup

-- --------------------------------------

mysql> -- 3. Client A
mysql> INSERT INTO Child (cid, pid, cs, csum) VALUES (5, 2, 'kid5', 50);
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

-- --------------------------------------

Now the session of Client B can continue

Query OK, 1 row affected (13.67 sec)

mysql> SELECT * FROM Parent;
+----+-----+-----+
| pid | ps  | psum|
+----+-----+-----+
|  1 | pa1 |    0|
|  2 | new |    0|
+----+-----+-----+
2 rows in set (0.00 sec)

mysql> SELECT * FROM Child;
+----+----+---+-----+
| cid | pid | cs | csum|
+----+----+---+-----+
|  1 |  1 | kid1|   10|
+----+----+---+-----+
We know that DML read operations of InnoDB on READ COMMITTED isolation level will access the latest committed version of the row avoiding S-locking. From the results of our experiment we conclude that for the foreign key RI lookup InnoDB will check the existence of the row, but using lock protection, which for client B led to over 13 second waiting and on INSERT command of client A on step 3 to deadlock and rollback of client A’s transaction.

PostgreSQL PL/pgSQL

Let’s proceed to test the behavior of PostgreSQL using two concurrent clients A and B which don’t touch any same rows in the following scenario, like we had with MySQL/InnoDB above:

testdb=> SELECT * FROM Parent;
  pid | ps  | psum
-----|-----|-----
  1  | pa1 |    0
  2  | pa2 |    0
(2 rows)

testdb=> SELECT * FROM Child;
cid | pid |  cs  | csum
-----|-----|-----|-----
  1  | 1   | kid1 |   10
  2  | 2   | kid2 |   20
  3  |     | orp3 |   30
(3 rows)

testdb=> COMMIT;
COMMIT
testdb=> -- 1. Client A
testdb=> BEGIN WORK;
BEGIN
testdb=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
testdb=> UPDATE Parent SET ps = 'new' WHERE pid = 1;
UPDATE 1

-- --------------------------------------
testdb=> -- 2. Client B
testdb=> BEGIN WORK;
BEGIN
testdb=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
testdb=> UPDATE Parent SET ps = 'new' WHERE pid = 2;
UPDATE 1
testdb=> INSERT INTO Child (cid, pid, cs, csum)
testdb- VALUES (4, 1, 'kid4', 40);
The INSERT command of Client B is blocked waiting for read lock of the foreign key RI lookup

-- --------------------------------------
testdb=> -- 3. Client A
testdb=> INSERT INTO Child (cid, pid, cs, csum) VALUES (5, 2, 'kid5', 50);
ERROR: deadlock detected
HINT: See server log for query details.
CONTEXT: SQL statement "SELECT 1 FROM ONLY "public"."parent" x WHERE "pid" OPERATOR(pg_catalog.=) $1 FOR SHARE OF x"

-- ---------------------------------------------------------------

Now the session of Client B can continue

INSERT 0 1
testdb=> select * from Parent;
pid | ps  | psum
-----+-----+-----
 1   | pa1 |    0
 2   | new |    0
(2 rows)

testdb=> select * from Child;
cid | pid | cs  | csum
-----+-----+-----+-----
 1   | 1   | kid1|   10
 2   | 2   | kid2|   20
 3   |     | orp3|   30
 4   | 1   | kid4|   40
(4 rows)

Just like InnoDB, PostgreSQL uses multi-versioning for READ COMMITTED avoiding locking for DML read operations. And just like with InnoDB, the INSERT command of client B is blocked on the foreign key RI lookup, and the INSERT command of client A in step 3 will lead to deadlock and rollback of its transaction. On PostgreSQL we get more detailed explanation from the server, which confirms the behavior what we suspected also on InnoDB.

PyrrhoDB 5.2

We will run the following test in 2 concurrent terminal windows: Client A and Client B. Note that Pyrrho client runs in autocommit mode. BEGIN TRANSACTION start a transaction, but after transaction Pyrrho returns to autocommit mode. The isolation level is SNAPSHOT and the concurrency control is optimistic concurrency: the first transaction of interleaved transactions to COMMIT will win the competition.
Even if the concurrent transactions don’t explicitly touch the same rows, due to foreign key references they interfere each other’s rows implicitly. Due to optimistic concurrency control of Pyrrho this will not be noticed before the COMMIT phase, and the first one to COMMIT wins.

**Summary**

On our journey on experimenting with the foreign key RI lookups implemented in various DBMS products we have learned that without explicitly touching the same rows, foreign key references may cause concurrency conflicts, and also that there are different techniques that the products sort out the RI rules.

### 3.2 Concurrency Side-effects due to Triggers

Let’s keep the example minimalistic and forget the lessons for normalization.

Our example is not from “real life”, the purpose is only to demonstrate possible side-effects of triggers created by DBA and hidden from the application developers. The isolation level defined for the transaction will/may effect also operations of the triggers and lead to some mysterious problems.
The FOREIGN KEY of Child was dropped first to eliminate its potential effects on the trigger test.

```
ALTER TABLE Child
    DROP CONSTRAINT Child_Parent_FK;
COMMIT;
```

Then the following triggers were created in Command Editor setting st-sign (@) temporarily as the command terminator

```
CREATE TRIGGER Upd_Child AFTER UPDATE ON Child
REFERENCING NEW AS NEW
FOR EACH ROW
BEGIN
    UPDATE Parent P SET psum =
        (SELECT SUM(csum) FROM Child C WHERE C.pid = NEW.pid)
    WHERE P.pid = NEW.pid;
END @

CREATE TRIGGER Ins_Child AFTER INSERT ON Child
REFERENCING NEW AS NEW
FOR EACH ROW
BEGIN
    UPDATE Parent P SET psum =
        (SELECT SUM(csum) FROM Child C WHERE C.pid = new.pid)
    WHERE P.pid = new.pid;
END @

CREATE TRIGGER Del_Child AFTER DELETE ON Child
REFERENCING OLD AS OLD
FOR EACH ROW
BEGIN
    UPDATE Parent P SET psum =
        (SELECT SUM(csum) FROM Child C WHERE C.pid = old.pid)
    WHERE P.pid = old.pid;
END @
```
Then the test was carried out as follows:

```
CREATE TRIGGER Del_Child AFTER DELETE ON Child
REFERENCING OLD AS OLD
FOR EACH ROW
BEGIN
  UPDATE Parent P SET issue =
  (SELECT SUM(csum) FROM Child C WHERE C.pid = OLD.pid)
  WHERE P.pid = OLD.pid;
END;

UPDATE Parent P SET issue =
(SELECT SUM(csum) FROM Child C WHERE C.pid = OLD.pid)
WHERE P.pid = OLD.pid;
END;
```

So deadlock detected now in 2 seconds without waiting the lock timeout!
Oracle PL/SQL

Triggers adapted to Oracle:

CREATE OR REPLACE TRIGGER Upd_Child
AFTER INSERT OR UPDATE ON Child
REFERENCING NEW AS NEW
FOR EACH ROW
DECLARE   id
       NUMBER; BEGIN
   id := :NEW.pid;
   UPDATE Parent P SET psum =
       (SELECT SUM(csum) FROM Child C WHERE C.pid = id)
       WHERE P.pid = id;
END;

CREATE OR REPLACE TRIGGER Del_Child
AFTER DELETE ON Child
REFERENCING OLD AS OLD
FOR EACH ROW
DECLARE   id
       NUMBER; BEGIN
   id := :OLD.pid;
   UPDATE Parent P SET psum =
       (SELECT SUM(csum) FROM Child C WHERE C.pid = id)
       WHERE P.pid = id;
END;

On the insert test we get following error messages:

INSERT INTO Child (cid, pid, cs, csum)
VALUES (4, 1, 'kid4', 40);

Error report:
SQL Error: ORA-04091: table USER1.CHILD is mutating, trigger/function may not see it
ORA-06512: at "USER1.UPD_CHILD", line 5
ORA-04088: error during execution of trigger 'USER1.UPD_CHILD'
04091. 00000 - "table %s.%s is mutating, trigger/function may not see it"

*Cause:    A trigger (or a user defined plsql function that is referenced in
this statement) attempted to look at (or modify) a table that was
in the middle of being modified by the statement which fired it. *Action:
Rewrite the trigger (or function) so it does not read that table.

So, Oracle seems to be over-protecting in this case compared with the other DBMS products. The PL/SQL Language Reference manual (page 9-24) explains this as follows:
“A mutating table is a table that is being modified by an UPDATE, DELETE, or INSERT statement ... The session that issued the triggering statement cannot query or modify a mutating table. This restriction prevents a trigger from seeing an inconsistent set of data.”
SQL Server Transact-SQL

SQL Server has different syntax for triggers and it does not support row-level triggers at all, only command-level triggers.

CREATE TRIGGER Upd_Child ON Child AFTER UPDATE
AS BEGIN
    UPDATE Parent SET psum =
        (SELECT SUM(csum) FROM Child C WHERE C.pid = pid)
    WHERE Parent.pid = pid;
END;

CREATE TRIGGER Ins_Child ON Child AFTER INSERT
AS BEGIN
    UPDATE Parent SET psum =
        (SELECT SUM(csum) FROM Child C WHERE C.pid = pid)
    WHERE Parent.pid = pid;
END;

CREATE TRIGGER Del_Child ON Child AFTER DELETE
AS BEGIN
    UPDATE Parent SET psum =
        (SELECT SUM(csum) FROM Child C WHERE C.pid = pid)
    WHERE Parent.pid = pid;
END;
MySQL/InnoDB

We will create the following MySQL triggers, which will maintain in the \texttt{psum} column of the parent rows the sum of the \texttt{csum} columns of the corresponding child rows as follows:

```sql
CREATE TRIGGER Upd_Child AFTER UPDATE ON Child
FOR EACH ROW
BEGIN
  UPDATE Parent P SET psum =
    (SELECT SUM(csum) FROM Child C WHERE C.pid = old.pid)
    WHERE P.pid = old.pid;
END;

CREATE TRIGGER Ins_Child AFTER INSERT ON Child
FOR EACH ROW
BEGIN
  UPDATE Parent P SET psum =
    (SELECT SUM(csum) FROM Child C WHERE C.pid = new.pid)
    WHERE P.pid = new.pid;
END;

CREATE TRIGGER Del_Child AFTER DELETE ON Child
FOR EACH ROW
BEGIN
  UPDATE Parent P SET psum =
    (SELECT SUM(csum) FROM Child C WHERE C.pid = old.pid)
    WHERE P.pid = old.pid;
END;
```

And let’s test with the following scenario:

<table>
<thead>
<tr>
<th>Client A</th>
<th>Client B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SET AUTOCOMMIT = 0;</td>
<td>SET AUTOCOMMIT = 0;</td>
</tr>
<tr>
<td>SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</td>
<td>SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</td>
</tr>
<tr>
<td>SELECT * FROM Parent WHERE pid = 1;</td>
<td>set lock_wait_timeout = 300;</td>
</tr>
<tr>
<td></td>
<td>(either)</td>
</tr>
<tr>
<td></td>
<td>UPDATE Child SET pid = 2</td>
</tr>
<tr>
<td></td>
<td>WHERE cid = 3;</td>
</tr>
<tr>
<td></td>
<td>(or)</td>
</tr>
<tr>
<td></td>
<td>INSERT INTO Child (cid, pid, cs, csum) VALUES</td>
</tr>
<tr>
<td></td>
<td>(4, 1, 'kid4', 40);</td>
</tr>
</tbody>
</table>

© Martti Laiho 2014
Following listing has been captured from a test run:

```sql
-- client A
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.00 sec)

SELECT * FROM Parent WHERE pid = 1;
+--------------+----------+----------+
| pid | ps   | psum    |
+--------------+----------+----------+
|   1 | pa1  |    0    |
+--------------+----------+----------+
1 row in set (0.00 sec)

-- client B
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.00 sec)

set lock_wait_timeout = 300;
Query OK, 0 rows affected (0.00 sec)

INSERT INTO Child (cid, pid, cs, csum) VALUES (4, 1, 'kid4', 40);

-- client A
UPDATE Parent SET ps = 'new' WHERE pid = 1;
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

So, MySql/InnoDB succeeds gracefully of this test.

PostgreSQL PL/pgSQL

PostgreSQL has implemented triggers in its own way: First a function which is declared to return trigger need to be created, and the trigger to be created has to execute the function (referred as procedure). In PostgreSQL procedure concept is a kind of function. Triggers can be created on statement-level or row-level, and the same trigger can be used to control multiple events of the table.

Script used in our test, first eliminating the foreign key effect:

```
ALTER TABLE Child
    DROP CONSTRAINT Child_Parent_FK;

DROP TRIGGER Upd_Child ON Child;
```
DROP FUNCTION After_Upd_Child();

CREATE FUNCTION After_Upd_Child()
RETURNS trigger AS
$BODY$
DECLARE id INT;
BEGIN
  IF (TG_OP = 'DELETE')
  THEN id = OLD.pid;
  ELSE id = NEW.pid;
  END IF;
  UPDATE Parent SET psum =
    (SELECT SUM(csum)
     FROM Child C WHERE C.pid = id)
  WHERE Parent.pid = id;
  RETURN null;
END;
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER Upd_Child AFTER UPDATE OR DELETE OR INSERT ON Child
FOR EACH ROW
EXECUTE PROCEDURE After_Upd_Child();

PyrrhoDB (to be tested)
3.3 Referential Integrity Rules and Triggers

The ISO SQL standard defines the following Referential Integrity (RI) rules, which can be included as ON UPDATE and ON DELETE clauses of foreign key constraint of a child table as follows:

```
[<constraint name>] FOREIGN KEY <column(s)>
               REFERENCES <parent table> [(<unique column(s)>)]
[ON UPDATE { CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION }]
[ON DELETE  { CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION }]
```

The exciting semantics of these are explained on basic SQL courses, but all of these are not supported in all DBMS products as shown in table 3.3.1. Rules marked by “Y” stand for the default rules. Rules marked by the red colour have raised our interest in the comparison of products.

Table 3.3.1 Support of RI rules

<table>
<thead>
<tr>
<th>RULES</th>
<th>ISO SQL 2006</th>
<th>DE2 LUV 9.7</th>
<th>Oracle 11g1</th>
<th>SQL Server 2012</th>
<th>MySQL 5.6</th>
<th>PostgreSQL 9.2</th>
<th>Pyhno 5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON UPDATE RESTRICT</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ON UPDATE NO ACTION</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>ON UPDATE CASCADE</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ON UPDATE SET NULL</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ON UPDATE SET DEFAULT</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ON DELETE RESTRICT</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ON DELETE NO ACTION</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>ON DELETE CASCADE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ON DELETE SET NULL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ON DELETE SET DEFAULT</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Note: RESTRICT and NO ACTION provide almost same kind of restriction on the primary key change of parent row or deletion of the parent row, in case the parent has children in the child table, but for RESTRICT the check of children is done before the update or delete of the parent row, whereas in case of NO ACTION the check is done after processing the parent row.

Referential Integrity rules implemented by PL/SQL triggers

From Table 3.3.1 we surprisingly see that Oracle SQL does not support the ON UPDATE rules at all. Tom Kyte from Oracle explains this on his blog reasoning that “primary keys should never be updated, otherwise the table design is wrong”. Anyhow, for providing the missing built-in rules, PL/SQL Language Reference 11.1 manual p 9-35..39 presents triggers, which we have modified for our parent-child table pair as follows:

First we will recreate the tables:

```
DROP TABLE Child;
DROP TABLE Parent;
/
CREATE TABLE Parent {
  pid INT NOT NULL CONSTRAINT Parent_PK PRIMARY KEY,
  ps VARCHAR(20),
  psum INT DEFAULT 0 );
```
CREATE TABLE Child (
cid INT NOT NULL CONSTRAINT Child_PK PRIMARY KEY,
pid INT DEFAULT 0,
cs VARCHAR(20),
csum INT,
CONSTRAINT Child_Parent_FK FOREIGN KEY (pid) REFERENCES Parent (pid)
);
-- and insert into these some contents:
INSERT INTO Parent (pid, ps) VALUES (0,'default parent');
INSERT INTO Parent (pid, ps) VALUES (1,'pa1');
INSERT INTO Parent (pid, ps) VALUES (2,'pa2');
INSERT INTO Child (cid, pid, cs, csum) VALUES (1, 1,'kid1',10);
INSERT INTO Child (cid, pid, cs, csum) VALUES (2, 1,'kid2',20);
INSERT INTO Child (cid, pid, cs, csum) VALUES (3,NULL,'orphan3', 30);
SELECT * FROM Parent;
SELECT * FROM Child;
COMMIT;

Foreign Key Trigger for the Child Table for the INSERT RULE of RI

CREATE OR REPLACE TRIGGER Child_Parent_check
BEFORE INSERT OR UPDATE OF pid ON Child
FOR EACH ROW WHEN (new.pid IS NOT NULL)
-- Before row is inserted or pid is updated in Child table,
-- fire this trigger to verify that new foreign key value (pid)
-- is present in Parent table.
DECLARE
  Dummy INTEGER; -- Use for cursor fetch
  Invalid_parent EXCEPTION;
  Valid_parent EXCEPTION;
  Mutating_table EXCEPTION;
  PRAGMA EXCEPTION_INIT (Mutating_table, -4091);
CURSOR Dummy_cursor (Dn NUMBER) IS
  SELECT pid FROM Parent
  WHERE pid = Dn
  FOR UPDATE OF pid;
BEGIN
  OPEN Dummy_cursor (:new.pid);
  FETCH Dummy_cursor INTO Dummy;
  -- Verify parent key.
  -- If not found, raise user-specified error number & message.
  -- If found, close cursor before allowing triggering statement to complete:
  IF Dummy_cursor%NOTFOUND THEN
    RAISE Invalid_parent;
  ELSE
    RAISE valid_parent;
  END IF;
  CLOSE Dummy_cursor;
EXCEPTION
  WHEN Invalid_parent THEN
    CLOSE Dummy_cursor;
    RAISE_application_error(-20000, 'Invalid Parent' || ' Number' || TO_CHAR(:new.pid));
  WHEN Mutating_table THEN
    CLOSE Dummy_cursor;
  END;

© Martti Laiho 2014
-- Commands for testing the INSERT RULE controlled by the trigger:
ALTER TABLE Child DISABLE CONSTRAINT Child_Parent_FK;
INSERT INTO Child (cid, pid, cs) VALUES (4, 4, 40);
UPDATE Child SET pid = 5 WHERE pid = 1;
SELECT * FROM Child;
ROLLBACK;
-- After the test restoring the declarative INSERT RULE
-- controlled by the FOREIGN KEY:
ALTER TABLE Child ENABLE CONSTRAINT Child_Parent_FK;
ALTER TRIGGER Child_Parent_check DISABLE;
INSERT INTO Child (cid, pid, cs) VALUES (4, 4, 40);
UPDATE Child SET pid = 5 WHERE pid = 1;
SELECT * FROM Child;
ROLLBACK;

The “FOR UPDATE OF” clause of the cursor in the trigger for locking the parent row is necessary, since
triggers operate in the transaction context of the firing commands, i.e. on the isolation level of the
transaction. We can verify this by removing the “FOR UPDATE OF pid” clause from the trigger, and
running the following concurrency scenario:

**Session of Client 1:**

SQL> INSERT INTO Parent (pid, ps, psum) VALUES (4, 'temp parent', 0);
1 row created.
SQL> COMMIT;
Commit complete.
SQL> DELETE FROM Parent WHERE pid = 4;
1 row deleted.

**Session of Client 2:**

SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> INSERT INTO Child (cid, pid, cs) VALUES (4, 4, 40);
1 row created.

**UPDATE and DELETE RESTRICT Trigger for Parent Table**

-- The following trigger is defined on the Parent table to enforce
-- the UPDATE and DELETE RESTRICT referential action on the primary key
-- of the Parent table. Trigger using explicit cursor processing.

CREATE OR REPLACE TRIGGER pid_RESTRICT
BEFORE DELETE OR UPDATE OF pid ON Parent
FOR EACH ROW
-- Before row is deleted from Parent or primary key (pid) of Parent is updated,
-- check for dependent foreign key values in Child;
-- if any are found, roll back.
DECLARE
    Dummy INTEGER; -- Use for cursor fetch
    Children_present EXCEPTION;
    Children_not_present EXCEPTION;
-- Cursor used to check for dependent foreign key values.
CURSOR Dummy_cursor (Dn NUMBER) IS
    SELECT pid FROM Child WHERE pid = Dn;
BEGIN
    OPEN Dummy_cursor (:old.pid);
    FETCH Dummy_cursor INTO Dummy;
    /* Check for dependent foreign key values. */
    IF Found THEN
        /* Roll back transaction. */
        ROLLBACK;
    END IF;
END;
-- If dependent foreign key is found, raise user-specified
-- error number and message. If not found, close cursor
-- before allowing triggering statement to complete.
IF Dummy_cursor%FOUND THEN
  RAISE Children_present; -- Dependent rows exist
ELSE
  RAISE Children_not_present; -- No dependent rows exist
END IF;
CLOSE Dummy_cursor;
EXCEPTION
  WHEN Children_present THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20001, 'Children present for'
                         || ' Parent ' || TO_CHAR(:old.pid));
  WHEN Children_not_present THEN
    CLOSE Dummy_cursor;
END;

-- Commands for testing the trigger:
SELECT * FROM Parent;-- to verify the initial content
SELECT * FROM Child; -- to verify the initial content
UPDATE parent SET pid = 4 WHERE pid = 1;
SELECT * FROM Parent;-- to verify the result of UPDATE
SELECT * FROM Child; -- to verify the result of the trigger
ROLLBACK; -- to restore the initial content
DELETE Parent WHERE pid = 1;
SELECT * FROM Parent;-- to verify the result of DELETE
SELECT * FROM Child; -- to verify the result of the trigger
ROLLBACK; -- to restore the initial content
ALTER TRIGGER pid_RESTRICT DISABLE;

UPDATE and DELETE NO ACTION Trigger for Parent Table

-- The following trigger is defined on the Parent table to enforce
-- the UPDATE and DELETE RESTRICT referential action on the primary key
-- of the Parent table. Trigger using implicit cursor processing.
-- Replacing the AFTER by BEFORE the same works as RESTRICT trigger!

CREATE OR REPLACE TRIGGER pid_NO_ACTION
AFTER DELETE OR UPDATE OF pid ON Parent
FOR EACH ROW
-- Before row is deleted from Parent or primary key (pid) of Parent is updated,
-- check for dependent foreign key values in Child;
-- if any are found, roll back.
DECLARE
  Dummy INTEGER; -- Use for cursor fetch
Children_present EXCEPTION;
BEGIN
  SELECT COUNT(*) INTO Dummy FROM Child WHERE pid = :OLD.pid;
  IF SQL%ROWCOUNT > 0 THEN
    RAISE Children_present; -- Dependent rows exist
  END IF;
EXCEPTION
  WHEN Children_present THEN
    Raise_application_error(-20001, 'Children present for'
                         || ' Parent ' || TO_CHAR(:old.pid));
END;

-- Commands for testing the trigger:
SELECT * FROM Parent;-- to verify the initial content
SELECT * FROM Child; -- to verify the initial content
UPDATE parent SET pid = 4 WHERE pid = 1;
SELECT * FROM Parent;-- to verify the result of UPDATE
UPDATE and DELETE SET NULL Triggers for Parent Table

Note: For DELETE SET NULL rule we actually don’t need triggers, since the functionality is available as declarative rule using the ON DELETE SET NULL clause of the foreign key.

-- The following trigger is defined on the Parent table to enforce the
-- UPDATE and DELETE SET NULL referential action on the primary key
-- of the Parent table:
CREATE OR REPLACE TRIGGER pid_set_null
AFTER DELETE OR UPDATE OF pid ON Parent
FOR EACH ROW
-- Before row is deleted from Parent or primary key (pid) of Parent is updated,
-- set all corresponding dependent foreign key values in Child to NULL:
BEGIN
  IF UPDATING AND :OLD.pid != :NEW.pid OR DELETING THEN
    UPDATE Child SET Child.pid = NULL
    WHERE Child.pid = :old.pid;
  END IF;
END;

-- Commands for testing the trigger:
SELECT * FROM Parent;-- to verify the initial content
SELECT * FROM Child; -- to verify the initial content
UPDATE parent SET pid = 4 WHERE pid = 1;
SELECT * FROM Parent;-- to verify the result of UPDATE
SELECT * FROM Child; -- to verify the result of the trigger
ROLLBACK; -- to restore the initial content
DELETE Parent WHERE pid = 1;
SELECT * FROM Parent;-- to verify the result of DELETE
SELECT * FROM Child; -- to verify the result of the trigger
ROLLBACK; -- to restore the initial content
ALTER TRIGGER pid_SET_NULL DISABLE;

UPDATE and DELETE SET DEFAULT Trigger for Parent Table

-- The following trigger is defined on the Parent table to enforce the
-- UPDATE and DELETE to parent's pid to SET DEFAULT as referential action
-- to foreign keys of the children of manipulated parent row.
-- The creator of the trigger needs DBA privileges to access
-- the ALL_TAB_COLUMNS in the Oracle data dictionary.
CREATE OR REPLACE TRIGGER pid_SET_DEFAULT
AFTER DELETE OR UPDATE OF pid ON Parent
FOR EACH ROW
-- Before row is deleted from Parent or primary key (pid) of Parent is updated,
-- set all corresponding dependent foreign key values in Child to default value
-- of the column:
DECLARE
default_pid NUMBER := NULL;
BEGIN
  SELECT data_default INTO default_pid
  FROM ALL_TAB_COLUMNS
  WHERE column_name = 'Parent_Pid';
END;

-- Commands for testing the trigger:
SELECT * FROM Parent;-- to verify the initial content
SELECT * FROM Child; -- to verify the initial content
UPDATE parent SET pid = 4 WHERE pid = 1;
SELECT * FROM Parent;-- to verify the result of UPDATE
SELECT * FROM Child; -- to verify the result of the trigger
ROLLBACK; -- to restore the initial content
DELETE Parent WHERE pid = 1;
SELECT * FROM Parent;-- to verify the result of DELETE
SELECT * FROM Child; -- to verify the result of the trigger
ROLLBACK; -- to restore the initial content
ALTER TRIGGER pid_SET_DEFAULT DISABLE;
WHERE TABLE_NAME = 'CHILD' AND COLUMN_NAME = 'PID';
IF default_pid IS NOT NULL AND 
UPDATING AND :OLD.pid != :NEW.pid OR DELETING THEN 
UPDATE Child SET Child.pid = default_pid 
WHERE Child.pid = :old.pid;
END IF;
END;

-- Commands for testing the trigger:
SELECT * FROM Parent; -- to verify the initial content
SELECT * FROM Child; -- to verify the initial content
UPDATE parent SET pid = 4 WHERE pid = 1;
SELECT * FROM Parent; -- to verify the result of UPDATE
SELECT * FROM Child; -- to verify the result of the trigger
ROLLBACK; -- to restore the initial content
DELETE Parent WHERE pid = 1;
SELECT * FROM Parent; -- to verify the result of DELETE
SELECT * FROM Child; -- to verify the result of the trigger
ROLLBACK; -- to restore the initial content
ALTER TRIGGER pid_SET_DEFAULT DISABLE;

UPDATE CASCADE and DELETE CASCADE triggers for the Parent Table

Note: For DELETE CASCADE rule we actually don’t need triggers, since the functionality is available as declarative rule using the ON DELETE CASCADE clause of the foreign key.

The following triggers cascade1-cascade3 ensure that if the primary key pid in the Parent table is updated, a PL/SQL SEQUENCE generator Update_sequence is used to generate a unique value which is propagated to technical Update_id column of all the dependent child rows and to control updating of the foreign key value to the new pid value. For storing the Update_sequence value we create the PL/SQL package IntegrityPackage. It serves also as a simple example on how to use the PL/SQL package structure, which is a powerful extension of stored routines in PL/SQL.

Note: This set of triggers needs Oracle XE 11.

-- Create flag col:
ALTER TABLE Child ADD Update_id NUMBER;
/
-- Generate sequence number to be used as flag
-- for determining if update occurred on column:
CREATE SEQUENCE Update_sequence
INCREMENT BY 1 MAXVALUE 5000 CYCLE;
/
CREATE OR REPLACE PACKAGE IntegrityPackage AS
  Updateseq NUMBER;
END IntegrityPackage;
/
CREATE OR REPLACE PACKAGE BODY IntegrityPackage AS
END IntegrityPackage;
/
-- Before updating Parent table (this is a statement trigger),
-- generate new sequence number & assign it to public variable UPDATESEQ of
-- user-defined package named INTEGRITYPACKAGE:
--
CREATE OR REPLACE TRIGGER Parent_cascade1
BEFORE DELETE OR UPDATE OF pid ON Parent
 BEGIN
  IntegrityPackage.Updateseq := Update_sequence.NEXTVAL;
END;
Note: Parent_cascade1 does not work in Oracle XE 10.2, but requires version 11!

For each parent number in Parent that is updated, cascade update to dependent foreign keys in Child table.
Cascade update only if child row was not already updated by this trigger:

```sql
CREATE OR REPLACE TRIGGER Parent_cascade2
AFTER DELETE OR UPDATE OF pid ON Parent
FOR EACH ROW
BEGIN
  IF UPDATING THEN
    UPDATE Child
    SET pid = :new.pid,
        Update_id = Integritypackage.Updateseq -- from cascade1
    WHERE Child.pid = :old.pid AND Update_id IS NULL;
    /* Only NULL if not updated by 3rd trigger
    fired by same triggering statement */
  END IF;
  IF DELETING THEN
    BEFORE row is deleted from Parent,
    delete all rows from Child table whose pid is same as
    pid being deleted from Parent table:
    DELETE FROM Child
    WHERE Child.pid = :old.pid;
    -- Note: Typically, the code for DELETE CASCADE is combined with the
    -- code for UPDATE SET NULL or UPDATE SET DEFAULT to account for
    -- both updates and deletes.
  END IF;
END;
/
CREATE OR REPLACE TRIGGER Parent_cascade3
AFTER UPDATE OF pid ON Parent
BEGIN
  UPDATE Child
  SET Update_id = NULL
  WHERE Update_id = Integritypackage.Updateseq;
END;
/
```

Assuming that the UPDATE CASCADE and DELETE CASCADE triggers above for the Parent Table have been created, let’s test updating a parent’s pid:

Commands for testing the trigger:

- SELECT * FROM Parent; -- to verify the initial content
- SELECT * FROM Child; -- to verify the initial content
- UPDATE parent SET pid = 4 WHERE pid = 1;
- SELECT * FROM Parent; -- to verify the result of UPDATE
- SELECT * FROM Child; -- to verify the result of the trigger
- ROLLBACK; -- to restore the initial content
- DELETE Parent WHERE pid = 1;
- SELECT * FROM Parent; -- to verify the result of DELETE
- SELECT * FROM Child; -- to verify the result of the trigger
- ROLLBACK; -- to restore the initial content
- ALTER TRIGGER Parent_cascade1 DISABLE;
- ALTER TRIGGER Parent_cascade2 DISABLE;
- ALTER TRIGGER Parent_cascade3 DISABLE;
- UPDATE Parent SET pid = 4 WHERE pid = 1;
- SELECT * FROM Parent;
- SELECT * FROM Child;
- ROLLBACK; -- to ensure the initial content
Here we have a sample test run:

```
UPDATE Parent SET pid = 4 WHERE pid = 1;
1 rows updated.
```

```
SELECT * FROM Parent;

<table>
<thead>
<tr>
<th>PID</th>
<th>PS</th>
<th>PSUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>default parent</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>pa1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>pa2</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
SELECT * FROM Child;

<table>
<thead>
<tr>
<th>CID</th>
<th>PID</th>
<th>CS</th>
<th>CSUM</th>
<th>UPDATE_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>kid1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>kid2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>orphan3</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
```

In Parent_cascade2 trigger we saw also examples on use of event predicates UPDATING and DELETING. However, if we would need only the DELETE CASCADE rule for our foreign key of the Child table, then the following would be more effective solution:

```
-- DELETE CASCADE Trigger for the Parent Table
-- The following trigger on the Parent table enforces the
-- DELETE CASCADE referential action on the primary key of the Parent table:
-- Caution: This trigger does not work with self-referential tables
-- (tables with both the primary/unique key and the foreign key). Also,
-- this trigger does not allow triggers to cycle (such as, A fires B fires A).

CREATE OR REPLACE TRIGGER Parent_DELETE_CASCADE
AFTER DELETE ON Parent
FOR EACH ROW
-- Before row is deleted from Parent,
-- delete all rows from Child table whose pid is same as
-- pid being deleted from Parent table:
BEGIN
    DELETE FROM Child
    WHERE Child.pid = :old.pid;
END;
-- Commands for testing the trigger:
SELECT * FROM Parent;-- to verify the initial content
SELECT * FROM Child;-- to verify the initial content
DELETE Parent WHERE pid = 1;
SELECT * FROM Parent;-- to verify the result of DELETE
SELECT * FROM Child;-- to verify the result of the trigger
ROLLBACK;-- to restore the initial content
ALTER TRIGGER Parent_DELETE_CASCADE DISABLE;
```

**Compound Triggers**

In Oracle 11g new possibilities to trigger processing are implemented. These include:

- **FOLLOWS** clause which allows controlling the order of triggers fired by the same event and timing on the same object
- Compound triggers
and are explained in the web article at http://www.oracle-base.com/articles/11g/trigger-enhancements-11gr1.php

For DML events on a table, using a compound trigger we can program actions in different timing sections as presented by the article as follows:

```sql
CREATE OR REPLACE TRIGGER <trigger-name>
    FOR <trigger-action> ON <table-name>
    COMPOUND TRIGGER
    -- Local declarations
    <variable> <data type>; ..

BEFORE STATEMENT IS
BEGIN
    NULL; -- Do something here.
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
    NULL; -- Do something here.
END BEFORE EACH ROW;

AFTER EACH ROW IS
BEGIN
    NULL; -- Do something here.
END AFTER EACH ROW;

AFTER STATEMENT IS
BEGIN
    NULL; -- Do something here.
END AFTER STATEMENT;

END <trigger-name>;
/
```

Using the following compound trigger we can change the UPDATE CASCADE trigger into more compact form than using a PL/SQL package, but using still the SEQUENCE we created above:

```sql
CREATE OR REPLACE TRIGGER pid_CASCADE
    FOR UPDATE OF pid ON Parent
    COMPOUND TRIGGER
    -- Declaration Section
    -- Variables declared here have firing-statement duration.
    Updateseq NUMBER;

BEFORE STATEMENT IS
BEGIN
    Updateseq := Update_sequence.NEXTVAL;
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
    UPDATE Child
       SET  Update_id = Updateseq
       WHERE Child.pid = :old.pid AND Update_id IS NULL;
END BEFORE EACH ROW;

AFTER EACH ROW IS
```

© Martti Laiho 2014
BEGIN
    UPDATE Child
    SET pid = :new.pid, Update_id = NULL
    WHERE Update_id = Updateseq;
END AFTER EACH ROW;

END pid_CASCADE;
/

Testing the trigger as follows

UPDATE parent set pid = 4 where pid = 1;
1 rows updated.
SELECT * FROM Parent;

<table>
<thead>
<tr>
<th>PID</th>
<th>PS</th>
<th>PSUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>default parent</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>pa1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>pa2</td>
<td>0</td>
</tr>
</tbody>
</table>

SELECT * FROM Child;

<table>
<thead>
<tr>
<th>CID</th>
<th>PID</th>
<th>CS</th>
<th>CSUM</th>
<th>UPDATE_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>kid1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>kid2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>orphan3</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

ROLLBACK;
rollback complete.
ALTER TRIGGER pid_CASCADE DISABLE;

Compound triggers can be used to avoid the mutating table problem. Compared with the auxiliary PL/SQL package structure for passing state of public variables between the trigger series, the compound trigger technique is more compact and protects the variable values as private between the sections of the raised compound trigger instance.

**Exercise 3.1** Design a test to compare execution times of some declarative FOREIGN KEY with RI rules versus trigger based solution.

**Exercise 3.2** Compare pros and cons of declarative RI rules and trigger based RI rules.

**Study problem 3.1:**
When a foreign key consists of multiple columns, the RI rule triggers get more complicated. Also the declarative RI rules for multi-column foreign key definitions in SQL:1999 add more options by the MATCH clause

\[ \text{MATCH \{FULL | PARTIAL | SIMPLE\}} \]

This is discussed with examples in the article of Peter Gulutzan and Trudy Pelzer at

How this would change the RI triggers?
## Part 4  Comparing SQL/PSM and Implementations

### Stored procedures:

Table 4.1 compares stored procedure features in SQL/PSM and some implementations.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>ISO SQL</th>
<th>DB2 Express-C</th>
<th>Oracle XE</th>
<th>SQL Server</th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>Pyrro</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>default</td>
<td>default</td>
<td>default</td>
<td>default</td>
<td>default</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td>OUT</td>
<td>YES</td>
<td>YES</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>INOUT</td>
<td>YES</td>
<td>YES</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Invoking</td>
<td>CALL p()</td>
<td>CALL p()</td>
<td>EXEC p (...)</td>
<td>EXEC p @v,...</td>
<td>CALL p()</td>
<td>CALL p()</td>
<td>CALL p()</td>
</tr>
<tr>
<td>positional</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>by name</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>SPECIFIC name</td>
<td>YES</td>
<td>YES</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>BEGIN ATOMIC</td>
<td>YES</td>
<td>YES</td>
<td>compiled procedures</td>
<td>N/A</td>
<td>default?</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>atomic execution context</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>COMMIT</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>label</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;&lt;label&gt;&gt; label:</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>GOTO label</td>
<td>N/A</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Exception handlers</td>
<td>N/A</td>
<td>N/A</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Condition handlers</td>
<td>N/A</td>
<td>N/A</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>YES</td>
<td>YES</td>
<td>N/A</td>
<td>N/A</td>
<td>YES</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>EXIT</td>
<td>YES</td>
<td>YES</td>
<td>N/A</td>
<td>N/A</td>
<td>YES</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>REDO</td>
<td>YES</td>
<td>YES</td>
<td>N/A</td>
<td>N/A</td>
<td>YES</td>
<td>N/A</td>
<td>YES</td>
</tr>
</tbody>
</table>

The comparisons in Table 4.1 are mainly based on observations and are not complete. For example, passing of result sets is not covered at all.
**Stored functions:**

Comparing possible parameter modes the Standard does not accept OUT and INOUT modes for functions, whereas DB2, Oracle and PostgreSQL accept. Note that PostgreSQL does not make a difference between stored procedure and stored function concepts, including the invoking methods. Considering accepted DML statements, all accept SELECT statements, but only MySQL, PostgreSQL and Pyrrho accept other DML statements in scalar stored functions. Most products don’t accept COMMIT nor ROLLBACK statements in stored functions.

Table 4.2 compares features of scalar stored functions in SQL/PSM and implementations.

<table>
<thead>
<tr>
<th>ISO SQL</th>
<th>DB2 Express-C</th>
<th>Oracle XE</th>
<th>SQL Server</th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>Pyrrho</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>10.5</td>
<td>11.2</td>
<td>2012</td>
<td>5.6</td>
<td>9.2</td>
<td>5.1</td>
</tr>
<tr>
<td>6IW6-02-Four DB2SQLRefVo; PL/SQL manual</td>
<td>BDL 2008</td>
<td>refman-5.6-en; postgresql-9.0 manual5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Parameters**

- **IN**
  - default
  - N/A

- **OUT**
  - YES
  - N/A

- **INOUT**
  - YES
  - N/A

- **returns**
  - RETURNS
  - N/A

**SELECT**

- Yes
  - YES
  - N/A

**other DML**

- ?
  - N/A
  - yes 1)

**COMMIT**

- ?
  - N/A
  - yes 1)

**ROLLBACK**

- ?
  - N/A
  - yes 1)

**SPECIFIC name**

- YES
  - N/A
  - N/A

**polymorphism / overloading**

- YES
  - N/A
  - N/A

**invoking**

- f()
  - f()
  - f()
  - f()
  - f()
  - [CALL] f()
  - f()

1) yes, but then function cannot be used in SELECT statements
2) between member functions of a PL/SQL package

The comparisons in Table 4.2 are mainly based on observations and are not complete. For example, table-valued functions are not yet covered at all.

Note: Considering use of scalar functions in SQL statements, the function body should be atomic by nature.
Triggers:

Oracle triggers seem to have paved the way to the standard and were a model for some other implementations. However, the implementations have many nonstandard extensions and syntax differences (Silberschatz et al 2011). The following table tries to catch some of those differences and inspire readers for further studies.

Table 4.3 compares trigger definition in the standard and implementations

<table>
<thead>
<tr>
<th></th>
<th>ISO SQL</th>
<th>DB2 Express-C</th>
<th>Oracle XE</th>
<th>SQL Server</th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>Pytho</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>10.5</td>
<td>11.2</td>
<td>2012</td>
<td>5.6</td>
<td>9.2</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>SQLRefVol, PL/SQL manual</td>
<td>DML 2008</td>
<td>refman-5.6-en, postgresql-9.0 manual5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DML triggers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>privilege needed</td>
<td>TRIGGER</td>
<td>TRIGGER</td>
<td>CREATE TRIGGER</td>
<td>ALTER TABLE</td>
<td>TRIGGER</td>
<td>TRIGGER</td>
<td>admin</td>
</tr>
<tr>
<td>ON INSERT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE statement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BEFORE for each row</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AFTER for each row</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AFTER statement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>INSERTING</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>TG_OP=INSERT</td>
<td>N/A</td>
</tr>
<tr>
<td>ON UPDATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE statement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BEFORE for each row</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AFTER for each row</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AFTER statement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>UPDATING</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>UPDATE()</td>
<td>N/A</td>
<td>TG_OP=UPDATE</td>
<td>N/A</td>
</tr>
<tr>
<td>ON DELETE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE statement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BEFORE for each row</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AFTER for each row</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AFTER statement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DELETING</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>TG_OP=DELETE</td>
<td>n/a?</td>
<td></td>
</tr>
<tr>
<td>INSTEAD OF</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>NEW TABLE</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>INSERTED</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>OLD TABLE</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>DELETED</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>DDL triggers</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LOGON</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>DATABASE</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>cascading of triggers?</td>
<td>controlled</td>
<td>not allowed?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transaction demarkation</td>
<td>?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>N/A</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

An evolving option is the ordering of triggers fired by the same event. In DB2 the order is the create order, in Oracle the new FOLLOWS clause can specify the order, and in SQL Server the first and last trigger to be fired can be defined by system procedure sp_settriggerorder, whereas MySQL does not allow multiple triggers to be fired by the same event.
References


Chong R. F. et al, “DB2 Application Development”, DB2 on Campus Book Series, 2010 (available online at resources pages of BigDataUniversity.com)


© Martti Laiho 2014
Appendix 1  Result Set Processing by Cursor Mechanism

Cursor processing is one of the most essential technologies in data access. In this appendix we will discuss on server-side explicit and implicit cursors and client-side result set processing.

Explicit SQL Cursor Programming

On abstract level SQL language is based on processing of sets or rows, and we have learned that a SELECT statement produces a result set of rows, which may contain zero, one or multiple rows. Using SQL-client tools we have accustomed to get result sets immediately for viewing, but in fact the tool has to do a lot of work for fetching the presented rows. SQL-client tools have been programmed to use the same data access interfaces as application programs (which in terms of SQL are also SQL-clients), and access the services of the DBMS using SQL commands.

Application development by procedural 3GL languages, such as COBOL, C, etc. is based on “record-by-record” ideology of processing which is not compatible with the set oriented processing in SQL. This incompatibility is called impedance mismatch between 3GL and SQL, and it has been solved extending SQL language by declaration of cursor object and procedural statements presented in table A1.1 and properties of cursor objects listed in table A1.2. These form the explicit cursor programming paradigm, served on the server-side.

<table>
<thead>
<tr>
<th>Statement</th>
<th>explanation</th>
</tr>
</thead>
</table>
| DECLARE <cursorname> <sensitivity> <scrollability> 
CURSOR <holdability> 
FOR SELECT .. [FOR <updatability>] | defines the name, properties and the SELECT statement for generating the result set. |
| OPEN <cursorname> | this actually executes the SELECT statement |
| FETCH [<orientation> FROM ]<cursorname> INTO <host variables> 
where <orientation> is one of the following: NEXT, PRIOR, FIRST, LAST, ABSOLUTE n, or RELATIVE [-]n | As default fetches always the next available row from the result set of the executed SELECT statement, but in case the <scrollability> clause was defined as SCROLL, then other <orientation> options are possible. |
| UPDATE .. WHERE CURRENT OF <cursorname> | positional update the current row |
| DELETE .. WHERE CURRENT OF <cursorname> | positional delete the current row |
| CLOSE <cursorname> | finally releases the result set and temporary storage of the result set |

Table A1.1  Cursor statements for processing result set of a SELECT statement (Melton & Simon 2002)
### Property Explanation

<table>
<thead>
<tr>
<th>Property</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;cursorname&gt;</code></td>
<td>bind name of the cursor to bind statement to processing of the proper cursor as the application may have multiple cursors open at the same time</td>
</tr>
<tr>
<td><code>&lt;sensitivity&gt;</code></td>
<td>INSENSITIVE defines processing of a result set that is a snapshot of the selected rows, SENSITIVE defines dynamic fetching from the database row-by-row</td>
</tr>
<tr>
<td><code>&lt;scrollability&gt;</code></td>
<td>NO SCROLL defines forward only processing (default), SCROLL allows forward and backward scrolling in the result set</td>
</tr>
<tr>
<td><code>&lt;holdability&gt;</code></td>
<td>WITH HOLD allows cursor processing to continue after COMMIT of the transaction, WITHOUT HOLD closes an open cursor at COMMIT (usually the default)</td>
</tr>
<tr>
<td><code>&lt;updatability&gt;</code></td>
<td>UPDATE [OF <code>&lt;column list&gt;</code>] allows positional updates/deletes, READ ONLY does not allow positional updates/deletes (default)</td>
</tr>
</tbody>
</table>

Table A1.2. Cursor properties according to ISO SQL standard (Melton & Simon 2002)

The procedural access using explicit server-side cursor programming proceeds as follows: the DECLARE CURSOR statement defines named cursor and its FOR SELECT clause defines the SQL query to be used. The SQL query consists of a single SELECT statement, or more complicated set operations by UNION, INTERSECT, EXCEPT on SELECT statements.

OPEN statement instantiates the named cursor object and executes the query generating the result set as an INSENSITIVE snapshot or in case of SENSITIVE starts dynamic fetching of the rows to be selected. Using FETCH statements the application gets one row at a time from the cursor’s result set available for local processing by the application. The default orientation in FETCH is NEXT, and if this is the only orientation used, we call the cursor as forward-only cursor, which provides usually best performance for result set reading.

After OPEN and every FETCH the SQL-client has to check from diagnostics if the request was served successfully. Especially FETCH after the last row returns SQLCODE value 100.

If the cursor is defined as updateable, then the current row can be updated or deleted just based on the WHERE CURRENT OF `<cursorname>` clause in the UPDATE or DELETE statement.

The cursor object and its data structures are deleted by the explicit CLOSE statement, otherwise the cursor is closed implicitly when the current transaction ends, unless the `<holdability>` of the cursor is defined as WITH HOLD.

### SQL Cursor Programming in ESQL

The explicit SQL cursor programming for accessing the result set row-by-row has been the primary mean for reading data from databases in the Embedded SQL (ESQL) as defined in the ISO SQL standard since the early version SQL-86.

© Martti Laiho 2014
In ESQL the SQL statements are included in EXEC SQL statements embedded in the actual host language. Before compiling the source code the mixed source code is processed by a pre-compiler which typically prepares optimized execution plans into stored modules in the database and replaces the SQL statements by calls for these modules at execution time. The pre-compiled source code is then free of SQL-statements and ready for actual compiling and linking of the program.

The following program sample presents an example of cursor processing in ESQL

```sql
#include SQLDA
// - contains SQLCODE, SQLSTATE and other diagnostics
EXEC SQL BEGIN DECLARE SECTION;
// following variables can be referenced in SQL statements
int sumAccts;
int balance;
EXEC SQL END DECLARE SECTION;
sumAccts = 0;
balance = 0;
EXEC SQL DECLARE cur_account CURSOR FOR
    SELECT balance FROM Accounts;
EXEC SQL OPEN cur_account;
EXEC SQL FETCH cur_account INTO :balance;
while (SQLCODE = 0) {
    sumAccts = sumAccts + balance;
    EXEC SQL FETCH cur_account INTO :balance;
}
EXEC SQL CLOSE cur_account;
println (sumAccts);
...
```

Recently ESQL has been dropped from the standard, but which is still in use in many DBMS products, such as Oracle and DB2.

**Cursor Processing in DBMS Products**

All DBMS products support some kind of cursor processing, but typically not all cursor properties. Some products implement also extensions to ISO SQL cursor programming, for example isolation level or locking hints can be set for a cursor. Optimistic locking can be configured to manage concurrency on cursor updates in SQL Server and DB2.

Explicit SQL cursor processing continues as essential part in SQL procedural extensions, such as PL/SQL and SQL/PSM based dialects, as we have seen in this tutorial.

Following dummy script should be replaced by a single “SELECT SUM(balance) FROM Accounts” command, but the purpose of the script is just to demonstrate cursor programming in Transact-SQL like we had in our ESQL example, to help applying the cursor programming for some more challenging exercises

```sql
DECLARE @sumAccts INTEGER = 0;
DECLARE @balance INTEGER = 0;
DECLARE cur_account CURSOR FOR SELECT balance FROM Accounts;
OPEN cur_account;
FETCH cur_account INTO @balance;
WHILE @@FETCH_STATUS = 0 BEGIN
```

© Martti Laiho 2014
SET @sumAccts = @sumAccts + @balance;
FETCH cur_account INTO @balance;
END;
CLOSE cur_account;
DEALLOCATE cur_account;
SELECT @sumAccts

Note: @@FETCH_STATUS indicator of Transact-SQL returns the value 0 for successful fetch, but -1 or -2 for failed fetch, so this is not the same indicator as SQLCODE of ISO SQL. Explicit DEALLOCATE after CLOSE releases the data structure resources of the cursor.

On explicit cursor processing implementations the <holdability> property is interesting in the context of SQL transactions. As cursor processing is actually row-by-row processing of the resultset of a query, so an SQL transaction is the natural context of an opened cursor, and according to ISO SQL standard any open cursor is by default closed at the end of the transaction where the cursor was opened. The automatic closing can be changed by WITH HOLD definition, which means that the cursor need not be closed at COMMIT, but the cursor and its current row can be available in the next transaction and cursor processing may continue. Opposite to the standard, MS SQL Server databases behave like this as default, but can be configured to behave according to the standard by setting the parameter value of “Close Cursor on Commit Enabled” from False to True, while its default is to allow cursors to live from transaction to transaction.

Oracle PL/SQL

According to Oracle’s PL/SQL Language Reference manual, “A cursor is a name for a specific private SQL area in which information for processing the specific statement is kept. PL/SQL uses both implicit and explicit cursors”. On low level all DML statements, such as INSERT, UPDATE and DELETE are executed by implicit cursor processing. For explicit SQL cursor programming PL/SQL supports the model of ISO SQL with minor syntax differences and except some cursor properties, such as <holdability>. In addition to the basic SQL cursor model, Oracle’s PL/SQL provides many cursor based control structures which seem to have been models for the ISO SQL/PSM control structures, but with syntax differences. For example instead of FOR – END FOR the corresponding PL/SQL uses SQL Cursor FOR LOOP control structure with following syntax (PL/SQL Language Reference)

```sql
FOR <record name> IN (<SELECT statement>)
LOOP <processing of next row fetched to the record>;
END LOOP;
```

where <record name> is name of the automatically generated record structure which is cloned from the row structure of the SELECT statements resultset, and which record will contain one row at a time to be processed in the LOOP structure. Another Cursor FOR LOOP format of PL/SQL is Explicit Cursor FOR LOOP having following syntax (PL/SQL Language Reference)

```sql
DECLARE
    <cursor declaration>;
BEGIN
    FOR <record name> IN <cursor name>
        LOOP <processing of next row fetched to the record>;
    END LOOP;
END;
```

5 SQL Server provides also support for cursor level optimistic locking, based on timestamps or column values, applied in sequence of SQL transactions of a user transaction, sharing the same cursor.
Following test run on SQL*Plus tool with “simple” PL/SQL block, which does the same as our Transact-SQL example, demonstrates the SQL Cursor FOR LOOP and some rich features of PL/SQL:

```
SQL> SET SERVEROUTPUT ON;
DECLARE
  sumAccts INTEGER := 0;
BEGIN
  FOR Acct_rec IN (SELECT * FROM Accounts)
  LOOP sumAccts := sumAccts + Acct_rec.balance;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('sum(balance)=' || sumAccts);
END;
/
SQL> 2 3 4 5 6 7 8 9 sum(balance)=2700
PL/SQL procedure successfully completed.
```

In a PL/SQL block we cannot print results directly by SQL*Plus client, but using PUT_LINE method of DBMS_OUTPUT package we can put strings to an internal pipe of the package, and by setting serveroutput ON before the test run SQL*Plus will print contents of the pipe to us.

`Acct_rec` used in the FOR LOOP is a PL/SQL record structure which inherits the row structure and content from one row at a time from the resultset of the IN (SELECT ..) set of rows. The implicit cursor will be opened automatically, fetch of the next row on every loop is automatic as well as test for the end of the result set, and finally the cursor will be closed automatically.

For the actual result, we would get much better performance by simple SELECT as follows:

```
SQL> SELECT SUM(balance) AS "sum(balance)" FROM Accounts;
sum(balance) 2700
```

Both for implicit and explicit cursors, PL/SQL provides following cursor attributes attached at the end of the cursor name to be used as diagnostic information:

- `%FOUND` returns TRUE after successful row fetch (compare with SQLCODE = 0).
- `%NOTFOUND` returns TRUE after unsuccessful row fetch (compare with SQLCODE = 100).
- `%ROWCOUNT` returns number of rows fetched successfully this far.
- `%ISOPEN` returns TRUE if the cursor is still open.

For implicit cursors the cursor name is SQL, and it applies only immediately after the latest DML command, for example as follows:

```sql
UPDATE ... WHERE ..
IF SQL%NOTFOUND THEN ..
```

We will use this in our modified BankTransfer procedure in appendix 4.
**Client-Side Result Sets**

Cursor processing of query result sets is the only option for using SELECT commands in embedded SQL and ODBC API.

Modern object-oriented languages use APIs with have object wrappers for cursor processing, such as ResultSet objects in JDBC API, but still cursor mechanisms are used by the driver in the low-level dialog with the database server.

Both `<holdability>` and `<scrollability>` are possible in client-side cursor processing of middleware APIs, in which the resultset is copied to a client-side cache. For example the resultset object of JDBC API is not closed automatically at transaction end.

An extreme of the result set cache is used in Microsoft’s ADO.NET for the datasets, which can be disconnected from the database, managed locally, and then after re-connecting to the server can synchronize the contents back to the database using optimistic locking automatically.

The different cursor models of DBMS products and the used middleware APIs influence in the way of transaction programming, and what could be the best practice in different cases.

The procedural SQL extensions of products also support statement variations of cursor processing in scripts and stored routines and, whereas in JDBC cursor processing can be understood as internals of the resultset objects.
Appendix 2  Nested Transactions and Transactions with Savepoints

Nested Transactions

Nested transactions has been an inspiring topic in transaction research and literature for decades, and has even been implemented in some products, such as IDMS/SQL, MaxDB, and several research-oriented DBMS systems, but has not been implemented in the current mainstream DBMS products. Several nesting models have been defined, and perhaps most widely referred is the theoretical model defined by Moss in 1981. Avoiding too theoretic discussion, we simplify the concept as follows: A nested transaction is based on explicit transactions building a “well-formed” tree-hierarchy of (possibly) named subtransactions so that every subtransaction has an explicit beginning, can contain subtransactions, and either ends on commit or rollback of its own. In figure A.1 transaction A is the root of the nested transaction, parent transaction which includes as subtransactions B and C as its child transactions.

According to Moss, the database can be accessed only by the leaf-level subtransactions, while Gray’s model does not have this requirement. According to Gray the behavior of nested transactions can be summarized in the following rules:

Commit rule: Commit of a subtransaction depends on commits of its parents, and finally on commit of the root transaction.

Rollback rule: Rollback of the root or subtransaction will rollback also all its subtransactions. However, rollback of a subtransaction has no effect on its parent (Connolly & Begg 2010).

---

6 Rothermel and Mohan list following systems in 1989: ARGUS, Camelot, CLOUDS, LOCUS, and Eden.
7 Compare the hierarchy with elements of well-formed XML document.
Visibility rule: All objects held by a parent transaction are available to its subtransactions. All changes made by a committed subtransaction will become visible to its parent transaction and the following siblings [Moss 1987]. However, changes will not be visible to concurrent siblings.

Research papers on nested transactions tend to discuss on distributed transactions without expressing the difference between the multi-connection, distributed transactions and the local, single connection SQL transactions. Moss explicitly addresses distributed computing in his 1981 study presenting a theoretical locking scheme. Discussion on concurrent siblings refers to context of distributed transactions, and is not relevant for local SQL transactions. SQL transactions may still appear as subtransactions of a distributed transaction, but the context of SQL transactions is a single, open database connection, and the basic SQL language does not provide means for binding SQL statements inside a SQL transaction to different database connections.

A different model of nesting is provided by the protocol of autonomous transactions, which are independent on the possible outer transaction. We will discuss these in next Appendix of this paper.

The main advantage of nested transactions is the modular discipline in data access and faster recovery of subtransactions without side effects to other subtransactions (Connolly and Begg 2010). However, benefits on concurrency control reported by Moss are not obvious in case of single-connection local nested transactions. End of a subtransaction does not release its locks and the locks are inherited by the parent. As pointed out earlier, no full implementations exist in current mainstream DBMS products.

Nested Transactions in SQL Server

SQL Server is the only mainstream DBMS product which has a kind of nested transaction implementation. However, it does play according to the rules presented above. COMMIT of a subtransaction is processed only as a comment, but ROLLBACK in a subtransaction will ROLLBACK the whole transaction, which can be verified in the following example:

```
DROP TABLE T;
CREATE TABLE T (col VARCHAR(10));
GO
SET NOCOUNT ON;
BEGIN TRANSACTION A
  INSERT INTO T VALUES ('A1')
  SELECT 'A:', @TRANCOUNT
BEGIN TRANSACTION B
  INSERT INTO T VALUES ('B1')
  SELECT 'B:', @TRANCOUNT
COMMIT TRANSACTION B
BEGIN TRANSACTION C
  INSERT INTO T VALUES ('C1')
  SELECT 'C:', @TRANCOUNT
ROLLBACK TRANSACTION
BEGIN TRANSACTION D
  INSERT INTO T VALUES ('D1')
  SELECT 'D:', @TRANCOUNT
COMMIT TRANSACTION
BEGIN TRANSACTION
  INSERT INTO T VALUES ('A3')
  SELECT 'A:', @TRANCOUNT
COMMIT TRANSACTION A
SELECT * FROM T;
```
In T-SQL @@TRANSCOUNT indicates the current nesting level of transactions.

These are the results of the script for transaction A:

Results:

<table>
<thead>
<tr>
<th>Level</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

Msg 3902, Level 16, State 1, Line 20
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.

According to this test ROLLBACK TRANSACTION of Transaction C actually rolls back the whole transaction A and transaction D is processed as a new flat transaction.

The purpose of “nested transactions” in SQL Server is allow explicit start of transaction in stored procedures so that the procedure may optionally be called from an outer transaction.

**Savepoints and Partial Rollbacks**

*Note*  The transaction savepoints were not considered important topic in the SQL Transactions Handbook the purpose of which is to teach only the basics of SQL transactions. Savepoints are not the first techniques to be used in transaction programming and in SQL stored routines. The concept appears occasionally and is applied implicitly in ATOMIC compound statements.

The protocol of nested transactions is not included in the SQL standard, but some nesting behavior can be applied by transaction savepoints, which are defined in the standard and implemented in all mainstream products, although using varying syntaxes.

As a general rule the transactions should be designed to be as short as possible. However, some transactions may need many database actions and application logic. This transaction logic can include also multiple tentative parts, starting with named savepoint definitions and ending optionally with the rollback of all database modifications made after the named savepoint. According to the ISO SQL a savepoint is defined by following SQL statement

```
SAVEPOINT <savepoint name> ;
```
and the partial rollback of database modifications made in the transaction after a named savepoint (including “nested savepoints”, meaning the savepoint definitions after that) is done using the command

```
ROLLBACK TO SAVEPOINT <savepoint name> ;
```

This will essentially rollback effects of all statements executed after the named savepoint, and remove also that savepoint, so that it is no more available for rollbacking.

The scope in which the defined savepoints are available is the active SQL transaction, but according to the standard it is also possible to remove an existing savepoint and the nested savepoints after that in the transaction using a command

```
RELEASE SAVEPOINT <savepoint name>
```

which means that these savepoints are no more available for partial rollbacks in the current execution of the transaction. The RELEASE SAVEPOINT is not supported by all DBMS products as can be seen in Table A2.1.

**Table A2.1  SAVEPOINT protocol implementations in the mainstream products**

<table>
<thead>
<tr>
<th>Products:</th>
<th>Supported Savepoint statements of the standard or alternate syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO SQL standard statements:</td>
<td>SAVEPOINT &lt;name&gt;</td>
</tr>
<tr>
<td>DB2 SQL</td>
<td></td>
</tr>
<tr>
<td>Oracle PL/SQL</td>
<td>yes</td>
</tr>
<tr>
<td>MySQL/InnoDB</td>
<td>yes</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>yes</td>
</tr>
<tr>
<td>SQL Server T-SQL</td>
<td>SAVE TRANSACTION &lt;name&gt;</td>
</tr>
</tbody>
</table>

Savepoint protocol can be used to apply on logical level the functionality of nested transactions, if following the discipline of nested transactions. SAVEPOINT statement kind of starts a nested transaction, which can be rolled back by the ROLLBACK TO SAVEPOINT statement, kind of committed by the optional RELEASE SAVEPOINT statement. However, we don’t get the same structural discipline support for visibility or control of a proper nesting from the DBMS - a flat transaction with savepointing is still a flat transaction.

**A Savepoint Experiment using MySQL/InnoDB**

Since MySQL/InnoDB has full implementation of the ISO SQL savepoint protocol, we will use it in our technical savepoint experiment. First we will create a test table as follows:

```
SET AUTOCOMMIT=0;
DROP TABLE T;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(20));
-- Inserting initial contents into the table
INSERT INTO T (id, s) VALUES (1, 'value 1')  ;
COMMIT ;
```

Following is our authentic test run with commands and results:

```
mysql> -- See the initial contents in table T
```

© Martti Laiho 2014
mysql> SELECT * FROM T ;
+--------+
| id | s   |
+--------+
| 1   | value 1 |
+--------+
1 row in set (0.00 sec)

mysql> INSERT INTO T (id, s) VALUES (2, 'value 2') ;
Query OK, 1 row affected (0.00 sec)

mysql> SAVEPOINT savepoint1 ;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO T VALUES (3,'value 3') ;
Query OK, 1 row affected (0.00 sec)

mysql> UPDATE T SET s='updated value' WHERE id=2 ;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SAVEPOINT savepoint2 ;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO T VALUES (4,'value 4') ;
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM T ;
+--------+
| id | s        |
+--------+
| 1   | value 1  |
| 2   | updated value |
| 3   | value 3  |
| 4   | value 4  |
+--------+
4 rows in set (0.00 sec)

mysql> ROLLBACK TO SAVEPOINT savepoint2 ;
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO T VALUES (5,'value 5') ;
Query OK, 1 row affected (0.00 sec)

mysql> ROLLBACK TO SAVEPOINT savepoint1 ;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM T ;
+--------+
| id | s   |
+--------+
| 1   | value 1 |
| 2   | value 2 |
+--------+
2 rows in set (0.00 sec)

mysql> RELEASE SAVEPOINT savepoint1 ;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO T VALUES (6,'the latest row') ;
Query OK, 1 row affected (0.00 sec)

mysql> -- Rollback to savepoints ?
mysql> ROLLBACK TO SAVEPOINT savepoint1 ;
ERROR 1305 (42000): SAVEPOINT savepoint1 does not exist
mysql> ROLLBACK TO SAVEPOINT savepoint2 ;
ERROR 1305 (42000): SAVEPOINT savepoint2 does not exist
mysql> SELECT * FROM T ;
<table>
<thead>
<tr>
<th>id</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>value 1</td>
</tr>
<tr>
<td>2</td>
<td>value 2</td>
</tr>
<tr>
<td>6</td>
<td>the latest row</td>
</tr>
</tbody>
</table>

3 rows in set (0.00 sec)

mysql> -- Let's Rollback the whole transaction
mysql> ROLLBACK ;
Query OK, 0 rows affected (0.02 sec)

Exercise A2.1  Apply our experiment to Oracle XE

Exercise A2.2  Verify that locks acquired by commands which have been rolled back have not been released by the ROLLBACK TO SAVEPOINT command.

References

Connolly T., Begg C., “DATABASE SYSTEMS”, 5th edition, Addison-Wesley, 2010


Appendix 3  Autonomous Transactions

Autonomous transactions is a transaction protocol extension available both in Oracle PL/SQL and in SQL PL of DB2 LUW since version 9.7. Autonomous transaction is like a subtransaction in nested transaction, with the difference that if it will be committed it will not be rolled back by the ROLLBACK of the possible outer transaction. Autonomous transactions are allowed in procedures, functions and triggers. In PL/SQL the protocol extends these to allow COMMIT even in functions and triggers. In DB2 the AUTONOMOUS clause generates implicit COMMIT at the end of the routine.

Autonomous transactions a mainly used for tracing transactions which may get rolled back. In the following we will experiment with this using the myTrace table and our BankTransfer procedure.

DB2 SQL PL

We start experimenting with the DB2 SQL PL implementation of autonomous transactions, first creating the tracing table, then creating tracing procedure with autonomous transaction of its own.

```sql
DROP TABLE myTrace;
CREATE TABLE myTrace (
    t_no     INT,
    t_user   CHAR(20),
    t_date   DATE,
    t_time   TIME,
    t_proc   VARCHAR(16),
    t_what   VARCHAR(30)
);

CREATE OR REPLACE PROCEDURE myTraceProc (IN p_app VARCHAR(30),
                                          IN p_step INT,
                                          IN p_txt VARCHAR(30))
LANGUAGE SQL AUTONOMOUS -- for autonomous transaction!
BEGIN
    INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
    VALUES (p_step, user, current date, current time, p_app, p_txt);
END @
```

We will then re-create the BankTransfer procedure including tracing steps to be registered into the myTrace table using calls to myTraceProc.

```sql
CREATE OR REPLACE PROCEDURE BankTransfer (IN fromAcct INT,
                                            IN toAcct   INT,
                                            IN amount   INT,
                                            OUT msg     VARCHAR(100))
LANGUAGE SQL MODIFIES SQL DATA
PL: BEGIN
    DECLARE acct INT;
    DECLARE EXIT HANDLER FOR NOT FOUND
    BEGIN ROLLBACK;
        SET msg = CONCAT('missing account ', CAST(acct AS VARCHAR(10)));
    END;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN ROLLBACK;
        SET msg = CONCAT('negative balance (?) in ', fromAcct);
    END;
    SET acct = fromAcct;
    CALL myTraceProc ('BankTransfer',1,'finding and slocking fromAcct');
    SELECT acctID INTO acct FROM Accounts WHERE acctID = fromAcct ;
```
CALL myTraceProc ('BankTransfer',2,'updating fromAcct');
UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
-- CALL SLEEP(15);
SET acct = toAcct;
CALL myTraceProc ('BankTransfer',3,'finding and slocking toAcct');
SELECT acctID INTO acct FROM Accounts WHERE acctID = toAcct;
CALL myTraceProc ('BankTransfer',4,'updating toAcct');
UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;
CALL myTraceProc ('BankTransfer',5,'committing');
COMMIT;
SET msg = 'committed';
END P1 @
GRANT EXECUTE ON myTraceProc TO PUBLIC @

And finally we will verify how these behave. Will the autonomous transactions get committed even when the outer transaction is rolled back?

--testing the autonomous transactions
db2 -c -t
connect to testdb;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
SELECT * FROM Accounts;
COMMIT;
CALL BankTransfer (101, 202, 100, ?); -- normal case
SELECT * FROM myTrace;
DELETE FROM myTrace;
SELECT * FROM Accounts;

CALL BankTransfer (101, 202, 3000, ?); -- violating the CHECK constraint
SELECT * FROM myTrace;
DELETE FROM myTrace;
SELECT * FROM Accounts;

CALL BankTransfer (101, 999, 100, ?); -- invalid account number
SELECT * FROM myTrace;
DELETE FROM myTrace;
SELECT * FROM Accounts;

Exercise 3.1 Verify the results of the autonomous transactions using the scripts above.

Oracle XE PL/SQL

For the PL/SQL we use the myTrace table we created for Oracle earlier. The format of PL/SQL myTraceProc differs from DB2 SQL PL, as follows

CREATE OR REPLACE PROCEDURE myTraceProc (p_app VARCHAR2, p_step INT, p_txt VARCHAR2)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO myTrace (t_no, t_user, t_date, t_proc, t_what)
  VALUES (p_step, user, current_date, p_app, p_txt);
  COMMIT;
END;
GRANT EXECUTE ON myTraceProc TO PUBLIC;

Exercise 3.2 Apply the DB2 experimenting steps above to Oracle XE.
Appendix 4  Calling Stored Procedures from Java Code

To extend our theme of demonstrating the Java/JDBC accesses started in Appendix 2 of the “SQL Transactions” handbook, let’s see a minimalistic example of a Java program applied to calling a modified BankTransfer procedure of MySQL and Oracle (see below). Main difference between the “stand-alone” stored procedures presented in our tutorial and version we now will be using is that instead of the procedure the explicit commits and rollbacks are requested by the application which starts the transactions.

The challenging exercise of how to apply this to stored procedures on other products, is left to the readers.

/*  DBTechNet Concurrency Lab 2014-06-22 Martti Laiho  
BankTransferProc.java

Updates:
BankTransfer.java
2.0 2008-05-26 ML preventing rollback by application after SQL Server deadlock
2.1 2012-09-24 ML restructured for presenting the Retry Wrapper block
2.2 2012-11-04 ML exception on non-existing accounts
2.3 2014-03-09 ML TransferTransaction returns 1 for retry, 0 for OK, < 0 for error
BankTransferProc.java
1.0 2014-06-22 ML modified BankTransferProc for testing with stored procedures

*******************************************************************************/
import java.io.*;
import java.sql.*;
public class BankTransferProc {
    public static void main (String args[]) throws Exception
    {
        System.out.println("BankTransferProc version 1.0");
        if (args.length != 6) {
            System.out.println("Usage:" +
                "BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%");
            System.exit(-1);
        }
        java.sql.Connection conn = null;
        boolean sqlServer = false;
        int counter = 0;
        int retry = 0;
        String driver = args[0];
        String URL = args[1];
        String user = args[2];
        String password = args[3];
        int amount = 100;
        int fromAcct = Integer.parseInt(args[4]);
        int toAcct = Integer.parseInt(args[5]);

        // SQL Server's explicit transactions will require special treatment
        if (URL.substring(5,14).equals("sqlserver")) {
            sqlServer = true;
        }
        // register the JDBC driver and open connection
        try {
            Class.forName( driver );
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
    }
}
System.exit(-1); // exit due to a driver problem
}
try {
    conn = DriverManager.getConnection(URL, user, password);
} catch (SQLException ex) {
    System.out.println("URL: "+ URL);
    System.out.println("** Connection failure: "+ ex.getMessage() + 
        \\
        SQLSTATE: " + ex.getSQLState() + 
        SQLcode: " + ex.getErrorCode());
    System.exit(-1);
}
do {
    // Retry wrapper block of TransferTransaction ---------------
    if (counter++ > 0) {
        System.out.println("retry "+ counter);
        if (sqlServer) {
            System.out.println("Connection closed");
            conn = java.sql.DriverManager.getConnection(URL, user, password);
            conn.setAutoCommit(true);
        }
    }
    retry = TransferTransaction(conn, fromAcct, toAcct, amount, sqlServer);
    if (retry == 1) {
        long pause = (long) (Math.random () * 1000); // max 1 sec.
        // just for testing:
        System.out.println("Waiting for "+pause+ " mseconds before retry");
        Thread.sleep(pause);
    } else
    if (retry < 0)
        System.out.println(" Error code: " + retry + ", cannot retry.");
} while (retry == 1 && counter < 10); // max 10 retries
// end of the Retry wrapper block -------------------------------
conn.close();
System.out.println("\nEnd of Program. ");
}

class BankTransaction {
    // Transaction class for transferring money
    public static int TransferTransaction (Connection conn,
            int fromAcct, int toAcct, int amount,
            boolean sqlServer)
            throws Exception {
        String SQLState = "*****";
        String msg = "";
        String errMsg = "";
        int rc = 0; // retrun code
        int retry = 0;
        try {
            conn.setAutoCommit(false); // transaction begins
            conn.setTransactionIsolation(
                    Connection.TRANSACTION_SERIALIZABLE);
            msg = "";
            // CallableStatement cstmt =
            // conn.prepareCall("CALL BankTransfer(?,?,?,?)");
            cstmt.setInt(1, fromAcct);
            cstmt.setInt(2, toAcct);
            cstmt.setInt(3, amount);
            cstmt.setInt(4, 0);
            cstmt.setString(5, msg);
            cstmt.registerOutParameter(4, java.sql.Types.INTEGER);
        }
    }
cstmt.registerOutParameter(5, java.sql.Types.VARCHAR);
cstmt.executeUpdate();
rc = cstmt.getInt(4);
switch (rc) {
case -1:
    msg = cstmt.getString(5);
    System.out.println("** procedure msg: " + msg);
    conn.rollback();
    break;
case 0:
    conn.commit();
    break;
case 1:
    msg = cstmt.getString(5);
    System.out.println("** procedure msg: " + msg);
    conn.rollback(); // needed for Oracle
    break;
default:
    break;
}
cstmt.close();
retry = rc;
}
catch (SQLException ex) {
    try {
        errMsg = 
            "\nSQLException:
            " + SQLState + "SQLState: " + SQLState;
        errMsg = errMsg + "Message: " + ex.getMessage();
        errMsg = errMsg + "Vendor: " + ex.getErrorCode() + "\n";
        ex = ex.getNextException();
    }
    catch (Exception e) {
        System.out.println("SQLException handling error: " + e);
        retry = -1; // This is reserved for potential exception handling
    } // SQLException
    catch (Exception e) {
        System.out.println("Some java error: " + e);
        retry = -1; // This is reserved for potential other exception handling
    } // other exceptions
    finally { return retry; }
}

The stored procedure declaration signature (name and parameter list) can define generic interface for calls by applications, but the signature can be applied for different procedure implementations on different DBMS products. In the following we test this using SQL/PSM based MySQL/InnoDB and Oracle’s PL/SQL procedures.

Instead of explicit COMMIT and ROLLBACK statements, we add a new OUT parameter rc in the parameter list to pass the information to the control variable retry in our application code.
Using MySQL/InnoDB

The BankTransfer procedure modified for MySQL/InnoDB as follows:

```
delimiter !
DROP PROCEDURE if exists BankTransfer !
CREATE PROCEDURE BankTransfer (IN fromAcct INT,
    IN toAcct INT,
    IN amount INT,
    OUT rc INT,
    OUT msg VARCHAR(100))
LANGUAGE SQL MODIFIES SQL DATA
P1: BEGIN
    DECLARE acct INT;
    DECLARE EXIT HANDLER FOR NOT FOUND
        BEGIN ROLLBACK;
        SET msg = CONCAT('missing account ', CAST(acct AS CHAR));
        SET rc = -1;
        END;
    DECLARE EXIT HANDLER FOR SQLSTATE '40001' -- deadlock
        BEGIN ROLLBACK;
        SET msg = 'Deadlock';
        SET rc = 1;
        END;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
        BEGIN GET DIAGNOSTICS CONDITION 1 @p1 = MESSAGE_TEXT,
           @p2 = RETURNED_SQLSTATE;
        SET msg = CONCAT(@p1, ', SQLSTATE=');
        SET msg = CONCAT(msg, @p2);
        SET msg = CONCAT(msg, ', acct=');
        SET msg = CONCAT(msg, acct);
        ROLLBACK;
        SET rc = -1;
        END;
    SET acct = fromAcct;
    SELECT acctID INTO acct FROM Accounts WHERE acctID = fromAcct ;
    UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
    SELECT SLEEP(15) INTO @dummy; -- just for synchronizing -- concurrent session in deadlock test
    SET acct = toAcct;
    SELECT acctID INTO acct FROM Accounts WHERE acctID = toAcct ;
    UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;
    COMMIT;
    SET msg = 'committed';
    SET rc = 0;
END P1 !
delimiter ;
-- allow execute of the procedure to any login user
GRANT EXECUTE ON BankTransfer TO PUBLIC;
```

and test scripts:

```
-- session A
-- 'Missing account'
CALL BankTransfer (101, 999, 100, @rc, @msg);
SELECT @rc, @msg;

-- 'Testing CHECK constraint by our triggers'
CALL BankTransfer (101, 202, 3000, @rc, @msg);
```
SELECT @rc, @msg;

-- Concurrency test, session A
-- 'As stand-alone this test run should be OK';
CALL BankTransfer (101, 202, 100, @rc, @msg);
SELECT @rc, @msg;

-- Concurrent session B
CALL BankTransfer (202, 101, 100, @rc, @msg);
SELECT @rc, @msg;

The scripts for running the application can be copied from Appendix 2 of the “SQL Transactions” handbook. Only the program name need to be changed. The SLEEP function in the procedure can be used for concurrency tests of the application code, but after successful tests this could be commented out from the re-created procedure.

# allowing access to user1
mysql -u root mysql
SELECT user, host FROM mysql.user;
-- if user1 is missing then create user and set the password
CREATE USER 'user1'@'localhost';
SET PASSWORD FOR 'user1'@'localhost' = PASSWORD('sql');
--
GRANT ALL ON testdb.* TO 'user1'@'localhost';
EXIT;

# First window:
cd $HOME/Java
export CLASSPATH=.:/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/testdb
export user=user1
export password=sql
export fromAcct=101
export toAcct=202
java BankTransferProc $driver $URL $user $password $fromAcct $toAcct

# Second window:
cd $HOME/Java
export CLASSPATH=.:/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/testdb
export user=user1
export password=sql
export fromAcct=202
export toAcct=101
java BankTransferProc $driver $URL $user $password $fromAcct $toAcct

**Using Oracle PL/SQL**
The BankTransfer procedure modified for Oracle XE PL/SQL. In this version we test the if the UPDATE statements will not find the row to be updated, by testing the %NOTFOUND attribute of implicit cursor’s variable SQL and raising the NO_DATA_FOUND exception explicitly by RAISE statement if %NOTFOUND is true. For synchronizing competing sessions in our concurrency tests we use the sleep method of PL/SQL package DBMS_LOCK.

CREATE OR REPLACE PROCEDURE BankTransfer
(fromAcct IN INT,
toAcct IN INT,
amount IN INT,
rc OUT INT,
msg OUT VARCHAR )
IS
acct INT;
ernum NUMBER;
mesg VARCHAR2(200);
BEGIN
acct := fromAcct;
UPDATE Accounts SET balance = balance - amount
WHERE acctID = fromAcct;
IF SQL%NOTFOUND THEN RAISE NO_DATA_FOUND; END IF;
DBMS_LOCK.sleep(15); -- 15 sec pause for deadlock testing
acct := toAcct;
UPDATE Accounts SET balance = balance + amount
WHERE acctID = toAcct;
IF SQL%NOTFOUND THEN RAISE NO_DATA_FOUND; END IF;
mesg := 'committed';
rc := 0;
EXCEPTION
WHEN NO_DATA_FOUND THEN
   mesg := 'missing account ' || TO_CHAR(acct);
   rc := -1;
WHEN OTHERS THEN
   ernum := SQLCODE;
   mesg := SUBSTR(SQLERRM, 1, 200);
   mesg := mesg || ' SQLcode=' || TO_CHAR(ernum); -- ???
   IF ( ernum = -60 OR -- deadlock
       ernum = -8177 ) -- cannot serialize
   THEN rc := 1;
   ELSE rc := -1;
   END IF;
END;
/
GRANT EXECUTE ON BankTransfer TO PUBLIC;

And the concurrent scripts

# First window:
cd $HOME/Java
export CLASSPATH=.:/opt/jdbc-drivers/ojdbc6.jar
export driver=oracle.jdbc.driver.OracleDriver
export URL=jdbc:oracle:thin:@localhost:1521:XE
export user=user1
export password=sql
export fromAcct=101
export toAcct=202
java BankTransferProc $driver $URL $user $password $fromAcct $toAcct

# Second window:
cd ${HOME}/Java
export CLASSPATH=.:/opt/jdbc-drivers/ojdbc6.jar
export driver=oracle.jdbc.driver.OracleDriver
export URL=jdbc:oracle:thin:@localhost:1521:XE
export user=user1
export password=sql
export fromAcct=202
export toAcct=101
java BankTransferProc $driver $URL $user $password $fromAcct $toAcct

End of Program.
Passing ResultSet to the Caller

Stored procedures can return even multiple resultsets to the caller. Next, just as a “proof of concept”, we will experiment with getting resultsets from a MySQL stored procedure, using a minimalistic procedure accessing the sample tables which we have used in our previous experiments:

First we will create a sample procedure

```
DELIMITER !
CREATE PROCEDURE getResultSets()
BEGIN
    SELECT acctID, balance FROM Accounts;
    SELECT t_time, t_what FROM myTrace;
END !
DELIMITER ;
```

and test it as follows

```
mysql> CALL getResultSets () ;
+-----------------+-----------------+
| acctID | balance |
+-----------------+-----------------+
|    101 |    1000 |
|    202 |    2000 |
+-----------------+-----------------+
2 rows in set (0.00 sec)

+-----------------+-------------+
| t_time | t_what      |
+-----------------+-------------+
| 19:22:01 | Hello MySQL |
+-----------------+-------------+
1 row in set (0.01 sec)
```

Query OK, 0 rows affected (0.01 sec)

mysql>

For experimenting the access of the resultsets by Java code from the called procedure we use following Java program

```
/*  DBTechNet / Martti Laiho 
Sample Java program accessing the resultsets returned by a sample procedure */ 

import java.sql.*;

class getResultSetTest {
    public static void main( String args[] ) {
        System.out.println("getResultSetTest version 1.0\n");
        if (args.length != 4) {
            System.out.println("Usage:" +
                "getResultSet <driver> <URL> <user> <password> ");
            System.exit(-1);
        }
        String driver   = args[0];
        String URL      = args[1];
        String user     = args[2];
        String password = args[3];
        Connection con  = null;
        try { Class.forName( driver );
            System.out.println("ClassNotFoundExce
```

© Martti Laiho 2014
try { con = DriverManager.getConnection(URL, user, password); // start transaction
   con.setAutoCommit(false); // transaction begins
   con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
   // CallableStatement cstmt = con.prepareCall("{CALL getResultSets}");
   cstmt.execute();
   ResultSet rsl = cstmt.getResultSet();
   System.out.println ("Accounts:
acctID | balance
------------------
" + rsl.getString("acctID") + " | " + rsl.getString("balance"));
}
rs1.close();
cstmt.getMoreResults();
System.out.println ("myTrace:
time | text
------------------
" + rs2.getString("t_time") + " | " + rs2.getString("t_what"));
}
rs2.close();
cstmt.close();
con.rollback();
con.close();
} catch(SQLException ex) {
   System.err.println("SQLException: " + ex.getMessage());
}
}

Below are results from a sample test run

export CLASSPATH=.:/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export DRIVER=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/testdb
export USER=user1
export PASSWORD=sql
java ResultSetTest $DRIVER $URL $USER $PASSWORD

getResultSet version 1.0

Accounts:
acctID | balance
----------
101    | 1000
202    | 2000

myTrace:
time | text
----------
19:22:01 | Hello MySQL

Exercise 4.1  Will this work with other DBMS products?

Passing Oracle Cursor as an OUT Parameter

On abstract level SQL is about set processing, but on lower level result sets are accessible by cursor processing. In PL/SQL we can use pointer variables of REF CURSOR type, by which procedure can pass
data structures of an opened explicit cursor to the caller. The JDBC standard does not have means for
direct use of a cursor to be passed to resultset objects of the JDBC interface, but Oracle has extended its
Java classes and JDBC drivers to support this explicit accessing. In the following experiment we bypass
the need of these extended classes by tricks found from various Internet sources using the Open Java
and Oracle’s Thin JDBC driver.

First we create a sample PL/SQL procedure which opens a cursor and passes pointer to the cursor’s data
structures as OUT parameter to the calling application:

```sql
CREATE OR REPLACE
PROCEDURE getCursor (cursor OUT SYS_REFCURSOR) AS
BEGIN
  OPEN cursor FOR
    SELECT acctID, balance
    FROM   Accounts
    ORDER BY acctID;
END ;
```

Following is our simple program testing the use of the procedure

```java
/* DBTechNet / Martti Laiho
Sample Java program accessing resultset which a stored PL/SQL procedure has
instantiated by explicitly opened cursor and passed by cursor pointer to the caller.
Since JDBC does not support explicit accessing the cursor, it will be accessed as
a generic object and casted to ResultSet object.
*/
import java.sql.*;

class getCursorTest {
  public static void main( String args[] ) {
    System.out.println("getCursorTest version 1.0\n");
    if (args.length != 4) {
      System.out.println("Usage:
    getCursorTest <driver> <URL> <user> <password> ");
      System.exit(-1);
    }
    String driver   = args[0];
    String URL      = args[1];
    String user     = args[2];
    String password = args[3];
    Connection con  = null;
    try {
      Class.forName( driver );
    } catch(java.lang.ClassNotFoundException e) {
      System.err.println("ClassNotFoundException: "+ e.getMessage());
      System.exit(-1); // exit due to a driver problem
    }
    try {
      con = DriverManager.getConnection(URL,user,password);
      // start transaction
      con.setAutoCommit(false); // transaction begins
      con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
      //
      System.out.println( "acctID | balance\n------------------");
      CallableStatement cstmt = con.prepareCall("CALL getCursor (?)");
      cstmt.registerOutParameter(1, -10); // -10 stands for OracleTypes.CURSOR
      cstmt.execute();
      ResultSet rs = (ResultSet) cstmt.getObject(1);
    }
  }
}
```

© Martti Laiho 2014
while (rs.next()) {
    System.out.println(rs.getString("acctID") + " | " + rs.getString("balance"));
}  
rs.close();
cstmt.close();
con.rollback();
con.close();
}  
catch(SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
}

Script for test runs:

```
export CLASSPATH=./opt/jdbc-drivers/ojdbc6.jar  
extport DRIVER=oracle.jdbc.driver.OracleDriver  
extport URL=jdbc:oracle:thin:@localhost:1521:XE  
extport USER=user1  
extport PASSWORD=sql  
java getCursorTest $DRIVER $URL $USER $PASSWORD  
```

```
student@debianDB:~/Java$ java getCursorTest $DRIVER $URL $USER $PASSWORD  
getCursorTest version 1.0  
acctID | balance  
--------|--------  
101    | 1000  
202    | 1400  
student@debianDB:~/Java$`
Appendix 5  External Routines

This is placeholder for the appendix on external routines, draft of which is available at www.dbtechnet.org/papers/ExternalRoutines.pdf
Index

%FOUND; 87
%ISOPEN; 87
%NOTFOUND; 87, 102
%ROWCOUNT; 87
@@FETCH_STATUS; 85
@@TRANCOUNT; 91
action granularity; 14
Ada; 28
AFTER trigger; 15
alias name; 13
anonymous block; 29, 31
atomic compound statement; 12
AUTONOMOUS clause; 95
autonomous transaction; 95
BEFORE trigger; 15
BEGIN ATOMIC; 12
cascading triggers; 16
CHECK constraint; 43
CLI; 2
client-side cache; 88
command terminator; 5
COMMUT; 19
compound triggers; 36
Compound Triggers; 76
concurrency problem; 53
CONDITION; 8
ccondition handler; 8
CONTAINS SQL; 4
CONTINUE; 8
cursor model; 88
cursor processing
  explicit SQL cursor; 85
  implicit cursor; 86
DB2 CLP utility; 5
DBMS_LOCK; 36, 102
DBMS_LOCK package; 33
DBMS_OUTPUT package; 87
DELETED table; 41
DELETING; 36
dETERMINISTIC; 4, 12
distributed transactions; 90
DO clause; 7
dYNAMIC RESULT SETS; 5
eMBEDded SQL; 85
Embedded SQL; 84
ESQL; 2, 84
EXCEPTION part; 33
EXEC SQL; 85
EXECUTE privilege; 5, 13, 16
EXIT; 8
FENCED; 10
FOLLOW; 76
FOR control structure; 7
FOR EACH clause; 14
FOR UPDATE clause; 33
GET DIAGNOSTICS; 8
GRANT EXECUTE; 33
handler action; 8
holding; 86
impedance mismatch; 83
implicit COMMIT; 95
implicit cursor; 102
Inline SQL PL; 28
INOUT parameter; 3, 6
INSENSITIVE; 7
INSERTED table; 41
INSERTING; 36
INSTEAD OF; 16
isolation level; 53
LEAVE; 7
LOOP control structure; 7
methods; 2
middleware API; 88
MODIFIES SQL DATA; 5
mutating table; 64, 78
nested savepoints; 92
nesting of triggers; 16
NEW row; 16
NO CASCADE; 27
NO SQL; 4
NO_DATA_FOUND; 33, 102
NOT FOUND; 8
OLD row; 16
optimistic concurrency control; 48
optimistic locking; 20, 85
partial rollback; 92
PL/SQL; 28
PL/SQL anonymous block; 29
PL/SQL Cursor FOR LOOP; 86
PL/SQL package; 29, 74
PL/SQL packages; 102
PL/SQL record; 87
privileges; 5
pseudo column; 38
READS SQLDATA; 4
recursive trigger; 16
REF CURSOR; 105

© Martti Laiho 2014
Comments of Readers

“Introduction to Procedural Extensions of #SQL” covers everything you need to know about stored #procedures, #triggers and stored functions.