www.DBTechNet.org

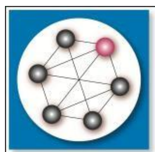
SQL Stored Routines

Procedural Extensions of SQL and External Routines in Transactional Context

Authors:
Martti Laiho, Fritz Laux, Ilia Petrov, Dimitris Dervos

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.

licensed on Creative Commons license for non-commercial use



www.DBTechNet.org

SQL Stored Routines

- *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 co-operation 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_4.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.

licensed on Creative Commons license for non-commercial use

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 24
Triggers 26
RVV example on optimistic locking 26
2.2 Oracle PL/SQL 27
Simple myProc and myFun tests 29
Exception handling and BankTransfer examples 32
On Oracle Triggers 34
RVV example on optimistic locking 35
2.3 SQL Server Transact-SQL 37
Simple myProc and myFun tests 37
Exception handling and BankTransfer examples 39
Triggers and RVV example on optimistic locking 40
2.4 MySQL/InnoDB 41
Simple myProc and myFun tests 41
Trigger workaround for the missing support of CHECK constraints: 42
Exception handling and BankTransfer examples 43
Triggers 44
RVV example on optimistic locking 44
2.5 PostgreSQL PL/pgSQL 45
Simple myProc and myFun tests 45
Exception handling and BankTransfer examples 46
Triggers and RVV example on optimistic locking 47
2.6 Pyrrho DBMS 47

licensed on Creative Commons license for non-commercial use

Simple myProc and myFun tests 48
Exception handling and BankTransfer examples 49
Triggers and RVV example on optimistic locking 51
Part 3 Concurrency Problems due to Side-effects 54
3.1 Side-effects due to Foreign Keys 54
SQL Server Transact-SQL 55
DB2 SQL PL 56
Oracle PL/SQL 57
MySQL/InnoDB 58
PostgreSQL PL/pgSQL 60
Pyrrho 61
3.2 Concurrency Side-effects due to Triggers 62
DB2 SQL PL 63
Oracle PL/SQL 65
SQL Server Transact-SQL 66
MySQL/InnoDB 67
PostgreSQL PL/pgSQL 68
Pyrrho 69
3.3 Referential Integrity Rules and Triggers 70
Part 4 Comparing SQL/PSM and Implementations 80
Stored procedures: 80
Stored functions: 81
Triggers: 82
Part 5 SQL-invoked External Routines 83
5.1 Introduction to SQL-invoked External Routines 83
SQL-invoked routines stored in the file system 83
SQL-invoked Java routines 84
Experiments on other platforms 87
Priorities and Challenges 87
Compatibility and Impedance Mismatching Issues 87
Experiment plan 88
5.2 External DB2 Stored Routines written in the C Language 89
DB2 troubleshooting 96
5.3 Java Routines in the DB2 JVM 97
5.4 Java Routines in the Oracle JVM 105

licensed on Creative Commons license for non-commercial use

5.5 C# Routines in the SQL Server CLR 111
Summary 116
Part 6 Server-Side Transaction Programming 117
6.0 Server-Side transactional mode: *'To be, or not to be, -that is the question'* 117
6.1 Retry Pattern and Protocol 119
6.2 Bank Transfer Procedures in DB2 SPL 125
6.3 Bank Transfer Procedures in PL/SQL 130
6.4 Bank Transfer Procedures in MySQL 133
Summary 135
References 137
Appendix 1 Result Set Processing with Cursor Mechanism 139
Appendix 2 Nested Transactions and Transactions with Savepoints 145
Appendix 3 Autonomous Transactions 151
Appendix 4 Calling Stored Procedures in Java Program 153
Index 164
Comments of Readers 166

licensed on Creative Commons license for non-commercial use

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.

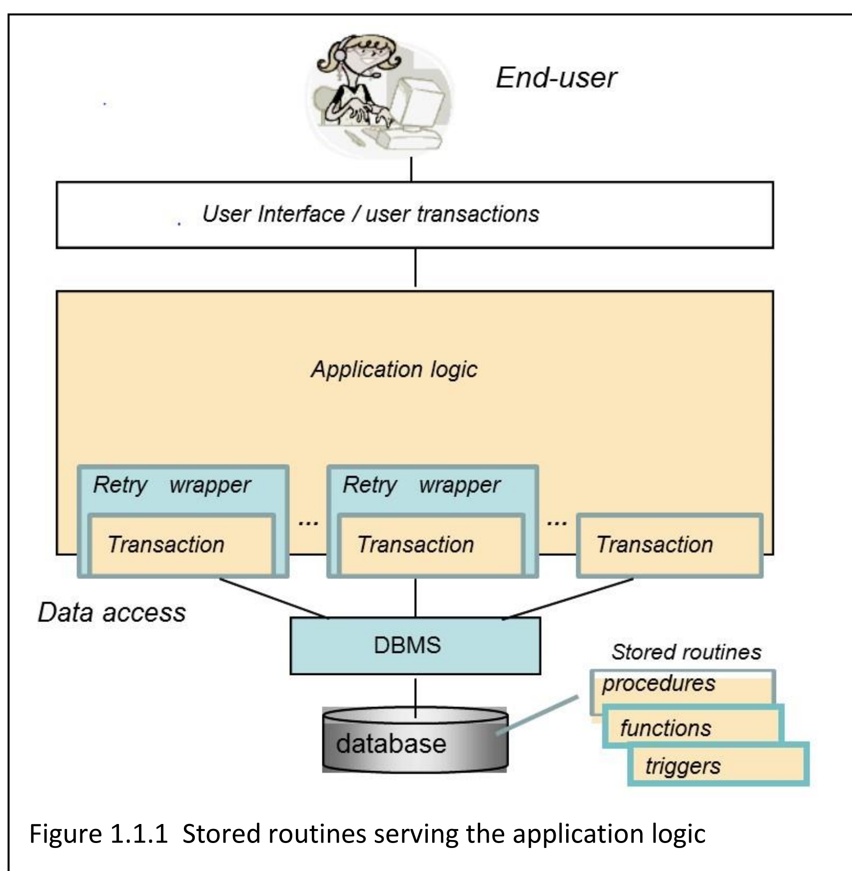


Figure 1.1.1 Stored routines serving the application logic

Figure 1.1.1 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

licensed on Creative Commons license for non-commercial use

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.

licensed on Creative Commons license for non-commercial use

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 acctId INT, OUT o_balance INT)
LANGUAGE SQL
SELECT balance INTO o_balance FROM Accounts
WHERE acctno = acctId ;
```

licensed on Creative Commons license for non-commercial use

An overview of SQL/PSM syntax for stored procedures is presented in Figure 1.1.2.

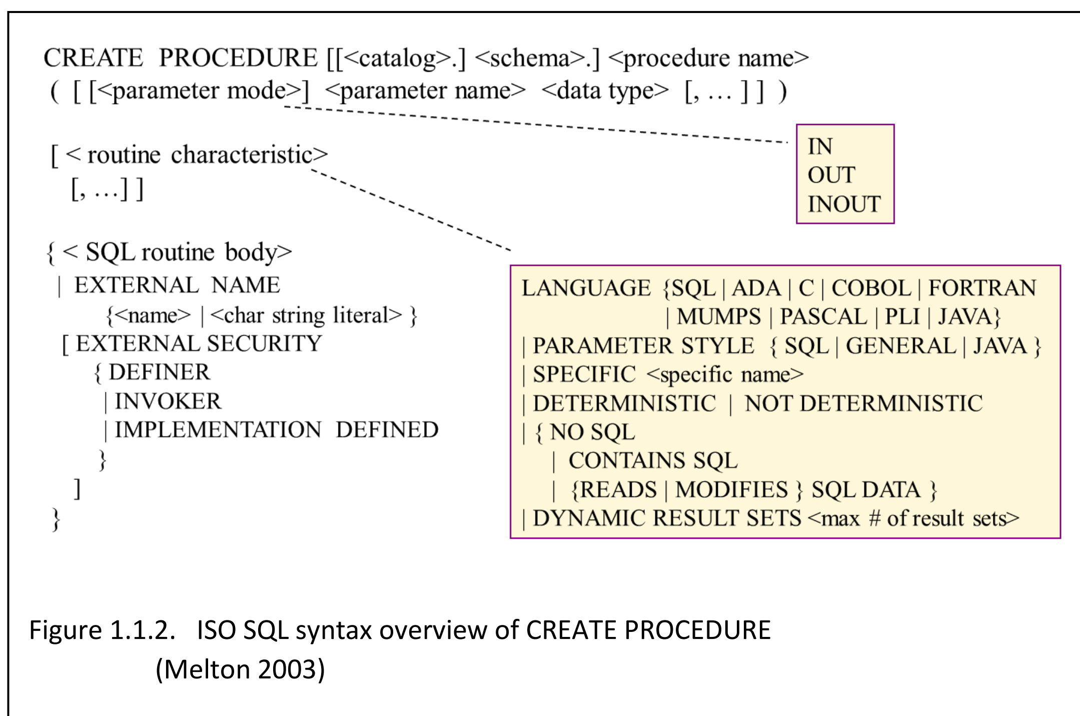


Figure 1.1.2. ISO SQL syntax overview of CREATE PROCEDURE (Melton 2003)

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.

licensed on Creative Commons license for non-commercial use

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

```
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

```
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:

```
db2 => call mySet(2)

  Result set 1
  --------------
  ID          S
  ----------- ----------
            2 second
            4 fourth

  2 record(s) selected.

db2 => call mySet(1)

  Result set 1
```

licensed on Creative Commons license for non-commercial use

```
--------------
ID          S
----------- ----------
          1 first
          3 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 label can be used to qualify the variables. The label name of the compound statement can also be used

[1] We will explain the savepoint concept in Appendix 2

licensed on Creative Commons license for non-commercial use

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

```
[<label> : ] LOOP
             <SQL statement> [, .. ]
END LOOP [<label>]
```

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

```
[<label> : ] FOR <loop name> AS
       [<cursor name> [INSENSITIVE] CURSOR FOR ]
       <SELECT statement>
       DO  <SQL statement> [, .. ]
END FOR [<label>]
```

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

licensed on Creative Commons license for non-commercial use

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 by 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.

It is important to remember that NOT FOUND condition test applies only SELECT statements. Since SQL is a **set-oriented** language, in terms of set processing execution of UPDATE or DELETE are considered successful even if no rows were found. Therefore the success after these statements need to be tested

licensed on Creative Commons license for non-commercial use

separately, for example testing the ROW_COUNT indicator of DIAGNOSTICS or some corresponding indicator provided by the DBMS product.

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 (
acctno  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL,
CONSTRAINT unloanable_account CHECK (balance >= 0)
);
INSERT INTO Accounts (acctno,balance) VALUES (101,1000);
INSERT INTO Accounts (acctno,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 acctno INTO acct FROM Accounts WHERE acctno = fromAcct ;
    UPDATE Accounts SET balance = balance - amount WHERE acctno = fromAcct;
    SET acct = toAcct;
    SELECT acctno INTO acct FROM Accounts WHERE acctno = toAcct ;
    UPDATE Accounts SET balance = balance + amount WHERE acctno = toAcct;
    COMMIT;
    SET msg = 'committed';
END P1
```

licensed on Creative Commons license for non-commercial use

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. Only the final result of the procedure will be transmitted to the client, while intermediate data is processed on the server.

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[3]. 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 can be 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, and aborted son process will be indicated only by raised exception by te 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.

licensed on Creative Commons license for non-commercial use

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

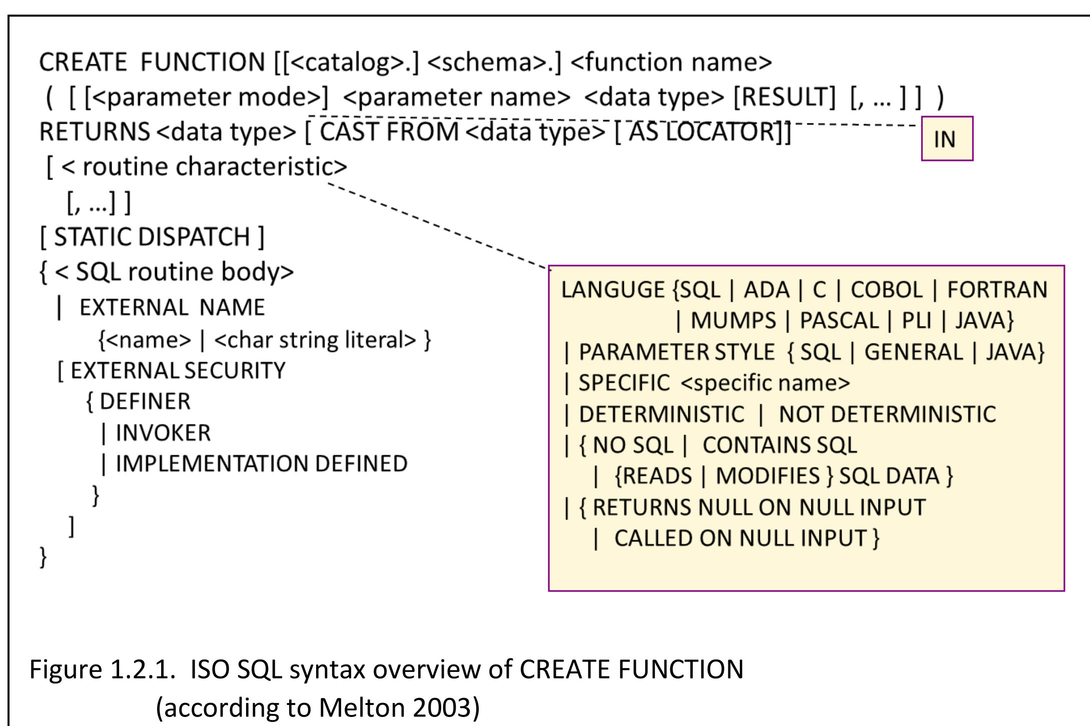


Figure 1.2.1. ISO SQL syntax overview of CREATE FUNCTION
(according to Melton 2003)

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.

Figure 1.2.1 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.

licensed on Creative Commons license for non-commercial use

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*2*3*..*(n-1)*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);
+--------------+
| factorial(3) |
+--------------+
|            6 |
+--------------+
1 row in set (0.00 sec)

mysql> SELECT factorial(0);
+--------------+
| factorial(0) |
+--------------+
|            1 |
+--------------+
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

licensed on Creative Commons license for non-commercial use

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

ID          STR
----------- ----------
          1 first
          3 third

  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.

licensed on Creative Commons license for non-commercial use

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.

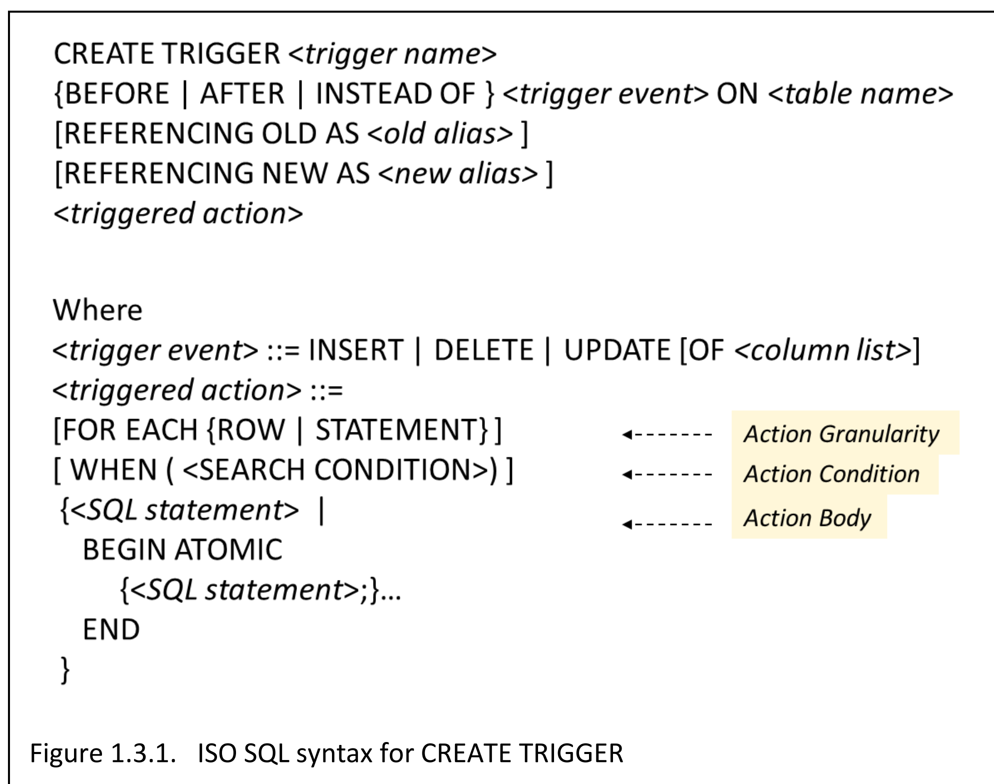


Figure 1.3.1. ISO SQL syntax for CREATE TRIGGER

Figure 1.3.1 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 1.3.2 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.

licensed on Creative Commons license for non-commercial use

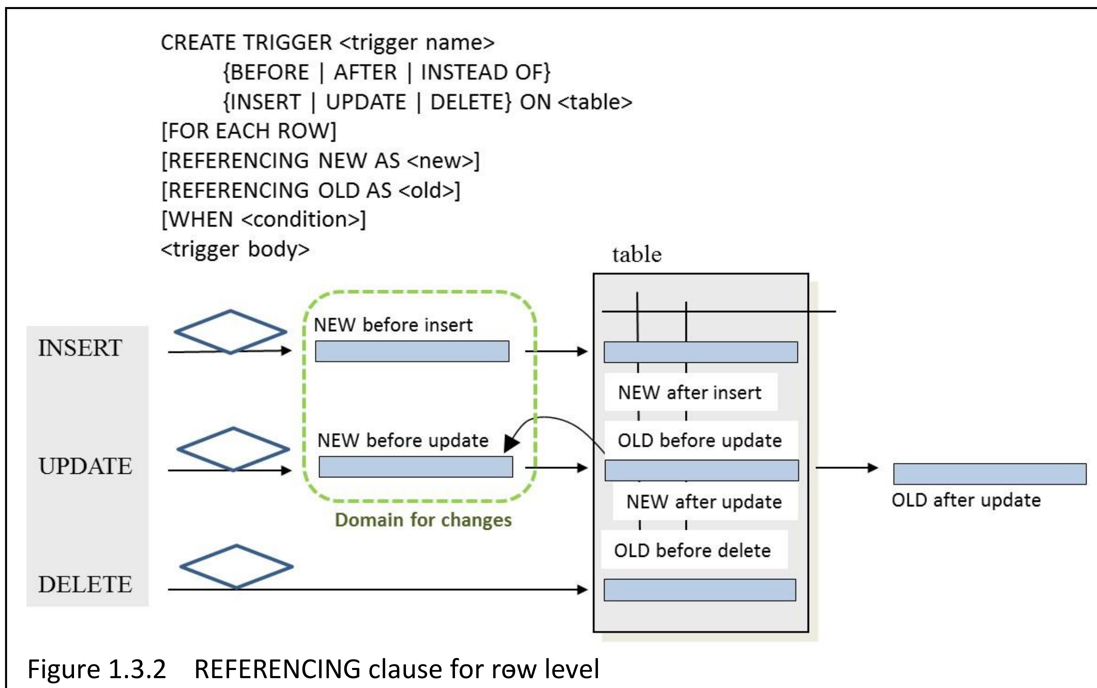


Figure 1.3.2 REFERENCING clause for row level

Figure 1.3.2 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.

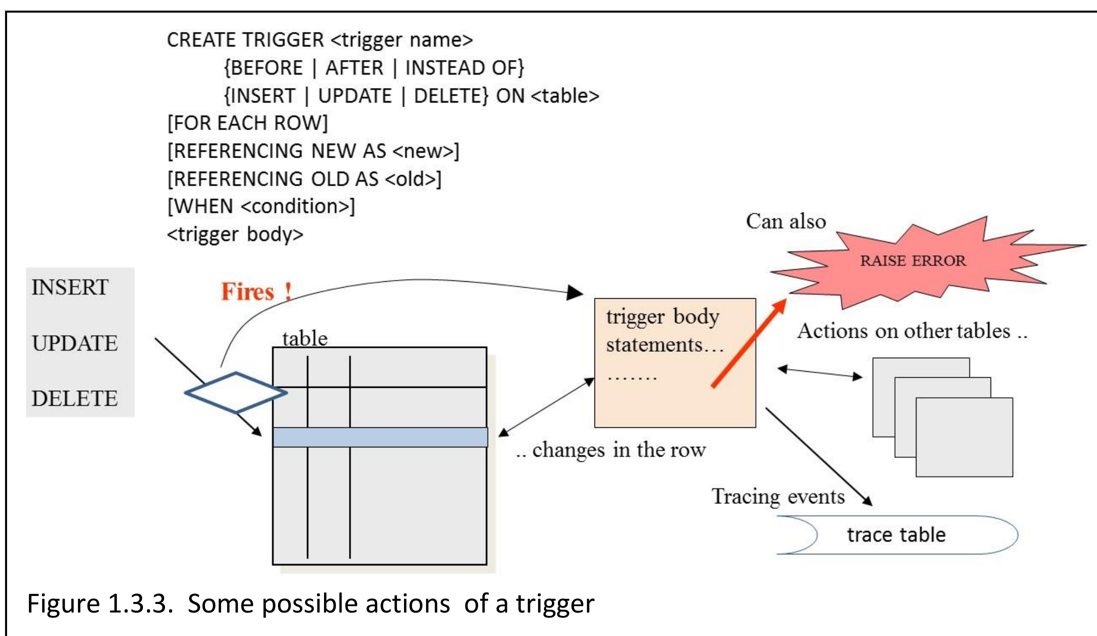


Figure 1.3.3. Some possible actions of a trigger

Figure 1.3.3 demonstrates some possible actions of a trigger. The action body (trigger body) consists of one SQL statement or an ATOMIC compound statement.

licensed on Creative Commons license for non-commercial use

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 AFTER UPDATE 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.

licensed on Creative Commons license for non-commercial use

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 over-using 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.

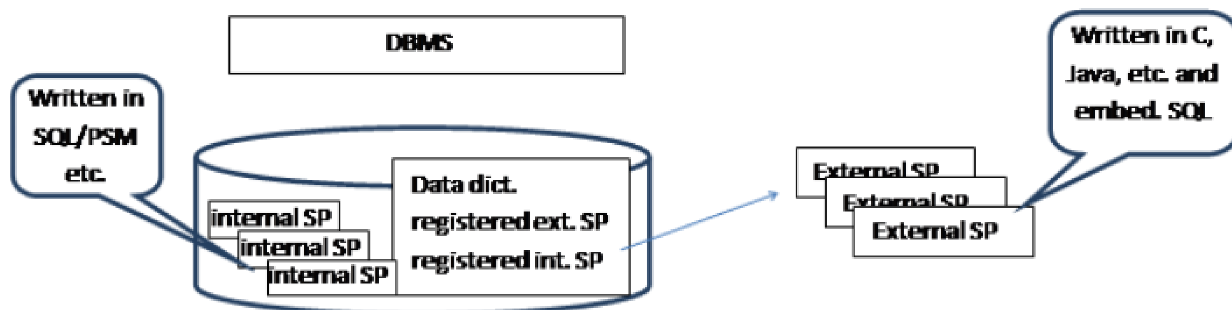
Database administrators can enable or disable triggers. For example, when dealing with bulk loading data, triggers may need to be disabled by the administrator, and enabled afterwards.

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.

licensed on Creative Commons license for non-commercial use

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

	ANSI/ISO SQL SQL/PSM	DB2 SQL PL	Oracle PL/SQL	SQL SERVER Transact SQL	MySQL SQL/PSM	PostgreSQL PL/pgSQL	Pyrrho
Control statements:							
	CASE .. WHEN	CASE .. WHEN	CASE .. WHEN	CASE .. WHEN	CASE .. WHEN	CASE .. WHEN	CASE .. WHEN
	IF .. THEN ..	IF .. THEN ..	IF .. THEN ..	IF .. ELSE ..	IF .. THEN ..	IF .. THEN ..	IF .. THEN ..
		DO .. UNTIL					
	ITERATE				ITERATE		ITERATE
	LEAVE		EXIT WHEN		LEAVE	EXIT WHEN	LEAVE
				BREAK			BREAK
	LOOP		LOOP .. END LOOP		LOOP	LOOP .. END LOOP	LOOP .. END LOOP
	WHILE .. END WHILE	WHILE .. DO	WHILE .. LOOP	WHILE	WHILE	WHILE .. LOOP	WHILE .. DO
	REPEAT .. UNTIL	(function)			REPEAT		REPEAT .. UNTIL
	FOR		FOR .. LOOP			FOR .. LOOP	FOR .. DO .. END FOR
			GOTO	GOTO			
			NULL				
Exception raising							
	SIGNAL	SIGNAL	RAISE_APPLICATION_ERROR	RAISERROR	SIGNAL	RAISE EXCEPTION	SIGNAL
Exception handling / diagnostics							
	DECLARE .. HANDLER	DECLARE .. HANDLER			DECLARE .. HANDLER		DECLARE .. HANDLER
	GET DIAGNOSTICS	GET DIAGNOSTICS			GET DIAGNOSTICS	GET DIAGNOSTICS	GET DIAGNOSTICS
			EXCEPTION WHEN	TRY .. CATCH		EXCEPTION WHEN	

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. 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.

licensed on Creative Commons license for non-commercial use

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

licensed on Creative Commons license for non-commercial use

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:

```
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:

```
CREATE TABLE DUAL (X CHAR(1) NOT NULL PRIMARY KEY,
                   CHECK (X IN 'X')) /
INSERT INTO DUAL VALUES ('X') /
COMMIT /
```

licensed on Creative Commons license for non-commercial use

Now we can test the overloading functionality as follows:

```
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:

```
db2 => DROP SPECIFIC FUNCTION myF1 /
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:

```
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:

```
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.

db2 =>
```

licensed on Creative Commons license for non-commercial use

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 "Getting Started with 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

T_NO        T_USER               T_DATE     T_TIME   T_PROC     T_WHAT
----------- -------------------- ---------- -------- ---------- ---------------
2              -                    -          -        -          -
1           STUDENT              04/04/2014 15:23:15 myProc     Hello DB2

  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);
```

licensed on Creative Commons license for non-commercial use

```
     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.

licensed on Creative Commons license for non-commercial use

Exception handling and BankTransfer examples

Using GET DIAGNOSTICS

```
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 acctno = 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 acctno = 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, ?);

------------------------------ Commands Entered ------------------------------
CALL BankTransfer (101, 202, 100, ?);
------------------------------------------------------------------------------
CALL BankTransfer (101, 202, 100, ?)

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

  Return Status = 0

------------------------------ Commands Entered ------------------------------
CALL BankTransfer (101, 202, 1000, ?);
------------------------------------------------------------------------------
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

licensed on Creative Commons license for non-commercial use

```
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 acctno INTO acct FROM Accounts WHERE acctno = fromAcct ;
    UPDATE Accounts SET balance = balance - amount WHERE acctno = fromAcct;
    SET acct = toAcct;
    SELECT acctno INTO acct FROM Accounts WHERE acctno = toAcct ;
    UPDATE Accounts SET balance = balance + amount WHERE acctno = 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
```

licensed on Creative Commons license for non-commercial use

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[4] 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
```

[4] Beside DB2, NO CASCADE clause has been implemented also in Apache DerbyDB

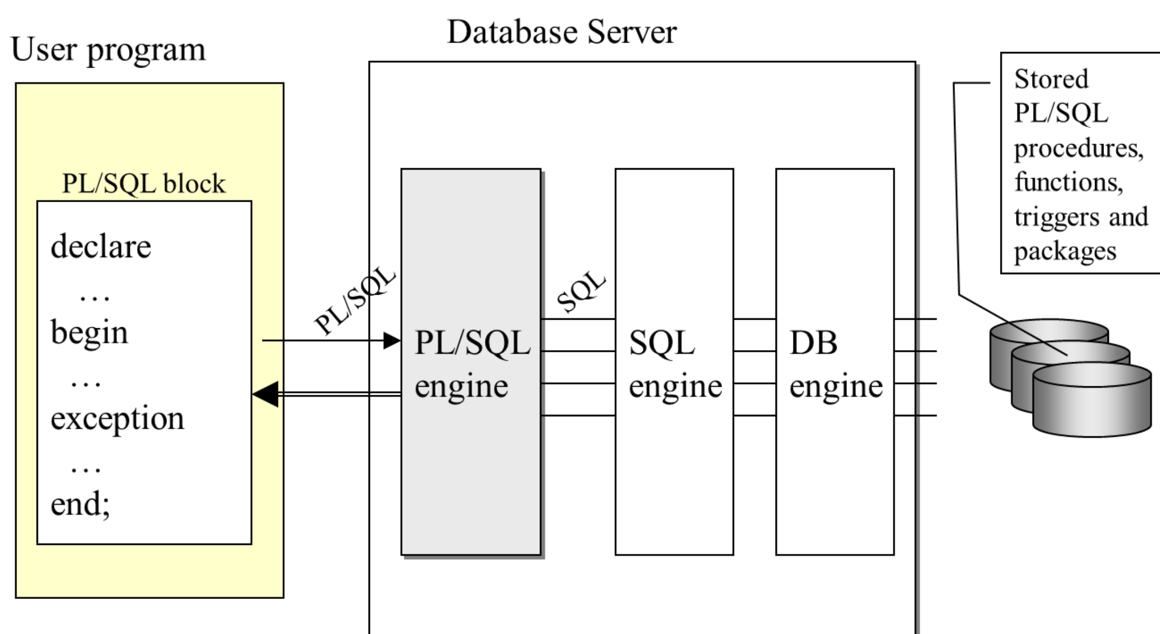
```
BEGIN ATOMIC
IF (old_row.rv = 2147483647) THEN
   SET new_row.rv = -2147483648;
ELSE
   SET new_row.rv = old_row.rv + 1;
END IF;
END @
COMMIT @

UPDATE Accounts SET balance = balance - 100 WHERE acctno = 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 2.2.1. 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.



licensed on Creative Commons license for non-commercial use

Figure 2.2.1. Oracle server's stack of engines and languages

The PL/SQL blocks can be included in stored procedures, functions, package methods, or used as independent anonymous blocks, as described in figures 2.2.2 - 2.2.4.

Procedure	Function	Anonymous

PROCEDURE myProc
  IS
  [-- declarations]
  BEGIN
  -- statements
  [EXCEPTION
  -- error handling]
  END;
	FUNCTION myFunc
	  RETURN someType
	  IS
	  [-- declarations]
	  BEGIN
	  -- statements
	  [EXCEPTION
	  -- error handling]
	  END;
		[DECLARE
		  -- declarations]
		  BEGIN
		  -- statements
		  [EXCEPTION
		  -- error handling]
		  END;

Figure 2.2.2. 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 2.2.3.

```
CREATE OR REPLACE PROCEDURE del_emp(
  p_id IN employees.employee_id%TYPE,
  p_ok OUT BOOLEAN)
IS BEGIN
  DELETE FROM employees
    WHERE employee_id = p_id;
  IF SQL%ROWCOUNT = 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;
```

```
CREATE FUNCTION get_emp_count RETURN NUMBER
IS
  v_count NUMBER;
BEGIN
  SELECT COUNT(*)
    INTO v_count
    FROM employees;
  RETURN v_count;
END;
…
SELECT get_emp_count FROM dual;
```

Figure 2.2.3. 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 2.2.4. 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.

licensed on Creative Commons license for non-commercial use

- Package is a collection of logically related PL/SQL types, variables, and subprograms
- Package specification = interface
- Package body = implementation
  - May contain private objects

```
CREATE OR REPLACE PACKAGE p AS
  v_x NUMBER := 0;
  PROCEDURE setX(newX NUMBER);
END p;
/
CREATE OR REPLACE PACKAGE BODY p AS
  PROCEDURE setX(newX NUMBER) IS
  BEGIN
    v_x := newX;
  END;
END;
...
BEGIN
  p.setX(123);
  DBMS_OUTPUT.PUT_LINE('X='||p.v_x);
END;
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. |

Figure 2.2.4. Creating and using of a PL/SQL stored package
(as presented by Timo Leppänen / Oracle Finland)

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

```
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:

```
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

```
CREATE TABLE myTrace (
 t_no     INT NOT NULL,
 t_user   VARCHAR(20),
 t_date   DATE,
 t_proc   VARCHAR(10),
```

licensed on Creative Commons license for non-commercial use

```
 t_what   VARCHAR(30)
);

CREATE VIEW myTraceV AS
SELECT t_no, t_user,
       TO_CHAR(t_date, 'YYYY-MM-DD') AS t_date,
       TO_CHAR(t_date, 'HH24:MI:SS') AS t_time,
       t_proc,
       t_what
FROM myTrace;
```

Now let’s verify how it looks as a native Oracle SQL table:

```
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:

```
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);
```

licensed on Creative Commons license for non-commercial use

```
END;
/
Hello Oracle

PL/SQL procedure successfully completed.

SELECT * FROM myTraceV;
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 myTraceV;
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;
/
```

licensed on Creative Commons license for non-commercial use

```
anonymous block completed
Hello again!
```

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.

licensed on Creative Commons license for non-commercial use

```
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 acctno INTO acct FROM Accounts
       WHERE acctno = acct FOR UPDATE;
     UPDATE Accounts SET balance = balance - amount
       WHERE acctno = fromAcct;
     dbms_lock.sleep(20); -- pause for deadlock testing

     acct := toAcct;
     SELECT acctno INTO acct FROM Accounts
       WHERE acctno = acct FOR UPDATE;
     UPDATE Accounts SET balance = balance + amount
       WHERE acctno = toAcct;
     COMMIT;
     msg := 'committed';
EXCEPTION
     WHEN NO_DATA_FOUND THEN
          ROLLBACK;
          msg := 'missing account ' || TO_CHAR(acct);
     WHEN OTHERS THEN
          ROLLBACK;
          ernum := SQLCODE;
          mesg := SUBSTR(SQLERRM, 1, 200);
          msg := 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 using sqlplus utility. Note the format of invoking PL/SQL procedures and how OUT parameters are handled.

```
sqplus user1/sql
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
sqplus user1/sql
-- concurrent session B
set serveroutput on;
DECLARE mesg VARCHAR2(200) := ' ';
```

licensed on Creative Commons license for non-commercial use

```
BEGIN
   BankTransfer (202,101,100, mesg);
   DBMS_OUTPUT.PUT_LINE(mesg);
END;
/
SELECT * FROM Accounts;
COMMIT;
```

```
student@debianDB: ~
File Edit View Terminal Help
SQL> 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;

SQL> SQL> SQL>   2    3    4    5    6  ORA-00060: deadlock de
tected while waiting for resource SQLcode=-60

PL/SQL procedure successfully completed.

SQL>
    ACCTNO    BALANCE
---------- ----------
       101       1100
       202       1900

SQL>
```

```
student@debianDB: ~
File Edit View Terminal Help
SQL> SQL> -- concurrent session B
set serveroutput on;
DECLARE mesg VARCHAR2(200) := ' ';
BEGIN
   BankTransfer (202,101,100, mesg);
   DBMS_OUTPUT.PUT_LINE(mesg);
END;
/
SELECT * FROM Accounts;
COMMIT;
SQL> SQL>   2    3    4    5    6  committed

PL/SQL procedure successfully completed.

SQL>
    ACCTNO    BALANCE
---------- ----------
       101       1100
       202       1900

SQL>
Commit complete.
```

Following PL/SQL anonymous block tests the case when the other account is missing

```
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

```
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 (acctno,balance) VALUES (101,1000);
INSERT INTO Accounts (acctno,balance) VALUES (202,2000);
COMMIT;
DESCRIBE Accounts;
-- if column rv is missing
```

licensed on Creative Commons license for non-commercial use

```
ALTER TABLE Accounts
   ADD 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;
/
```

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 acctno = 101;
   COMMIT;
   DBMS_LOCK.SLEEP(20);
-- step 3 after client B
   UPDATE Accounts SET balance = balance - 500
   WHERE acctno = 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 acctno = 101;
SELECT acctno, balance FROM Accounts WHERE acctno = 101;
COMMIT;
```

Note the assignment operator “:=” of Ada and colons “:” in front of the row qualifiers OLD and NEW.

licensed on Creative Commons license for non-commercial use

```
student@debianDB: ~
File Edit View Terminal Help
SQL> -- step 1 Client A
SELECT * FROM Accounts;
SQL>
    ACCTNO    BALANCE         RV
---------- ---------- ----------
       101       1000          0
       202       2000          0

SQL> DECLARE  p_balance INT;
         p_rv INT;
BEGIN
   SELECT balance, rv INTO p_balance, p_rv FROM Accounts WHERE acctn
o = 101;
   COMMIT;
   DBMS_LOCK.SLEEP(20);
-- step 3 after client B
   UPDATE Accounts SET balance = balance - 500
   WHERE acctno = 101 AND rv = p_rv;
END;
/
  2    3    4    5    6    7    8    9   10   11
PL/SQL procedure successfully completed.

SQL> SELECT * FROM Accounts;

    ACCTNO    BALANCE         RV
---------- ---------- ----------
       101        900          1
       202       2000          0

SQL> ROLLBACK;

Rollback complete.
```

```
student@debianDB: ~
File Edit View Terminal Help
SQL> -- step 2 Client B while A is sleeping
UPDATE Accounts SET balance = balance - 100 WHERE acctno = 101;
SELECT acctno, balance FROM Accounts WHERE acctno = 101;
COMMIT;
SQL>
1 row updated.

SQL>
    ACCTNO    BALANCE
---------- ----------
       101        900

SQL>
Commit complete.
```

Figure 2.2.5 Results of the RVV test runs

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

```
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;
```

licensed on Creative Commons license for non-commercial use

```
END
GO
```

Following some test result

```
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
 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:

```
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,GETDATE(), CAST(GETDATE() AS time(0)),
            'myProc', @p_in);
     IF (@p_no = 1) COMMIT; ELSE ROLLBACK;
     RETURN @p_in;
END;
GO
Msg 443, Level 16, State 15, Procedure myFun, Line 4
Invalid use of a side-effecting operator 'INSERT' within a function.
Msg 443, Level 16, State 15, Procedure myFun, Line 7
Invalid use of a side-effecting operator 'COMMIT TRANSACTION' within a function.
Msg 443, Level 16, State 15, Procedure myFun, Line 7
Invalid use of a side-effecting operator 'ROLLBACK TRANSACTION' within a function.
```

=> 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.

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;
```

licensed on Creative Commons license for non-commercial use

```
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 acctno = @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 acctno = @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;
```

licensed on Creative Commons license for non-commercial use

```
SELECT @msg AS Msg;

DECLARE @msg VARCHAR(100);
SET @msg = 'Missing account';
EXEC BankTransfer 101, 999, 100, @msg OUTPUT;
SELECT @msg AS Msg;
```

Results | Messages

	ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
1	50000	11	1	BankTransfer	19	Missing account 999

	Msg
1	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;
```

Results | Messages

	ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
1	547	16	0	BankTransfer	9	The UPDATE statement conflicted with the CHECK c...

	Msg
1	The UPDATE statement conflicted with the CHECK constraint "unloanable_account". The conflict occurre

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

licensed on Creative Commons license for non-commercial use

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;
```

licensed on Creative Commons license for non-commercial use

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.

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

delimiter !
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; !
delimiter ;

delimiter !
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; !
delimiter ;

--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;

INSERT INTO Accounts VALUES (2, 100);
UPDATE Accounts SET balance = -100 WHERE acctno = 2;
GET DIAGNOSTICS @rowcount = ROW_COUNT;
```

licensed on Creative Commons license for non-commercial use

```
GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE, @sqlcode =
MYSQL_ERRNO ;
SELECT @sqlstate, @sqlcode, @rowcount;
DELETE FROM Accounts WHERE acctno = 2; COMMIT;
```

Exception handling and BankTransfer examples

First a version using **DIAGNOSTICS**

```
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 acctno = 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 acctno = 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 !
```

Then another version using **condition handlers**

```
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 acctno INTO acct FROM Accounts WHERE acctno = fromAcct ;
```

licensed on Creative Commons license for non-commercial use

```
        UPDATE Accounts SET balance = balance - amount WHERE acctno = fromAcct;
        SET acct = toAcct;
        SELECT acctno INTO acct FROM Accounts WHERE acctno = toAcct ;
        UPDATE Accounts SET balance = balance + amount WHERE acctno = 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; END
#
delimiter ;


COMMIT;


-- Concurrency test by 2 clients
--
-- step 1 Client A
SET AUTOCOMMIT = 0;
SET @balanceB = 0;  -- init value
SET @rv = 0;  -- init value
```

licensed on Creative Commons license for non-commercial use

```
SELECT balance, rv INTO @balance, @rv FROM Accounts WHERE acctno = 101;
SELECT @balance, @rv;
COMMIT;

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

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

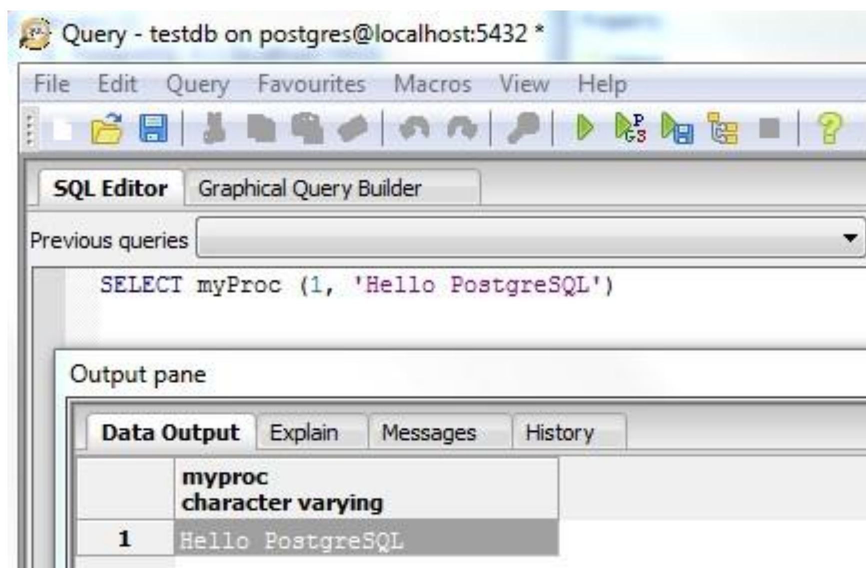
In PLPGSQL language procedures are written as functions which return NULL. There is no other essential difference.

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.
```

licensed on Creative Commons license for non-commercial use

```
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 plpsql 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 acctno = fromAcct;
    IF (NOT FOUND) THEN
        RAISE EXCEPTION USING MESSAGE = '* Unknown fromAccount ' || fromAcct;
    ELSE
        UPDATE Accounts SET balance = balance + amount WHERE acctno = toAcct;
        IF (NOT FOUND) THEN
            RAISE EXCEPTION USING MESSAGE = '* Unknown toAccount ' || toAcct;
        END IF;
    END IF;
    RETURN 1;
EXCEPTION
```

licensed on Creative Commons license for non-commercial use

```
    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.

licensed on Creative Commons license for non-commercial use

Simple myProc and myFun tests

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

```
[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:

```
[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;
|----|---------|---------|----------------|------|------------|
|T_NO|T_USER   |T_DATE   |T_TIME          |T_PROC|T_WHAT      |
|----|---------|---------|----------------|------|------------|
|1   |WIN764\ML|19.5.2014|12:34:38.7783366|myProc|Hello Pyrrho|
|----|---------|---------|----------------|------|------------|
SQL>
```

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

```
[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;
|----------------|
|Col$1           |
|----------------|
|Hello Pyrrho fun|
|----------------|
SQL> SELECT * FROM myTrace;
|----|---------|---------|----------------|------|----------------|
```

licensed on Creative Commons license for non-commercial use

```
|T_NO|T_USER   |T_DATE   |T_TIME          |T_PROC|T_WHAT          |
|----|---------|---------|----------------|------|----------------|
|1   |WIN764\ML|19.5.2014|12:52:38.4150883|myProc|Hello Pyrrho fun|
|----|---------|---------|----------------|------|----------------|
```

Exception handling and BankTransfer examples

Latest versions of Pyrrho support both exception handling and currently the whole specification of DIAGNOSTICS in SQL Standard. Pyrrho does not support COMMIT nor ROLLBACK statements in stored routines. So in case of errors a return code needs to be used as indicator to the invoking code on need for ROLLBACK. In the following BankTransfer procedure the output parameter “rc” serves this purpose, but also show how to avoid non-matching updates by testing existence by SELECT / NOT_FOUND arrangement in the following:

```
[CREATE TABLE Accounts (
 acctno  INTEGER NOT NULL PRIMARY KEY,
 balance INTEGER NOT NULL
 CONSTRAINT uncreditable_account CHECK (balance >= 0)
)]
INSERT INTO Accounts (acctno,balance) VALUES (101,1000);
INSERT INTO Accounts (acctno,balance) VALUES (202,2000);
SELECT * FROM Accounts;

[CREATE PROCEDURE BankTransfer
    (IN fromAcct INT,
     IN toAcct   INT,
     IN amount   INT,
     OUT rc      INT,
     OUT msg     VARCHAR(100))
BEGIN
     DECLARE cnt  INT;
     DECLARE mesg VARCHAR(100);
     DECLARE NOT_FOUND BOOLEAN DEFAULT FALSE;
     DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET NOT_FOUND=TRUE;
     DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN
         GET DIAGNOSTICS mesg = MESSAGE_TEXT;
         SET msg = mesg
         END ;
     SET rc = -1;
     SET msg = 'ok';
     SET cnt = 0;
     SELECT 1 INTO cnt FROM Accounts WHERE acctno = toAcct;
     IF (NOT_FOUND) THEN
          SET msg = '* Unknown toAccount ' || CAST(toAcct AS VARCHAR(10))
     ELSE BEGIN
         UPDATE Accounts SET balance = balance - amount WHERE acctno = fromAcct;
         GET DIAGNOSTICS cnt = ROW_COUNT;
         IF (cnt = 0) THEN
             SET msg = '* Unknown fromAccount ' || CAST(fromAcct AS VARCHAR(10))
         ELSE
           BEGIN
             UPDATE Accounts SET balance = balance + amount
                    WHERE acctno = toAcct;
             GET DIAGNOSTICS cnt = ROW_COUNT;
             IF (cnt = 0) THEN
                 SET msg = '* Unknown toAccount ' || CAST(toAcct AS VARCHAR(10))
             ELSE
```

licensed on Creative Commons license for non-commercial use

```
                  SET rc = 0
              END IF
            END
          END IF
        END
     END IF
END;]

-- A wrapper function for testing the procedure:
[CREATE FUNCTION BT_test
    (IN fromAcct INT,
     IN toAcct   INT,
     IN amount   INT)
     RETURNS VARCHAR(100)
 BEGIN
     DECLARE rc INT;
     DECLARE msg VARCHAR(100);
     CALL BankTransfer (fromAcct, toAcct, amount, rc, msg);
     RETURN msg;
 END;]

Tested as follows:

SQL> select * from accounts;
|------|-------|
|ACCTNO|BALANCE|
|------|-------|
|101   |1000   |
|202   |2000   |
|------|-------|
SQL> BEGIN TRANSACTION;
SQL-T>SELECT BT_test (101, 202, 100) FROM STATIC;
|-------|
|BT_TEST|
|-------|
|ok     |
|-------|
SQL-T>select * from Accounts;
|------|-------|
|ACCTNO|BALANCE|
|------|-------|
|101   |900    |
|202   |2100   |
|------|-------|
SQL-T>ROLLBACK;
SQL> -- Transfer of too big amount?
SQL> BEGIN TRANSACTION;
SQL-T>SELECT BT_test (101, 202, 2000) FROM STATIC;
|----------------------------------------------------------|
|BT_TEST                                                   |
|----------------------------------------------------------|
|Table check UNCREDITABLE_ACCOUNT fails for table ACCOUNTS|
|----------------------------------------------------------|
SQL-T>select * from Accounts;
|------|-------|
|ACCTNO|BALANCE|
|------|-------|
|101   |1000   |
|202   |2000   |
|------|-------|
SQL-T>ROLLBACK;
```

licensed on Creative Commons license for non-commercial use

```
SQL> BEGIN TRANSACTION;
SQL-T>SELECT BT_test (100, 202, 100) FROM STATIC;
|-------------------------|
|BT_TEST                  |
|-------------------------|
|* Unknown fromAccount 100|
|-------------------------|
SQL-T>select * from Accounts;
|------|-------|
|ACCTNO|BALANCE|
|------|-------|
|101   |1000   |
|202   |2000   |
|------|-------|
SQL-T>ROLLBACK;
SQL> BEGIN TRANSACTION;
SQL-T>SELECT BT_test (101, 200, 100) FROM STATIC;
|-----------------------|
|BT_TEST                |
|-----------------------|
|* Unknown toAccount 200|
|-----------------------|
SQL-T>select * from Accounts;
|------|-------|
|ACCTNO|BALANCE|
|------|-------|
|101   |1000   |
|202   |2000   |
|------|-------|
SQL-T>ROLLBACK;
SQL> quit
```

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

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

and create the following function and trigger pair for the table

```
[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> UPDATE Accounts SET balance = balance - 100 WHERE acctno = 101;
SQL> SELECT * FROM Accounts;
|------|-------|--|
```

licensed on Creative Commons license for non-commercial use

```
|ACCTNO|BALANCE|RV|
|------|-------|--|
|101   |900    |1 |
|202   |2000   |0 |
|------|-------|--|
```

More examples on Pyrrho triggers can be found at 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> [CREATE TABLE Accounts (
> acctno  INTEGER NOT NULL PRIMARY KEY,
> balance INTEGER NOT NULL,
> CONSTRAINT unloanable_account CHECK (balance >= 0)
> ); ]
SQL> INSERT INTO Accounts (acctno, balance) VALUES (101, 1000);
SQL> INSERT INTO Accounts (acctno, balance) VALUES (202, 2000);
SQL>
SQL> SELECT acctno, balance, check, versioning FROM Accounts;
|------|-------|------------------|----------|
|ACCTNO|BALANCE|CHECK             |VERSIONING|
|------|-------|------------------|----------|
|101   |1000   |testdb.osp:228:228|228       |
|202   |2000   |testdb.osp:264:264|264       |
|------|-------|------------------|----------|
SQL> UPDATE Accounts SET balance = balance - 100 WHERE acctno = 101;
1 records affected
SQL> SELECT acctno, balance, check, versioning FROM Accounts;
|------|-------|------------------|----------|
|ACCTNO|BALANCE|CHECK             |VERSIONING|
|------|-------|------------------|----------|
|101   |900    |testdb.osp:228:301|301       |
|202   |2000   |testdb.osp:264:264|264       |
|------|-------|------------------|----------|
SQL>
SQL> SELECT acctno, balance, VERSIONING AS RV FROM Accounts;
|------|-------|---|
|ACCTNO|BALANCE|RV |
|------|-------|---|
|101   |900    |301|
|202   |2000   |264|
|------|-------|---|
SQL>
```

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 VERIONING 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 acctno = 101;
1 records affected
```

licensed on Creative Commons license for non-commercial use

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

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

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 acctno = 101;
```

we shall not include the VERSIONING test.

licensed on Creative Commons license for non-commercial use

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:

```
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:

```
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:

SQLQuery1.sql - WI...B (WIN764\ML (54))*

```
-- 1. Client A
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE Parent SET psum = 100 WHERE pid = 1;
-- 3. Client A
INSERT INTO Child (cid,pid,cs,csum) VALUES (5,2,'kid5',10);
```

Messages

```
(1 row(s) affected)
```

SQLQuery2.sql - WI...B (WIN764\ML (55))*

```
-- 2. Client B
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE Parent SET psum = 200 WHERE pid = 2;
INSERT INTO Child (cid,pid,cs,csum) VALUES (4,1,'kid4',40);
```

Messages

```
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 55) was deadlocked on lock resources with a
```

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:

```
exec sp_lock
```

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	54	5	0	0	DB		S	GRANT
2	54	5	565577053	1	PAG	1:121	IX	GRANT
3	54	5	565577053	1	KEY	(8194443284a0)	X	GRANT
4	54	5	565577053	0	TAB		IX	GRANT
5	55	5	565577053	0	TAB		IX	GRANT
6	55	5	565577053	1	KEY	(61a06abd401c)	X	GRANT
7	55	5	613577224	0	TAB		IX	GRANT
8	55	5	565577053	1	KEY	(8194443284a0)	S	WAIT
9	55	5	565577053	1	PAG	1:121	IX	GRANT
10	55	5	613577224	1	KEY	(a0c936a3c965)	X	GRANT
11	55	5	0	0	DB		S	GRANT
12	55	5	613577224	1	PAG	1:127	IX	GRANT
13	56	4	0	0	DB		S	GRANT
14	58	5	0	0	DB		S	GRANT
15	58	1	1467152272	0	TAB		IS	GRANT

On line 8 we can verify that SQL Server uses lock protected read operation to check the existence of the referred row in the Parent table, and the needed S-lock is waiting for release of the granted X-lock of process 54 (on line 3).
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

licensed on Creative Commons license for non-commercial use

```
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 :

```
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:

```
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 => DB20000I  The SQL command completed successfully.
db2 => UPDATE Parent SET psum = 100 WHERE pid = 1;
DB20000I  The SQL command completed successfully.
db2 => -- 3. Client A
SET CURRENT LOCK TIMEOUT = 20;
db2 => DB20000I  The SQL command completed successfully.
db2 => INSERT INTO Child (cid,pid,cs,csum) VALUES (5,2,'kid5',10);
DB20000I  The SQL command completed successfully.
db2 => SELECT * FROM Parent;

PID         PS                   PSUM
----------- -------------------- -----------
          1 pa1                          100
          2 pa2                            0

  2 record(s) selected.

db2 =>
```

```
db2 => SELECT * FROM Parent;

PID         PS                   PSUM
----------- -------------------- -----------
          1 pa1                            0
          2 pa2                            0

  2 record(s) selected.

db2 => SELECT * FROM Child;

CID         PID         CS                   CSUM
----------- ----------- -------------------- -----------
          1           1 kid1                          10
          2           1 kid2                          20
          3           - orp3                          30

  3 record(s) selected.

db2 => -- 2. Client B
SET ISOLATION = RS;
db2 => DB20000I  The SQL command completed successfully.
db2 => SET CURRENT LOCK TIMEOUT = 20;
DB20000I  The SQL command completed successfully.
db2 => UPDATE Parent SET psum = 200 WHERE pid = 2;
DB20000I  The SQL command completed successfully.
db2 => INSERT INTO Child (cid,pid,cs,csum) VALUES (4,1,'kid4',40);
DB20000I  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 "68".  SQLSTATE=40001
db2 =>
```

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.

licensed on Creative Commons license for non-commercial use

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:

```
SELECT * FROM Parent;
SELECT * FROM Child;
-- 1. Client A
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE Parent SET ps = 'new' WHERE pid = 1;
-- 3. Client A
INSERT INTO Child (cid, pid, cs, csum)
VALUES (5, 1, 'kid5', 50);
SELECT * FROM Parent;
SELECT * FROM Child;
```

```
Script Output
Task completed in 0.319 seconds
PID                  PS                  PSUM
-------------------- ------------------- --------------------
1                    pa1                 0
2                    pa2                 0

CID                  PID                  CS                  CSUM
-------------------- -------------------- ------------------- -----
1                    1                    kid1                10
2                    1                    kid2                20
3                                         orp3                30

SET TRANSACTION succeeded.
1 rows updated
1 rows inserted
PID                  PS                  PSUM
-------------------- ------------------- --------------------
1                    new                 0
2                    pa2                 0

CID                  PID                  CS                  CSUM
-------------------- -------------------- ------------------- -----
1                    1                    kid1                10
2                    1                    kid2                20
3                                         orp3                30
5                    1                    kid5                50
```

```
-- 2. Client B
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE Child SET pid = 2 WHERE cid = 3;
INSERT INTO Child (cid, pid, cs, csum)
VALUES (4, 1, 'kid4', 40);
-- 4. Client B
SELECT * FROM Parent;
SELECT * FROM Child;
```

```
Script Output
Task completed in 0.309 seconds
SET TRANSACTION succeeded.
1 rows updated
1 rows inserted
PID                  PS                  PSUM
-------------------- ------------------- --------------------
1                    pa1                 0
2                    pa2                 0

CID                  PID                  CS                  CSUM
-------------------- -------------------- ------------------- -----
1                    1                    kid1                10
2                    1                    kid2                20
3                    2                    orp3                30
4                    1                    kid4                40
```

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:

licensed on Creative Commons license for non-commercial use

```
INSERT INTO Parent (pid,  ps) VALUES (3,'ghost');
COMMIT;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM Parent;
DELETE FROM Parent WHERE pid = 3;
INSERT INTO Child (cid, pid, cs, csum) VALUES (6,3,'orp3?', 60);
```

Script Output ×

Task completed in 0.366 seconds

```
SET TRANSACTION succeeded.
PID                    PS                   PSUM
---------------------- -------------------- ----------------------
1                      pa1                  0
2                      pa2                  0
3                      ghost                0

1 rows deleted

Error starting at line 8 in command:
INSERT INTO Child (cid, pid, cs, csum) VALUES (6,3,'orp3?', 60)
Error report:
SQL Error: ORA-02291: integrity constraint (USER1.CHILD_PARENT_FK) violated - parent key not found
02291. 00000 - "integrity constraint (%s.%s) violated - parent key not found"
*Cause:    A foreign key value has no matching primary key value.
*Action:   Delete the foreign key or add a matching primary key.
```

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:

```
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 |
```

licensed on Creative Commons license for non-commercial use

```
|   2 |    1 | kid2 |   20 |
|   3 | NULL | orp3 |   30 |
|   4 |    1 | kid4 |   40 |
+-----+------+------+------+
4 rows in set (0.00 sec))
```

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
```

licensed on Creative Commons license for non-commercial use

```
testdb=> INSERT INTO Child (cid, pid, cs, csum)
testdb->  VALUES (5, 2, 'kid5', 50);
ERROR:  deadlock detected
DETAIL:  Process 2574 waits for ShareLock on transaction 692; blocked by process 2576.
Process 2576 waits for ShareLock on transaction 691; blocked by process 2574.
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"
testdb=>

-- -------------------------------------
        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.

Pyrrho

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.

licensed on Creative Commons license for non-commercial use

```
SQL> [CREATE TABLE Parent (
> pid  INT NOT NULL,
> ps   VARCHAR(20),
> psum INT DEFAULT 0,
> CONSTRAINT Parent_PK PRIMARY KEY (pid)
> );]
SQL> [CREATE TABLE Child (
> cid  INT NOT NULL ,
> pid  INT DEFAULT 0,
> cs   VARCHAR(20),
> csum INT,
> CONSTRAINT Child_PK PRIMARY KEY (cid),
> CONSTRAINT Child_Parent_FK FOREIGN KEY (pid) REFERENCES Parent (pid)
> );]
SQL> INSERT INTO Parent (pid,ps) VALUES (0,'default parent');
SQL> INSERT INTO Parent (pid,ps) VALUES (1,'pa1');
SQL> INSERT INTO Parent (pid,ps) VALUES (2,'pa2');
SQL> SELECT * FROM Parent;
|---|--------------|----|
|PID|PS            |PSUM|
|---|--------------|----|
|0  |default parent|0   |
|1  |pa1           |0   |
|2  |pa2           |0   |
|---|--------------|----|
SQL> -- step 1, Client A
SQL> BEGIN TRANSACTION;
SQL-T>UPDATE Parent SET psum = 100 WHERE pid = 1;
1 records affected
                                  SQL> -- step 2, Client B
                                  SQL> BEGIN TRANSACTION;
                                  SQL-T>UPDATE Parent SET psum = 200 WHERE pid = 2;
                                  1 records affected
                                  SQL-T>INSERT INTO Child (cid,pid,cs,csum) VALUES (4,1,'kid4',40);

SQL-T>-- step 3, Client A
SQL-T>INSERT INTO Child (cid,pid,cs,csum) VALUES (5,2,'kid5',10);
SQL-T>COMMIT;
SQL>
                                  SQL-T>-- step 4, Client B
                                  SQL-T>COMMIT;
                                  Database testdb incorrectly terminated or damaged
                                  The transaction has been rolled back
                                  SQL>
```

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.

licensed on Creative Commons license for non-commercial use

DB2 SQL PL

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 @
```

licensed on Creative Commons license for non-commercial use

Command Editor 1

```
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 @
```

```
   UPDATE Parent P SET psum =
    (SELECT SUM(csum) FROM Child C WHERE C.pid = old.pid)
   WHERE P.pid = old.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
DB20000I  The SQL command completed successfully.
```

Statement termination character @

Then the test was carried out as follows

Command Editor 1

```
rollback
-- 1. Client A using CE
SET ISOLATION = RS;
UPDATE Parent SET ps = 'new' WHERE pid = 1

-- 3. Client A using CE
SET CURRENT LOCK TIMEOUT = 20;
UPDATE Parent SET psum =
(SELECT SUM(csum) FROM Child WHERE pid = 1)
WHERE pid = 1;
```

```
DB20000I  The SQL command completed successfully.

---------------------------- Commands Entered ----------------------------
SET CURRENT LOCK TIMEOUT = 20;
------------------------------------------------------------------------
SET CURRENT LOCK TIMEOUT = 20
DB20000I  The SQL command completed successfully.

---------------------------- Commands Entered ----------------------------
UPDATE Parent SET psum =
(SELECT SUM(csum) FROM Child WHERE pid = 1)
WHERE pid = 1;
------------------------------------------------------------------------
UPDATE Parent SET psum =  (SELECT SUM(csum) FROM Child WHERE pid = 1) WH
DB20000I  The SQL command completed successfully.
```

Command Editor 2

```
connect to testdb
rollback
-- 2. Client B
SET ISOLATION = RR
SET CURRENT LOCK TIMEOUT = 20
UPDATE Child SET pid = 3 WHERE cid = 3;
INSERT INTO Child (cid, pid, cs, csum) VALUES (4, 1, 'kid4', 40)
```

```
Deadlocks are often normal or expected while processing certain
combinations of SQL statements. It is recommended that you design
applications to avoid deadlocks to the extent possible.

If a deadlock state was reached because of a queuing threshold such as
the CONCURRENTDBCOORDACTIVITIES threshod, increase the value of the
queuing threshold.

sqlcode: -911

sqlstate: 40001


  Related information
  Lock waits and timeouts
  Lock management
  Resolving deadlock problems
```

So deadlock detected now in 2 seconds without waiting the lock timeout!

licensed on Creative Commons license for non-commercial use

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."

licensed on Creative Commons license for non-commercial use

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 ET psum =
    (SELECT SUM(csum) FROM Child C WHERE C.pid = pid)
   WHERE Parent.pid = pid;
END;
```

```
SQLQuery1.sql - WI...B (WIN764\ML (54))*
ALTER TABLE Child
      DROP CONSTRAINT Child_Parent_FK;

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

-- 3. Client A
UPDATE Parent SET psum =
(SELECT SUM(csum) FROM Child WHERE pid = 2)
WHERE pid = 2;

Messages
Msg 1205, Level 13, State 51, Line 2
Transaction (Process ID 54) was deadlocked on lock resour
```

```
SQLQuery2.sql - WI...B (WIN764\ML (55))*
-- 2. Client B
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE Child SET pid = 2 WHERE cid = 3;
INSERT INTO Child (cid, pid, cs, csum)
VALUES (4, 1, 'kid4', 40);

Messages
(2 row(s) affected)

(1 row(s) affected)

(2 row(s) affected)

(1 row(s) affected)
```

licensed on Creative Commons license for non-commercial use

MySQL/InnoDB

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

```
delimiter |
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; |
delimiter
;

delimiter |
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; |
delimiter
;
 delimiter
|
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;
|
delimiter ;
```

And let’s test with the following scenario:

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

licensed on Creative Commons license for non-commercial use

3	UPDATE Parent SET ps = 'new' WHERE pid = 1;	
4		

Following listing has been captured from a test run:

```
-- 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

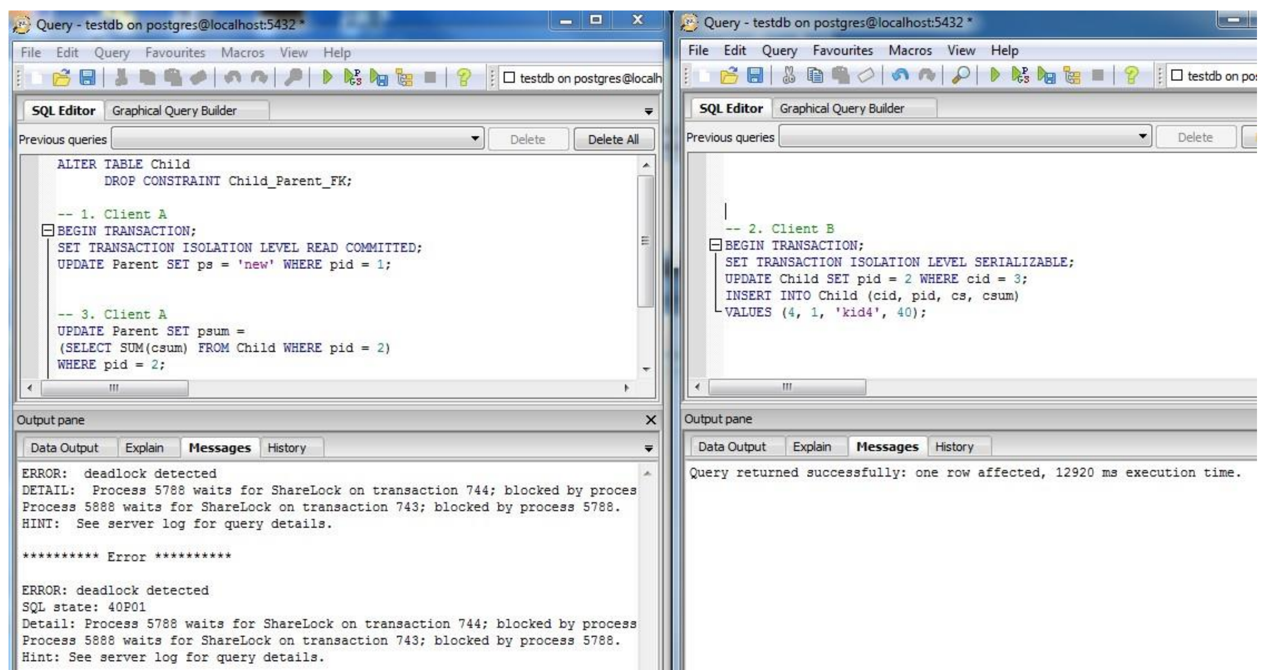
PostgreSQL has implemented triggers in its own way: First a function which is declared to return trigger need to be created, and the trigger to be created has to execute the function (referred as procedure). In PostgreSQL procedure concept is a kind of function.
Triggers can be created on statement-level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

licensed on Creative Commons license for non-commercial use

```
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();
```



Pyrrho

Since a Pyrrho transaction sees contents of the database in the state as it was in the beginning of the transaction, no conflicts, including trigger actions,with concurrent or concurrently committed transactions are noticed before trying to commit the transaction. The first transaction of the concurrent transaction wins, and others will be rolled back.

licensed on Creative Commons license for non-commercial use

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 colum(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

	ISO SQL SQL:2006	DB2 LUW 9.7	Oracle 11g1	SQL Server 2012	MySQL 5.6	PostgreSQL 9.2	Pyrrho 5.2
RULES							
ON UPDATE RESTRICT	Y	Y	N	N	Y	Y	Y
ON UPDATE NO ACTION	Y	Y	N	Y	Y	Y	N
ON UPDATE CASCADE	Y	N	N	Y	Y	Y	Y
ON UPDATE SET NULL	Y	N	N	Y	Y	Y	Y
ON UPDATE SET DEFAULT	Y	N	N	Y	N	Y	Y
ON DELETE RESTRICT	Y	Y	N	N	Y	Y	Y
ON DELETE NO ACTION	Y	Y	N	Y	Y	Y	N
ON DELETE CASCADE	Y	Y	Y	Y	Y	Y	Y
ON DELETE SET NULL	Y	Y	Y	Y	Y	Y	Y
ON DELETE SET DEFAULT	Y	N	N	Y	N	Y	Y
RESTRICT - not deferrable							
NO ACTION - deferrable							

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 );
```

licensed on Creative Commons license for non-commercial use

```
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 used to verify that the parent key value exists.
-- If present, lock parent key's row so it cannot be deleted
-- by another transaction until this transaction is
-- committed or rolled back.
    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 Valid_parent THEN
        CLOSE Dummy_cursor;
    WHEN Mutating_table THEN
        NULL;
END;
```

licensed on Creative Commons license for non-commercial use

```
-- 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.

SQL>
```

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.

SQL>
```

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;
```

licensed on Creative Commons license for non-commercial use

```
-- 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
```

```
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_NO_ACTION DISABLE;
```

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
```

licensed on Creative Commons license for non-commercial use

```
    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;
```

licensed on Creative Commons license for non-commercial use

```
/
-- 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:
--
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
```

licensed on Creative Commons license for non-commercial use

Here we have a sample test run:

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

SELECT * FROM Parent;
       PID PS                         PSUM
---------- -------------------- ----------
         0 default parent                0
         4 pa1                           0
         2 pa2                           0

SELECT * FROM Child;
       CID        PID CS                         CSUM  UPDATE_ID
---------- ---------- -------------------- ---------- ----------
         1          4 kid1                         10
         2          4 kid2                         20
         3            orphan3                      30
```

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:

```
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:

```
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
BEGIN
    UPDATE Child
    SET pid = :new.pid, Update_id = NULL
    WHERE Update_id = Updateseq;
END AFTER EACH ROW;

END pid_CASCADE;
```

licensed on Creative Commons license for non-commercial use

```
/
```

Testing the trigger as follows

```
UPDATE parent set pid = 4 where pid = 1;
1 rows updated.
SELECT * FROM Parent;
       PID PS                         PSUM
---------- -------------------- ----------
         0 default parent                0
         4 pa1                           0
         2 pa2                           0

SELECT * FROM Child;
       CID        PID CS                         CSUM  UPDATE_ID
---------- ---------- -------------------- ---------- ----------
         1          4 kid1                         10
         2          4 kid2                         20
         3            orphan3                      30

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

```
MATCH {FULL | PARTIAL | SIMPLE}
```

This is discussed with examples in the article of Peter Gulutzan and Trudy Pelzer at
https://mariadb.com/kb/en/sql-99-complete-really/20-sql-constraints-and-assertions/constraint_type-foreign-key-constraint/

How this would change the RI triggers?

licensed on Creative Commons license for non-commercial use

Part 4 Comparing SQL/PSM and Implementations

Stored procedures:

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

DBMS		ISO SQL	DB2 Express-C	Oracle XE	SQL Server	MySQL	PostgreSQL	Pyrrho
version		2011	10.5	11.2	2012	5.6	9.2	5.3
document		6IWD6-02-Four	DB2SQLRefVol2	PL/SQL manual	BOL 2008	refman-5.6-en.	postgresql-9.0	PyrrhoBook
							PL/pgSQL	
Parameters:								
IN		default	default	default	default	default	default	default
OUT		YES	YES	Yes	N/A	N/A	Yes	N/A
INOUT		YES	YES	Yes	N/A	N/A	Yes	N/A
Invoking		CALL p()	CALL p()	[EXEC] p(..)	EXEC p @v, ..	CALL p()	CALL p()	CALL p()
positional		YES	YES	YES	YES	YES	YES	YES
by name		YES	YES	YES	YES	YES	YES	?
BEGIN ATOMIC		YES	YES	implicit	compiled procedures	N/A	implicit	implicit
atomic execution context		yes						
COMMIT		YES	YES	YES	YES	YES	N/A	N/A
ROLLBACK		YES	YES	YES	YES	YES	N/A	N/A
label		YES	YES	N/A	N/A	YES	N/A	YES
<<label>>	label:	N/A	YES	YES	yes	YES	N/A	N/A
GOTO label		N/A	YES	YES	YES	YES	N/A	N/A
GET DIAGNOSTICS		definition	partly	N/A	N/A	partly	row_count	full
Exception handlers		N/A	N/A	YES	YES	N/A	YES	N/A
Condition handlers								
CONTINUE		YES	YES	N/A	N/A	YES	N/A	YES
EXIT		YES	YES	N/A	N/A	YES	N/A	YES
REDO		YES	YES	N/A	N/A	N/A	N/A	YES

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.

licensed on Creative Commons license for non-commercial use

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

DBMS	ISO SQL	DB2 Express-C	Oracle XE	SQL Server	MySQL	PostgreSQL	Pyrrho
version	2011	10.5	11.2	2012	5.6	9.2	5.3
document	6IWD6-02-Four	DB2SQLRefVol2	PL/SQL manual	BOL 2008	refman-5.6-en.	postgresql-9.0	PyrrhoBook
Parameters							
IN	default	default	default	default	default	default	default
OUT	N/A	YES	Yes	N/A	N/A	Yes	N/A
INOUT	N/A	YES	Yes	N/A	N/A	Yes	N/A
returns	RETURNS	RETURNS	IS RETURN	RETURNS	RETURNS	RETURNS	RETURNS
SELECT	Yes	YES	YES	YES	YES	YES	YES
other DML	?	N/A	yes 1)	N/A	YES	YES	YES
COMMIT	?	N/A	yes 1)	N/A	N/A	N/A	N/A
ROLLBACK	?	N/A	yes 1)	N/A	N/A	N/A	N/A
SPECIFIC name	YES	YES	N/A	N/A	N/A	N/A	N/A
polymorphism / overloading	YES	YES	yes 2)	N/A	N/A	YES	YES
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.

licensed on Creative Commons license for non-commercial use

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

DBMS		ISO SQL	DB2 Express-C	Oracle XE	SQL Server	MySQL	PostgreSQL	Pyrrho
version		2011	10.5	11.2	2012	5.6	9.2	5.3
document		6IWD6-02-Four	DB2SQLRefVol2	PL/SQL manual	BOL 2008	refman-5.6-en.a	postgresql-9.0	PyrrhoBook
DML triggers								
privilege needed		TRIGGER	TRIGGER	CREATE TRIGGER	ALTER TABLE	TRIGGER	TRIGGER	admin
ON INSERT								
BEFORE	statement	Yes	Yes	Yes	N/A	N/A	Yes	Yes
BEFORE	for each row	Yes	Yes	Yes	N/A	Yes	Yes	Yes
AFTER	for each row	Yes	Yes	Yes	N/A	Yes	Yes	Yes
AFTER	statement	Yes	Yes	Yes	Yes	N/A	Yes	Yes
	INSERTING	N/A	Yes	Yes	N/A	N/A	TG_OP='INSERT'	N/A
ON UPDATE								
BEFORE	statement	Yes	Yes	Yes	N/A	N/A	Yes	Yes
BEFORE	for each row	Yes	Yes	Yes	N/A	Yes	Yes	Yes
AFTER	for each row	Yes	Yes	Yes	N/A	Yes	Yes	Yes
AFTER	statement	Yes	Yes	Yes	Yes	N/A	Yes	Yes
	UPDATING	N/A	Yes	Yes	UPDATE()	N/A	TG_OP='UPDATE	N/A
ON DELETE								
BEFORE	statement	Yes	Yes	Yes	N/A	N/A	Yes	Yes
BEFORE	for each row	Yes	Yes	Yes	N/A	Yes	Yes	Yes
AFTER	for each row	Yes	Yes	Yes	N/A	Yes	Yes	Yes
AFTER	statement	Yes	Yes	Yes	Yes	N/A	Yes	Yes
	DELETING	N/A	Yes	Yes	N/A	N/A	TG_OP='DELETE'	N/A
INSTEAD OF		Yes	Yes	Yes	Yes	N/A	N/A	N/A
Multi-event		Yes	Yes	Yes	?	N/A	?	N/A
Referencing	old/new ROW	Yes	Yes	Yes	N/A	N/A	N/A	Yes
	NEW TABLE	Yes	Yes	N/A	INSERTED	N/A	N/A	N/A
	OLD TABLE	Yes	Yes	N/A	DELETED	N/A	N/A	N/A
DDL triggers		N/A	N/A	Yes	Yes	N/A	N/A	N/A
LOGON		N/A	N/A	Yes	Yes	N/A	N/A	N/A
SCHEMA		N/A	N/A	Yes	N/A	N/A	N/A	N/A
DATABASE		N/A	N/A	Yes	N/A	N/A	N/A	N/A
cascading of triggers?			controlled	not allowed?				
transaction demarkation		?		yes	yes	yes	N/A	N/A

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.

licensed on Creative Commons license for non-commercial use

Part 5 SQL-invoked External Routines

5.1 Introduction to SQL-invoked External Routines

Oracle, Sybase and Microsoft SQL Server have had procedural SQL implementations already before the SQL PSM standard, but still in the nineties, before version 7.1 of IBM's DB2 [UDB/LUW], the SQL-invoked external routines were the only available stored routine technology for DB2 users. SQL-invoked external routines means routines written in other languages, such as Ada, C/C++, COBOL, Fortran, MUMPS, Pascal or PL/1, and to be invoked from SQL. Typically, the executable code for these routines is stored in local library files located also physically outside the database as presented in Figure 5.1. External routines in C/C++ had become available earlier in SQL Server, but **without support for transaction context** (Melton, 1998). Support for external routines in Java and C/C++ has been included in the Oracle DBMS since version 8i.

SQL-invoked external routines provide the means for extending the SQL implementation by accessing operating system services, function libraries, existing applications, or extending functionalities of the SQL implementation. A simple and generally useful example is our SLEEP procedure wrapper on the Java sleep function, extending the DB2 SQL (created in Listing 5.3.1 and used in Appendix 3). Besides being directly useful for developers in concurrency tests, this shows how easy it is to extend SQL by implementing wrappers on built-in Java functions.

An advanced example on extending the SQL functionalities is presented in the article "Custom Aggregates in SQL Server" by Mika Wendelius at http://www.codeproject.com/Articles/170061/Custom-Aggregates-in-SQL-Server.

SQL-invoked routines stored in the file system

According to ISO SQL PSM standard the external routines are registered in the SQL environment (system tables) by the CREATE PROCEDURE/FUNCTION declaration as presented earlier in Figures 1.2.1 and 1.2.2 of Chapter 1 like following:

```
CREATE PROCEDURE <procname>  ( <parameter list> )
LANGUAGE C
PARAMETER STYLE SQL
DYNAMIC RESULT SETS <n>
{ NO SQL | READS SQL DATA | MODIFIES SQL DATA }
EXTERNAL NAME '<assembly file>:<classname>!<method>' ;
```

The actual architectural and implementation details depend on the DBMS product and the operating system platform used. To provide a general explanation of the technologies, Figure 5.1.1 presents a simplified overview and the sequence steps relating to the use of SQL-invoked external routines:

licensed on Creative Commons license for non-commercial use

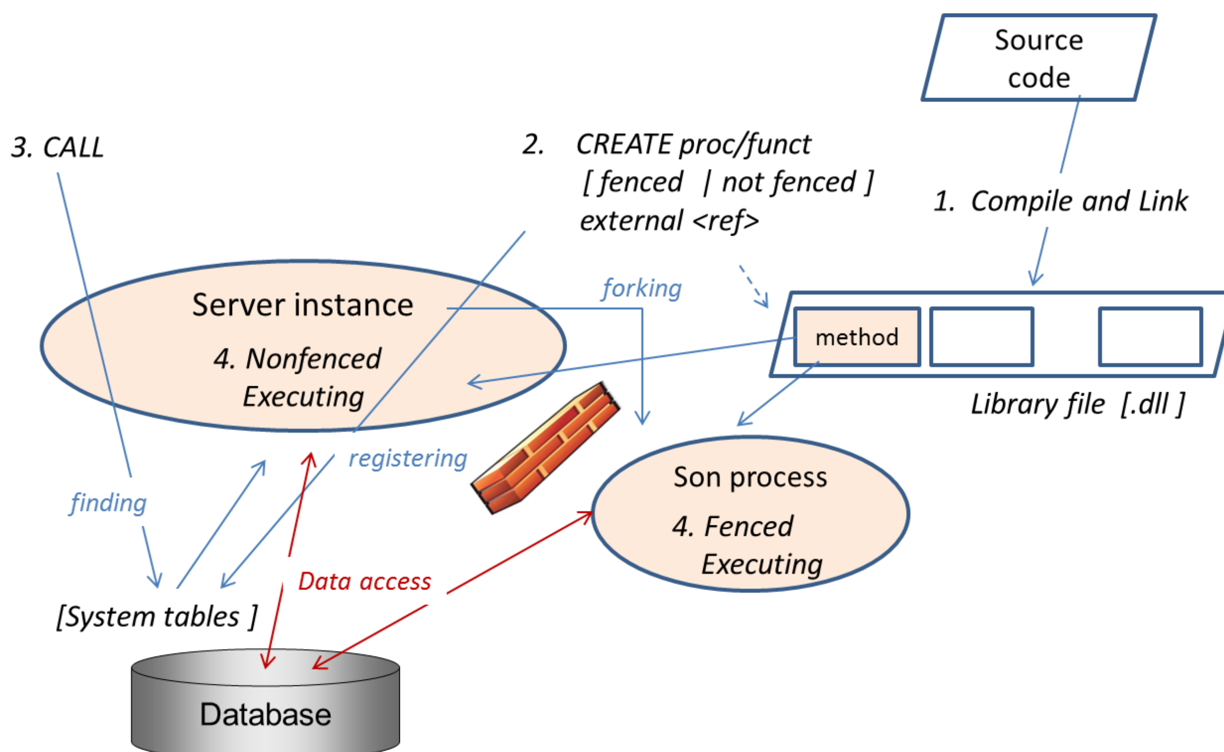


Figure 5.1.1. Overview and steps of uses of for SQL-invoked external routines

In step 1 the program code for the stored routine(s) is compiled and linked into a library file (typically: a dll file) in a file system folder which is local to the database server instance. The code may contain methods for multiple procedures and functions.

In step 2 the SQL wrapper for the external stored routine is created into the database, actually as definitions in the system tables of the database. The **EXTERNAL** clause in the command refers to the library file and the method to be used as the execution body of the routine.

When a client invokes (step 3) an external routine, the server locates the external code to be executed by means of the reference stored in the system tables.

For step 4, in the extended DB2 SPL syntax, the CREATE command in step 2 can define the routine as **FENCED** or NOT FENCED. FENCED execution means that the external code is not processed by the server processes. Instead, the code executes in the form of a separate son process, forked by the server. Even if this means some performance overhead, this processing in a separate address space protects the server from aborting in case the routine code aborts. FENCED execution is the default behavior for external routines in DB2, as an act of prioritizing reliability higher over performance. The FENCED keyword is not included in the SQL PSM standard, but it comprises the typical mode of execution for external routines in the DBMS products.

In Chapter 5.2 we will experiment with SQL-invoked C routine examples using DB2 Express-C.

SQL-invoked Java routines

According to Jim Melton and Andrew Eisenberg (2000), the SQLJ group of experts from database and database tool vendor companies (Oracle, IBM, Sybase, Tandem, JavaSoft, Microsoft, and Informix),

licensed on Creative Commons license for non-commercial use

started to develop the specifications for embedding SQL in the Java language code, back in 1997. The developed set of specifications consists of the following parts:

SQLJ Part 0 – Embedded SQL API in the Java Language (generally known as “The SQLJ”)
SQLJ Part 1 – SQL Routines Using the Java Programming Language
SQLJ Part 2 – Java Classes as SQL Types

Later, the SQLJ Part 0 was adopted in the ISO SQL standard Part 10 “Object Language Bindings (SQL/OLB)” and the SQLJ Part 1 into Part 13 “SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)”, with tutorials on SQLJ Part 2 topics in its Annex H.

Figure 5.1.2 presents an overview on the uses of external Java routines according to the SQL standard Part 13 JRT, assuming that the Java virtual machine (JVM) is integrated into the database server instance. It depends on the DBMS implementation whether the JVM is activated when the server instance is initiated, or whether it is activated by the invoked routine.

The Java programs are compiled and the compiled class files are packed into jar files outside the database environment. The jar files are loaded into the database by the built-in procedure SQLJ.INSTALL_JAR (kind of an extended DDL command) giving it a local name in the database environment. SQLJ is a new built-in schema for the procedures INSTALL_JAR, REPLACE_JAR, REMOVE_JAR, and ALTER_JAVA_PATH, and for managing the jar files in the database. The local name of the jar file is used in the EXTERNAL NAME clause of the CREATE PROCEDURE/FUNCTION commands (wrapping SQL signatures) for locating the Java class and method to be used for the routine body, as follows:

```
EXTERNAL NAME '<jar file>:<class name>.<method>[(<parameter data types>)]'
```

where the <parameter data types> specification is needed when the data types are not directly mappable (simple Java data types). See Table 1.1.

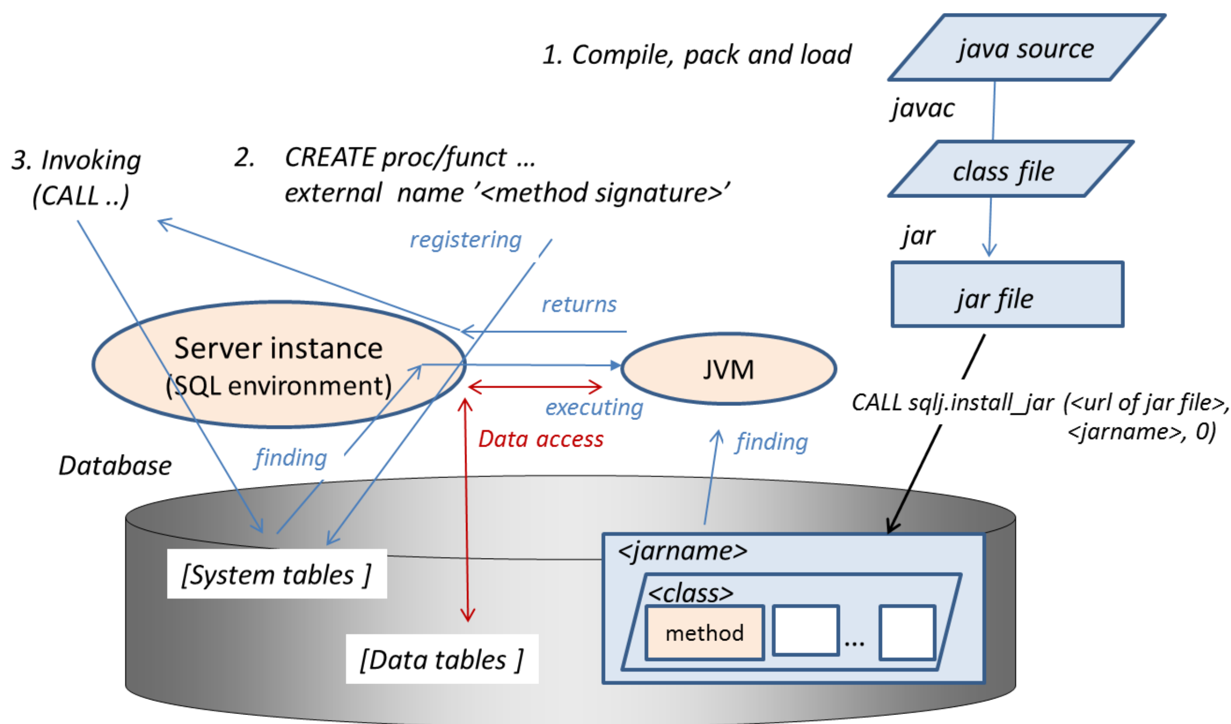


Figure 5.1.2. Technology overview and steps of uses of external Java routines according to the SQL standard Part 13 JRT

licensed on Creative Commons license for non-commercial use

SQL data type:	Simply mappable	Object mappable
CHARACTER		String
VARCHAR		String
LONGVARCHAR		String
NUMERIC		java.math.BigDecimal
DECIMAL		java.math.BigDecimal
BIT *)	Boolean	Boolean
TINYINT	Byte	Integer
SMALLINT	Short	Integer
INTEGER	Int	Integer
BIGINT	Long	Long
REAL	Float	Float
FLOAT	Double	Double
DOUBLE PRECISION	Double	Double
BINARY		byte[]
VARBINARY		byte[]
LONGVARBINARY		byte[]
DATE		java.sql.Date
TIME		java.sql.Time
TIMESTAMP		java.sql.Timestamp

Table 1.1 Mappable SQL and Java data types (Source: Melton and Eisenberg 2000)
*) SQL data type BIT has been replaced by BOOLEAN in the ISO SQL standard, but neither of these is usually implemented

In Chapter 5.2 we will experiment with the DB2 implementation based on SQL/JRT using DB2 Express-C in the DBTechNet DebianDB laboratory environment.

licensed on Creative Commons license for non-commercial use

Experiments on other platforms

Oracle has brought the support of SQL-invoked external routines in Java language onto a new level of integration by integrating the Java compiler into the database server, as we will see in Chapter 5.3. Unfortunately the Oracle JVM is not included in the free Oracle XE edition, so for our experiments you need to have a commercial Oracle edition or download and install the Oracle instance from otn.oracle.com, which you can use only for educational purposes.

Microsoft has followed the JVM integration development by .NET byte code engine CLR integration in SQL Server, as we will see later. Since the SQL Server runs only on Windows platforms, you need to either have SQL Server already installed on your computer, or you can download and install the SQL Server Express edition which is freely available Microsoft.

IBM also provides the .NET CLR integration in DB2, used in some experiments, later in this document.

Both the Java byte code and the CLR engines support the use of byte code from other languages. For further study, interesting technologies of external routines implementations include languages like Perl, Python, etc., for example, in MySQL (Curtis and Herman 2008) and PostgreSQL.

Priorities and Challenges

Without getting too enthusiastic on possibilities provided by the technologies for SQL-invoked external routines, we need to set and remember the priorities for application development:

1. Reliability, especially by proper transaction programming
2. Security
3. Performance
4. Scalability, preserving reasonable performance while number of concurrent clients is increasing.

In this tutorial we are mainly focusing on reliability.

SQL external routines extend computing capabilities to use existing statistical and other function libraries, existing application modules written in other languages, and open integration with the “world outside the database”. However, often better alternative for open integrations can be provided by the extended file based data types and streaming functions of the SQL implementations. Also it is important to bear in mind that too fancy integrated application features may not support the current transactional context of the invoking SQL session.

Routines are preferably run locally in the server computer, and single-user/client-side libraries which are not designed for re-entrant use don’t suit in transactional environment (Johnson & Reimer). Also libraries with database connections of their own would violate atomicity of the current transactional context, since multi-connection transactions need to be managed by separate transaction manager while the extra database connections in external routines cannot participate in such distributed transactions.

In case the routine will access the database content, it needs a database connection. For this purpose it will only open the default connection which uses the connection context and the transaction context of the invoking SQL-session.

Compatibility and Impedance Mismatching Issues

Well known impedance mismatch problem between SQL’s set level processing and the record-at-a-time processing in programming languages, which has been sorted out by cursor processing. In JDBC the

licensed on Creative Commons license for non-commercial use

cursor processing is wrapped by navigational ResultSet objects.

More complicated impedance mismatch between data types of SQL and programming languages, and processing of NULL values. This has been solved in Embedded SQL using for every possible parameter an additional null indicator parameter, and its value -1 indicates that the actual parameter should be understood to have NULL value, while value 0 indicates that the value of the actual parameter can be used.

There is impedance mismatches between SQL data types and data types of the used programming language.
For C/C++ languages these are solved by macros in library files of C language. Some basic SQL-92 data types usually have compatible built-in data types in the programming languages, but as a whole the data type compatibility issues may need books of their own. In our examples we will focus on the easy cases only.

Experiment plan

We start experiments with one of the most important programs for learners and developers, the "Hello World" example ☺, which is used as "first steps" to verify basics of the technology to be studied.

As we are just focusing on the basics and in transaction programming, we will use a minimalistic procedure example "insertRow" in which we insert a simple row into a small table T created as follows:

```
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, n INT, s VARCHAR(10));
```

Without concern on reasonability of building external routines just for an INSERT statement in application terms, our purpose Is only to test *technically* the connection and transaction contexts, passing parameters, use of SQL NULL values (on columns n and s), the available diagnostics, and exception handling.

The planned signature part of our procedure consists of the following

```
CREATURE PROCEDURE InsertRow (IN id INT,
                              IN n INT,
                              IN s VARCHAR(20),
                              OUT rcount INT,
                              OUT msg VARCHAR(200))
                 RETURNS rc INT
```

In input parameters the procedure gets values for the columns id, n and s, and in output parameter rcount the procedure returns number of the inserted rows: for successful execution it returns the number 1, and in case of failure it returns the number 0. The output parameter msg returns an error message (if any). Lastly, upon successful execution the function returns a 0, or a non-zero error code set by the server and communicated by means of the rc (return code) parameter. The SQL/PSM standard does not define the return code, but some products support it.

To keep things simple, we need to make compromises in applying the parameters. For example, instead of the error message we will use just SQLSTATE for the C/ESQL example in DB2.

Our test plan is to verify

1) successful execution,
2) proper exception handling in unsuccessful execution, and
3) proper transactional execution

licensed on Creative Commons license for non-commercial use

in the context of connectivity and transaction processing of the invoking SQL-client.

As an example of traditional external routines we will first look at DB2 procedures written in the C language, and then proceed to consider DB2 procedures written in the Java language. They can both be tested in the free DBTechNet DebianDB laboratory environment. We will continue with Java procedures of the Oracle integrated Java virtual machine (JVM). Unfortunately, the Oracle JVM is not included in the free Oracle XE edition which is pre-installed in the DBTechNet DebianDB laboratory environment, so in order to experiment with the JVM one needs to register with and download the Oracle edition from otn.oracle.com.

Finally, we look on Windows platform at the integrated CLR engine in SQL Server and running .NET classes inside the database.

5.2 External DB2 Stored Routines written in the C Language

External routines of DB2 in the C language use the embedded SQL (ESQL)[5] of DB2. Our examples below are based on the CSTP example in the IBM Knowledge Center web document titled ”DB2 Version 9.7 for Linux, UNIX, and Windows” (”Parameter style SQL C and C++ procedures” chapter, at http://www-01.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.apdv.routines.doc/doc/c0023762.html).

In addition to our basic InsertRow example, we present in the following a simple “Hello World” function, and a modified version of IBM’s CSTP example returning a result set. The source codes of all these routines are included in the library file source presented in Listing 5.2.1. The “lib” part in the naming of the “cstrlib.sqc” file is there to signify the fact that the source file may contain multiple routine bodies.

Listing 5.2.1 Library file `cstrlib.sqc` using C/ESQL

```
/*****************************************************
DBTechNet / Martti Laiho 2015-10-02

The source code of the library of external C/ESQL routines

*/
#include <stdio.h>
#include <sqlca.h>
#include <sqludf.h>
#include <stdlib.h>
#include <string.h>
// -----------------------------
extern SQL_API_RC SQL_API_FN hello (
  char  message[13],
  SQLUDF_NULLIND *messageNullInd,
  SQLUDF_TRAIL_ARGS )
{
  EXEC SQL INCLUDE SQLCA;

  sprintf (message, "Hello World\n");
  return ;
}
// -------------------------------------
extern SQL_API_RC SQL_API_FN insertRow (
  sqlint32 *id,
  sqlint32 *n,
```

[5] Use of ESQL of DB2 is explained in the “Getting Started With DB2 Application Development” eBook in the “DB2 on Campus” book series

licensed on Creative Commons license for non-commercial use

```
  char  s[21],
  sqlint32 *rcount,
  char  sqlstate[6],
  SQLUDF_NULLIND *idNullInd,
  SQLUDF_NULLIND *nNullInd,
  SQLUDF_NULLIND *sNullInd,
  SQLUDF_NULLIND *rcountNullInd,
  SQLUDF_NULLIND *sqlstateNullInd,
  SQLUDF_TRAIL_ARGS )
{
EXEC SQL INCLUDE SQLCA;
sqlint32 rc;
char temp[6];
EXEC SQL BEGIN DECLARE SECTION;
sqlint32 ID;
sqlint32 N;
char S[21];
EXEC SQL END DECLARE SECTION;

     ID = *id;
     strcpy (S, s);
     if (*sNullInd == -1 && *nNullInd == -1) {
        EXEC SQL INSERT INTO T (id, n, s) VALUES (:ID, NULL, NULL) ;
     }
     else {
        if (*sNullInd == -1 && *nNullInd == 0) {
           N = *n;
           EXEC SQL INSERT INTO T (id, n, s) VALUES (:ID, :N, NULL) ;
        }
        else {
           if (*sNullInd == 0 && *nNullInd == -1) {
              strcpy (S, s);
              EXEC SQL INSERT INTO T (id, n, s) VALUES (:ID, NULL, :S) ;
           }
           else {
              N = *n;
              strcpy (S, s);
              EXEC SQL INSERT INTO T (id, n, s) VALUES (:ID, :N, :S) ;
           }
        }
     }
     *rcount  = sqlca.sqlerrd[2];
     strncpy (temp, sqlca.sqlstate, 6);
     sprintf (sqlstate, "%s", temp);
     return -1;  // doesn't have impact on the Return Status!
}
// -------------------------------------
extern SQL_API_RC SQL_API_FN cstp (
  sqlint32 *inParm,
  sqlint32 *outParm,
  SQLUDF_NULLIND *inParmNullInd,
  SQLUDF_NULLIND *outParmNullInd,
  SQLUDF_TRAIL_ARGS )
{
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
     sqlint32 sql_inParm2;
EXEC SQL END DECLARE SECTION;
     if (*inParmNullInd == -1) {
         sql_inParm2 = 0 ;
         *outParmNullInd = -1;
     }
     else {
         sql_inParm2 = *inParm;
         *outParm = sql_inParm2 + 1;
         *outParmNullInd = 0;
     }
EXEC SQL DECLARE cur2 CURSOR FOR
```

licensed on Creative Commons license for non-commercial use

```
          SELECT *
          FROM T
          WHERE id = :sql_inParm2;
EXEC SQL OPEN cur2;
     return (0);
}
```

In the DBTechNet DebianDB laboratory environment we use the Gnu C compiler which does not accept the "C" option after the keyword “extern” as used in the cstp example on IBM’s web site. Other changes made by us include the use of sqlint32 data types only, and making use of the null indicators in the cstp method.

As the first parameter inParm the procedure cstp expects to get as input an integer value, and the second parameter outParm is used as output parameter in which the procedure simply returns the integer value of inParm added by 1. The NullInd parameters serve as null indicators of inParm and outParm parameters. SQLUDF_TRAIL_ARGS is a C language macro in sqludf.h generating the extra output parameters sqlState, qualified and special names of the procedure, and sqlMessageText (however, it is still open how to access these values).

In addition cstp opens an SQL cursor for returning in a result set to the invoker the row defined by the inParm value.

Studying the source code we can see that there is no explicit mark on database connection. This is because in external C/ESQL routines to be called will work **implicitly in the connection and transaction context of the invoking SQL session.**

To keep the experiment simple, skipping the security and discussion on privileges, we apply the experiment using the db2inst1 login, who is the installer and creator of the DB2 Express-C instance in our database laboratory.

We start by precompiling the source file cstrlib.sqc into a modified C program cstrlib.c and into the bind file cstrlib.bnd using the DB2 precompile command PREP. The bind file contains the execution plans (sections) of the embedded SQL commands, and by the BIND command we load these into the optimized package CSTRLIB in our database TESTDB:

```
# Precompile the library file:
cd $HOME/StoredRoutines/ExternalRoutines
db2 CONNECT TO TESTDB
db2 PREP cstrlib.sqc BINDFILE
db2 BIND cstrlib.bnd
db2 TERMINATE
```

Using the Gnu C compiler gcc, we compile and link the generated C program into the executable library file `cstrlib`. The option “-m32” is needed on the 32bit platform, whereas “-m64” need to be used on our future 64bit database laboratory environment. Finally we link the `cstrlib` file by the Linux `ln` command to be seen by the DB2 server in the /home/db2inst1/sqllib/function folder as follows:

```
# Compile the generated C code into object code
export DB2PATH=$HOME/sqllib
gcc -c -m32 -I$DB2PATH/include cstrlib.c

# Link the object code into program code
gcc -o cstrlib -m32 -shared -D_REENTRANT -L$DB2PATH/lib -ldb2 cstrlib.o
```

licensed on Creative Commons license for non-commercial use

```
# Link by ln command the program code available for the DB2 server
rm $HOME/sqllib/function/cstrlib
ln cstrlib $HOME/sqllib/function/cstrlib
```

Now we can experiment with these routines, starting with the Hello World function.

Hello World

As the SQL-client utility we use the command line processor db2, called also as CLP. The command line option “+c” sets SQL session to use transactional mode, and the option “-t;” sets CLP to expect semicolon “;” as the terminating character of all statements. First we need to connect to our TESTDB database using the CONNECT command:

```
db2 +c -t;
CONNECT TO TESTDB;
```

For the “hello” method in Listing 5.1.1 we create the wrapping SQL function signature as follows:

```
CREATE OR REPLACE FUNCTION c_hello ()
 RETURNS VARCHAR(12)
 LANGUAGE C
 SPECIFIC c_hello
 PARAMETER STYLE SQL
 READS SQL DATA
 FENCED  THREADSAFE
 EXTERNAL NAME 'cstrlib!hello';
```

Now we can test the function, invoking it as a derived column on the DB2’s single-row dummy table SYSIBM.SYSDUMMY1 as follows:

```
db2 => SELECT c_hello() "my message" FROM SYSIBM.SYSDUMMY1;

my message
------------
Hello World

  1 record(s) selected.

db2 =>
```

InsertRow

Next we experiment with the routine c_insertRow, for which we create/recreate the test table T with one initial row, as follows:

```
DROP TABLE T;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, n INT, s VARCHAR(20));
INSERT INTO T (id, s) VALUES (1, 'first');
COMMIT;
```

To make the routine available for SQL-invoking, we create the following SQL wrapper as the procedure signature (the “c_” prefix differentiates the procedure from the corresponding Java procedure):

```
CREATE OR REPLACE PROCEDURE c_insertRow (IN id INT, IN n INT, IN s VARCHAR(21),
       OUT rcount INT,
       OUT sqlstate CHAR(5))
 LANGUAGE C
 SPECIFIC c_insertRow
 PARAMETER STYLE sql
```

licensed on Creative Commons license for non-commercial use

```
 MODIFIES SQL DATA
 FENCED THREADSAFE
 EXTERNAL NAME 'cstrlib!insertRow';
COMMIT;
```

By setting the SPECIFIC name identical to the actual procedure name we eliminate the possibility of accidental creating extra procedures with the same name in DB2 (which otherwise allows name overloading).

Comparing the procedure signature with the insertRow function in the C code, we notice that the NULL indicators don't appear at the SQL (procedure) level, however the corresponding SQLUDF_NULLIND macros of the sqludf.h library need be included in the parameters list section of the C (ESQL) code.

Note: The error message token sqlerrmc of the SQLCA Communication Area would require complicated interpretation, so to keep the experiment simple we have replaced the error message parameter by the SQLSTATE value of SQLCA.

To allow use of the procedure also to other users we simply grant the execute privilege on the procedure signature to PUBLIC.

```
GRANT EXECUTE ON PROCEDURE c_insertRow TO PUBLIC;
COMMIT;
```

Considering security, since the file containing the procedure body is in the folder available to DB2 and not to the public, no direct access to the library file is needed for other users but the db2inst login user, so we can say that the code is fairly safe.

If we want to run the test as user student, then the db2inst1 (table owner) user needs to grant read access on the T table to the student user, as follows:

```
GRANT SELECT ON TABLE T TO USER student;
COMMIT;
```

Finally we can test the procedure using db2 CLP in transactional mode, for example as the student user, varying with parameters as follows:

```
db2 +c -t;
CONNECT TO TESTDB;
SELECT * FROM db2inst1.T;
CALL db2inst1.c_insertRow (2, 7, 'second', ?, ?);
CALL db2inst1.c_insertRow (2, 8, 'duplicate', ?, ?);
CALL db2inst1.c_insertRow (3, NULL, 'third', ?, ?);
CALL db2inst1.c_insertRow (4, 9, NULL, ?, ?);
CALL db2inst1.c_insertRow (5, NULL, NULL, ?, ?);
SELECT * FROM db2inst1.T;
ROLLBACK;
SELECT * FROM db2inst1.T;
```

Here follow are our sample test results:

```
db2 => CALL db2inst1.c_insertRow (2, 7, 'second', ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 1

  Parameter Name  : SQLSTATE
  Parameter Value : 00000
```

licensed on Creative Commons license for non-commercial use

```
  Return Status = 0
db2 => CALL db2inst1.c_insertRow (2, 8, ' duplicate', ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 0

  Parameter Name  : SQLSTATE
  Parameter Value : 23505

  Return Status = 0
db2 => CALL db2inst1.c_insertRow (3, NULL, 'third', ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 1

  Parameter Name  : SQLSTATE
  Parameter Value : 00000

  Return Status = 0
db2 => CALL db2inst1.c_insertRow (4, 9, NULL, ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 1

  Parameter Name  : SQLSTATE
  Parameter Value : 00000

  Return Status = 0
db2 => CALL db2inst1.c_insertRow (5, NULL, NULL, ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 1

  Parameter Name  : SQLSTATE
  Parameter Value : 00000

  Return Status = 0
db2 => SELECT * FROM db2inst1.T;

ID          N           S
----------- ----------- --------------------
          1           - first
          2           7 second
          5           - -
          3           - third
          4           9 -

  5 record(s) selected.

db2 => ROLLBACK;
DB20000I  The SQL command completed successfully.
db2 => SELECT * FROM db2inst1.T;

ID          N           S
----------- ----------- --------------------
          1           - first

  1 record(s) selected.
```

licensed on Creative Commons license for non-commercial use

```
db2 =>
```

This proves that the SQL-invoked external routine c_insertRow was executed in the connection context and transactional context of our CLP session.

The schema qualification "db2inst1." in the commands above was not necessary
If we want to run the test as user student, then for the use of SELECT commands db2inst1 login needs to allow first the read access to table T by command

```
GRANT SELECT ON TABLE T TO USER student;
COMMIT;
```

CSTP

Our last external C/ESQL routine experiment presented in Listing 5.1.1 is a modified version of the CSTP example from the "DB2 Version 9.7 for Linux, UNIX, and Windows" IBM Knowledge Center web document ("Parameter style SQL C and C++ procedures" chapter at http://www-01.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.apdv.routines.doc/doc/c0023762.html).

A similar example is presented also in the IBM DB2 9.7 manual titled "Developing User-defined Routines (SQL and External)", Nov 2009.

The CSTP example reads an integer and returns value increased by 1 in an output parameter, but also uses the entered integer as a key for reading the corresponding row from table T, returning the table row in the result set.

The wrapping SQL procedure signature is created as follows:

```
CREATE OR REPLACE PROCEDURE c_stp ( IN inParm INT, OUT outParm INT )
 LANGUAGE C
 SPECIFIC c_stp
 PARAMETER STYLE sql
 DYNAMIC RESULT SETS 1
 FENCED
 THREADSAFE
 EXTERNAL NAME 'cstrlib!cstp' ;
COMMIT;
```

These are the our test cases for CSTP:

```
CALL c_stp (1, ?);    -- key for existing row ;
CALL c_stp (0, ?);    -- key for non-existing row ;
CALL c_stp (NULL, ?); -- unknown key ;
```

Below are our test results. As we can see, the db2 CLP client program prints NULL values as hyphen marks "-".

```
db2 => CALL c_stp (1, ?);    -- key for existing row ;

  Value of output parameters
  --------------------------
  Parameter Name  : OUTPARM
  Parameter Value : 2


  Result set 1
  --------------
```

licensed on Creative Commons license for non-commercial use

```
  ID          N           S
  ----------- ----------- --------------------
            1           - first

  1 record(s) selected.

  Return Status = 0
db2 => CALL c_stp (0, ?);    -- key for non-existing row ;

  Value of output parameters
  --------------------------
  Parameter Name  : OUTPARM
  Parameter Value : 1


  Result set 1
  --------------

  ID          N           S
  ----------- ----------- --------------------

  0 record(s) selected.

  Return Status = 0
db2 => CALL c_stp (NULL, ?); -- unknown key ;

  Value of output parameters
  --------------------------
  Parameter Name  : OUTPARM
  Parameter Value : -


  Result set 1
  --------------

  ID          N           S
  ----------- ----------- --------------------

  0 record(s) selected.

  Return Status = 0
db2 =>
```

DB2 troubleshooting

The existing DB2 packages can be checked as follows

```
db2 list packages
                                Bound      Total                           Isolation
Package     Schema    Version   by         sections    Valid   Format   level
Blocking
----------- --------- --------- --------- ----------- ------- -------- --------- ----
CSTRLIB     DB2INST1            DB2INST1             5 Y       0        CS        U

  1 record(s) selected.
```

The default isolation level for packages is cursor stability (CS). Generally, this can be set by specifying the corresponding BIND command options. This may contradict with the isolation level of the transactional context of the invoking SQL session!

While experimenting with various settings in procedure code, due to the caching of the latter, it may be

licensed on Creative Commons license for non-commercial use

necessary to reboot the DB2 instance before the rebinding of the code. The DB2 commands for instance stopping and (re-)initiation are the following:

```
db2stop
db2start
```

In case we want to get rid of the package in the database, this can be done by the db2inst1 user as follows:

```
db2 -t;
CONNECT TO TESTDB;
DROP FUNCTION c_hello;
DROP PROCEDURE c_insertRow;
DROP PROCEDURE c_stp;
DROP PACKAGE db2inst1.cstrlib;
TERMINATE;
```

5.3 Java Routines in the DB2 JVM

In DB2 the Java routines are external routines, the compiled Java classes of which are packed in a jar file and loaded into the database by SQLJ.INSTALL_JAR procedure according to the SQL/JRT specification. The routines are made available for SQL sessions registering them in the system tables in the database by CREATE PROCEDURE or CREATE FUNCTION declarative statements. The procedures can be invoked by CALL statement from application, or stored procedure, or even from triggers. Functions can be used where corresponding expressions can be used, for example expressions in result columns of SELECT clause of SELECT, columns in VALUE clause of INSERT, or in search conditions of SELECT, UPDATE or DELETE.

The domain of the DB2 Java routines is the local database only, so no actions to outer world, for example accessing files, are permitted. Developing User-defined Routines manual lists the restrictions on Java routines called from triggers.

Writing DB2 routines in Java requires knowledge on Java/JDBC programming, DB2 stored SQL routines, and fitting these technologies together, especially in parameter passing. The parameter passing is controlled in the stored procedure signature (declaration) by the PARAMETER STYLE JAVA clause, or by the DB2 specific PARAMETER STYLE DB2GENERAL. We will stick to the STYLE JAVA. Table 43 in the "Developing User-defined Routines" manual lists the proper mappings of DB2 SQL data types[6] to corresponding Java data types.

Challenging issue is the mapping of Java routine parameters to the SQL OUT and INOUT parameters, since Java doesn't support passing variables by reference. As workaround in Java routines the scalar OUT and INOUT parameters are passed as single item arrays. (IBM Knowledge Center, "Parameter styles for external routines")

An array of ResultSet objects need to be appended to the parameter list for every dynamic result set generated in the procedure, as demonstrated by the following example modified from the "Developing User-defined Routines" manual topic "Parameters in Java routines" in chapter 7.

Listing 5.3.1 presents our example Java class containing collection of methods which we will later register by SQL wrappers to be used as SQL-invoked external procedures or functions in our test database.

[6] It is worth to remember that the DB2 data types are not exactly same as the data types in ISO SQL standard, and SQL data types of many other DBMS products may differ even more.

licensed on Creative Commons license for non-commercial use

Listing 5.3.1. Java source of jstrlib.java for the routine bodies

```
/* DBTechNet / Martti Laiho
Examples of DB2 external Java stored routines
-----------------------------------------------
to be run by login db2inst1 in the DBTechNet DebianDB database laboratory

*/
import java.sql.*;

public class jstrlib {

  public static String hello ()
  {
    return "Hello World!";
  }

  public static void sleep (int secs)
  {
    try {
      Thread.sleep(secs * 1000);
    }
    catch (Exception e) {}
  }

  // method without exception handling:
  public static void insert1 (int id,
                              String s,
                              int[] rcount
                             ) throws SQLException
  {
    Connection conn = null;
    PreparedStatement pstmt = null;
      conn = DriverManager.getConnection("jdbc:default:connection");
      String sql = "INSERT INTO T (id, s) VALUES (?, ?)";
      //Prepare the INSERT
      pstmt = conn.prepareStatement( sql );
      pstmt.setInt( 1, id );
      pstmt.setString( 2, s );
      rcount[0] = pstmt.executeUpdate();
      return;
  }

  // method for experimenting with exception hanling:
  public static void insertRow (int id,
                                int n,
                                int n_ind,
                                String s,
                                int[] rcount,
                                int[] sqlcode,
                                String[] sqlstate,
                                String[] errmsg,
                                ResultSet[] rs
                               ) throws SQLException
  {
    int sqlCode = 0;
    Connection conn = null;
    PreparedStatement pstmt = null;
    String sql = null;
    Integer N = null;
    String sqlState = "00000";
```

licensed on Creative Commons license for non-commercial use

```
    String errMsg = "";
    try {
      conn = DriverManager.getConnection("jdbc:default:connection");
      if (n_ind == 0) { N = n; }
      sql = "INSERT INTO T (id, n, s) VALUES (?, ?, ?)";
      pstmt = conn.prepareStatement( sql );
      pstmt.setInt( 1, id );
      pstmt.setInt( 2, N );
      pstmt.setString( 3, s );
      rcount[0] = pstmt.executeUpdate();
      pstmt.close();
      // Returning the table contents in a result set
      sql = "SELECT * FROM T";
      pstmt = conn.prepareStatement( sql );
      rs[0] = pstmt.executeQuery();
    }
    catch (SQLException se) {
      rcount[0]= 0;
      sqlState = se.getSQLState();
      sqlCode  = se.getErrorCode();
      errMsg   = se.getMessage();
    }
    finally {
      sqlcode[0]  = sqlCode;
      sqlstate[0] = sqlState;
      errmsg[0]   = errMsg;
      return;
    }
  }
}
```

The hello() method can be used to verify the language mappings, but the sleep() method is really useful example on mapping Java built-in functions for extending the SQL implementation.

Comparing with data access of client-side Java/JDBC programming, server-side programming of DB2 does not support autocommit. The connection string “jdbc:default:connection” doesn’t really open a new connection, but joins the procedure execution in the context of the invoking session’s connection and its current transaction. The routine should not end the transaction, nor close the connection.

Coping with SQL NULLs is tricky: Java does not support null values for built-in data types, but Java objects can be assigned to null values of Java. This works fine with Java String objects matching SQL VARCHAR data type passing NULL value to the routine. However, at least in our tests we failed to get Integer objects in the Java method parameter list to match the SQL INTEGER data type used in the procedure CALL. We have sorted this for int parameters by adding extra null indicator parameter in the procedure call and in the routine parameter list, and passing info of the NULL value locally to corresponding Integer object to be used in parametrizing the SQL statement.

Method insert1() is a variant of insertRow for experimenting with unhandled exceptions. These will be passed to the invoking client, which may proper exception handling on application level.

Method insertRow() demonstrates routine body with local exception handling and passing indicators and error message to the invoker. The method demonstrates also passing a dynamic result set to the invoker, which in this case is the **db2** client utility, and which will present the result set to the user.

licensed on Creative Commons license for non-commercial use

Installing the Java class into database

Trial and error experiments, and the excellent article by Law and Shere (IBM developerWorks 2005) gave hint to suspect compatibility of Java compiler and the Java engine used by DB2. It appeared that DB2 Express-C software included only Java Engine, and the OpenJava compiler used in the DBTechNet DebianDB laboratory environment was perhaps not compatible with that. Since Data Studio 2.2 had both Java compiler and engine we configured DB2 server to use the Java engine of Data Studio as follows (skipping some output messages):

```
$ /opt/ibm/DS2.2/jdk/bin/javac -version
javac 1.6.0-internal

$ db2 update dbm cfg using JDK_PATH /opt/ibm/DS2.2/jdk
DB20000I  The UPDATE DATABASE MANAGER CONFIGURATION command completed
successfully.
```

To activate the configured JDK_PATH value we stop and restart the server as follows:

```
$ db2stop
$ db2start


$ cd $HOME/StoredRoutines/ExternalRoutines
```

The Java class is compiled and packed into a jar file jstrlib.jar as follows:

```
$ /opt/ibm/DS2.2/jdk/bin/javac jstrlib.java
$ # Removing the possible previous version of the jar file
$ rm jstrlib.jar
$ # Packing the compiled class into jstrlib.jar:
$ jar cvf jstrlib.jar jstrlib.class
```

Next the jar file containing the class file is loaded into the database as follows:

```
$ db2 +c -t;
 ...
connect to testdb;
call sqlj.install_jar
('file:/home/db2inst1/StoredRoutines/ExternalRoutines/jstrlib.jar', 'jstrlib', 0);
call sqlj.refresh_classes();
commit;
```

Creating the SQL signatures for the functions procedures and testing the use

Hello World

Creating the SQL/Java function for “hello”:

```
CREATE OR REPLACE FUNCTION j_hello ()
RETURNS VARCHAR(12)
LANGUAGE JAVA
PARAMETER STYLE JAVA
NO SQL
FENCED THREADSAFE
EXTERNAL NAME 'jstrlib!hello(java.lang.String)';
```

then testing it:

```
SELECT j_hello() "my message" FROM SYSIBM.sysdummy1;

my message
```

licensed on Creative Commons license for non-commercial use

```
------------
Hello World!
```

1 record(s) selected.

Note: Java routines are processed by the JVM engine “fenced” from the SQL engine and Java language by default is thread safe language, so the FENCED and THREADSAFE clauses could be left out. We have kept them for “documentation purposes” ☺.

SLEEP

Creating the SQL/Java function for the sleep() method :

```
CREATE OR REPLACE PROCEDURE sleep ( IN secs INT )
LANGUAGE JAVA
PARAMETER STYLE JAVA
NO SQL
FENCED THREADSAFE
EXTERNAL NAME 'jstrlib!sleep(int)';
```

then testing it:

```
db2 => CALL sleep(10);

  Return Status = 0
db2 =>
```

and - Yes, we were waiting for 10 seconds before we got the “`Return Status = 0`” message.

insertRow

Next we will experiment with the insertRow variants, first with the insert1 for which we create the following SQL wrapper signature:

```
CREATE OR REPLACE PROCEDURE Insert1 (IN id INT, IN s VARCHAR(20),
                                     OUT rcount INT)
 LANGUAGE JAVA
 PARAMETER STYLE JAVA
 MODIFIES SQL DATA
 FENCED THREADSAFE
 EXTERNAL NAME 'jstrlib!insert1(int, java.lang.String, int[])';
```

and apply the following test run:

```
db2 => CALL Insert1 (2, 'test', ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 1

  Return Status = 0
db2 => CALL Insert1 (2, 'duplicate!', ?);
SQL4302N  Procedure or user-defined function "DB2INST1.INSERT1", specific name
"SQL151104154656300" aborted with an exception "DB2 SQL Error: SQLCODE=-803,
```

```
SQLST".  SQLSTATE=23505
db2 => CALL Insert1 (3, NULL, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 1

  Return Status = 0
```

Following test tries to enter NULL value for the first parameter id, but this is of type int which does not allow null value:

```
db2 => CALL Insert1 (NULL, NULL, ?);
SQL0470N  The user defined routine "DB2INST1.INSERT1" (specific name
"SQL151104154656300") has a null value for argument "1" that could not be
passed.  SQLSTATE=39004
db2 =>
db2 => select * from t;

ID          N           S
----------- ----------- --------------------
          1           - first
          2           - test
          3           - -

  3 record(s) selected.

db2 => rollback;
DB20000I  The SQL command completed successfully.
db2 => select * from t;

ID          N           S
----------- ----------- --------------------
          1           - first

  1 record(s) selected.

db2 =>
```

It appears that if the routine will not handle the exceptions, the exception is passed to the invoking client with as good diagnostics as DB2 server can provide, the values of SQLcode and SQLSTATE as part of the error message. In case the routine contains only one SQL statement, the reason can be concluded with this information, but in case the routine contains more SQL statements, then more diagnostic information is needed from the routine. By our extended insertRow routine we will experiment with different diagnostic information that can be provided using output parameters.

Next we will create the SQL/Java wrapper" j_insertRow" for the insertRow() method. Note that for text contents Java uses Unicode character set, so for accessing the output parameter values of String type we need to use NCHAR or NVARCHAR data types of DB2 by the invoking SQL client.

```
CREATE OR REPLACE PROCEDURE j_insertRow (
       IN id INT,
       IN n INT, IN n_ind INT,
       IN s VARCHAR(20),
       OUT rcount INT,
       OUT sqlcode INT,
       OUT sqlstate NCHAR(5),
```

```
       OUT msg NVARCHAR(100))
 LANGUAGE JAVA
 PARAMETER STYLE JAVA
 MODIFIES SQL DATA
 DYNAMIC RESULT SETS 1
 FENCED THREADSAFE
 EXTERNAL NAME 'jstrlib!insertRow(int, int, int, java.sql.String, int[], int[],
java.sql.String, java.sql.String)';
```

testing it as follows:

```
db2 => delete from t where id > 1;
DB20000I  The SQL command completed successfully.
db2 => CALL j_insertRow (5, 5, 0, 'test5', ?, ?, ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 1

  Parameter Name  : SQLCODE
  Parameter Value : 0

  Parameter Name  : SQLSTATE
  Parameter Value : 00000

  Parameter Name  : MSG
  Parameter Value :


  Result set 1
  --------------

  ID          N           S
  ----------- ----------- --------------------
            1           - first
            5           5 test5

  2 record(s) selected.

  Return Status = 0
db2 => CALL j_insertRow (6, 0, -1, 'test6', ?, ?, ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 1

  Parameter Name  : SQLCODE
  Parameter Value : 0

  Parameter Name  : SQLSTATE
  Parameter Value : 00000

  Parameter Name  : MSG
  Parameter Value :


  Result set 1
  --------------

  ID          N           S
```

licensed on Creative Commons license for non-commercial use

```
  ----------- ----------- --------------------
            1           - first
            5           5 test5
            6           - test6

  3 record(s) selected.

  Return Status = 0
db2 => CALL j_insertRow (7, 7, -1, NULL, ?, ?, ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 1

  Parameter Name  : SQLCODE
  Parameter Value : 0

  Parameter Name  : SQLSTATE
  Parameter Value : 00000

  Parameter Name  : MSG
  Parameter Value :


  Result set 1
  --------------

  ID          N           S
  ----------- ----------- --------------------
            1           - first
            5           5 test5
            6           - test6
            7           - -

  4 record(s) selected.

  Return Status = 0
db2 => CALL j_insertRow (5, 5, 0, 'Duplicate', ?, ?, ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RCOUNT
  Parameter Value : 0

  Parameter Name  : SQLCODE
  Parameter Value : -803

  Parameter Name  : SQLSTATE
  Parameter Value : 23505

  Parameter Name  : MSG
  Parameter Value : DB2 SQL Error: SQLCODE=-803, SQLSTATE=23505, SQLERRMC=1;DB2INST1.T,
DRIVER=3.59.81

  Return Status = 0
db2 => select * from t;

ID          N           S
----------- ----------- --------------------
          1           - first
          6           - test6
          7           - -
```

licensed on Creative Commons license for non-commercial use

```
          5           5 test5

  4 record(s) selected.

db2 => rollback;
DB20000I  The SQL command completed successfully.
db2 => select * from t;

ID          N           S
----------- ----------- --------------------
          1           - first

  1 record(s) selected.

db2 =>
```

This proves that the insertRow method wrapped with the j_InsertRow procedure signature accesses transactionally the content of the database.

More diagnostic information could be easily added to the output parameters, for example the used SQL statement.

5.4 Java Routines in the Oracle JVM

External routines written in various programming languages, for example C/C++, are supported also in Oracle wrapped by PL/SQL procedure layer, but a really interesting implementation is the Oracle JVM, integrated Java Virtual Machine environment as part of the Oracle database server. This allows compiling and running of Java bytecode classes written in JVM languages[7], especially Java, locally inside the server. These are external in the sense that processing occurs "fenced" from the SQL engine in the address base of the JVM.

This allows additional functionalities and integration possibilities over the PL/SQL capabilities, although Oracle environment already contains rich collection of PL/SQL packages allowing, for example, accessing files, messaging, etc.

[7] See the List of JVM Languages at https://en.wikipedia.org/wiki/List_of_JVM_languages

licensed on Creative Commons license for non-commercial use

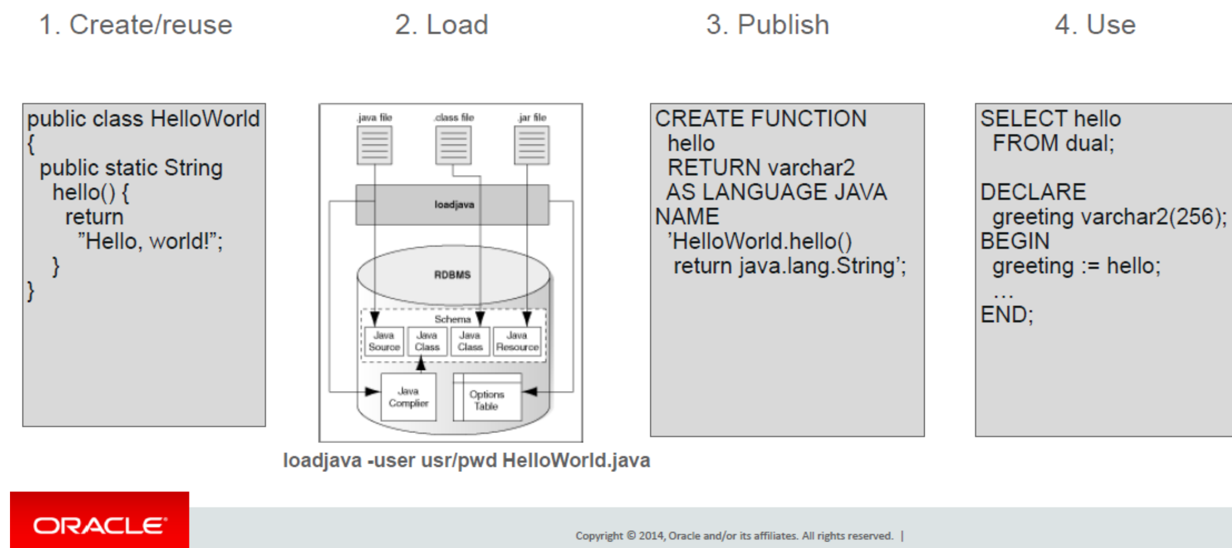


Figure 5.4.1. Overview on use of Oracle JVM
(Source: presentation of Oracle Finland / Timo Leppänen at DBTech VET seminar on 2014-10-22)

Figure 5.4.1 presents the basic Hello World experiment using the Oracle JVM environment on any platform. Java programs can be compiled outside the database and the class file can loaded into database as such or packaged in a jar file using the `loadjava` utility of Oracle. The program can also be loaded and compiled automatically using the `loadjava` utility as presented in step 2 of the Figure 5.4.1. Benefit of this is that the Java version will be the version supported by Oracle JVM.

Methods of local Java classes can be integrated in procedures, functions, and triggers. Following function experiment is based on the Figure 5.4.1 and using Oracle 12.1.0 on Windows workstation:

```
sqlplus sys as sysdba /

CREATE USER scott IDENTIFIED BY password;
GRANT DEBUG CONNECT SESSION, DEBUG ANY PROCEDURE TO scott;
EXIT;
```

Hello World

We can compile and load the HelloWorld.java source presented in Figure 5.4.1 as follows:

```
loadjava -user scott/password -resolve C:\Oracle\HelloWorld.java
```

and create the SQL wrapper for it as follows:

```
sqlplus scott/password

SQL*Plus: Release 12.1.0.2.0 Production on Thu Sep 24 08:13:54 2015
Copyright (c) 1982, 2014, Oracle.  All rights reserved.
Last Successful login time: Thu Sep 24 2015 08:12:29 +03:00
Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production
With the Partitioning, OLAP, Advanced Analytics and Real Application Testing options

SQL> CREATE FUNCTION hello RETURN varchar2
  2  AS LANGUAGE JAVA
  3  NAME 'HelloWorld.hello() return java.lang.String';
```

```
  4  /

Function created.

SQL> SELECT hello FROM dual;

HELLO
--------------------------------------------------------------------------------
Hello World!

SQL> EXIT;
```

For data access the external Java routines of Oracle use server-side internal JDBC driver, which supports only default connection to the local Oracle database. Auto-commit mode is not possible and data access occurs in the current connection context and transaction context of the invoking client. Java code is not allowed to close the physical connection.

Listing 5.4.1 presents jstrlib.java, Java program in which we have collected the methods for our Oracle 12.1 experiments on Windows.

Of the insertRow variants we present below only the variant with local exception handling. We have left out the insert1 method, which skipped the local exception handling. The error message passed to the invoking client on inserting a duplicate row was following:

> Error report:
> SQL Error: ORA-29532: Java call terminated by uncaught Java exception: java.sql.SQLIntegrityConstraintViolationException: ORA-00001: unique constraint (SCOTT.SYS_C0010043) violated 29532. 00000 - "Java call terminated by uncaught Java exception: %s"
> *Cause: A Java exception or error was signaled and could not be
> resolved by the Java code.
> *Action: Modify Java code, if this behavior is not intended.

We have also omitted the test of dynamic result sets, since the Oracle client tools don't support easy presentation of the generated results sets (like the db2 client), and receiving the generated result sets would have required much more client-side programming ☺.

Listing 5.4.1 `jstrlib.java`

```
/* DBTechNet / Martti Laiho
Examples of Oracle external Java stored routines
*/
import java.sql.*;
import java.util.*;
import oracle.jdbc.*;

public class jstrlib {

  public static String hello ()
  {
    return "Hello World!";
  }

  public static void sleep (int secs)
  {
    try {
      Thread.sleep(secs * 1000);
    }
    catch (Exception e) {}
  }
```

```
  public static void insertRow (int id,
                                int n,
                                int n_ind,
                                String s,
                                int[] rcount,
                                int[] sqlcode,
                                String[] sqlstate,
                                String[] errmsg
                               ) throws SQLException
  {
    Connection conn = null;
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    String sql = null;
    Integer N = null;
    sqlcode[0]= 0;
    sqlstate[0] = "00000";
    errmsg[0] = "no errors";
    try {
      OracleDriver ora = new OracleDriver();
      conn = ora.defaultConnection();
      if (n_ind != -1) {
         N = n;
      }
       sql = "INSERT INTO T (id, n, s) VALUES (?, ?, ?)";
       pstmt = conn.prepareStatement( sql );
       pstmt.setInt( 1, id );
       pstmt.setInt( 2, N );
       pstmt.setString( 3, s );
      rcount[0] = pstmt.executeUpdate();
      pstmt.close();
    }
    catch (SQLException e) {
      sqlcode[0]  = e.getErrorCode();
      sqlstate[0] = e.getSQLState();
      errmsg[0]   = e.getMessage();
    }
  }
}
```

The Java code (stored in G:\StoredRoutines\Oracle directory) is compiled and loaded into the database as follows

```
loadjava -user scott/password -resolve G:\StoredRoutines\Oracle\jstrlib.java
```

We create our test table T into the database as follows:

```
sqlplus scott/password

CREATE TABLE T (
id    INT  NOT NULL PRIMARY KEY,
n     INT,
s     VARCHAR2(20)
);
```

The syntax for creating the SQL procedure/function wrappers for Java methods is slightly different from the ISO SQL as can be seen from the BNF syntax, presented in the "Oracle Database Java Developer's Guide":

```
CREATE [OR REPLACE]
{ PROCEDURE procedure_name [(param[, param]...)]
| FUNCTION function_name [(param[, param]...)] RETURN sql_type}
```

licensed on Creative Commons license for non-commercial use

```
[AUTHID {DEFINER | CURRENT_USER}]
[PARALLEL_ENABLE]
[DETERMINISTIC]
{IS | AS} LANGUAGE JAVA
NAME 'method_fullname (java_type_fullname[, java_type_fullname]...)
[return java_type_fullname]';
```

For example, the word "EXTERNAL" is not used.

In the following we present our wrappers and tests for our external routines:

Hello World

We just change the method reference from our example above as follows:

```
CREATE FUNCTION j_hello RETURN varchar2
AS LANGUAGE JAVA
NAME 'jstrlib.hello() return java.lang.String';
```

and the test works as presented above.

SLEEP

```
CREATE OR REPLACE PROCEDURE sleep ( secs IN NUMBER )
IS LANGUAGE JAVA
NAME 'jstrlib.sleep(int)';
```

Tested using SQLDeveloper client tool as follows:

```
CALL sleep(10);

sleep 10) succeeded.
[Task completed in 10.015 seconds]
```

This shows how easy it is to create wrappers to Java functions, but for the sleep method Oracle already has ready method available in the PL/SQL package DBMS_LOCK.

InsertRow

Next we will apply our data access procedure InsertRow to Oracle JVM.

```
CREATE OR REPLACE PROCEDURE insertRow (id IN NUMBER,
                                       n IN NUMBER,
                                       n_ind IN NUMBER,
                                       s IN VARCHAR2,
                                       rcount OUT NUMBER,
                                       errcode OUT NUMBER,
                                       sqlstate OUT NVARCHAR2,
                                       errmsg OUT NVARCHAR2)
 IS LANGUAGE JAVA
 NAME 'jstrlib.insertRow (int, int, int, java.lang.String, int[], int[],
java.lang.String[], java.lang.String[] )';
/
COMMIT;

-- test script:
set serveroutput on;
DECLARE
```

licensed on Creative Commons license for non-commercial use

```
  rcount  NUMBER := 0;
  errcode NUMBER := 0;
  sqlstate VARCHAR2(5)    := '00000';
  errmsg   VARCHAR2(200) := '';
BEGIN
-- Comment out all following insertRow calls, except one at a time:
insertRow (2, 1, 0, 'first try', rcount, errcode, sqlstate, errmsg) ;
--  insertRow (2, 2, 0, 'duplicate',    rcount, errcode, sqlstate, errmsg) ;
--  insertRow (3, 3,-1, 'null for n',   rcount, errcode, sqlstate, errmsg) ;
--  insertRow (4, 4, 0, NULL, rcount, errcode, sqlstate, errmsg) ;
--  insertRow (NULL, 2, 0, NULL, rcount, errcode, sqlstate, errmsg) ;
  DBMS_OUTPUT.put_line ('rcount= ' || TO_CHAR(rcount));
  DBMS_OUTPUT.put_line ('errcode=' || TO_CHAR(errcode));
  DBMS_OUTPUT.put_line ('SQLSTATE=' || sqlstate);
  DBMS_OUTPUT.put_line ('errMsg=' || errmsg);
END;
```

In the following test runs we show the current insertRow call and the results of the script run:

```
  insertRow (2, 1, 0, 'first try', rcount, errcode, sqlstate, errmsg) ;

anonymous block completed
rcount= 1
errcode=0
SQLSTATE=00000
errMsg=no errors

  insertRow (2, 2, 0, 'duplicate',    rcount, errcode, sqlstate, errmsg) ;

anonymous block completed
rcount= 0
errcode=1
SQLSTATE=23000
errMsg=ORA-00001: unique constraint (SCOTT.SYS_C0010043) violated

  insertRow (3, 3,-1, 'null for n',   rcount, errcode, sqlstate, errmsg) ;

anonymous block completed
rcount= 1
errcode=0
SQLSTATE=00000
errMsg=no errors
  insertRow (4, 4, 0, NULL, rcount, errcode, sqlstate, errmsg) ;

anonymous block completed
rcount= 1
errcode=0
SQLSTATE=00000
errMsg=no errors

  insertRow (NULL, 2, 0, NULL, rcount, errcode, sqlstate, errmsg) ;

anonymous block completed
rcount= 1
errcode=0
SQLSTATE=00000
errMsg=no errors

SQL> SELECT * FROM T;

        ID          N S
---------- ---------- --------------------
         3            null for n
         4          4
         0          2
         2          1 first try
```

licensed on Creative Commons license for non-commercial use

```
SQL> ROLLBACK;

Rollback complete.

SQL> SELECT * FROM T;

        ID          N S
---------- ---------- --------------------

SQL>
```

This proves that the SQL-invoked external Java routine insertRow was executed in the connection context and transaction context of our SQL session.

The tests show that the server-side JDBC drive the getErrorCode method of SQLException does not use SQLCode values, but for all error cases returns value 1 and 0 for successful execution.

Other findings are that NULL value entered as input parameter for int variable id will generate value 0 to the primary key id in database, and NULL value passed to String typed variable s generates empty string value for column s. This is OK for Oracle users, who are accustomed to Oracle's interpretation of NULL value for variable character string to be assigned as empty string, even if the correct interpretation according to ISO SQL standard and other products is that empty string is a value and NULL is not any value.

5.5 C# Routines in the SQL Server CLR

see: https://msdn.microsoft.com/en-us/library/ms131043(v=sql.110).aspx
https://msdn.microsoft.com/en-us/library/orm-9780596101404-02-12.aspx

Competing with the Oracle JVM, Microsoft has integrated the Common Language Runtime (CLR) engine in SQL Server to run .NET assemblies compiled from programs on any .NET language. Luckily this CLR integration is available also in the free SQL Server Express edition, so the following experiment can be verified on any computer running SQL Server on Windows platform.

SQL Server CLR integration is not automatically available in databases, but it needs to be enabled separately per database by the following configuring commands:

```
EXEC sp_configure 'clr enabled',1
GO
RECONFIGURE
GO
```

Listing 5.5.1 below presents our test program written in C# language. This time we experiment only with the insertRow routine. Like in using external routines of other products, for the database connection we can use the current connection context of the invoking client, and we expect to use also current transaction context.

Listing 5.5.1. C# code in file MyUdfLib.cs

```
/* DBTechNet / Martti Laiho 2015-09-21
   C# source for the library file of user-defined procedures/functions
   MyUdfLib.dll
*/
using System;
```

licensed on Creative Commons license for non-commercial use

```
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;
using System.Text;

public class MyUdfs
{

  [Microsoft.SqlServer.Server.SqlProcedure]
  public static int InsertRow(int id, int n, int n_ind, string s,
                              out int rcount,
                              out string erMsg)
  {
    StringBuilder errorMessages = new StringBuilder("");
    SqlCommand cmd = null;
    using (SqlConnection conn
           = new SqlConnection("context connection=true")) {
      int rc = -1;
      rcount = 0;
      try
      {
          conn.Open();
          if (n_ind == 0) {
              cmd = new SqlCommand(
                  "INSERT INTO T (id, n, s) VALUES (@ID, @N, @S)", conn);
              cmd.Parameters.Clear();
              cmd.Parameters.Add("@ID", SqlDbType.Int).Value = id;
              cmd.Parameters.Add("@N", SqlDbType.Int).Value = n;
              cmd.Parameters.Add("@S", SqlDbType.Char, 10).Value = s;
          } else {
              cmd = new SqlCommand(
                  "INSERT INTO T (id, s) VALUES (@ID, @S)", conn);
              cmd.Parameters.Clear();
              cmd.Parameters.Add("@ID", SqlDbType.Int).Value = id;
              cmd.Parameters.Add("@S", SqlDbType.Char, 10).Value = s;
          }
          rcount = (int) cmd.ExecuteNonQuery();
          rc  = 0; // OK
      }
      catch (SqlException ex) {
          for (int i = 0; i < ex.Errors.Count; i++) {
              errorMessages.Append("Index #" + i + "\n" +
                  "Message: " + ex.Errors[i].Message + "\n" +
                  "Error Number: " + ex.Errors[i].Number + "\n" +
                  "LineNumber: " + ex.Errors[i].LineNumber + "\n" +
                  "Source: " + ex.Errors[i].Source + "\n" +
                  "Procedure: " + ex.Errors[i].Procedure + "\n");
              if (rc == -1) rc = ex.Errors[i].Number;
         }
      }
      catch (Exception e) {
         errorMessages.Append("Exception on InsertRow: " + e.ToString());
      }
      erMsg = errorMessages.ToString();
      return rc;
    }
  }
}
```

Note: The SqlException investigation code in Listing 5-3 above has partly been copied from the MSDN C# example on the SqlException class at https://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlexception(v=vs.90).aspx

Unfortunately the Microsoft ADO.NET designers have omitted support of the SQLSTATE and SQLcode indicators of ISO SQL standard, and provide only the Transact-SQL proprietary error as the indicator on

licensed on Creative Commons license for non-commercial use

SQL exceptions/errors, even if SQLSTATE is supported in SQL Server's Transact-SQL language. Differently from the SQLcode indicator, the error values of error code are positive integer values.

In the following experiment our C# source program has been stored in the folder C:\work and it will be first compiled into .NET bytecode assembly format in the file MyUdfLib.dll using the C# compiler csc as follows:

```
CD \work
csc.exe /t:library /out:MyUdfLib.dll MyUdfLib.cs
```

Now we can proceed to Transact-SQL session to load the assembly to the database and create the wrapper procedure for the InsertRow method. The extra DROP commands are needed in case we need to repeat the installing routine:

```
DROP PROCEDURE dbo.InsertRow;
GO
DROP ASSEMBLY MyUdfLib;
GO
CREATE ASSEMBLY MyUdfLib FROM 'C:\work\MyUdfLib.dll';
GO
```

For data type compatibilities between string of C# and Transact-SQL of SQL Server the data type of column s definition is changed to NVARCHAR as follows:

```
CREATE TABLE T (id INT PRIMARY KEY, n INT, s NVARCHAR(10));
GO
```

Considering possible SQL NULL passed in parameters, we have noticed that for the string typed columns (column s in our test) is "SQL NULL compatible", just like Java Strings. Just like in Java built-in scalar type int of C# is not SQL NULL compatible, but in C# routine signature we cannot use Integer objects (like in Java), so to pass NULL value for column n we need to use a NULL indicator n_ind. This leeds us to use the following kind of wrapper as the SQL signature for our insertROw routine:

```
CREATE PROCEDURE InsertRow(@id INT, @n INT, @n_ind INT, @s NVARCHAR(10),
                 @count INT OUTPUT, @erMsg NVARCHAR(500) OUTPUT)
AS EXTERNAL NAME MyUdfLib.MyUdfs.InsertRow;
GO
```

To experiment with transactional data access we start by command

```
BEGIN TRANSACTION
```

To investigate the output parameters we call the procedure inside a compound command reporting the output values of return code rc, row count rcount, error message errmsg, and by SELECT statement current content of table T. To shorten the report, all variants of our test are listed in the script, but all InsertRow calls except the one to be currently tested are commented out.

```
BEGIN
DECLARE @rc INT, @rcount INT, @errmsg VARCHAR(500)
-- Comment out all following insertRow calls, except one at a time:
EXEC @rc = dbo.InsertRow 2, 1, 0, 'first try', @rcount OUTPUT, @errmsg OUTPUT
--EXEC @rc = dbo.InsertRow 2, 2, 0, 'duplicate', @rcount OUTPUT, @errmsg OUTPUT
--EXEC @rc = dbo.InsertRow 3, 1, 0, NULL, @rcount OUTPUT, @errmsg OUTPUT
--EXEC @rc = dbo.InsertRow 4, 0, -1, 'NULL for n', @rcount OUTPUT, @errmsg OUTPUT
--EXEC @rc = dbo.InsertRow NULL, 1, 'NULL for id', @rcount OUTPUT, @errmsg OUTPUT
SELECT @rc ReturnCode, @rcount RowCount
SELECT @errmsg ErrorMessages
SELECT * FROM T
```

licensed on Creative Commons license for non-commercial use

```
END;
GO
```

In the following we have the experiment results of every InsertRow starting always with the currently used InsertRow parameters.

```
..
EXEC @rc = dbo.InsertRow 2, 1, 0, 'first try', @rcount OUTPUT, @errmsg OUTPUT
..
ReturnCode  RowCount
----------- -----------
0           1

(1 row(s) affected)

ErrorMessages
--------------------------------------------------------------------------------

(1 row(s) affected)

id          n           s
----------- ----------- ----------
2           1           first try

(1 row(s) affected)
```

Note: We have omitted use of the "SET NOCOUNT OFF" command, since due to some SQL Server bug our RCOUNT parameter would otherwise be -1 instead of value 1.

```
..
EXEC @rc = dbo.InsertRow 2, 2, 0, 'duplicate', @rcount OUTPUT, @errmsg OUTPUT
..
ReturnCode  RowCount
----------- -----------
2627        0

(1 row(s) affected)

ErrorMessages
--------------------------------------------------------------------------------Index
#0
Message: Violation of PRIMARY KEY constraint 'PK__T__3213E83F4E783394'. Cannot insert
duplicate key in object 'dbo.T'. The duplicate key value is (2).
Error Number: 2627
LineNumber: 0
Source: .Net SqlClient Data Provider
Procedure:
Index #1
Messa

(1 row(s) affected)

id          n           s
----------- ----------- ----------
2           1           first try

(1 row(s) affected)

...
EXEC @rc = dbo.InsertRow 3, 1, 0, NULL, @rcount OUTPUT, @errmsg OUTPUT
..
ReturnCode  RowCount
----------- -----------
0           1

(1 row(s) affected)
```

licensed on Creative Commons license for non-commercial use

```
ErrorMessages
--------------------------------------------------------------------------------


(1 row(s) affected)

id          n           s
----------- ----------- ----------
2           1           first try
3           1           NULL

(2 row(s) affected)

..
EXEC @rc = dbo.InsertRow 4, 0, -1, 'NULL for n', @rcount OUTPUT, @errmsg OUTPUT
..
ReturnCode  RowCount
----------- -----------
0           1

(1 row(s) affected)

ErrorMessages
--------------------------------------------------------------------------------

(1 row(s) affected)

id          n           s
----------- ----------- ----------
2           1           first try
3           1           NULL
4           NULL        NULL for n

(3 row(s) affected)


EXEC @rc = dbo.InsertRow NULL, 1, 0, 'NULL for id', @rcount OUTPUT, @errmsg OUTPUT
..
Msg 6569, Level 16, State 1, Procedure InsertRow, Line 0
'InsertRow' failed because parameter 1 is not allowed to be null.
```

Now let's test the transactionality:

```
ROLLBACK
Command(s) completed successfully.

SELECT * FROM T;
id          n           s
----------- ----------- ----------

(0 row(s) affected)
```

This proves the correct transactional context of the routine.

Note: Opposite to DB2 and Oracle, SQL Server can operate in auto-commit mode also on server-side, so if we repeat our first InsertRow CALL in auto-commit mode of client session and without COMMIT command disconnect the session, then start a new session, we can see the recently inserted row in the database.

licensed on Creative Commons license for non-commercial use

Summary

The SQL-invoked external routines have been available at least in DB2, Oracle, and SQL Server already for decades, and the concept is noticed also in ISO SQL standard starting at SQL/PSM in 1996. The documentation in product manuals varies, and this is clearly not a mainstream technology in application development. The technologies are challenging and mainly meant for integration-oriented developers, not for dummies. If applied without proper discipline, lots of damage can be caused both in the consistency of the database and also outside the database.

However, this is a rich area for various studies, for example on implementations, programming languages, data type compatibilities, reliability, security, performance, scalability of solutions, etc.

licensed on Creative Commons license for non-commercial use

Part 6 Server-Side Transaction Programming

When a stored routine is invoked from a client-side transaction it will participate in that transaction (except the autonomic transactions). Writing transaction demarcations, the explicit BEGIN TRANSACTION, COMMIT or ROLLBACK statements, in stored procedures is a controversial issue. For example, we have learned that PostgreSQL and Pyrrho don't allow these in stored routines at all. Also the server-side transaction programming does not fit in the protocol of distributed transactions in Java EE architecture, since decision on commit or rollback is not done by the local DBMS but by the external transaction coordinator of the application server. However, even if a transaction is started by the invoking application code, the transaction may get rolled back automatically by the DBMS server (in which case the invoker needs to get some indication of the case). Also, transaction programming embedded in stored procedures is an in-house policy adopted by some companies. So, let's have a closer look on the possibility of server-side transaction programming.

6.0 Server-Side transactional mode: *'To be, or not to be, -that is the question'*

We will start experimenting with server-side transactional behavior when invoked from autocommit mode on client-side. For our test we create a table having single column, and a stored procedure inserting a row followed by ROLLBACK statement. After control returns to the client-side we test contents of the table.

We start experimenting with DB2. As default the db2 client program uses autocommit mode, unless configured otherwise on the command line.

```
db2 -td@
connect to testdb@
drop table test@
CREATE TABLE test (s  VARCHAR(43))@
drop procedure impltr@
CREATE PROCEDURE impltr (dummy INT)
BEGIN
  INSERT INTO test VALUES ('To be, or not to be, - that is the question');
  ROLLBACK;
END@
CALL impltr(0)@
SELECT * FROM test@
```

On invoking the procedure and testing the table contents we get following result:

```
db2 => CALL impltr(0)@

  Return Status = 0
db2 => SELECT * FROM test@

S
-------------------------------------------

  0 record(s) selected.
```

This proves that on server-side DB2 does not support autocommit mode, and an implicit transaction started for the procedure's INSERT statement.

licensed on Creative Commons license for non-commercial use

We then apply the same test to Oracle. As default the sqlplus client program uses transactional mode, but by SET command can be configured to use autocommit mode:

```
drop table test;
CREATE TABLE test (s  VARCHAR(43));
drop procedure impltr;
CREATE PROCEDURE impltr (dummy INT)
IS BEGIN
  INSERT INTO test VALUES ('To be, or not to be? That is the question');
  ROLLBACK;
END;
/
SET AUTOCOMMIT ON;
CALL impltr(0);
SELECT * FROM test;
```

Calling the procedure and testing the table contents after that we get the following results:

```
SQL> CALL impltr(0);
Call completed.

SQL> SELECT * FROM test;

no rows selected
```

This proves that also Oracle does not support autocommit mode on server-side, but starts an implicit transaction for the first SQL statement.

MySQL and its variant MariaDB clients by default use autocommit mode. For the following experiment we use MariaDB:

```
drop table test;
CREATE TABLE test (s  VARCHAR(43));
drop procedure impltr;
delimiter #
CREATE PROCEDURE impltr (dummy INT)
BEGIN
  INSERT INTO test VALUES ('To be, or not to be, - that is the question');
  ROLLBACK;
END#
delimiter ;
CALL impltr(0);
SELECT * FROM test;
```

And the test results differ from the previous ones:

```
MariaDB [testdb]> CALL impltr(0);
Query OK, 0 rows affected (0.04 sec)

MariaDB [testdb]> SELECT * FROM test;
+---------------------------------------------+
| s                                           |
+---------------------------------------------+
| To be, or not to be, - that is the question |
+---------------------------------------------+
1 row in set (0.00 sec)
```

licensed on Creative Commons license for non-commercial use

This time the processing at the database server continued in autocommit mode, and the ROLLBACK statement has no effect.

Also SQL Server on client sessions and at server-side use autocommit mode by default, but if the client session uses autocommit mode, then the ROLLBACK statement in stored procedure raises exception unless transaction was started explicitly.

```
drop table test;
CREATE TABLE test (s  VARCHAR(43));
drop procedure impltr;
GO
CREATE PROCEDURE impltr
AS BEGIN
  INSERT INTO test VALUES ('To be, or not to be, - that is the question');
  ROLLBACK;
END;
GO
EXEC impltr;
SELECT * FROM test;

(1 row(s) affected)
Msg 3903, Level 16, State 1, Procedure impltr, Line 4
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM test
```

Results | Messages

	s
1	To be, or not to be, - that is the question

6.1 Retry Pattern and Protocol

For our experiments we will extend the Bank Transfer example of Appendix 4, moving both the transaction and the Retry Pattern for invoking the transaction into stored procedures.

The general solution for restarting the cancelled transaction is to **restart the transaction by the application**, and since concurrency conflict may continue to occur the restart should be retried again and again until the transaction manages to avoid the concurrency conflict. However, we cannot allow the retries to continue forever, which is called the **livelock** problem. To avoid the livelock problem we need to limit the number of retries so that the retry loop does not take too long, considering a tolerable limit of **response time** to the end user in case of OLTP applications, whereas for batch programs the number of retries can be bigger. As a general data access programming model this looping with a maximum number of retries is called **Retry Pattern**.

Note: *The Retry Pattern does not apply to the optimistic locking cases, in the update transactions, which are dependent on previous read-transaction(s), and which need to see the rows to be updated in consistent state with the previous reads. If the status of some row has changed, then there is no sense to retry the failed updates.*

A Java/JDBC example of applying the Retry Pattern on client- side to control a transaction is presented in Appendix 4. Client-side control is necessary when the DBMS product does not support server-side transactions, and also for restarting connections in case of **communication failures** of server-side transactions (- we will skip this topic).

In this chapter we will focus on the server-side transactions: the stored procedure **interface** and the product dependent implementations of the transaction logic inside the stored procedures.

licensed on Creative Commons license for non-commercial use

The **interface** consists of the following categories of parameters:

- Input parameters for use by the application logic in the transaction,
- Optional application results from the transaction, either as output parameters of the generated result sets,
- Parameters for configuring the the transaction behavior (timeouts, etc),
- Parameters for controlling concurrency tests, and
- Various statistics observed from the server-side processing.

As examples we use variants of the Bank Transfer transaction for which Listing 6.1.2 presents Java/JDBC test bench program calling the wrapper procedure BankTransferWithRetry. The invoked procedure implements the Retry Pattern based on configuration parameters and calls the actual BankTransfer procedure which finally implements the bank transfer transaction tuned according to the services of the used DBMS product.

Listing 6.1.1 presents the signature of the retry wrapper i.e the parameters of the retry procedure which also acts as the parameter intermediator of the procedure of the transaction. Now the signature is written using the syntax of DB2 SQL, whereas the accurate syntax varies depending on the DBMS product, as seen later.

Listing 6.1.1 Parameters of the procedure BankTransferWithRetry

```
BankTransferWithRetry (
-- Input parameters for the transaction logic:
    IN fromAcct     INT,  -- from account
    IN toAcct       INT,  -- to account
    IN amount       INT,  -- to be transferred
 -- Technical parameters for transaction control:
    IN lockTimeout  INT,  -- lock wait timeout of UPDATEs
 -- Technical parameters for testing:
    IN sleepSecs    INT,  -- sleeping seconds
 -- Statistics from the server-side processing;
    OUT rc          INT,  -- return code
    OUT msg         VARCHAR(500), -- error message
    OUT retry       INT,  -- number of used retries
    OUT elapsedSecs VARCHAR(50) -- elapsed time on server-side
)
```

The first three parameters: “fromAcct”, “toAcct” and “amount” have been introduced in many examples in preceding chapters.

Parameter “lockTimeout” configures the lock wait timeout for the UPDATE statements to prevent too long waiting. The use of this depends on the DBMS product.

Parameter “sleepSecs” configures a sleeping time in seconds between updates of the “from account” and “to account” to allow a reasonable time slot for some competing transaction to test for concurrency conflict cases of the transaction. When applied to production the value should be set to zero, as we don’t want there extra delays.

The return code “rc” from the BankTransfer procedure is used to control possible retries of the BankTransfer procedure, and passed to the invoker from final call of the procedure. The values of “rc” as explained in Table 6.1, define our protocol for controlling the retries.

Value of rc	Explanation
-1	Non recoverable error
0	Successful execution
1	(should be retried, internal for the retry pattern)
2	(Lock wait timeout, retry? internal for the retry pattern)

licensed on Creative Commons license for non-commercial use

3	Livelock, user should decide on retry

Table 6.1 Return code values used in our retry pattern

The output parameter “msg” returns the latest error message from the server, or “OK” in case of successful execution.

The output parameter “retry” returns the number of executed retries of the transaction.

Finally the ouput parameter “elapsedSecs” reports as “bonus” the total execution time on the server-side, i.e without the time spent for invoking on the client-side and the network traffic. The format depends on the defaults used by the DBMS product. (*Design of a generic format could be a practical exercise*!)

Note: The maximum number of the retries in the retry wrapper is set to 10, but in future versions of the interface this might be included as an input parameter.

Now that we have defined the interface, we will first look at listing 6.1.2 presenting the Java/JDBC test bench of the transaction, the program BankTransferProcs. We will then look at the actual BankTransfor procedure and its retry wrapper BankTransferWithRetry implemented in SPL of DB2, PL/SQL of Oracle, and SQL of MySQL.

Listing 6.1.2 BankTransferProcs.java

```
/*  DBTechNet Concurrency Lab 2014-06-22 Martti Laiho
Updates:
3.2 2016-02-16 ML changed into test bench for the BankTransfer[WithRetry] procedure
4.0 2016-02-28 added lockTimeout and measuring the server-side elapsed seconds

************************************************************/
import java.io.Console;
import java.util.Arrays;
import java.io.IOException;
import java.io.*;
import java.util.*;
import java.sql.*;

public class BankTransferProcs {
   public static void main (String args[]) throws Exception   {
      System.out.println("\nBankTransferProcs version 4.0");
      if (args.length != 4) {
         System.out.println("Usage: " +
         "BankTransferProc $driver $URL $user $password \n");
         System.out.println("Length of args was " + args.length);
         System.exit(-1);
      }
      java.sql.Connection conn = null;
      boolean sqlServer = false;
      int counter = 0;
      int retries = 0;
      int rc = 0;
      String contin = "n";
      String msg = "";
      String elapsedSecs = "";
      String URL = args[1];
      String user = args[2];
      String password = args[3];
      int fromAcct = 101;  //Integer.parseInt(args[4]);
      int toAcct = 202;    //Integer.parseInt(args[5]);
```

```
int amount = 100;    //Integer.parseInt(args[6]);
int sleepSecs = 2;   //Integer.parseInt(args[7]);
int lockTimeout = 1;
      // register the JDBC driver and open connection
try {
    Class.forName(args[0]);
    conn = java.sql.DriverManager.getConnection(URL,user,password);
    conn.setAutoCommit(false);
    System.out.println("Connected\n");
}
catch (SQLException ex) {
    System.out.println("URL: " + URL);
    System.out.println("** Connection failure: " + ex.getMessage() +
                 "\n SQLSTATE: " + ex.getSQLState() +
                 " SQLcode: " + ex.getErrorCode());
    System.exit(-1);
}
do {
   String prompt;
   System.out.println(" ");
   prompt = "Enter fromAcct (cr=" + fromAcct + "): ";
   fromAcct = getInteger (prompt, fromAcct);
   prompt = "Enter toAcct (cr=" + toAcct + "): ";
   toAcct = getInteger (prompt, toAcct);
   prompt = "Enter amount (cr=" + amount + "): ";
   amount = getInteger (prompt, amount);
   prompt = "Enter sleep seconds (cr=" + sleepSecs + "): ";
   sleepSecs = getInteger (prompt, sleepSecs);
   System.out.format("Calling BankTransferWithRetry (%d,%d,%d,%d, ..)\n",
                        fromAcct,toAcct,amount,sleepSecs);
   CallableStatement cstmt =
     conn.prepareCall("{CALL BankTransferWithRetry(?,?,?,?,?, ?,?,?,?)}");
   cstmt.setInt(1, fromAcct);
   cstmt.setInt(2, toAcct);
   cstmt.setInt(3, amount);
   cstmt.setInt(4, sleepSecs);
   cstmt.setInt(5, lockTimeout);
   cstmt.setInt(6, 0);
   cstmt.setString(7, msg);
   cstmt.setInt(8, 0);
   cstmt.setString(9, msg);
   cstmt.registerOutParameter(6, java.sql.Types.INTEGER);
   cstmt.registerOutParameter(7, java.sql.Types.VARCHAR);
   cstmt.registerOutParameter(8, java.sql.Types.INTEGER);
   cstmt.registerOutParameter(9, java.sql.Types.VARCHAR);
   cstmt.executeUpdate();
   rc = cstmt.getInt(6);
   msg = cstmt.getString(7);
   retries = cstmt.getInt(8);
   elapsedSecs = cstmt.getString(9);
   System.out.format("\nrc = %d, retry# = %d\nmsg = %s\nserver-side elapsed secs = %s\n",
                 rc, retries, msg, elapsedSecs );
   switch (rc) {
   case -1:
      System.out.println("Transfer failed!\n");
      conn.rollback();
      break;
```

licensed on Creative Commons license for non-commercial use

```
        case 0:
          System.out.println("Transfer successfull\n");
          break;
        case 1:
          System.out.println("Deadlock! Retry needed\n");
          conn.rollback(); // needed for Oracle
          break;
        case 2:
          System.out.println("Lock Wait Timeout!\n");
          break;
        case 3:
          System.out.println("Livelock\n");
          break;
        default:
          break;
        }
        cstmt.close();
        //
        prompt = "Do you want to continue (y | cr=n): ";
        contin = getString (prompt, contin);
      } while (contin.equals("y"));
      conn.close();
      System.out.println("\n End of Program. ");
  }
 // subroutines:
  public static int getInteger (String prompt, int oldVal) throws Exception
  {   Integer newVal = null;
      String line = "";
      Console cons = System.console();
      try {
        line =  cons.readLine(prompt);
        if (line != "") newVal = Integer.parseInt(line);
        //System.out.format("oldval='%d', you entered '%d'\n", oldVal, newVal);
      }
      catch (Exception ex) {
        // System.out.format ("* Error: %s\n", ex.getMessage());
        newVal = oldVal;
      }
      finally {
        return newVal;
      }
  }
  public static String getString (String prompt, String oldVal) throws Exception
  {   String newVal = "n";
      String line = "";
      Console cons = System.console();
      try {
        line =  cons.readLine(prompt);
        if (!(line.equals(""))) newVal = line.substring(0,1);
        //System.out.format("oldval='%d', you entered '%d'\n", oldVal, newVal);
      }
      catch (Exception ex) {
        // System.out.format ("* Error: %s\n", ex.getMessage());
        newVal = oldVal;
      }
      finally {
        return newVal;
```

licensed on Creative Commons license for non-commercial use

```
      }
   }
}
```

Listing 6.1.3 presents some test runs of the program calling the BankTransferWithRetry procedure.

Listing 6.1.3 Sample test runs in DebianDB 6 lab:

```
# first session
cd $HOME/AppDev/JDBC
export CLASSPATH=.:/opt/jdbc-drivers/db2jcc4.jar
export driver="com.ibm.db2.jcc.DB2Driver"
export URL="jdbc:db2://localhost:50001/TESTDB"
export user="student"
export password="password"
java BankTransferProcs $driver $URL $user $password

BankTransferProcs version 4.0
Connected
Enter fromAcct (cr=101):
Enter toAcct (cr=202):
Enter amount (cr=100):
Enter sleep seconds (cr=2): 0
Calling BankTransferWithRetry (101,202,100,0, ..)
rc = 0, retry# = 0
msg = OK
server-side elapsed secs = .004399 on server-side
Transfer successfull
Do you want to continue (y | cr=n): y
Enter fromAcct (cr=101):
Enter toAcct (cr=202):
Enter amount (cr=100):
Enter sleep seconds (cr=0): 2
Calling BankTransferWithRetry (101,202,100,2, ..)
rc = 0, retry# = 1
msg = OK
server-side elapsed secs = 6.016587 on server-side
Transfer successfull
Do you want to continue (y | cr=n): n
End of Program.
student@debianDB:~/AppDev/JDBC$

# competing session (getting done first!)
cd $HOME/AppDev/JDBC
export CLASSPATH=.:/opt/jdbc-drivers/db2jcc4.jar
export driver="com.ibm.db2.jcc.DB2Driver"
export URL="jdbc:db2://localhost:50001/TESTDB"
export user="student"
export password="password"
java BankTransferProcs $driver $URL $user $password

BankTransferProcs version 4.0
Connected
Enter fromAcct (cr=101): 202
Enter toAcct (cr=202): 101
Enter amount (cr=100): 200
Enter sleep seconds (cr=2):
```

licensed on Creative Commons license for non-commercial use

Calling BankTransferWithRetry (202,101,200,2, ..)
rc = 0, retry# = 0
msg = OK
server-side elapsed secs = 2.021564 on server-side
Transfer successfull
Do you want to continue (y | cr=n): n

End of Program.

Exercise: Add measuring the total elapsed time on client-side.

6.2 Bank Transfer Procedures in DB2 SPL

We will first present the BankTransfer procedure which implements the actual transaction. This should be tested in stand-alone mode using the CLP client program “db2”, before testing it with the wrapper procedure BankTransferWithRetry.

Listing 6.2.1. BankTransfer Procedure written in DB2 SPL

```
db2 -td@
CONNECT TO TESTDB@
CREATE OR REPLACE PROCEDURE BankTransfer (
    IN fromAcct  INT,
    IN toAcct    INT,
    IN amount    INT,
    IN sleepSecs INT,
    OUT rc       INT,
    OUT msg      VARCHAR(500)
) LANGUAGE SQL SPECIFIC BankTransfer
P1: BEGIN
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLCODE  INTEGER DEFAULT 0;
  DECLARE ERRMSG   VARCHAR(500);
  DECLARE deadlock CONDITION FOR SQLSTATE '40001';
  DECLARE EXIT HANDLER FOR deadlock
  BEGIN
     GET DIAGNOSTICS EXCEPTION 1 ERRMSG = MESSAGE_TEXT;
     SET msg = ERRMSG;
     IF (LOCATE_IN_STRING (ERRMSG, '"68"', 50) > 0) THEN
         SET rc = 2; -- reason code 68 = timeout
     ELSE
         SET rc = 1; -- reason code 2 = deadlock
     END IF;
     ROLLBACK;
  END;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
     GET DIAGNOSTICS EXCEPTION 1 ERRMSG = MESSAGE_TEXT;
     SET msg = '* SQLSTATE: ' || SQLSTATE || ', ' || ERRMSG;
     SET rc = 1;
     ROLLBACK;
  END;
  SET ISOLATION = CS; -- This should fail if a transaction is active
  ROLLBACK;   -- This is needed to exclude processing by XA JDBC driver
  -- Note: the next DML statement starts the transaction!
  UPDATE Accounts SET balance = balance - amount WHERE acctno = fromAcct;
  IF (SQLCODE = 100) THEN
  BEGIN
      ROLLBACK;
      SET rc = -1;
      SET msg = '* Unknown from account ' || fromAcct;
```

licensed on Creative Commons license for non-commercial use

```
      END;
  ELSE
      BEGIN
       CALL db2inst1.SLEEP(sleepSecs); -- sync point for concurrency tests
       UPDATE Accounts SET balance = balance + amount WHERE acctno = toAcct;
       IF (SQLCODE = 100) THEN
       BEGIN
         SET rc = -1;
         ROLLBACK;
         SET msg = '* Unknown to account ' || toAcct;
         END;
       ELSE
       BEGIN
         COMMIT;
         SET rc = 0;
         SET msg = 'OK';
         END;
       END IF;
      END;
  END IF;
END P1 @
GRANT EXECUTE ON PROCEDURE BankTransfer TO PUBLIC @
COMMIT @
```

We use the identical SPECIFIC name for simplicity and to eliminate the possibility of multiple procedure copies in database since DB2 supports overloading of stored routine names.

The LOCATE_IN_STRING function starts searching for string “68” from as early as column 50 to cope with error messages in various native languages, although in the English version the text could be tested faster just comparing it with SUBSTRING (msg, 103, 4).

Note: The SLEEP procedure (created by db2inst1 login) is used only for concurrency testing between the SQL commands of the transaction so that there is enough time to start manually another session with a conflicting transaction. The procedure is based on the Java method sleep() for which we have created an SQL wrapper as we have documented in the Chapter 5 “External Routines”. For the production environment the line “`CALL db2inst1.SLEEP(sleepSecs);`” could be commented out, as follows:

```
-- CALL db2inst1.SLEEP (sleepSecs);
```

The following test run moving amount 150 from account 101 to account 202 without pause in between the account actions (sleepSecs = 0) presents how the procedure can be easily tested using the db2 client (CLP). The output parameters are marked by question marks (?) resulting CLP to report the parameter names and values in output as follows:

```
student@debianDB:~$ db2 +c -t;
(c) Copyright IBM Corporation 1993,2007
...
db2 => connect to TESTDB;
...
db2 => CALL BankTransfer (101, 202, 150, 0, ?, ?);

  Value of output parameters
  --------------------------
  Parameter Name  : RC
  Parameter Value : 0

  Parameter Name  : MSG
  Parameter Value : OK

  Return Status = 0
db2 =>
```

licensed on Creative Commons license for non-commercial use

When the procedure BankTransfer has been tested to produce proper return codes on exceptions, concurrency conflicts, and successful transaction executions, we can proceed to build and test its retry wrapper implemented as DB2 SPL stored procedure “BankTransferWithRetry”, presented in Listing 6.2.2.

Listing 6.2.2. Retry wrapper procedure BankTransferWithRetry for the BankTransfer procedure

```
db2 -td@
CONNECT TO TESTDB@
CREATE OR REPLACE PROCEDURE BankTransferWithRetry (
    IN fromAcct    INT,
    IN toAcct      INT,
    IN amount      INT,
    IN lockTimeout INT,
    IN sleepSecs   INT,
    OUT rc         INT,
    OUT msg        VARCHAR(500),
    OUT retry      INT,
    OUT elapsedSecs VARCHAR(50)
) LANGUAGE SQL SPECIFIC BankTransferWithRetry
P0: BEGIN
  DECLARE retryCount INTEGER DEFAULT -1;
  DECLARE startTimestamp  TIMESTAMP ;
  SET startTimestamp = CURRENT TIMESTAMP;
  SET CURRENT LOCK TIMEOUT = lockTimeout;
  SET msg = '';
  SET rc = 1;
  WHILE (( rc = 1 OR rc = 2) AND retryCount < 5 ) DO
     SET retryCount = retryCount + 1;
     CALL BankTransfer (fromAcct, toAcct, amount, sleepSecs, rc, msg);
     IF ( rc = 1 OR rc = 2 ) THEN
         CALL db2inst1.RANDSLEEP (1);
     END IF;
  END WHILE;
  IF (( rc = 1 OR rc = 2) AND retryCount = 5) THEN
     SET rc = 3;
  END IF;
  SET retry = retryCount;
  SET elapsedSecs = CAST ((CURRENT TIMESTAMP - startTimestamp) AS VARCHAR(12)) ||
                         ' on server-side';
END P0 @
GRANT EXECUTE ON PROCEDURE BankTransferWithRetry TO PUBLIC @
COMMIT @
```

Here the procedure call RANDSLEEP (1) is used for having a random pause of max 1 second, which is considered necessary in concurrency conflict cases to allow competing transactions to proceed to end. The code is cloned from the sleep procedure in Chapter 5.3 as follows:

```
 public static void randSleep (int maxSecs) {  // max n seconds
    try {
        long pause = (long) (Math.random () * 1000 * maxSecs);
        Thread.sleep(pause);
    }
    catch (Exception e) {}
  }
```

The timeout parameter and strategy on handling the timeout cases need to be planned and configured according to the needs of the application. In case the lock wait timeout is set shorter than the deadlock detection period of DB2, the tests will lead to the return code of timeout when the competing transaction continues to be blocking.

In the following we list some test runs using the DB2 client CLP

```
db2 +c -t;
connect to testdb;
```

```
db2 => -- no sleep, no timeout
db2 => CALL BankTransferWithRetry (101, 202, 100, 0, -1, ?, ?, ?, ?) ;

  Value of output parameters
  --------------------------
  Parameter Name  : RC
  Parameter Value : 0

  Parameter Name  : MSG
  Parameter Value : OK

  Parameter Name  : RETRY
  Parameter Value : 0

  Parameter Name  : ELAPSEDSECS
  Parameter Value : .007577 on server-side

  Return Status = 0
db2 =>
```

Figure 6.2.1 No competing sessions

```
db2 => -- first session
db2 => CALL BankTransferWithRetry (101, 202, 100, 5, -1, ?, ?, ?, ?) ;

  Value of output parameters
  --------------------------
  Parameter Name  : RC
  Parameter Value : 0

  Parameter Name  : MSG
  Parameter Value : OK

  Parameter Name  : RETRY
  Parameter Value : 0

  Parameter Name  : ELAPSEDSECS
  Parameter Value : 11.622718 on server-side

  Return Status = 0
db2 =>
```

```
db2 => -- second session
db2 => CALL BankTransferWithRetry (202, 101, 100, 5, -1, ?, ?, ?, ?) ;

  Value of output parameters
  --------------------------
  Parameter Name  : RC
  Parameter Value : 0

  Parameter Name  : MSG
  Parameter Value : OK

  Parameter Name  : RETRY
  Parameter Value : 1

  Parameter Name  : ELAPSEDSECS
  Parameter Value : 15.866468 on server-side

  Return Status = 0
db2 =>
```

Figure 1.b Competing transactions with 5 sec sync. sleeping in the middle of transaction
no lock wait timeout

licensed on Creative Commons license for non-commercial use

```
db2 => -- first session
db2 => CALL BankTransferWithRetry (101, 202, 100, 3, 1, ?, ?, ?, ?) ;

  Value of output parameters
  --------------------------
  Parameter Name  : RC
  Parameter Value : 0

  Parameter Name  : MSG
  Parameter Value : OK

  Parameter Name  : RETRY
  Parameter Value : 0

  Parameter Name  : ELAPSEDSECS
  Parameter Value : 3.004797 on server-side

  Return Status = 0
db2 =>
```

```
db2 => -- second session
db2 => CALL BankTransferWithRetry (202, 101, 100, 3, 1, ?, ?, ?, ?) ;

  Value of output parameters
  --------------------------
  Parameter Name  : RC
  Parameter Value : 0

  Parameter Name  : MSG
  Parameter Value : OK

  Parameter Name  : RETRY
  Parameter Value : 1

  Parameter Name  : ELAPSEDSECS
  Parameter Value : 7.900330 on server-side

  Return Status = 0
db2 =>
```

Figure 6.2.2 Competing transactions with 3 sec sync. sleeping in the middle of transaction, 1 s lock wait timeout

```
db2 => connect to testdb;

   Database Connection Information

 Database server        = DB2/LINUX 9.7.2
 SQL authorization ID   = STUDENT
 Local database alias   = TESTDB

db2 => UPDATE Accounts SET balance = 2000 WHERE acctId = 101;
DB20000I  The SQL command completed successfully.
db2 =>
```

First session executes a blocking UPDATE

```
db2 => CALL BankTransferWithRetry (202, 101, 100, 0, 1, ?, ?, ?, ?) ;

  Value of output parameters
  --------------------------
  Parameter Name  : RC
  Parameter Value : 3

  Parameter Name  : MSG
  Parameter Value : SQL0911N  The current transaction has been rolled back becau
se of a deadlock or timeout.  Reason code "68".  SQLSTATE=40001

  Parameter Name  : RETRY
  Parameter Value : 5

  Parameter Name  : ELAPSEDSECS
  Parameter Value : 13.467262 on server-side

  Return Status = 0
db2 =>
```

Using lock wait timeout of 1 second a competing session after 5 retries reports of timeout problem

Figure 6.2.3 Lock wait timeout cutting the waiting time in case of a blocking transaction

licensed on Creative Commons license for non-commercial use

6.3 Bank Transfer Procedures in PL/SQL

Corresponding to the procedures in DB2 SPL, we present first the BankTransfer procedure implemented in PL/SQL, and then its retry wrapper BankTransferWithRetry.

Listing 6.3.1 BankTransfer procedure written in PL/SQL

```
sqlplus scott/tiger
CREATE OR REPLACE PROCEDURE BankTransfer (
    fromAcct  IN INT,
    toAcct    IN INT,
    amount    IN INT,
    sleepSecs IN INT,
    rc       OUT INT,
    msg      OUT VARCHAR2
) IS
  Cannot_Serialize EXCEPTION;
  Deadlock_Detected EXCEPTION;
  PRAGMA EXCEPTION_INIT(Deadlock_Detected, -60);
  PRAGMA exception_init(Cannot_Serialize, -8177);
  acct INT;
  ernum INT;
  mesg VARCHAR2(400);
BEGIN
  acct := fromAcct;  -- for error message
  SELECT acctno INTO acct FROM Accounts
  WHERE acctno = acct FOR UPDATE WAIT 10;  -- to check current existence
  UPDATE Accounts SET balance = balance - amount
  WHERE acctno = fromAcct;
  DBMS_LOCK.SLEEP(1); -- 1 sec pause for concurrency testing
  acct := toAcct;
  SELECT acctno INTO acct FROM Accounts
  WHERE acctno = acct FOR UPDATE WAIT 10;  -- to check current existence
  UPDATE Accounts SET balance = balance + amount
  WHERE acctno = toAcct;
  COMMIT;
  msg := 'committed';
  rc := 0;
EXCEPTION
WHEN NO_DATA_FOUND THEN
  ROLLBACK;
  msg := 'missing account ' || TO_CHAR(acct);
  rc := -1;
WHEN Deadlock_Detected OR Cannot_Serialize THEN
  ROLLBACK;
  rc := 1;
WHEN OTHERS THEN
  ROLLBACK;
  ernum := SQLCODE;
  mesg := SUBSTR(SQLERRM, 1, 200);
  msg := mesg || ' SQLcode=' || TO_CHAR(ernum);
  rc := -1;
END;
/
show errors
GRANT EXECUTE ON BankTransfer TO PUBLIC;
```

Testing the procedures using the Oracle SQL*Plus client is a bit more complicated than using DB2 CLP. A parameterized procedure need to be run in an anonymous PL/SQL block, building variables for the parameters (at least to output parameters), and after invoking the procedure we need to report the

licensed on Creative Commons license for non-commercial use

values of the output parametrs using put() or put_line() methods of the PL/SQL package DBMS_OUTPUT which puts the values to an internal pipe on the server session. For reporting the values out of the pipe we need to enter “SET SERVEROUTPUT ON” command before running the script, as follows:

```
set serveroutput on;
DECLARE
 fromAcct  INT := 101;
 toAcct    INT := 202;
 amount    INT := 100;
 sleeping  INT := 0;
 rc        INT := -1;
 msg       VARCHAR2(500):= '';
BEGIN
 BankTransfer (fromAcct, toAcct, amount, sleeping, rc, msg);
 DBMS_OUTPUT.put_line ('rc=' || TO_CHAR(rc));
 DBMS_OUTPUT.put_line ('msg: ' || msg);
END;
/
```

When the procedure BankTransfer has been tested to produce proper return codes on exceptions, concurrency conflicts, and successful transaction executions, we can proceed to build and test its retry wrapper implemented as PL/SQL stored procedure “BankTransferWithRetry”, presented in Listing 6.3.2.

Listing 6.3.2 The retry wrapper procedure BankTransferWithRetry using PL/SQL

```
CREATE OR REPLACE PROCEDURE create or replace
PROCEDURE BankTransferWithRetry (
    fromAcct    IN  INT,
    toAcct      IN  INT,
    amount      IN  INT,
    lockTimeout IN  INT,  -- not used in PL/SQL
    sleepSecs   IN  INT,
    rc          OUT INT,
    msg         OUT VARCHAR2,
    retry       OUT INT,
    elapsedSecs OUT VARCHAR2
) IS
  retryCount INTEGER := -1;
  startTimestamp  TIMESTAMP(6);
  randno  NUMBER;
BEGIN
  startTimestamp := SYSTIMESTAMP; -- current
  SET TRANSACTION ISOLATION LEVEL READ COMMITTED; -- before transaction(s)O
  msg := '';
  rc := 1;
  WHILE (( rc = 1 OR rc = 2) AND retryCount < 5 ) LOOP
     retryCount := retryCount + 1;
     BankTransfer (fromAcct, toAcct, amount, sleepSecs, rc, msg);
     IF ( rc = 1 OR rc = 2 ) THEN
        randno := DBMS_RANDOM.value(); -- pause for random sleep of 0..1 seconds
        DBMS_LOCK.sleep(randno);
     END IF;
  END LOOP;
  IF (( rc = 1 OR rc = 2) AND retryCount = 5) THEN
      rc:= 3;
  END IF;
  retry := retryCount;
  elapsedSecs := CAST ((SYSTIMESTAMP - startTimestamp) AS VARCHAR2) || ' on server-
side';
END;/
show errors
```

```
GRANT EXECUTE ON BankTransferWithRetry TO PUBLIC;
```

Oracle XE does not come with the integrated Java platform server, but with a large library of PL/SQL packages from which we have used the packages DBMS_RANDOM and DBMS_LOCK for the random sleeping time between the transaction retries.

Compared with the DB2 SPL procedures, we have skipped the LOCK WAIT TIMEOUT setting, since it is not supported in Oracle. However, corresponding lock wait is available in the format of SELECT .. FOR UPDATE WAIT <max wait seconds> statement. The WAIT clause with literal timeout is available in Oracle only for SELECT statements, and this is why we have those SELECT statements with 10 second lockwait timeouts in front of the UPDATE statements.

To use a wait parameter instead, we should build the SQL statement dynamically and run it by EXECUTE IMMEDIATE statement, but this is left as extra exercise.

To test the transaction with its retry wrapper we use following kind of PL/SQL scripts:

```
sqlplus scott/tiger
set serveroutput on;
DECLARE
 fromAcct  INT := 101;
 toAcct    INT := 202;
 amount    INT := 100;
 sleeping  INT := 0;
 timeout   INT := 0;  -- N/A
 rc        INT := -1;
 msg       VARCHAR2(500):= '';
 retry     INT := -1;
 elapsed   VARCHAR2(50) := '';
BEGIN
 BankTransferWithRetry (fromAcct, toAcct, amount, sleeping, timeout, rc, msg, retry,
elapsed);
 DBMS_OUTPUT.put_line ('rc=' || TO_CHAR(rc) || ', retry=' || TO_CHAR(rc));
 DBMS_OUTPUT.put_line ('msg: ' || msg);
 DBMS_OUTPUT.put_line ('elapsed: ' || elapsed);
END;
/
```

Following is a test run of the script:

```
SQL> set serveroutput on;
DECLARE
 fromAcct  INT := 101;
 toAcct    INT := 202;
 amount    INT := 100;
 sleeping  INT := 0;
 timeout   INT := 0;  -- N/A
 rc        INT := -1;
 msg       VARCHAR2(500):= '';
 retry     INT := -1;
 elapsed   VARCHAR2(50) := '';
BEGIN
 BankTransferWithRetry (fromAcct, toAcct, amount, sleeping, timeout, rc, msg, retry,
elapsed);
 DBMS_OUTPUT.put_line ('rc=' || TO_CHAR(rc) || ', retry=' || TO_CHAR(rc));
 DBMS_OUTPUT.put_line ('msg: ' || msg);
 DBMS_OUTPUT.put_line ('elapsed: ' || elapsed);
END;
/
```

licensed on Creative Commons license for non-commercial use

```
SQL>   2     3     4     5     6     7     8     9    10    11    12    13    14    15    16    17
rc=0, retry=0
msg: committed
elapsed: +000000000 00:00:01.001968000 on server-side

PL/SQL procedure successfully completed.
```

6.4 Bank Transfer Procedures in MySQL

Corresponding to the procedures in DB2 SPL, we present first the BankTransfer procedure implemented also in MySQL Stored Program Language, and then the retry wrapper procedure BankTransferWithRetry.

Listing 6.4.1 BankTransfer procedure modified for MySQL

```
mysql testdb
DROP PROCEDURE BankTransfer ;
delimiter #
CREATE PROCEDURE BankTransfer (
    IN fromAcct  INT,
    IN toAcct    INT,
    IN amount    INT,
    IN sleepSecs INT,
    OUT rc       INT,
    OUT msg      VARCHAR(500)
) LANGUAGE SQL MODIFIES SQL DATA
P1: BEGIN
  DECLARE SQLSTAT  CHAR(5);
  DECLARE ERRMSG   VARCHAR(200);
  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 SQLEXCEPTION
  BEGIN
     GET DIAGNOSTICS CONDITION  1
                     SQLSTAT = RETURNED_SQLSTATE,
                     ERRMSG  = MESSAGE_TEXT;
     SET msg =  CONCAT(SQLSTAT, CONCAT(': ',ERRMSG)) ;
     IF SQLSTAT = '40001' THEN
         SET rc = 1;
     ELSE
         ROLLBACK;
         SET rc = -1;
     END IF;
  END;
  SET acct = fromAcct;
  -- transactions starts now
  START TRANSACTION;
  SELECT acctno INTO acct FROM accounts WHERE acctno = fromAcct LIMIT 1;
  UPDATE accounts SET balance = balance - amount WHERE acctno = fromAcct;
  DO SLEEP(sleepSecs); -- sync point for concurrency tests
  SET acct = toAcct;
  SELECT acctno INTO acct FROM accounts WHERE acctno = toAcct LIMIT 1;
  UPDATE accounts SET balance = balance + amount WHERE acctno = toAcct;
  COMMIT;
  SET rc = 0;
  SET msg = 'OK';
END P1 #
delimiter ;
```

licensed on Creative Commons license for non-commercial use

Testing MySQL procedures in a mysql client session is made easy by the local, untyped variables which are marked by a preceding at-sign (@).
By default the MySQL sessions use AUTOCOMMIT mode. For concurrency tests this needs to be turned off by command:
SET AUTOCOMMIT = 0;

The following commands execute a successful transfer of 100 euros from account 101 to account 202, and the output parameters are stored to the local variables @rc and @msg. Their values can be read with the following SELECT command:

```
CALL BankTransfer (101, 202, 100, 0, @rc, @msg);
SELECT @rc, @msg;
```

When the procedure BankTransfer has been tested to produce proper return codes on exceptions, concurrency conflicts, and successful transaction executions, we can proceed to build and test its retry wrapper implemented as MySQL stored procedure "BankTransferWithRetry", presented in Listing 6.4.2.

Listing 6.4.2 The retry wrapper procedure BankTransferWithRetry using MySQL procedure language

```
DROP PROCEDURE BankTransferWithRetry ;
delimiter #
CREATE PROCEDURE BankTransferWithRetry (
    IN fromAcct    INT,
    IN toAcct      INT,
    IN amount      INT,
    IN lockTimeout INT,
    IN sleepSecs   INT,
    OUT rc         INT,
    OUT msg        VARCHAR(500),
    OUT retry      INT,
    OUT elapsedSecs VARCHAR(50)
) LANGUAGE SQL
P0: BEGIN
  DECLARE retryCount INT DEFAULT -1;
  DECLARE startTimestamp  TIMESTAMP(6) ;
  DECLARE currentStamp    TIMESTAMP(6);
  DECLARE elapsed VARCHAR(40);
  SET startTimestamp = SYSDATE(6);
  SET LOCK_WAIT_TIMEOUT = lockTimeout;
  SET msg = '';
  SET rc = 1;
  WHILE (( rc = 1 OR rc = 2) AND retryCount < 5 ) DO
     SET retryCount = retryCount + 1;
     CALL BankTransfer (fromAcct, toAcct, amount, sleepSecs, rc, msg);
     IF ( rc = 1 OR rc = 2 ) THEN
         DO SLEEP (RAND());  -- Random Sleep of 0..1 seconds
     END IF;
  END WHILE;
  IF (( rc = 1 OR rc = 2) AND retryCount = 5) THEN
     SET rc = 3;
  END IF;
  SET retry = retryCount;
  SET currentStamp = SYSDATE(6);
  SET elapsed = TIMEDIFF(currentStamp,startTimestamp);
  SET elapsedSecs = CONCAT('elapsed time ',elapsed);
END P0 #
delimiter ;
COMMIT ;
```

licensed on Creative Commons license for non-commercial use

```
mysql testdb;

SET AUTOCOMMIT = 0;
-- Testing of the parameter interface
-- no lock timeout
-- missing fromAcct:
CALL BankTransferWithRetry (100, 202, 100, 0, 0, @rc, @msg, @retries, @elapsed);
SELECT @rc, @msg, @retries, @elapsed;

-- missing toAcct:
CALL BankTransferWithRetry (101, 200, 100, 0,  0, @rc, @msg, @retries, @elapsed);
SELECT @rc, @msg, @retries, @elapsed;

-- check constraint violation:
CALL BankTransferWithRetry (101, 202, 2000, 0,  0, @rc, @msg, @retries, @elapsed);
SELECT @rc, @msg, @retries, @elapsed;
Exit

# Concurrency test using clients in 2 concurrent terminal windows:
# first client
mysql  testdb
CALL BankTransferWithRetry (101, 202, 100, 10, 5, @rc, @msg, @retries, @elapsed);
SELECT @rc, @msg, @retries, @elapsed;

# second  client
mysql  testdb
CALL BankTransferWithRetry (202, 101, 100, 10, 5, @rc, @msg, @retries, @elapsed);
SELECT @rc, @msg, @retries, @elapsed;
SELECT * from accounts;
```

Summary

In this tutorial we have used Java as the host language, but usually it is possible to invoke the same stored procedures from C/C++, C#, Perl, PHP, Python, Ruby, or other programming languages. In fact, we continue this theme in our next tutorial in “Introduction to Transacction Programming” focusing on client-side programming, but invoking these BankTransfer procedures from those listed programming/scripting languages.

Moving transaction programming to server-side stored procedures requires knowledge of the procedural SQL dialect of the DBMS product, the transactional extensions and services of the DBMS product, but it really simplifies the application code – just compare the Listing 6.1.2 with the Java example in Appendix 4, and have a look at the examples in our next tutorial.

When the native SQL with the transaction and concurrency control tuning of the DBMS is used in the stored procedures, the stored procedures provide the best opportunity to write reliable transactions with good performance and security. So, benefits of implementing transactions and their retry wrappers as stored procedures include the following:

Reliability: Isolation of transactions in stored procedures from the application code provides stand-

alone testing opportunity using SQL client utilities, tailored testbench programs or debugging in IDE workbenches (such as IBM Data Studio).

Modern procedural SQL languages (DB2 SPL, Oracle PL/SQL, etc) provide good exception handling facilities, which may not be of the same level in some scripting languages used.

The transaction logic may be easier to implement in stored procedure than splitted in the application code, especially in Web applications. As programming by procedural SQL languages requires good knowledge in SQL and services of the DBMS product, skills of database professionals, but it helps to write better and more reliable code than by ordinary web programmers. Just read the study by Bailis et al[8].

The tested retry wrapper code is isolated from the application code and can be easily copied to new transactions.

Performance: Naturally, for good performance of the application, tables and indexes in the database need to be properly designed, implemented and tuned. Stored procedures improve performance of the applications as the SQL code in stored procedures is pre-optimized, and the network traffic is minimized, especially on retries and in case of a remote database. In DB2 and Oracle the stored procedures are precompiled and with execution plans stored in the database, whereas in MySQL the procedures are compiled when first invoked, but only the latest plans are immediately available in a main memory cache.

Measurement of the elapsed time comes for free using the wrapper procedure. For a proper understanding of the performance, the measurements should be done using test materials of relevant size compared with contents of the final production database.

Minor performance improvement can be achieved if the retry wrapping code is programmed in the same procedure with the transaction, but the separate procedure improves readability of the code.

Security, since the SQL code in stored procedure by default run using the privileges of the creator, and users to whom have been granted execute privilege on the procedure don't need direct privileges to access the data in the database. Also the compiled procedure code is protected against SQL injections.

Disadvantages of Transactions in Stored Procedures:

The required knowledge and skills for coding the transactions by procedural SQL are a disadvantage in terms of shortage of the skilled professionals.

Transactions inside a stored procedure break the scope and the transactional structure of possible distributed transaction of the invoker. This will be discussed in our tutorial "Introduction to Transaction Programming".

[8] Bailis P. et al, "Feral Concurrency Control: An Empirical Investigation of Modern application Integrity", SIGMOD'15, 2015, available at http://www.bailis.org/papers/feral-sigmod2015.pdf

licensed on Creative Commons license for non-commercial use

References

Astrahan M. M. et al, "System R: Relational Approach to Database Management", ACM Transactions on Database Systems, Vol. 1, No. 2. June 1976, Pages 97-137.

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

Connolly T., Begg C., "Database Systems", 5th ed. , Addison-Wesley, 2010

Crowe M., "The Pyrrho Book", ebook at www.pyrrhodb.org, 2015

Curtis A. T., Herman E., "A Tour of External Language Stored Procedures for MySQL", MySQL Conference & Expo 2008

Gulutzan P., Pelzer T. "SQL-99 Complete, Really", R&D Books, 1999 (available online at https://mariadb.com/kb/en/sql-99-complete-really/)

Harrison G., Feurstein S., "MySQL Stored Procedure Programming", O'Reilly, 2006

IBM, "Developing User-defined Routines (SQL and External)", IBM DB2 9.7 for Linux, UNIX, and Windows, 2009

IBM, "External routines" articles of DB2 version 10.1 LUW, IBM Knowledge Center at http://www-01.ibm.com/support/knowledgecenter/SSEPGG_10.1.0/com.ibm.db2.luw.apdv.routines.doc/doc/c0020569.html?

Johnson R. D., Reimer D., "Issues in the development of transactional Web applications", IBM Systems Journal vol 43 no 2, 2004

Kline K. E., Kline D., Hunt B., "SQL in a Nutshell", 3rd ed. , O'Reilly, 2009

Laiho M., Dervos D., Silpiö K., "SQL Transactions – Theory and hands-on labs", 2013-2014 (at http://www.dbtechnet.org/download/SQL-Transactions_handbook_EN.pdf)

Laiho M., Laux F., "On Row Version Verifying (RVV) Data Access Discipline for avoiding Blind Overwriting of Data", "RVV Paper" version 2, 2011 (available online at http://www.dbtechnet.org/papers/RVV_Paper.pdf)

Laiho M., Laux F., "On SQL Concurrency Technologies", 2011 (available online at www.dbtechnet.org/papers/SQL_ConcurrencyTechnologies.pdf)

Law C., Shere S., "Solve common problems with DB2 UDB Java stored procedures", article in IBM developerWorks, 2005

Leppänen T., "Oracle Database Programming", presentation at HBC on Oct 2014

Melton J. (editor), "SQL Routines and Types for the JavaTM Programming Language", (IWD draft) ISO/IEC 9075-13:2011

Melton J., "Understanding SQL's Stored Procedures", Morgan Kaufmann Publishers, Inc. , 1998

Melton J., Eisenberg A., "Understanding SQL and Java Together", Morgan Kaufman, 2000

licensed on Creative Commons license for non-commercial use

Melton J., Simon A. R., ”SQL:1999 Understanding Relational Language Components”, Morgan Kaufmann Publishers, Inc. , 2002

Melton J., “Advanced SQL:1999”, Morgan Kaufmann Publishers, Inc. , 2003

North K., “Database Magic with Ken North”, Prentice Hall PTR, 1999

Oracle, “PL/SQL Language Reference”, Oracle, 2008

Silberschatz A., Korth H. A., Sudarshan S., “Database System Concepts”, 3rd ed. , McGraw-Hill, 2011

Stürner G., “Oracle 7: A User’s and Developer’s Guide Including Release 7.1”, Thompson Computer Press, 1995

Wendelius M. “Custom Aggregates in SQL Server”, article at http://www.codeproject.com/Articles/170061/Custom-Aggregates-in-SQL-Server,

licensed on Creative Commons license for non-commercial use

Appendix 1 Result Set Processing with 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**.

Statement	explanation
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)

licensed on Creative Commons license for non-commercial use

Property	Explanation
<cursorname>	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
<sensitivity>	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
<scrollability>	NO SCROLL defines forward only processing (default), SCROLL allows forward and backward scrolling in the result set
<holdability>	WITH HOLD allows cursor processing to continue after COMMIT of the transaction, WITHOUT HOLD closes an open cursor at COMMIT (usually the default)
<updatability>	UPDATE [OF <column list>] allows positional updates/deletes, READ ONLY does not allow positional updates/deletes (default)

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.

licensed on Creative Commons license for non-commercial use

In ESQL the SQL statements are include 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

```
#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

```
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
```

```
    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[9].

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)

```
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)

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

[9] 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.

licensed on Creative Commons license for non-commercial use

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:

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

We will use this in our modified BankTransfer procedure in appendix 4.

licensed on Creative Commons license for non-commercial use

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.

licensed on Creative Commons license for non-commercial use

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[10], 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"[11] 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 A2.1 transaction A is the root of the nested transaction, parent transaction which includes as subtransactions B and C as its child transactions.

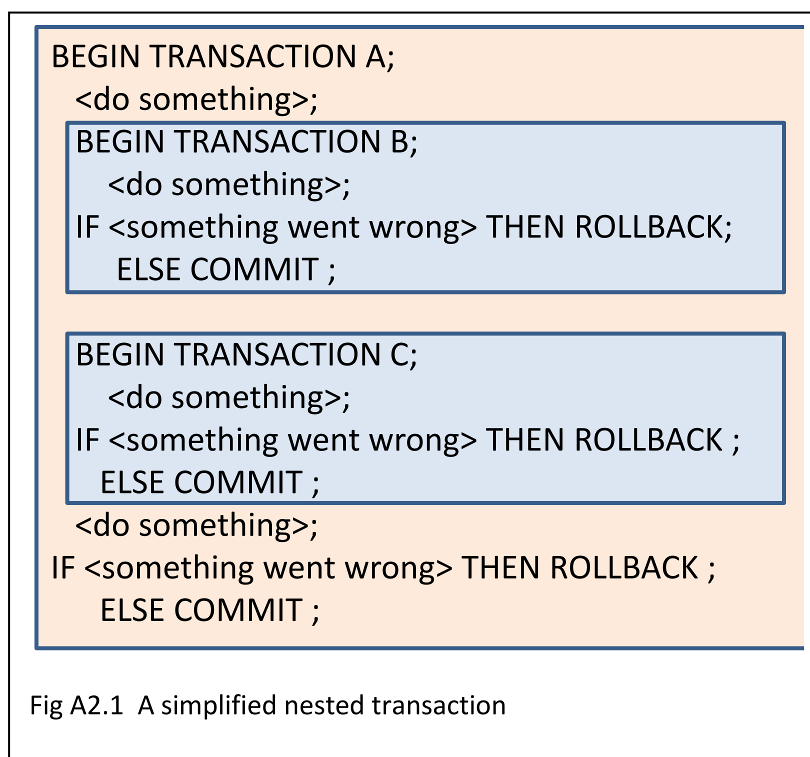


Fig A2.1 A simplified nested transaction

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).

[10] Rothermel and Mohan list following systems in 1989: ARGUS, Camelot, CLOUDS, LOCUS, and Eden.
[11] Compare the hierarchy with elements of well-formed XML document.

licensed on Creative Commons license for non-commercial use

Visibility rule: All objects held by a parent transaction are available to its subtransactions. All changes made by a commited 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
  INSERT INTO T VALUES ('A2')
  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
  INSERT INTO T VALUES ('A3')
  SELECT 'A:', @@TRANCOUNT
COMMIT TRANSACTION A
SELECT * FROM T;
```

licensed on Creative Commons license for non-commercial use

In T-SQL @@TRANCOUNT indicates the current nesting level of transactions.

These are the results of the script for transaction A:

Results:

```
---- -----------
A:    1


---- -----------
B:    2


---- -----------
C:    2


---- -----------
D:    1


---- -----------
A:    0

Msg 3902, Level 16, State 1, Line 20
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.
col
----------
D1
A3
```

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

ISO SQL standard statements: Products:	Supported Savepoint statements of the standard or alternate syntax		
	SAVEPOINT <name>	RELEASE SAVEPOINT <name>	ROLLBACK TO SAVEPOINT <name>
DB2 SQL	SAVEPOINT <name> ON ROLLBACK RETAIN CURSORS	yes	yes
Oracle PL/SQL	yes	N/A	ROLLBACK TO <name>
MySQL/InnoDB	yes	yes	yes
PostgreSQL	yes	yes	yes
SQL Server T-SQL	SAVE TRANSACTION <name>	N/A	ROLLBACK TRANSACTION <name>

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 implemention 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
```

licensed on Creative Commons license for non-commercial use

```
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 ;
```

licensed on Creative Commons license for non-commercial use

```
+----+-----------------+
| id | s               |
+----+-----------------+
|  1 | value 1         |
|  2 | value 2         |
|  6 | the latest row  |
+----+-----------------+
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

Gray J., Reuter A., “Transaction Processing: Concepts and Techniques”, Morgan Kaufmann Publishers, 1993

Moss J. E. B., “Nested Transactions: an approach to reliable distributed computing”, MIT, 1981

Moss J. E. B., “Log-based Recovery for Nested Transactions”, Proceedings of 13th International Conference on VLDB, 1987

Rothermel K., Mohan C., “ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions”, Proceedings of 15th International Conference on VLDB, 1989

licensed on Creative Commons license for non-commercial use

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.

```
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.

```
CREATE OR REPLACE 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;
    CALL myTraceProc ('BankTransfer',1,'finding and slocking fromAcct');
    SELECT acctno INTO acct FROM Accounts WHERE acctno = fromAcct ;
```

licensed on Creative Commons license for non-commercial use

```
    CALL myTraceProc ('BankTransfer',2,'updating fromAcct');
    UPDATE Accounts SET balance = balance - amount WHERE acctno = fromAcct;
    -- CALL SLEEP(15);
    SET acct = toAcct;
    CALL myTraceProc ('BankTransfer',3,'finding and slocking toAcct');
    SELECT acctno INTO acct FROM Accounts WHERE acctno = toAcct ;
    CALL myTraceProc ('BankTransfer',4,'updating toAcct');
    UPDATE Accounts SET balance = balance + amount WHERE acctno = 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 (acctno, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctno, 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.

licensed on Creative Commons license for non-commercial use

Appendix 4 Calling Stored Procedures in Java Program

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() +
                                   "\n SQLSTATE: " + ex.getSQLState() +
                                   " SQLcode: " + ex.getErrorCode());
               System.exit(-1);
         }
         do {
         // Retry wrapper block of TransaferTransaction -------------
             if (counter++ > 0) {
                System.out.println("retry #" + counter);
                if (sqlServer) {
                   conn.close();
                   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("\n End of Program. ");
    }

    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);
```

licensed on Creative Commons license for non-commercial use

```
                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:";
                    while (ex != null) {
                            SQLState = ex.getSQLState();
                            errMsg = errMsg + "SQLState: "+ SQLState;
                            errMsg = errMsg + ", Message:  "+ ex.getMessage();
                            errMsg = errMsg + ", Vendor:   "+ ex.getErrorCode() + "\n";
                            ex = ex.getNextException();
                    }
                    // println for testing purposes
                    System.out.println("** Database error: " + errMsg);
                }
                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.

licensed on Creative Commons license for non-commercial use

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 acctno INTO acct FROM Accounts WHERE acctno = fromAcct ;
    UPDATE Accounts SET balance = balance - amount WHERE acctno = fromAcct;
    SELECT SLEEP(15) INTO @dummy;  -- just for synchronizing
                                   -- concurrent session in deadlock test
    SET acct = toAcct;
    SELECT acctno INTO acct FROM Accounts WHERE acctno = toAcct ;
    UPDATE Accounts SET balance = balance + amount WHERE acctno = 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);
```

licensed on Creative Commons license for non-commercial use

```
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
```

licensed on Creative Commons license for non-commercial use

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 acctno = 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 acctno = toAcct;
    IF SQL%NOTFOUND THEN RAISE NO_DATA_FOUND; END IF;
    msg := 'committed';
    rc := 0;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        msg := 'missing account ' || TO_CHAR(acct);
        rc := -1;
    WHEN OTHERS THEN
        ernum := SQLCODE;
        mesg := SUBSTR(SQLERRM, 1, 200);
        msg := 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:

licensed on Creative Commons license for non-commercial use

```
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
```

```
student@debianDB: ~/Java
File Edit View Terminal Help
student@debianDB:~/Java$ # First window:
student@debianDB:~/Java$ cd $HOME/Java
student@debianDB:~/Java$ export CLASSPATH=.:/opt/jdbc-drivers/ojdbc6.jar
student@debianDB:~/Java$
student@debianDB:~/Java$ export driver=oracle.jdbc.driver.OracleDriver
student@debianDB:~/Java$
student@debianDB:~/Java$ export URL=jdbc:oracle:thin:@localhost:1521:XE
student@debianDB:~/Java$
student@debianDB:~/Java$ export user=user1
student@debianDB:~/Java$ export password=sql
student@debianDB:~/Java$ export fromAcct=101
student@debianDB:~/Java$ export toAcct=202
student@debianDB:~/Java$ java BankTransferProc $driver $URL $user $password $fro
mAcct $toAcct
BankTransferProc version 1.0
** procedure msg: ORA-00060: deadlock detected while waiting for resource SQLcod
e=-60
Waiting for 871 mseconds before retry
retry #2

 End of Program.
student@debianDB:~/Java$
```

```
Oracle SQL Developer
rsioning Tools Help
Start Page   user1
0.29899999 seconds
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
GRANT SELECT, UPDATE ON Accounts TO User1;
SELECT * FROM Accounts;
COMMIT;

Script Output
Task completed in 0.299 seconds
ACCTID                 BALANCE
---------------------- ----------------------
101                    1000
202                    2000
```

```
student@debianDB: ~/Java
File Edit View Terminal Help
student@debianDB:~/Java$ # Second window:
student@debianDB:~/Java$
student@debianDB:~/Java$ cd $HOME/Java
student@debianDB:~/Java$ export CLASSPATH=.:/opt/jdbc-drivers/ojdbc6.jar
student@debianDB:~/Java$
student@debianDB:~/Java$ export driver=oracle.jdbc.driver.OracleDriver
student@debianDB:~/Java$
student@debianDB:~/Java$ export URL=jdbc:oracle:thin:@localhost:1521:XE
student@debianDB:~/Java$
student@debianDB:~/Java$ export user=user1
student@debianDB:~/Java$ export password=sql
student@debianDB:~/Java$ export fromAcct=202
student@debianDB:~/Java$ export toAcct=101
student@debianDB:~/Java$ java BankTransferProc $driver $URL $user $password $fro
mAcct $toAcct
BankTransferProc version 1.0

 End of Program.
```

licensed on Creative Commons license for non-commercial use

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 acctno, balance FROM Accounts;
    SELECT t_time, t_what FROM myTrace;
END !
DELIMITER ;
```

and test it as follows

```
mysql> CALL getResultSets ();
+--------+---------+
| acctno | 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 );
  } 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);
     //
     CallableStatement cstmt = con.prepareCall("{CALL getResultSets}");
     cstmt.execute();
     ResultSet rs1 = cstmt.getResultSet();
        System.out.println ("Accounts:\nacctno | balance\n------------------");
     while (rs1.next()) {
           System.out.println(rs1.getString("acctno") + "    | "
                    + rs1.getString("balance"));
     }
     rs1.close();
     cstmt.getMoreResults();
     System.out.println ("\nmyTrace:\ntime        | text\n------------------");
     ResultSet rs2 = cstmt.getResultSet();
     while (rs2.next()) {
           System.out.println(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

getResultSetTest version 1.0

Accounts:
acctno | 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

licensed on Creative Commons license for non-commercial use

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:

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

Following is our simple program testing the use of the procedure

/*  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 ("acctno | 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);
```

licensed on Creative Commons license for non-commercial use

```
        while (rs.next()) {
          System.out.println(rs.getString("acctno") + "    | "
                         + 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
export DRIVER=oracle.jdbc.driver.OracleDriver
export URL=jdbc:oracle:thin:@localhost:1521:XE
export USER=user1
export PASSWORD=sql
java getCursorTest $DRIVER $URL $USER $PASSWORD
student@debianDB:~/Java$ java getCursorTest $DRIVER $URL $USER $PASSWORD
getCursorTest version 1.0

acctno | balance
------------------
101    | 1000
202    | 1400
student@debianDB:~/Java$ student@debianDB:~/Java$
```

licensed on Creative Commons license for non-commercial use

Index

%FOUND; 144
%ISOPEN; 144
%NOTFOUND; 144, 159
%ROWCOUNT; 144
@@FETCH_STATUS; 142
@@TRANCOUNT; 148
action granularity; 15
Ada; 28
AFTER trigger; 16
alias name; 14
anonymous block; 29, 31
ATOMIC; 7
atomic compound statement; 13
ATOMIC compound statement; 14, 17
AUTONOMOUS clause; 152
autonomous transaction; 152
BEFORE trigger; 16
BEGIN ATOMIC; 13
BIND command; 92
cascading triggers; 18
CHECK constraint; 43
CLI; 2
client-side cache; 145
CLR integration; 112
command terminator; 5
COMMIT; 20
compound triggers; 36
Compound Triggers; 78
concurrency problem; 55
CONDITION; 8
condition handler; 8
connection context; 88
CONTAINS SQL; 5
CONTINUE; 9
cursor model; 145
cursor processing
  explicit SQL cursor; 142
  implicit cursor; 143
DB2 CLP utility; 6
DBMS_LOCK; 36, 133, 159
DBMS_LOCK package; 33
DBMS_OUTPUT; 132
DBMS_OUTPUT package; 144
DBMS_RANDOM; 133
default connection; 88
DELETED table; 41
DELETING; 36
DETERMINISTIC; 4, 13
distributed transactions; 147
DO clause; 8
dynamic result set; 101
DYNAMIC RESULT SETS; 5
embedded SQL; 142
Embedded SQL; 141
ESQL; 2, 141
exception handling; 100
EXCEPTION part; 33
EXEC SQL; 142
EXECUTE privilege; 5, 13, 17
execution plan; 92
EXIT; 9
EXTERNAL clause; 85
fenced; 106
FENCED; 10, 85
FOLLOWS; 79
FOLLOWS clause; 83
FOR control structure; 7
FOR EACH clause; 15
FOR UPDATE clause; 33
GET DIAGNOSTICS; 8
GRANT EXECUTE; 33
handler action; 8
holdability; 143
impedance mismatch; 140
implicit COMMIT; 152
implicit cursor; 159
indicators; 100
Inline SQL PL; 28
INOUT parameter; 3, 6
INSENSITIVE; 7
INSERTED table; 41
INSERTING; 36
INSTEAD OF; 17
isolation level; 55
ITERATE; 7
LEAVE; 7
livelock; 120
LOCATE_IN_STRING; 127
lock wait timeout; 121
LOOP control structure; 7
MATCH; 80
methods; 2
middleware API; 145
MODIFIES SQL DATA; 5
mutating table; 66, 80
nested savepoints; 149
nesting of triggers; 18
NEW row; 17

licensed on Creative Commons license for non-commercial use

NO ACTION; 71
NO CASCADE; 27
NO SQL; 4
NO_DATA_FOUND; 33, 159
NOT FOUND; 8, 9
NULL indicator; 114
NULL value; 89, 100, 112
OLD row; 17
optimistic concurrency control; 48
optimistic locking; 21, 142
OUT and INOUT parameters; 98
partial rollback; 149
PL/SQL; 28
PL/SQL anonymous block; 29
PL/SQL Cursor FOR LOOP; 143
PL/SQL package; 29, 76
PL/SQL packages; 159
PL/SQL record; 144
priorities for application development; 88
privileges; 5
pseudo column; 38
READS SQL DATA; 5
rebooting DB2 instance; 98
recursive trigger; 18
REF CURSOR; 162
Referential Integrity; 55
RELEASE SAVEPOINT; 149
RESTRICT; 71
RESULT; 12
result set cursor; 5
retry; 122
Retry Pattern; 120
return code; 121
RI rules; 71, 80
ROLLBACK; 20
ROLLBACK TO SAVEPOINT; 149
ROW_COUNT; 9
row-level trigger; 15
ROWSCN; 38
ROWVERSION; 41
RVV; 21
RVV trigger; 36, 41, 48, 52
savepoint; 148
SAVEPOINT; 148
scrollability; 145
section; 92
security threat; 17
SEQUENCE; 76
SERVEROUTPUT; 132
side-effect; 63
SIGNAL; 7, 8
SLEEP function; 158
sleep method; 159
sleep() method; 102
SPECIFIC; 21, 127
SPECIFIC name; 94
SQL Server; 147
SQL wrapper; 85
SQL/JRT; 86
SQL/OLB; 86
SQLCODE; 8, 33
SQLEXCEPTION; 8
SQLJ group; 86
SQLJ.INSTALL_JAR; 86
sqlplus; 37
SQLSTATE; 8, 33
SQLWARNING; 8
statement-level trigger; 15
TABLE function; 14
table-valued function; 14
Transact SQL; 2
transaction context; 88
trigger body; 17
TRIGGER privilege; 17
triggering event; 15
T-SQL; 2, 38
UDF; 1, 12
UDT; 2
UNDO; 9
Unicode; 103
UPDATE-UPDATE scenario; 34
UPDATING; 36
user-defined function; 12
version stamping; 21
visibility rule; 147
WAIT clause; 133
WHENEVER; 8
WITH HOLD; 143
WITH RETURN; 5

licensed on Creative Commons license for non-commercial use

Comments of Readers

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

- Ken North at http://www.pinterest.com/knorth2/technology-spotlight/ and Twitter on 2014-07-24. Ken North is technology analyst, consultant, editor, and conference chair. His publications are listed at the website http://www.kncomputing.com/index.htm

licensed on Creative Commons license for non-commercial use