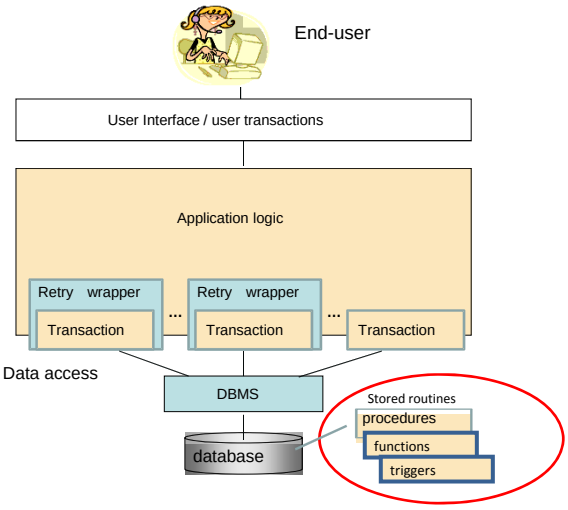
www.DBTechNet.org

SQL Stored Routines

DBTechNet workshop in Reutlingen University 2014-11-04
Martti.Laiho@gmail.com

Stored Routines for Application Logic and Business Rules



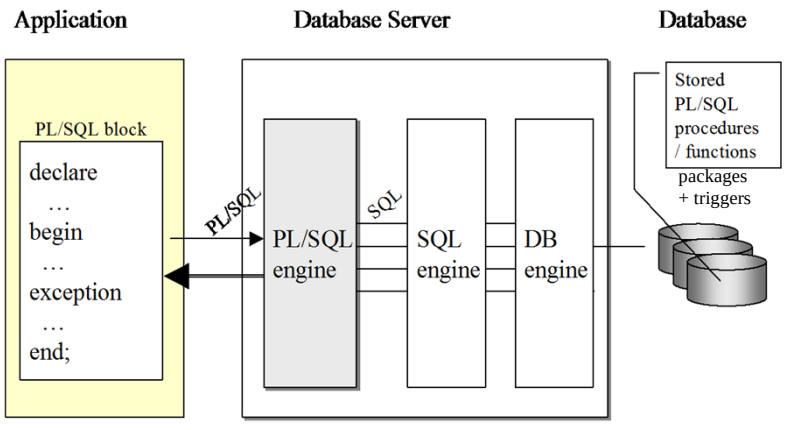
Procedural Languages

Languages:

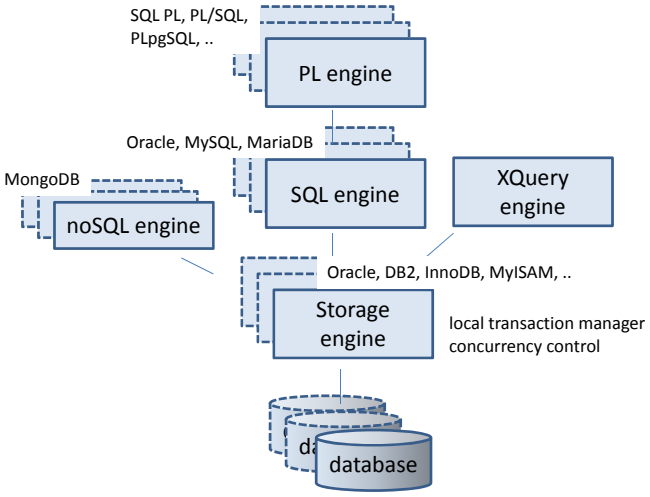
- ISO SQL: SQL/PSM language definition 1996..
- Oracle: PL/SQL
- DB2: SQL PL (SQL/PSM), PL/SQL
- MySQL/MariaDB: SQL/PSM
- PostgreSQL: PL/pgSQL, …
- SQL Server: Transact SQL (”T-SQL”)

Used for: Stored Routines (modules, packages of),
Scripting

Oracle Server’s Stack of Engines and Languages



Database Server Stacks



.. Procedural Languages

- Control structures

	ANSI/ISO SQL	DB2	Oracle	SQL SERVER	MySQL	PostgreSQL	Pyrrho
	SQL/PSM	SQL PL	PL/SQL	Transact SQL	SQL/PSM	PL/pgSQL	
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				
Procedure call							
	CALL	CALL	CALL	EXECUTE	CALL	CALL	CALL
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	

PL/SQL Control Structures

by Timo Leppänen / Oracle Finland

Procedural Statements

- SQL single row functions
- SQL statements
- Control structures:
  - IF – THEN – ELIF – ELSE – END IF;
  - CASE – WHEN – ELSE – END CASE;
  - LOOP – EXIT [WHEN] – END LOOP;
  - WHILE – LOOP – END LOOP;
  - FOR – IN – LOOP – END LOOP;
  - CONTINUE

```
BEGIN
  SELECT balance
    INTO v_bal
    FROM accounts
   WHERE acct_no = '1234567890' FOR UPDATE;
  v_bal := v_bal +
     CASE v_act WHEN 'deposit'    THEN v_amount
                WHEN 'withdrawal' THEN -v_amount
                ELSE 0
     END;
  IF v_bal < 0 THEN
    DBMS_OUTPUT.PUT_LINE('No money!');
  ELSE
    CASE WHEN v_bal < 10000
              THEN v_lvl := 'bronze';
         WHEN v_bal < 100000
              THEN v_lvl := 'silver';
         ELSE v_lvl := 'gold';
    END CASE;
    FOR i IN 1..TRUNC(v_bal/100) LOOP
      DBMS_OUTPUT.PUT_LINE('Century bill');
    END LOOP;
  END IF;
  …
```

ORACLE

Stored Routines

- Procedures : Callable ”subprograms”
- Functions : User Defined Functions (UDF)
  Scalar and table-valued functions
  extending the SQL language
- Methods: Function methods of User Defined Type
  (UDT) objects
- Triggers: Programmable constraints

- Packages: (non-standard) collections of
  procedures/functions

PL/SQL Block Types

by Timo Leppänen / Oracle Finland

Block Types

Procedure

```
PROCEDURE myProc
  IS
  [-- declarations]
  BEGIN
  -- statements
  [EXCEPTION
  -- error handling]
  END;
```

Function

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

Anonymous

```
[DECLARE
  -- declarations]
  BEGIN
  -- statements
  [EXCEPTION
  -- error handling]
  END;
```

- *Used in scripts and compound commands by SQL-clients*

PL/SQL Package Concept Explained

by Timo Leppänen / Oracle Finland

Compare with UDT and methods

Packages

- 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;
```

ISO SQL: Create Procedure

```
CREATE  PROCEDURE [[<catalog>.] <schema>.] <procedure name>
(  [ [<parameter mode>]  <parameter name>  <data type>  [, … ] ]  )

[ < routine characteristic>
   [, …] ]

{ < SQL routine body>
 |  EXTERNAL  NAME
        {<name> | <char string literal> }
   [ PARAMETER STYLE
         <parameter style> ]
   [ EXTERNAL SECURITY
      { DEFINER
       | INVOKER
       | IMPLEMENTATION  DEFINED
      }
   ]
}
```

IN
OUT
INOUT

```
LANGUAGE {SQL | ADA | C | COBOL | FORTRAN
                  | MUMPS | PASCAL | PLI | JAVA}
| PARAMETER STYLE  { SQL | GENERAL | JAVA }
| SPECIFIC  <specific name>
| DETERMINISTIC  |  NOT DETERMINISTIC
| { NO SQL
    |  CONTAINS SQL
    |  {READS | MODIFIES } SQL DATA }
| { RETURNS NULL ON NULL INPUT
    |  CALLED ON NULL INPUT }
| DYNAMIC RESULT SETS <max # of result sets>
```

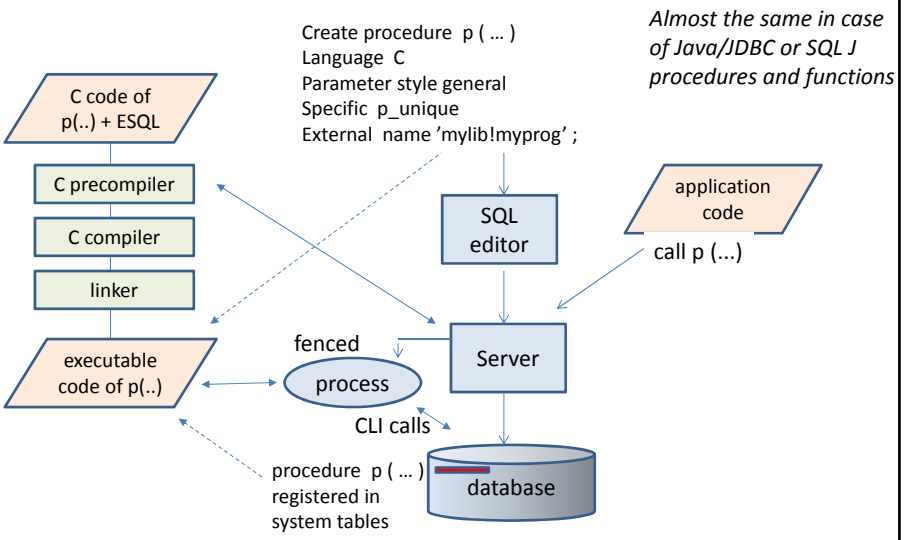
A Deterministic Procedure

Adapted from "DB2 Application Development" of R. Chong et al

Testing by the CLP client of DB2

```
$ db2 -td/
...
db2 => connect to testdb /
db2 => CREATE PROCEDURE P ( IN p1 INT, INOUT p2 INT, OUT p3 INT);
LANGUAGE SQL
DETERMINISTIC
BEGIN SET p2 = p2 + p1;
      SET p3 = p1;
END /

db2 => CALL P (2, 3, ?) /

  Value of output parameters
  --------------------------
  Parameter Name  : P2
  Parameter Value : 5

  Parameter Name  : P3
  Parameter Value : 2

  Return Status = 0
db2 =>
```

[Fenced] External Routines



PL/SQL and Java Classes

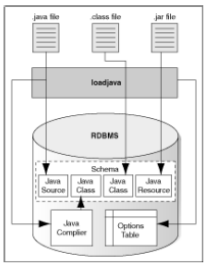
by Timo Leppänen / Oracle Finland

Example: Java Stored Procedures / Functions

1. Create/reuse

```
public class HelloWorld
{
  public static String
    hello() {
      return
        "Hello, world!";
    }
}
```

2. Load



loadjava -user usr/pwd HelloWorld.java

3. Publish

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

4. Use

```
SELECT hello
  FROM dual;

DECLARE
  greeting varchar2(256);
BEGIN
  greeting := hello;
  …
END;
```

ORACLE

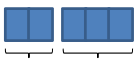
A Table with CHECK Constraint, used in the Labs

```
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL,
CONSTRAINT unloanable_account CHECK (balance >= 0)
);

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;
```

Diagnostics: SQLcode, SQLSTATE

ISO SQL-89 SQLcode: Integer:
100 No data
0 successful execution
< 0 errors

ISO SQL-92 SQLSTATE: String of 5 characters:



	class	subclass	
Successful execution	0 0	0 0 0	
Warning	0 1	n n n	
No data	0 2	0 0 0	
. . .			
Transaction rollback	4 0	0 0 0	
		0 0 1	Serialization failure
		0 0 2	Integrity constraint violation
		0 0 3	Statement completion unknown
		0 0 4	Triggred action exception

etc - lots of standardized and implementation dependent codes

ISO SQL:1999 Get Diagnostics …
List of diagnostic items, including SQLSTATE and number of rows. Only few implementations this far

Exception Handling using Diagnostics

DB2 SQL:

```
<SQL statement>
IF (SQLSTATE <> '00000') THEN
     <error handling>
END IF;
```

Oracle PL/SQL:

```
BEGIN
     <processing>
EXCEPTION
WHEN <exception name> THEN
     <exception handling>;
...
WHEN OTHERS THEN
     err_code := sqlcode;
     err_text := sqlerrm;
     <exception handling>;
END;
```

compare with Java:

```
... throws SQLexception {
...
try {
     ...
     <JDBC statement(s)>
}
catch (SQLException ex) {
     <exception handling>
}
```

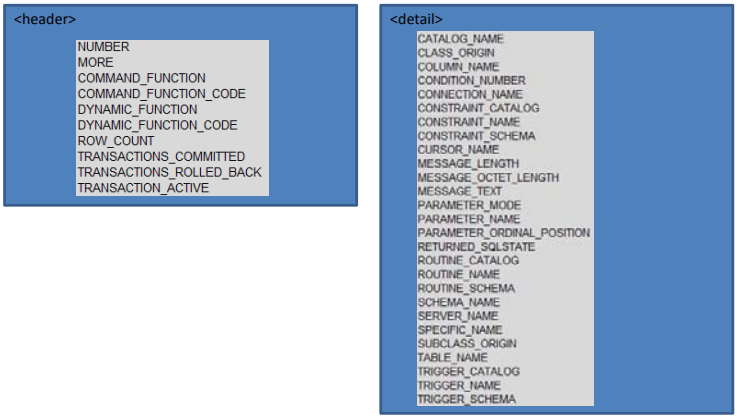
Transact-SQL of SQL Server:

```
BEGIN TRY
        <T-SQL statement(s)>
END TRY
BEGIN CATCH
       <exception handling based on
                ERROR_NUMBER(),
                ERROR_SEVERITY(),
                ERROR_STATE(),
                ERROR_PROCEDURE(),
                ERROR_LINE(),
                ERROR_MESSAGE()> ;
END CATCH;
```


PL/SQL Exception Handling

```
CREATE OR REPLACE PROCEDURE vatcalc (amount IN REAL,
                                     vatpc  IN REAL,
                                     vat    OUT REAL)
IS BEGIN
    DECLARE
        amount_out_of_range EXCEPTION;
        vatpc_out_of_range  EXCEPTION;
    BEGIN
        IF (amount < 0.0) THEN
            RAISE amount_out_of_range;
        END IF ;
        IF (vatpc < 0.0 OR vatpc > 30.0) THEN
            RAISE vatpc_out_of_range;
        END IF ;
        vat := vatpc * amount / 100.0;
    EXCEPTION
           WHEN amount_out_of_range THEN
             raise_application_error (-20001,
                   'amount cannot be negative') ;
        WHEN vatpc_out_of_range THEN
              raise_application_error (-20001,
                   'VAT% not between 0 and 30') ;
    END;
END;
/
```

ISO SQL Diagnostics Items



(1) .. (<max diagnostics detail count>)

<SQL statement> ;
GET DIAGNOSTICS <target> = <item> [, . . .]
If SQLSTATE = . . .

SQL GET DIAGNOSTICS

Example of getting diagnostics in MySQL 5.6:

INSERT INTO T (id, s) VALUES (2, NULL);
INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');

```
mysql> INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'
```

GET DIAGNOSTICS @rowcount = ROW_COUNT;
GET DIAGNOSTICS CONDITION 1
    @sqlstate = RETURNED_SQLSTATE,
    @sqlcode = MYSQL_ERRNO ;
SELECT @sqlstate, @sqlcode, @rowcount;

```
mysql> SELECT @sqlstate, @sqlcode, @rowcount;
+-----------+----------+-----------+
| @sqlstate | @sqlcode | @rowcount |
+-----------+----------+-----------+
| 23000     |     1062 |        -1 |
+-----------+----------+-----------+
1 row in set (0.00 sec)
```

xx

Diagnostics

	SQL-89	SQL-92	X/Open SC V2 1996	SQL/CLI	SQL:1999	SQL:2011	Mimer	DB2 LUW 9.5 /	Oracle 11g1	SQL Server 2012	MySQL 5.6.4	PostgreSQL 8.4	Pyrrho 4.8	Rdb7 7.1
SQLCA			Y					ESQL	ESQL	ESQL/C		ECPG		
SQLCODE	Y	Y	Y	Y				Y	Y	@@error error_number()				
SQLSTATE		Y	Y	Y	Y	Y		Y	Y	error_state()			Y	
FOUND												y		
GET DIAGNOSTICS	no	no	Y		Y	Y		Y	no	no	Y	Y	Y	Y
<statement info> i.e. diag. Header														
<target>=<st.info item>[, ...]														
<satement information item name>														
NUMBER				Y	Y	Y	Y				Y			
MORE				Y	Y	Y	Y							
COMMAND_FUNCTION					Y	Y	Y						Y	
COMMAND_FUNCTION_CODE					Y	Y							Y	
DYNAMIC_FUNCTION					Y	Y	Y							
DYNAMIC_FUNCTION_CODE					Y	Y								
ROW_COUNT				Y	Y	Y	Y	Y		@@rowcount	Y	Y	Y	Y
TRANSACTIONS_COMMITTED				Y	Y	Y							Y	Y
TRANSACTIONS_ROLLED_BACK				Y	Y	Y							Y	Y
TRANSACTION_ACTIVE				Y	Y	Y	Y							Y
product extensions:														
ACCESS_MODE														Y
CALLING_ROUTINE														Y
CONNECTION_NAME														Y
CURRENT_ROW														Y
GLOBAL_TRANSACTION														Y
ISOLATION_LEVEL														Y
DB2_RETURN_STATUS								Y						
DB2_SQL_NESTING_LEVEL								Y						
RESULT_OID												Y		
SQL/CLI extensions:														
RETURNCODE				Y										
<condition info> i.e. detail(s)														
EXCEPTION			Y		Y	no	Y	Y			no	no	no	Y
CONDITION			no		no	Y		no			Y	no	no	
<condition nr> <target>=<c.i.item> [, ...]														
<condition information item name>														
CATALOG_NAME			Y	Y	Y	Y	Y				Y		Y	
CLASS_ORIGIN			Y	Y	Y	Y	Y				Y		Y	
COLUMN_NAME			Y	Y	Y	Y	Y				Y			
CONDITION_NUMBER			Y	Y	Y	Y	Y							

Condition Handlers

DECLARE <condition name> CONDITION ;

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

SQLSTATE class:
nn
01
02

SIGNAL { <condition name> | SQLSTATE <value> } [SET MESSAGE_TEXT= '<text>']

Stored Procedure with Exception Handlers

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

May generate NOT FOUND

Generates SQLEXCEPTION on CHECK violation

Comparison of Procedure Features

	ISO SQL	DB2 Express-C	Oracle XE	SQL Server	MySQL	PostgreSQL	Pyrrho
	2011	10.5	11.2	2012	5.6	9.2	5.1
	6IWD6-02-Four	DB2SQLRefVol2	PL/SQL manual	BOL 2008	refman-5.6-en.	postgresql-9.0 PL/pgSQL	manual5.1
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()	p(..)	EXEC p @v, ..	CALL p()	CALL p()	CALL p()
positional	YES		YES	YES	YES		
by name	YES		YES	YES	YES		
BEGIN ATOMIC	YES	YES	default	compiled proce	N/A	default?	YES
atomic execution context	yes						
COMMIT	YES	YES	YES	YES	YES	N/A	YES
ROLLBACK	YES	YES	YES	YES	YES	N/A	YES
polymorphism	YES	?	YES	no	?	YES	YES
label	YES	YES	N/A	N/A	YES	N/A	?
<<label>> label:	N/A	YEs	YES	yes	YES	N/A	N/A
GOTO label	N/A	YES	YES	YES	YES	N/A	N/A
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

ISO SQL: Create Function

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

IN

```
LANGUGE {SQL | ADA | C | COBOL | FORTRAN
              | MUMPS | PASCAL | PLI | JAVA}
| PARAMETER STYLE  { SQL | GENERAL | JAVA}
| SPECIFIC  <specific name>
| DETERMINISTIC  |  NOT DETERMINISTIC
| { NO SQL |  CONTAINS SQL
    |  {READS | MODIFIES } SQL DATA }
| { RETURNS NULL ON NULL INPUT
    |  CALLED ON NULL INPUT }
```

A Scalar Function

Adapted from "DB2 Application Development" of R. Chong et al, p 106

IN parameter

```
CREATE FUNCTION deptname(p_empid VARCHAR(6))
RETURNS VARCHAR(30)
LANGUAGE SQL
BEGIN ATOMIC
  DECLARE v_department_name VARCHAR(30);
  DECLARE v_err VARCHAR(70);
  SET v_department_name =
     ( SELECT d.deptname FROM department d, employee e
       WHERE e.workdept=d.deptno AND e.empno= p_empid );
  SET v_err = 'Error: employee ' || p_empid || ' was not found';
  IF v_department_name IS NULL THEN
       SIGNAL SQLSTATE '80000' SET MESSAGE_TEXT=v_err;
  END IF;
  RETURN v_department_name;
END /

# test by CLP
db2 "SELECT (deptname ('000300')) FROM sysibm.sysdummy1"
```

local variables

A Table-valued Function

Adapted from "DB2 Application Development" of R. Chong et al, p 107

```
CREATE FUNCTION getEnumEmployee(p_dept VARCHAR(3))
RETURNS TABLE
     ( empno CHAR(6),
       lastname VARCHAR(15),
       firstnme VARCHAR(12))
SPECIFIC getEnumEmployee
RETURN
    SELECT e.empno, e.lastname, e.firstnme
    FROM employee e
    WHERE e.workdept=p_dept
/
```

Example on invoking the function:

```
SELECT * FROM
   TABLE (getEnumEmployee('E01')) T
```

TABLE() function

alias

Scalar Function Implementations

	ISO SQL	DB2 Express-C	Oracle XE	SQL Server	MySQL	PostgreSQL	Pyrrho
	2011	10.5	11.2	2012	5.6	9.2	5.1
	6IWD6-02-Four	DB2SQLRefVol2	PL/SQL manual	BOL 2008	refman-5.6-en.	postgresql-9.0	manual5.1
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
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
Invoking	f( )	f( )	f( )	f( )	f( )	CALL f() | f()	f()
			1) yes, but function cannot be used in SELECT statements				

Triggers

Application cannot call a trigger directly, but
an INSERT, UPDATE or DELETE statement may **activate** (fire) triggers which are controlling accesses to the table.
A trigger may

- Control or change the data of accessed rows validating the data against other tables or rows in the same table
- Prevent the action or replace the action by some other
- Execute actions to some other tables, for example for *tracing (logging)* of events, or even replicating data to external databases
- Stamping the updated version of the row for row version verifying (RVV) i.e. Optimistic locking

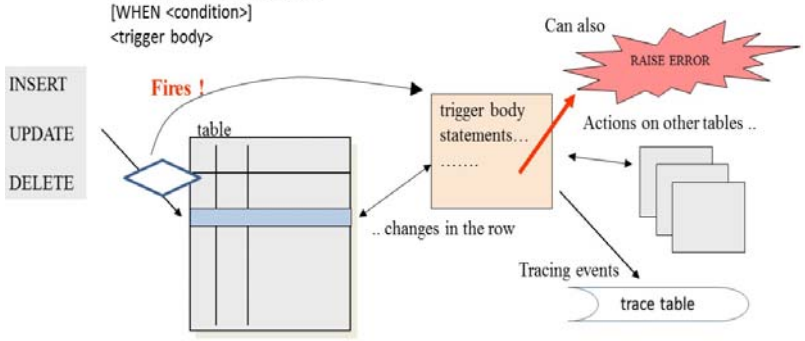
ISO SQL: CREATE TRIGGER

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

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

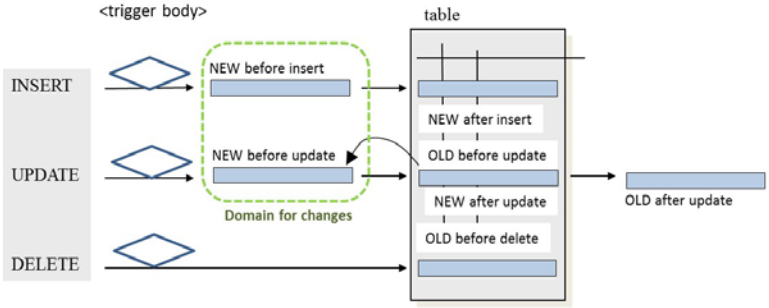
Some Possible Actions of a Trigger

```
CREATE TRIGGER <trigger name>
      {BEFORE | AFTER | INSTEAD OF}
      {INSERT | UPDATE | DELETE} ON <table>
[FOR EACH ROW]
[REFERENCING NEW AS <new>]
[REFERENCING OLD AS <old>]
[WHEN <condition>]
<trigger body>
```



REFERENCING clause for row-level triggers

```
CREATE TRIGGER <trigger name>
      {BEFORE | AFTER | INSTEAD OF}
      {INSERT | UPDATE | DELETE} ON <table>
[FOR EACH ROW]
[REFERENCING NEW AS <new>]
[REFERENCING OLD AS <old>]
[WHEN <condition>]
<trigger body>
```



Execution Order of the Triggers

1. Statement-level BEFORE trigger

For every row in turn affected by the statement:

2. Row-level BEFORE trigger for the row in turn

3. *<execution of the statement*

   with its immediate constraints>

4. Row-level AFTER trigger for the row in turn

5. Statement-level AFTER trigger.

MySQL: Workaround for CHECKs

```
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 ;
```

MySQL: RVV Trigger for 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 ;
```

Row version stamp

A simple RVV scenario using MySQL

step	Client A	Client B
1	SET AUTOCOMMIT = 0; SET @balanceB = 0; -- init value SET @rv = 0; -- init value SELECT balance, rv INTO @balance, @rv FROM Accounts WHERE acctID = 101; SELECT @balance, @rv; COMMIT;	
2		SET AUTOCOMMIT = 0; UPDATE Accounts SET balance = balance - 1000 WHERE acctID = 101; COMMIT;
3	UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101 AND rv = @rv; GET DIAGNOSTICS @rowcount = ROW_COUNT; SELECT @rowcount; ..	

Support of Referential Integrity (RI) rules

	ISO SQL	DB2	Oracle	SQL Server	MySQL	PostgreSQL	Pyrrho
	SQL:2006	LUW 9.7	11g1	2012	5.6	9.2	5.1
RULES							
ON UPDATE RESTRICT	Y	Y	N	N	Y	Y	?
ON UPDATE NO ACTION	Y	Y	N	Y	Y	Y	?
ON UPDATE CASCADE	Y	N	N	Y	Y	Y	?
ON UPDATE SET NULL	Y	N	N	Y	Y	Y	?
ON UPDATE SET DEFAULT	Y	N	N	Y	N	Y	?
ON DELETE RESTRICT	Y	Y	N	N	Y	Y	?
ON DELETE NO ACTION	Y	Y	N	Y	Y	Y	?
ON DELETE CASCADE	Y	Y	Y	Y	Y	Y	?
ON DELETE SET NULL	Y	Y	Y	Y	Y	Y	?
ON DELETE SET DEFAULT	Y	N	N	Y	N	Y	?
RESTRICT - not deferrable							
NO ACTION - deferrable							

The unsupported (N) RI rules of FOREIGN KEY constraints
can be implemented using triggers (- see the Stored Routines paper)

Compound Triggers of Oracle

Available in Oracle 11

Sections for timings and levels

```
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>;
/
```

Questions on Triggers

- ”Duplicate” triggers on the same target allowed?
  - timing, event, level
- Execution order of triggers watching the same target?
- Nesting (cascading) of triggers ?
- Maximum level of nesting (16, 32 ?)
- Recursion of trigger:
  - trigger fires action that fires the same trigger
  - prevention by NO CASCADE option?

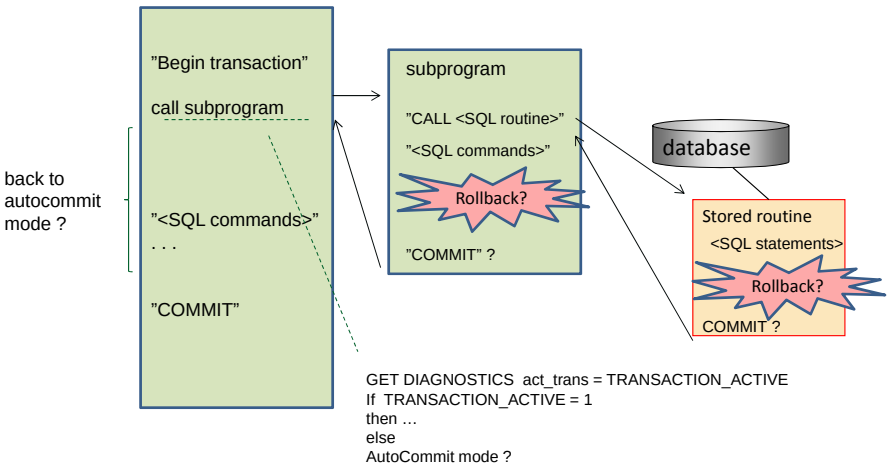
Comparison of Trigger Implementations

		ISO SQL	DB2 Express-C	Oracle XE	SQL Server	MySQL	PostgreSQL	Pyrrho
		2011	10.5	11.2	2012	5.6	9.2	5.1
		6IWD6-02-Four	DB2SQLRefVol2	PL/SQL manual	BOL 2008	refman-5.6-en.	postgresql-9.0	manual5.1
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
	NEW TABLE	Yes	Yes	N/A	INSERTED	N/A	N/A	
	OLD TABLE	Yes	Yes	N/A	DELETED	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	yes

Stored Routines and Transactions

- Can function contain DML ?
  - *Note: transaction may get aborted at any time*
- Calling autonomous transactions ?
- Can routines contain COMMIT or ROLLBACK ?
- How the application gets information on ROLLBACK ?
- etc ..

Get Diagnostics: Transaction_Active ?



42

Appendix

1. Result Set Processing by Cursor
2. Nested Transactions and Savepoints
3. Autonomous Transactions
   DB2 and Oracle
4. Calling Stored Procedures from Java
5. External Routines
   Procedures and functions

Cursor Programming

- Embedded SQL
- Procedural language implementations
  - Implicit FOR LOOP cursor
  - Explicit FOR LOOP cursor
- Implicit cursor of DML statementes
- Client-side Result Sets

Cursor in Embedded SQL (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);
```

PL/SQL Cursors

Implicit cursor of immediate DML

```
SQL.<attribute>
```

SQL Cursor FOR LOOP:

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

SQL Cursor FOR LOOP:

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

”Nested Transactions”

```
BEGIN TRANSACTION A;
  <do something>;
  BEGIN TRANSACTION B;
     <do something>;
  IF <something went wrong> THEN ROLLBACK;
     ELSE COMMIT ;

  BEGIN TRANSACTION C;
     <do something>;
  IF <something went wrong> THEN ROLLBACK ;
    ELSE COMMIT ;
  <do something>;
IF <something went wrong> THEN ROLLBACK ;
    ELSE COMMIT ;
```

Savepoints inside a transaction

- Savepoints allow **'partial ROLLBACKS'** inside a transaction
- Defined in SQL standard
- Syntax in implementations varies
- SQL statements:
  - **SAVEPOINT** sets a named savepoint for the current transaction
  - **RELEASE SAVEPOINT** clears named savepoint from the currently running transaction
  - **ROLLBACK SAVEPOINT** rolls back all statements done in the transaction after the named savepoint, and clears the savepoint from the currently running transaction

48

Compare the Implementations

Savepoint support	**SAVEPOINT** <name>	**RELEASE SAVEPOINT** <name>	**ROLLBACK TO SAVEPOINT** <name>
ISO SQL Standard	**yes**	**yes**	**yes**
DB2 SQL	SAVEPOINT <name> ON ROLLBACK RETAIN CURSORS	**yes**	**yes**
MySQL/InnoDB SQL	**yes**	**yes**	**yes**
PostgreSQL	**yes**	**yes**	**yes**
Oracle PL/SQL	**yes**	-	ROLLBACK TO <name>
SQL Server Transact-SQL	SAVE TRANSACTION <name>	-	ROLLBACK TRANSACTION <name>

49

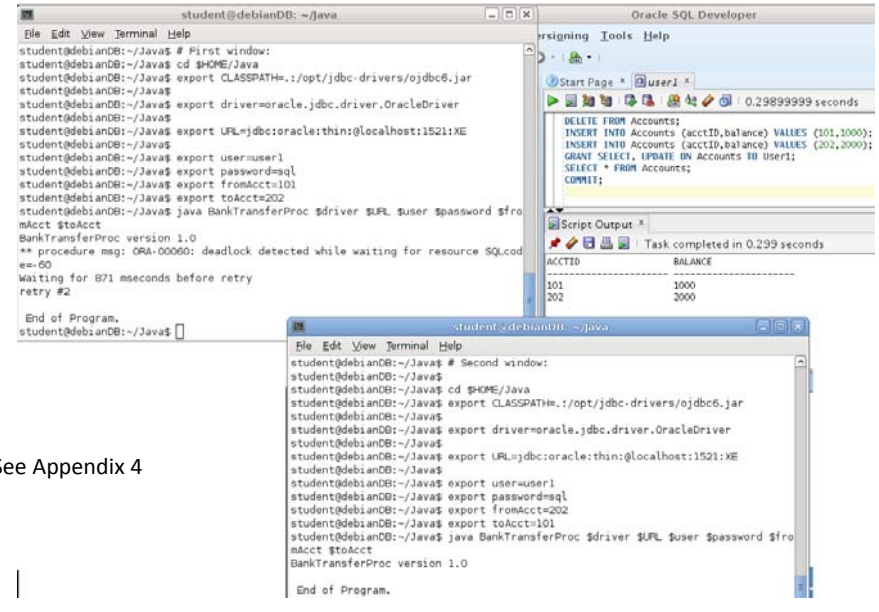
PL/SQL Autononoums Transaction

```
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;

. . .
myTraceProc ('BankTransfer',2,'updating fromAcct');
```

DB2 Autononoums Transaction

```
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 @    -- implicit COMMIT

GRANT EXECUTE ON myTraceProc TO PUBLIC;

. . .
CALL myTraceProc ('BankTransfer',2,'updating fromAcct');
```

4. Calling Stored Procedures from Java



See Appendix 4

5. External Routines

- A separate and still groving tutorial