Virtual DDBMS Stored Procedure Specification
Draft v0.8.0.0

Note: In this draft (*major* version is zero), compatibility between versions are not guaranteed.

Introduction

Syntax summary :

1. Each line must be a complete sentence (a valid sequence of statements).
2. Each line must only contains one complete sentence. Multiple sentences can be separated using two semicolons (';;').
3. A line is a group of strings (texts) separated either by one or more new-line characters ('\n' and/or '\r').
4. In case of a sentence is too long, the character '\' or character sequence '\\' can be used to split the sentence into multiple lines (the lines will be combined into one line by the interpreter).
5. Character sequence '//' indicates the beginning of a comment string (anything behind the character sequence is a comment).
6. The interpreter is trying to be not case-sensitive when parsing reserved keywords. However, it will be always case-sensitive when parsing all other symbols (function names, variable names, database names, table names, column names, etc.).

Creating/Defining Stored Procedure

```
  CREATE PROCEDURE sp_name ([<param_def>[,...]])
  ACCESS { PUBLIC | OWNER | DATABASE db_name }
[ COMMENT '<some text>' ]
  BEGIN
      <sp_body>
  END

  <param_def>
      { IN | OUT | INOUT } $param_name : <data_type>

  <data_type>
      { STRING | INTEGER | REAL | BOOLEAN }

  <sp_body>
      Valid VDDMS stored-procedure statements

  Example:

      PROCEDURE SelectFirstYearStudent (IN $Age : INTEGER)
      ACCESS PUBLIC
      BEGIN
          VAR $N : STRING = TableName
          VAR $Q : CURSOR FOR SELECT * FROM  #TableName \\
                                      WHERE Student.Age = $Age AND Student.Year = 1
          RETURN ROWSET $Q
      END

  Note: Operator '#' will convert a variable into a direct 'STRING' without quoting.
```

```
Note: - Access type 'PUBLIC' means the stored procedure is executable by all users. In case
        the stored procedure accessing any table, at least read-only access to the table is
        needed.
      - Access type 'OWNER' means the stored procedure is executable by only the owner.
      - Access type 'DATABASE' means the stored procedure is executable by only the owner
        only for the given database.
      - In case more than one stored procedures with the same names exist, the one belongs
        to the current owner will take precedence.
```

Deleting Stored Procedure

```
DROP PROCEDURE [IF EXISTS] sp_name
```

Displaying the Content of a Stored Procedure

```
DISPLAY PROCEDURE sp_name
```

Listing or Counting the Available Stored Procedures

```
SHOW | COUNT PROCEDURES
```

Calling Stored Procedure

```
CALL sp_name ([<param>[,...]])

Example:

    CALL SelectFirstYearStudent (18)
```

Defining Variable

```
VAR $var_name : <data_type> [ = <initial_value> ]

<data_type>
    { STRING | INTEGER | REAL | BOOLEAN |
      CURSOR FOR <select_statement>     |
      CALL sp_name ([<param>[,...]])
    }

Example:

    VAR $A : STRING  = 'Test'
    VAR $B : INTEGER = NULL
    VAR $C : REAL
    VAR $D : CURSOR FOR SELECT * FROM Student
    VAR $E : CURSOR FOR CALL SelectFirstYearStudent (18)

Note: A variable is only valid within the block it was defined.
```

Setting/Modifying Variable

```
SET $dvar_name = function_name ([<param>[,...]])

    or

SET $dvar_name = <f-operator> { $svar_name | constant }

    or

SET $dvar_name = { $svar1_name | constant1 } [ <m-operator> { $svar2_name | constant2 } ]

<f-operator>
    !        : BOOLEAN logical NOT
    HAS NEXT : CURSOR checking
    HAS NONE : CURSOR checking

<m-operator>
    .        : STRING concatenation
```

```
    %        : INTEGER modulus
    +        : INTEGER or REAL addition
    -        : INTEGER or REAL subtraction
    *        : INTEGER or REAL multiplication
    /        : INTEGER or REAL division

    ==       : STRING, INTEGER, REAL, or BOOLEAN comparison
    !=       : STRING, INTEGER, REAL, or BOOLEAN comparison
    <        : INTEGER or REAL comparison
    <=       : INTEGER or REAL comparison
    >        : INTEGER or REAL comparison
    >=       : INTEGER or REAL comparison

    &&       : BOOLEAN logical AND
    ||       : BOOLEAN logical OR

<boolean-constant>
    { true | false }

Example:

    SET $S1 = builtin::toupper ($S2)
    SET $I1 = $I2 + $I3
    SET $B1 = $I1 != $I4
    SET $B2 = ! $B1
    SET $S1 = $S2 . $S3 . $S4

Current limitation: - Except for STRING concatenation, one line can only contains
                      one operator.
                    - No grouping using '(' and ')' can be used.
                    - CURSOR variables cannot be assigned, compared, etc..
```

FETCH – INTO Statement

```
FETCH $cursor_var_name INTO { NULL | $dvar_name[,...] }

Example:

    VAR $ID    : INTEGER
    VAR $Name  : STRING
    VAR $Query : CURSOR FOR SELECT ID, Name FROM Student

    FETCH $Query INTO NULL // Discard one row
    FETCH $Query INTO $ID, $Name
```

RETURN ROWSET Statement

```
RETURN ROWSET { $var_name | constant }[,...]

Example:

    // Will return a row-set with two rows:
    //     the_sp_name
    //     0.0
    // and then end the execution of the stored procedure
    RETURN ROWSET 0.0

    // Will return a row-set with two rows:
    //     the_sp_name   the_sp_name      the_sp_name
    //     100           'Student Name'   'Student Address'
    // and then end the execution of the stored procedure
    RETURN ROWSET 100, 'Student Name', 'Student Address'

    // Will return a row-set with two rows:
    //     $A    $B
    //     100   200
    // and then end the execution of the stored procedure
    VAR $A : INTEGER = 200
    VAR $A : INTEGER = 100
    RETURN ROWSET $A, $B


    // Will return a row-set with a variable number of rows (just as if executing
    // the query directly) and then end the execution of the stored procedure
    VAR $Query : CURSOR FOR SELECT ID, Name FROM Student
    RETURN ROWSET $Query
```

EXIT, BREAK, and CONTINUE Statements

```
Example:

    EXIT     // Will exit the stored procedure (stop any further execution of statements)
    BREAK    // Error, cannot use the statement outside a loop
    CONTINUE // Error, cannot use the statement outside a loop

    WHILE true DO
        EXIT     // Will exit the stored procedure
        BREAK    // Will break the loop and execute any statement after the loop body
        CONTINUE // Will go to the start of the loop
    END WHILE

    <more statements>
```

THROW EXCEPTION Statement

```
THROW EXCEPTION { $var_name | constant }

Example:

    VAR $ID    : INTEGER
    VAR $Name  : STRING
    VAR $Query : CURSOR FOR SELECT ID, Name FROM Student

    IF HAS NEXT $Query THEN
        FETCH $Query INTO $ID, $Name
    ELSE
        THROW EXCEPTION 'No result'
    END IF
```

LABEL and GOTO Statements

```
LABEL <label_name>:

GOTO <label_name>

Example:

    LABEL Start:

        <statements>

    IF $I1 == $I2 THEN
        GOTO Start
    END IF

        <statements>

Note: - 'GOTO' statement to inside a block is forbidden.
      - 'GOTO' statement crossing variable initialization is forbidden.
```

IF Statement

```
IF <condition1> THEN
    <statements1>
[ ELSE IF <conditsion2> THEN
    <statements2>
]...
[ ELSE
    <statements3>
]
END IF

<condition>
    <f-operator> { $svar_name | constant }
or
    { $svar1_name | constant1 } <m-operator> { $svar2_name | constant2 }
```

```
<f-operator>
    !        : BOOLEAN logical NOT
    HAS NEXT : CURSOR checking
    HAS NONE : CURSOR checking

<m-operator>
    ==       : STRING, INTEGER, REAL, or BOOLEAN comparison
    !=       : STRING, INTEGER, REAL, or BOOLEAN comparison
    <        : INTEGER or REAL comparison
    <=       : INTEGER or REAL comparison
    >        : INTEGER or REAL comparison
    >=       : INTEGER or REAL comparison

    &&       : BOOLEAN logical AND
    ||       : BOOLEAN logical OR

Example:

    IF $I1 == $I2 THEN
        <statements>
    VAR $Q : CURSOR FOR SELECT ID, Name FROM Student
    ELSE IF $B1 && $B2
        <statements>
    ELSE IF ! $B3
        <statements>
    ELSE
        <statements>
    END IF


    IF HAS NEXT $Q THEN
        <statements>
    END IF

Note: - The sequence 'ELSE IF' must be written in the same line for it to be interpreted
        correctly. Hence, this code will cause syntax error:

            IF $I1 == $I2 THEN
                <statements>
            ELSE
            IF $B1 && $B2
                <statements>
            END IF

        as the interpreter will expect two 'END IF's.

      - Writing

            IF $B1 THEN

        is the same as writing

            IF $B1 == true THEN

        if B1 is BOOLEAN, otherwise it is an error.

Current limitation: - Each condition can only contains one operator.
                    - No grouping using '(' and ')' can be used.
                    - CURSOR variables cannot be assigned, compared, etc..
```

WHILE Statement

```
WHILE <condition> DO
    <statements>
    [ CONTINUE ]
    [ BREAK ]
END WHILE

Example:

    VAR $CNT   : INTEGER = 0
    VAR $ID    : INTEGER
    VAR $Name  : STRING
    VAR $Query : CURSOR FOR SELECT ID, Name FROM Student
```

```
    WHILE HAS NEXT $Query DO
        FETCH $Query INTO $ID, $Name
        IF $ID == NULL THEN
            CONTINUE
        ELSE IF $ID > 1000 THEN
            BREAK
        ELSE
           SQLX INSERT INTO Processed VALUES ($Cnt, $ID)
           SET $CNT = $CNT + 1
        END IF
    END WHILE

    VAR $Ret : CURSOR FOR SELECT * FROM Processed

    RETURN ROWSET $RET
```

Note: Please refer to the “IF Statement” for more details about the <*condition*>.

REPEAT – UNTIL Statement

```
REPEAT
    <statements>
    [ CONTINUE ]
    [ BREAK ]
UNTIL <condition>
END REPEAT
```

Note: Please refer to the “WHILE Statement” for more details.

LOOP Statement

```
LOOP
    <statements>
    [ CONTINUE ]
    [ BREAK ]
END LOOP
```

Example:

```
    LOOP
        <statements>
    END LOOP
```

Note: Please refer to the “WHILE Statement” for more details.

SQLX Statement

SQLX { <*insert_statement*> | <*delete_statement*> | <*update_statement*> }

Example:

```
    IF $I1 == $I2 THEN
        SQLX INSERT INTO Student VALUES (23, 1, $SName)
    ELSE IF $I1 == $I3 THEN
        SQLX DELETE FROM Student WHERE Student.Name = $SName
    ELSE
        SQLX UPDATE Student SET Student.Age = 23 WHERE Student.Name = $SName
    END IF
```

ON ERROR Statement

ON ERROR { THROW EXCEPTION | EXIT | IGNORE | SET $*var_name* = *constant* [RESET TO *constant*] }

Example:

```
    ON ERROR THROW EXCEPTION // The default behavior
    VAR $Q : CURSOR FOR SELECT * FROM Student

    ON ERROR EXIT
    WHILE true DO
        FETCH $Q INTO $A, $B // Will exit the stored procedure if no more record can be
                             // fetched
    END WHILE
```

```
    // Assume that the Student.ID must be unique
    ON ERROR IGNORE
    SQLX INSERT INTO Student VALUES (1, 1, 'Name 1')
    SQLX INSERT INTO Student VALUES (1, 1, 'Name 1') // Will continue to the next line
    SQLX INSERT INTO Student VALUES (2, 2, 'Name 2')

    VAR $FLAG : INTEGER = 0
    VAR $TMP1 : INTEGER
    VAR $TMP2 : INTEGER

    ON ERROR SET $FLAG = 1
    SQLX ...          // Assume the SQL command cause error
    SET $TMP1 = $FLAG // $TMP1 will be 1
    SET $TMP2 = $FLAG // $TMP2 will be 1

    ON ERROR SET $FLAG = 1 RESET TO 9
    SQLX ...          // Assume the SQL command cause error
    SET $TMP1 = $FLAG // $TMP1 will be 1
    SET $TMP2 = $FLAG // $TMP2 will be 9
```

Note: Only the “FETCH-INTO” and “SQLX” statements error behavior that can be modified.

IMPORT – AS Statement

IMPORT *lib_name* [AS *shortcut_name*]

Example:

```
    // In Linux, it will try to load the corresponding extension library
    // (let's say the name is gclddbms_sp_math.so) and bring all the exported
    // functions accessible under the namespace 'math'.
    IMPORT math

    // Same as above but under the namespace 'm'.
    IMPORT math AS m

    // Use the exported function
    VAR $A : REAL
    SET $A = math::cos(30) // These two lines are the same as the 'math' namespace is now
    SET $A = m::cos(30)    // also aliased as namespace 'm'
```

Note: All built-in functions are accessible from the namespace 'builtin'. The namespace can be aliased for easier use using the 'IMPORT' statement:

```
        IMPORT builtin AS b  // b::toupper() is now accessible
        IMPORT builtin AS :: // toupper() is now accessible without namespace specifier
```

Built-in Functions

Builtin functions for data conversion:

```
FUNCTION toupper
       (IN $Value : STRING)
RETURN STRING
```

Convert the given string to uppercase.

```
FUNCTION tolower
       (IN $Value : STRING)
RETURN STRING
```

Convert the given string to lowercase.

```
FUNCTION cnv_s2i
       (IN $Value : STRING)
RETURN INTEGER
```

Convert the given string to integer.

```
FUNCTION cnv_s2r
       (IN $Value : STRING)
RETURN REAL

       Convert the given string to real.


FUNCTION cnv_s2b
       (IN $Value : STRING)
RETURN BOOLEAN

       Convert the given string to boolean.


FUNCTION cnv_i2b
       (IN $Value : INTEGER)
RETURN BOOLEAN

       Convert the given integer to boolean.


FUNCTION cnv_r2i
       (IN $Value : REAL)
RETURN INTEGER

       Convert the given real to integer.


FUNCTION cnv_b2i
       (IN $Value : BOOLEAN)
RETURN INTEGER

       Convert the given boolean to integer.
```

Builtin functions for formatting date and/or time:

```
FUNCTION formatDate
       (IN $Year : INTEGER, IN $Month : INTEGER, IN $Day : INTEGER)
RETURN STRING

       Format date in the VDDBMS engine's internal format from the given integers.


FUNCTION formatTime
       (IN $Hour   : INTEGER, IN $Minute : INTEGER, IN $Second : INTEGER,
        IN $GMTOff : INTEGER)
RETURN STRING

       Format time in the VDDBMS engine's internal format from the given integers.


FUNCTION formatDateTime
       (IN $Year   : INTEGER, IN $Month  : INTEGER, IN $Day    : INTEGER,
        IN $Hour   : INTEGER, IN $Minute : INTEGER, IN $Second : INTEGER,
        IN $GMTOff : INTEGER)
RETURN STRING

       Format date and time in the VDDBMS engine's internal format from the given integers.
```

Builtin functions for atomic owner's-global-integer-variable (OGIV) manipulation and comparison:

```
FUNCTION atomicIntDefine
       (IN $Name : STRING, IN $InitVal : INTEGER)
RETURN INTEGER

       Define a new OGIV with the given name and initial value; returns back the initial
       value.


FUNCTION atomicIntDefineIfNotExists
       (IN $Name : STRING, IN $InitVal : INTEGER)
RETURN INTEGER

       Define a new OGIV with the given name and initial value; returns the initial value or
       the current value in case the variable already exists.
```

```
FUNCTION atomicIntUndefine
       (IN $Name : STRING)
RETURN INTEGER
```

Undefine the OGIV with the given name; returns the last value.

```
FUNCTION atomicIntUndefineIfExists
       (IN $Name : STRING)
RETURN INTEGER
```

Undefine the OGIV with the given name; returns the last value or zero in case the variable not exists.

```
FUNCTION atomicIntIsDefined
       (IN $Name : STRING)
RETURN BOOLEAN
```

Returns 'true' if an OGIV with the given name exists.

```
FUNCTION atomicIntLock
       (IN $Name : STRING)
RETURN BOOLEAN
```

Try to lock the OGIV with the given name; returns 'true' if can acquire the lock.

```
FUNCTION atomicIntUnlock
       (IN $Name : STRING)
RETURN BOOLEAN
```

Try to unlock the OGIV with the given name; returns 'true' if can unlock.

```
FUNCTION atomicIntIsLock
       (IN $Name : STRING)
RETURN BOOLEAN
```

Returns 'true' if an OGIV with the given name islocked.

```
FUNCTION atomicIntSet
       (IN $Name : STRING, IN $NewVal : INTEGER)
RETURN INTEGER
```

Set the OGIV of the given name with the new value; returns back the new value.

```
FUNCTION atomicIntGet
       (IN $Name : STRING)
RETURN INTEGER
```

Returns the value of the OGIV with the given name.

```
FUNCTION atomicIntGetAndSet
       (IN $Name : STRING, IN $NewVal : INTEGER)
RETURN INTEGER
```

Set the OGIV of the given name with the new value; returns the old value.

```
FUNCTION atomicIntGetAndSetIf
       (IN $Name : STRING, IN $NewVal : INTEGER, IN $Oper : STRING, IN $CmpVal : INTEGER)
RETURN INTEGER
```

Set the OGIV of the given name with the new value if the comparison between the current value and the comparator value produced a 'true'; returns the old value.

Supported comparison operators are: ==, !=, <, <=, >, and >=.

```
FUNCTION atomicIntGetAndInc
       (IN $Name : STRING, IN $NewVal : INTEGER)
RETURN INTEGER
```

Increment the OGIV with the given name by one; returns the old value.

```
FUNCTION atomicIntGetAndDec
       (IN $Name : STRING, IN $NewVal : INTEGER)
RETURN INTEGER
```

Increment the OGIV with the given name by one; returns the old value.

Other builtin functions:

```
FUNCTION msleep
       (IN $Min : INTEGER, IN $Max : INTEGER)
RETURN INTEGER
```

Sleep for few milliseconds; the duration of the sleep is randomly chosen between the given minimum and maximum value (currently the valid range is between 0 and 2500 milliseconds).