PUBLIC

SAP HANA Platform 2.0 SPS 00
Document Version: 1.0 – 2016-11-30

SAP HANA SQLScript Reference

Content

1 About SAP HANA SQLScript

SQLScript is a collection of extensions to the Structured Query Language (SQL). The extensions include:

- Data extension, which allows the definition of table types without corresponding tables
- Functional extension, which allows the definition of (side-effect free) functions which can be used to express and encapsulate complex data flows
- Procedural extension, which provides imperative constructs executed in the context of the database process

2 Backus Naur Form Notation

This document uses BNF (Backus Naur Form) which is the notation technique used to define programming languages. BNF describes the syntax of a grammar by using a set of production rules and by employing a set of symbols.

Symbols used in BNF

Table 1:

Symbol	Description
< >	Angle brackets are used to surround the name of a syntax element (BNF non-terminal) of the SQL language.
::=	The definition operator is used to provide definitions of the element appearing on the left side of the operator in a production rule.
[ ]	Square brackets are used to indicate optional elements in a formula. Optional elements may be specified or omitted.
{ }	Braces group elements in a formula. Repetitive elements (zero or more elements) can be specified within brace symbols.
|	The alternative operator indicates that the portion of the formula following the bar is an alternative to the portion preceding the bar.
...	The ellipsis indicates that the element may be repeated any number of times. If ellipsis appears after grouped elements, the grouped elements enclosed with braces are repeated. If ellipsis appears after a single element, only that element is repeated.
!!	Introduces normal English text. This is used when the definition of a syntactic element is not expressed in BNF.

BNF Lowest Terms Representations

Throughout the BNF used in this document each syntax term is defined to one of the lowest term representations shown below.

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
r | s | t | u | v | w | x | y | z
           | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
R | S | T | U | V | W | X | Y | Z

<any_character> ::= !!any character.

<comma> ::= ,

<dollar_sign> ::= $

<double_quotes> ::= "

<greater_than_sign> ::= >

<hash_symbol> ::= #

<left_bracket> ::= [
<left_curly_bracket> ::= {

<lower_than_sign> ::= <

<period> ::= .

<pipe_sign> ::= |

<right_bracket> ::= ]

<right_curly_bracket> ::= }

<sign> ::= + | -

<single_quote> ::= '

<underscore> ::= _

<apostrophe> ::= <single_quote>

<approximate_numeric_literal> ::= <mantissa>E<exponent>

<cesu8_restricted_characters> ::= <double_quote> | <dollar_sign> |
<single_quote> | <sign> | <period> | <greater_than_sign> | <lower_than_sign> |
<pipe_sign> | <left_bracket> | <right_bracket> | <left_curly_bracket> |
<right_curly_bracket> | ( | ) | ! | % | * |  , |  / | : | ; |  = | ? | @ | \ | ^
|  `

<exact_numeric_literal> ::= <unsigned_integer>[<period>[<unsigned_integer>]]
                          | <period><unsigned_integer>

<exponent> ::= <signed_integer>

<hostname> ::= {<letter> | <digit>}[{ <letter> | <digit> | <period> | - }...]

<identifier> ::= simple_identifier | special_identifier

<mantissa> ::= <exact_numeric_literal>

<numeric_literal> ::= <signed_numeric_literal> | <signed_integer>

<password> ::= {<letter> | <underscore> | <hash_symbol> | <dollar_sign> |
<digit>}... | <double_quotes> <any_character>...<double_quotes>

<port_number> ::= <unsigned_integer>

<schema_name> ::= <unicode_name>

<simple_identifier> ::= {<letter> | <underscore>} [{<letter> | <digit> |
<underscore> | <hash_symbol> | <dollar_sign>}...]

<special_identifier> ::= <double_quotes><any_character>...<double_quotes>

<signed_integer> ::= [<sign>] <unsigned_integer>

<signed_numeric_literal> ::= [<sign>] <unsigned_numeric_literal>

<string_literal> ::= <single_quote>[<any_character>...]<single_quote>

<unicode_name> ::= !! CESU-8 string excluding any characters listed in
<cesu8_restricted_characters>

<unsigned_integer> ::= <digit>...

<unsigned_numeric_literal> ::= <exact_numeric_literal> |
<approximate_numeric_literal>
```

```
<user_name> ::= <unicode_name>
```

3 What is SQLScript?

The motivation for SQLScript is to embed data-intensive application logic into the database. As of today, applications only offload very limited functionality into the database using SQL, most of the application logic is normally executed on an application server. The effect of that is that data to be operated upon needs to be copied from the database onto the application server and vice versa. When executing data intensive logic, this copying of data can be very expensive in terms of processor and data transfer time. Moreover, when using an imperative language like ABAP or JAVA for processing data, developers tend to write algorithms which follow a one-tuple-at-a-time semantics (for example, looping over rows in a table). However, these algorithms are hard to optimize and parallelize compared to declarative set-oriented languages like SQL.

The SAP HANA database is optimized for modern technology trends and takes advantage of modern hardware, for example, by having data residing in the main memory and allowing massive parallelization on multi-core CPUs. The goal of the SAP HANA database is to support application requirements by making use of such hardware. The SAP HANA database exposes a very sophisticated interface to the application, consisting of many languages. The expressiveness of these languages far exceeds that attainable with OpenSQL. The set of SQL extensions for the SAP HANA database, which allows developers to push data-intensive logic to the database, is called SQLScript. Conceptually SQLScript is related to stored procedures as defined in the SQL standard, but SQLScript is designed to provide superior optimization possibilities. SQLScript should be used in cases where other modeling constructs of SAP HANA, for example analytic views or attribute views are not sufficient. For more information on how to best exploit the different view types, see "Exploit Underlying Engine".

The set of SQL extensions are the key to avoid massive data copies to the application server and for leveraging sophisticated parallel execution strategies of the database. SQLScript addresses the following problems:

- Decomposing an SQL query can only be done using views. However when decomposing complex queries using views, all intermediate results are visible and must be explicitly typed. Moreover SQL views cannot be parameterized which limits their reuse. In particular they can only be used like tables and embedded into other SQL statements.
- SQL queries do not have features to express business logic (for example a complex currency conversion). As a consequence such a business logic cannot be pushed down into the database (even if it is mainly based on standard aggregations like SUM(Sales), etc.).
- An SQL query can only return one result at a time. As a consequence the computation of related result sets must be split into separate, usually unrelated, queries.
- As SQLScript encourages developers to implement algorithms using a set-oriented paradigm and not using a one tuple at a time paradigm, imperative logic is required, for example by iterative approximation algorithms. Thus it is possible to mix imperative constructs known from stored procedures with declarative ones.

Related Information

Exploit Underlying Engine [page 157]

3.1 SQLScript Security Considerations

You can develop secure procedures using SQLScript in SAP HANA by observing the following recommendations.

Using SQLScript, you can read and modify information in the database. In some cases, depending on the commands and parameters you choose, you can create a situation in which data leakage or data tampering can occur. To prevent this, SAP recommends using the following practices in all procedures.

- Mark each parameter using the keywords `IN` or `OUT`. Avoid using the `INOUT` keyword.
- Use the `INVOKER` keyword when you want the user to have the assigned privileges to start a procedure. The default keyword, `DEFINER`, allows only the owner of the procedure to start it.
- Mark read-only procedures using `READS SQL DATA` whenever it is possible. This ensures that the data and the structure of the database are not altered.

  Tip

  Another advantage to using `READS SQL DATA` is that it optimizes performance.
- Ensure that the types of parameters and variables are as specific as possible. Avoid using `VARCHAR`, for example. By reducing the length of variables you can reduce the risk of injection attacks.
- Perform validation on input parameters within the procedure.

Dynamic SQL

In SQLScript you can create dynamic SQL using one of the following commands: `EXEC` and `EXECUTE IMMEDIATE`. Although these commands allow the use of variables in SQLScript where they might not be supported. In these situations you risk injection attacks unless you perform input validation within the procedure. In some cases injection attacks can occur by way of data from another database table.

To avoid potential vulnerability from injection attacks, consider using the following methods instead of dynamic SQL:

- Use static SQL statements. For example, use the static statement, `SELECT` instead of `EXECUTE IMMEDIATE` and passing the values in the `WHERE` clause.
- Use server-side JavaScript to write this procedure instead of using SQLScript.
- Perform validation on input parameters within the procedure using either SQLScript or server-side JavaScript.
- Use `APPLY_FILTER` if you need a dynamic `WHERE` condition
- Use the SQL Injection Prevention Function

Escape Code

You might need to use some SQL statements that are not supported in SQLScript, for example, the `GRANT` statement. In other cases you might want to use the Data Definition Language (DDL) in which some `<name>`

elements, but not `<value>` elements, come from user input or another data source. The `CREATE TABLE` statement is an example of where this situation can occur. In these cases you can use dynamic SQL to create an escape from the procedure in the code.

To avoid potential vulnerability from injection attacks, consider using the folowing methods instead of escape code:

- Use server-side JavaScript to write this procedure instead of using SQLScript.
- Perform validation on input parameters within the procedure using either SQLScript or server-side JavaScript.

Related Information

SAP HANA Security Guide
SAP HANA SQL and System Views Reference
SQL Injection Prevention Functions [page 110]

3.2 SQLScript Processing Overview

To better understand the features of SQLScript, and their impact on execution, it can be helpful to understand how SQLScript is processed in the SAP HANA database.

When a user defines a new procedure, for example using the CREATE PROCEDURE statement, the SAP HANA database query compiler processes the statement in a similar way to an SQL statement. A step-by-step analysis of the process flow follows below:

- Parse the statement - detect and report simple syntactic errors.
- Check the statements semantic correctness - derive types for variables and check if their use is consistent.
- Optimize the code - optimization distinguishes between declarative logic displayed in the upper branch and imperative logic displayed in the lower branch. We shall discuss how the SAP HANA database recognizes them below.

When the procedure starts, the invoke activity can be divided into two phases:

1. Compilation
    - Code generation - for declarative logic the calculation models are created to represent the dataflow defined by the SQLScript code. It is optimized further by the calculation engine, when it is instantiated. For imperative logic the code blocks are translated into L-nodes.
    - The calculation models generated in the previous step are combined into a stacked calculation model.
2. Execution - the execution commences with binding actual parameters to the calculation models. When the calculation models are instantiated they can be optimized based on concrete input provided. Optimizations include predicate or projection embedding in the database. Finally, the instantiated calculation model is executed by using any of the available parts of the SAP HANA database.

With SQLScript you can implement applications by using both imperative orchestration logic and (functional) declarative logic, and this is also reflected in the way SQLScript processing works for those two coding styles.

Imperative logic is executed sequentially and declarative logic is executed by exploiting the internal architecture of the SAP HANA database and utilizing its potential for parallelism.

3.2.1 Orchestration Logic

Orchestration logic is used to implement data-flow and control-flow logic using imperative language constructs such as loops and conditionals. The orchestration logic can also execute declarative logic, which is defined in the functional extension by calling the corresponding procedures. In order to achieve an efficient execution on both levels, the statements are transformed into a dataflow graph to the maximum extent possible. The compilation step extracts data-flow oriented snippets out of the orchestration logic and maps them to data-flow constructs. The calculation engine serves as execution engine of the resulting dataflow graph. Since the language L is used as intermediate language for translating SQLScript into a calculation model, the range of mappings may span the full spectrum – from a single internal L-node for a complete SQLScript script in its simplest form, up to a fully resolved data-flow graph without any imperative code left. Typically, the dataflow graph provides more opportunities for optimization and thus better performance.

To transform the application logic into a complex data-flow graph two prerequisites have to be fulfilled:

- All data flow operations have to be side-effect free, that is they must not change any global state either in the database or in the application logic.
- All control flows can be transformed into a static dataflow graph.

In SQLScript the optimizer will transform a sequence of assignments of SQL query result sets to table variables into parallelizable dataflow constructs. The imperative logic is usually represented as a single node in the dataflow graph, and thus it is executed sequentially.

3.2.1.1 Example for Orchestration-Logic

```
CREATE PROCEDURE orchestrationProc LANGUAGE SQLSCRIPT READS
SQL DATA
AS
BEGIN
    DECLARE v_id BIGINT;
    DECLARE v_name VARCHAR(30);
    DECLARE v_pmnt BIGINT;
    DECLARE v_msg VARCHAR(200);
    DECLARE CURSOR c_cursor1 (p_payment BIGINT) FOR
        SELECT id, name, payment FROM control_tab
            WHERE payment > :p_payment
            ORDER BY id ASC;
    CALL init_proc();
    OPEN c_cursor1(250000);
    FETCH c_cursor1 INTO v_id, v_name, v_pmnt;
    v_msg = :v_name || ' (id ' || :v_id || ') earns ' || :v_pmnt || ' $.';
    CALL ins_msg_proc(:v_msg);
    CLOSE c_cursor1;
END
```

This procedure features a number of imperative constructs including the use of a cursor (with associated state) and local scalar variables with assignments.

Related Information

ins_msg_proc [page 169]

3.2.2 Declarative Logic

Declarative logic is used for efficient execution of data-intensive computations. This logic is represented internally as data flows which can be executed in a parallel manner. As a consequence, operations in a data-flow graph have to be free of side effects. This means they must not change any global state neither in the database, nor in the application. The first condition is ensured by only allowing changes on the dataset that is passed as input to the operator. The second condition is achieved by only allowing a limited subset of language features to express the logic of the operator. If those prerequisites are fulfilled, the following types of operators are available:

- SQL SELECT Statement
- Custom operators provided by SAP

Logically each operator represents a node in the data-flow graph. Custom operators have to be manually implemented by SAP.

4 Data Type Extension

Besides the built-in scalar SQL datatypes, SQLScript allows you to use user-defined types for tabular values.

4.1 Scalar Data Types

The SQLScript type system is based on the SQL-92 type system. It supports the following primitive data types:

Table 2:

Numeric types	TINYINT SMALLINT INT BIGINT DECIMAL SMALL-DECIMAL REAL DOUBLE
Character String Types	VARCHAR NVARCHAR ALPHANUM
Date-Time Types	TIMESTAMP SECONDDATE DATE TIME
Binary Types	VARBINARY
Large Object Types	CLOB NCLOB BLOB
Spatial Types	ST_GEOMETRY

Note

This also holds true for SQL statements, apart from the TEXT and SHORTTEXT types.

For more information on scalar types, see SAP HANA SQL and System Views Reference, Data Types.

4.2 Table Types

The SQLScript data type extension allows the definition of table types. These types are used to define parameters for procedures representing tabular results.

4.2.1 CREATE TYPE

Syntax

```
CREATE TYPE <type_name> AS TABLE (<column_list_definition>)
```

Syntax Elements

```
<type_name> ::= [<schema_name>.]<identifier>
```

Identifies the table type to be created and, optionally, in which schema the creation should take place

```
<column_list_definition> ::= <column_elem> [{, <column_elem>}...]
<column_elem> ::= <column_name> <data_type> [<column_store_data_type>]
[<ddic_data_type>]
<column_name> ::= <identifier>
```

Defines a table column

```
 <data_type> ::= DATE | TIME | SECONDDATE | TIMESTAMP | TINYINT | SMALLINT |
INTEGER | BIGINT | SMALLDECIMAL | DECIMAL
               | REAL | DOUBLE | VARCHAR | NVARCHAR | ALPHANUM | SHORTTEXT |
VARBINARY | BLOB | CLOB | NCLOB | TEXT
 <column_store_data_type> ::= CS_ALPHANUM | CS_INT | CS_FIXED | CS_FLOAT |
CS_DOUBLE | CS_DECIMAL_FLOAT | CS_FIXED(p-s, s)
                            | CS_SDFLOAT | CS_STRING | CS_UNITEDECFLOAT |
CS_DATE | CS_TIME | CS_FIXEDSTRING | CS_RAW
                            | CS_DAYDATE | CS_SECONDTIME | CS_LONGDATE |
CS_SECONDDATE
 <ddic_data_type> ::= DDIC_ACCP | DDIC_ALNM | DDIC_CHAR | DDIC_CDAY | DDIC_CLNT
| DDIC_CUKY | DDIC_CURR | DDIC_D16D
                    | DDIC_D34D | DDIC_D16R | DDIC_D34R | DDIC_D16S | DDIC_D34S
| DDIC_DATS | DDIC_DAY  | DDIC_DEC
                    | DDIC_FLTP | DDIC_GUID | DDIC_INT1 | DDIC_INT2 | DDIC_INT4
| DDIC_INT8 | DDIC_LANG | DDIC_LCHR
                    | DDIC_MIN  | DDIC_MON  | DDIC_LRAW | DDIC_NUMC | DDIC_PREC
| DDIC_QUAN | DDIC_RAW  | DDIC_RSTR
                    | DDIC_SEC  | DDIC_SRST | DDIC_SSTR | DDIC_STRG | DDIC_STXT
| DDIC_TIMS | DDIC_UNIT | DDIC_UTCM
                    | DDIC_UTCL | DDIC_UTCS | DDIC_TEXT | DDIC_VARC | DDIC_WEEK
```

The available data types

For more information on data types, see Scalar Data Types [page 15].

Description

The CREATE TYPE statement creates a user-defined type.

The syntax for defining table types follows the SQL syntax for defining new tables. The table type is specified using a list of attribute names and primitive data types. For each table type, attributes must have unique names.

Example

You create a table type called **`tt_publishers`**.

```
CREATE TYPE tt_publishers AS TABLE (
   publisher INTEGER,
   name VARCHAR(50),
   price DECIMAL,
   cnt INTEGER);
```

You create a table type called **`tt_years`**.

```
CREATE TYPE tt_years AS TABLE (
   year VARCHAR(4),
   price DECIMAL,
   cnt INTEGER);
```

4.2.2 DROP TYPE

Syntax

```
DROP TYPE <type_name> [<drop_option>]
```

Syntax Elements

```
<type_name> ::= [<schema_name>.]<identifier>
```

The identifier of the table type to be dropped, with optional schema name

```
<drop_option> ::= CASCADE | RESTRICT
```

When the <drop_option> is not specified, a non-cascaded drop is performed. This drops only the specified type, dependent objects of the type are invalidated but not dropped.

The invalidated objects can be revalidated when an object with the same schema and object name is created.

Description

The DROP TYPE statement removes a user-defined table type.

Example

You create a table type called `my_type`.

```
CREATE TYPE my_type AS TABLE ( column_a DOUBLE );
```

You drop the `my_type` table type.

```
DROP TYPE my_type;
```

5 Logic Container

In SQLScript there are two different logic containers, Procedure and User-Defined Function. The User-Defined Function container is separated into Scalar User-Defined Function and Table User-Defined Function.

The following sections provide an overview of the syntactical language description for both containers.

5.1 Procedures

Procedures allows you to describe a sequence of data transformations on data passed as input and database tables.

Data transformations can be implemented as queries that follow the SAP HANA database SQL syntax by calling other procedures. Read-only procedures can only call other read-only procedures.

The use of procedures has some advantages compared to using SQL:

- You can parameterize and reuse calculations and transformations described in one procedure in other procedures.
- You can use and express knowledge about relationships in the data; related computations can share common sub-expressions, and related results can be returned using multiple output parameters.
- You can define common sub-expressions. The query optimizer decides if a materialization strategy (which avoids recomputation of expressions) or other optimizing rewrites are best to apply. In any case, it eases the task of detecting common sub-expressions and improves the readability of the SQLScript code.
- You can use scalar variables or imperative language features if required.

5.1.1 CREATE PROCEDURE

You use this SQL statement to create a procedure.

Syntax

```
 CREATE PROCEDURE <proc_name> [(<parameter_clause>)] [LANGUAGE <lang>] [SQL
SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name>]
 [READS SQL DATA ] AS
 BEGIN [SEQUENTIAL EXECUTION]
   <procedure_body>
 END
```

Syntax Elements

The following syntax elements are available:

- Identifier of the procedure with an optional schema name

  ```
  <proc_name> ::= [<schema_name>.]<identifier>
  ```

- Input and output parameters of the procedure

  ```
  <parameter_clause> ::= <parameter> [{, <parameter>}...]
  ```

- Procedure parameter with associated data type

  ```
  <param_inout> ::= IN | OUT | INOUT
  ```

  Note

  The default is `IN`. Each parameter is marked using the keywords `IN`/`OUT`/`INOUT`. Input and output parameters must be explicitly assigned a type (that means that tables without a type are note supported)

- Variable name for a parameter

  ```
  <param_name> ::= <identifier>
  ```

- The input and output parameters of a procedure can have any of the primitive SQL types or a table type. `INOUT` parameters can only be of the scalar type.

  ```
  <param_type> ::= <sql_type> | <table_type> | <table_type_definition>
  ```

- Data type of the variable

  ```
   <sql_type> ::= DATE | TIME| TIMESTAMP | SECONDDATE | TINYINT | SMALLINT |
  INTEGER | BIGINT | DECIMAL | SMALLDECIMAL | REAL | DOUBLE
                | VARCHAR | NVARCHAR | ALPHANUM | VARBINARY | CLOB | NCLOB |
  BLOB | ST_GEOMETRY
  ```

  Note

  For more information on data types see Data Types in the SAP HANA SQL and System Views Reference.

- A table type previously defined with the `CREATE TYPE` command, see CREATE TYPE [page 16].

  ```
  <table_type> ::= <identifier>
  ```

- A table type implicitly defined within the signature

  ```
  <table_type_defintion>   ::=  TABLE (<column_list_definition>)
  <column_list_definition> ::= <column_elem>[{, <column_elem>}...]
  <column_elem> ::= <column_name> <data_type>
  <column_name> ::= <identifier>
  ```

- Definition of the programming language in the procedure. The default is `SQLSCRIPT`.

  ```
   LANGUAGE <lang>
   <lang> ::= SQLSCRIPT | R
  ```

➜ Tip

It is a good practice to define the language in all procedure definitions.

- Specification of the security mode of the procedure. The default is `DEFINER`.

```
SQL SECURITY <mode>
 <mode> ::= DEFINER | INVOKER
```

- Indication that that the execution of the procedure is performed with the privileges of the definer of the procedure

```
DEFINER
```

- Indication that the execution of the procedure is performed with the privileges of the invoker of the procedure

```
INVOKER
```

- Specifies the schema for unqualified objects in the procedure body; if nothing is specified, then the `current_schema` of the session is used.

```
DEFAULT SCHEMA <default_schema_name>
<default_schema_name> ::= <unicode_name>
```

- Marks the procedure as being read-only and side-effect free - the procedure does not make modifications to the database data or its structure. This means that the procedure does not contain DDL or DML statements and that it only calls other read-only procedures. The advantage of using this parameter is that certain optimizations are available for read-only procedures.

```
READS SQL DATA
```

- Defines the main body of the procedure according to the programming language selected

```
<procedure_body> ::= [<proc_decl_list>]
                     [<proc_handler_list>]
                      <proc_stmt_list>
```

- This statement forces sequential execution of the procedure logic. No parallelism takes place.

```
SEQUENTIAL EXECUTION
```

- Condition handler declaration

```
<proc_decl_list> ::= <proc_decl> [{, <proc_decl>}…]
<proc_decl> ::= DECLARE  {<proc_variable>|<proc_table_variable>|<proc_cursor>|
<proc_condition>} ;
<proc_table_variable> ::= <variable_name_list> {<table_type_definition>|
<table_type>}
<proc_variable>::= <variable_name_list> [CONSTANT] {<sql_type>|
<array_datatype>}[NOT NULL][<proc_default>]
<variable_name_list>    ::= <variable_name>[{, <variable_name}...]
<column_list_elements> ::= (<column_definition>[{,<column_definition>}...])
<array_datatype>    ::= <sql_type> ARRAY [ = <array_constructor> ]
<array_constructor>    ::= ARRAY (<expression> [ { , <expression> }...] )
<proc_default> ::= (DEFAULT | '=' ) <value>|<expression>
<value>    ::= An element of the type specified by <type> or an expression
<proc_cursor> ::= CURSOR <cursor_name> [ ( proc_cursor_param_list ) ] FOR
<subquery> ;
<proc_cursor_param_list> ::= <proc_cursor_param> [{, <proc_cursor_param>}...]
<variable_name>           ::= <identifier>
<cursor_name>    ::= <identifier>
```

```
<proc_cursor_param>    ::= <param_name> <datatype>
<proc_condition>    ::= <variable_name> CONDITION | <variable_name> CONDITION 
FOR <sql_error_code>
```

- Declares exception handlers to catch SQL exceptions.

```
<proc_handler_list> ::= <proc_handler> [{, <proc_handler>}...]
<proc_handler>::= DECLARE EXIT HANDLER FOR <proc_condition_value_list> 
<proc_stmt> ;
```

- One or more condition values

```
 <proc_condition_value_list> ::= <proc_condition_value>
{,<proc_condition_value>}...]
```

- An error code number or a condition name declared for a condition variable

```
 <proc_condition_value> ::= SQLEXCEPTION
                          | <sql_error_code> | <condition_name>
```

- Procedure body statements.

```
 <proc_stmt_list> ::= {<proc_stmt>}...
 <proc_stmt> ::= <proc_block>
               | <proc_assign>
               | <proc_single_assign>
               | <proc_multi_assign>
               | <proc_if>
               | <proc_loop>
               | <proc_while>
               | <proc_for>
               | <proc_foreach>
               | <proc_exit>
               | <proc_continue>
               | <proc_signal>
               | <proc_resignal>
               | <proc_sql>
               | <proc_open>
               | <proc_fetch>
               | <proc_close>
               | <proc_call>
               | <proc_exec>
               | <proc_return>
```

- Sections of your procedures can be nested using BEGIN and END terminals

```
<proc_block> ::= BEGIN <proc_block_option>
                  [<proc_decl_list>]
                  [<proc_handler_list>]
                  <proc_stmt_list>
              END ;
<proc_block_option> ::=  [SEQUENTIAL EXECUTION ]| [AUTONOMOUS TRANSACTION] | 
[PARALLEL EXECUTION]
```

- Assignment of values to variables - an <expression> can be either a simple expression, such as a character, a date, or a number, or it can be a scalar function or a scalar user-defined function.

```
<proc_assign> ::= <variable_name> = { <expression> | <array_function> } ;
                | <variable_name> '[' <expression> ']' = <expression>  ;
```

- The ARRAY_AGG function returns the array by aggregating the set of elements in the specified column of the table variable. Elements can optionally be ordered.
  The CARDINALITY function returns the number of the elements in the array, <array_variable_name>.

The TRIM_ARRAY function returns the new array by removing the given number of elements, <numeric_value_expression>, from the end of the array, <array_value_expression>.
The ARRAY function returns an array whose elements are specified in the list <array_variable_name>. For more information see the chapter ARRAY [page 101].

```
<array_function> = ARRAY_AGG   ( :<table_variable>.<column_name> [ ORDER BY
<sort_spec_list> ] )
                    | CARDINALITY ( :<array_variable_name>)
                    | TRIM_ARRAY  ( :<array_variable_name> ,
<array_variable_name>)
                    | ARRAY ( <array_variable_name_list> )
 <table_variable>      ::= <identifier>
 <column_name>         ::= <identifier>
 <array_variable_name> ::= <identifier>
```

- Assignment of values to a list of variables with only one function evaluation. For example, `<function_expression>` must be a scalar user-defined function and the number of elements in `<var_name_list>` must be equal to the number of output parameters of the scalar user-defined function.

```
<proc_multi_assign> ::= (<var_name_list>) = <function_expression>
```

```
 <proc_single_assign> ::= <variable_name> = <subquery>
                        |  <variable_name> = <proc_ce_call>
                        |  <variable_name> = <proc_apply_filter>
                        |  <variable_name> = <unnest_function>
                        |  <variable_name> = <map_merge_op>
```

- The MAP_MERGE operator is used to apply each row of the input table to the mapper function and unite all intermediate result tables. For more information, see Map Merge [page 66].

```
<map_merge_op> ::= MAP_MERGE(<table_or_table_variable>,
<mapper_identifier>(<table_or_table_variable>.<column_name> [ {,
<table_or_table_variable>.<column_name>} … ] [, <param_list>])
<table_or_table_variable> ::= <table_variable_name> | <identifier>
<table_variable_name> ::= <identifier>
<mapper_identifier> ::= <identifier>
<column_name> ::= <identifier>
<param_list> ::= <param> [{, <param>} …]
<paramter> = <table_or_table_variable> | <string_literal> | <numeric_literal>
| <identifier>
```

- For more information about the CE operators, see Calculation Engine Plan Operators [page 116].

```
 <proc_ce_call> ::= TRACE ( <variable_name> ) ;
                  | CE_LEFT_OUTER_JOIN ( <table_variable> ,
<table_variable> , '[' <expr_alias_comma_list> ']' [ <expr_alias_vector>]  ) ;
                  | CE_RIGHT_OUTER_JOIN ( <table_variable> ,
<table_variable> , '[' <expr_alias_comma_list> ']' [ <expr_alias_vector>] ) ;
                  | CE_FULL_OUTER_JOIN ( <table_variable> ,
<table_variable> , '[' <expr_alias_comma_list> ']' [ <expr_alias_vector>]  );
                  | CE_JOIN ( <table_variable> , <table_variable> , '['
<expr_alias_comma_list> ']' [<expr_alias_vector>]  ) ;
                  | CE_UNION_ALL ( <table_variable> , <table_variable> ) ;
                  | CE_COLUMN_TABLE ( <table_name> [ <expr_alias_vector>]  ) ;
                  | CE_JOIN_VIEW ( <table_name> [ <expr_alias_vector>] ) ;
                  | CE_CALC_VIEW ( <table_name> [ <expr_alias_vector>] ) ;
                  | CE_OLAP_VIEW ( <table_name> [ <expr_alias_vector>] ) ;
                  | CE_PROJECTION ( <table_variable> , '['
<expr_alias_comma_list> ']' <opt_str_const> ) ;
                  | CE_PROJECTION ( <table_variable> <opt_str_const> ) ;
                  | CE_AGGREGATION ( <table_variable> , '['
<agg_alias_comma_list> ']' [ <expr_alias_vector>] );
```

```
                  | CE_CONVERSION ( <table_variable> , '['
<proc_key_value_pair_comma_list> ']' [ <expr_alias_vector>] ) ;
                  | CE_VERTICAL_UNION ( <table_variable> , '['
<expr_alias_comma_list> ']' <vertical_union_param_pair_list> ) ;
                  | CE_COMM2R ( <table_variable> , <int_const> ,
<str_const> , <int_const> , <int_const> , <str_const> ) ;
 <table_name>  ::= [<schema_name>.]<identifier>
```

- APPLY_FILTER defines a dynamic WHERE-condition <variable_name> that is applied during runtime. For more information about that, see the chapter APPLY_FILTER [page 93].

```
 <proc_apply_filter> ::= APPLY_FILTER ( {<table_name> | :<table_variable>},
<variable_name> ) ;
```

- The UNNEST function returns a table including a row for each element of the specified array.

```
<unnest_function> ::= UNNEST ( <variable_name_list> ) [ WITH ORDINALITY ]
[<as_col_names>] ;
<variable_name_list> ::= :<variable_name> [{, :<variable_name>}...]
```

- Appends an ordinal column to the return values.

```
WITH ORDINALTIY
```

- Specifies the column names of the return table.

```
<as_col_names>     ::= AS [table_name] ( <column_name_list> )
 <column_name_list> ::= <column_name>[{, <column_name>}...]
 <column_name>      ::= <identifier>
```

- You use IF - THEN - ELSE IF to control execution flow with conditionals.

```
 <proc_if> ::= IF <condition> THEN [SEQUENTIAL EXECUTION][<proc_decl_list>]
[<proc_handler_list>] <proc_stmt_list>
               [<proc_elsif_list>]
               [<proc_else>]
               END IF ;
 <proc_elsif_list> ::= ELSEIF <condition> THEN [SEQUENTIAL EXECUTION]
[<proc_decl_list>] [<proc_handler_list>] <proc_stmt_list>
 <proc_else> ::= ELSE [SEQUENTIAL EXECUTION][<proc_decl_list>]
[<proc_handler_list>] <proc_stmt_list>
```

- You use loop to repeatedly execute a set of statements.

```
 <proc_loop> ::= LOOP [SEQUENTIAL EXECUTION][<proc_decl_list>]
[<proc_handler_list>] <proc_stmt_list> END LOOP ;
```

- You use WHILE to repeatedly call a set of trigger statements while a condition is true.

```
 <proc_while> ::= WHILE <condition> DO [SEQUENTIAL EXECUTION]
[<proc_decl_list>] [<proc_handler_list>] <proc_stmt_list> END WHILE ;
```

- You use FOR - IN loops to iterate over a set of data.

```
 <proc_for> ::= FOR <column_name> IN [ REVERSE ] <expression> [...]
<expression>
             DO [SEQUENTIAL EXECUTION][<proc_decl_list>]
[<proc_handler_list>] <proc_stmt_list>
             END FOR ;
```

- You use FOR - EACH loops to iterate over all elements in a set of data.

```
 <proc_foreach> ::= FOR <column_name> AS <column_name> [<open_param_list>] DO
```

```
                  [SEQUENTIAL EXECUTION][<proc_decl_list>]
[<proc_handler_list>] <proc_stmt_list>
                  END FOR ;
 <open_param_list> ::= ( <expression> [ { , <expression> }...] )
```

- Terminates a loop

```
<proc_exit>     ::= BREAK ;
```

- Skips a current loop iteration and continues with the next value.

```
<proc_continue> ::= CONTINUE ;
```

- You use the SIGNAL statement to explicitly raise an exception from within your trigger procedures.

```
<proc_signal>    ::=  SIGNAL <signal_value> [<set_signal_info>] ;
```

- You use the RESIGNAL statement to raise an exception on the action statement in an exception handler. If an error code is not specified, RESIGNAL will throw the caught exception.

```
<proc_resignal> ::= RESIGNAL [<signal_value>] [<set_signal_info>] ;
```

- You can SIGNAL or RESIGNAL a signal name or an SQL error code.

```
<signal_value>    ::= <signal_name> | <sql_error_code>
<signal_name>     ::= <identifier>
<sql_error_code> ::= <unsigned_integer>
```

- You use SET MESSAGE_TEXT to deliver an error message to users when specified error is thrown during procedure execution.

```
<set_signal_info> ::= SET MESSAGE_TEXT = '<message_string>'
<message_string>  ::= <any_character>
```

-
  ```
  <proc_sql> ::=  <subquery>
                | <select_into_stmt>
                | <insert_stmt>
                | <delete_stmt>
                | <update_stmt>
                | <replace_stmt>
                | <call_stmt>
                | <create_table>
                | <drop_table>
                | <truncate_statement>
  ```

  For information on `<insert_stmt>`, see `INSERT` in the SAP HANA SQL and System Views Reference.
  For information on `<delete_stmt>`, see `DELETE` in the SAP HANA SQL and System Views Reference.
  For information on `<update_stmt>`, see `UPDATE` in the SAP HANA SQL and System Views Reference.
  For information on `<replace_stmt>` and `<upsert_stmt>`, see `REPLACE` and `UPSERT` in the SAP HANA SQL and System Views Reference.
  For information on `<truncate_stmt>`, see `TRUNCATE` in the SAP HANA SQL and System Views Reference.

-
  ```
  <select_into_stmt> ::= SELECT <select_list> INTO <var_name_list>
                         <from_clause >
                         [<where_clause>]
                         [<group_by_clause>]
                         [<having_clause>]
                         [{<set_operator> <subquery>, ... }]
                         [<order_by_clause>]
                         [<limit>] ;
  ```

- `<var_name>` is a scalar variable. You can assign selected item value to this scalar variable.

```
<var_name_list> ::= <var_name>[{, <var_name>}...]
<var_name>      ::= <identifier>
```

- Cursor operations

```
<proc_open>  ::= OPEN <cursor_name> [ <open_param_list>] ;
<proc_fetch> ::= FETCH <cursor_name> INTO <column_name_list> ;
<proc_close> ::= CLOSE <cursor_name> ;
```

- Procedure call. For more information, see CALL - Internal Procedure Call [page 32]

```
<proc_call> ::= CALL <proc_name> (<param_list>) ;
```

- Use EXEC to make dynamic SQL calls

```
<proc_exec>   ::= {EXEC | EXECUTE IMMEDIATE} <proc_expr> ;
```

- Return a value from a procedure

```
<proc_return> ::= RETURN [<proc_expr>] ;
```

Description

The CREATE PROCEDURE statement creates a procedure by using the specified programming language <lang>.

Example

Example: Creating a Procedure

You create an SQLScript procedure with the following definition:

```
 CREATE PROCEDURE orchestrationProc
 LANGUAGE SQLSCRIPT AS
 BEGIN
   DECLARE v_id BIGINT;
   DECLARE v_name VARCHAR(30);
   DECLARE  v_pmnt BIGINT;
   DECLARE v_msg VARCHAR(200);
   DECLARE CURSOR c_cursor1 (p_payment BIGINT) FOR
     SELECT id, name, payment FROM control_tab
       WHERE payment > :p_payment ORDER BY id ASC;
   CALL init_proc();
   OPEN c_cursor1(250000);
   FETCH c_cursor1 INTO v_id, v_name, v_pmnt; v_msg = :v_name || ' (id '
|| :v_id || ') earns ' || :v_pmnt || ' $.';
   CALL ins_msg_proc(:v_msg);
   CLOSE c_cursor1;
 END;
```

The procedure features a number of imperative constructs including the use of a cursor (with associated state) and local scalar variables with assignments.

5.1.2 DROP PROCEDURE

Syntax

```
DROP PROCEDURE <proc_name> [<drop_option>]
```

Syntax Elements

```
<proc_name> ::= [<schema_name>.]<identifier>
```

The name of the procedure to be dropped, with optional schema name

```
<drop_option> ::= CASCADE | RESTRICT
```

If you do not specify the `<drop_option>`, the system performs a non-cascaded drop. This will only drop the specified procedure; dependent objects of the procedure will be invalidated but not dropped.

The invalidated objects can be revalidated when an object that uses the same schema and object name is created.

```
CASCADE
```

Drops the procedure and dependent objects

```
RESTRICT
```

This parameter drops the procedure only when dependent objects do not exist. If this drop option is used and a dependent object exists an error will be sent.

Description

This statement drops a procedure created using `CREATE PROCEDURE` from the database catalog.

Examples

You drop a procedure called `my_proc` from the database using a non-cascaded drop.

```
DROP PROCEDURE my_proc;
```

5.1.3 ALTER PROCEDURE

You can use ALTER PROCEDURE if you want to change the content and properties of a procedure without dropping the object.

```
ALTER PROCEDURE <proc_name> [(<parameter_clause>)] [LANGUAGE <lang>]
[DEFAULT SCHEMA <default_schema_name>]
[READS SQL DATA] AS
BEGIN [SEQUENTIAL EXECUTION]
  <procedure_body>
END
```

For more information about the parameters, refer to CREATE PROCEDURE [page 19].

For instance, with ALTER PROCEDURE you can change the content of the body itself. Consider the following GET_PROCEDURES procedure that returns all procedure names on the database.

```
CREATE PROCEDURE GET_PROCEDURES(OUT procedures TABLE(schema_name NVARCHAR(256),
name NVARCHAR(256)))
AS
BEGIN
   procedures = SELECT schema_name AS schema_name, procedure_name AS name FROM
PROCEDURES;
END;
```

The procedure GET_PROCEDURES should now be changed to return only valid procedures. In order to do so, use ALTER PROCEDURE:

```
ALTER PROCEDURE GET_PROCEDURES( OUT procedures TABLE(schema_name NVARCHAR(256),
name NVARCHAR(256)))
AS
BEGIN
   procedures = SELECT schema_name AS schema_name, procedure_name AS name FROM
PROCEDURES WHERE IS_VALID = 'TRUE';
END;
```

Besides changing the procedure body, you can also change the language <lang> of the procedure, the default schema <default_schema_name> as well as change the procedure to read only mode (READS SQL DATA).

Note

The following properties cannot be changed with ALTER PROCEDURE:

- procedure owner
- security mode (INVOKER, DEFINER)
- procedure type (table function, scalar function, procedure)
- parameter signature (parameter name, parameter type, default value)

If you need to change these properties you have to drop and recreate the procedure by using DROP PROCEDURE and CREATE PROCEDURE.

Note that if the default schema and read only mode are not explicitly specified, they will be removed. Language is defaulted to SQLScript.

Note

You must have the `ALTER` privilege for the object you want to change.

5.1.4 ALTER PROCEDURE RECOMPILE

Syntax

```
ALTER PROCEDURE <proc_name> RECOMPILE [WITH PLAN]
```

Syntax Elements

```
<proc_name> ::= [<schema_name>.]<identifier>
```

The identifier of the procedure to be altered, with the optional schema name.

```
WITH PLAN
```

Specifies that internal debug information should be created during execution of the procedure.

Description

The `ALTER PROCEDURE RECOMPILE` statement manually triggers a recompilation of a procedure by generating an updated execution plan. For production code a procedure should be compiled without the `WITH PLAN` option to avoid overhead during compilation and execution of the procedure.

Example

You trigger the recompilation of the `my_proc` procedure to produce debugging information.

```
ALTER PROCEDURE my_proc RECOMPILE WITH PLAN;
```

5.1.5 Procedure Calls

A procedure can be called either by a client on the outer-most level, using any of the supported client interfaces, or within the body of a procedure.

➜ Recommendation

SAP recommends that you use parameterized `CALL` statements for better performance. The advantages follow.

- The parameterized query compiles only once, thereby reducing the compile time.
- A stored query string in the SQL plan cache is more generic and a precompiled query plan can be reused for the same procedure call with different input parameters.
- By not using query parameters for the `CALL` statement, the system triggers a new query plan generation.

5.1.5.1 CALL

Syntax

```
CALL <proc_name> (<param_list>) [WITH OVERVIEW]
```

Syntax Elements

```
<proc_name> ::= [<schema_name>.]<identifier>
```

The identifier of the procedure to be called, with optional schema name.

```
<param_list> ::= <proc_param>[{, <proc_param>}...]
```

Specifies one or more procedure parameters.

```
 <proc_param> ::= <identifier> | <string_literal> | <unsigned_integer> |
<signed_integer>| <signed_numeric_literal> | <unsigned_numeric_literal> |
<expression>
```

Procedure parameters

For more information on these data types, see Backus Naur Form Notation [page 7] and Scalar Data Types [page 15].

Parameters passed to a procedure are scalar constants and can be passed either as `IN`, `OUT` or `INOUT` parameters. Scalar parameters are assumed to be NOT NULL. Arguments for IN parameters of table type can either be physical tables or views. The actual value passed for tabular OUT parameters must be`?`.

```
WITH OVERVIEW
```

Defines that the result of a procedure call will be stored directly into a physical table.

Calling a procedure `WITH OVERVIEW` returns one result set that holds the information of which table contains the result of a particular table's output variable. Scalar outputs will be represented as temporary tables with only one cell. When you pass existing tables to the output parameters `WITH OVERVIEW` will insert the result-set tuples of the procedure into the provided tables. When you pass '?' to the output parameters, temporary tables holding the result sets will be generated. These tables will be dropped automatically once the database session is closed.

Description

Calls a procedure defined with CREATE PROCEDURE [page 19].

`CALL` conceptually returns a list of result sets with one entry for every tabular result. An iterator can be used to iterate over these results sets. For each result set you can iterate over the result table in the same manner as you do for query results. SQL statements that are not assigned to any table variable in the procedure body are added as result sets at the end of the list of result sets. The type of the result structures will be determined during compilation time but will not be visible in the signature of the procedure.

`CALL` when executed by the client the syntax behaves in a way consistent with the SQL standard semantics, for example, Java clients can call a procedure using a JDBC `CallableStatement`. Scalar output variables are a scalar value that can be retrieved from the callable statement directly.

i Note

Unquoted identifiers are implicitly treated as upper case. Quoting identifiers will respect capitalization and allow for using white spaces which are normally not allowed in SQL identifiers.

Examples

In these examples, consider the following procedure signature:

```
CREATE PROCEDURE proc(
            IN value integer,IN currency nvarchar(10),OUT outTable typeTable,
            OUT valid integer)
AS
BEGIN
    …
END;
```

Calling the `proc` procedure:

```
CALL proc (1000, 'EUR', ?, ?);
```

Calling the `proc` procedure in debug mode:

```
 CALL proc (1000, 'EUR', ?, ?) IN DEBUG MODE;
```

Calling the `proc` procedure using the `WITH OVERVIEW` option:

```
CALL proc(1000, 'EUR', ?, ?) WITH OVERVIEW;
```

It is also possible to use scalar user defined function as parameters for procedure call:

```
CALL proc(udf(),’EUR’,?,?);
CALL proc(udf()* udf()-55,’EUR’, ?, ?);
```

In this example, `udf()` is a scalar user-defined function. For more information about scalar user-defined functions, see CREATE FUNCTION [page 44]

5.1.5.2 CALL - Internal Procedure Call

Syntax:

```
CALL <proc_name > (<param_list>)
```

Syntax Elements:

```
<param_list> ::= <param>[{, <param>}...]
```

Specifies procedure parameters

```
<param>::= <in_table_param> | <in_scalar_param> |<out_scalar_param> |
<out_table_param>
```

Parmeters can be either table or scalar type.

```
<in_table_param> ::= <read_variable_identifier>|<sql_identifier>
<in_scalar_param>::=<read_variable_identifier>|<scalar_value>|<expression>
```

```
<in_param> ::= :<identifier>
```

Specifies a procedure input parameter

Note

Use a colon before the identifier name.

```
<out_param> ::= <identifier>
```

Specifies a procedure input parameter

Description:

For an internal procedure, in which one procedure calls another procedure, all existing variables of the caller or literals are passed to the `IN` parameters of the callee and new variables of the caller are bound to the `OUT` parameters of the callee. That is to say, the result is implicitly bound to the variable that is given in the function call.

Example:

```
CALL addDiscount (:lt_expensive_books, lt_on_sale);
```

When procedure `addDiscount` is called, the variable `<:lt_expensive_books>` is assigned to the function and the variable `<lt_on_sales>` is bound by this function call.

Related Information

CALL

5.1.5.3 Call with Named Parameters

You can call a procedure passing named parameters by using the token =>.

For example:

```
CALL myproc (i => 2)
```

When you use named parameters, you can ignore the order of the parameters in the procedure signature. Run the following commands and you can try some of the examples below.

```
 create type mytab_t as table (i int);
create table mytab (i int);
insert into mytab values (0);
insert into mytab values (1);
insert into mytab values (2);
insert into mytab values (3);
insert into mytab values (4);
insert into mytab values (5);
create procedure myproc (in intab mytab_t,in i int, out outtab mytab_t) as
begin
    outtab = select i from :intab where i > :i;
end;
```

Now you can use the following `CALL` possibilities:

```
call myproc(intab=>mytab, i=>2, outtab =>?);
```

Or

```
 call myproc( i=>2, intab=>mytab, outtab =>?)
```

Both call formats produce the same result.

5.1.6 Procedure Parameters

Parameter Modes

The following table lists the parameters you can use when defining your procedures.

Table 3: Parameter modes

Mode	Description
IN	An input parameter
OUT	An output parameter
INOUT	Specifies a parameter that passes in and returns data to and from the procedure Note This is only supported for scalar values.

Supported Parameter Types

Both scalar and table parameter types are supported. For more information on datatypes, see **Datatype Extension**

Related Information

Data Type Extension [page 15]

5.1.6.1 Value Binding during Call

Scalar Parameters

Consider the following procedure:

```
CREATE PROCEDURE test_scalar (IN i INT, IN a VARCHAR)
AS
BEGIN
SELECT i AS "I", a AS "A" FROM DUMMY;
END;
```

You can pass parameters using scalar value binding:

```
CALL test_scalar (1, 'ABC');
```

You can also use expression binding.

```
CALL test_scalar (1+1, upper('abc'))
```

Table Parameters

Consider the following procedure:

```
CREATE TYPE tab_type AS TABLE (I INT, A VARCHAR);
CREATE TABLE tab1  (I INT, A VARCHAR);
CREATE PROCEDURE test_table (IN tab tab_type)
AS
BEGIN
SELECT * FROM :tab;
END;
```

You can pass tables and views to the parameter of this function.

```
CALL test_table (tab1)
```

> **i Note**
>
> Implicit binding of multiple values is currently **not** supported.

You should always use SQL special identifiers when binding a value to a table variable.

```
CALL test_table ("tab1")
```

> **i Note**
>
> Do **not** use the following syntax:
>
> ```
> CALL test_table ('tab')
> ```

5.1.6.2 Default Values for Parameters

In the signature you can define default values for input parameters by using the `DEFAULT` keyword:

```
IN <param_name>  (<sql_type>|<table_type>|<table_type_definition>) DEFAULT
(<value>|<table_name>)
```

The usage of the default value will be illustrated in the next example. Therefore the following tables are needed:

```
CREATE COLUMN TABLE NAMES(Firstname NVARCHAR(20), LastName NVARCHAR(20));
INSERT INTO NAMES VALUES('JOHN', 'DOE');
CREATE COLUMN TABLE MYNAMES(Firstname NVARCHAR(20), LastName NVARCHAR(20));
INSERT INTO MYNAMES VALUES('ALICE', 'DOE');
```

The procedure in the example generates a `FULLNAME` by the given input table and delimiter. Whereby default values are used for both input parameters:

```
CREATE PROCEDURE FULLNAME(
IN INTAB TABLE(FirstName NVARCHAR (20), LastName NVARCHAR (20)) DEFAULT NAMES,
IN delimiter VARCHAR(10) DEFAULT ', ',
OUT outtab TABLE(fullname NVarchar(50))
)
AS
BEGIN
    outtab = SELECT lastname||:delimiter|| firstname AS FULLNAME FROM :intab;
```

```
END;
```

For the tabular input parameter `INTAB` the default table `NAMES` is defined and for the scalar input parameter `DELIMITER` the ',' is defined as default. To use the default values in the signature, you need to pass in parameters using Named Parameters. That means to call the procedure `FULLNAME` and using the default value would be done as follows:

```
CALL FULLNAME (outtab=>?);
```

The result of that call is:

```
FULLNAME
--------
DOE,JOHN
```

Now we want to pass a different table, i.e. `MYNAMES` but still want to use the default delimiter value, the call looks then as follows:

```
CALL FULLNAME(INTAB=> MYNAMES, outtab => ?)
```

And the result shows that now the table `MYNAMES` was used:

```
FULLNAME
--------
DOE,ALICE
```

> Note
>
> Please note that default values are not supported for output parameters.

Related Information

Call with Named Parameters [page 33]

5.1.6.3 DEFAULT EMPTY for Tabular Parameters

For a tabular `IN` and `OUT` parameter the `EMPTY` keyword can be used to define an empty input table as a default:

```
(IN|OUT) <param_name> (<table_type>|<table_type_definition>) DEFAULT EMPTY
```

Although the general default value handling is supported for input parameters only, the `DEFAULT EMPTY` is supported for both tabular `IN` and `OUT` parameters.

In the following example use the `DEFAULT EMPTY` for the tabular output parameter to be able to declare a procedure with an empty body.

```
CREATE PROCEDURE PROC_EMPTY (OUT OUTTAB TABLE(I INT) DEFAULT EMPTY)
AS
BEGIN

END;
```

Creating the procedure without `DEFAULT EMPTY` causes an error indicating that `OUTTAB` is not assigned. The `PROC_EMPTY` procedure can be called as usual and it returns an empty result set:

```
call PROC_EMPTY (?);
```

The following example illustrates the use of a tabular input parameter.

```
CREATE PROCEDURE CHECKINPUT (IN intab TABLE(I INT ) DEFAULT EMPTY,
                             OUT result NVARCHAR(20)
                             )
AS
BEGIN
    IF  IS_EMPTY(:intab)  THEN
        result = 'Input is empty';
    ELSE
        result = 'Input is not empty';
    END IF;
END;
```

An example of calling the procedure without passing an input table follows.

```
call CHECKINPUT(result=>?)
```

This leads to the following result:

```
OUT(1)
-----------------
'Input is empty'
```

For Functions only tabular input parameter supports the EMPTY keyword :

```
CREATE FUNCTION CHECK_INPUT_FUNC (IN intab TABLE (I INT) DEFAULT EMPTY)
RETURNS TABLE(i INT)
AS
BEGIN
    IF  IS_EMPTY(:intab)  THEN
        ...
    ELSE
        ...
    END IF;
    ...
    RETURN :result;
END;
```

An example of calling the funtion without passing an input table looks as follows:

```
SELECT * FROM CHECK_INPUT_FUNC();
```

5.1.7 Procedure Metadata

When a procedure is created, information about the procedure can be found in the database catalog. You can use this information for debugging purposes.

The procedures observable in the system views vary according to the privileges that a user has been granted. The following visibility rules apply:

- **CATALOG READ** or **DATA ADMIN** – All procedures in the system can be viewed.
- **SCHEMA OWNER**, or **EXECUTE** – Only specific procedures where the user is the owner, or they have execute privileges, will be shown.

Procedures can be exported and imported as are tables, see the SQL Reference documentation for details. For more information see Data Import Export Statements in the SAP HANA SQL and System Views Referenece.

The system views for procedures are summarized below:

Related Information

SAP HANA SQL and System Views Reference

5.1.7.1 SYS.PROCEDURES

Available stored procedures

Structure

Table 4:

Column name	Data type	Description
SCHEMA_NAME	NVARCHAR(256)	Schema name of the stored procedure
PROCEDURE_NAME	NVARCHAR(256)	Name of the stored procedure
PROCEDURE_OID	BIGINT	Object ID of the stored procedure
SQL_SECURITY	VARCHAR(7)	SQL security setting of the stored procedure: 'DEFINER' / 'INVOKER'
DEFAULT_SCHEMA_NAME	NVARCHAR(256)	Schema name of the unqualified objects in the procedure
INPUT_PARAMETER_COUNT	INTEGER	Input type parameter count

Column name	Data type	Description
OUTPUT_PARAMETER_COUNT	INTEGER	Output type parameter count
INOUT_PARAMETER_COUNT	INTEGER	In-out type parameter count
RESULT_SET_COUNT	INTEGER	Result set count
IS_UNICODE	VARCHAR(5)	Specifies whether the stored procedure contains Unicode or not: 'TRUE'/ 'FALSE'
DEFINITION	NCLOB	Query string of the stored procedure
PROCEDURE_TYPE	VARCHAR(10)	Type of the stored procedure
READ_ONLY	VARCHAR(5)	Specifies whether the procedure is read-only or not: 'TRUE'/ 'FALSE'
IS_VALID	VARCHAR(5)	Specifies whether the procedure is valid or not. This becomes 'FALSE' when its base objects are changed or dropped: 'TRUE'/ 'FALSE'
IS_HEADER_ONLY	VARCHAR(5)	Specifies whether the procedure is header-only procedure or not: 'TRUE'/'FALSE'
HAS_TRANSACTION_CON-TROL_STATEMENTS	VARCHAR(5)	Specifies whether the procedure has transaction control statements or not:'TRUE'/'FALSE'
OWNER_NAME	NAVARCHAR(256)	Name of the owner of the procedure

5.1.7.2 SYS. PROCEDURE_PARAMETERS

Parameters of stored procedures

Structure

Table 5:

Column name	Data type	Description
SCHEMA_NAME	NVARCHAR(256)	Schema name of the stored procedure
PROCEDURE_NAME	NVARCHAR(256)	Name of the stored procedure

Column name	Data type	Description
PROCEDURE_OID	BIGINT	Object ID of the stored procedure
PARAMETER_NAME	NVARCHAR(256)	Parameter name
DATA_TYPE_ID	SMALLINT	Data type ID
DATA_TYPE_NAME	VARCHAR(16)	Data type name
LENGTH	INTEGER	Parameter length
SCALE	INTEGER	Scale of the parameter
POSITION	INTEGER	Ordinal position of the parameter
TABLE_TYPE_SCHEMA	NVARCHAR(256)	Schema name of table type if DATA_TYPE_NAME is TABLE_TYPE
TABLE_TYPE_NAME	NVARCHAR(256)	Name of table type if DATA_TYPE_NAME is TABLE_TYPE
IS_INPLACE_TYPE	VARCHER(5)	Specifies whether the tabular parameter type is an inplace table type: 'TRUE'/'FALSE'
PARAMETER_TYPE	VARCHAR(7)	Parameter mode: 'IN', 'OUT', 'INOUT'
HAS_DEFAULT_VALUE	VARCHAR(5)	Specifies whether the parameter has a default value or not: 'TRUE', 'FALSE'
IS_NULLABLE	VARCHAR(5)	Specifies whether the parameter accepts a null value: 'TRUE', 'FALSE'

5.1.7.3 SYS.OBJECT_DEPENDENCIES

Dependencies between objects, for example, views that refer to a specific table

Structure

Table 6:

Column name	Data type	Description
BASE_SCHEMA_NAME	NVARCHAR(256)	Schema name of the base object

Column name	Data type	Description
BASE_OBJECT_NAME	NVARCHAR(256)	Object name of the base object
BASE_OBJECT_TYPE	VARCHAR(32)	Type of the base object
DEPENDENT_SCHEMA_NAME	NVARCHAR(256)	Schema name of the dependent object
DEPENDENT_OBJECT_NAME	NVARCHAR(256)	Object name of the dependent object
DEPENDENT_OBJECT_TYPE	VARCHAR(32)	Type of the base dependent
DEPENDENCY_TYPE	INTEGER	Type of dependency between base and dependent object. Possible values are: • 0: NORMAL (default) • 1: EXTERNAL_DIRECT (direct dependency between dependent object and base object) • 2: EXTERNAL_INDIRECT (indirect dependency between dependent object und base object) • 5: REFERENTIAL_DIRECT (foreign key dependency between tables)

5.1.7.3.1 Object Dependencies View Examples

This section explores the ways in which you can query the OBJECT_DEPENDENCIES system view.

You create the following database objects and procedures.

```
CREATE SCHEMA deps;
CREATE TYPE mytab_t AS TABLE (id int, key_val int, val int);
CREATE TABLE mytab1 (id INT PRIMARY KEY, key_val int, val INT);
CREATE TABLE mytab2 (id INT PRIMARY key, key_val int, val INT);
CREATE PROCEDURE deps.get_tables(OUT outtab1 mytab_t, OUT outtab2 mytab_t)
LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
    outtab1 = SELECT * FROM mytab1;
    outtab2 = SELECT * FROM mytab2;
END;
CREATE PROCEDURE deps.my_proc (IN val INT, OUT outtab mytab_t) LANGUAGE
SQLSCRIPT READS SQL DATA
AS
BEGIN
    CALL deps.get_tables(tab1, tab2);
    IF :val > 1 THEN
        outtab = SELECT * FROM :tab1;
    ELSE
        outtab = SELECT * FROM :tab2;
    END IF;
END;
```

Object dependency examination

Find all the (direct and indirect) base objects of the DEPS.GET_TABLES procedure using the following statement.

```
SELECT * FROM OBJECT_DEPENDENCIES WHERE dependent_object_name = 'GET_TABLES' and
dependent_schema_name = 'DEPS';
```

The result obtained is as follows:

Table 7:

BASE_SCHEMA_NAME	BASE_OBJECT_NAME	BASE_OBJECT_TYPE	DEPENDENT_SCHEMA_NAME	DEPENDENT_OBJECT_NAME	DEPENDENT_OBJECT_TYPE	DEPENDENCY_TYPE
SYSTEM	MYTAB_T	TABLE	DEPS	GET_TABLES	PROCEDURE	1
SYSTEM	MYTAB1	TABLE	DEPS	GET_TABLES	PROCEDURE	2
SYSTEM	MYTAB2	TABLE	DEPS	GET_TABLES	PROCEDURE	2
DEPS	GET_TABLES	PROCEDURE	DEPS	GET_TABLES	PROCEDURE	1

Look at the *DEPENDENCY_TYPE* column in more detail. You obtained the results in the table above using a select on all the base objects of the procedure; the objects shown include both persistent and transient objects. You can distinguish between these object dependency types using the *DEPENDENCY_TYPE* column, as follows:

1. EXTERNAL_DIRECT: base object is directly used in the dependent procedure.
2. EXTERNAL_INDIRECT: base object is not directly used in the dependent procedure.

To obtain only the base objects that are used in `DEPS.MY_PROC`, use the following statement.

```
SELECT * FROM OBJECT_DEPENDENCIES WHERE dependent_object_name = 'MY_PROC' and
dependent_schema_name = 'DEPS' and dependency_type = 1;
```

The result obtained is as follows:

Table 8:

BASE_SCHEMA_NAME	BASE_OBJECT_NAME	BASE_OBJECT_TYPE	DEPENDENT_SCHEMA_NAME	DEPENDENT_OBJECT_NAME	DEPENDENT_OBJECT_TYPE	DEPENDENCY_TYPE
SYSTEM	MYTAB_T	TABLE	DEPS	MY_PROC	PROCEDURE	1
DEPS	GET_TABLES	PROCEDURE	DEPS	MY_PROC	PROCEDURE	1

Finally, to find all the dependent objects that are using DEPS.MY_PROC, use the following statement.

```
SELECT * FROM OBJECT_DEPENDENCIES WHERE base_object_name = 'GET_TABLES' and
base_schema_name = 'DEPS' ;
```

The result obtained is as follows:

Table 9:

BASE_SCHEMA_NAME	BASE_OBJECT_NAME	BASE_OBJECT_TYPE	DEPENDENT_SCHEMA_NAME	DEPENDENT_OBJECT_NAME	DEPENDENT_OBJECT_TYPE	DEPENDENCY_TYPE

DEPS	GET_TABLES	PROCEDURE	DEPS	MY_PROC	PROCEDURE	1

5.1.7.4 PROCEDURE_PARAMETER_COLUMNS

PROCEDURE_PARAMETER_COLUMNS provides information about the columns used in table types which appear as procedure parameters. The information is provided for all table types in use, in-place types and externally defined types.

Table 10:

Column name	Data type	Description
SCHEMA_NAME	NVARCHAR(256)	Schema name of the procedure
PROCEDURE_NAME	NVARCHAR(256)	Name of the procedure
PROCEDURE_OID	BIGINT	Object ID of the procedure
PARAMETER_NAME	NVARCHAR(256)	Parameter name
PARAMETER_POSITION	INTEGER	Ordinal position of the parameter
COLUMN_NAME	NVARCHAR(256)	Name of the column of the parameter type
POSITION	INTEGER	Ordinal position of the column in a record
DATA_TYPE_NAME	VARCHAR(16)	SQL data type name of the column
LENGTH	INTEGER	Number of chars for char types, number of max digits for numeric types; number of chars for datetime types, number of bytes for LOB types
SCALE	INTEGER	Numeric types: the maximum number of digits to the right of the decimal point; time, timestamp: the decimal digits are defined as the number of digits to the right of the decimal point in the second's component of the data
IS_NULLABLE	VARCHAR(5)	Specifies whether the column is allowed to accept null value: 'TRUE'/'FALSE'

5.2 User Defined Function

There are two different kinds of user defined functions (UDF): Table User Defined Functions and Scalar User Defined Functions. They are referred to as Table UDF and Scalar UDF in the following table. They differ by input/output parameters, supported functions in the body, and the way they are consumed in SQL statements.

Table 11:

	Table UDF	**Scalar UDF**
Functions Calling	A table UDF can only be called in the FROM –clause of an SQL statement in the same parameter positions as table names. For example, SELECT * FROM myTableUDF(1)	A scalar UDF can be called in SQL statements in the same parameter positions as table column names. These occur in the SELECT and WHERE clauses of SQL statements. For example, SELECT myScalarUDF(1) AS myColumn FROM DUMMY
Input Parameter	• Primitive SQL type • Table types	• Primitive SQL type
Output	Must return a table whose type is defined in <return_type>.	Must return scalar values specified in <return_parameter_list>
Supported functionality	The function is tagged as read only by default. DDL, DML are not allowed and only other read-only functions can be called	The function is tagged as read only function by default. This type of function does not support any kind of SQL –Statements.

5.2.1 CREATE FUNCTION

This SQL statement creates read-only user-defined functions that are free of side effects. This means that neither DDL, nor DML statements (INSERT, UPDATE, and DELETE) are allowed in the function body. All functions or procedures selected or called from the body of the function must be read-only.

Syntax

```
CREATE FUNCTION <func_name> [(<parameter_clause>)] RETURNS <return_type>
[LANGUAGE <lang>] [SQL SECURITY <mode>][DEFAULT SCHEMA <default_schema_name>
[DETERMINISTIC]]
AS
BEGIN
    <function_body>
END
```

Syntax Elements

```
<func_name > ::= [<schema_name>.]<identifier>
```

The identifier of the function to be created, with optional schema name.

```
<parameter_clause> ::= <parameter> [{,<parameter>}...]
```

The input parameters of the function.

```
<parameter> ::= [IN] <param_name> <param_type>
```

A function parameter with associated data type.

```
<param_name> ::= <identifier>
```

The variable name for a parameter.

```
<param_type> ::= <sql_type> | <table_type> | <table_type_definition>
```

Scalar user-defined functions support only primitive SQL types as input, whereas table user-defined functions also supports table types as input. Currently, the following primitive SQL types are allowed in scalar user-defined functions:

```
<sql_type> ::= DATE | TIME | TIMESTAMP | SECONDDATE | TINYINT | SMALLINT |
INTEGER | BIGINT | DECIMAL | SMALLDECIMAL | REAL | DOUBLE | VARCHAR | NVARCHAR |
VARBINARY | ST_GEOMETRY
```

Table user-defined functions allows a wider range of primitive SQL types:

```
<sql_type> ::= DATE | TIME | TIMESTAMP | SECONDDATE | TINYINT | SMALLINT |
INTEGER | BIGINT | DECIMAL | SMALLDECIMAL | REAL | DOUBLE | VARCHAR | NVARCHAR |
ALPHANUM | VARBINARY | CLOB | NCLOB | BLOB | ST_GEOMETRY
<table_type> ::= <identifier>
```

To look at a table type previously defined with the `CREATE TYPE` command, see CREATE TYPE [page 16].

```
<table_type_defintion>    ::=  TABLE (<column_list_definition>)
<column_list_definition > ::= <column_elem>[{, <column_elem>}...]
<column_elem> ::= <column_name> <data_type>
<column_name> ::= <identifier>
```

A table type implicitly defined within the signature.

```
<return_type> ::= <return_parameter_list> | <return_table_type>
```

Table UDFs must return a table whose type is defined by `<return_table_type>`. And scalar UDF must return scalar values specified in `<return_parameter_list>`.

```
 <return_parameter_list> ::= <return_parameter>[{, <return_parameter>}...]
 <return_parameter>      ::= <parameter_name> <sql_type>
```

The following expression defines the output parameters:

```
<return_table_type> ::= TABLE ( <column_list_definition> )
```

The following expression defines the structure of the returned table data.

```
LANGUAGE <lang>
 <lang> ::= SQLSCRIPT
```

Default: SQLSCRIPT

Defines the programming language used in the function.

> **Note**
>
> Only SQLScript UDFs can be defined.

```
 SQL SECURITY <mode>
 <mode> ::= DEFINER | INVOKER
```

Default: DEFINER (Table UDF) / INVOKER (Scalar UDF)

Specifies the security mode of the function.

```
DEFINER
```

Specifies that the execution of the function is performed with the privileges of the definer of the function.

```
INVOKER
```

Specifies that the execution of the function is performed with the privileges of the invoker of the function.

```
DEFAULT SCHEMA <default_schema_name>
<default_schema_name> ::= <unicode_name>
```

Specifies the schema for unqualified objects in the function body. If nothing is specified, then the `current_schema` of the session is used.

```
 <function_body>        ::= <scalar_function_body>|<table_function_body>
 <scalar_function_body> ::= [DECLARE <func_var>]
                             <proc_assign>
 <table_function_body>  ::= [<func_block_decl_list>]
                            [<func_handler_list>]
                             <func_stmt_list>
                             <func_return_statement>
```

Defines the main body of the table user-defined functions and scalar user-defined functions. Since the function is flagged as read-only, neither DDL, nor DML statements (INSERT, UPDATE, and DELETE), are allowed in the function body. A scalar UDF does not support table operations in the function body and variables of type TABLE as input.

> **Note**
>
> Scalar functions can be marked as DETERMINISTIC, if they always return the same result any time they are called with a specific set of input parameters.

For the definition of <proc_assign>, see CREATE PROCEDURE [page 19].

```
 <func_block_decl_list> ::= DECLARE { <func_var>|<func_cursor>|<func_condition> }
 <func_var>             ::= <variable_name_list> [CONSTANT] { <sql_type>|
<array_datatype> } [NOT NULL][<func_default>];
```

```
<array_datatype>        ::= <sql_type> ARRAY [ = <array_constructor> ]
<array_constructor>     ::= ARRAY ( <expression> [{,<expression>}...] )
<func_default>          ::= { DEFAULT | = } <func_expr>
<func_expr>             ::= !!An element of the type specified by <sql_type>
```

Defines one or more local variables with associated scalar type or array type.

An array type has <type> as its element type. An Array has a range from 1 to 2,147,483,647, which is the limitation of underlying structure.

You can assign default values by specifying <expression>s. See Expressions in the SAP HANA SQL and System Views Reference .

```
<func_handler_list> ::= <proc_handler_list>
```

See CREATE PROCEDURE [page 19].

```
<func_stmt_list> ::= <func_stmt>| <func_stmt_list> <func_stmt>
<func_stmt>      ::= <proc_block>
                   | <proc_assign>
                   | <proc_single_assign>
                   | <proc_if>
                   | <proc_while>
                   | <proc_for>
                   | <proc_foreach>
                   | <proc_exit>
                   | <proc_signal>
                   | <proc_resignal>
                   | <proc_open>
                   | <proc_fetch>
                   | <proc_close>
```

For further information of the definitions in <func_stmt>, see CREATE PROCEDURE [page 19]..

```
<func_return_statement> ::= RETURN <function_return_expr>
<func_return_expr>      ::= <table_variable> | <subquery>
```

A table function must contain a return statement.

Example

How to create a table function is shown in the following example:

```
CREATE FUNCTION scale (val INT)
RETURNS TABLE (a INT, b INT) LANGUAGE SQLSCRIPT AS
BEGIN
   RETURN SELECT a, :val * b AS  b FROM mytab;
END;
```

How to call the table function scale is shown in the following example:

```
SELECT * FROM scale(10);
SELECT * FROM scale(10) AS a, scale(10) AS b where a.a =    b.a
```

How to create a scalar function of **`name func_add_mul`** that takes two values of type double and returns two values of type double is shown in the following example:

```
CREATE FUNCTION func_add_mul(x Double, y Double)
```

```
 RETURNS result_add Double, result_mul Double
 LANGUAGE SQLSCRIPT READS SQL DATA AS
 BEGIN
     result_add = :x + :y;
     result_mul = :x * :y;
 END;
```

In a query you can either use the scalar function in the projection list or in the where-clause. In the following example the **func_add_mul** is used in the projection list:

```
 CREATE TABLE TAB (a Double, b Double);
 INSERT INTO TAB VALUES (1.0, 2.0);
 INSERT INTO TAB VALUES (3.0, 4.0);

 SELECT a, b, func_add_mul(a, b).result_add as ADD, func_add_mul(a,
b).result_mul as MUL FROM TAB ORDER BY a;
 A   B    ADD   MUL
 -------------------
 1   2    3      2
 3   4    7     12
```

Besides using the scalar function in a query you can also use a scalar function in scalar assignment, e.g.:

```
 CREATE FUNCTION func_mul(input1 INT)
 RETURNS output1 INT LANGUAGE SQLSCRIPT
 AS
 BEGIN
    output1 = :input1 * :input1;
 END;

 CREATE FUNCTION func_mul_wrapper(input1 INT)
 RETURNS output1 INT LANGUAGE SQLSCRIPT AS
 BEGIN
     output1 = func_mul(:input1);
 END;
 SELECT func_mul_wrapper(2) as RESULT FROM dummy;
 RESULT
 -----------------
 4
```

5.2.2 ALTER FUNCTION

You can use `ALTER FUNCTION` if you want to change the content and properties of a function without dropping the object.

```
ALTER FUNCTION <func_name> RETURNS <return_type> [LANGUAGE <lang>]
[DEFAULT SCHEMA <default_schema_name>]
AS
BEGIN
  <function_body>
END
```

For more information about the parameters please refer to `CREATE FUNCTION`. For instance, with `ALTER FUNCTION` you can change the content of the body itself. Consider the following procedure `GET_FUNCTIONS` that returns all function names on the database.

```
CREATE FUNCTION GET_FUNCTIONS
returns TABLE(schema_name NVARCHAR(256),
              name        NVARCHAR(256))

AS
BEGIN
     return SELECT schema_name    AS schema_name,
               function_name  AS name
         FROM FUNCTIONS;
END;
```

The function `GET_FUNCTIONS` should now be changed to return only valid functions. In order to do so, we will use `ALTER FUNCTION`:

```
ALTER FUNCTION  GET_FUNCTIONS
returns TABLE(schema_name NVARCHAR(256),
              name        NVARCHAR(256))

AS
BEGIN
    return SELECT schema_name    AS schema_name,
                  function_name AS name
          FROM FUNCTIONS
          WHERE IS_VALID = 'TRUE';
END;
```

Besides changing the function body, you can also change the default schema `<default_schema_name>`.

i Note

Please note that the following properties cannot be changed with `ALTER FUNCTION`:

- function owner
- security mode (`INVOKER`, `DEFINER`)
- function type (table function, scalar function, procedure)
- parameter signature (parameter name, parameter type, default value)

If you need to change these properties you have to drop and re-create the procedure again by using `DROP FUNCTION` and `CREATE FUNCTION`.

Note that if the default schema is not explicitly specified, it will be removed.

i Note

Note that you need the ALTER privilege for the object you want to change.

5.2.3 Function Parameters

The following tables list the parameters you can use when defining your user-defined functions.

Table 12:

Function	Parameter
Table user-defined functions	• Can have a list of input parameters and must return a table whose type is defined in <return type> • Input parameters must be explicitly typed and can have any of the primitive SQL type or a table type.
Scalar user-defined functions	• Can have a list of input parameters and must returns scalar values specified in <return parameter list>. • Input parameters must be explicitly typed and can have any primitive SQL type. • Using a table as an input is not allowed.

5.2.4 Function Metadata

When a function is created, information about the function can be found in the database catalog. You can use this information for debugging purposes. The functions observable in the system views vary according to the privileges that a user has been granted. The following visibility rules apply:

- **CATALOG READ** or **DATA ADMIN** – All functions in the system can be viewed.
- **SCHEMA OWNER**, or **EXECUTE** – Only specific functions where the user is the owner, or they have execute privileges, will be shown.

5.2.4.1 SYS.FUNCTIONS

A list of available functions

Structure

Table 13:

Column name	Data type	Description
SCHEMA_NAME	NVARCHAR(256)	Schema name of the function
FUNCTION_NAME	NVARCHAR(256)	Name of the function
FUNCTION_OID	BIGINT	Object ID of the function
SQL_SECURITY	VARCHAR(7)	SQL Security setting of the func- tion:'DEFINER'/'INVOKER'

Column name	Data type	Description
DEFAULT_SCHEMA_NAME	NVARCHAR(256)	Schema name of the unqualified objects in the function
INPUT_PARAMETER_COUNT	INTEGER	Input type parameter count
RETURN_VALUE_COUNT	INTEGER	Return value type parameter count
IS_UNICODE	VARCHAR(5)	Specifies whether the function contains Unicode or not: 'TRUE', 'FALSE'
DEFINITION	NCLOB	Query string of the function
FUNCTION_TYPE	VARCHAR(10)	Type of the function
FUNCTION_USAGE_TYPE	VARCHAR(9)	Usage type of the function:'SCALAR', 'TABLE', 'AGGREGATE','WINDOW'
IS_VALID	VARCHAR(5)	Specifies whether the function is valid or not. This becomes 'FALSE' when its base objects are changed or dropped: 'TRUE', 'FALSE'
IS_HEADER_ONLY	VARCHAR(5)	Specifies whether the function is header-only function or not: 'TRUE'/'FALSE'
OWNER_NAME	NVARCHAR(256)	Name of the owner of the function

5.2.4.2 SYS.FUNCTION_PARAMETERS

A list of parameters of functions

Structure

Table 14:

Column name	Data type	Description
SCHEMA_NAME	NVARCHAR(256)	Schema name of the function
FUNCTION_NAME	NVARCHAR(256)	Name of the function
FUNCTION_OID	BIGINT	Object ID of the function

Column name	Data type	Description
PARAMETER_NAME	NVARCHAR(256)	Parameter name
DATA_TYPE_ID	INTEGER	Data type ID
DATA_TYPE_NAME	VARCHAR(16)	Data type name
LENGTH	INTEGER	Parameter length
SCALE	INTEGER	Scale of the parameter
POSITION	INTEGER	Ordinal position of the parameter
TABLE_TYPE_SCHEMA	NVARCHAR(256)	Schema name of table type if DATA_TYPE_NAME is TABLE_TYPE
TABLE_TYPE_NAME	NVARCHAR(256)	Name of table type if DATA_TYPE_NAME is TABLE_TYPE
IS_INPLACE_TYPE	VARCHAR(5)	Specifies whether the tabular parameter type is an inplace table type: 'TRUE'/'FALSE'
PARAMETER_TYPE	VARCHAR(7)	Parameter mode: IN, OUT, INOUT
HAS_DEFAULT_VALUE	VARCHAR(5)	Specifies whether the parameter has a default value or not: 'TRUE', 'FALSE'
IS_NULLABLE	VARCHAR(5)	Specifies whether the parameter accepts a null value: 'TRUE', 'FALSE'

5.2.4.3 FUNCTION_PARAMETER_COLUMNS

FUNCTION_PARAMETER_COLUMNS provides information about the columns used in table types which appear as function parameters. The information is provided for all table types in use, in-place types and externally defined types.

Table 15:

Column name	Data type	Description
SCHEMA_NAME	NVARCHAR(256)	Schema name of the function
FUNCTION_NAME	NVARCHAR(256)	Name of the function
FUNCTION_OID	BIGINT	Object ID of the function

Column name	Data type	Description
PARAMETER_NAME	NVARCHAR(256)	Parameter name
PARAMETER_POSITION	INTEGER	Ordinal position of the parameter
COLUMN_NAME	NVARCHAR(256)	Name of the column in the table parameter
POSITION	INTEGER	Ordinal position of the column in the table parameter
DATA_TYPE_NAME	VARCHAR(16)	SQL data type name of the column
LENGTH	INTEGER	Number of chars for char types, number of max digits for numeric types; number of chars for datetime types, number of bytes for LOB types
SCALE	INTEGER	Numeric types: the maximum number of digits to the right of the decimal point; time, timestamp: the decimal digits are defined as the number of digits to the right of the decimal point in the second's component of the data
IS_NULLABLE	VARCHAR(5)	Specifies whether the column is allowed to accept null values: 'TRUE'/'FALSE'

5.2.5 Default Values for Parameters

In the signature you can define default values for input parameters by using the `DEFAULT` keyword:

```
IN <param_name>  (<sql_type>|<table_type>|<table_type_definition>) DEFAULT
(<value>|<table_name>)
```

The usage of the default value will be illustrated in the next example. Therefore the following tables are needed:

```
CREATE COLUMN TABLE NAMES(Firstname NVARCHAR(20), LastName NVARCHAR(20));
INSERT INTO NAMES VALUES('JOHN', 'DOE');
CREATE COLUMN TABLE MYNAMES(Firstname NVARCHAR(20), LastName NVARCHAR(20));
INSERT INTO MYNAMES VALUES('ALICE', 'DOE');
```

The function in the example generates a `FULLNAME` by the given input table and delimiter. Whereby default values are used for both input parameters:

```
CREATE FUNCTION FULLNAME(
IN INTAB TABLE(FirstName NVARCHAR (20), LastName NVARCHAR (20)) DEFAULT NAMES,
IN delimiter VARCHAR(10) DEFAULT ', ')
```

```
returns TABLE(fullname NVarchar(50))
AS
BEGIN
    return SELECT lastname||:delimiter|| firstname AS FULLNAME FROM :intab;

END;
```

For the tabular input parameter `INTAB` the default table `NAMES` is defined and for the scalar input parameter `DELIMITER` the ‘,’ is defined as default.

That means to query the function `FULLNAME` and using the default value would be done as follows:

```
SELECT * FROM  FULLNAME();
```

The result of that query is:

```
FULLNAME
--------
DOE,JOHN
```

Now we want to pass a different table, i.e. `MYNAMES` but still want to use the default delimiter value. To do so you need to use using Named Parameters to pass in parameters. The query looks then as follows:

```
SELECT * FROM  FULLNAME(INTAB=> MYNAMES);
```

And the result shows that now the table MYNAMES was used:

```
FULLNAME
--------
DOE,ALICE
```

In a scalar function, default values can also be used, as shown in the next example:

```
CREATE FUNCTION GET_FULLNAME(
                  firstname NVARCHAR(20),
                  lastName  NVARCHAR(20),
                  delimiter NVARCHAR(10) DEFAULT ','
                 )
RETURNS fullname NVARCHAR(50)
AS
BEGIN
  fullname = :lastname||:delimiter|| :firstname;
END;
```

Calling that function by using the default value of the variable delimiter would be the following:

```
SELECT GET__FULLNAME(firstname=>firstname, lastname=>lastname) AS FULLNAME FROM
NAMES;
```

Note

Please note that default values are not supported for output parameters.

Related Information

Call with Named Parameters [page 33]

5.2.6 Deterministic Scalar Functions

Deterministic scalar user-defined functions always return the same result any time they are called with a specific set of input values.

When you use such funtions, it is not necessary to recalculate the result every time - you can refer to the cached result. If you want to make a scalar user-defined function explicitly deterministic, you need to use the optional keyword DETERMINISTIC when you create your function, as demonstrated in the example below. The lifetime of the cache entry is bound to the query execution (for example, SELECT/DML). After the execution of the query, the cache will be destroyed.

Sample Code

```
create function sudf(in a int)
returns ret int deterministic as
begin
  ret = :a;
end;select sudf(a) from tab;
```

Note

In the system view SYS.FUNCTIONS, the column IS_DETERMINISTIC provides information about whether a function is deterministic or not.

Non-Deterministic Functions

The following not-deterministic functions cannot be specified in deterministic scalar user-defined functions. They return an error at function creation time.

- nextval/currval of sequence
- current_time/current_timestamp/current_date
- current_utctime/current_utctimestamp/current_utcdate
- rand/rand_secure
- window functions

5.2.7 DROP FUNCTION

Syntax

```
DROP FUNCTION <func_name> [<drop_option>]
```

Syntax Elements

```
<func_name> ::= [<schema_name>.]<identifier>
```

The name of the function to be dropped, with optional schema name.

```
<drop_option> ::= CASCADE | RESTRICT
```

When <drop_option> is not specified a non-cascaded drop will be performed. This will only drop the specified function, dependent objects of the function will be invalidated but not dropped.

The invalidated objects can be revalidated when an object that has same schema and object name is created.

```
CASCADE
```

Drops the function and dependent objects.

```
RESTRICT
```

Drops the function only when dependent objects do not exist. If this drop option is used and a dependent object exists an error will be thrown.

Description

Drops a function created using CREATE FUNCTION from the database catalog.

Examples

You drop a function called my_func from the database using a non-cascaded drop.

```
DROP FUNCTION my_func;
```

5.3 Anonymous Block

Anonymous block is an executable DML statement which can contain imperative or declarative statements.

All SQLScript statements supported in procedures are also supported in anonymous blocks. Compared to procedures, an anonymous block has no corresponding object created in the metadata catalog.

An anonymous block is defined and executed in a single step by using the following syntax:

```
DO [(<parameter_clause>)]
BEGIN [SEQUENTIAL EXECUTION]
    <body>
END
```

```
<body> ::= !! supports the same feature set as procedure did
```

For more information on <body> see <procedure_body> in CREATE in the SAP HANA SQL and System Views Reference.

With the parameter clause you can define a signature, whereby the value of input and output parameters needs to be bound by using named parameters.

```
<parameter_clause> ::=  <named_parameter> [{,<named_parameter>}...]
<named_parameter>  ::= (IN|OUT) <param_name> <param_type> => <proc_param>
```

Note

INOUT parameters and DEFAULT EMPTY are not supported.

For more information on <proc_param> see CALL [page 30].

The following example illustrates how to call an anonymous block with parameter clause:

```
DO (IN in_var NVARCHAR(24)=> 'A',OUT outtab TABLE (J INT,K INT ) => ?)
BEGIN
    T1 = SELECT I, 10 AS J FROM TAB where z = :in_var;
    T2 = SELECT I, 20 AS K FROM TAB where z = :in_var;
    T3 = SELECT J, K FROM :T1 as a, :T2 as b WHERE a.I = b.I;
    outtab = SELECT * FROM :T3;
END
```

For output parameters only ? is a valid value and cannot be omitted as otherwise query parameter cannot be bound. For the scalar input parameter any scalar expression can be used.

You can also parameterize the scalar parameters if needed. For example for the above given example it would look as follows:

```
DO (IN in_var NVARCHAR(24)=> ?,OUT outtab TABLE (J INT,K INT ) => ?)
BEGIN
    T1 = SELECT I, 10 AS J FROM TAB where z = :in_var;
    T2 = SELECT I, 20 AS K FROM TAB where z = :in_var;
    T3 = SELECT J, K FROM :T1 as a, :T2 as b WHERE a.I = b.I;
    outtab = SELECT * FROM :T3;
END
```

Contrary to a procedure, an anonymous block has no container-specific properties (for example, language, security mode, and so on.) However the body of an anonymous block is similar to the procedure body.

Note

An anonymous block cannot be used in a procedure or function.

In the following you find further examples for anonymous blocks:

Example 1

```
DO
BEGIN
    DECLARE I INTEGER;
    CREATE TABLE TAB1 (I INTEGER);
    FOR I IN 1..10 DO
        INSERT INTO TAB1 VALUES (:I);
    END FOR;
```

```
END;
```

This example contains an anonymous block that creates a table and inserts values into that table.

Example 2

In this example, an anonymous block calls another procedure.

```
DO
BEGIN
    T1 = SELECT * FROM TAB;
    CALL PROC3(:T1, :T2);
    SELECT * FROM :T2;
END
```

Example 3

In this example, an anonymous block uses the exception handler.

```
DO (IN J INTEGER => ?)
BEGIN
    DECLARE I, J INTEGER;
    BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
        IF ::SQL_ERROR_CODE = 288 THEN
            DROP TABLE TAB;
            CREATE TABLE TAB (I INTEGER PRIMARY KEY);
        ELSE
            RESIGNAL;
        END IF;
        CREATE TABLE TAB (I INTEGER PRIMARY KEY);
    END;
    FOR I in 1..3 DO
        INSERT INTO TAB VALUES (:I);
    END FOR;
    IF :J <> 3 THEN
        SIGNAL SQL_ERROR_CODE 10001;
    END IF;
END
```

6 Declarative SQLScript Logic

Each table assignment in a procedure or table user defined function specifies a transformation of some data by means of classical relational operators such as selection, projection. The result of the statement is then bound to a variable which either is used as input by a subsequent statement data transformation or is one of the output variables of the procedure. In order to describe the data flow of a procedure, statements bind new variables that are referenced elsewhere in the body of the procedure.

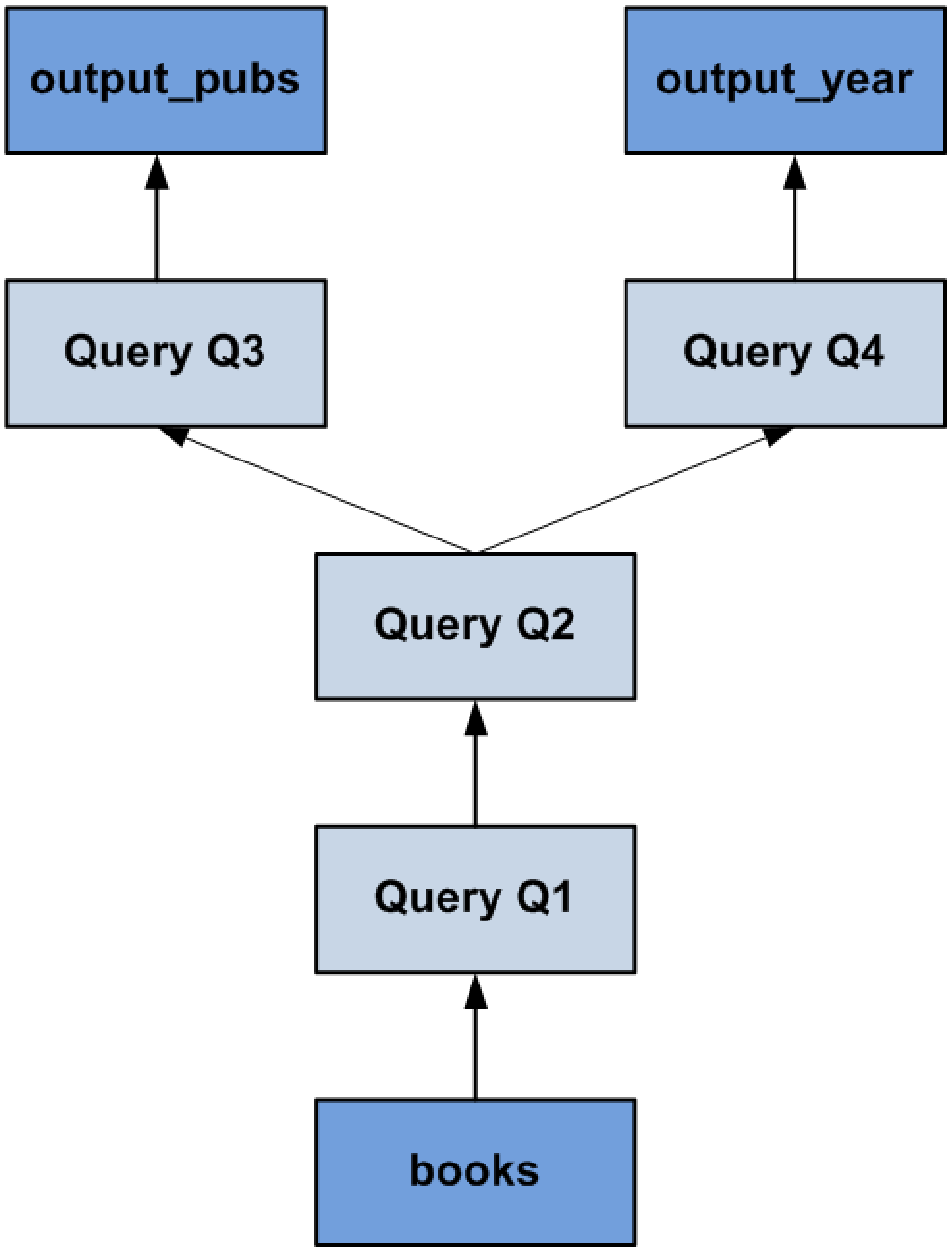
This approach leads to data flows which are free of side effects. The declarative nature to define business logic might require some deeper thought when specifying an algorithm, but it gives the SAP HANA database freedom to optimize the data flow which may result in better performance.

The following example shows a simple procedure implemented in SQLScript. To better illustrate the high-level concept, we have omitted some details.

```
CREATE PROCEDURE getOutput( IN cnt INTEGER, IN currency VARCHAR(3),
                OUT output_pubs tt_publishers, OUT output_year tt_years)
    LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
    big_pub_ids = SELECT publisher AS pid FROM books    -- Query Q1 GROUP BY
publisher HAVING COUNT(isbn) > :cnt;
    big_pub_books = SELECT title, name, publisher,    -- Query Q2 year, price
            FROM :big_pub_ids, publishers, books
            WHERE pub_id = pid AND pub_id = publisher
            AND crcy = :currency;
    output_pubs = SELECT publisher, name,    -- Query Q3
        SUM(price) AS price, COUNT(title) AS cnt FROM :big_pub_books GROUP BY
publisher, name;
    output_year = SELECT year, SUM(price) AS price,    -- Query Q4 COUNT(title)
AS cnt
        FROM :big_pub_books GROUP BY year;
END;
```

This SQLScript example defines a read-only procedure that has 2 scalar input parameters and 2 output parameters of type table. The first line contains an SQL query Q1, that identifies big publishers based on the number of books they have published (using the input parameter `cnt`). Next, detailed information about these publishers along with their corresponding books is determined in query Q2. Finally, this information is aggregated in 2 different ways in queries Q3 (aggregated per publisher) and Q4 (aggregated per year) respectively. The resulting tables constitute the output tables of the function.

A procedure in SQLScript that only uses declarative constructs can be completely translated into an acyclic dataflow graph where each node represents a data transformation. The example above could be represented as the dataflow graph shown in the following image. Similar to SQL queries, the graph is analyzed and optimized before execution. It is also possible to call a procedure from within another procedure. In terms of the dataflow graph, this type of nested procedure call can be seen as a sub-graph that consumes intermediate results and returns its output to the subsequent nodes. For optimization, the sub-graph of the called procedure is merged with the graph of the calling procedure, and the resulting graph is then optimized. The optimization applies similar rules as an SQL optimizer uses for its logical optimization (for example filter pushdown). Then the plan is translated into a physical plan which consists of physical database operations (for example hash joins). The translation into a physical plan involves further optimizations using a cost model as well as heuristics.



6.1 Table Parameter

Syntax

```
[IN|OUT] <param_name> {<table_type>|<table_type_definition>}
<table_type> ::= <identifier>
<table_type_definition> ::= TABLE(<column_list_elements>)
```

Description

Table parameters that are defined in the Signature are either input or output. They must be typed explicitly. This can be done either by using a table type previously defined with the `CREATE TYPE` command or by writing it directly in the signature without any previously defined table type.

Example

```
(IN inputVar TABLE(I INT),OUT outputVar TABLE (I INT, J DOUBLE))
```

Defines the tabular structure directly in the signature.

```
(IN inputVar tableType, OUT outputVar outputTableType)
```

Using previously defined `tableType` and `outputTableType` table types.

The advantage of previously defined table type is that it can be reused by other procedure and functions. The disadvantage is that you must take care of its lifecycle.

The advantage of a table variable structure that you directly define in the signature is that you do not need to take care of its lifecycle. In this case, the disadvantage is that it cannot be reused.

6.2 Local Table Variables

Local table variables are, as the name suggests, variables with a reference to tabular data structure. This data structure originates from an SQL Query.

6.3 Table Variable Type Definition

The type of a table variable in the body of a procedure or a table function is either derived from the SQL Query, or declared explicitly.

If the table variable has derived its type from the SQL query, the SQLScript compiler determines its type from the first assignments of the variable thus providing a lot of flexibility. One disadvantage of this procedure is that it also leads to many type conversions in the background because sometimes the derived table type does not match the typed table parameters in the signature. This can lead to additional conversions, which are unnecessary. Another disadvantage is the unnecessary internal statement compilation to derive the types. To avoid this unnecessary effort, you can declare the type of a table variable explicitly. A declared table variable is always initialized with empty content.

Signature

```
DECLARE <sql_identifier> [{,<sql_identifier> }...] [CONSTANT] {TABLE
(<column_list_definition>)|<table_type>} [ <proc_table_default> ]
<proc_table_default> ::= { DEFAULT | '=' } { <select_statement> | <proc_ce_call>
| <proc_apply_filter> | <unnest_function> }
```

Local table variables are declared by using the DECLARE keyword. For the referenced type, you can either use a previously declared table type, or the type definition TABLE (`<column_list_definition>`). The next example illustrates both variants:

```
DECLARE temp TABLE (n int);
DECLARE temp MY_TABLE_TYPE;
```

You can also directly assign a default value to a table variable by using the DEFAULT keyword or '='. By default all statements are allowed all statements that are also supported for the typical table variable assignment.

```
DECLARE temp MY_TABLE_TYPE = UNNEST (:arr) as (i);
DECLARE temp MY_TABLE_TYPE DEFAULT SELECT * FROM TABLE;
```

The table variable can be also flagged as read-only by using the CONSTANT keyword. The consequence is that you cannot override the variable any more. Note that in case the CONSTANT keyword is used, the table variable should have a default value, it cannot be NULL.

```
DECLARE temp CONSTANT TABLE(I INT) DEFAULT SELECT * FROM TABLE;
```

Description

Local table variables are declared by using the `DECLARE` keyword. A table variable `temp` can be referenced by using `:temp`. For more information, see Referencing Variables [page 64]. The `<sql_identifier>` must be unique among all other scalar variables and table variables in the same code block. However, you can use names that are identical to the name of another variable in a different code block. Additionally, you can reference those identifiers only in their local scope.

```
CREATE PROCEDURE exampleExplicit (OUT outTab TABLE(n int))
LANGUAGE SQLScript READS SQL DATA AS
BEGIN
     DECLARE temp TABLE (n int);
     temp = SELECT 1 as n FROM DUMMY ;
     BEGIN
        DECLARE temp TABLE (n int);
        temp = SELECT 2 as n FROM DUMMY ;
        outTab = Select * from :temp;
    END;
    outTab = Select * from :temp;
END;
call exampleExplicit(?);
```

In each block there are table variables declared with identical names. However, since the last assignment to the output parameter `<outTab>` can only have the reference of variable `<temp>` declared in the same block, the result is the following:

```
 N
----
 1
```

```
CREATE PROCEDURE exampleDerived (OUT outTab TABLE(n int))
LANGUAGE SQLScript READS SQL DATA
AS
BEGIN
    temp = SELECT 1 as n FROM DUMMY ;
    BEGIN
        temp = SELECT 2 as n FROM DUMMY ;
        outTab = Select * from :temp;
    END;
    outTab = Select * from :temp;
END;
call exampleDerived (?);
```

In this code example there is no explicit table variable declaration where done, that means the `<temp>` variable is visible among all blocks. For this reason, the result is the following:

```
 N
----
 2
```

For every assignment of the explicitly declared table variable, the derived column names and types on the right-hand side are checked against the explicitly declared type on the left-hand side.

Another difference, compared to derived types, is that a reference to a table variable without an assignment, returns a warning during the compilation.

```
BEGIN
    DECLARE a TABLE (i DECIMAL(2,1), j INTEGER);
    IF :num = 4
    THEN
        a = SELECT i, j FROM tab;
    END IF;
END;
```

The example above returns a warning because the table variable `<a>` is unassigned if `<:num>` is not 4. This behavior can be controlled by the configuration parameter UNINITIALIZED_TAVLE_VARIABLE_USAGE. Besides issuing a warning, it also offers the follwoing options:

- Error: an error message is issued, a procedure or a function cannot be created
- Silent: no message is issued

The following table shows the differences:

Table 16:

	Derived Type	Explicitly Declared
Create new variable	First SQL query assignment `tmp = select * from table;`	Table variable declaration in a block: `DECLARE tmp TABLE(i int);`
Variable scope	Global scope, regardless of the block where it was first declared	Available in declared block only. Variable hiding is applied.
Unassigned variable check	No warning during the compilation	Warning during compilation if it is possible to refer to the unassigned table variable. The check is performed only if a table variable is used.

Note

The NOT NULL option is not supported (see also the information about scalar variable declaration).

6.4 Binding Table Variables

Table variables are bound using the equality operator. This operator binds the result of a valid `SELECT` statement on the right-hand side to an intermediate variable or an output parameter on the left-hand side. Statements on the right hand side can refer to input parameters or intermediate result variables bound by other statements. Cyclic dependencies that result from the intermediate result assignments or from calling other functions are not allowed, that is to say recursion is not possible.

6.5 Referencing Variables

Bound variables are referenced by their name (for example, `<var>`). In the variable reference the variable name is prefixed by `<:>` such as `<:var>`. The procedure or table function describe a dataflow graph using their statements and the variables that connect the statements. The order in which statements are written in a body can be different from the order in which statements are evaluated. In case a table variable is bound multiple times, the order of these bindings is consistent with the order they appear in the body. Additionally, statements are only evaluated if the variables that are bound by the statement are consumed by another subsequent statement. Consequently, statements whose results are not consumed are removed during optimization.

Example:

```
lt_expensive_books = SELECT title, price, crcy FROM :it_books
                     WHERE price > :minPrice AND crcy = :currency;
```

In this assignment, the variable `<lt_expensive_books>` is bound. The `<:it_books>` variable in the `FROM` clause refers to an `IN` parameter of a table type. It would also be possible to consume variables of type table in the `FROM` clause which were bound by an earlier statement. `<:minPrice>` and `<:currency>` refer to `IN` parameters of a scalar type.

6.6 Column View Parameter Binding

Syntax

```
SELECT * FROM <column_view> ( <named_parameter_list> );
```

Syntax Elements

```
<column_view> ::= <identifier>
```

The name of the column view.

```
<named_parameter_list> ::= <named_parameter> [{,<named_parameter>}…}]
```

A list of parameters to be used with the column view.

```
<named_parameter> ::= <parameter_name> => <expression>
```

Defines the parameter used to refer to the given expression.

```
<parameter_name> ::= {PLACEHOLDER.<identifier> | HINT.<identifier> |
<identifier>}
```

The parameter name definition. PLACEHOLDER is used for place holder parameters and HINT for hint parameters.

Description

Using column view parameter binding it is possible to pass parameters from a procedure/scripted calculation view to a parameterized column view e.g. hierarchy view, graphical calculation view, scripted calculation view.

Examples:

Example 1 - Basic example

In the following example, assume you have the calculation view CALC_VIEW with placeholder parameters "client" and "currency". You want to use this view in a procedure and bind the values of the parameters during the execution of the procedure.

```
CREATE PROCEDURE my_proc_caller (IN in_client INT, IN in_currency INT, OUT
outtab mytab_t) LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
    outtab = SELECT * FROM CALC_VIEW (PLACEHOLDER."$$client$$" => :in_client ,
PLACEHOLDER."$$currency$$" => :in_currency );
END;
```

Example 2 - Using a Hierarchical View

The following example assumes that you have a hierarchical column view "H_PROC" and you want to use this view in a procedure. The procedure should return an extended expression that will be passed via a variable.

```
CREATE PROCEDURE "EXTEND_EXPRESSION"(
    IN in_expr nvarchar(20),
    OUT out_result "TTY_HIER_OUTPUT")
LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
    DECLARE expr VARCHAR(256) = 'leaves(nodes())';
    IF :in_expr <> '' THEN
        expr = 'leaves(' || :in_expr || ')';
    END IF;
    out_result = SELECT query_node, result_node FROM h_proc ("expression"
=> :expr ) as h order by h.result_node;
END;
```

You call this procedure as follows.

```
CALL "EXTEND_EXPRESSION"('',?);
CALL "EXTEND_EXPRESSION"('subtree("B1")',?);
```

6.7 Map Merge

Description

The MAP_MERGE operator is used to apply each row of the input table to the mapper function and unite all intermediate result tables. The purpose of the operator is to replace sequential FOR-loops and union patterns, like in the example below, with a parallel operator.

```
declare cursor cur for select * from tab;
for r as cur do
    call mapper(r.col_a, out_tab)
    ret_tab = select * from :out_tab union selec * from :ret_tab;
```

```
end for;
```

Syntax

```
<table_variable_name> = MAP_MERGE(<table_or_table_variable>,
<mapper_identifier>(<table_or_table_variable>.<column_name> [ {,
<table_or_table_variable>.<column_name>} … ] [, <param_list>])
<table_or_table_variable> ::= <table_variable_name> | <identifier>
<table_variable_name> ::= <identifier>
<mapper_identifier> ::= <identifier>
<column_name> ::= <identifier>
<param_list> ::= <param> [{, <param>} …]
<paramter> = <table_or_table_variable> | <string_literal> | <numeric_literal> |
<identifier>
```

Key Elements

- Mapper Table (<table_or_table_variable>): persistence table or table variable as an input to be iterated by rows
- Mapper Function (<mapper_identifier>): table user-defined function being evaluated per row of input table
- Mapping Argument (<table_or_table_variable>.<column_name>): table column as a scalar argument
- Additional Argument: any other arguments except the mapper argument

Example

In the following example there is an anonymous block that calls a function within a FOR loop and unites the result sets of the table function.

Sample Code

```
CREATE FUNCTION mapper_func (IN a nvarchar(200))
 RETURNS TABLE (col_a nvarchar(200))
 AS
 BEGIN
    ot = SELECT :a AS COL_A from dummy;
    RETURN :ot;
 END;
 DO (OUT ret_tab TABLE(col_a nvarchar(200))=>?)
 BEGIN
    DECLARE i int;
    DECLARE varb nvarchar(200);
    t = SELECT * FROM tab;
    FOR i IN 1 .. record_count(:t)  DO
        varb = :t.col_a[:i];
        CALL mapper(:varb, out_tab);
        ret_tab = SELECT * FROM :out_tab
                  UNION SELECT * FROM :ret_tab;
    END FOR;
 END;
```

This pattern can be replaced by the **MAP_MERGE** operator which executes the **mapper_func** in parallel and assigns the union result to **ret_tab**.

Sample Code

```
DO (OUT ret_tab TABLE(col_a nvarchar(200))=>?)
 BEGIN

     t = SELECT * FROM tab;
     ret_tab = MAP_MERGE(:t, mapper_func(:t.col_a));


 END;
```

6.8 HINTS: NO_INLINE and INLINE

The SQLScript compiler combines statements to optimize code. Hints enable you to block or enforce the inlining of table variables.

Note

Using a `HINT` needs to be considered carefully. In some cases, using a `HINT` could end up being more expensive.

Block Statement-Inlining

The overall optimization guideline in SQLScript states that dependent statements are combined if possible. For example, you have two table variable assignments as follows:

```
tab   = select A, B, C from T where A = 1;
tab2  = select C from :tab where  C = 0;
```

The statements are combined to one statement and executed:

```
select C from (select A,B,C from T where A = 1) where C=0;
```

There can be situations, however, when the combined statements lead to a non-optimal plan and as a result, to less-than-optimal performance of the executed statement. In these situations it can help to block the combination of specific statements. Therefore SAP has introduced a `HINT` called `NO_INLINE`. By placing that `HINT` at the end of select statement, it blocks the combination (or inlining) of that statement into other statements. An example of using this follows:

```
tab   = select A, B, C from T where A = 1 WITH HINT(NO_INLINE);
tab2  = select C from :tab where  C = 0;
```

By adding `WITH HINT (NO_INLINE)` to the table variable `tab`, you can block the combination of that statement and ensure that the two statements are executed separately.

Enforce Statement-Inlining

Using the hint called `INLINE` helps in situations when you want to combine the statement of a nested procedure into the outer procedure.

Currently statements that belong to nested procedure are not combined into the statements of the calling procedures. In the following example, you have two procedures defined.

```
CREATE PROCEDURE procInner (OUT tab2 TABLE(I int))
LANGUAGE SQLSCRIPT READS SQL DATA
AS
BEGIN
     tab2 = SELECT I FROM T;
END;
CREATE PROCEDURE procCaller (OUT table2 TABLE(I int))
LANGUAGE SQLSCRIPT READS SQL DATA
AS
BEGIN
    call procInner (outTable);
    table2 = select I from :outTable where I > 10;
END;
```

By executing the procedure, `ProcCaller`, the two table assignments are executed separately. If you want to have both statements combined, you can do so by using `WITH HINT (INLINE)` at the statement of the output table variable. Using this example, it would be written as follows:

```
CREATE PROCEDURE procInner (OUT tab2 TABLE(I int))
LANGUAGE SQLSCRIPT READS SQL DATA
AS
BEGIN
     tab2 = SELECT I FROM T WITH HINT (INLINE);
END;
```

Now, if the procedure, `ProcCaller`, is executed, then the statement of table variable `tab2` in `ProcInner` is combined into the statement of the variable, `tab`, in the procedure, `ProcCaller`:

```
SELECT I FROM (SELECT I FROM T WITH HINT (INLINE)) where I > 10;
```

7 Imperative SQLScript Logic

In this section we will focus on imperative language constructs such as loops and conditionals. The use of imperative logic splits the logic among several dataflows. For additional information, see Orchestration Logic [page 13] and Declarative SQLScript Logic [page 59].

7.1 Local Scalar Variables

Syntax

```
DECLARE <sql_identifier> [CONSTANT] <type> [NOT NULL] [<proc_default>]
```

Syntax Elements

```
<proc_default> ::= (DEFAULT | '=' ) <value>|<expression>
```

Default value expression assignment.

```
<value>   !!= An element of the type specified by <type>
```

The value to be assigned to the variable.

Description

Local variables are declared using DECLARE keyword and they can optionally be initialized with their declaration. By default scalar variables are initialized with NULL. A scalar variable `var` can be referenced the same way as described above using `:var`.

Tip

If you want to access the value of the variable, then use `:var` in your code. If you want to assign a value to the variable, then use `var` in your code.

Assignment is possible multiple times, overwriting the previous value stored in the scalar variable. Assignment is performed using the = operator.

Recommendation

SAP recommends that you use only the = operator in defining scalar variables. (The := operator is still available, however.)

Example

```
CREATE PROCEDURE proc (OUT z INT) LANGUAGE SQLSCRIPT READS SQL DATA
AS
BEGIN
    DECLARE a int;
    DECLARE b int = 0;
    DECLARE c int DEFAULT 0;

       t = select * from baseTable ;
       select count(*) into a from :t;
       b = :a + 1;
       z = :b + :c;
end;
```

In the example you see the various ways of making declarations and assignments.

Note

Before the SAP HANA SPS 08 release, scalar UDF assignment to the scalar variable was not supported. If you wanted to get the result value from a scalar UDF and consume it in a procedure, the scalar UDF had to be used in a `SELECT` statement, even though this was expensive.

Now you can assign a scalar UDF to a scalar variable with 1 output or more than 1 output, as depicted in the following code examples.

Consuming the result using an SQL statement:

```
DECLARE i INTEGER DEFAULT 0;
SELECT SUDF_ADD(:input1, :input2) into i from dummy;
```

Assign the scalar UDF to the scalar variable:

```
DECLARE i INTEGER DEFAULT 0;
i = SUDF_ADD(:input1, :input2);
```

Assign the scalar UDF with more than 1 output to scalar variables:

```
DECLARE i INTEGER DEFAULT 0;
DECLARE j NVARCHAR(5);
(i,j) = SUDF_EXPR(:input1);
DECLARE a INTEGER DEFAULT 0;
a = SUDF_EXPR(:input1).x;
```

7.2 Global Session Variables

Global session variables can be used in SQLScript to share a scalar value between procedures and functions that are running in the same session. The value of a global session variable is not visible from another session.

To set the value of a global session variable you use the following syntax:

```
SET <key> = <value>;
```

While <key> can only be a constant string, <values> can be any expression, scalar variable or function which returns a value that is convertible to string. Both have maximum length of 5000 characters. The session variable cannot be explicitly typed and is of type string. If <value> is not of type string the value will be implicitly converted to string.

The next examples illustrates how you can set the value of session variable in a procedure:

```
CREATE PROCEDURE CHANGE_SESSION_VAR (IN NEW_VALUE NVARCHAR(50))
AS
BEGIN
    SET 'MY_VAR' = :new_value;
END
```

To retrieve the session variable, the function SESSION_CONTEXT (<key>) can be used.

For more information on SESSION_CONTEXT see SESSION_CONTEXT in the SAP HANA SQL and System Views Reference.

For example the following function retrieves the value of session variable 'MY_VAR'

```
CREATE FUNCTION GET_VALUE ()
RETURNS var NVARCHAR(50)
AS
BEGIN
    var = SESSION_CONTEXT('MY_VAR');
END;
```

Note

SET <key> = <value> cannot not be used in functions and procedure flagged as READ ONLY (scalar and table functions are implicitly READ ONLY)

Note

The maximum number of session variables can be configured with the configuration parameter max_session_variables under the section session (min=1, max=UINT32_MAX) . The default is 1024.

Note

Session variables are null by default and can be reset to null using UNSET <key>.

For more information on UNSET see UNSET in the SAP HANA SQL and System Views Reference.

7.3 Variable Scope Nesting

SQLScript supports local variable declaration in a nested block. Local variables are only visible in the scope of the block in which they are defined. It is also possible to define local variables inside LOOP / WHILE /FOR / IF-ELSE control structures.

Consider the following code:

```
CREATE PROCEDURE nested_block(OUT val INT) LANGUAGE SQLSCRIPT
READS SQL DATA AS
BEGIN
    DECLARE a INT = 1;
    BEGIN
        DECLARE a INT = 2;
        BEGIN
            DECLARE a INT;
            a = 3;
        END;
        val = a;
    END;
END;
```

When you call this procedure the result is:

```
call nested_block(?)
--> OUT:[2]
```

From this result you can see that the inner most nested block value of 3 has not been passed to the `val` variable. Now let's redefine the procedure without the inner most `DECLARE` statement:

```
DROP PROCEDURE nested_block;
CREATE PROCEDURE nested_block(OUT val INT) LANGUAGE SQLSCRIPT
READS SQL DATA AS
BEGIN
    DECLARE a INT = 1;
    BEGIN
        DECLARE a INT = 2;
        BEGIN
            a = 3;
        END;
        val = a;
    END;
END;
```

Now when you call this modified procedure the result is:

```
call nested_block(?)
--> OUT:[3]
```

From this result you can see that the innermost nested block has used the variable declared in the second level nested block.

Local Variables in Control Structures

Conditionals

```
 CREATE PROCEDURE nested_block_if(IN inval INT, OUT val INT) LANGUAGE SQLSCRIPT
READS SQL DATA AS
BEGIN
    DECLARE a INT = 1;
    DECLARE v INT = 0;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        val = :a;
    END;
    v = 1 /(1-:inval);
    IF :a = 1 THEN
        DECLARE a INT = 2;
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
        BEGIN
            val = :a;
        END;
        v = 1 /(2-:inval);
        IF :a = 2 THEN
            DECLARE a INT = 3;
            DECLARE EXIT HANDLER FOR SQLEXCEPTION
            BEGIN
                val = :a;
            END;
            v = 1 / (3-:inval);
        END IF;
        v = 1 / (4-:inval);
    END IF;
    v = 1 / (5-:inval);
END;
call nested_block_if(1, ?)
-->OUT:[1]
call nested_block_if(2, ?)
-->OUT:[2]
call nested_block_if(3, ?)
-->OUT:[3]
call nested_block_if(4, ?)
--> OUT:[2]
call nested_block_if(5, ?)
--> OUT:[1]
```

While Loop

```
 CREATE PROCEDURE nested_block_while(OUT val INT) LANGUAGE SQLSCRIPT READS SQL
DATA AS
BEGIN
    DECLARE v int = 2;
    val = 0;
    WHILE v > 0
    DO
        DECLARE a INT = 0;
        a = :a + 1;
        val = :val + :a;
        v = :v - 1;
    END WHILE;
END;
call nested_block_while(?)
--> OUT:[2]
```

For Loop

```
  CREATE TABLE mytab1(a int);
CREATE TABLE mytab2(a int);
CREATE TABLE mytab3(a int);
INSERT INTO mytab1 VALUES(1);
INSERT INTO mytab2 VALUES(2);
INSERT INTO mytab3 VALUES(3);
CREATE PROCEDURE nested_block_for(IN inval INT, OUT val INT) LANGUAGE SQLSCRIPT
READS SQL DATA AS
BEGIN
    DECLARE a1 int default 0;
    DECLARE a2 int default 0;
    DECLARE a3 int default 0;
    DECLARE v1 int default 1;
    DECLARE v2 int default 1;
    DECLARE v3 int default 1;
    DECLARE CURSOR C FOR SELECT * FROM mytab1;
    FOR R as C DO
        DECLARE CURSOR C FOR SELECT * FROM mytab2;
        a1 = :a1 + R.a;
        FOR R as C DO
            DECLARE CURSOR C FOR SELECT * FROM mytab3;
            a2 = :a2 + R.a;
            FOR R as C DO
                a3 = :a3 + R.a;
            END FOR;
        END FOR;
    END FOR;
    IF inval = 1 THEN
        val = :a1;
    ELSEIF inval = 2 THEN
        val = :a2;
    ELSEIF inval = 3 THEN
        val = :a3;
    END IF;
END;
call nested_block_for(1, ?)
--> OUT:[1]
call nested_block_for(2, ?)
--> OUT:[2]
call nested_block_for(3, ?)
--> OUT:[3]
```

Loop

Note

The example below uses tables and values created in the *For Loop* example above.

```
 CREATE PROCEDURE nested_block_loop(IN inval INT, OUT val INT) LANGUAGE
SQLSCRIPT READS SQL DATA AS
BEGIN
    DECLARE a1 int;
    DECLARE a2 int;
    DECLARE a3 int;
    DECLARE v1 int default 1;
    DECLARE v2 int default 1;
    DECLARE v3 int default 1;
    DECLARE CURSOR C FOR SELECT * FROM mytab1;
    OPEN C;
    FETCH C into a1;
    CLOSE C;
    LOOP
        DECLARE CURSOR C FOR SELECT * FROM mytab2;
        OPEN C;
        FETCH C into a2;
        CLOSE C;
        LOOP
            DECLARE CURSOR C FOR SELECT * FROM mytab3;
            OPEN C;
            FETCH C INTO a3;
            CLOSE C;
            IF :v2 = 1 THEN
                BREAK;
            END IF;
        END LOOP;
        IF :v1 = 1 THEN
```

```
            BREAK;
        END IF;
    END LOOP;
    IF :inval = 1 THEN
        val = :a1;
    ELSEIF :inval = 2 THEN
        val = :a2;
    ELSEIF :inval = 3 THEN
        val = :a3;
    END IF;
END;
call nested_block_loop(1, ?)
--> OUT:[1]
call nested_block_loop(2, ?)
--> OUT:[2]
call nested_block_loop(3, ?)
--> OUT:[3]
```

7.4 Control Structures

7.4.1 Conditionals

Syntax:

```
IF <bool_expr1>
THEN
    <then_stmts1>
[{ELSEIF <bool_expr2>
THEN
    <then_stmts2>}...]
[ELSE
    <else_stmts3>]
END IF
```

Syntax elements:

```
<bool_expr1>   ::= <condition>
<bool_expr2>   ::= <condition>
<condition>    ::= <comparison> | <null_check>
<comparison>   ::= <comp_val> <comparator> <comp_val>
<null_check>   ::= <comp_val> IS [NOT] NULL
```

Tests if `<comp_val>` is `NULL` or `NOT NULL`.

i Note

`NULL` is the default value for all local variables.

See *Example 2* for an example use of this comparison.

```
<comparator>   ::= < | > | = | <= | >= | !=
<comp_val>    ::= <scalar_expression>|<scalar_udf>
<scalar_expression> ::=<scalar_value>[{operator}<scalar_value>…]
<scalar_value> ::= <numeric_literal> | <exact_numeric_literal>|
<unsigned_numeric_literal>
<operator>::=+|-|/|*
```

Specifies the comparison value. This can be based on either scalar literals or scalar variables.

```
<then_stmts1> ::= <proc>
<then_stmts2> ::= <proc_stmts>
<else_stmts3> ::= <proc_stmts>
<proc_stmts> ::= !! SQLScript procedural statements
```

Defines procedural statements to be executed dependent on the preceding conditional expression.

Description:

The `IF` statement consists of a Boolean expression `<bool_expr1>`. If this expression evaluates to true then the statements `<then_stmts1>` in the mandatory `THEN` block are executed. The `IF` statement ends with `END IF`. The remaining parts are optional.

If the Boolean expression `<bool_expr1>` does not evaluate to true the `ELSE`-branch is evaluated. The statements`<else_stmts3>` are executed without further checks. After an else branch no further `ELSE` branch or `ELSEIF` branch is allowed.

Alternatively, when `ELSEIF` is used instead of `ELSE` a further Boolean expression `<bool_expr2>` is evaluated. If it evaluates to true, the statements `<then_stmts2>` are executed. In this manner an arbitrary number of `ELSEIF` clauses can be added.

This statement can be used to simulate the switch-case statement known from many programming languages.

Examples:

Example 1

You use the `IF` statement to implementing the functionality of the SAP HANA database`s `UPSERT` statement.

```
CREATE PROCEDURE upsert_proc (IN v_isbn VARCHAR(20))
LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE found INT = 1;
    SELECT count(*) INTO found FROM books WHERE isbn = :v_isbn;
    IF :found = 0
    THEN
        INSERT INTO books
        VALUES (:v_isbn, 'In-Memory Data Management', 1, 1,
                '2011', 42.75, 'EUR');
    ELSE
        UPDATE books SET price = 42.75 WHERE isbn =:v_isbn;
    END IF;
END;
```

Example 2

You use the `IF` statement to check if variable `:found` is `NULL`.

```
SELECT count(*) INTO found FROM books WHERE isbn = :v_isbn;
IF :found IS NULL THEN
   CALL ins_msg_proc('result of count(*) cannot be NULL');
ELSE
   CALL ins_msg_proc('result of count(*) not NULL - as expected');
```

```
END IF;
```

Example 3

It is also possible to use a scalar UDF in the condition, as shown in the following example.

```
CREATE PROCEDURE proc (in input1 INTEGER, out output1 TYPE1)
AS
BEGIN
    DECLARE i INTEGER DEFAULT :input1;
    IF SUDF(:i) = 1 THEN
         output1 = SELECT value FROM T1;
    ELSEIF SUDF(:i) = 2 THEN
         output1 = SELECT value FROM T2;
    ELSE
         output1 = SELECT value FROM T3;
    END IF;
END;
```

Related Information

ins_msg_proc [page 169]

7.4.2 While Loop

Syntax:

```
WHILE <condition> DO
    <proc_stmts>
END WHILE
```

Syntax elements:

```
<null_check>    ::= <comp_val> IS [NOT] NULL
<comparator>    ::= < | > | = | <= | >= | !=
<comp_val>    ::= <scalar_expression>|<scalar_udf>
<scalar_expression> ::=<scalar_value>[{operator}<scalar_value>…]
<scalar_value> ::= <numeric_literal> | <exact_numeric_literal>|
<unsigned_numeric_literal>
<operator>::=+|-|/|*
```

Defines a Boolean expression which evaluates to true or false.

```
<proc_stmts> ::= !! SQLScript procedural statements
```

Description:

The while loop executes the statements `<proc_stmts>` in the body of the loop as long as the Boolean expression at the beginning `<condition>` of the loop evaluates to true.

Example 1

You use `WHILE` to increment the `:v_index1` and `:v_index2` variables using nested loops.

```
CREATE PROCEDURE procWHILE (OUT V_INDEX2 INTEGER) LANGUAGE SQLSCRIPT
READS SQL DATA
AS
BEGIN
    DECLARE v_index1 INT = 0;
    WHILE :v_index1 < 5 DO
        v_index2 = 0;
        WHILE :v_index2 < 5 DO
            v_index2 = :v_index2 + 1;
        END WHILE;
        v_index1 = :v_index1 + 1;
    END WHILE;
END;
```

Example 2

You can also use scalar UDF for the while condition as follows.

```
CREATE PROCEDURE proc (in input1 INTEGER, out output1 TYPE1)
AS
BEGIN
    DECLARE i INTEGER DEFAULT :input1;
    DECLARE cnt INTEGER DEFAULT 0;
    WHILE SUDF(:i) > 0 DO
        cnt = :cnt + 1;
        i = :i - 1;
    END WHILE;
    output1 = SELECT value FROM T1 where id = :cnt ;
END;
```

⚠ Caution

No specific checks are performed to avoid infinite loops.

7.4.3 For Loop

Syntax:

```
FOR <loop-var> IN [REVERSE] <start_value> .. <end_value> DO
    <proc_stmts>
END FOR
```

Syntax elements:

```
<loop-var> ::= <identifier>
```

Defines the variable that will contain the loop values.

```
REVERSE
```

When defined causes the loop sequence to occur in a descending order.

```
<start_value> ::= <signed_integer>
```

Defines the starting value of the loop.

```
<end_value> ::=  <signed_integer>
```

Defines the end value of the loop.

```
<proc_stmts> ::= !! SQLScript procedural statements
```

Defines the procedural statements that will be looped over.

Description:

The for loop iterates a range of numeric values and binds the current value to a variable `<loop-var>` in ascending order. Iteration starts with the value of `<start_value>` and is incremented by one until the `<loop-var>` is greater than `<end_value>` .

If `<start_value>` is larger than `<end_value>`, `<proc_stmts>` in the loop will not be evaluated.

Example 1

You use nested `FOR` loops to call a procedure that traces the current values of the loop variables appending them to a table.

```
CREATE PROCEDURE proc (out output1 TYPE1) LANGUAGE SQLSCRIPT
READS SQL DATA
AS
BEGIN
    DECLARE pos INTEGER DEFAULT 0;
    DECLARE i INTEGER;
    FOR i IN 1..10 DO
        pos = :pos + 1;
    END FOR;
    output1 = SELECT value FROM T1 where position = :i ;
END;
```

Example 2

You can also use scalar UDF in the `FOR` loop, as shown in the following example.

```
CREATE PROCEDURE proc (out output1 TYPE1)LANGUAGE SQLSCRIPT
READS SQL DATA
AS
BEGIN
    DECLARE pos INTEGER DEFAULT 0;
    DECLARE i INTEGER;
    FOR i IN 1..SUDF_ADD(1, 2) DO
        pos = :pos + 1;
    END FOR;
    output1 = SELECT value FROM T1 where position = :i ;
END;
```

7.4.4 Break and Continue

Syntax:

```
BREAK
CONTINUE
```

Syntax elements:

```
BREAK
```

Specifies that a loop should stop being processed.

```
CONTINUE
```

Specifies that a loop should stop processing the current iteration, and should immediately start processing the next.

Description:

These statements provide internal control functionality for loops.

Example:

You defined the following loop sequence. If the loop value `:x` is less than 3 the iterations will be skipped. If `:x` is 5 then the loop will terminate.

```
CREATE PROCEDURE proc () LANGUAGE SQLSCRIPT
READS SQL DATA
AS
BEGIN
    DECLARE x  integer;
    FOR x IN 0 .. 10 DO
        IF :x < 3 THEN
            CONTINUE;
        END IF;
        IF :x = 5 THEN
            BREAK;
        END IF;
    END FOR;
END;
```

Related Information

ins_msg_proc [page 169]

7.5 Cursors

Cursors are used to fetch single rows from the result set returned by a query. When the cursor is declared it is bound to a query. It is possible to parameterize the cursor query.

7.5.1 Define Cursor

Syntax:

```
CURSOR <cursor_name> [({<param_def>{,<param_def>} ...)]
           FOR <select_stmt>
```

Syntax elements:

```
<cursor_name> ::= <identifier>
```

Specifies the name of the cursor.

```
<param_def> = <param_name> <param_type>
```

Defines an optional SELECT parameter.

```
<param_name> ::= <identifier>
```

Defines the variable name of the parameter.

```
<param_type> ::= DATE | TIME | SECONDDATE | TIMESTAMP | TINYINT
               | SMALLINT | INTEGER | BIGINT | SMALLDECIMAL | DECIMAL
               | REAL | DOUBLE | VARCHAR | NVARCHAR | ALPHANUM
               | VARBINARY | BLOB | CLOB | NCLOB
```

Defines the datatype of the parameter.

```
<select_stmt> !!= SQL SELECT statement.
```

Defines an SQL select statement. See SELECT.

Description:

Cursors can be defined either after the signature of the procedure and before the procedure's body or at the beginning of a block with the `DECLARE` token. The cursor is defined with a name, optionally a list of parameters, and an SQL `SELECT` statement. The cursor provides the functionality to iterate through a query result row-by-row. Updating cursors is not supported.

i Note

Avoid using cursors when it is possible to express the same logic with SQL. You should do this as cursors cannot be optimized the same way SQL can.

Example:

You create a cursor `c_cursor1` to iterate over results from a `SELECT` on the `books` table. The cursor passes one parameter `v_isbn` to the `SELECT` statement.

```
DECLARE CURSOR c_cursor1 (v_isbn VARCHAR(20)) FOR
             SELECT isbn, title, price, crcy FROM books
             WHERE isbn = :v_isbn ORDER BY isbn;
```

Related Information

SAP HANA SQL and System Views Reference

7.5.2 Open Cursor

Syntax:

```
OPEN <cursor_name>[(<argument_list>)]
```

Syntax elements:

```
<cursor_name> ::= <identifier>
```

Specifies the name of the cursor to be opened.

```
<argument_list> ::= <arg>[,{<arg>}...]
```

Specifies one or more arguments to be passed to the select statement of the cursor.

```
<arg> ::= <scalar_value>
```

Specifies a scalar value to be passed to the cursor.

Description:

Evaluates the query bound to a cursor and opens the cursor so that the result can be retrieved. When the cursor definition contains parameters then the actual values for each of these parameters must be provided when the cursor is opened.

This statement prepares the cursor so the results can be fetched for the rows of a query.

Example:

You open the cursor `c_cursor1` and pass a string `'978-3-86894-012-1'` as a parameter.

```
OPEN c_cursor1('978-3-86894-012-1');
```

7.5.3 Close Cursor

Syntax:

```
CLOSE <cursor_name>
```

Syntax elements:

```
<cursor_name> ::= <identifier>
```

Specifies the name of the cursor to be closed.

Description:

Closes a previously opened cursor and releases all associated state and resources. It is important to close all cursors that were previously opened.

Example:

You close the cursor `c_cursor1`.

```
CLOSE c_cursor1;
```

7.5.4 Fetch Query Results of a Cursor

Syntax:

```
FETCH <cursor_name> INTO <variable_list>
```

Syntax elements:

```
<cursor_name> ::= <identifier>
```

Specifies the name of the cursor where the result will be obtained.

```
<variable_list> ::= <var>[,{<var>}...]
```

Specifies the variables where the row result from the cursor will be stored.

```
<var> ::= <identifier>
```

Specifies the identifier of a variable.

Description:

Fetches a single row in the result set of a query and advances the cursor to the next row. This assumes that the cursor was declared and opened before. One can use the cursor attributes to check if the cursor points to a valid row. See **Attributes of a Cursor**

Example:

You fetch a row from the cursor `c_cursor1` and store the results in the variables shown.

```
 FETCH c_cursor1 INTO v_isbn, v_title, v_price, v_crcy;
```

Related Information

Attributes of a Cursor [page 85]

7.5.5 Attributes of a Cursor

A cursor provides a number of methods to examine its current state. For a cursor bound to variable `c_cursor1`, the attributes summarized in the table below are available.

Table 17: Cursor Attributes

Attribute	Description
`c_cursor1::ISCLOSED`	Is true if cursor `c_cursor1` is closed, otherwise false.
`c_cursor1::NOTFOUND`	Is true if the previous fetch operation returned no valid row, false otherwise. Before calling `OPEN` or after calling `CLOSE` on a cursor this will always return true.
`c_cursor1::ROWCOUNT`	Returns the number of rows that the cursor fetched so far. This value is available after the first `FETCH` operation. Before the first fetch operation the number is 0.

Example:

The example below shows a complete procedure using the attributes of the cursor `c_cursor1` to check if fetching a set of results is possible.

```
CREATE PROCEDURE cursor_proc LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE v_isbn    VARCHAR(20);
    DECLARE v_title VARCHAR(20);
    DECLARE v_price DOUBLE;
    DECLARE v_crcy VARCHAR(20);
    DECLARE CURSOR c_cursor1 (v_isbn VARCHAR(20)) FOR
       SELECT isbn, title, price, crcy FROM books
       WHERE isbn = :v_isbn ORDER BY isbn;
    OPEN c_cursor1('978-3-86894-012-1');
    IF c_cursor1::ISCLOSED THEN
       CALL ins_msg_proc('WRONG: cursor not open');
    ELSE
       CALL ins_msg_proc('OK: cursor open');
    END IF;
    FETCH c_cursor1 INTO v_isbn, v_title, v_price, v_crcy;
    IF c_cursor1::NOTFOUND THEN
       CALL ins_msg_proc('WRONG: cursor contains no valid data');
    ELSE
       CALL ins_msg_proc('OK: cursor contains valid data');
    END IF;
    CLOSE c_cursor1;
END
```

Related Information

ins_msg_proc [page 169]

7.5.6 Looping over Result Sets

Syntax:

```
FOR <row_var> AS <cursor_name>[(<argument_list>)] DO
<proc_stmts> | {<row_var>.<column>}
END FOR
```

Syntax elements:

```
<row_var> ::= <identifier>
```

Defines an identifier to contain the row result.

```
<cursor_name> ::= <identifier>
```

Specifies the name of the cursor to be opened.

```
<argument_list> ::= <arg>[,{<arg>}...]
```

Specifies one or more arguments to be passed to the select statement of the cursor.

```
<arg> ::= <scalar_value>
```

Specifies a scalar value to be passed to the cursor.

```
<proc_stmts> ::= !! SQLScript procedural statements
```

Defines the procedural statements that will be looped over.

```
<row_var>.<column> ::= !! Provides attribute access
```

To access the row result attributes in the body of the loop you use the syntax shown.

Description:

Opens a previously declared cursor and iterates over each row in the result set of the query bound to the cursor. For each row in the result set the statements in the body of the procedure are executed. After the last row from the cursor has been processed, the loop is exited and the cursor is closed.

Tip

As this loop method takes care of opening and closing cursors, resource leaks can be avoided. Consequently this loop is preferred to opening and closing a cursor explicitly and using other loop-variants.

Within the loop body, the attributes of the row that the cursor currently iterates over can be accessed like an attribute of the cursor. Assuming `<row_var>` is`a_row` and the iterated data contains a column `test`, then the value of this column can be accessed using `a_row.test`.

Example:

The example below demonstrates using a `FOR`-loop to loop over the results from `c_cursor1` .

```
CREATE PROCEDURE foreach_proc() LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE v_isbn    VARCHAR(20) = '';
    DECLARE CURSOR c_cursor1 (v_isbn VARCHAR(20)) FOR
               SELECT isbn, title, price, crcy FROM books
               ORDER BY isbn;
    FOR cur_row AS c_cursor1(v_isbn)
    DO
        CALL ins_msg_proc('book title is: ' || cur_row.title);
    END FOR;
END;
```

Related Information

ins_msg_proc [page 169]

7.6 Autonomous Transaction

Syntax:

```
<proc_bloc> :: = BEGIN AUTONOMOUS TRANSACTION
        [<proc_decl_list>]
        [<proc_handler_list>]
        [<proc_stmt_list>]
END;
```

Description:

The autonomous transaction is independent from the main procedure. Changes made and committed by an autonomous transaction can be stored in persistency regardless of commit/rollback of the main procedure transaction. The end of the autonomous transaction block has an implicit commit.

```
BEGIN AUTONOMOUS TRANSACTION
    …(some updates) –(1)
    COMMIT;
    …(some updates) –(2)
    ROLLBACK;
    …(some updates) –(3)
END;
```

The examples show how commit and rollback work inside the autonomous transaction block. The first updates (1) are committed, whereby the updates made in step (2) are completely rolled back. And the last updates (3) are committed by the implicit commit at the end of the autonomous block.

```
CREATE PROCEDURE PROC1( IN p INT , OUT outtab TABLE (A INT)) LANGUAGE SQLSCRIPT
AS
BEGIN
          DECLARE errCode INT;
          DECLARE errMsg VARCHAR(5000);
          DECLARE EXIT HANDLER FOR SQLEXCEPTION
          BEGIN AUTONOMOUS TRANSACTION
              errCode= ::SQL_ERROR_CODE;
              errMsg=  ::SQL_ERROR_MESSAGE ;
              INSERT INTO ERR_TABLE (PARAMETER,SQL_ERROR_CODE, SQL_ERROR_MESSAGE)
                      VALUES ( :p, :errCode, :errMsg);
          END;
          outtab = SELECT 1/:p as A FROM DUMMY;    -- DIVIDE BY ZERO Error if p=0
END
```

In the example above, an autonomous transaction is used to keep the error code in the `ERR_TABLE` stored in persistency.

If the exception handler block were not an autonomous transaction, then every insert would be rolled back because they were all made in the main transaction. In this case the result of the `ERR_TABLE` is as shown in the following example.

```
 P |SQL_ERROR_CODE| SQL_ERROR_MESSAGE
--------------------------------------------
0  |      304     | division by zero undefined:  at function /()
```

It is also possible to have nested autonomous transactions.

```
CREATE PROCEDURE P2()
AS BEGIN
    BEGIN AUTONOMOUS TRANSACTION
           INSERT INTO LOG_TABLE VALUES ('MESSAGE');
           BEGIN AUTONOMOUS TRANSACTION
                  ROLLBACK;
           END;
    END;
END;
```

The `LOG_TABLE` table contains `'MESSAGE'`, even though the inner autonomous transaction rolled back.

Supported statements inside the block

- `SELECT`, `INSERT`, `DELETE`, `UPDATE`, `UPSERT`, `REPLACE`
- `IF`, `WHILE`, `FOR`, `BEGIN/END`
- `COMMIT`, `ROLLBACK`, `RESIGNAL`, `SIGNAL`
- Scalar variable assignment

Unsupported statements inside the block

- Calling other procedures
- DDL
- Cursor
- Table assignments

Note

You have to be cautious if you access a table both before and inside an autonomous transaction started in a nested procedure (e.g. TRUNCATE, update the same row), because this can lead to a deadlock situation. One solution to avoid this is to commit the changes before entering the autonomous transaction in the nested procedure.

7.7 COMMIT and ROLLBACK

The `COMMIT` and `ROLLBACK` commands are supported natively in SQLScript.

The `COMMIT` command commits the current transaction and all changes before the `COMMIT` command is written to persistence.

The `ROLLBACK` command rolls back the current transaction and undoes all changes since the last `COMMIT`.

Example 1:

```
CREATE PROCEDURE PROC1() AS
BEGIN
    UPDATE B_TAB SET V = 3 WHERE ID = 1;
    COMMIT;
    UPDATE B_TAB SET V = 4 WHERE ID = 1;
    ROLLBACK;
END;
```

In this example, the `B_TAB` table has one row before the `PROC1` procedure is executed:

Table 18:

V	ID
0	1

After you execute the `PROC1` procedure, the `B_TAB` table is updated as follows:

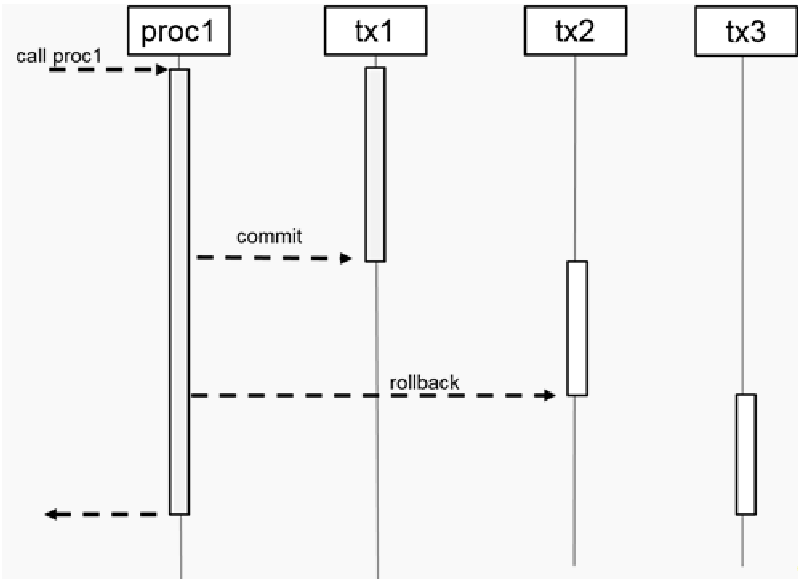
Table 19:

V	ID
3	1

This means only the first update in the procedure affected the `B_TAB` table. The second update does not affect the `B_TAB` table because it was rolled back.

The following graphic provides more detail about the transactional behavior. With the first `COMMIT` command, transaction `tx1` is committed and the update on the `B_TAB` table is written to persistence. As a result of the `COMMIT`, a new transaction starts, `tx2`.

By triggering `ROLLBACK`, all changes done in transaction `tx2` are reverted. In Example 1, the second update is reverted. Additionally after the rollback is performed, a new transaction starts, `tx3`.



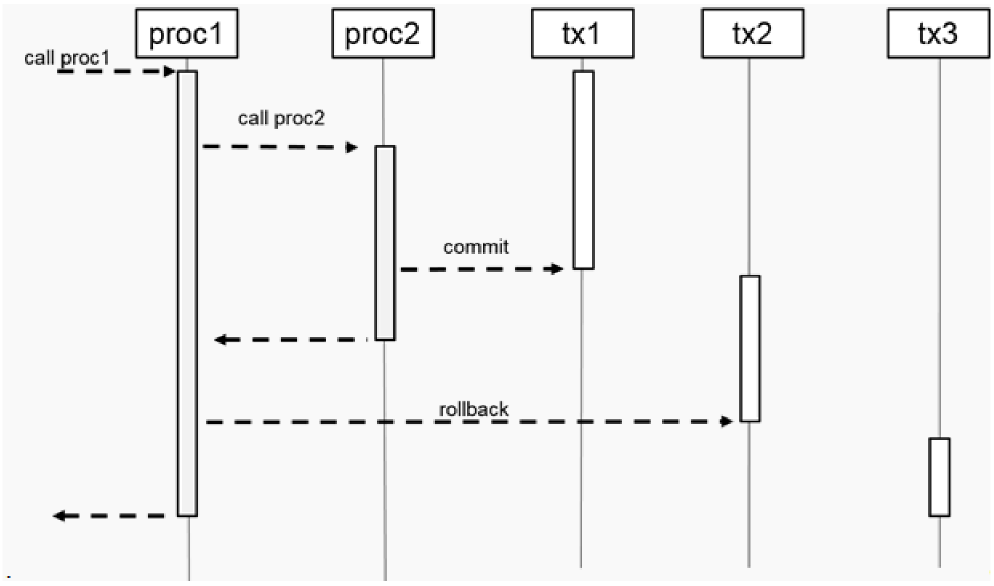
The transaction boundary is not tied to the procedure block. This means that if a nested procedure contains a `COMMIT`/`ROLLBACK`, then all statements of the top-level procedure are affected.

Example 2:

```
CREATE PROCEDURE PROC2() AS
BEGIN
    UPDATE B_TAB SET V = 3 WHERE ID = 1;
    COMMIT;
END;
CREATE PROCEDURE PROC1() AS
BEGIN
    UPDATE A_TAB SET V = 2 WHERE ID = 1;
    CALL PROC2();
    UPDATE A_TAB SET V = 3 WHERE ID = 1;
    ROLLBACK;
END;
```

In Example 2, the `PROC1` procedure calls the `PROC2`procedure. The `COMMIT` in `PROC2` commits all changes done in the `tx1` transaction (see the following graphic). This includes the first update statement in the `PROC1` procedure as well as the update statement in the `PROC2` procedure. With `COMMIT` a new transaction starts implicitly, `tx2`.

Therefore the `ROLLBACK` command in `PROC1` only affects the previous update statement; all other updates were committed with the `tx1` transaction.



> **Note**
>
> - If you used DSQL in the past to execute these commands (for example, `EXEC 'COMMIT'`, `EXEC 'ROLLBACK'`), SAP recommends that you replace all occurrences with the native commands `COMMIT`/`ROLLBACK` because they are more secure.
> - The `COMMIT`/`ROLLBACK` commands are **not** supported in Scalar UDF or in Table UDF.

7.8 Dynamic SQL

Dynamic SQL allows you to construct an SQL statement during the execution time of a procedure. While dynamic SQL allows you to use variables where they might not be supported in SQLScript and also provides more flexibility in creating SQL statements, it does have the disadvantage of an additional cost at runtime:

- Opportunities for optimizations are limited.
- The statement is potentially recompiled every time the statement is executed.
- You cannot use SQLScript variables in the SQL statement.
- You cannot bind the result of a dynamic SQL statement to a SQLScript variable.
- You must be very careful to avoid SQL injection bugs that might harm the integrity or security of the database.

> **Note**
>
> You should avoid dynamic SQL wherever possible as it can have a negative impact on security or performance.

7.8.1 EXEC

Syntax:

```
EXEC '<sql-statement>'
```

Description:

`EXEC` executes the SQL statement passed in a string argument.

Example:

You use dynamic SQL to insert a string into the `message_box` table.

```
v_sql1 = 'Third message from Dynamic SQL';
EXEC 'INSERT INTO message_box VALUES (''' || :v_sql1 || ''')';
```

7.8.2 EXECUTE IMMEDIATE

Syntax:

```
EXECUTE IMMEDIATE '<sql-statement>'
```

Description:

`EXECUTE IMMEDIATE` executes the SQL statement passed in a string argument. The results of queries executed with `EXECUTE IMMEDIATE` are appended to the procedures result iterator.

Example:

You use dynamic SQL to delete the contents of table `tab`, insert a value and finally to retrieve all results in the table.

```
CREATE TABLE tab (i int);
CREATE PROCEDURE proc_dynamic_result2(i int) AS
BEGIN
    EXEC 'DELETE from tab';
    EXEC 'INSERT INTO tab VALUES (' || :i || ')';
    EXECUTE IMMEDIATE 'SELECT * FROM tab ORDER BY i';
 END;
```

7.8.3 APPLY_FILTER

Syntax

```
<variable_name> = APPLY_FILTER(<table_or_table_variable>,
<filter_variable_name>);
```

Syntax Elements

```
<variable_name> ::= <identifier>
```

The variable where the result of the APPLY_FILTER function will be stored.

```
<table_or_table_variable> ::= <table_name> | <table_variable>
```

You can use APPLY_FILTER with persistent tables and table variables.

```
<table_name> :: = <identifier>
```

The name of the table that is to be filtered.

```
<table_variable> ::= :<identifier>
```

The name of the table variable to be filtered.

```
<filter_variable_name> ::= <string_literal>
```

The filter command to be applied.

Note

The following constructs are not supported in the filter string <filter_variable_name>:

- • sub-queries, for example: CALL GET_PROCEDURE_NAME(' PROCEDURE_NAME in (SELECT object_name FROM SYS.OBJECTS), ?);
- • fully-qualified column names, for example: CALL GET_PROCEDURE_NAME(' PROCEDURE.PROCEDURE_NAME = 'DSO', ?);

Description

The APPLY_FILTER function applies a dynamic filter on a table or table variable. Logically it can be considered a partial dynamic sql statement. The advantage of the function is that you can assign it to a table variable and will not block sql – inlining. Despite this all other disadvantages of a full dynamic sql yields also for the APPLY_FILTER.

Examples

Example 1 - Apply a filter on a persistent table

You create the following procedure

```
CREATE PROCEDURE GET_PROCEDURE_NAME (IN filter NVARCHAR(100), OUT procedures
outtype) AS
BEGIN
temp_procedures = APPLY_FILTER(SYS.PROCEDURES,:filter);
procedures = SELECT SCHEMA_NAME, PROCEDURE_NAME FROM :temp_procedures;
END;
```

You call the procedure with two different filter variables.

```
CALL GET_PROCEDURE_NAME(' PROCEDURE_NAME like ''TREX%''', ?);
CALL GET_PROCEDURE_NAME(' SCHEMA_NAME = ''SYS''', ?);
```

Example 2 - Using a table variable

```
CREATE TYPE outtype AS TABLE (SCHEMA_NAME NVARCHAR(256), PROCEDURE_NAME
NVARCHAR(256));
CREATE PROCEDURE GET_PROCEDURE_NAME (IN filter NVARCHAR(100), OUT procedures
outtype)
AS
BEGIN
    temp_procedures = SELECT SCHEMA_NAME, PROCEDURE_NAME FROM SYS.PROCEDURES;
    procedures = APPLY_FILTER(:temp_procedures,:filter);
END;
```

7.9 Exception Handling

Exception handling is a method for handling exception and completion conditions in an SQLScript procedure.

7.9.1 DECLARE EXIT HANDLER

The `DECLARE EXIT HANDLER` parameter allows you to define an exit handler to process exception conditions in your procedure or function.

```
DECLARE EXIT HANDLER FOR <proc_condition_value> {,<proc_condition_value>}...]
<proc_stmt>

<proc_condition_value> ::= SQLEXCEPTION
   | SQL_ERROR_CODE <error_code>
```

```
    | <condition_name>
```

For example the following exit handler catches all `SQLEXCEPTION` and returns the information that an exception was thrown:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'EXCEPTION was thrown' AS ERROR
FROM dummy;
```

For getting the error code and the error message the two system variables `::SQL_ERROR_CODE` and `::SQL_ERROR_MESSAGE` can be used as it is shown in the next example:

```
CREATE PROCEDURE MYPROC (IN in_var INTEGER, OUT outtab TABLE(I INTEGER) ) AS
BEGIN
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
   SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
   outtab = SELECT 1/:in_var as I FROM dummy;
END;
```

By setting `<in_var>` = 0 the result of the procedure execution would be:

::SQL_ERROR_CODE	::SQL_ERROR_MESSAGE
304	Division by zero undefined: the right-hand value of the division cannot be zero at function /() (please check lines: 6)

Besides defining an exit handler for an arbitrary `SQLEXCEPTION` you can also define it for a specific error code number by using the keyword `SQL_ERROR_CODE` followed by an SQL error code number.

For example if only the “division-by-zero” error should be handled the exception handler looks as follows:

```
DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 304
                SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
```

Please note that only the SQL (code strings starting with ERR_SQL_*) and SQLScript (code strings starting with ERR_SQLSCRIPT_*) error codes are supported in the exit handler. You can use the system view `M_ERROR_CODES` to get more information about the error codes.

Instead of using an error code the exit handler can be also defined for a condition.

```
DECLARE EXIT HANDLER FOR MY_COND
                SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
```

How a condition is declared will be explained in section DECLARE CONDITION [page 96].

If you want to do more in the exit handler you have to use a block by using `BEGIN…END`. For instance preparing some additional information and inserting the error into a table:

```
DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 304
BEGIN

   DECLARE procedure_name NVARCHAR(500) =
      ::CURRENT_OBJECT_SCHEMA || '.' ||::CURRENT_OBJECT_NAME;

   DECLARE parameters NVARCHAR(255) =
      'IN_VAR = '||:in_var;
   INSERT INTO LOG_TABLE VALUES ( ::SQL_ERROR_CODE,
      ::SQL_ERROR_MESSAGE,
      :procedure_name,
      :parameters );

END;
tab = SELECT 1/:in_var as I FROM dummy;
```

Note

Please notice in the example above that in case of an unhandled exception the transaction will be rolled back. Thus the new row in the table LOG_TABLE is gone as well. To avoid this you can use an autonomous transaction. You will find more information in Autonomous Transaction [page 87].

7.9.2 DECLARE CONDITION

Declaring a CONDITION variable allows you to name SQL error codes or even to define a user-defined condition.

```
DECLARE <condition name> CONDITION [ FOR SQL_ERROR_CODE <error_code> ];
```

These variables can be used in EXIT HANDLER declaration as well as in SIGNAL and RESIGNAL statements. Whereby in SIGNAL and RESIGNAL only user-defined conditions are allowed.

Using condition variables for SQL error codes makes the procedure/function code more readable. For example instead of using the SQL error code 304, which signals a division by zero error, you can declare a meaningful condition for it:

```
DECLARE division_by_zero CONDITION FOR SQL_ERROR_CODE 304;
```

The corresponding EXIT HANDLER would then look as follows:

```
DECLARE EXIT HANDLER FOR division_by_zero
                SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
```

Besides declaring a condition for an already existing SQL error code, you can also declare a user-defined condition. Either define it with or without a user-defined error code.

Considering you would need a user-defined condition for an invalid procedure input you have to declare it as in the following example:

```
DECLARE invalid_input CONDITION;
```

Optional you can also associate a user-defined error code, e.g. 10000:

```
DECLARE invalid_input CONDITION FOR SQL_ERROR_CODE 10000;
```

Note

Please note the user-defined error codes must be within the range of 10000 to 19999.

How to signal and/or resignal a user-defined condition will be handled in the section SIGNAL and RESIGNAL [page 97].

7.9.3 SIGNAL and RESIGNAL

The `SIGNAL` statement is used to explicitly raise a user-defined exception from within your procedure or function.

```
SIGNAL (<user_defined_condition> | SQL_ERROR_CODE <int_const> )[SET MESSAGE_TEXT
= '<message_string>']
```

The error value returned by the `SIGNAL` statement is either an `SQL_ERROR_CODE` or a user_defined_condition that was previously defined with DECLARE CONDITION [page 96]. The used error code must be within the user-defined range of 10000 to 19999.

E.g. to signal an SQL_ERROR_CODE 10000 is done as follows:

```
SIGNAL SQL_ERROR_CODE 10000;
```

To raise a user-defined condition e.g. invalid_input that we declared in the previous section (see DECLARE CONDITION [page 96]) looks like this:

```
SIGNAL invalid_input;
```

But none of these user-defined exception does have an error message text. That means the value of the system variable `::SQL_ERROR_MESSAGE` is empty. Whereas the value of `::SQL_ERROR_CODE` is 10000.

In both cases you are receiving the following information in case the user-defined exception was thrown:

```
[10000]: user-defined error: "MY_SCHEMA"."MY_PROC": line 3 col 2 (at pos 37):
                [10000] (range 3) user-defined error exception
```

To set a corresponding error message you have to use `SET MESSAGE_TEXT`, e.g.:

```
SIGNAL invalid_input SET MESSAGE_TEXT = 'Invalid input arguments';
```

The result of the user-defined exception looks then as follows:

```
[10000]: user-defined error: "SYSTEM"."MY": line 4 col 2 (at pos 96): [10000]
(range 3) user-defined error exception: Invalid input arguments
```

In the next example the procedure signals an error in case the input argument of start_date is greater than the input argument of end_date:

```
CREATE PROCEDURE GET_CUSTOMERS( IN start_date DATE,
                IN end_date DATE,
                OUT aCust TABLE (first_name NVARCHAR(255),
                last_name NVARCHAR(255))
                )
                AS
                BEGIN
                DECLARE invalid_input CONDITION FOR SQL_ERROR_CODE 10000;

                IF :start_date > :end_date THEN
                SIGNAL invalid_input SET MESSAGE_TEXT =
                'START_DATE = '||:start_date||' > END_DATE =
'
                ||:end_date;
                END IF;

                aCust = SELECT first_name, last_name
                FROM CUSTOMER C
                WHERE    c.bdate >= :start_date
                AND c.bdate <= :end_date;

                END;
```

In case of calling the procedures with invalid input arguments you receive the following error message:

```
user-defined error:  [10000] "MYSCHEMA"."GET_CUSTOMERS": line 9 col 3 (at pos
373): [10000] (range 3) user-defined error exception: START_DATE = 2011-03-03 >
END_DATE = 2010-03-03
```

How to handle the exception and continue with procedure execution will be explained in section DECLARE EXIT HANDLER FOR A NESTED BLOCK.

The `RESIGNAL` statement is used to pass on the exception that is handled in the exit handler.

```
RESIGNAL [<user_defined_condition > | SQL_ERROR_CODE <int_const> ] [SET
MESSAGE_TEXT = '<message_string>']
```

Besides pass on the original exception by simple using `RESIGNAL` you can also change some information before pass it on. Please note that the `RESIGNAL` statement can only be used in the exit handler.

Using `RESIGNAL` statement without changing the related information of an exception is done as follows:

```
CREATE PROCEDURE MYPROC (IN in_var INTEGER, OUT outtab TABLE(I INTEGER) ) AS
                BEGIN
                DECLARE EXIT HANDLER FOR SQLEXCEPTION
                RESIGNAL;

                outtab = SELECT 1/:in_var as I FROM dummy;
                END;
```

In case of `<in_var>` = 0 the raised error would be the original SQL error code and message text.

To change the error message of an SQL error can be done by using `SET MESSAGE _TEXT`:

```
CREATE PROCEDURE MY (IN in_var INTEGER, OUT outtab TABLE(I INTEGER) )
                AS
                BEGIN
                DECLARE EXIT HANDLER FOR SQLEXCEPTION
                RESIGNAL SET MESSAGE_TEXT = 'for the input parameter in_var = '||
                :in_var || ' exception was raised ';

                outtab = SELECT 1/:in_var as I FROM dummy;
                END;
```

The original SQL error message will be now replaced by the new one:

```
[304]: division by zero undefined:  [304] "SYSTEM"."MY": line 4 col 10 (at pos
131): [304] (range 3) division by zero undefined exception: for the input
parameter in_var = 0 exception was raised
```

The original error message you can get via the system variable `::SQL_ERROR_MESSAGE`, e.g. this is useful if you still want to keep the original message, but like to add additional information:

```
CREATE PROCEDURE MY (IN in_var INTEGER, OUT outtab TABLE(I INTEGER) )
                AS
                BEGIN
                DECLARE EXIT HANDLER FOR SQLEXCEPTION
                RESIGNAL SET MESSAGE_TEXT = 'for the input parameter in_var = '||
                :in_var || ' exception was raised '
                || ::SQL_ERROR_MESSAGE;

                outtab = SELECT 1/:in_var as I FROM dummy;
                END;
```

7.9.4 Exception Handling Examples

General exception handling

General exception can be handled with exception handler declared at the beginning of statements which make an explicit or implicit signal exception.

```
  CREATE TABLE MYTAB (I INTEGER PRIMARYKEY);
CREATE PROCEDURE MYPROC AS BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
    INSERT INTO MYTAB VALUES (1);
    INSERT INTO MYTAB VALUES (1);  -- expected unique violation error: 301
    -- will not be reached
END;
CALL MYPROC;
```

Error code exception handling

An exception handler can be declared that catches exceptions with a specific error code numbers.

```
CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
CREATE PROCEDURE MYPROC AS
BEGIN
    DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 301
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
    INSERT INTO MYTAB VALUES (1);
    INSERT INTO MYTAB VALUES (1);  -- expected unique violation error: 301
    -- will not be reached
END;
CALL MYPROC;
```

```
CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
CREATE PROCEDURE MYPROC AS
```

```
BEGIN
    DECLARE myVar INT;
    DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 1299
        BEGIN
               SELECT 0 INTO myVar FROM DUMMY;
                SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
                SELECT :myVar FROM DUMMY;
        END;
    SELECT I INTO myVar FROM MYTAB; --NO_DATA_FOUND exception
    SELECT 'NeverReached_noContinueOnErrorSemantics' FROM DUMMY;
END;
CALL MYPROC;
```

Conditional Exception Handling

Exceptions can be declared using a `CONDITION` variable. The `CONDITION` can optionally be specified with an error code number.

```
CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
CREATE PROCEDURE MYPROC AS
BEGIN
    DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 301;
    DECLARE EXIT HANDLER FOR MYCOND SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
FROM DUMMY;
    INSERT INTO MYTAB VALUES (1);
    INSERT INTO MYTAB VALUES (1);  -- expected unique violation error: 301
    -- will not be reached
END;
CALL MYPROC;
```

Signal an exception

The `SIGNAL` statement can be used to explicitly raise an exception from within your procedures.

Note

The error code used must be within the user-defined range of 10000 to 19999.

```
CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
CREATE PROCEDURE MYPROC AS
BEGIN
    DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 10001;
    DECLARE EXIT HANDLER FOR MYCOND SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
FROM DUMMY;
    INSERT INTO MYTAB VALUES (1);
    SIGNAL MYCOND SET MESSAGE_TEXT = 'my error';
    -- will not be reached
END;
CALL MYPROC;
```

Resignal an exception

The RESIGNAL statement raises an exception on the action statement in exception handler. If error code is not specified, RESIGNAL will throw the caught exception.

```
CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
CREATE PROCEDURE MYPROC AS
BEGIN
    DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 10001;
    DECLARE EXIT HANDLER FOR MYCOND RESIGNAL;
    INSERT INTO MYTAB VALUES (1);
    SIGNAL MYCOND SET MESSAGE_TEXT = 'my error';
    -- will not be reached
END;
CALL MYPROC;
```

Nested block exceptions.

You can declare exception handlers for nested blocks.

```
CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
CREATE PROCEDURE MYPROC AS
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION RESIGNAL SET MESSAGE_TEXT = 'level 1';
    BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION RESIGNAL SET MESSAGE_TEXT = 'level 
2';
        INSERT INTO MYTAB VALUES (1);
        BEGIN
            DECLARE EXIT HANDLER FOR SQLEXCEPTION RESIGNAL SET MESSAGE_TEXT = 
'level 3';
            INSERT INTO MYTAB VALUES (1);  -- expected unique violation error: 
301
            -- will not be reached
        END;
    END;
END;
CALL MYPROC;
```

7.10 ARRAY

An array is an indexed collection of elements of a single data type. In the following section we explore the varying ways to define and use arrays in SQLScript.

7.10.1 DECLARE ARRAY-TYPED VARIABLE

An array-typed variable will be declared by using the keyword ARRAY.

```
DECLARE <variable_name> <sql_type> ARRAY;
```

You can declare an array `<variable_name>` with the element type `<sql_type>`. The following SQL types are supported:

```
<sql_type> ::=
DATE  | TIME| TIMESTAMP | SECONDDATE | TINYINT | SMALLINT | INTEGER | BIGINT |
DECIMAL | SMALLDECIMAL | REAL | DOUBLE | VARCHAR | NVARCHAR | ALPHANUM |
VARBINARY | CLOB | NCLOB |BLOB
```

You can declare the **`arr`** array of type `INTEGER` as follows:

```
DECLARE arr INTEGER ARRAY;
```

Note that only unbounded arrays are supported with a maximum cardinality of 2^31. You cannot define a static size for an array.

You can use the array constructor to directly assign a set of values to the array.

```
DECLARE <variable_name> [{, <variable_name>}...] <sql_type> ARRAY = ARRAY
( <value_expression> [{, <value_expression>}...] );

                <value_expression> !!= An array element of the type specified by
<type>
```

The array constructor returns an array containing elements specified in the list of value expressions. The following example illustrates an array constructor that contains the numbers 1, 2 and 3:

```
DECLARE array_int INTEGER ARRAY = ARRAY(1, 2, 3);
```

Besides using scalar constants you can also use scalar variables or parameters instead, as shown in the next example.

```
CREATE PROCEDURE ARRAYPROC (IN a NVARCHAR(20), IN b NVARCHAR(20))
                AS
                BEGIN
                DECLARE arrayNvarchar NVARCHAR(20) ARRAY;
                arrayNvarchar = ARRAY(:a,:b);
                END;
```

i Note

Note you cannot use `TEXT` or `SHORTTEXT` as the array type.

7.10.2 SET AN ELEMENT OF AN ARRAY

The syntax for setting a value in an element of an array is:

```
<array_variable>'[' <array_index> ']' = <value_expression>
```

The `<array_index>` indicates the index of the element in the array to be modified whereby `<array_index>` can have any value from 1 to 2^31. For example the following statement stores the value 10 in the second element of the array **`id`**:

```
id[2] = 10;
```

Please note that all unset elements of the array are `NULL`. In the given example `id[1]` is then `NULL`.

Instead of using a constant scalar value it is also possible to use a scalar variable of type `INTEGER` as `<array_index>`. In the next example, variable **`I`** of type `INTEGER` is used as an index.

```
DECLARE i INT ;
DECLARE arr NVARCHAR(15) ARRAY ;
for i in 1 ..10 do
    arr [:i] = 'ARRAY_INDEX '|| :i;
end for;
```

SQL Expressions and Scalar User Defined Functions (Scalar UDF) that return a number also can be used as an index. For example, a Scalar UDF that adds two values and returns the result of it

```
CREATE FUNCTION func_add(x INTEGER, y INTEGER)
RETURNS result_add INTEGER
LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
     result_add = :x + :y;
END;
```

is used to determine the index:

```
CREATE procedure PROC (…) AS
BEGIN
     DECLARE VARCHAR_ARRAY VARCHAR ARRAY;
     DECLARE value VARCHAR;
     VARCHAR_ARRAY[func_add(1,0)] = 'i';
END;
```

Note

The array starts with the index 1.

7.10.3 RETURN AN ELEMENT OF AN ARRAY

The value of an array element can be accessed with the index `<array_index>`, where `<array_index>` can be any value from 1 to 2^31. The syntax is:

```
:<array_variable_name> '[' <array_index>']';
```

For example, the following copies the value of the second element of array arr to variable var. Since the array elements are of type NVARCHAR(15) the variable var has to have the same type:

```
DECLARE var NVARCHAR(15);
var = :arr[2];
```

Please note that you have to use ':' before the array variable if you read from the variable.

Instead of assigning the array element to a scalar variable it is possible to directly use the array element in the SQL expression as well. For example, using the value of an array element as an index for another array.

```
DO
BEGIN
    DECLARE arr TINYINT ARRAY = ARRAY(1,2,3);
    DECLARE index_array INTEGER ARRAY = ARRAY(1,2);
    DECLARE value TINYINT;
    arr[:index_array[1]] = :arr[:index_array[2]];
    value = :arr[:index_array[1]];
    select :value from dummy;
END;
```

7.10.4 UNNEST

The UNNEST function converts one or many arrays into a table. The result table includes a row for each element of the specified array. The result of the UNNEST function needs to be assigned to a table variable. The syntax is:

```
<variable_name> = UNNEST(:<array_variable> [ {, :<array_variable>} ...] )[WITH
ORDINALITY] [AS ((<column_name> [ {, <column_name>}… ])) ]
```

For example, the following statements convert the array id of type INTEGER and the array name of type VARCHAR(10) into a table and assign it to the tabular output parameter **rst**:

```
CREATE PROCEDURE ARRAY_UNNEST_SIMPLE(OUT rst TTYPE)
READS SQL DATA
AS
BEGIN
    DECLARE arr_id INTEGER ARRAY = ARRAY (1,2);
DECLARE arr_name VARCHAR(10) ARRAY = ARRAY('name1', 'name2', 'name3');
    rst = UNNEST(:arr_id, :arr_name);
END;
```

For multiple arrays, the number of rows will be equal to the largest cardinality among the cardinalities of the arrays. In the returned table, the cells that are not corresponding to any elements of the arrays are filled with NULL values. The example above would result in the following tabular output of **rst**:

```
:ARR_ID  :ARR_NAME
-------------------
1        name1
2        name2
?        name3
```

Furthermore the returned columns of the table can also be explicitly named be using the AS clause. In the following example, the column names for `:ARR_ID` and `:ARR_NAME` are changed to `ID` and `NAME`.

```
rst = UNNEST(:arr_id, :arr_name) AS (ID, NAME);
```

The result is:

```
ID        NAME
-------------------
1         name1
2         name2
?         name3
```

As an additional option an ordinal column can be specified by using the `WITH ORDINALITY` clause.

The ordinal column will then be appended to the returned table. An alias for the ordinal column needs to be explicitly specified. The next example illustrates the usage. SEQ is used as an alias for the ordinal column:

```
CREATE PROCEDURE ARRAY_UNNEST(OUT rst TABLE(AMOUNT INTEGER, SEQ INTEGER))
LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
    DECLARE amount    INTEGER    ARRAY = ARRAY(10, 20);
    rst = UNNEST(:amount) WITH ORDINALITY AS ( "AMOUNT", "SEQ");
END;
```

The result of calling this procedure is as follows:

```
AMOUNT SEQ
----------------
10     1
20     2
```

Note

The `UNNEST` function cannot be referenced directly in a `FROM` clause of a `SELECT` statement.

7.10.5 ARRAY_AGG

The `ARRAY_AGG` function converts a column of a table variable into an array.

```
<array_variable_name> = ARRAY_AGG ( :<table_variable_name>.<column_name> [ORDER
BY { <expression> [ {, <expression>}… ] [ ASC | DESC ] [ NULLS FIRST | NULLS
LAST ] , ... } ] )
```

In the following example the column **`A`** of table variable **`tab`** is aggregated into array `id`:

```
DECLARE id NVARCHAR(10) ARRAY;
DECLARE tab TABLE (A NVARCHAR(10), B INTEGER);
tab = SELECT A , B FROM tab1;
id  = ARRAY_AGG(:tab.A);
```

The type of the array needs to have the same type as the column.

Optionally the ORDER BY clause can be used to determine the order of the elements in the array. If it is not specified, the array elements are ordered non-deterministic. In the following example all elements of array id are sorted descending by column **B**.

```
id  = ARRAY_AGG(:tab.A ORDER BY B DESC);
```

Additionally it is also possible to define where NULL values should appear in the result set. By default NULL values are returned first for ascending ordering, and last for descending ordering. You can override this behavior using NULLS FIRST or NULLS LAST to explicitly specify NULL value ordering. The next example shows how the default behavior for the descending ordering can be overwritten by using NULLS FIRST:

```
CREATE COLUMN TABLE CTAB (A NVARCHAR(10));
INSERT INTO CTAB VALUES ('A1');
INSERT INTO CTAB VALUES (NULL);
INSERT INTO CTAB VALUES ('A2');
INSERT INTO CTAB VALUES (NULL);
DO
BEGIN
    DECLARE id NVARCHAR(10) ARRAY;
    tab = SELECT A FROM ctab;
    id  = ARRAY_AGG(:tab.A ORDER BY A DESC NULLS FIRST);

    tab2 = UNNEST(:id) AS (A);

    SELECT * FROM :tab2;
END;
```

i Note

ARRAY_AGG function does not support using value expressions instead of table variables.

7.10.6 TRIM_ARRAY

The TRIM_ARRAY function removes elements from the end of an array. TRIM_ARRAY returns a new array with a <trim_quantity> number of elements removed from the end of the array <array_variable>.

```
TRIM_ARRAY"(":<array_variable>, <trim_quantity>")"
<array_variable> ::= <identifier>
<trim_quantity> ::= <unsigned_integer>
```

For example, removing the last 2 elements of array **array_id**:

```
CREATE PROCEDURE ARRAY_TRIM(OUT rst TABLE (ID INTEGER))
LANGUAGE SQLSCRIPT SQL SECURITY INVOKER AS
BEGIN
    DECLARE array_id    Integer ARRAY := ARRAY(1, 2, 3, 4);
    array_id = TRIM_ARRAY(:array_id, 2);
    rst      = UNNEST(:array_id) as ("ID");
END;
```

The result of calling this procedure is as follows:

```
ID
---
```

```
1
2
```

7.10.7 CARDINALITY

The `CARDINALITY` function returns the highest index of a set element in the array `<array_variable>`. It returns N (>= 0) if the index of the N-th element is the largest among the indices.

```
CARDINALITY(:<array_variable>)
```

For example, get the size for array `<array_id>`.

```
CREATE PROCEDURE CARDINALITY_2(OUT n INTEGER) AS
BEGIN
    DECLARE array_id Integer ARRAY;
    n = CARDINALITY(:array_id);
END;
```

The result is n=0 because there is no element in the array. In the next example the cardinality is 20, as the 20th element is set. This implicitly sets the elements 1-19 to `NULL`:

```
CREATE PROCEDURE CARDINALITY_3(OUT n INTEGER) AS
BEGIN
    DECLARE array_id Integer ARRAY;
    array_id[20] = NULL;
    n = CARDINALITY(:array_id);

END;
```

The `CARDINALITY` function can also directly be used everywhere where expressions are supported, for example in a condition:

```
CREATE PROCEDURE CARDINALITY_1(OUT n INTEGER) AS
BEGIN
    DECLARE array_id Integer ARRAY := ARRAY(1, 2, 3);
    If CARDINALITY(:array_id) > 0 THEN
          n = 1 ;
    ELSE
        n = 0;
END IF;
END;
```

7.10.8 CONCATENATE TWO ARRAYS

The `CONCAT` function concatenates two arrays. It returns the new array that contains a concatenation of `<array_variable_left>` and `<array_variable_right>`. Both `||` and the `CONCAT` function can be used for concatenation:

```
:<array_variable_left> “||” :<array_variable_right>
|
CONCAT'(':<array_variable_left> , :<array_variable_right> ')'
```

The next example illustrates the usage of the `CONCAT` function:

```
CREATE PROCEDURE ARRAY_COMPLEX_CONCAT3(OUT OUTTAB TABLE (SEQ INT, ID INT))
LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE id1,id2,id3, id4, id5, card INTEGER ARRAY;
id1[1]  = 0;
    id2[1]  = 1;
    id3  = CONCAT(:id1, :id2);
    id4  = :id1 || :id2;
    rst  = UNNEST(:id3) WITH ORDINALITY AS ("ID", "SEQ");
    id5  = :id4 || ARRAY_AGG(:rst."ID" ORDER BY "SEQ");
    rst1 = UNNEST(:id5 || CONCAT(:id1, :id2) || CONCAT(CONCAT(:id1, :id2),
CONCAT(:id1, :id2))) WITH ORDINALITY AS ("ID", "SEQ");
outtab = SELECT SEQ, ID FROM :rst1 ORDER BY SEQ;
END;
```

7.11 Index-Based Cell Access to Table Variables

The index-based cell access allows you random access (read and write) to each cell of table variable.

```
<table_variable>.<column_name>[<index>]
```

For example, writing to certain cell of a table variable is illustrated in the following example. Here we simply change the value in the second row of column A.

```
create procedure procTCA (
                           IN intab TABLE(A INTEGER, B VARCHAR(20)),
                           OUT outtab TABLE(A INTEGER, B VARCHAR(20))
                          )
AS
BEGIN
    intab.A[2] = 5;
    outtab = select * from :intab;
END;
```

Reading from a certain cell of a table variable is done in similar way. Note that for the read access, the ':' is needed in front of the table variable.

```
create procedure procTCA (
      IN intab TABLE(A INTEGER, B VARCHAR(20)),
      OUT outvar VARCHAR(20)
   )
AS
BEGIN
   outvar  = :intab.B[100];
END;
```

The same rules apply for `<index>` as for the array index. That means that the `<index>` can have any value from 1 to 2^31 and that SQL Expression and Scalar User Defined Functions (Scalar UDF) that return a number also can be used as an index. Instead of using a constant scalar values, it is also possible to use a scalar variable of type INTEGER as `<index>`.

Restrictions:

- Physical tables cannot be accessed
- Not applicable in SQL queries like `SELECT :MY_TABLE_VAR.COL[55] AS A FROM DUMMY`. You need to assign the value to be used to a scalar variable first.

7.12 Emptiness Check for Tables and Table Variables

To determine whether a table or table variable is empty you can use the predicate `IS_EMPTY`:

```
IS_EMPTY( <table_name> | <table_variable> )
```

`IS_EMPTY` takes as an argument a `<table_name>` or a `<table_variable>`. It returns `true` if the table or table variable is empty and `false` otherwise.

You can use `IS_EMPTY` in conditions like in if-statements or while loops. For instance in the next example `IS_EMPTY` is used in an if-statement:

```
CREATE PROCEDURE PROC_IS_EMPTY ( IN tabvar TABLE(ID INTEGER),
                       OUT outtab TABLE(ID INTEGER)
                        )
AS
BEGIN
    IF IS_EMPTY(:tabvar) THEN
          RETURN;
    END IF;
    CALL INTERNAL_LOGIC (:tabvar, outtab);
END;
```

Besides that you can also use it in scalar variable assignments. But note since SQLScript does not support `BOOLEAN` as scalar type, you need to assign the result of the value to a variable of type `INTEGER`, if needed. That means `true` will be converted to 1 and `false` will be converted to 0.

Note

Note that the `IS_EMPTY` cannot be used in SQL queries or in expressions.

7.13 Get Number of Records for Tables and Table Variables

To get the number of records of a table or a table variable, you can use the operator RECORD_COUNT:

```
RECORD_COUNT( <table_name> | <table_variable> )
```

RECORD_COUNT takes as the argument <table_name> or <table_variable> and returns the number of records of type BIGINT.

You can use RECORD_COUNT in all places where expressions are supported such as IF-statements, loops or scalar assignments. In the following example it is used in a loop:

```
CREATE table tab (COL_A int);
INSERT INTO tab VALUES (1);
INSERT INTO tab VALUES (2);
DO (IN inTab TABLE(col_a int) => TAB, OUT v INT => ?)
 BEGIN
    DECLARE i int;
    v = 0;
    FOR i IN 1 .. RECORD_COUNT(:inTab)
    DO
        v = :v + :inTab.col_a[:i];

    END FOR;
END
```

7.14 SQL Injection Prevention Functions

If your SQLScript procedure needs execution of dynamic SQL statements where the parts of it are derived from untrusted input (e.g. user interface), there is a danger of an SQL injection attack. The following functions can be utilized in order to prevent it:

- ESCAPE_SINGLE_QUOTES(string_var) to be used for variables containing a SQL string literal
- ESCAPE_DOUBLE_QUOTES(string_var) to be used for variables containing a delimited SQL identifier
- IS_SQL_INJECTION_SAFE(string_var[, num_tokens]) to be used to check that a variable contains safe simple SQL identifiers (up to num_tokens, default is 1)

Example:

```
create table mytab(myval varchar(20));
insert into mytab values('Val1');
create procedure change_value(
   in tabname varchar(20),
   in field varchar(20),
   in old_val varchar(20),
   in new_val varchar(20)
) as
begin
  declare sqlstr nclob;
   sqlstr := 'UPDATE "' ||:tabname || '" SET ' || field || ' = ''' ||
new_val || ''' WHERE ' || field || ' = ''' || old_val || '''';
   exec(:sqlstr);
end
```

The following values of input parameters can manipulate the dynamic SQL statement in an unintended way:

- tabname: mytab" set myval = ' ' --
- field: myval = ' ' --
- new_val: ' --
- old_val: ' or 1 = 1 --

This cannot happen if you validate and/or process the input values:

```
create procedure change_value(
   in tabname varchar(20),
   in field varchar(20),
   in old_val varchar(20),
   in new_val varchar(20)
) as
begin
   declare sqlstr nclob;
   declare mycond condition for sql_error_code 10001;
   if is_sql_injection_safe(field) <> 1 then
      signal mycond set message_text = 'Invalid field ' || field;
   end if;
   sqlstr := 'UPDATE "' || escape_double_quotes(:tabname) || '" SET ' ||
field || ' = ''' || escape_single_quotes(:new_val) || ''' WHERE ' || field
|| ' = ''' || escape_single_quotes(:old_val) || '''';
exec(:sqlstr);
end
```

Syntax IS_SQL_INJECTION_SAFE

```
IS_SQL_INJECTION_SAFE(<value>[, <max_tokens>])
```

Syntax Elements

```
<value> ::= <string>
```

String to be checked.

```
<max_tokens> ::= <integer>
```

Maximum number of tokens that is allowed to be in `<value>`. The default value is 1.

Description

Checks for possible SQL injection in a parameter which is to be used as a SQL identifier. Returns 1 if no possible SQL injection is found, otherwise 0.

Example

The following code example shows that the function returns 0 if the number of tokens in the argument is different from the expected number of a single token (default value).

```
SELECT IS_SQL_INJECTION_SAFE('tab,le') "safe" FROM DUMMY;

safe
```

```
-------
0
```

The following code example shows that the function returns 1 if the number of tokens in the argument matches the expected number of 3 tokens.

```
SELECT IS_SQL_INJECTION_SAFE('CREATE STRUCTURED PRIVILEGE', 3) "safe" FROM DUMMY;

safe
-------
1
```

Syntax ESCAPE_SINGLE_QUOTES

```
ESCAPE_SINGLE_QUOTES(<value>)
```

Description

Escapes single quotes (apostrophes) in the given string `<value>`, ensuring a valid SQL string literal is used in dynamic SQL statements to prevent SQL injections. Returns the input string with escaped single quotes.

Example

The following code example shows how the function escapes a single quote. The one single quote is escaped with another single quote when passed to the function. The function then escapes the parameter content **`Str'ing`** to **`Str''ing`**, which is returned from the SELECT.

```
SELECT ESCAPE_SINGLE_QUOTES('Str''ing') "string_literal" FROM DUMMY;

string_literal
---------------
Str''ing
```

Syntax ESCAPE_DOUBLE_QUOTES

```
ESCAPE_DOUBLE_QUOTES(<value>)
```

Description

Escapes double quotes in the given string `<value>`, ensuring a valid SQL identifier is used in dynamic SQL statements to prevent SQL injections. Returns the input string with escaped double quotes.

Example

The following code example shows that the function escapes the double quotes.

```
SELECT ESCAPE_DOUBLE_QUOTES('TAB"LE') "table_name" FROM DUMMY;

table_name
--------------
TAB""LE
```

7.15 Explicit Parallel Execution

So far, implicit parallelization has been applied to table variable assignments as well as read-only procedure calls that are independent from each other. DML statements and read-write procedure calls had to be executed sequentially. From now on, it is possible to parallelize the execution of independent DML statements and read-write procedure calls by using parallel execution blocks:

```
BEGIN PARALLEL EXECUTION
   <stmt>
END;
```

For example, in the following procedure several `UPDATE` statements on different tables are parallelized:

```
CREATE COLUMN TABLE CTAB1(A INT);
CREATE COLUMN TABLE CTAB2(A INT);
CREATE COLUMN TABLE CTAB3(A INT);
CREATE COLUMN TABLE CTAB4(A INT);
CREATE COLUMN TABLE CTAB5(A INT);
CREATE PROCEDURE ParallelUpdate AS
BEGIN
    BEGIN PARALLEL EXECUTION
      UPDATE CTAB1 SET A = A + 1;
      UPDATE CTAB2 SET A = A + 1;
      UPDATE CTAB3 SET A = A + 1;
      UPDATE CTAB4 SET A = A + 1;
      UPDATE CTAB5 SET A = A + 1;
    END;
END;
```

Note

Only DML statements on column store tables are supported within the parallel execution block.

In the next example several records from a table variable are inserted into different tables in parallel.

Sample Code

```
CREATE PROCEDURE ParallelInsert (IN intab TABLE (A INT, I INT)) AS
BEGIN
DECLARE tab TABLE(A INT);
tab = SELECT t.A AS A from TAB0 t
LEFT OUTER JOIN :intab s
ON s.A = t.A;
BEGIN PARALLEL EXECUTION
SELECT * FROM :tab s where s.A = 1 INTO CTAB1;
SELECT * FROM :tab s where s.A = 2 INTO CTAB2;
SELECT * FROM :tab s where s.A = 3 INTO CTAB3;
SELECT * FROM :tab s where s.A = 4 INTO CTAB4;
SELECT * FROM :tab s where s.A = 5 INTO CTAB5;
END;
END;
```

You can also parallelize several calls to read-write procedures. In the following example several procedures that performing independent insert operation are executed in parallel.

Sample Code

```
create column table ctab1 (i int);
create column table ctab2 (i int);
create column table ctab3 (i int);

create procedure cproc1 as begin
  insert into ctab1 values (1);
end;


create procedure cproc2 as begin
  insert into ctab2 values (2);
end;


create procedure cproc3 as begin
  insert into ctab3 values (3);
end;


create procedure cproc as begin
  begin parallel execution
    call cproc1 ();
    call cproc2 ();
    call cproc3 ();
  end;
end;

call cproc;
```

Note

Only the following statements are allowed in read-write procedures, which can be called within a parallel block:

- DML
- Imperative logic
- Autonomous transaction

The following restrictions apply:

- Modification of tables with a foreign key or triggers are not allowed
- Updating the same table in different statements is not allowed
- Reading / writing the same table is not allowed
- Calling procedures containing dynamic SQL (for example, EXEC, EXECUTE IMMEDIATE) is not supported in parallel blocks
- Mixing read-only procedure calls and read-write procedure calls in a parallel block is not allowed.

8 Calculation Engine Plan Operators

Recommendation

SAP recommends that you use SQL rather than Calculation Engine Plan Operators with SQLScript.

The execution of Calculation Engine Plan Operators currently is bound to processing within the calculation engine and does not allow a possibility to use alternative execution engines, such as L native execution. As most Calculation Engine Plan Operators are converted internally and treated as SQL operations, the conversion requires multiple layers of optimizations. This can be avoided by direct SQL use. Depending on your system configuration and the version you use, mixing Calculation Engine Plan Operators and SQL can lead to significant performance penalties when compared to to plain SQL implementation.

Table 20: Overview: Mapping between CE_* Operators and SQL

CE Operator	CE Syntax	SQL Equivalent
`CE_COLUMN_TABLE`	`CE_COLUMN_TABLE(<table_name>[,<attributes>])`	`SELECT [<attributes>] FROM <table_name>`

`CE_JOIN_VIEW`	`CE_JOIN_VIEW(<column_view_name>[,<attributes>])`
	
	`out = CE_JOIN_VIEW("PRODUCT_SALES", ["PRODUCT_KEY", "PRODUCT_TEXT", "SALES"]);`
		`SELECT [<attributes>] FROM <column_view_name>`
		
		`out = SELECT product_key, product_text, sales FROM product_sales;`

`CE_OLAP_VIEW`	`CE_OLAP_VIEW (<olap_view_name>[,<attributes>])`
	
	`out = CE_OLAP_VIEW("OLAP_view", ["DIM1", SUM("KF")]);`
		`SELECT [<attributes>] FROM <olap_view_name>`
		
		`out = select dim1, SUM(kf) FROM OLAP_view GROUP BY dim1;`

`CE_CALC_VIEW`	`CE_CALC_VIEW(<calc_view_name>,[<attributes>])`
	
	`out = CE_CALC_VIEW("TESTCECTABLE", ["CID", "CNAME"]);`
		`SELECT [<attributes>] FROM <calc_view_name>`
		
		`out = SELECT cid, cname FROM "TESTCECTABLE";`

CE Operator	CE Syntax	SQL Equivalent

CE_JOIN	CE_JOIN(<left_table>,<right_table>,<join_attributes>[<projection_list>])
	
	ot_pubs_books1 = CE_JOIN (:lt_pubs, :it_books, ["PUBLISHER"]);
		SELECT [<projection_list>] FROM <left_table>,<right_table> WHERE <join_attributes>
		
		ot_pubs_books1 = SELECT P.publisher AS publisher, name, street,post_code, city, country, isbn, title, edition, year, price, crcy FROM :lt_pubs AS P, :it_books AS B WHERE P.publisher = B.publisher;

CE_LEFT_OUTER_JOIN	CE_LEFT_OUTER_JOIN(<left_table>,<right_table>,<join_attributes>[<projection_list>])	SELECT [<projection_list>] FROM <left_table> LEFT OUTER JOIN <right_table> ON <join_attributes>
CE_RIGHT_OUTER_JOIN	CE_RIGHT_OUTER_JOIN(<left_table>,<right_table>,<join_attributes>[<projection_list>])	SELECT [<projection_list>] FROM <left_table> RIGHT OUTER JOIN <right_table> ON <join_attributes>

CE_PROJECTION	CE_PROJECTION(<table_variable>,<projection_list>[,<filter>])
	
	ot_books1 = CE_PROJECTION (:it_books, ["TITLE","PRICE", "CRCY" AS "CURRENCY"], '"PRICE" > 50');
		SELECT <projection_list> FROM <table_variable> where [<filter>]
		
		ot_book2= SELECT title, price, crcy AS currency FROM :it_b ooks WHERE price > 50;

CE_UNION_ALL	CE_UNION_ALL(<table_variable1>,<table_variable2>)
	
	ot_all_books1 = CE_UNION_ALL (:lt_books, :it_audiobooks);
		SELECT * FROM <table_variable1> UNION ALL SELECT * FROM <table_variable2>
		
		ot_all_books2 = SELECT * FROM :lt_books UNION ALL SELECT * FROM :it_audiobooks;

CE_CONVERSION	CE_CONVERSION(<table_variable>,<conversion_params>, [<rename_clause>])	SQL-Function CONVERT_CURRENCY

CE Operator	CE Syntax	SQL Equivalent

`CE_AGGREGATION`	`CE_AGGREGATION(<table_variable>,<aggregate_list> [,<group_columns>])`
	 `ot_books1 = CE_AGGREGATION (:it_books, [COUNT ("PUBLISHER") AS "CNT"], ["YEAR"]);`
		`SELECT <aggregate_list> FROM <table_variable> [GROUP BY <group_columns>]`
		 `ot_books2 = SELECT COUNT (publisher) AS cnt, year FROM :it_books GROUP BY year;`

`CE_CALC`	`CE_CALC('<expr>', <result_type>)`
	 `TEMP = CE_PROJECTION(:table_var, ["ID" AS "KEY", CE_CALC('rownum()', INTEGER) AS "T_ID"] );`
		SQL Function
		 `TEMP = SELECT "ID" AS "KEY", ROW_NUMBER() OVER () AS "T_ID" FROM :table_var`

Calculation engine plan operators encapsulate data-transformation functions and can be used in the definition of a procedure or a table user-defined function. They constitute a no longer recommended alternative to using SQL statements. Their logic is directly implemented in the calculation engine, which is the execution environments of SQLScript.

There are different categories of operators.

- Data Source Access operators that bind a column table or a column view to a table variable.
- Relational operators that allow a user to bypass the SQL processor during evaluation and to directly interact with the calculation engine.
- Special extensions that implement functions.

8.1 Data Source Access Operators

The data source access operators bind the column table or column view of a data source to a table variable for reference by other built-in operators or statements in a SQLScript procedure.

8.1.1 CE_COLUMN_TABLE

Syntax:

```
CE_COLUMN_TABLE(<table_name> [<attributes>])
```

Syntax Elements:

```
<table_name>    ::= [<schema_name>.]<identifier>
```

Identifies the table name of the column table, with optional schema name.

```
<attributes>  ::= '[' <attrib_name>[{, <attrib_name> }…] ']'
<attrib_name> ::= <string_literal>
```

Restricts the output to the specified attribute names.

Description:

The `CE_COLUMN_TABLE` operator provides access to an existing column table. It takes the name of the table and returns its content bound to a variable. Optionally a list of attribute names can be provided to restrict the output to the given attributes.

Note that many of the calculation engine operators provide a projection list for restricting the attributes returned in the output. In the case of relational operators, the attributes may be renamed in the projection list. The functions that provide data source access provide no renaming of attributes but just a simple projection.

Note

Calculation engine plan operators that reference identifiers must be enclosed with double-quotes and capitalized, ensuring that the identifier's name is consistent with its internal representation.

If the identifiers have been declared without double-quotes in the `CREATE TABLE` statement (which is the normal method), they are internally converted to upper-case letters. Identifiers in calculation engine plan operators must match the internal representation, that is they must be upper case as well.

In contrast, if identifiers have been declared with double-quotes in the `CREATE TABLE` statement, they are stored in a case-sensitive manner. Again, the identifiers in operators must match the internal representation.

8.1.2 CE_JOIN_VIEW

Syntax:

```
CE_JOIN_VIEW(<column_view_name>[{,<attributes>,}...])
```

Syntax elements:

```
<column_view_name> ::= [<schema_name>.]<identifier>
```

Identifies the column view, with optional schema name.

```
<attributes>  ::= '[' <attrib_name>[{, <attrib_name> }…] ']'
<attrib_name> ::= <string_literal>  [AS <column_alias>]
```

Specifies the name of the required columns from the column view.

```
column_alias ::= <string literal>
```

A string representing the desired column alias.

Description:

The `CE_JOIN_VIEW` operator returns results for an existing join view (also known as Attribute View). It takes the name of the join view and an optional list of attributes as parameters of such views/models.

8.1.3 CE_OLAP_VIEW

Syntax:

```
CE_OLAP_VIEW(<olap_view_name>, '['<attributes>']')
```

Syntax elements:

```
<olap_view_name> ::= [<schema_name>.]<identifier>
```

Identifies the olap view, with optional schema name.

```
<attributes> ::= <aggregate_exp> [{, <dimension>}…] [{, <aggregate_exp>}…]
```

Specifies the attributes of the OLAP view.

Note

Note you must have at least one <aggregation_exp> in the attributes.

```
<aggregate_exp> ::= <aggregate_func>(<aggregate_column> [AS <column_alias>])
```

Specifies the required aggregation expression for the key figure.

```
<aggregate_func> ::= COUNT | SUM | MIN | MAX
```

Specifies the aggregation function to use. Supported aggregation functions are:

- `count("column")`
- `sum("column")`
- `min("column")`
- `max("column")`
- use `sum("column") / count("column")` to compute the average

```
<aggregate_column> ::= <string_literal>
```

The identifier for the aggregation column.

```
<column_alias> ::= <string_literal>
```

Specifies an alias for the aggregate column.

```
<dimension> ::= <string_literal>
```

The dimension on which the OLAP view should be grouped.

Description:

The `CE_OLAP_VIEW` operator returns results for an existing OLAP view (also known as an Analytical View). It takes the name of the OLAP view and an optional list of key figures and dimensions as parameters. The OLAP cube that is described by the OLAP view is grouped by the given dimensions and the key figures are aggregated using the default aggregation of the OLAP view.

8.1.4 CE_CALC_VIEW

Syntax:

```
CE_CALC_VIEW(<calc_view_name>, [<attributes>])
```

Syntax elements:

```
<calc_view_name> ::= [<schema_name>.]<identifier>
```

Identifies the calculation view, with optional schema name.

```
<attributes>  ::= '[' <attrib_name>[{, <attrib_name> }…] ']'
<attrib_name> ::= <string_literal>
```

Specifies the name of the required attributes from the calculation view.

Description:

The `CE_CALC_VIEW` operator returns results for an existing calculation view. It takes the name of the calculation view and optionally a projection list of attribute names to restrict the output to the given attributes.

8.2 Relational Operators

The calculation engine plan operators presented in this section provide the functionality of relational operators that are directly executed in the calculation engine. This allows exploitation of the specific semantics of the calculation engine and to tune the code of a procedure if required.

8.2.1 CE_JOIN

Syntax:

```
CE_JOIN (<left_table>, <right_table>, <join_attributes> [<projection_list>])
```

Syntax elements:

```
<left_table>  ::= :<identifier>
```

Identifies the left table of the join.

```
<right_table> ::= :<identifier>
```

Identifies the right table of the join.

```
<join_attributes> ::= '[' <join_attrib>[{, <join_attrib> }…] ']'
<join_attrib>     ::= <string_literal>
```

Specifies a list of join attributes. Since `CE_JOIN` requires equal attribute names, one attribute name per pair of join attributes is sufficient. The list must at least have one element.

```
<projection_list> ::= '[' {, <attrib_name> }… ']'
<attrib_name>     ::= <string_literal>
```

Specifies a projection list for the attributes that should be in the resulting table.

Note

If the optional projection list is present, it must at least contain the join attributes.

Description:

The `CE_JOIN` operator calculates a natural (inner) join of the given pair of tables on a list of join attributes. For each pair of join attributes, only one attribute will be in the result. Optionally, a projection list of attribute names can be given to restrict the output to the given attributes. Finally, the plan operator requires each pair of join attributes to have identical attribute names. In case of join attributes having different names, one of them must be renamed prior to the join.

8.2.2 CE_LEFT_OUTER_JOIN

Calculate the left outer join. Besides the function name, the syntax is the same as for CE_JOIN.

8.2.3 CE_RIGHT_OUTER_JOIN

Calculate the right outer join. Besides the function name, the syntax is the same as for `CE_JOIN`.

Note

CE_FULL_OUTER_JOIN is not supported.

8.2.4 CE_PROJECTION

Syntax:

```
CE_PROJECTION(<var_table>, <projection_list>[, <filter>])
```

Syntax elements:

```
<var_table> ::= :<identifier>
```

Specifies the table variable which is subject to the projection.

```
<projection_list> ::= '[' <attrib_name>[{, <attrib_name> }…] ']'
<attrib_name>     ::= <string_literal>  [AS <column_alias>]
<column_alias>    ::= <string_literal>
```

Specifies a list of attributes that should be in the resulting table. The list must at least have one element. The attributes can be renamed using the SQL keyword `AS`, and expressions can be evaluated using the `CE_CALC` function.

```
<filter> ::= <filter_expression>
```

Specifies an optional filter where Boolean expressions are allowed. See CE_CALC [page 124] for the filter expression syntax.

Description:

Restricts the columns of the table variable `<var_table>` to those mentioned in the projection list. Optionally, you can also rename columns, compute expressions, or apply a filter.

With this operator, the `<projection_list>` is applied first, including column renaming and computation of expressions. As last step, the filter is applied.

Caution

Be aware that `<filter>` in `CE_PROJECTION` can be vulnerable to SQL injection because it behaves like dynamic SQL. Avoid use cases where the value of `<filter>` is passed as an argument from outside of the procedure by the user himself or herself, for example:

```
create procedure proc (in filter nvarchar (20), out output ttype)
begin
tablevar = CE_COLUMN_TABLE(TABLE);
output = CE_PROJECTION(:tablevar,
         ["A", "B"], '"B" = :filter );
end;
```

It enables the user to pass any expression and to query more than was intended, for example: `'02 OR B = 01'`.

SAP recommends that you use plain SQL instead.

8.2.5 CE_CALC

Syntax:

```
CE_CALC ('<expr>', <result_type>)
```

Syntax elements:

```
<expr> ::= <expression>
```

Specifies the expression to be evaluated. Expressions are analyzed using the following grammar:

- b --> b1 ('or' b1)*
- b1 --> b2 ('and' b2)*
- b2 --> 'not' b2 | e (('<' | '>' | '=' | '<=' | '>=' | '!=') e)*
- e --> '-'? e1 ('+' e1 | '-' e1)*
- e1 --> e2 ('*' e2 | '/' e2 | '%' e2)*
- e2 --> e3 ('**' e2)*
- e3 --> '-' e2 | id ('(' (b (',' b)*)? ')')? | const | '(' b ')'

Where terminals in the grammar are enclosed, for example 'token' (denoted with id in the grammar), they are like SQL identifiers. An exception to this is that unquoted identifiers are converted into lower-case. Numeric constants are basically written in the same way as in the C programming language, and string constants are enclosed in single quotes, for example, 'a string'. Inside string, single quotes are escaped by another single quote.

An example expression valid in this grammar is: `"col1" < ("col2" + "col3")`. For a full list of expression functions, see the following table.

```
<result_type> ::= DATE | TIME | SECONDDATE | TIMESTAMP | TINYINT
                      | SMALLINT | INTEGER | BIGINT | SMALLDECIMAL | DECIMAL
                      | REAL | DOUBLE | VARCHAR | NVARCHAR | ALPHANUM
                      | SHORTTEXT | VARBINARY | BLOB | CLOB | NCLOB | TEXT
```

Specifies the result type of the expression as an SQL type

Description:

`CE_CALC` is used inside other relational operators. It evaluates an expression and is usually then bound to a new column. An important use case is evaluating expressions in the `CE_PROJECTION` operator. The `CE_CALC` function takes two arguments:

The following expression functions are supported:

Table 21: Expression Functions

Name	Description	Syntax
Conversion Functions	Convert between data types	
float	convert arg to float type	float float(arg)
double	convert arg to double type	double double(arg)
decfloat	convert arg to decfloat type	decfloat decfloat(arg)

Name	Description	Syntax
fixed	convert arg to fixed type	fixed fixed(arg, int, int)
string	convert arg to string type	string string(arg)
date	convert arg to date type daydate[1]	daydate(stringarg), daydate daydate(fixedarg)
String Functions	Functions on strings	
charpos	returns the one-based position of the nth character in a string. The string is interpreted as using a UTF-8 character encoding	charpos(string, int)
chars	returns the number of characters in a UTF-8 string. In a CESU-8 encoded string this function returns the number of 16-bit words utilized by the string, just the same as if the string is encoded using UTF-16.	chars(string)
strlen	returns the length of a string in bytes, as an integer number[1]	int strlen(string)
midstr	returns a part of the string starting at arg2, arg3 bytes long. arg2 is counted from 1 (not 0) [2]	string midstr(string, int, int)
leftstr	returns arg2 bytes from the left of the arg1. If arg1 is shorter than the value of arg2, the complete string will be returned. [1]	string leftstr(string, int)
rightstr	returns arg2 bytes from the right of the arg1. If arg1 is shorter than the value of arg2, the complete string will be returned. [1]	string rightstr(string, int)
instr	returns the position of the first occurrence of the second string within the first string (>= 1) or 0, if the second string is not contained in the first. [1]	int instr(string, string)
hextoraw	converts a hexadecimal representation of bytes to a string of bytes. The hexadecimal string may contain 0-9, upper or lowercase a-f and no spaces between the two digits of a byte; spaces between bytes are allowed.	string hextoraw(string)
rawtohex	converts a string of bytes to its hexadecimal representation. The output will contain only 0-9 and (upper case) A-F, no spaces and is twice as many bytes as the original string.	string rawtohex(string)

Name	Description	Syntax
ltrim	removes a whitespace prefix from a string. The Whitespace characters may be specified in an optional argument. This functions operates on raw bytes of the UTF8-string and has no knowledge of multi byte codes (you may not specify multi byte whitespace characters).	• string ltrim(string) • string ltrim(string, string)
rtrim	removes trailing whitespace from a string. The Whitespace characters may be specified in an optional argument. This functions operates on raw bytes of the UTF8-string and has no knowledge of multi byte codes (you may not specify multi byte whitespace characters).	• string rtrim(string) • string rtrim(string, string)
trim	removes whitespace from the beginning and end of a string. The following statements are functionally: • trim(s) = ltrim(rtrim(s)) • trim(s1, s2) = ltrim(rtrim(s1, s2), s2)	• string trim(string) • string trim(string, string)
lpad	adds whitespace to the left of a string. A second string argument specifies the whitespace which will be added repeatedly until the string has reached the intended length. If no second string argument is specified, chr(32) (' ') will be added.	• string lpad(string, int) • string lpad(string, int, string)
rpad	adds whitespace to the end of a string. A second string argument specifies the whitespace which will be added repeatedly until the string has reached the intended length. If no second string argument is specified, chr(32) (' ') will be added.	• string rpad(string, int) • string rpad(string, int, string)
Mathematical Functions	The math functions described here generally operate on floating point values; their inputs will automatically convert to double, the output will also be a double.	

Name	Description	Syntax
• double log(double) • double exp(double) • double log10(double) • double sin(double) • double cos(double) • double tan(double) • double asin(double) • double acos(double) • double atan(double) • double sinh(double) • double cosh(double) • double floor(double) • double ceil(double)	These functions have the same functionality as in the Cprogramming language.	
sign	sign returns -1, 0 or 1 depending on the sign of its argument. Sign is implemented for all numeric types, date, and time.	• int sign(double), etc. • int sign(date) • int sign(time)
abs	Abs returns arg, if arg is positive or zero, -arg else. Abs is implemented for all numeric types and time.	• int abs(int). • double abs(double) • decfloat abs(decfloat) • time abs(time)
Date Functions	Functions operating on date or time data	
utctolocal	interpret datearg (a date, without timezone) as utc and convert it to the timezone named by timezonearg (a string)	iutctolocal(datearg, timezonearg)
localtoutc	convert the local datetime datearg to the timezone specified by the string timezonearg, return as a date	localtoutc(datearg, timezonearg)
weekday	returns the weekday as an integer in the range 0..6, 0 is Monday.	weekday(date)
now	returns the current date and time (localtime of the server timezone) as date	now()
daysbetween	returns the number of days (integer) between date1 and date2. This is an alternative to date2 - date1	daysbetween(date1, date2)
Further Functions		
if	return arg2 if intarg is considered true (not equal to zero), else return arg3. Currently, no shortcut evaluation is implemented, meaning that both arg2 and arg3 are evaluated in any case. This means you cannot use if to avoid a divide by zero error which has the side effect of terminating expression evaluation when it occurs.	if(intarg, arg2, arg3)

Name	Description	Syntax
case	return value1 if arg1 == cmp1, value2 if arg1 == cmp2 etc, default if there no match	• case(arg1, default) • case(arg1, cmp1, value1, cmp2, value2, ..., default)
isnull	return 1 (= true), if arg1 is set to null and null checking is on during evaluator run	isnull(arg1)
rownum	returns the number of the row in the currently scanned table structure. The first row has number 0	rownum()

[1] Due to calendar variations with dates earlier that 1582, the use of the `date` data type is deprecated; you should use the `daydate` data type instead.

> **Note**
>
> `date` is based on the proleptic Gregorian calendar. `daydate` is based on the Gregorian calendar which is also the calendar used by SAP HANA SQL.

[2] These Calculation Engine string functions operate using single byte characters. To use these functions with multi-byte character strings please see section: Using String Functions with Multi-byte Character Encoding below. Note, this limitation does not exist for the SQL functions of the SAP HANA database which support Unicode encoded strings natively.

8.2.5.1 Using String Functions with Multi-byte Character Encoding

To allow the use of the string functions of Calculation Engine with multi-byte character encoding you can use the `charpos` and `chars` (see table above for syntax of these commands) functions. An example of this usage for the single byte character function `midstr` follows below:-

```
midstr(<input_string>, charpos(<input_string>, 32), 1)
```

8.2.6 CE_AGGREGATION

Syntax:

```
CE_AGGREGATION (<var_table>, <aggregate_list> [, <group_columns>]);
```

Syntax elements:

```
<var_table>  ::= :<identifier>
```

A variable of type table containing the data that should be aggregated.

Note

CE_AGGREGATION cannot handle tables directly as input.

```
<aggregate_list> ::= '['<aggregate_exp>[{, <aggregate_exp>}] ']'
```

Specifies a list of aggregates. For example, `[SUM ("A"), MAX("B")]` specifies that in the result, column "A" has to be aggregated using the SQL aggregate `SUM` and for column B, the maximum value should be given.

```
<aggregate_exp> ::= <aggregate_func>(<aggregate_column>[AS <column_alias>])
```

Specifies the required aggregation expression.

```
<aggregate_func> ::= COUNT | SUM | MIN | MAX
```

Specifies the aggregation function to use. Supported aggregation functions are:

- `count("column")`
- `sum("column")`
- `min("column")`
- `max("column")`
- use `sum("column") / count("column")` to compute the average

```
<aggregate_column> ::= <string_literal>
```

The identifier for the aggregation column.

```
<column_alias> ::= <string_literal>
```

Specifies an alias for the aggregate column.

```
<group_columns> ::= '['<group_column_name> [{,<group_column_name>}...]']'
```

Specifies an optional list of group-by attributes. For instance, `["C"]` specifies that the output should be grouped by column C. Note that the resulting schema has a column named C in which every attribute value from the input table appears exactly once. If this list is absent the entire input table will be treated as a single group, and the aggregate function is applied to all tuples of the table.

```
<group_column_name> ::= <identifier>
```

Specifies the name of the column attribute for the results to be grouped by.

Note

CE_AGGREGATION implicitly defines a projection: All columns that are not in the list of aggregates, or in the group-by list, are not part of the result.

Description:

Groups the input and computes aggregates for each group.

The result schema is derived from the list of aggregates, followed by the group-by attributes. The order of the returned columns is defined by the order of columns defined in these lists. The attribute names are:

- For the aggregates, the default is the name of the attribute that is aggregated.
- For instance, in the example above `([SUM("A"),MAX("B")])`, the first column is called A and the second is B.
- The attributes can be renamed if the default is not appropriate.
- For the group-by attributes, the attribute names are unchanged. They cannot be renamed using `CE_AGGREGATION`.

i Note

Note that `count(*)` can be achieved by doing an aggregation on any integer column; if no group-by attributes are provided, this counts all non-null values.

8.2.7 CE_UNION_ALL

Syntax:

```
CE_UNION_ALL (<var_table1>, :var_table2)
```

Syntax elements:

```
<var_table1> ::= :<identifier>
<var_table2> ::= :<identifier>
```

Specifies the table variables to be used to form the union.

Description:

The `CE_UNION_ALL` function is semantically equivalent to `SQL UNION ALL` statement. It computes the union of two tables which need to have identical schemas. The `CE_UNION_ALL` function preserves duplicates, so the result is a table which contains all the rows from both input tables.

8.3 Special Operators

In this section we discuss operators that have no immediate counterpart in SQL.

8.3.1 CE_VERTICAL_UNION

Syntax:

```
CE_VERTICAL_UNION(<var_table>, <projection_list> [{,<var_table>,
<projection_list>}...])
```

Syntax elements:

```
<var_table> ::= :<identifier>
```

Specifies a table variable containing a column for the union.

```
<projection_list> ::= ‘[’ <attrib_name>[{, <attrib_name> }…] ‘]’
<attrib_name>     ::= <string_literal>  [AS <column_alias>]
<column_alias>    ::= <string_literal>
```

Specifies a list of attributes that should be in the resulting table. The list must at least have one element. The attributes can be renamed using the SQL keyword `AS`.

Description:

For each input table variable the specified columns are concatenated. Optionally columns can be renamed. All input tables must have the same cardinality.

⚠ Caution

The vertical union is sensitive to the order of its input. SQL statements and many calculation engine plan operators may reorder their input or return their result in different orders across starts. This can lead to unexpected results.

.

8.3.2 CE_CONVERSION

Syntax:

```
CE_CONVERSION(<var_table>, <conversion_params>, [<rename_clause>])
```

Syntax elements:

```
<var_table> ::= :<identifier>
```

Specifies a table variable to be used for the conversion.

```
<conversion_params> ::= '['<key_val_pair>[{,<key_val_pair>}...]']'
```

Specifies the parameters for the conversion. The `CE_CONVERSION`operator is highly configurable via a list of key-value pairs. For the exact conversion parameters permissible, see the *Conversion parameters* table.

```
<key_val_pair> ::= <key> = <value>
```

Specify the key and value pair for the parameter setting.

```
<key> ::= <identifier>
```

Specifies the parameter key name.

```
<value> ::= <string_literal>
```

Specifies the parameter value.

```
<rename_clause> ::= <rename_att>[{,<rename_att>}]
```

Specifies new names for the result columns.

```
<rename_att>     ::= <convert_att> AS <new_param_name>
<convert_att>    ::= <identifier>
<new_param_name> ::= <identifier>
```

Specifies the new name for a result column.

Description:

Applies a unit conversion to input table `<var_table>` and returns the converted values. Result columns can optionally be renamed. The following syntax depicts valid combinations. Supported keys with their allowed domain of values are:

Table 22: Conversion parameters

Key	Values	Type	Mandatory	Default	Documentation
'family'	'currency'	key	Y	none	The family of the conversion to be used.
'method'	'ERP'	key	Y	none	The conversion method.
‘error_handling’	'fail on error', 'set to null', 'keep un-converted'	key	N	'fail on error'	The reaction if a rate could not be determined for a row.
'output'	combinations of 'input', 'uncon-verted', 'con-verted', 'passed_through', 'output_unit', 'source_unit', 'tar-get_unit', 'refer-ence_date'	key	N	'converted, passed_through, output_unit'	Specifies which at-tributes should be included in the output.
'source_unit'	Any	Constant	N	None	The default source unit for any kind of conversion.
'target_unit'	Any	Constant	N	None	The default target unit for any kind of conversion.

Key	Values	Type	Mandatory	Default	Documentation
'reference_date'	Any	Constant	N	None	The default reference date for any kind of conversion.
'source_unit_column'	column in input table	column name	N	None	The name of the column containing the source unit in the input table.
'target_unit_column'	column in input table	column name	N	None	The name of the column containing the target unit in the input table.
'reference_date_column'	column in input table	column name	N	None	The default reference date for any kind of conversion.
'output_unit_column'	Any	column name	N	"OUTPUT_UNIT"	The name of the column containing the target unit in the output table.

For ERP conversion specifically:

Table 23:

Key	Values	Type	Mandatory	Default	
'client'	Any	Constant		None	The client as stored in the tables.
'conversion_type'	Any	Constant		'M'	The conversion type as stored in the tables.
'schema'	Any	schema name		current schema	The default schema in which the conversion tables should be looked-up.

8.3.3 TRACE

Syntax:

```
TRACE(<var_input>)
```

Syntax elements:

```
<var_input>  ::= :<identifier>
```

Identifies the SQLScript variable to be traced.

Description:

The TRACE operator is used to debug SQLScript procedures. It traces the tabular data passed as its argument into a local temporary table and returns its input unmodified. The names of the temporary tables can be retrieved from the `SYS.SQLSCRIPT_TRACE` monitoring view. See SQLSCRIPT_TRACE below.

Example:

You trace the content of variable `input` to a local temporary table.

```
out = TRACE(:input);
```

i Note

This operator should not be used in production code as it will cause significant runtime overhead. Additionally, the naming conventions used to store the tracing information may change. Thus, this operator should only be used during development for debugging purposes.

Related Information

SQLSCRIPT_TRACE

9 Procedure and Function Headers

To eliminate the dependency of having a procedure or a function that already exist when you want to create a new procedure consuming them, you can use headers in their place.

When creating a procedure, all nested procedures that belong to that procedure must exist beforehand. If procedure `P1` calls `P2` internally, then `P2` must have been created earlier than `P1`. Otherwise, `P1` creation fails with the error message,"P2 does not exist". With large application logic and no export or delivery unit available, it can be difficult to determine the order in which the objects need to be created.

To avoid this kind of dependency problem, SAP introduces `HEADERS`. `HEADERS` allow you to create a minimum set of metadata information that contains only the interface of the procedure or function.

```
AS HEADER ONLY
```

You create a header for a procedure by using the `HEADER ONLY` keyword, as in the following example:

```
CREATE PROCEDURE <proc_name> [(<parameter_clause>)] AS HEADER ONLY;
```

With this statement you are creating a procedure `<proc_name>` with the given signature `<parameter_clause>`. The procedure `<proc_name>` has no body definition and thus has no dependent base objects. Container properties (for example, `security mode`, `default_schema`, and so on) cannot be defined with the header definition. These are included in the body definition.

The following statement creates the procedure `TEST_PROC` with a scalar input `INVAR` and a tabular output `OUTTAB`:

```
CREATE PROCEDURE TEST_PROC (IN INVAR NVARCHAR(10), OUT OUTTAB TABLE(no INT)) AS
HEADER ONLY
```

You can create a function header similarly.

```
CREATE FUNCTION <func_name> [(<parameter_clause>)] RETURNS <return_type> AS
HEADER ONLY
```

By checking the `is_header_only` field in the system view `PROCEDURE`, you can verify that a procedure only header is defined.

```
SELECT procedure_name, is_header_only from SYS.PROCEDURES
```

If you want to check for functions, then you need to look into the system view `FUNCTIONS`.

Once a header of a procedure or function is defined, other procedures or functions can refer to it in their procedure body. Procedures containing these headers can be compiled as shown in the following example:

```
CREATE PROCEDURE OUTERPROC (OUT OUTTAB TABLE (NO INT)) LANGUAGE SQLSCRIPT
AS
BEGIN
       DECLARE s INT;
       s = 1;
      CALL TEST_PROC (:s, outtab);
END;
```

As long as the procedure and/or the function contain only a header definition, they cannot be executed. Furthermore, all procedures and functions that use this procedure or function containing headers cannot be executed because they are all invalid.

To change this and to make a valid procedure or function from the header definition, you must replace the header by the full container definition. Use the `ALTER` statement to replace the header definition of a procedure, as follows:

```
ALTER PROCEDURE <proc_name> [(<parameter_clause>)] [LANGUAGE <lang>] [SQL
SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name>][READS SQL DATA] AS
BEGIN [SEQUENTIAL EXECUTION]
  <procedure_body>
END
```

For a function header, the task is similar, as shown in the following example:

```
ALTER FUNCTION <func_name> RETURNS <return_type> [LANGUAGE <lang>] [SQL SECURITY
<mode>][DEFAULT SCHEMA <default_schema_name>]
AS
BEGIN
  <function_body>
END
```

For example, if you want to replace the header definition of `TEST_PROC` that was defined already, then the `ALTER` statement might look as follows:

```
ALTER PROCEDURE TEST_PROC (IN INVAR NVARCHAR(10), OUT OUTTAB TABLE(no INT))
LANGUAGE SQLSCRIPT SQL SECURITY INVOKER READS SQL DATA
AS
BEGIN
     DECLARE tvar TABLE (no INT, name nvarchar(10));
      tvar = SELECT * FROM TAB WHERE name = :invar;
      outtab = SELECT no FROM :tvar;
END
```

You cannot change the signature with the `ALTER` statement. If the name of the procedure or the function or the input and output variables do not match, you will receive an error.

Note

The `ALTER PROCEDURE` and the `ALTER FUNCTION` statements are supported only for a procedure or a function respectively, that contain a header definition.

10 HANA Spatial Support

SQLScript supports the spatial data type `ST_GEOMETRY` and SQL spatial functions to access and manipulate spatial data. In addition SQLScript also supports the objective style function calls, which are needed for some SQL spatial functions.

The next example illustrates a small scenario of using spatial data type and function in SQLScript.

The function `get_distance` calculates the distance between the two given parameters `<first>` and `<second>` of type `ST_GEOMETRY` by using the spatial function `ST_DISTANCE`.

Note the ':' in front of the variable `<first>`. This needed because you are reading from the variable.

The function `get_distance` itself is called by the procedure `nested_call`. The procedure returns the distance and the text representation of the `ST_GEOMETRY` variable `<first>`.

```
CREATE FUNCTION get_distance( IN first ST_GEOMETRY, IN second ST_GEOMETRY )
RETURNS distance
double
AS
BEGIN
      distance =  :first.st_distance(:second);
END;
CREATE PROCEDURE nested_call(   IN first ST_GEOMETRY,
                                IN second ST_GEOMETRY,
                                OUT distance  double,
                                OUT res3 CLOB
                             )
AS
BEGIN

      Distance = get_distance (:first, :second);
      res3 = :first.st_astext();
END;
```

The procedure call

```
CALL nested_call(   first    => st_geomfromtext('Point(7 48)'),
                    second   => st_geomfromtext('Point(2 55)'),
                    distance => ?,
                    res3     => ?);
```

will then return the following result:

```
Out(1)               Out(2)
----------------------------------------------------------------------
8,602325267042627    POINT(7 48)
```

Note that the optional SRID (Spatial reference Identifier) parameter in SQL spatial functions is mandatory if the function is used within SQLScript. If you not specify the SRID you will receive an error as demonstrated with the function `ST_GEOMFROMTEXT` in the following example. Here SRID 0 is used which specifies the default spatial reference system.

```
DO
BEGIN
```

```
    DECLARE arr ST_GEOMETRY ARRAY;
    DECLARE line1 ST_GEOMETRY = ST_GEOMFROMTEXT('LINESTRING(1 1, 2 2, 5 5)', 0);
    DECLARE line2 ST_GEOMETRY = ST_GEOMFROMTEXT('LINESTRING(1 1, 3 3, 5 5)', 0);
    arr[1] = :line1;
    arr[2] = :line2;
    tmp2 = UNNEST(:arr) AS (A);
    select A from :tmp2;
END;
```

If you do not use the same SRID for the `ST_GEOMETRY` variables `<line1>` and `<line2>` latest the `UNNEST` will throw an error because it is not allowed that the values in one column have different SRID.

Besides this there is a consistency check for output table variables to ensure that all elements of a spatial column have the same SRID.

Note that the following function are not currently not supported in SQLScript:

- `ST_CLUSTERID`
- `ST_CLUSTERCENTEROID`
- `ST_CLUSTERENVELOPE`
- `ST_CLUSTERCONVEXHULL`
- `ST_AsSVG`

The construction of objects with the NEW keyword is also not supported in SQLScript. Instead you can use `ST_GEOMFROMTEXT('POINT(1 1)', srid).`

Please refer to the "SAP HANA Spatial Reference" available from the SAP HANA platform page for having detailed information about the SQL spatial functions and their usage.

11 System Variables

System Variables are build-in variables in SQLScript that provide you information about the current context.

11.1 ::CURRENT_OBJECT_NAME and ::CURRENT_OBJECT_SCHEMA

To identify the name of the current running procedure or function you can use the following two system variables:

::CURRENT_OBJECT_NAME	Returns the name of the current procedure or function
::CURRENT_OBJECT_SCHEMA	Returns the name of the schema of current procedure or function

Both return a string of type NVARCHAR(256).

The following example illustrates the usage of the system variables.

```
CREATE FUNCTION RETURN_NAME ()
RETURNS name        nvarchar(256),
        schema_name nvarchar(256)
AS
BEGIN
     name        = ::CURRENT_OBJECT_NAME;
     schema_name = ::CURRENT_OBJECT_SCHEMA;
END;
```

By calling that function, e.g.

```
SELECT RETURN_NAME().schema_name, RETURN_NAME().name from dummy
```

the result of that function is then the `name` and the `schema_name` of the function:

```
SCHEMA_NAME       NAME
----------------------------------------
MY_SCHEMA         RETURN_NAME
```

The next example shows that you can also pass the two system variables as arguments to procedure or function call.

```
CREATE FUNCTION GET_FULL_QUALIFIED_NAME (schema_name nvarchar(256),name
nvarchar(256))
RETURNS fullname nvarchar(256)
AS
BEGIN
     fullname = schema_name || '.' || name ;
END;
```

```
CREATE PROCEDURE MAIN_PROC (IN INPUT_VALUE INTEGER)
AS
BEGIN
     DECLARE full_qualified_name NVARCHAR(256);
     DECLARE error_text NVARCHAR(256);
     full_qualified_name = get_full_qualified_name (::CURRENT_OBJECT_SCHEMA,
                                                       ::CURRENT_OBJECT_NAME);

     IF :input_value > 1 OR :input_value < 0 THEN
         SIGNAL SQL_ERROR_CODE 10000 SET MESSAGE_TEXT =  'ERROR IN '
            || :full_qualified_name || ': invalid input value ';
     END IF;
END;
```

Note

Note that in anonymous blocks the value of both system variables is NULL.

The two system variable will always return the schema name and the name of the procedure or function. Creating a synonym on top of the procedure or function and calling it with the synonym will still return the original name as shown in the next example.

We create a synonym on the `RETURN_NAME` function from above and will query it with the synonym:

```
CREATE SYNONYM SYN_FOR_FUNCTION FOR RETURN_NAME;
SELECT SYNONYM_FOR_FUNCTION().schema_name, SYNONYM_FOR_FUNCTION().name FROM
dummy;
```

The result is the following:

```
SCHEMA_NAME          NAME
------------------------------------------------------
MY_SCHEMA          RETURN_NAME
```

11.2 ::ROWCOUNT

The System Variable `::ROWCOUNT` stores the number of updated row counts of the previously executed DML statement. There is no accumulation of all previously executed DML statements.

The next examples shows you how you can use `::ROWCOUNT` in a procedure. Consider we have the following table T:

```
CREATE TABLE T (NUM INT, VAL INT);
INSERT INTO T VALUES (1, 1);
INSERT INTO T VALUES (2, 2);
INSERT INTO T VALUES (1, 2);
```

Now we want to update table T and want to return the number of updated rows:

```
CREATE PROCEDURE PROC_UPDATE (OUT updated_rows INT) AS
BEGIN
    UPDATE T SET VAL = VAL + 1 WHERE VAL = 2;
    updated_rows = ::ROWCOUNT;
END;
```

By calling the procedure with

```
CALL PROC_UPDATE (updated_rows => ?);
```

We get the following result back:

```
UPDATED_ROWS
-------------------------
2
```

In the next example we change the procedure by having two update statements and in the end we again get the row count:

```
ALTER PROCEDURE PROC_UPDATE (OUT updated_rows INT) AS
BEGIN
   UPDATE T SET VAL = VAL + 1 WHERE VAL = 3;
   UPDATE T SET VAL = VAL + 1 WHERE VAL = 1;
   updated_rows = ::ROWCOUNT;
END;
```

By calling the procedure you will see that the number of updated rows is now 1. That is because the las update statements only updated one row.

```
UPDATED_ROWS
-------------------------
1
```

If you now want to have the number of all updated rows you have to retrieve the row count information after each update statement and accumulate them:

```
ALTER PROCEDURE PROC_UPDATE (OUT updated_rows INT) AS
BEGIN
    UPDATE T SET VAL = VAL + 1 WHERE VAL = 4;
    updated_rows = ::ROWCOUNT;
    UPDATE T SET VAL = VAL + 1 WHERE VAL = 2;
    updated_rows = :updated_rows + ::ROWCOUNT;
END;
```

By now calling this procedure again the number of updated row is now 3:

```
UPDATED_ROWS
-------------------------
3
```

12 Query Parameterization: BIND_AS_PARAMETER and BIND_AS_VALUE

All scalar variables used in queries of procedures, functions or anonymous blocks, are represented either as query parameters, or as constant values during query compilation. Which option shall be chosen is a decision of the optimizer.

Example

The following procedure uses two scalar variables (var1 and var2) in the WHERE-clause of a nested query.

Sample Code

```
CREATE PROCEDURE PROC (IN var1 INT, IN var2 INT, OUT tab mytab)
AS
BEGIN
    tab = SELECT * FROM MYTAB WHERE MYCOL >:var1
                                 OR MYCOL =:var2;
END;
```

Calling the procedure by using query parameters in the callable statement

Sample Code

```
CALL PROC (var1=>?, var2=>?, mytab=>?)
```

will prepare the nested query of the table variable tab by using query parameters for the scalar parameters:

Sample Code

```
SELECT * FROM MYTAB WHERE MYCOL >? OR MYCOL =?
```

Before the query is executed, the parameter values will be bound to the query parameters.

Calling the procedure without query parameters and using constant values directly

Sample Code

```
CALL PROC (var1=>1, var2=>2, mytab=>?)
```

will lead to the following query string, which uses the parameter values directly:

Sample Code

```
SELECT * FROM MYTAB WHERE MYCOL >1 OR MYCOL =2;
```

The advantage of using query parameters is that the generated query plan cache entry can be used even if the values of the variables var1 and var2 change. A potential disadvantage is that there is a chance of not getting the most optimal query plan because optimizations using parameter values cannot be performed directly during compilation time. Using constant values will always lead to preparing a new query plan and therefore to different query plan cache entries for the different parameter values. This comes along with additional time spend for query preparation and potential cache flooding effects in fast-changing parameter value scenarios.

In order to control the parameterization behavior of scalar parameters explicitly, you can use the function BIND_AS_PARAMETER and BIND_AS_VALUE. The decision of the optimizer and the general configuration are overridden when you use these functions.

Syntax

```
<bind_as_function> ::= BIND_AS_PARAMETER ( <scalar_variable> )|
                       BIND_AS_VALUE(<scalar_variable> )
```

Using BIND_AS_PARAMETER will always use a query parameter to represent a <scalar_variable> during query preparation.

Using BIND_AS_VALUE will always use a value to represent a <scalar_variable> during query preparation.

The following example represents the same procedure from above but now using the functions BIND_AS_PARAMETER and BIND_AS_VALUE instead of referring to the scalar parameters directly:

Sample Code

```
CREATE PROCEDURE PROC (IN var1 INT, IN var2 INT, OUT tab mytab)
 AS
 BEGIN
    tab = SELECT * FROM MYTAB WHERE MYCOL > BIND_AS_PARAMETER(:var1)
                                 OR MYCOL = BIND_AS_VALUE(:var2);
 END;
```

If you call the procedure again with

Sample Code

```
CALL PROC (var1=>?, var2=>?, mytab=>?)
```

and bind the values

Sample Code

```
1 for var1 and 2 for var2
```

the following query string will be prepared

Sample Code

```
SELECT * FROM MYTAB WHERE MYCOL >? OR MYCOL = 2;
```

The same query string will be prepared even if you call this procedure with constant values because the functions override the optimizer's decisions.

Sample Code

```
CALL PROC (var1=>1, var2=>2, mytab=>?)
```

13 Supportability

13.1 M_ACTIVE_PROCEDURES

The view M_ACTIVE_PROCEDURES monitors all internally executed statements starting from a procedure call. That also includes remotely executed statements.

M_ACTIVE_PROCEDURES is similar to M_ACTIVE_STATEMENTS but keeps the records of completed internal statements until the parent procedure finishes, and shows them in hierarchical order of nested level. The structure of M_ACTIVE_PROCEDURES looks as follows:

Table 24:

Column name	Data type	Description
PROCEDURE_HOST	VARCHAR(64)	Procedure Host
PROCEDURE_PORT	INTEGER	Procedure Internal Port
PROCEDURE_SCHEMA_NAME	NVARCHAR(256)	Schema name of the stored procedure
PROCEDURE_NAME	NVARCHAR(256)	Name of the stored procedure
PROCEDURE_CONNECTION_ID	INTEGER	Procedure connection ID
PROCEDURE_TRANSACTION_ID	INTEGER	Procedure transaction ID
STATEMENT_ID	VARCHAR(20)	Logical ID of the statement
STATEMENT_STRING	NCLOB	SQL statement
STATEMENT_PARAMETERS	NCLOB	Statement parameters
STATEMENT_STATUS	VARCHAR(16)	Status of the statement: EXECUTING: statement is still running COMPLETED: statement is completed COMPILING: statement will be com-piled ABORTED: statement was aborted
STATEMENT_EXECUTION_COUNT	INTEGER	Count of statement execution

Column name	Data type	Description
STATEMENT_DEPTH	INTEGER	Statement depth
STATEMENT_COMPILE_TIME	BIGINT	Elapsed time for compiling statement (mircoseconds)
STATEMENT_EXECUTION_TIME	BIGINT	Elapsed time for executing statement (mircoseconds)
STATEMENT_START_TIME	TIMESTAMP	Statement start time
STATEMENT_END_TIME	TIMESTAMP	Statement end time
STATEMENT_CONNECTION_ID	INTEGER	Connection ID of the statement
STATEMENT_TRANSACTION_ID	INTEGER	Transaction ID of the statement

Among other things the M_ACTIVE_PROCEDURES is helpful for analyzing long running procedures and to determine their current status. You can run the following query from another session to find more about the status of a procedure, like MY_SCHEMA.MY_PROC in the example:

```
select * from M_ACTIVE_PROCEDURES where procedure_name = ‘my_proc’ and
procedure_schema_name = ‘my_schema’;
```

There is also the INI-configuration `execution_monitoring_level` available to control the granularity of monitoring level:

Table 25:

Level	Description
0	Disabling the feature
1	Full information but no values for 'STATEMENT_PARAMETER'
2	Default monitoring level, full information for the monitoring view
3	Intermediate status of internal statements will be traced into indexserver trace.

To prevent flooding of the memory with irrelevant data, the number of records is limited. If the record count exceeds the given threshold then the first record will be erased independent from its status. The limit can be adjust the INI-Parameter `execution_monitoring_limit`, e.g. `execution_monitoring_limit = 100 000`.

Limitations:

- No triggers and functions are supported.
- Information other than EAPI layer is not monitored. (but might be included in the total compilation time or execution time)

As mentioned above, M_ACTIVE_PROCEDURES keeps the records of completed internal statements until the parent procedure finishes which is the default behavior. This behavior can be changed with the following two configuration parameters: `NUMBER_OF_CALLS_TO_RETAIN_AFTER_EXECUTION` and `RETENTION_PERIOD_FOR_SQLSCRIPT_CONTEXT`.

With `NUMBER_OF_CALLS_TO_RETAIN_AFTER_EXECUTION` you can specify how many calls retain after execution and `RETENTION_PERIOD_FOR_SQLSCRIPT_CONTEXT` defines how long the result should be kept in M_ACTIVE_PROCEDURES, unit is in seconds. The interaction of these two parameters are as follows:

- Both parameters are set: M_ACTIVE_PROCEDURES keeps the specified numbers of records for the specified amount of time
- Only `NUMBER_OF_CALLS_TO_RETAIN_AFTER_EXECUTION` is set: M_ACTIVE_PROCEDURES keeps the specified number for the default amount of time (= 3600 seconds)
- Only `RETENTION_PERIOD_FOR_SQLSCRIPT_CONTEXT` is set: M_ACTIVE_PROCEDURES keeps the default number of records (= 100) for the specified amount of time
- Nothing set: no records are kept.

Note

All configuration parameters needs to be defined under the `sqlscript` section

13.1.1 Retention and Memory Tracking in M_ACTIVE_PROCEDURES

13.2 Query Export

The Query Export is an enhancement of the EXPORT statement. It allows exporting queries, that is database objects used in a query together with the query string and parameters. This query can be either standalone, or executed as a part of a SQLScript procedure.

13.2.1 SQLScript Query Export

Prerequisites

In order to execute the query export as a developer you need an EXPORT system privilege.

Procedure

To export one or multiple queries of a procedure, use the following syntax:

```
EXPORT ALL AS <export_format> INTO <path> [WITH <export_option_list>]ON
<sqlscript_location_list> FOR <procedure_call_statement>
```

With <export_format> you define whether the export should use a BINARY format or a CSV format.

```
<export_format> ::= BINARY | CSV
```

i Note

Currently the only format supported for SQLScript query export is CSV . If you choose BINARY, you get a warning message and the export is performed in CSV.

The server path where the export files are be stored is specified as <path>.

```
<path> ::= <string_literal>
```

For more information about <export_option_list>, see EXPORT in the SAP HANA SQL and System Views Reference.

Apart from SELECT statements, you can export the following statement types as well:

- Nested calls DMLs (INSERT, DELETE, ...)
- DDLs (CREATE TABLE, ...)
- Dynamic SQL (anything except EXPORT)

The information about the queries to be exported is defined by <sqlscript_location_list>.

```
<sqlscript_location_list> ::= <sqlscript_location> [{,
<sqlscript_location_list>}]
<sqlscript_location>       ::= ( [ <procedure_name> ] LINE <line_number> [ COLUMN
<column_number> ] [ PASS (<pass_number> | ALL)] )
<procedure_name>           ::= [<schema_name>.]<identifier>
<line_number>              ::= <unsigned_integer>
<column_number>            ::= <unsigned_integer>
<pass_number>              ::= <unsigned_integer>
```

With the <sqlscript_location_list> you can define in a comma-separated list several queries that you want to export. For each query you have to specify the name of the procedure with <procedure_name> to indicate where the query is located. <procedure_name> can be omitted if it is the same procedure as the procedure in <procedure_call_statement>.

You also need to specify the line information, <line_number>, and the column information, <column_number>. The line number must correspond to the first line of the statement. If the column number is omitted, all statements (usually there is just one) on this line are exported. Otherwise the column must match the first character of the statement.

The line and column information is usually contained in the comments of the queries generated by SQLScript and can be taken over from there. For example, the monitoring view M_ACTIVE_PROCEDURES or the statement statistic in PlanViz shows the executed queries together with the comment.

Consider the following two procedures:

```
1 CREATE PROCEDURE proc_one (...)
2 AS
3 BEGIN
    ...
15     tab = SELECT * FROM :t;
    ...
30     CALL proc_two (...);
    ...
98 END;
1 CREATE PROCEDURE proc_two (...)
2 AS
3 BEGIN
    ...
27     temp = SELECT * FROM :v; temp2 = SELECT * FROM :v2;
    ...
40 END;
```

If you want to export both queries of table variables **tabtemp**, then the <sqlscript_location> looks as follows: and

```
(proc_one LINE 15), (proc_two LINE 27 COLUMN 4)
```

For the query of table variable temp we also specified the column number because there are two table variable assignments on one line and we only wanted to have the first query.

To export these queries, the export needs to execute the procedure call that triggers the execution of the procedure containing the queries. Therefore the procedure call has to be specified as well by using <procedure_call_statement>:

```
<procedure_call_statement> ::= CALL <procedure_name> (<param_list>)
```

For information on <procedure_call_statement> see CALL [page 30].

The export statement of the above given example is the following:

```
EXPORT ALL AS CSV INTO '/tmp'  ON (proc_one LINE 15), ( proc_two LINE 27 COLUMN
4) FOR CALL PROC_ONE (...);
```

If you want to export a query that is executed multiple times, you can use <pass_number> to specify which execution should be exported. If <pass_number> is omitted, only the first execution of the query is exported. If you need to export multiple passes, but not all of them, you need to specify the same location multiple times with the corresponding pass numbers.

```
1 CREATE PROCEDURE MYSCHEMA.PROC_LOOP (...)
2 AS
3 BEGIN
    ...
       FOR i IN 1 .. 1000 DO
        ...
34         temp = SELECT * FROM :v;
        ...
37     END FOR;
    ...
40 END;
```

Given the above example, we want to export the query on line 34 but only the snapshot of the 2nd and 30th loop iteration. The export statement is then the following, considering that PROC_LOOP is a procedure call:

```
EXPORT ALL AS CSV INTO '/tmp' ON (myschema.proc_loop LINE 34 PASS 2),
(myschema.proc_loop LINE 34 PASS 30) FOR CALL PROC_LOOP(...);
```

If you want to export the snapshots of all iterations you need to use PASS ALL:

```
EXPORT ALL AS CSV INTO '/tmp' ON (myschema.proc_loop LINE 34 PASS ALL) FOR CALL
PROC_LOOP(...);
```

Overall the SQLScript Query Export creates one subdirectory for each exported query under the given path <path> with the following name pattern <schema_name>-<procedure_name>-<line_number>-<column_number>-<pass_number >. For example the directories of the first above mentioned export statement would be the following:

```
|_ /tmp
    |_ MYSCHEMA-PROC_LOOP-34-10-2
        |_Query.sql
        |_index
        |_export
    |_ MYSCHEMA-PROC_LOOP-34-10-30
        |_Query.sql
        |_index
        |_export
```

The exported SQLScript query is stored in a file named Query.sql and all related base objects of that query are stored in the directories index and export, as it is done for a typical catalog export.

You can import the exported objects, including temporary tables and their data, with the IMPORT statement.

For more information about IMPORT, see IMPORT in the SAP HANA SQL and System Views Reference.

i Note

Queries within a function are not supported and cannot be exported.

i Note

Query export is not supported on distributed systems. Only single-node systems are supported.

13.3 Type and Length Check for Table Parameters

The derived table type of a tabular variable should always match the declared type of the corresponding variable, both for the type code as well as length or precision/scale information. This is particular important for signature variables as they can be considered the contract a caller will follow. The derived type code will be implicitly converted if this conversion is possible without loss in information (see SQL guide for further details on which data types conversion are supported).

If the derived type is larger (e.g. `BIGINT`) than the expected type (e.g. `INTEGER`) can this lead to errors as shown in the following example.

The procedure PROC_TYPE_MISMATCH has a defined tabular output variable RESULT with a single column of type VARCHAR with a length of 2. The derived type from the table variable assignment has a single column of type VARCHAR with a length of 10.

```
CREATE COLUMN TABLE  tab_vc10 (A VARCHAR(10));
INSERT INTO tab_vc10 VALUES ('ab');
INSERT INTO tab_vc10 VALUES ('ab');
CREATE PROCEDURE PROC_WITH_TYPE_MISMATCH (OUT result TABLE(A VARCHAR(2))) AS
BEGIN
    result = select A from tab_vc10;
END;
```

Calling this procedure will work fine as long as the difference in length does not matter e.g. calling this procedure from any SQL client will not cause an issues. However using the result for further processing can lead to an error as shown in the following example:

```
CREATE PROCEDURE PROC_WITH_TYPE_MISMATCH_CALLER() AS
BEGIN
    CALL PROC_WITH_TYPE_MISMATCH (result);
    INSERT INTO tab_vc2(select * from :result);
END
```

The procedure PROC_WITH_TYPE_MISMATCH_CALLER tries to insert the result of procedure PROC_WITH_TYPE_MISMTACH into the table tab_vc2 which has a single column of type VARCHAR with a length of 2. In case the length of the values in the received result are longer than 2 characters this operation will throw an error: inserted value to large. Please note that the INSERT operation will run fine in case the length of the values in the received result will not exceed 2 characters.

To avoid such errors the configuration parameters Typecheck_Procedure_Output_Var and Typecheck_Procedure_Input_Var were introduced. These parameters are intended to expose differences between expected and derived type information. The default behavior of the parameters is to throw a warning in case of type mismatch. For example during the creation or call of procedure PROC_WITH_TYPE_MISMATCH the following warning will be thrown:

```
Declared type "VARCHAR(2)" of attribute "A" not same as assigned type "VARCHAR(10)"
```

The configuration parameters have three different levels to reveal differences between expected and derived types if the derived type is larger than the expected type:

Table 26:

Level	Output	Description
silent	--	Ignore potential type error
warn	general warning: Declared type "VAR-CHAR(2)" of attribute "A" not same as assigned type "VARCHAR(10)"	Print warning in case of type mis-match(default behavior)
strict	return type mismatch: Declared type "VARCHAR(2)" of attribute "A" not same as assigned type "VARCHAR(10)"	Error in case of potential type error

Note

Both configuration parameters needs to be defined under the `sqlscript` section

13.4 SQLScript Debugger

With the SQLScript debugger you can investigate functional issues. The debugger is available in the SAP WebIDE for SAP HANA (WebIDE) and in ABAP in Eclipse (ADT Debugger). In the following we want to give you an overview of the available functionality and also in which IDE it is supported. For a detailed description of how to use the SQLScript debugger, see the documentation of SAP WebIDE for SAP HANA and ABAP in Eclipse available at the SAP HANA Help Portal.

Table 27:

Feature	Procedures	Table Functions	Scalar Functions	Anonymous Blocks
Debugging	WebIDE ADT Debugger	WebIDE ADT Debugger [1]	WebIDE[2]	-
Breakpoints	WebIDE ADT Debugger	WebIDE ADT Debugger	WebIDE	-
Conditonal Breakpoint	WebIDE	WebIDE	WebIDE	-
Watchpoints	WebIDE	WebIDE	-	-
Break on Error	WebIDE	WebIDE	WebIDE	-
Save Table	WebIDE	WebIDE	WebIDE	-

13.4.1 Conditional Breakpoints

A conditional breakpoint can be used to break the debugger in the breakpoint-line only when certain conditions are met. This is especially useful when a Breakpoint is set within a loop.

Each breakpoint can have only one condition. The condition expressions can contain any SQL function. A condition must either contain an expression that results in true or false, or can contain a single variable or a complex expression without restrictions in the return type.

[1] NetWeaver 751, NetWeaver 765
[2] Only works if the scalar function is assigned to a variable within a procedure or a table function that also has a breakpoint set - the user will get this information in a warning when setting a breakpoint

When setting a conditional breakpoint, the debugger will check all conditions for potential syntax errors. It checks for:

- syntax errors like missing brackets or misuse of operators
- unknown or wrong function calls
- unknown variables
- wrong return type (isTrue condition must return true or false)

At execution time the debugger will check and evaluate the conditions of the conditional breakpoints, but with the given variables and its values. If the value of a variable in a condition is not accessible and therefor the condition cannot be evaluated, the debugger will send a warning and will break for the breakpoint anyway.

Note

The debugger will also break and send a warning, if there are expressions set, that access a variable that is not yet accessible at this point (NULL value).

Note

Conditional breakpoints are only supported for scalar variables.

For more information on SQL functions, see FUNCTION in the SAP HANA SQL and System Views Reference.

13.4.2 Watchpoints

Watchpoints give you the possibility to watch the values of variables or complex expressions and break the debugger if certain conditions are met.

For each watchpoint an arbitrary number of conditions can be defined. The conditions can either contain an expression that results in true or false or contain a single variable or complex expression without restrictions in the return type.

When setting a watchpoint, the debugger will check all conditions for potential syntax errors. It checks for:

- syntax errors like missing brackets or misuse of operators
- unknown or wrong function calls

At execution time the debugger will check and evaluate the conditions of the watchpoints, but with the given variables and its values. A watchpoint will be skipped, if the value of a variable in a condition is not accessible. But in case the return type of the condition is wrong , the debugger will send a warning to the user and will break for the watchpoint anyway.

Note

If a variable value changes to `NULL`, the debugger will not break since it cannot evaluate the expression anymore.

13.4.3 Break on Error

You can activate the Exception Mode to allow the Debugger to break if an error in the execution of a procedure or function occurs. User defined exceptions are also handled.

The debugger will stop in the line where the exception was thrown and allows access to the current value of all local variables, the call stack and a short information about the error. Afterwards the execution can be continued and you might step into the exception handler or further exceptions (e.g. on a call statement).

13.4.4 Save Table

Save tables allows you to store the result set of a table variable into persistent table in a predefined schema in a debugging session.

14 Best Practices for Using SQLScript

So far this document has introduced the syntax and semantics of SQLScript. This knowledge is sufficient for mapping functional requirements to SQLScript procedures. However, besides functional correctness, non-functional characteristics of a program play an important role for user acceptance. For instance, one of the most important non-functional characteristics is performance.

The following optimizations all apply to statements in SQLScript. The optimizations presented here cover how dataflow exploits parallelism in the SAP HANA database.

- Reduce Complexity of SQL Statements: Break up a complex SQL statement into many simpler ones. This makes a SQLScript procedure easier to comprehend.
- Identify Common Sub-Expressions: If you split a complex query into logical sub queries it can help the optimizer to identify common sub expressions and to derive more efficient execution plans.
- Multi-Level-Aggregation: In the special case of multi-level aggregations, SQLScript can exploit results at a finer grouping for computing coarser aggregations and return the different granularities of groups in distinct table variables. This could save the client the effort of reexamining the query result.
- Understand the Costs of Statements: Employ the explain plan facility to investigate the performance impact of different SQL queries.
- Exploit Underlying Engine: SQLScript can exploit the specific capabilities of the OLAP- and JOIN-Engine by relying on views modeled appropriately.
- Reduce Dependencies: As SQLScript is translated into a dataflow graph, and independent paths in this graph can be executed in parallel, reducing dependencies enables better parallelism, and thus better performance.
- Avoid Mixing Calculation Engine Plan Operators and SQL Queries: Mixing calculation engine plan operators and SQL may lead to missed opportunities to apply optimizations as calculation engine plan operators and SQL statements are optimized independently.
- Avoid Using Cursors: Check if use of cursors can be replaced by (a flow of) SQL statements for better opportunities for optimization and exploiting parallel execution.
- Avoid Using Dynamic SQL: Executing dynamic SQL is slow because compile time checks and query optimization must be done for every invocation of the procedure. Another related problem is security because constructing SQL statements without proper checks of the variables used may harm security.

14.1 Reduce Complexity of SQL Statements

Best Practices: Reduce Complexity of SQL Statements

Variables in SQLScript enable you to arbitrarily break up a complex SQL statement into many simpler ones. This makes a SQLScript procedure easier to comprehend. To illustrate this point, consider the following query:

```
 books_per_publisher = SELECT publisher, COUNT (*) AS cnt
FROM :books GROUP BY publisher;
largest_publishers = SELECT * FROM :books_per_publisher
WHERE cnt >= (SELECT MAX (cnt)
FROM :books_per_publisher);
```

Writing this query as a single SQL statement either requires the definition of a temporary view (using WITH) or repeating a sub query multiple times. The two statements above break the complex query into two simpler SQL statements that are linked via table variables. This query is much easier to comprehend because the names of the table variables convey the meaning of the query and they also break the complex query into smaller logical pieces.

The SQLScript compiler will combine these statements into a single query or identify the common sub-expression using the table variables as hints. The resulting application program is easier to understand without sacrificing performance.

14.2 Identify Common Sub-Expressions

Best Practices: Identify Common Sub-Expressions

The query examined in the previous sub section contained common sub-expressions. Such common sub-expressions might introduce expensive repeated computation that should be avoided. For query optimizers it is very complicated to detect common sub-expressions in SQL queries. If you break up a complex query into logical sub queries it can help the optimizer to identify common sub-expressions and to derive more efficient execution plans. If in doubt, you should employ the EXPLAIN plan facility for SQL statements to investigate how the HDB treats a particular statement.

14.3 Multi-Level Aggregation

Best Practices: Multi-level aggregation

Computing multi-level aggregation can be achieved by using grouping sets. The advantage of this approach is that multiple levels of grouping can be computed in a single SQL statement.

```
SELECT publisher, name, year, SUM(price)
 FROM :it_publishers, :it_books
WHERE publisher=pub_id AND crcy=:currency
GROUP BY GROUPING SETS ((publisher, name, year), (year))
```

To retrieve the different levels of aggregation, the client typically has to examine the result repeatedly, for example by filtering by NULL on the grouping attributes.

In the special case of multi-level aggregations, SQLScript can exploit results at a finer grouping for computing coarser aggregations and return the different granularities of groups in distinct table variables. This could save the client the effort of re-examining the query result. Consider the above multi-level aggregation expressed in SQLScript.

```
books_ppy = SELECT publisher, name, year, SUM(price)
FROM :it_publishers, :it_books
WHERE publisher = pub_id AND crcy = :currency
GROUP BY publisher, name, year;
 books_py = SELECT year, SUM(price)
FROM :books_ppy
```

```
GROUP BY year;
```

14.4 Understand the Costs of Statements

It is important to keep in mind that even though the SAP HANA database is an in-memory database engine and that the operations are fast, each operation has its associated costs and some are much more costly than others.

As an example, calculating a UNION ALL of two result sets is cheaper than calculating a UNION of the same result sets because of the duplicate elimination the UNION operation performs. The calculation engine plan operator CE_UNION_ALL (and also UNION ALL) basically stacks the two input tables over each other by using references without moving any data within the memory. Duplicate elimination as part of UNION, in contrast, requires either sorting or hashing the data to realize the duplicate removal, and thus a materialization of data. Various examples similar to these exist. Therefore it is important to be aware of such issues and, if possible, to avoid these costly operations.

You can get the query plan from the view SYS.QUERY_PLANS. The view is shared by all users. Here is an example of reading a query plan from the view.

```
EXPLAIN PLAN [ SET PLAN_ID = <plan_id> ] FOR <dml_stmt>
SELECT lpad(' ', level) || operator_name AS operator_name,
       operator_details, object_name, subtree_cost,
       input_cardinality, output_cardinality, operator_id,
       parent_operator_id, level, position
       FROM sys.query_plans
       WHERE PLAN_ID = <plan_id> ORDER BY operator_id;
```

Sometimes alternative formulations of the same query can lead to faster response times. Consequently reformulating performance critical queries and examining their plan may lead to better performance.

The SAP HANA database provides a library of application-level functions which handle frequent tasks, e.g. currency conversions. These functions can be expensive to execute, so it makes sense to reduce the input as much as possible prior to calling the function.

14.5 Exploit Underlying Engine

Best Practices: Exploit Underlying Engine

SQLScript can exploit the specific capabilities of the built-in functions or SQL statements. For instance, if your data model is a star schema, it makes sense to model the data as an Analytic view. This allows the SAP HANA database to exploit the star schema when computing joins producing much better performance.

Similarly, if the application involves complex joins, it might make sense to model the data either as an Attribute view or a Graphical Calculation view. Again, this conveys additional information on the structure of the data which is exploited by the SAP HANA database for computing joins. When deciding to use Graphical Calculation views involving complex joins refer to SAP note 1857202 for details on how, and under which conditions, you may benefit from SQL Engine processing with Graphical Calculation views.

Using CE functions only, or alternatively SQL statements only, in a procedure allows for many optimizations in the underlying database system. However when SQLScript procedures using imperative constructs are called by other programs, for example predicates to filter data early, can no longer be applied. The performance impact of using these constructs must be carefully analyzed when performance is critical.

Finally, note that not assigning the result of an SQL query to a table variable will return the result of this query directly to the client as a result set. In some cases the result of the query can be streamed (or pipelined) to the client. This can be very effective as this result does not need to be materialized on the server before it is returned to the client.

14.6 Reduce Dependencies

Best Practices: Reduce Dependencies

One of the most important methods for speeding up processing in the SAP HANA database is a massive parallelization of executing queries. In particular, parallelization is exploited at multiple levels of granularity: For example, the requests of different users can be processed in parallel, and also single relational operators within a query are executed on multiple cores in parallel. It is also possible to execute different statements of a single SQLScript in parallel if these statements are independent of each other. Remember that SQLScript is translated into a dataflow graph, and independent paths in this graph can be executed in parallel.

From an SQLScript developer perspective, we can support the database engine in its attempt to parallelize execution by avoiding unnecessary dependencies between separate SQL statements, and also by using declarative constructs if possible. The former means avoiding variable references, and the latter means avoiding imperative features, for example cursors.

14.7 Avoid Mixing Calculation Engine Plan Operators and SQL Queries

Best Practices: Avoid Mixing Calculation Engine Plan Operators and SQL Queries

The semantics of relational operations as used in SQL queries and calculation engine operations are different. In the calculation engine operations will be instantiated by the query that is executed on top of the generated data flow graph.

Therefore the query can significantly change the semantics of the data flow graph. For example consider a calculation view that is queried using attribute publisher (but not year) that contains an aggregation node (CE_AGGREGATION) which is defined on publisher and year. The grouping on year would be removed from the grouping. Evidently this reduces the granularity of the grouping, and thus changes the semantics of the model. On the other hand, in a nested SQL query containing a grouping on publisher and year this aggregation-level would not be changed if an enclosed query only queries on publisher.

Because of the different semantics outlined above, the optimization of a mixed data flow using both types of operations is currently limited. Hence, one should avoid mixing both types of operations in one procedure.

14.8 Avoid Using Cursors

Best Practices: Avoid Using Cursors

While the use of cursors is sometime required, they imply row-at-a-time processing. As a consequence, opportunities for optimizations by the SQL engine are missed. So you should consider replacing the use of cursors with loops, by SQL statements as follows:

Read-Only Access

For read-only access to a cursor consider using simple selects or join:

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS
Reads SQL DATA
BEGIN
    DECLARE val decimal(34,10) = 0;
    DECLARE CURSOR c_cursor1 FOR
    SELECT isbn, title, price FROM books;
    FOR r1 AS c_cursor1 DO
    val = :val + r1.price;
    END FOR;
END;
```

This sum can also be computed by the SQL engine:

```
SELECT sum(price) into val FROM books;
```

Computing this aggregate in the SQL engine may result in parallel execution on multiple CPUs inside the SQL executor.

Updates and Deletes

For updates and deletes, consider using the

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE val INT = 0;
    DECLARE CURSOR c_cursor1 FOR
    SELECT isbn, title, price FROM books;
    FOR r1 AS c_cursor1 DO
        IF r1.price > 50 THEN
            DELETE FROM Books WHERE isbn = r1.isbn;
        END IF;
    END FOR;
END;
```

This delete can also be computed by the SQL engine:

```
DELETE FROM Books
WHERE isbn IN (SELECT isbn FROM books WHERE price > 50);
```

Computing this in the SQL engine reduces the calls through the runtime stack of HDB and potentially benefits from internal optimizations like buffering or parallel execution.

Insertion into Tables

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE val INT = 0;
    DECLARE CURSOR c_cursor1 FOR SELECT isbn, title, price FROM books;
    FOR r1 AS c_cursor1 DO
        IF r1.price > 50
        THEN
            INSERT INTO ExpensiveBooks VALUES(..., r1.title, ...);
        END IF;
    END FOR;
END;
```

This insertion can also be computed by the SQL engine:

```
 SELECT ..., title, ... FROM Books WHERE price > 50
INTO ExpensiveBooks;
```

Similar to updates and deletes, computing this statement in the SQL engine reduces the calls through the runtime stack of the SAP HANA database, and potentially benefits from internal optimizations like buffering or parallel execution.

14.9 Avoid Using Dynamic SQL

Best Practices: Avoid using Dynamic SQL

Dynamic SQL is a powerful way to express application logic. It allows for constructing SQL statements at execution time of a procedure. However, executing dynamic SQL is slow because compile time checks and query optimization must be done for every start up of the procedure. So when there is an alternative to dynamic SQL using variables, this should be used instead.

Another related problem is security because constructing SQL statements without proper checks of the variables used can create a security vulnerability, for example, SQL injection. Using variables in SQL statements prevents these problems because type checks are performed at compile time and parameters cannot inject arbitrary SQL code.

Summarizing potential use cases for dynamic SQL are:

Table 28: Dynamic SQL Use Cases

Feature	Proposed Solution
Projected attributes	Dynamic SQL
Projected literals	SQL + variables
`FROM` clause	SQL + variables; result structure must remain unchanged

Feature	Proposed Solution
`WHERE` clause – attribute names & Boolean operators	`APPLY_FILTER`

15 Developing Applications with SQLScript

This section contains information about creating applications with SQLScript for SAP HANA.

15.1 Handling Temporary Data

In this section we briefly summarize the concepts employed by the SAP HANA database for handling temporary data.

Table Variables are used to conceptually represent tabular data in the data flow of a SQLScript procedure. This data may or may not be materialized into internal tables during execution. This depends on the optimizations applied to the SQLScript procedure. Their main use is to structure SQLScript logic.

Temporary Tables are tables that exist within the life time of a session. For one connection one can have multiple sessions. In most cases disconnecting and reestablishing a connection is used to terminate a session. The schema of global temporary tables is visible for multiple sessions. However, the data stored in this table is private to each session. In contrast, for local temporary tables neither the schema nor the data is visible outside the present session. In most aspects, temporary tables behave like regular column tables.

Persistent Data Structures are like sequences and are only used within a procedure call. However, sequences are always globally defined and visible (assuming the correct privileges). For temporary usage – even in the presence of concurrent invocations of a procedure, you can invent a naming schema to avoid sequences. Such a sequence can then be created using dynamic SQL.

15.2 SQL Query for Ranking

Ranking can be performed using a Self-Join that counts the number of items that would get the same or lower rank. This idea is implemented in the sales statistical example below.

```
create table sales (product int primary key, revenue int);
select product, revenue,
       (select count(*)
        from sales s1 where s1.revenue <= s2.revenue) as rank
from sales s2
order by rank asc
```

Related Information

SAP HANA SQL and System Views Reference

15.3 Calling SQLScript From Clients

In this document we have discussed the syntax for creating SQLScript procedures and calling them. Besides the SQL command console for invoking a procedure, calls to SQLScript will also be embedded into client code. In this section we present examples how this can be done.

15.3.1 Calling SQLScript from ABAP

Using CALL DATBASE PROCEDURE

The best way to call SQLScript from ABAP is to create a procedure proxy which can be natively called from ABAP by using the built in command `CALL DATABASE PROCEDURE`.

The SQLScript procedure has to be created normally in the SAP HANA Studio with the HANA Modeler. After this a procedure proxy can be creating using the ABAP Development Tools for Eclipse. In the procedure proxy the type mapping between ABAP and HANA data types can be adjusted. The procedure proxy is transported normally with the ABAP transport system while the HANA procedure may be transported within a delivery unit as a TLOGO object.

Calling the procedure in ABAP is very simple. The example below shows calling a procedure with two inputs (one scalar, one table) and one (table) output parameter:

```
CALL DATABASE PROCEDURE z_proxy
EXPORTING   iv_scalar = lv_scalar
            it_table  = lt_table
IMPORTING   et_table1 = lt_table_res.
```

Using the connection clause of the `CALL DATABASE PROCEDURE` command, it is also possible to call a database procedure using a secondary database connection. Please consult the ABAP help for detailed instructions of how to use the `CALL DATABASE PROCEDURE` command and for the exceptions may be raised.

It is also possible to create procedure proxies with an ABAP API programmatically. Please consult the documentation of the class `CL_DBPROC_PROXY_FACTORY` for more information on this topic.

Using ADBC

```
*&---------------------------------------------------------------------*
*& Report ZRS_NATIVE_SQLSCRIPT_CALL
*&---------------------------------------------------------------------*
*&
*&---------------------------------------------------------------------*
report zrs_native_sqlscript_call.
parameters:
  con_name type dbcon-con_name default 'DEFAULT'.
```

```
types:
* result table structure
  begin of result_t,
    key   type i,
    value type string,
  end of result_t.
data:
* ADBC
  sqlerr_ref type ref to cx_sql_exception,
  con_ref type ref to cl_sql_connection,
  stmt_ref type ref to cl_sql_statement,
  res_ref type ref to cl_sql_result_set,
* results
  result_tab type table of result_t,
  row_cnt type i.
start-of-selection.
  try.
      con_ref = cl_sql_connection=>get_connection( con_name ).
      stmt_ref = con_ref->create_statement( ).
*************************************
** Setup test and procedure
*************************************
* Create test table
      try.
          stmt_ref->execute_ddl( 'DROP TABLE zrs_testproc_tab' ).
        catch cx_sql_exception.
      endtry.
      stmt_ref->execute_ddl(
        'CREATE TABLE zrs_testproc_tab( key INT PRIMARY KEY, value
NVARCHAR(255) )' ).
      stmt_ref->execute_update(
        'INSERT INTO zrs_testproc_tab VALUES(1, ''Test value'' )' ).
* Create test procedure with one output parameter
      try.
          stmt_ref->execute_ddl( 'DROP PROCEDURE zrs_testproc' ).
        catch cx_sql_exception.
      endtry.
      stmt_ref->execute_ddl(
        `CREATE PROCEDURE zrs_testproc( OUT t1 zrs_testproc_tab ) ` &&
        `READS SQL DATA AS ` &&
        `BEGIN ` &&
        `   t1 = SELECT * FROM zrs_testproc_tab; ` &&
        `END`
      ).
*************************************
** Execution time
*************************************
      perform execute_with_transfer_table.
      perform execute_with_gen_temptables.
      con_ref->close( ).
    catch cx_sql_exception into sqlerr_ref.
      perform handle_sql_exception using sqlerr_ref.
  endtry.
form execute_with_transfer_table.
  data lr_result type ref to data.
* Create transfer table for output parameter
* this table is used to transfer data for parameter 1 of proc zrs_testproc
* for each procedure a new transfer table has to be created
* when the procedure is executed via result view, this table is not needed
* If the procedure has more than one table type parameter, a transfer table is
needed for each parameter
* Transfer tables for input parameters have to be filled first before the call
is executed
  try.
      stmt_ref->execute_ddl( 'DROP TABLE zrs_testproc_p1' ).
    catch cx_sql_exception.
  endtry.
  stmt_ref->execute_ddl(
    'CREATE GLOBAL TEMPORARY COLUMN TABLE zrs_testproc_p1( key int, value 
NVARCHAR(255) )'
  ).
* clear output table in session
* should be done each time before the procedure is called
  stmt_ref->execute_ddl( 'TRUNCATE TABLE zrs_testproc_p1' ).
* execute procedure call
  res_ref = stmt_ref->execute_query( 'CALL zrs_testproc( zrs_testproc_p1 ) WITH 
OVERVIEW' ).
  res_ref->close( ).
* read result for output parameter from output transfer table
  res_ref = stmt_ref->execute_query( 'SELECT * FROM zrs_testproc_p1' ).
* assign internal output table
  clear result_tab.
  get reference of result_tab into lr_result.
  res_ref->set_param_table( lr_result ).
* get the complete result set in the internal table
  row_cnt = res_ref->next_package( ).
  write: / 'EXECUTE WITH TRANSFER TABLE:', / 'Row count: ', row_cnt.
  perform output_result.
endform.
form execute_with_gen_temptables.
* mapping between procedure output parameters
* and generated temporary tables
  types:
    begin of s_outparams,
      param_name type string,
      temptable_name type string,
    end of s_outparams.
  data lt_outparam type standard table of s_outparams.
  data lr_outparam type ref to data.
  data lr_result type ref to data.
  field-symbols <ls_outparam> type s_outparams.
* call the procedure which returns the mapping between procedure parameters
* and the generated temporary tables
  res_ref = stmt_ref->execute_query( 'CALL zrs_testproc(null) WITH OVERVIEW' ).
  clear lt_outparam.
  get reference of lt_outparam into lr_outparam.
  res_ref->set_param_table( lr_outparam ).
  res_ref->next_package( ).
* get the temporary table name for the parameter T1
  read table lt_outparam assigning <ls_outparam>
    with key param_name = 'T1'.
  assert sy-subrc is initial.
* retrieve the procedure output from the generated temporary table
  res_ref = stmt_ref->execute_query( 'SELECT * FROM ' && <ls_outparam>-
temptable_name ).
  clear result_tab.
  get reference of result_tab into lr_result.
  res_ref->set_param_table( lr_result ).
  row_cnt = res_ref->next_package( ).
  write: / 'EXECUTE WITH GENERATED TEMP TABLES:', / 'Row count:', row_cnt.
  perform output_result.
endform.
form handle_sql_exception
  using p_sqlerr_ref type ref to cx_sql_exception.
  format color col_negative.
  if p_sqlerr_ref->db_error = 'X'.
    write: / 'SQL error occured:', p_sqlerr_ref->sql_code,  "#EC NOTEXT
    / p_sqlerr_ref->sql_message.
  else.
    write:
    / 'Error from DBI (details in dev-trace):',             "#EC NOTEXT
    p_sqlerr_ref->internal_error.
  endif.
endform.
form output_result.
  write / 'Result table:'.
  field-symbols <ls> type result_t.
  loop at result_tab assigning <ls>.
    write: / <ls>-key, <ls>-value.
  endloop.
endform.
```

Output:

```
EXECUTE WITH TRANSFER TABLE:
Row count:         1
Result table:
         1 Test value
EXECUTE WITH GENERATED TEMP TABLES:
Row count:         1
Result table_
         1 Test value
```

15.3.2 Calling SQLScript from Java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.CallableStatement;
import java.sql.ResultSet;
…
import java.sql.SQLException;CallableStatement cSt = null;
String sql = "call SqlScriptDocumentation.getSalesBooks(?,?,?,?)";
ResultSet rs = null;
Connection conn = getDBConnection();  // establish connection to database using
jdbc
try {
    cSt = conn.prepareCall(sql);
    if (cSt == null) {
        System.out.println("error preparing call: " + sql);
        return;
    }
    cSt.setFloat(1, 1.5f);
    cSt.setString(2, "'EUR'");
    cSt.setString(3, "books");
    int res = cSt.executeUpdate();
    System.out.println("result: " + res);
    do {
        rs = cSt.getResultSet();
        while (rs != null && rs.next()) {
            System.out.println("row: " + rs.getString(1) + ", " +
                      rs.getDouble(2) + ", " + rs.getString(3));
        }
    } while (cSt.getMoreResults());
} catch (Exception se) {
    se.printStackTrace();
} finally {
    if (rs != null)
        rs.close();
    if (cSt != null)
        cSt.close();
}
```

15.3.3 Calling SQLScript from C#

Given procedure:

```
CREATE PROCEDURE TEST_PRO1(IN strin NVARCHAR(100), OUT SorP NVARCHAR(100))
language sqlscript AS
BEGIN
    select 10 from dummy;
    SorP = N'input str is ' || strin;
END;
```

This procedure can be called as follows:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.Common;
using ADODB;
using System.Data.SqlClient;
namespace NetODBC
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                DbConnection conn;
                DbProviderFactory _DbProviderFactoryObject;
                String connStr = "DRIVER={HDBODBC32};UID=SYSTEM;PWD=<password>;
                                  SERVERNODE=<host>:<port>;DATABASE=SYSTEM";
                String ProviderName = "System.Data.Odbc";
                _DbProviderFactoryObject =
DbProviderFactories.GetFactory(ProviderName);
                conn = _DbProviderFactoryObject.CreateConnection();
                conn.ConnectionString = connStr;
                conn.Open();
                System.Console.WriteLine("Connect to HANA database
successfully");
                DbCommand cmd = conn.CreateCommand();
                //call Stored Procedure
                cmd = conn.CreateCommand();
                cmd.CommandText = "call SqlScriptDocumentation.scalar_proc (?)";
                DbParameter inParam = cmd.CreateParameter();
                inParam.Direction = ParameterDirection.Input;
                inParam.Value = "asc";
                cmd.Parameters.Add(inParam);
                DbParameter outParam = cmd.CreateParameter();
                outParam.Direction = ParameterDirection.Output;
                outParam.ParameterName = "a";
                outParam.DbType = DbType.Integer;
                cmd.Parameters.Add(outParam);
                reader = cmd.ExecuteReader();
                System.Console.WriteLine("Out put parameters = " +
outParam.Value);
                reader.Read();
                String row1 = reader.GetString(0);
                System.Console.WriteLine("row1=" + row1);
            }
            catch(Exception e)
            {
                System.Console.WriteLine("Operation failed");
                System.Console.WriteLine(e.Message);
            }
```

```
                    }
          }
}
```

16 Appendix

16.1 Example code snippets

The examples used throughout this manual make use of various predefined code blocks. These code snippets are presented below.

16.1.1 ins_msg_proc

This code is used in the examples in this reference manual to store outputs so the action of the examples can be seen. It simple stores some text along with a timestamp of the entry.

Before you can use this procedure you must create the following table.

```
CREATE TABLE message_box (p_msg VARCHAR(200), tstamp TIMESTAMP);
```

You can create the procedure as follows.

```
CREATE PROCEDURE ins_msg_proc (p_msg VARCHAR(200)) LANGUAGE SQLSCRIPT AS
BEGIN
    INSERT INTO message_box VALUES (:p_msg, CURRENT_TIMESTAMP);
END;
```

To view the contents of the message_box you select the messages in the table.

```
select * from message_box;
```

Important Disclaimer for Features in SAP HANA Platform, Options and Capabilities

SAP HANA server software and tools can be used for several SAP HANA platform and options scenarios as well as the respective capabilities used in these scenarios. The availability of these is based on the available SAP HANA licenses and the SAP HANA landscape, including the type and version of the back-end systems the SAP HANA administration and development tools are connected to. There are several types of licenses available for SAP HANA. Depending on your SAP HANA installation license type, some of the features and tools described in the SAP HANA platform documentation may only be available in the SAP HANA options and capabilities, which may be released independently of an SAP HANA Platform Support Package Stack (SPS). Although various features included in SAP HANA options and capabilities are cited in the SAP HANA platform documentation, each SAP HANA edition governs the options and capabilities available. Based on this, customers do not necessarily have the right to use features included in SAP HANA options and capabilities. For customers to whom these license restrictions apply, the use of features included in SAP HANA options and capabilities in a production system requires purchasing the corresponding software license(s) from SAP. The documentation for the SAP HANA optional components is available in SAP Help Portal at http://help.sap.com/hana_options. If you have additional questions about what your particular license provides, or wish to discuss licensing features available in SAP HANA options, please contact your SAP account team representative.

Important Disclaimers and Legal Information

go.sap.com/registration/contact.html