PUBLIC
SAP HANA Platform 2.0 SPS 05
Document Version: 1.1 – 2021-07-09

SAP HANA SQLScript Reference for SAP HANA Platform

THE BEST RUN SAP

Content

1 SAP HANA SQLScript Reference

This reference describes how to use the SQL extension SAP HANA SQLScript to embed data-intensive application logic into SAP HANA.

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 that 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 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 that can be used to express and encapsulate complex data flows
- Procedural extension, which provides imperative constructs executed in the context of the database process.

3 What is SQLScript?

The motivation behind SQLScript is to embed data-intensive application logic into the database. Currently, 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 avoiding massive data copies to the application server and to leveraging sophisticated parallel execution strategies of the database. SQLScript addresses the following problems:

- Decomposing an SQL query can only be performed by using views. However, when decomposing complex queries by 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 business logic cannot be pushed down into the database (even if it is mainly based on standard aggregations like SUM(Sales), and so on).
- 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.

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 following 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.

→ Tip

For more information about security in SAP HANA, see the *SAP HANA Security Guide*.

Related Information

SQL Injection Prevention Functions [page 209]

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 executes a procedure, for example by using the CALL statement, the SAP HANA database query compiler processes the statement in a way similar to the processing of an SQL statement.

A step-by-step analysis of the process flow follows below:

- Parsing the statement: detecting and reporting simple syntactic errors.
- Checking the semantic correctness of the statement: deriving types for variables and checking if their use is consistent.
- Optimizing the code: the original definition from the user is optimized for a better execution plan. For example, the optimizer simplifies the control flow, merges statements, and embeds nested procedure CALLs. Also, the optimizer analyzes data dependency to exploit parallelism behind the logic.
- Generating the execution plan: based on the optimized code, the SQLScript execution plan is generated.
- Execution: the execution starts with binding the actual parameters to the SQLScript execution plan. The plan is executed sequentially or in parallel.

4 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

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>
```

5 Data Type Extension

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

5.1 Scalar Data Types

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

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
Boolean Type	BOOLEAN

i Note

SQLScript currently allows a length of 8388607 characters for the NVARCHAR and the VARCHAR data types, unlike SQL where the length of that data type is limited to 5000.

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

5.2 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.

5.2.1 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 it should be created.

```
<column_list_definition> ::= <column_elem> [{, <column_elem>}...]
<column_elem> ::= <column_name> <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 | BOOLEAN
```

The available data types

For more information on data types, see Scalar Data Types [page 14].

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 by using a list of attribute names and primitive data types. The attributes of each table type 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);
```

5.2.2 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;
```

5.3 Row-Type Variable

You can declare a row-type variable, which is a collection of scalar data types, and use it to easily fetch a single row from a table.

Syntax

```
DECLARE <sql_identifier> [ {, <sql_identifier> }… ] [ CONSTANT ] ROW
{ <row_element_list> | <row_like> } [ { DEFAULT | '=' } <row_default_value> ] ;
```

Syntax Elements

```
<row_element_list> ::= '(' <row_element> [ { , <row_element> }… ] ')'
<row_element> ::= <identifier> <sql_type>
<row_like> ::= LIKE { <table_name> | <variable > }
<table_name> ::= [<schema_name> '.'] <identifier>
<variable> ::= ':' <identifier>
<row_default_value> ::= <row_constructor> | <variable>
<row_constructor> ::= ROW '(' <expression> [ { , <expression> }… ] ')'
```

Assigning Values to a Row-Type Variable

To assign values to a row-type variable or to reference values of a row-type variable, proceed as follows:

```
DO BEGIN
    DECLARE x, y ROW (a INT, b VARCHAR(16), c TIMESTAMP);
    x = ROW(1, 'a', '2000-01-01');
    x.a = 2;
    y = :x;
    SELECT :y.a, :y.b, :y.c FROM DUMMY;
    -- Returns [2, 'a', '2000-01-01']
END;
```

Selecting Values into a Row-Type Variable

You can fetch or select multiple values into a **single** row-type variable.

```
DO BEGIN
    DECLARE CURSOR cur FOR SELECT 1 as a, 'a' as b, to_timestamp('2000-01-01')
as c FROM DUMMY;
    DECLARE x ROW LIKE :cur;
    OPEN cur;
    FETCH cur INTO x;
    SELECT :x.a, :x.b, :x.c FROM DUMMY;
    -- Returns [1, 'a', '2000-01-01']
    SELECT 2, 'b', '2000-02-02' INTO x FROM DUMMY;
    SELECT :x.a, :x.b, :x.c FROM DUMMY;
    -- Returns [2, 'b', '2000-02-02']
END;
```

Limitations

- EXEC INTO is not supported.
- It is not possible to pass row-type variables as parameters of procedures or functions.

6 Logic Container

The following types of logic containers are available in SQLScript: Procedure, Anonymous Block, User-Defined Function, and User-Defined Library.

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 the containers.

6.1 Procedures

Procedures allow you to describe a sequence of data transformations on data that is 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 that are 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 between a materialization strategy (that avoids re-computation of expressions) and other optimizing strategies. That makes the task of detecting common sub-expressions easier and improves the readability of the SQLScript code.
- You can use scalar variables or imperative language features, if required.

6.1.1 CREATE PROCEDURE

You use this SQL statement to create a procedure.

Syntax

```
 CREATE [OR REPLACE] PROCEDURE <proc_name> [(<parameter_clause>)] [LANGUAGE
<lang>] [SQL SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name>]
 [READS SQL DATA ] [<variable_cache_clause>] [ DETERMINISTIC ] [WITH ENCRYPTION]
[AUTOCOMMIT DDL { ON|OFF } ]
 AS
 { BEGIN [ SEQUENTIAL EXECUTION | PARALLEL EXECUTION ]
```

```
  <procedure_body>
END | HEADER ONLY }
```

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
```

> **i 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 or the array type.
  Array variables or constant arrays can be passed to procedures as input, output, and inout parameters with the following limitations:
  - LOB type array parameter is not supported.
  - DEFAULT VALUE for an array parameter is not supported.
  - Using an array parameter in the USING clause of Dynamic SQL is not supported.

```
<param_type> ::= <sql_type> [ARRAY] | <table_type> | <table_type_definition> |
<any_table_type>
```

- 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
```

> **i Note**
>
> For more information on data types see *Data Types* in the SAP HANA SQL Reference Guide on the SAP Help Portal.

- A table type previously defined with the `CREATE TYPE` command, see CREATE TYPE [page 15].

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

- A table type implicitly defined within the signature

```
<table_type_definition>   ::=  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, the `current_schema` of the session, when the procedure is defined, 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
```

- For more information on `<variable_cache_clause>`, see SQLScript Variable Cache [page 105].
- By default, every SQLScript procedure or function runs with AUTOCOMMIT mode OFF and AUTOCOMMIT DDL mode OFF. In some cases, AUTOCOMMIT DDL mode ON may be required, like administrative operations (for example, IMPORT) that cannot run with DDL AUTOCOMMIT mode OFF. Now you can explicitly specify whether the procedure should be run with AUTOCOMMIT DDL mode ON or OFF. The default value for the property remains 'OFF'.

```
AUTOCOMMIT DDL ON|OFF
```

  You can find out the AUTOCOMMIT DDL mode for each procedure by using the column 'AUTO_COMMIT_DDL' in the system view 'PROCEDURES'.
  The following restrictions apply:
  - It cannot be used in functions

    - It cannot be used in non-SQLScript procedures
    - It cannot be used in read-only procedures.
- Defines the main body of the procedure according to the programming language selected

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

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

```
SEQUENTIAL EXECUTION
```

- Procedure variable, cursor, and condition 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 | CONTINUE } 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>
                  | <proc_insert>
                  | <proc_update>
                  | <proc_delete>
```

- Insert a new data record at a specific position into a table variable

```
<proc_insert> ::= :<table_variable>.INSERT((<value_1>,…, <value_n>), <index>)
```

  For more information on inserting, updating and deleting data records, see Modifying the Content of Table Variables [page 115].
- You can modify a data record at a specific position. There are two equivalent syntax options:

```
<proc_update> ::= :<table_variable>.UPDATE((<value_1>,…, <value_n>), <index>)
```

```
<proc_update> ::= <table_variable>[<index>] = (<value_1>,…, <value_n>)
```

- You can delete data records from a table variable. Wth the following syntax you can delete a single record.

```
<proc_delete> ::= :<table_variable>.DELETE(<index>)
```

- To delete blocks of records from table variables, you can use the following syntax:

```
<proc_delete> ::= :<table_variable>.DELETE(<from_index>..<to_index>)
```

- 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 Variables [page 201].

```
<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 Operator [page 101].

```
<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 216].

```
 <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> ) ;
```

```
<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 169].

```
<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> [DEFAULT
  <scalar_expr_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.

6.1.2 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 you use this drop option and a dependent object exists, you will get an error.

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;
```

6.1.3 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] [<variable_cache_clause>] [ DETERMINISTIC ] [WITH ENCRYPTION]
[AUTOCOMMIT DDL { ON|OFF } ]  AS
BEGIN [SEQUENTIAL EXECUTION]
  <procedure_body>
END
```

For more information about the parameters, see 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`).

> **i Note**
>
> If the default schema and read-only mode are not explicitly specified, they will be removed. The default language is SQLScript.

> **i Note**
>
> You must have the `ALTER` privilege for the object you want to change.

6.1.4 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 are as follows:

- 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.

6.1.4.1 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 12] and Scalar Data Types [page 14].

Parameters passed to a procedure are scalar constants and can be passed as IN, OUT or INOUT parameters. Scalar parameters are assumed to be NOT NULL. Arguments for IN parameters of table type can be either 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` 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 way you do that for query results. SQL statements, which 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.

When executed by the client, the `CALL` 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 written in upper case. Quoting identifiers will take into account capitalization and allow the usage of white spaces that 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 using the `WITH OVERVIEW` option:

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

It is also possible to use scalar user-defined function as parameters for a 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]

6.1.4.2 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>| <inout_scalar_param>
```

The type of the parameters can be either table or scalar.

```
<in_table_param> ::= <in_param>
<in_scalar_param> ::= <in_param>|<scalar_value>|<expression>
```

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

Specifies a procedure input parameter

> **i Note**
>
> Use a colon before the identifier name.

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

```
<out_scalar_param> ::= <out_param>
<out_table_param> ::= <out_param>
```

```
<inout_scalar_param> ::= <out_param>
```

Specifies a procedure output 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. The result is implicitly bound to the variable given in the function call.

Example:

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

When the 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 [page 30]

6.1.4.3 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.

6.1.5 Procedure Parameters

Parameter Modes

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

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 **i Note** This is only supported for scalar values. The parameter needs to be parameterized if you call the procedure. For example, `CALL PROC ( inout_var=>?)`. A non-parameterized call of a procedure with an `INOUT` parameter is not supported.

Supported Parameter Types

Both scalar and table parameter types are supported. For more information on data types, see **Data Type Extension**

Related Information

Data Type Extension [page 14]

6.1.5.1 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')
> ```

6.1.5.2 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
```

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

Related Information

CALL with Named Parameters [page 33]

6.1.5.3 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();
```

6.1.6 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. For more information see Data Import Export Statements in the SAP HANA SQL and System Views Reference.

Related Information

SAP HANA SQL and System Views Reference

6.1.6.1 SYS.PROCEDURES

Available stored procedures

Structure

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
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'

Column name	Data type	Description
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

6.1.6.2 SYS. PROCEDURE_PARAMETERS

Parameters of stored procedures

Structure

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
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

Column name	Data type	Description
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'

6.1.6.3 SYS.OBJECT_DEPENDENCIES

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

Structure

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

Column name	Data type	Description
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)

6.1.6.3.1 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:

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:

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:

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

6.1.6.4 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.

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'

6.2 User-Defined Functions

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 and differ in terms of their input and output parameters, functions supported in the body, and in the way they are consumed in SQL statements.

	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. That takes place 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 • Table types (with limitations)
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 and DML are not allowed and only other read-only functions can be called.	The function is tagged as a read-only function by default.

6.2.1 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 [OR REPLACE] FUNCTION <func_name> [(<parameter_clause>)] RETURNS
<return_type>
[LANGUAGE <lang>] [SQL SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name> ]
[READS SQL DATA] [<variable_cache_clause>] [ DETERMINISTIC ]
[WITH ENCRYPTION]
AS
BEGIN
     <function_body>
END
[ <cache_clause> ]
<cache_clause> ::=
   WITH [ STATIC ] CACHE
   RETENTION <minute_value>
   [ OF <projection_list> ]
   [ FILTER <filter_condition> ]
   [ <location_clause> ]
   [ FORCE ]
```

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> [ARRAY] | <table_type> | <table_type_definition> |
<any_table_type>
```

Scalar user-defined functions (SUDF) support the following primitive SQL types. Table types (table variables, physical tables, or views) are also supported as input in SUDFs. Arrays are supported as input and return types.

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

SUDFs with table parameters can be used like any other SUDF with following exceptions:

- Aliases (in FROM or WITH clauses) are not allowed.
- Parameterized views, scripted calculation views or TUDFs as input are not supported.
- ANY TABLE TYPE parameters are not supported.
- SQLScript internal types, such as cursor variables or ROW types, are not supported.

> **i Note**
>
> Take into consideration the following note on performance. SUDFs operate on table data row by row. In the following example, the operation would be at least `O(record_count(t1) * record_count(t2))`.
>
> ```
> select sudf_taking_table_parameter(t1) from t2;
> ```

Table user-defined functions (TUDF) allow the following range of primitive SQL types. They also support table types and array types as input.

```
<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 15].

```
<table_type_definition>    ::=  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> [ARRAY]
```

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.

i 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> ::= [<func_block_decl_list>]
[<func_handler_list>]
<func_stmt_list>
```

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.

> **i 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. For more information, see *Expressions* in the SAP HANA Reference Guide on the SAP Help Portal.

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

See CREATE PROCEDURE [page 19].

```
 <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>
                    | <func_return_statement>
```

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

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
```

6.2.2 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> [(<parameter_clause>)] RETURNS <return_type>
[LANGUAGE <lang>] [SQL SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name> ]
[READS SQL DATA] [<variable_cache_clause>]
[DETERMINISTIC][WITH ENCRYPTION]
AS
BEGIN
     <function_body>
END
```

For more information about the parameters, see `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**
>
> If the default schema is not explicitly specified, it will be removed.

> **i Note**
>
> You need the ALTER privilege for the object you want to change.

6.2.3 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;
```

6.2.4 Function Parameters

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

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.

6.2.5 Consistent Scalar Function Result

The implicit SELECT statements used within a procedure (or an anonymous block) are executed after the procedure is finished and scalar user-defined functions (SUDF) are evaluated at the fetch time of the SELECT statement, due to the design of late materialization. To avoid unexpected results for statements, that are out of the statement snapshot order within a procedure or a SUDF, implicit result sets will now be materialized in case the SUDF references a persistent table.

```
CREATE TABLE t1(C1 VARCHAR(20));
CREATE FUNCTION my_count RETURNS v_result INTEGER AS
BEGIN
    SELECT COUNT(*) INTO v_result FROM t1;
END;
CREATE PROCEDURE proc_insert_delete AS
BEGIN
    INSERT INTO t1 VALUES ('test');
    SELECT 'TRACE 1: COUNT AFTER INSERT', COUNT(*) FROM t1;
    SELECT 'TRACE 2: COUNT DURING FUNCTION CALL', my_count() FROM DUMMY;
    DELETE FROM t1;
    SELECT 'TRACE 3: COUNT AFTER DELETE', COUNT(*) FROM t1;
    COMMIT;
END;
CALL proc_insert_delete;
-- ('TRACE 1: COUNT AFTER INSERT', 1),
-- ('TRACE 2: COUNT DURING FUNCTION CALL', 1),
-- ('TRACE 3: COUNT AFTER DELETE', 0),
```

6.2.6 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.

6.2.6.1 SYS.FUNCTIONS

A list of available functions

Structure

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 function:'DEFINER'/'INVOKER'
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'

Column name	Data type	Description
OWNER_NAME	NVARCHAR(256)	Name of the owner of the function

6.2.6.2 SYS.FUNCTION_PARAMETERS

A list of parameters of functions

Structure

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
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'

Column name	Data type	Description
IS_NULLABLE	VARCHAR(5)	Specifies whether the parameter accepts a null value: 'TRUE', 'FALSE'

6.2.6.3 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.

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
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'

6.2.7 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;
```

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

Related Information

CALL with Named Parameters [page 33]

6.2.8 SQL Embedded Function

SQLScript allows a table function to be embedded inside an SQL query without the creation of any additional metadata. The HANA SQL query now accepts `SQL FUNCTION` block as a table that can embed imperative SQLScript logic inside a single query.

Syntax

```
<from_clause> = FROM <table_from>
<table_from> = <table> | <table_from> ',' <table>
<table> = <basetable> | <subquery_with_parens> <opt_table_alias> |
<joined_table> | <tablesample>
<basetable> = <table_ref> <opt_table_alias> | ….. | <anonymous_function>
<opt_table_alias>
<anonymous_function> = SQL FUNCTION <anonymous_func_param_list> <func_return>
BEGIN <sqlscript_body> END
<anonymous_func_param_list> = (empty string) | '(' ')' |
'(' <anonymous_func_param> ')'
<anonymous_func_param> = <proc_param_mode> <proc_param_name> <proc_data_type>
ARG_ASSIGN_OP <proc_expr>
<func_return> = RETURNS <table_ref> | RETURNS TABLE
'(' <opt_cv_array_column_list> ')' | RETURNS proc_param_name func_data_type
```

Description

It is possible to create a one-time SQLScript function that can embed imperative SQLScript logic inside an SQL query. Earlier it was necessary to create an SQLScript function as a metadata object and consume it inside a single query. Similarly to the anonymous procedure block `DO BEGIN…END`, the `SQL FUNCTION RETURNS… BEGIN… END` block supports that kind of one-time table functions.

Example

User's Original Intention	Query with SQLScript TUDF	SQL Embedded SQLScript Table Function
`SELECT A, B, SUM(C) FROM (SELECT 1 as A, 2 as B, 3 as C FROM DUMMY UNION ALL SELECT 1 as A, 2 as B, 4 as C FROM DUMMY UNION ALL SELECT 2 as A, 3 as B, 2 as C FROM DUMMY UNION ALL SELECT 2 as A, 3 as B, 4 as C FROM DUMMY UNION ALL SELECT 2 as A, 5 as B, 7 as C FROM DUMMY) GROUP BY A, B ORDER BY A, B;`	`CREATE FUNCTION TEMP_FUNC() RETURNS TABLE (A INT, B INT, C INT) AS BEGIN DECLARE buffer TABLE (A INT, B INT, C INT); :buffer.insert((1, 2, 3)); :buffer.insert((1, 2, 4)); :buffer.insert((2, 3, 2)); :buffer.insert((2, 3, 4)); :buffer.insert((2, 5, 7)); RETURN :buffer; END; SELECT A, B, SUM(C) FROM TEMP_FUNC() GROUP BY A, B ORDER BY A, B;`	`SELECT A, B, SUM(C) FROM SQL FUNCTION RETURNS TABLE (A INT, B INT, C INT) BEGIN DECLARE buffer TABLE (A INT, B INT, C INT); :buffer.insert((1, 2, 3)); :buffer.insert((1, 2, 4)); :buffer.insert((2, 3, 2)); :buffer.insert((2, 3, 4)); :buffer.insert((2, 5, 7)); RETURN :buffer; END GROUP BY A, B ORDER BY A, B;`

Sample Code

```
select sum(a) from
sql function
returns table (a int, b int)
begin
  declare t table(a int, b int);
  :t.insert((1, 2));
  :t.insert((1, 3));
  :t.insert((2, 2));
  :t.insert((3, 3));
  return :t;
end

-- fails, because it is read-only
select a from
sql function
returns table (a int)
begin
  create column table temptable(a int);
  return select 1 as a from dummy;
end
```

Sample Code

```
-- input parameter
select a from
  sql function (in a int => 1)
  returns table (a int)
  begin
    return select :a as a from dummy;
  end;

-- nested SQL FUNCTION clause
select a from
  sql function
  returns table (a int)
  begin
    return select * from
      sql function
      returns table (a int)
      begin
        return select 1 as a from dummy;
      end;
  end;
```

Limitations

If the SQL FUNCTION clause is nested inside another SQLScript object, most of the SQLScript system variables are not available, if they are not defined as INPUT parameters.

- ROWCOUNT is not shared between the caller object and the SQL FUNCTION but it can still show the selected ROWCOUNT from the SELECT statement itself.
- SQL_ERROR_CODE and SQL_ERROR_MESSAGE are not inherited, although it is possible to define them explicitly within the SQL FUNCTION

6.2.9 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 functions, 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 is destroyed.

Sample Code

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

i 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

6.2.10 Procedure Result Cache

Procedure Result Cache (PRC) is a server-wide in-memory cache that caches the output arguments of procedure calls using the input arguments as keys.

Deterministic Procedure Cache is an automatic application of PRC for deterministic procedures.

i Note

Currently, PRC is enabled only for deterministic procedures.

Related Information

Deterministic Procedures [page 59]
Deterministic Procedure Cache [page 61]

6.2.10.1 Deterministic Procedures

Syntax

```
create procedure add (in a int, in b int, out c int) deterministic as begin
```

```
    c = :a + :b;
end
```

Description

You can use the keyword DETERMINISTIC when creating a new procedure, if the following conditions are met:

- The procedure always returns the same output arguments when it is called with the same input arguments, even if the session and database state is not the same.
- The procedure has no side effects.

You can also create a procedure with the keyword DETERMINISTIC, even if it does not satisfy the above conditions, by changing the configuration parameters described in the configuration section. Procedures created with the keyword DETERMINISTIC are described below as "deterministic procedures", regardless of whether they are logically deterministic or not.

By default, you cannot create a deterministic procedure that contains the following:

- Non-deterministic functions (for example, rand(), rand_secure(), session_context(), session_user, sysuuid)
- Statements with side effects (for example, implicit result sets, DML, DDL, commit/rollback/exec)
- Reading/writing persistence objects (for example, sequence)
- Invoking non-deterministic functions or procedures

You can skip the determinism check when creating deterministic procedures on your responsibility. It is useful when you want to create logically deterministic procedures that may contain non-deterministic statements. When disabling the check, please be aware that the cache can be shared among users, so if the procedure results depend on the current user (for example, the procedure security is invoker and there are user-specific functions or use of tables with analytic privileges), it may not behave as you expect. Disabling the check is not recommended.

If a deterministic procedure is logically non-deterministic, you may expect the following:

- If a deterministic procedure has side effects, the side effects may or may not be visible when you call the procedure.
- If a deterministic procedure has implicit result sets, they may or may not be returned when you call the procedure.
- If a deterministic procedure returns different output arguments for the same input arguments, you may or may not get the same output arguments when you call the procedure multiple times with the same input arguments.

Configuration

The configuration parameter below refers to Procedure Result Cache (PRC) under the section "sqlscript".

Name	Values	Default	Description
procedure_re-sult_cache_gc_interval	0-4294967295	60	Number of minutes between PRC garbage collection. When this value changes, the next GC will run after the specified minutes. Settings this value to 0 (not recom-mended) pauses the GC in-definitely, until a non-zero value is set.

Related Information

Procedure Result Cache [page 59]
Deterministic Procedure Cache [page 61]

6.2.10.2 Deterministic Procedure Cache

Description

By default Procedure Result Cache (PRC) is enabled for deterministic procedures.

The scope of the cache is the current server (for example, indexserver or cacheserver). If you call the same deterministic procedure in the same server with the same arguments multiple times, the cached results will be used except for the first call, unless the cached results are evicted. Since the cache is global in the current server, the results are shared even among different query plans.

> **i Note**
>
> Currently, only scalar parameters are supported for PRC. You can create deterministic procedures having table parameters, but automatic caching will be disabled for such procedures.

Deterministic Procedure Cache and Scalar UDF Result Cache

The same keyword, DETERMINISTIC, can be used for both procedures and functions, but currently the meaning is not the same.

For scalar user-defined functions, a new cache is created for each statement execution and destroyed after execution. The cache is local to the current statement which has a fixed snapshot of the persistence at a point in time. Due to this behavior, more things can be considered "deterministic" in deterministic scalar UDFs, such as reading a table.

Related Information

Procedure Result Cache [page 59]
Deterministic Procedures [page 59]

6.3 User-Defined Libraries

Syntax

Code Syntax

```
CREATE [OR REPLACE] LIBRARY <lib_name>
[LANGUAGE SQLSCRIPT] [DEFAULT SCHEMA <default_schema_name>]
AS BEGIN
  [<lib_var_decl_list>]
  [<lib_proc_func_list>]
END;
```

```
ALTER LIBRARY <lib_name>
[LANGUAGE SQLSCRIPT] [DEFAULT SCHEMA <default_schema_name>]
AS BEGIN
  [<lib_var_decl_list>]
  [<lib_proc_func_list>]
END;
```

```
DROP LIBRARY <lib_name>;
```

```
<lib_name> ::= [<schema_name>.]<identifier>;
```

```
<lib_var_decl_list> ::= <lib_var_decl> [{<lib_var_decl>}...]
<lib_var_decl> ::= <access_mode> <var_decl> ;
<var_decl> ::= VARIABLE <member_name> [CONSTANT] <sql_type> [NOT NULL]
[<proc_default>]

<access_mode> ::= PUBLIC | PRIVATE
<member_name> ::= <identifier>
<proc_default> ::= { DEFAULT | '=' } <expression>
```

```
<lib_proc_func_list> ::= <lib_proc_func> [{<lib_proc_func>}...]
<lib_proc_func> ::= <access_mode> <proc_func_def> ;
<proc_func_def> ::= <proc_def> | <func_def>

<proc_def> ::= PROCEDURE <member_name> [<parameter_clause>] [<proc_property>]
AS BEGIN [SEQUENTIAL EXECUTION] <procedure_body> END
<proc_property> ::= [LANGUAGE <lang>] [SQL SECURITY <mode>] [READS SQL DATA]

<func_def> ::= FUNCTION <member_name> [<parameter_clause>] RETURNS
<return_type> [<func_property>] AS BEGIN <function_body> END
<func_property> ::= [LANGUAGE <lang>] [SQL SECURITY <mode>] [READS SQL DATA]
```

Description

A library is a set of related variables, procedures and functions. There are two types of libraries: built-in libraries and user-defined libraries. A built-in library is a system-provided library with special functions. A user-defined library is a library written by a user in SQLScript. Users can make their own libraries and utilize them in other procedures or functions. Libraries are designed to be used only in SQLScript procedures or functions and are not available in other SQL statements.

A user-defined library has the following characteristics:

- A single metadata object is created for multiple procedures and functions. By combining all relevant procedures and functions into a single metadata object, you reduce metadata management cost. On the other hand, if one function or a procedure of the library becomes invalid, the whole library becomes invalid.
- The atomicity of the relevant objects is guaranteed because they are managed as a single object.
- It is easy to handle the visibility of a procedure or a function in a library. When an application gets bigger and complex, developers might want to use some procedures or functions only in their application and not to open them to application users. A library can solve this requirement easily by using the access modes PUBLIC and PRIVATE for each library member.
- Constant and non-constant variables are available in a library. You can declare a constant variable for a frequently used constant value and use the variable name instead of specifying the value each time. A non-constant value is alive during a session and you can access the value at any time if the session is available.

> **i Note**
>
> Any user having the EXECUTE privilege on a library can use that library by means of the USING statement and can also access its public members.

Limitations

The following limitations apply currently:

- The usage of library variables is currently limited. For example, it is not possible to use library variables in the INTO clause of a SELECT INTO statement and in the INTO clause of dynamic SQL. This limitation can be easily circumvented by using a normal scalar variable as intermediate value.
- It is not possible to call library procedures with hints.
- Since session variables are used for library variables, it is possible (provided you the necessary privileges) to read and modify arbitrary library variables of (other) sessions.
- Variables cannot be declared by using LIKE for specifying the type.
- Non-constant variables cannot have a default value.
- The table type library variable is not supported.
- A library member function cannot be used in queries.

Related Information

Library Members [page 64]

System Views [page 66]

6.3.1 Library Members

Syntax

Code Syntax

Using a Library Member

```
<procedure_body> ::= [<proc_using_list>] [<proc_handle_list>] <proc_stmt_list>
<proc_using_list> ::= {<proc_using>}...
<proc_using> ::= USING <lib_name> AS <lib_alias> ;
<lib_name> ::= [<schema_name>.]<identifier>
<lib_alias> ::= <identifier>
<lib_member_ref> ::= [ <schema_name> . ] <identifier> ':' <member_name>


<proc_assign> ::= <variable_name> = { <expression> | <array_function> |
<lib_member_func_call>} ;
                | <variable_name> '[' <expression> ']' = { <expression> |
<lib_member_func_call> } ;
                | <lib_member_ref> = { <expression> |
<lib_member_func_call> } ;
<lib_member_func_call> ::= <lib_member_ref> ( [<expression> [ {,
<expression> }...] ] )


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

Description

Access Mode

Each library member can have a PUBLIC or a PRIVATE access mode. PRIVATE members are not accessible outside the library, while PUBLIC members can be used freely in procedures and functions.

Library Member Variable

The scope of a library member variable is bound to its session. The value of a library variable persists throughout a session. If the variable is accessed by different statements within the same session, these statements access the same variable. However, a library member variable can display different values if accessed from different sessions.

Library member variables support the following primitive data types:

Boolean Type	BOOLEAN
Numeric Types	TINYINT SMALLINT INT BIGINT DECIMAL SMALLDECIMAL REAL DOUBLE
Character String Types	VARCHAR NVARCHAR ALPHANUM
Date-Time Types	TIMESTAMP SECONDDATE DATE TIME

Library Member Functions and Procedures

Library functions and procedures can be declared as private or public. Private functions and procedures are for internal use within the library. They cannot be called from outside the library. Public functions and procedures can be used by anyone who has the EXECUTE privilige for the library. These functions and procedures can be used and declared like non-library functions and procedures, but they have access to the library private variables, private functions and private procedures. It is also possible to call procedures and functions from outside the library, as well as other libraries. The use of library functions is limited to the right-hand side of assignments and cannot be used in queries.

Resolving Unqualified Names

A library member is not a metadata object, so it may have the same name as another procedure or function. When resolving an unqualified name in a library definition, the system first examines library members defined before the current library member. If the name is not found within the library, then the name is searched for in the library schema. To reduce ambiguity and to avoid duplicate names, it is recommended to use a fully qualified name for user-defined functions.

Example

Sample Code

Setup

```
create table data_table(col1 int);

do begin
  declare idx int = 0;
  for idx in 1..200 do
    insert into data_table values (:idx);
  end for;
end;
```

Sample Code

Library DDL

```
create library mylib as begin
  public variable maxval constant int = 100;

  public function bound_with_maxval(i int) returns x int as begin
    x = case when :i > :maxval then :maxval else :i end;
  end;

  public procedure get_data(in size int, out result table(col1 int)) as begin
    result = select top :size col1 from data_table;
```

```
  end;
end;
```

Sample Code

Procedure Using Library

```
create procedure myproc (in inval int) as begin
  using mylib as mylib;
  declare var1 int = mylib:bound_with_maxval(:inval);

  if :var1 > mylib:maxval then
    select 'unexpected' from dummy;
  else
    declare tv table (col1 int);
    call mylib:get_data(:var1, tv);
    select count(*) from :tv;
  end if;
end;
```

Sample Code

Result

```
call myproc(10);
Result:
count(*)
10
```

```
call myproc(150);
Result:
count(*)
100
```

Related Information

User-Defined Libraries [page 62]
System Views [page 66]

6.3.2 System Views

System views for user-defined libraries.

LIBRARIES

LIBRARIES shows available libraries.

Column name	Column description
SCHEMA_NAME	Schema name of the library
LIBRARY_NAME	Name of the library
LIBRARY_OID	Object ID of the library
OWNER_NAME	Owner name of the library
DEFAULT_SCHEMA_NAME	Schema of the unqualified objects in the library
DEFINITION	Definition of the library
LIBRARY_TYPE	Language type of the library
IS_VALID	Specifies whether the library is valid or not. This becomes false when its base objects are changed or dropped.
CREATE_TIME	Creation time

LIBRARY_MEMBERS

Library members of SQLScript libraries.

Column name	Column description
SCHEMA_NAME	Schema name of the library
LIBRARY_NAME	Name of the library
LIBRARY_OID	Object ID of the library
MEMBER_NAME	Name of the library member
MEMBER_TYPE	Type of the library member: 'VARIABLE', 'PROCEDURE', 'FUNCTION'
ACCESS_MODE	Access mode of the library member: 'PUBLIC', 'PRIVATE'
DEFINITION	Definition string of the library member

Related Information

User-Defined Libraries [page 62]
Library Members [page 64]

6.3.3 UDL Member Procedure Call Without SQLScript Artifacts

Description

Until now it was possible to use library members of user-defined libraries (UDL) only within the scope of other SQLScript objects like procedures, functions or anonymous blocks. For example, even if you only wanted to run a single library member procedure, you had to create a procedure or execute the member procedure within an anonymous block. Wrapping the member access into an anonymous block is simple when there are no parameters, but it can get more complex, if there are input and output parameters. You can now directly call library member procedures without the use of additional SQLScript objects.

Syntax

Code Syntax

```
<call_stmt> ::= CALL <proc_name> ( <param_list> ) [WITH OVERVIEW] [IN DEBUG
MODE]
              | CALL <lib_member_ref> ( <param_list> );

<proc_call> ::= CALL <proc_name> ( <param_list> ) ;
              | CALL <lib_member_ref> ( <param_list> ) ;

<lib_member_ref> ::= [<schema_name> '.'] <library_name_or_alias> ':'
<member_name>

<schema_name> ::= <identifier>
<library_name_or_alias> ::= <identifier>
<member_name> ::= <identifier>
```

Behavior

Old Behavior	New Behavior

`create library mylib as begin`
`  public procedure memberproc(in i int, out tv table(col1 nvarchar(10)))`
`as begin`
`    tv = select :i * 100 as col1 from dummy;`
`  end;`
`end;`
 	
`do (in iv int => 1, out otv table(col1 nvarchar(10)) => ?) begin`
`  using mylib as mylib;`
`  call mylib:memberproc(:iv, otv);`
`end;`
	`create library mylib as begin`
	`  public procedure memberproc(in i int, out tv table(col1 nvarchar(10)))`
	`as begin`
	`    tv = select :i * 100 as col1 from dummy;`
	`  end;`
	`end;`
	
	`call mylib:memberproc(1, ?);`

Library members can be referenced by library name and library member name. If a library alias is set by a USING statement, the alias can be used instead of the library name.

If an alias is specified, SQLScript first tries to resolve the unqualified library name as a library alias. If the name is not found in the list of library aliases, then SQLScript will resolve the name with a default schema. However, if a schema name is specified, the library is always searched for inside the schema and any existing alias is ignored.

Examples

Sample Code

Example Library

```
create schema myschema1;
create schema myschema2;

create library myschema1.mylib as begin
  public procedure memberproc (out ov varchar(10)) as begin
    ov = 'myschema1';
  end;
end;

create library myschema2.mylib as begin
  public procedure memberproc (out ov varchar(10)) as begin
    ov = 'myschema2';
  end;
end;
```

Sample Code

Example 1

```
create or replace procedure myproc1 (out ov varchar(10))
default schema myschema2
```

```
as begin
  using myschema1.mylib as mylib;
  call mylib:memberproc(ov);
end;

call myproc1(?); -- result: 'myschema1'
```

In this example, the library name in the CALL statement is not fully qualified and there is an alias with the same name. In that case, `mylib` is resolved as library `mylib` and it refers to `myschema1.mylib`.

Sample Code

Example 2

```
create or replace procedure myproc2 (out ov varchar(10))
default schema myschema2
as begin
  call mylib:memberproc(ov);
end;

call myproc2(?); -- result: 'myschema2'
```

In this example, the library name in the CALL statement is not fully qualified and there is no alias with the same name. In that case, `mylib` is found only in the default schema and refers to `myschema2.mylib`.

Sample Code

Exaple 3

```
create or replace procedure myproc3 (out ov varchar(10))
as begin
  using myschema1.mylib as mylib;
  call myschema2.mylib:memberproc(ov); -- Resolved as myschema2 because the
schema is explicitly described.
end;

call myproc3(?); -- result: 'myschema2'
```

In this example, the library name in the CALL statement is `mylib` and there is an alias with the same name. However, the library name is fully qualified with the schema name `myschema2` and is resolved as `myschema2.mylib`.

Limitations

The following limitations apply:

- WITH option is not supported for library member CALL statement. For example CALL MYLIB:PROC() WITH HINT (...)
- EXPLAIN PLAN is not supported.
- QUERY EXPORT is not supported.
- Built-in library member procedures with variable arguments are not supported.

6.3.4 Library Member Functions and Variables

Library member functions and variables can be used directly in SQL or expressions in SQLScript.

Syntax

The syntax for library table functions, scalar functions and variables accepts a library member reference.

Code Syntax

```
<expression> ::= <case_expression> | <function_expression> | ... |
<variable_name> | ...
<function_expression> ::= <function_name> ( <expression> [{,
<expression>} ...])
<function_name> ::= [[ <database_name> '.' ] <schema_name> '.' ]]
<identifier> | <lib_member_ref>
<variable_name> ::= <identifier> | <lib_member_ref>


<from_clause> ::= FROM <table_expression> [, <table_expression> ...]
<table_expression> ::= <table_ref> | ... | <function_reference> | ...
<function_reference> ::= <function_name> ( <proc_arg_list> |
<opt_parameter_key_value_list> )


<lib_member_ref> ::= [<schema_name> '.' ] <library_name_or_alias> ':'
<member_name>


<schema_name> ::= <identifier>
<library_name_or_alias> ::= <identifier>
<member_name> ::= <identifier>
```

Behavior

Sample Code

```
create table r_tab (r decimal);
insert into r_tab values (50);
insert into r_tab values (100);


create library mylib as begin
  public variable phi constant decimal = 3.14;
  public function circumference(r decimal) returns a int as begin
    a = 2 * :phi * :r;
  end;
  public function circumference_table(r_table table(r decimal)) returns
table(c decimal) as begin
    return select 2 * :phi * r as c from :r_table;
  end;
end;
```

Old Behavior	New Behavior
`select mylib:phi from dummy;`	`select mylib:phi from dummy;`
ERR-00467: cannot use parameter variable: MYLIB:PHI: line 1 col 8 (at pos 7)	Succeed: [(3.14)]
`select mylib:circumference(r) from r_tab;`	`select mylib:circumference(r) from r_tab;`
ERR-00007: feature not supported: using library member function on the outer boundary of SQLScript: CIRCUMFER-ENCE: line 1 col 8 (at pos 7)	Succeed: [(314), (628)]
`select * from mylib:circumference_table(r_tab);`	`select * from mylib:circumference_table(r_tab);`
ERR-00257: sql syntax error: incorrect syntax near "(": line 1 col 40 (at pos 40)	Succeed: [(314), (628)]

Limitations

- EXPLAIN PLAN is not supported.
- QUERY EXPORT is not supported.
- Built-in library member functions with variable arguments are not supported.
- Library member functions and variables are not supported in generated columns and table check conditions.
- PRIVATE functions are not supported in SQL.
- Library member variable is not supported in DDL.

Related Information

UDL Member Procedure Call Without SQLScript Artifacts [page 68]

6.4 CREATE OR REPLACE

When creating a SQLScript procedure or function, you can use the OR REPLACE option to change the defined procedure or function, if it already exists.

Syntax

```
CREATE [OR REPLACE] FUNCTION <func_name> [(<parameter_clause>)] RETURNS
<return_type>
[LANGUAGE <lang>] [SQL SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name> ]
[READS SQL DATA] [<variable_cache_clause>] [ DETERMINISTIC ]
[WITH ENCRYPTION]
AS
BEGIN
     <function_body>
END
[ <cache_clause> ]
<cache_clause> ::=
   WITH [ STATIC ] CACHE
   RETENTION <minute_value>
   [ OF <projection_list> ]
   [ FILTER <filter_condition> ]
   [ <location_clause> ]
   [ FORCE ]
CREATE [OR REPLACE] PROCEDURE <proc_name> [(<parameter_clause>)] [LANGUAGE
<lang>] [SQL SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name>]
 [READS SQL DATA ] [<variable_cache_clause>] [ DETERMINISTIC ] [WITH ENCRYPTION]
[AUTOCOMMIT DDL { ON|OFF } ]
 AS
 { BEGIN [ SEQUENTIAL EXECUTION | PARALLEL EXECUTION ]
  <procedure_body>
 END | HEADER ONLY }
```

Behavior

The behavior of this command depends on the existence of the defined procedure or function. If the procedure or the function already exist, it will be modified according to the new definition. If you do not explicitly specify a property (for example, read only), this property will be set to the default value. Please refer to the example below. If the procedure or the function do not exist yet, the command works like CREATE PROCEDURE or CREATE FUNCTION.

Compared to using DROP PROCEDURE followed by CREATE PROCEDURE, CREATE OR REPLACE has the following benefits:

- DROP and CREATE incur object re-validation twice, while CREATE OR REPLACE incurs it only once
- If a user drops a procedure, its privileges are lost, while CREATE OR REPLACE preserves them.

Example

Sample Code

```
create or replace procedure proc(out o table(a int))
default schema system reads sql data deterministic with encryption as
begin
    o = select 1 as a from dummy;
end;
call proc(?);
-- Returns 1
create or replace procedure proc(out o table(a int))
language llang as
begin
    export Void main(Table<Int32 "A"> "o" & o)
    {
        Column<Int32> col = o.getColumn<Int32>("A");
        col.setElement(0z, 2);
    }
end;
call proc(?);
-- Returns 2
-- Note that this procedure is not set to read-only, deterministic,
encrypted, or default schema system any more.
create or replace procedure proc(out o int) as
begin
    o = 3;
end;
-- Returns an error because the signature of the new procedure does not match
to that of the predefined procedure
```

Sample Code

```
CREATE OR REPLACE PROCEDURE test1 as
begin
    select * from dummy;
end;
call test1;

-- new parameter
CREATE OR REPLACE PROCEDURE test1 (IN i int) as
begin
    select :i from dummy;
    select * from dummy;
end;
call test1(?);

-- default value
CREATE OR REPLACE PROCEDURE test1 (IN i int default 1) as
begin
    select :i from dummy;
end;
call test1();

-- change the number of parameter and name of parameter
ALTER PROCEDURE test1 (j int, k int) as
begin
    select :j from dummy;
    select :k from dummy;
end;
call test1(?, ?);

-- change the type of the parameter and name of parameter
CREATE OR REPLACE PROCEDURE test1 (t1 TIMESTAMP, t2 TIMESTAMP) as
```

```
begin
    select :t1 from dummy;
    select :t2 from dummy;
end;
call test1(?, ?);

-- support also ddl command 'ALTER'
ALTER PROCEDURE test1 as
begin
    select * from dummy;
end;
call test1;

-- table type
create column table tab1 (a INT);
create column table tab2 (a INT);

CREATE OR REPLACE PROCEDURE test1(out ot1 table(a INT), out ot2 table(a INT))
as begin
    insert into tab1 values (1);
    select * from tab1;
    insert into tab2 values (2);
    select * from tab2;
    insert into tab1 values (1);
    insert into tab2 values (2);
    ot1 = select * from tab1;
    ot2 = select * from tab2;
end;
call test1(?, ?);

-- change the number of parameter
ALTER PROCEDURE test1(out ot1 table(a INT)) as begin
    insert into tab1 values (1);
    select * from tab1;
    insert into tab2 values (2);
    select * from tab2;
    insert into tab1 values (1);
    insert into tab2 values (2);
    ot1 = select * from tab1;
end;
call test1(?);

-- security
CREATE OR REPLACE PROCEDURE test1(out o table(a int))
sql security invoker as
begin
    o = select 5 as a from dummy;
end;
call test1(?);

-- change security
ALTER PROCEDURE test1(out o table(a int))
sql security definer as
begin
    o = select 8 as a from dummy;
end;
call test1(?);

-- result view
ALTER PROCEDURE test1(out o table(a int))
reads sql data with result view rv1 as
begin
    o = select 0 as A from dummy;
end;
call test1(?);

-- change result view
CREATE OR REPLACE PROCEDURE test1 (out o table(a int))
```

```
reads sql data with result view rv2 as
begin
    o = select 1 as A from dummy;
end;
call test1(?);

-- table function
CREATE TYPE TAB_T1 AS TABLE(a int);

CREATE OR REPLACE FUNCTION func1()
returns TAB_T1 LANGUAGE SQLSCRIPT
as begin
    return select * from TAB1;
end;
select * from func1();

CREATE OR REPLACE FUNCTION func1(a int)
returns table(a INT) LANGUAGE SQLSCRIPT
as begin
    if a > 4
    then
        return select * from TAB1;
    else
        return select * from TAB2;
    end if;
end;
select * from func1(1);

-- scalar function
CREATE OR REPLACE FUNCTION sfunc_param returns a int as
begin
    A = 0;
end;
select sfunc_param() from dummy;

CREATE OR REPLACE FUNCTION sfunc_param (x int) returns a int as
begin
    A = :x;
end;
select sfunc_param(3) from dummy;
```

6.5 Procedure and Function Headers

When you have a procedure or a function that already exist and you want to create a new procedure consuming them, to avoid dependency problems you can use headers in their place.

When you create a procedure, all nested procedures that belong to that procedure must exist beforehand. If the procedure `P1` calls `P2` internally, then `P2` must have been created earlier than `P1`. Otherwise, the creation of `P1` 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 that kind of dependency problems, SAP introduces `HEADERS`. `HEADERS` allow you to create a minimum set of metadata information that contains only the interface of a procedure or a 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 create 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 in a similar way.

```
CREATE FUNCTION <func_name> [(<parameter_clause>)] RETURNS <return_type> AS
HEADER ONLY
```

By checking the `is_header_only` field in the system view `PROCEDURES`, you can verify that a header-only procedure 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 a function is defined, the 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 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 a function from the header definition, you need to 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>]
[DEFAULT SCHEMA <default_schema_name>]
[READS SQL DATA] [<variable_cache_clause>] [ DETERMINISTIC ] [WITH ENCRYPTION]
[AUTOCOMMIT DDL { ON|OFF } ]  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> [(<parameter_clause>)] RETURNS <return_type>
```

```
[LANGUAGE <lang>] [SQL SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name> ]
[READS SQL DATA] [<variable_cache_clause>]
[DETERMINISTIC][WITH ENCRYPTION]
AS
BEGIN
     <function_body>
END
```

For example, if you want to replace the header definition of `TEST_PROC` that has already been defined, the `ALTER` statement will 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
```

6.6 Anonymous Block

An 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, anonymous blocks have no corresponding object created in the metadata catalog - they are cached in the SQL Plan Cache.

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

```
DO [(<parameter_clause>)]
BEGIN [SEQUENTIAL EXECUTION]
    <body>
END WITH HINT (...)
<body> ::= !! supports the same feature set as the procedure
```

For more information, see the CREATE PROCEDURE statement in the SAP HANA SQL and System Views Reference on the SAP Help Portal.

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>
```

> **i 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 a 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, otherwise the query parameter cannot be bound. Any scalar expression can be used for the scalar input parameter.

You can also parameterize the scalar parameters, if needed. For example, for the example above, 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.

> **i Note**
>
> An anonymous block cannot be used in a procedure or in a function.

It is now possible to use HINTs for anonymous blocks. However, not all hints that are supported for CALL, are also supported for anonymous blocks (for example, routing hints).

> **Sample Code**
>
> Anonymous Block Hint
>
> ```
> DO BEGIN
>   DECLARE i INT;
>   FOR i in 1..5 DO
>     SELECT * FROM dummy;
>   END FOR;
> END WITH HINT(ignore_plan_cache)
> ```

Below you find further examples of 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
```

6.7 SQLScript Encryption

Procedure and function definitions may contain delicate or critical information but a user with system privileges can easily see all definitions from the public system views PROCEDURES, FUNCTIONS or from traces, even if the procedure or function owner has controlled the authorization rights in order to secure their objects. If application developers want to protect their intellectual property from any other users, even system users, they can use SQLScript encryption.

> **Note**
>
> Decryption of an encrypted procedure or function is **not** supported and cannot be performed even by SAP. Users who want to use encrypted procedures or functions are responsible for saving the original source code and providing supportability because there is no way to go back and no supportability tools for that purpose are available in SAP HANA.

Syntax

> Code Syntax

```
[CREATE | ALTER] PROCEDURE <proc_name> [(<parameter_clause>)]
```

```
    [LANGUAGE <lang>] [SQL SECURITY <mode>] [DEFAULT SCHEMA
    <default_schema_name>] [READS SQL DATA ]
    [<sqlscript_route_option>]
    [WITH ENCRYPTION]
AS BEGIN
    ...
END;
```

Code Syntax

```
[CREATE | ALTER] FUNCTION <func_name> [(<parameter_clause>)] RETURNS
<return_type>
    [LANGUAGE <lang>] [SQL SECURITY <mode>] [DEFAULT SCHEMA
<default_schema_name>] [READS SQL DATA]
    [<sqlscript_route_option>] [DETERMINISTIC]
    [WITH ENCRYPTION]
AS BEGIN
    ...
END;
```

Code Syntax

```
ALTER PROCEDURE <proc_name> ENCRYPTION ON;
ALTER FUNCTION <func_name> ENCRYPTION ON;
```

Behavior

If a procedure or a function is created by using the WITH ENCRYPTION option, their definition is saved as an encrypted string that is not human readable. That definition is decrypted only when the procedure or the function is compiled. The body in the CREATE statement is masked in various traces or monitoring views.

Encrypting a procedure or a function with the ALTER PROCEDURE/FUNCTION statement can be achieved in the following ways. An ALTER PROCEDURE/FUNCTION statement, accompanying a procedure body, can make use of the WITH ENCRYPTION option, just like the CREATE PROCEDURE/FUNCTION statement.

If you do not want to repeat the procedure or function body in the ALTER PROCEDURE/FUNCTION statement and want to encrypt the existing procedure or function, you can use `ALTER PROCEDURE/FUNCTION <proc_func_name> ENCRYPTION ON`. However, the CREATE statement without the WITH ENCRYPTION property is not secured.

i Note

A new encryption key is generated for each procedure or function and is managed internally.

SQLScript Debugger, PlanViz, traces, monitoring views, and others that can reveal procedure definition are not available for encrypted procedures or functions.

Additional Considerations

Nested Procedure Call

Not encrypted procedures or functions can be used inside encrypted procedures or functions. However, encryption in the outer call does not mean that nested calls are also secured. If a nested procedure or a function is not encrypted, then its compilation and execution details are available in monitoring views or traces.

Object Dependency

The object dependency of encrypted procedures or functions is not secured. The purpose of encryption is to secure the logic of procedures or functions and object dependency cannot reveal how a procedure or a function works.

Criteria What to Hide

There is a large amount of information related to a procedure or a function and hiding all information is hard and makes problem analysis difficult. Therefore, compilation or execution information, which cannot reveal the logic of a procedure or a function, can be available to users.

Limitation in Optimization

Some optimizations, which need analysis of the procedure or function definition, are turned off for encrypted procedures and functions.

Calculation Views

An encrypted procedure cannot be used as a basis for a calculation view. It is recommended to use table user-defined functions instead.

System Views

An additional column IS_ENCRYPTED is added to the views PROCEDURES and FUNCTIONS.

PROCEDURES

SCHEMA_NAME	PROCEDURE_NAME	...	IS_ENCRYPTED	DEFINITION
SYSTEM	TEST_PROC	...	TRUE	CREATE PROCEDURE TEST_PROC(IN x INT) <encrypted_definition>

FUNCTIONS

SCHEMA_NAME	FUNCTION_NAME	...	IS_ENCRYPTED	DEFINITION
SYSTEM	TEST_FUNC	...	TRUE	CREATE FUNCTION TEST_FUNC(IN x INT) RETURNS i <encrypted definition>

For every public interface that shows procedure or function definitions, such as PROCEDURES or FUNCTIONS, the definition column displays only the signature of the procedure, if it is encrypted.

Sample Code

```
CREATE PROCEDURE TEST_PROC(IN x INT) WITH ENCRYPTION AS BEGIN
  SELECT 1 AS I FROM DUMMY;
END;
CREATE FUNCTION TEST_FUNC(IN x INT) RETURNS i INT WITH ENCRYPTION AS BEGIN
  i = 1;
END;
```

System View PROCEDURES

Sample Code

```
SELECT PROCEDURE_NAME, DEFINITION FROM PROCEDURES WHERE PROCEDURE_NAME =
'TEST_PROC';
```

Result:

PROCEDURE_NAME	DEFINITON
TEST_PROC	CREATE PROCEDURE TEST_PROC(IN x INT) <encrypted definition>

System View FUNCTIONS

Sample Code

```
SELECT FUNCTION_NAME, DEFINITION FROM FUNCTIONS WHERE FUNCTION_NAME =
'TEST_FUNC';
```

Result:

FUNCTION_NAME	DEFINITON
TEST_FUNC	CREATE FUNCTION TEST_FUNC(IN x INT) RETURNS i INT <encrypted definition>

Supportability

For every monitoring view showing internal queries, the internal statements will also be hidden, if its parent is an encrypted procedure call. Debugging tools or plan analysis tools are also blocked.

The following supportability tools are blocked:

- SQLScript Debugger
- EXPLAIN PLAN FOR Call
- PlanViz

The following views display less information:

- Statement-related views
- Plan Cache-related views
- M_ACTIVE_PROCEDURES

In these monitoring views, the SQL statement string is replaced with the string <statement from encrypted procedure <proc_schema>.<proc_name> (<sqlscript_context_id>)>.

6.7.1 Import and Export of Encrypted SQLScript Objects

Default Behavior

Encrypted procedures or functions cannot be exported, if the option ENCRYPTED OBJECT HEADER ONLY is not applied. When the export target is an encrypted object or if objects, which are referenced by the export object, include an encrypted object, the export will fail with the error FEATURE_NOT_SUPPORTED. However, when exporting a schema and an encrypted procedure or function in the schema does not have any dependent objects, the procedure or function will be skipped during the export.

With the Option ENCRYPTED OBJECT HEADER ONLY

To enable export of any other objects based on an encrypted procedure, the option ENCRYPTED OBJECT HEADER ONLY is introduced for the EXPORT statement. This option does not export encrypted objects in encrypted state, but exports the encrypted object as a header-only procedure or function. After an encrypted procedure or a function has been exported with the HEADER ONLY option, objects based on encrypted objects will be invalid even after a successful import. You should alter the exported header-only procedure or function to its original body or dummy body to make dependent objects valid.

Sample Code

Original Procedure

```
create procedure enc_proc with encryption as
begin
  select 1 as i from dummy;
end;
```

Sample Code

Export Statement

```
export all as binary into <path> with encrypted object header only;
```

Sample Code

Exported create.sql

```
create procedure enc_proc /* WITH ENCRYPTION */ AS HEADER ONLY;
```

7 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.

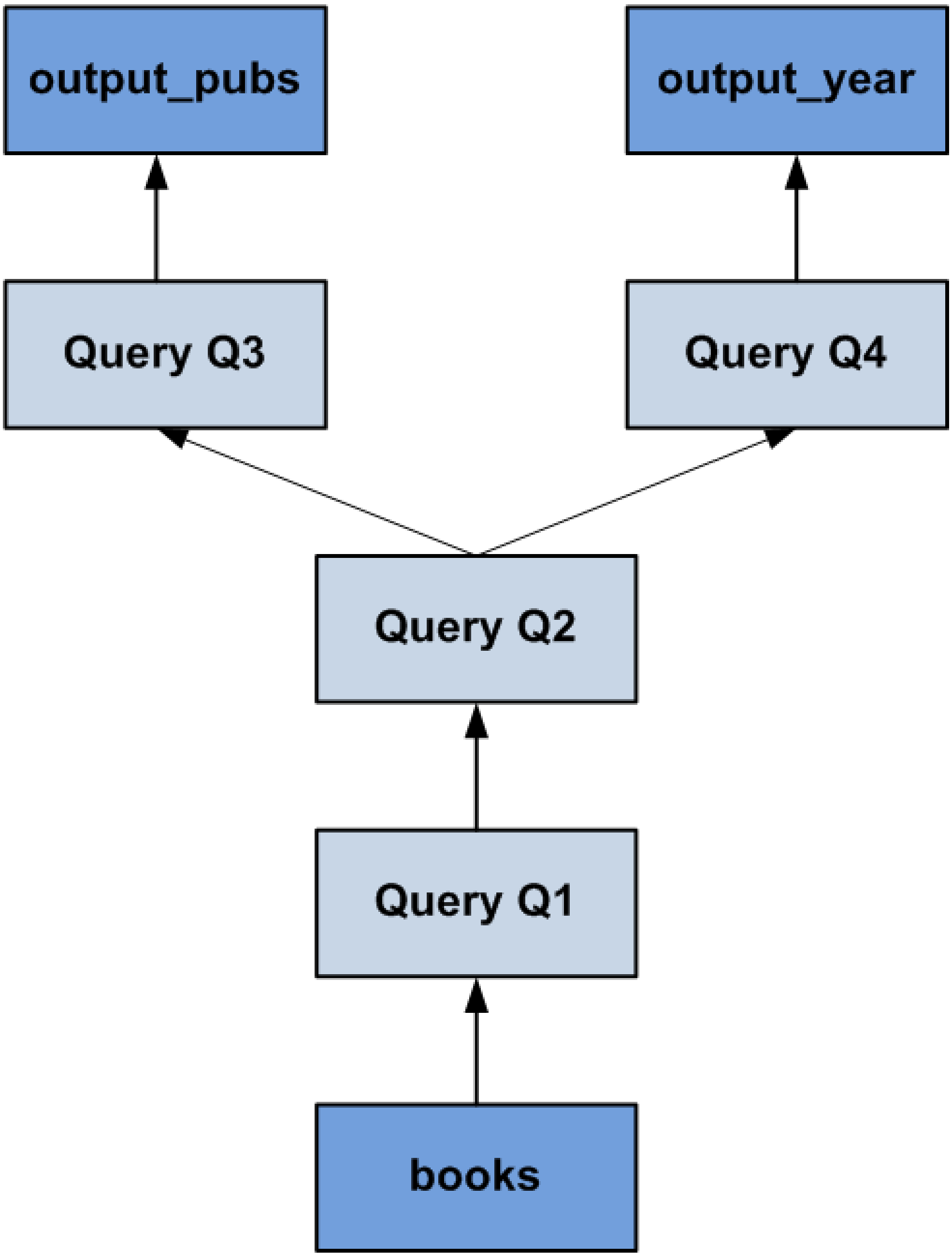
This approach leads to data flows which are free of side effects. The declarative nature to define business logic might require some deeper thought when specifying an algorithm, but it gives the SAP HANA database freedom to optimize the data flow which may result in better performance.

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.


output_pubs
output_year
Query Q3
Query Q4
Query Q2
Query Q1
books

7.1 Table Parameter

Syntax

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

Description

Table parameters that are defined in the signature are either input or output parameters. The parameters can be typed 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.

7.1.1 Any Table Type Parameter

The any table type parameter is a table parameter whose type is defined during DDL time as a wildcard and is determined later during query compilation.

Syntax

As a result of the new any table type support, the syntax of table parameters has changed as follows:

Code Syntax

```
<table_param> ::= [IN|OUT] <param_name> {<table_type>|<table_type_definition>|
<any_table_type>}
<any_table_type> ::= TABLE(...)
```

Examples

The following examples illustrate some use cases of the `any_table_type` parameter for DML and SELECT statements.

Sample Code

```
create procedure myproc1(out ott table(...)) as
begin
    ott = select * from ctab1;
end;

-- use of nested call statements inside a procedure
drop procedure myproc1;
create procedure myproc1(in itt table(...), out ott table(c int)) as
begin
    ott = select * from :itt;
end;

drop procedure myproc2;
create procedure myproc2 as
begin
    it0 = select 1 c from ctab3;
    call myproc1(:it0, :ott);
end;

-- nested call with any table parameters
drop procedure subproc1;
create procedure subproc1 (in itt table(...)) as
begin
    ott = select * from ctab1;
end;

drop procedure subproc2;
create procedure subproc2(in itt table(...)) as
begin
    call subproc1(:itt);
end;
```

```
create procedure myproc2(in itt table(...)) as
begin
    lt0 = select * from :itt;
    lt1 = select * from :lt0;
    select * from :lt1, ctab1;
end;
```

The `any_table_type` parameter can also be used in other scenarios with different statements.

Sample Code

```
-- unnest statement
create procedure unst_proc1(in itt table(a int), out ott table(...)) as
begin
    tmp = SELECT '1','2','3' as A from :itt;
    tmp2 = unnest(ARRAY_AGG(:tmp.a));
    ott = select * from :tmp2;
end;

call unst_proc1(ctab1,?);

-- ce functions
create procedure ce_proc1 (out outtab table(...)) as
begin
    t = ce_column_table(temptable);
    outtab = ce_projection(:t, [b]);
end
call ce_proc1(?);

-- apply filters
CREATE PROCEDURE apply_p1(IN inputtab table(...), IN dynamic_filter_1
VARCHAR(5000)) as
begin
  outtab = APPLY_FILTER (:inputtab, :dynamic_filter_1);
  select * from :outtab;
end;

call apply_p1(ctab3, ' a like ''%fil%'' ');
call apply_p1(ctab3, ' a =  ''
```

Scope and Limitations

The `any_table_type` parameter can be used in procedures and table UDFs in the SQLScript laguage and procedures in the AFL language with some limitations:

- the `any_table_type` parameter cannot be used within anonymous blocks, other languages or outside the scope of SQLScript
- `any_table_type` parameters are supported only as input parameter of table UDFs, but not as return parameters
- scalar UDFs do not support `any_table_type` parameters.
- If an output any table type parameter cannot be resolved during procedure creation (for example, `out_any_table = select * from in_any_table`), the procedure cannot be called inside SQLScript.

7.2 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. An explicitly declared table variable is initialized with empty content if a default value is not assigned.

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 if 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;
```

An alternative way to declare a table variable is to use the LIKE keyword. You can specify the variable type by using the type of a persistent table, a view, or another table variable.

```
DECLARE <list_of_variable_names> [CONSTANT] LIKE { <table_name>
| :<table_variable_name> }.<column_name> [NOT NULL] [default_value]
DECLARE <list_of_variable_names> [CONSTANT] TABLE LIKE { <table_name>
| :<table_variable_name> } [default_value]
```

> **i Note**
>
> When you declare a table variable using LIKE <table_name>, all the attributes of the columns (like unique, default value, and so on) in the referenced table are ignored in the declared variable except the `not null` attribute.
>
> When you use LIKE <table_name> to declare a variable in a procedure, the procedure will be dependent on the referenced 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 93]. 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_TABLE_VARIABLE_USAGE. Besides issuing a warning, it also offers the following 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:

	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 performmed only if a table variable is used.

NOT NULL Constraint

You can specify the NOT NULL constraint on columns in table types used in SQLScript. Historically, this was not allowed by the syntax and existing NOT NULL constraints on tables and table types were ignored when used as types in SQLScript. Now, NOT NULL constraints are taken into consideration, if specified directly in the column list of table types. NOT NULL constraints in persistent tables and table types are still ignored by default for backward compatibility but you can make them valid by changing the configuration, as follows:

- Global: `indexserver.ini (sqlscript, not_null_column_mode) = 'ignore' (default), 'ignore_with_warning', 'respect'`
- Session variable: `set '__SQLSCRIPT_NOT_NULL_COLUMN_MODE' = 'ignore' (default), 'ignore_with_warning', 'respect'`

If both are set, the session variable takes precedence. Setting it to `'ignore_with_warning'` has the same effect as `'ignore'`, except that you additionally get a warning whenever the constraint is ignored. With `'respect'`, the NOT NULL constraints (including primary keys) in tables and table types will be taken into consideration but that could invalidate existing procedures. Consider the following example:

Sample Code

```
create table mytab (i int primary key);
create table mytab2 (i int);
create procedure myproc (out ot mytab) as begin
    ot = select * from mytab2;  -- error if not_null_column_mode is set to
'respect'
end;
```

7.3 Binding Table Variables

Table variables are bound by 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, which means that recursion is not possible.

7.4 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.

7.5 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")',?);
```

7.6 Map Reduce Operator

MAP_REDUCE is a programming model introduced by Google that allows easy development of scalable parallel applications for processing big data on large clusters of commodity machines. The MAP_MERGE operator is a specialization of the MAP_REDUCE operator.

Syntax

Code Syntax

```
MAP_REDUCE(<input table/table variable name>, <mapper specification>,
<reducer specification>)
<mapper spec> ::= <mapper TUDF>(<list of mapper parameters>) group by <list
of columns in the TUDF> as <ID>
<reducer spec> ::= <reduce TUDF>(<list of reducer TUDF parameters>)
                 | <reduce procedure>(<list of reducer procedure parameters>)
<mapper parameter> ::= <table/table variable name>.<column name> | <other
scalar parameter>
<reducer TUDF parameter> ::= <ID> | <ID>.<key column name> | <other scalar
parameter>
<reducer procedure parameter> ::= <reducer TUDF parameter> | <output table
parameter>
```

Example

We take as an example a table containing sentences with their IDs. If you want to count the number of sentences that contain a certain character and the number of occurrences of each character in the table, you can use the MAP_REDUCE operator in the following way:

Mapper Function

Sample Code

Mapper Function

```
create function mapper(in id int, in sentence varchar(5000))
returns table (id int, c varchar, freq int) as begin
    using sqlscript_string as lib;
    declare tv table(result varchar);
    tv = lib:split_to_table(:sentence, ' ');
    return select :id as id, result as c, count(result) as freq from :tv
group by result;
end;
```

Reducer Function

Sample Code

Reducer Function

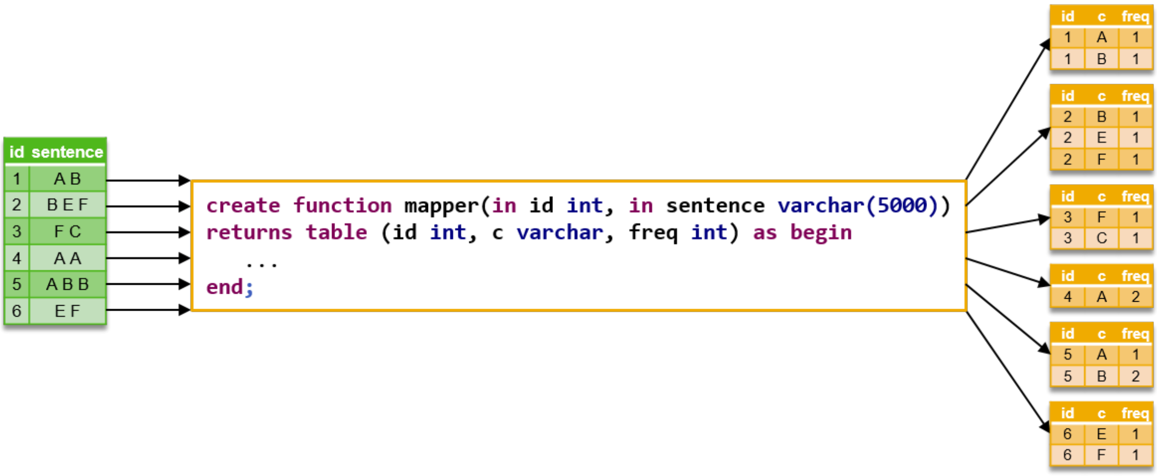
```
create function reducer(in c varchar, in vals table(id int, freq int))
returns table (c varchar, stmt_freq int, total_freq int) as begin
    return select :c as c, count(distinct(id)) as stmt_freq, sum(freq) as
total_freq from :vals;
end;
```

Sample Code

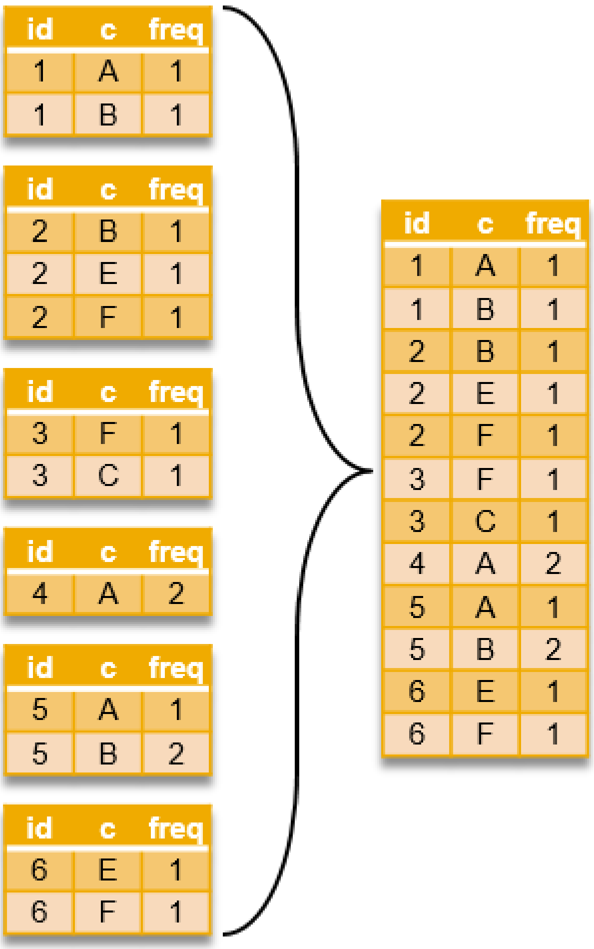
```
do begin
    declare result table(c varchar, stmt_freq int, total_freq int);
    result = MAP_REDUCE(tab, mapper(tab.id, tab.sentence) group by c as X,
                         reducer(X.c, X));
    select * from :result order by c;
end;
```

The code above works in the following way:

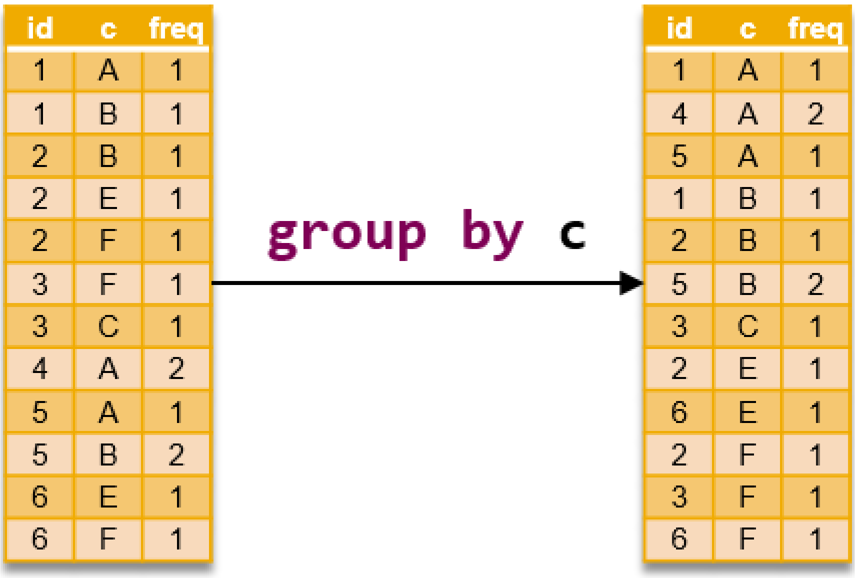
1. The mapper TUDF processes each row of the input table and returns a table.



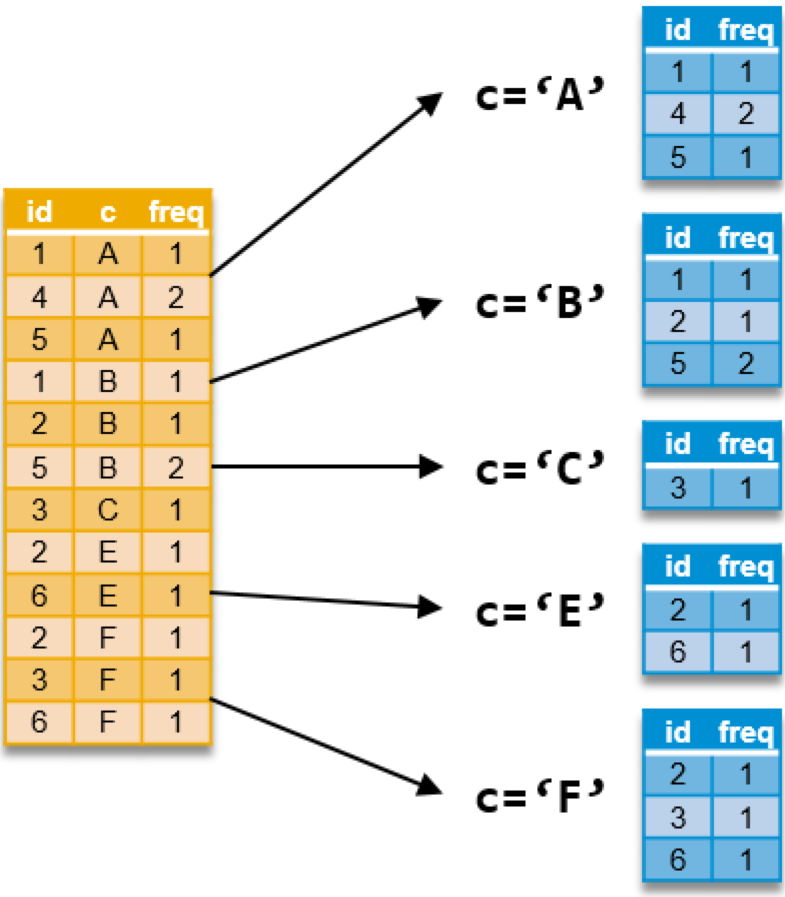
2. When all rows are processed by the mapper, the output tables of the mapper are aggregated into a single big table (like MAP_MERGE).



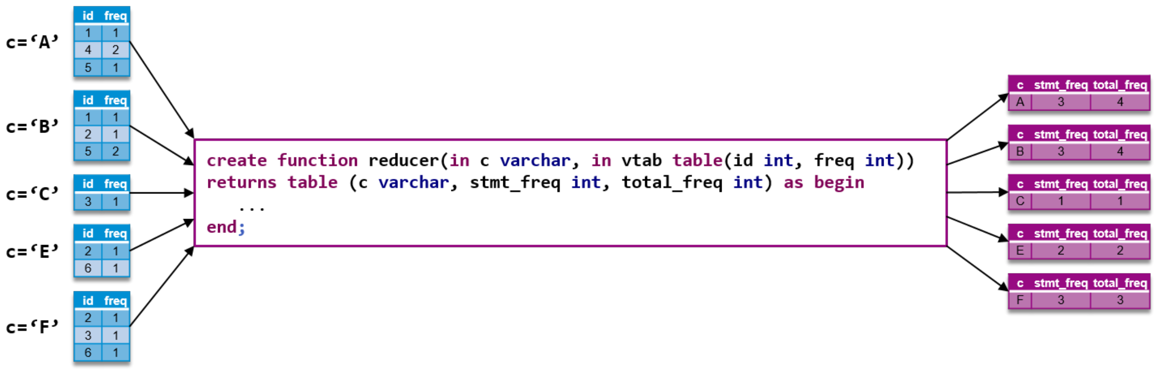
3. The rows in the aggregated table are grouped by key columns.



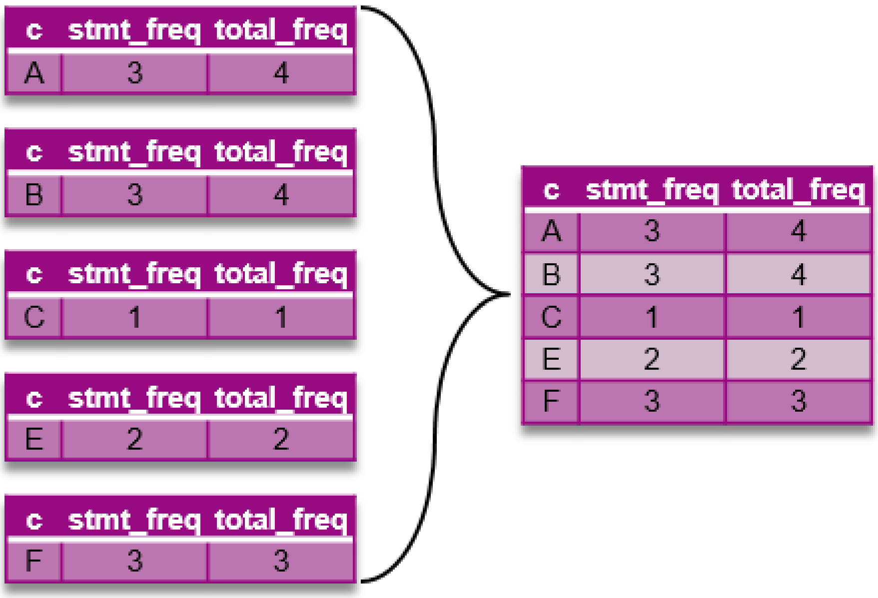
4. For each group, the key values are separated from the table. The grouped table without key columns is called 'value table'. The order of the rest of columns is preserved. It is possible to have multiple key columns. If the layout of the output table is `table(a int, b varchar, c timestamp, d int)` and the key column is `b` and `c`, the layout of the value table is `table(a int, d int)`.



5. The reducer TUDF (or procedure) processes each group and returns a table (or multiple tables).



6. When all groups are processed, the output tables of the reducer are aggregated into a single big table (or multiple tables, if the reducer is a procedure).



Retrieving Multiple Outputs from MAP_REDUCE

If you use a read-only procedure as a reducer, you can fetch multiple table outputs from a MAP_REDUCE operator. To bind the output of MAP_REDUCE operators, you can simply apply the table variable as the parameter of the reducer specification. For example, if you want to change the reducer in the example above to a read-only procedure, apply the following code.

```
create procedure reducer_procedure(in c varchar, in values table(id int, freq
int), out otab table (c varchar, stmt_freq int, total_freq int))
reads sql data as begin
    otab = select :c as c, count(distinct(id)) as stmt_freq, sum(freq) as
total_freq from :values;
end;
```

```
do begin
    declare result table(c varchar, stmt_freq int, total_freq int);
    MAP_REDUCE(tab, mapper(tab.id, tab.sentence) group by c as X,
                    reducer_procedure(X.c, X, result));
```

```
    select * from :result order by c;
end;
```

Passing Extra Arguments as a Parameter to a Mapper or a Reducer

It is possible to pass extra arguments as parameters of a mapper or a reducer.

Sample Code

```
create function mapper(in id int, in sentence varchar(5000), in
some_extra_arg1 int, in some_extra_arg2 table(...), ...)
returns table (id int, c varchar, freq int) as begin
    ...
end;
```

```
create function reducer(in c varchar, in values table(id int, freq int), in
some_extra_arg1 int, in some_extra_arg2 table(...), ...)
returns table (c varchar, stmt_freq int, total_freq int) as begin
    ...
end;
```

```
do begin
    declare result table(c varchar, stmt_freq int, total_freq int);
    declare extra_arg1, extra_arg2 int;
    declare extra_arg3, extra_arg4 table(...);
    ... more extra args ...
    result = MAP_REDUCE(tab, mapper(tab.id,
tab.sentence, :extra_arg1, :extra_arg3, ...) group by c as X,
                             reducer(X.c, X, :extra_arg2, :extra_arg4,
1+1, ...));
    select * from :result order by c;
end;
```

Note

There is no restriction about the order of input table parameters, input column parameters, extra parameters and so on. It is also possible to use default parameter values in mapper/reducer TUDFs or procedures.

Restrictions

The following restrictions apply:

- Only Mapper and Reducer are supported (no other Hadoop functionalities like group comparator, key comparator and so on).
- The alias ID in the mapper output and the ID in the Reducer TUDF (or procedure) parameter must be the same.
- The Mapper must be a TUDF, not a procedure.
- The Reducer procedure should be a read-only procedure and cannot have scalar output parameters.

* The order of the rows in the output tables is not deterministic.

Related Information

Map Merge Operator [page 101]

7.7 Map Merge Operator

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.

Sample Code

```
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;
```

i Note

The mapper procedure is a read-only procedure with only one output that is a tabular output.

Syntax

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

The first input of the `MAP_MERGE` operator is the mapper table `<table_or_table_variable>` . The mapper table is a table or a table variable on which you want to iterate by rows. In the above example, it would be table variable `t`.

The second input is the mapper function `<mapper_identifier>` itself. The mapper function is a function you want to have evaluated on each row of the mapper table `<table_or_table_variable>`. Currently, the `MAP_MERGE` operator supports only table functions as `<mapper_identifier>`. This means that in the above example you need to convert the mapper procedure into a table function.

You also have to pass the mapping argument `<table_or_table_variable>.<column_Name>` as an input of the mapper function. Going back to the example above, this would be the value of the variable `varb`.

Example

As an example, let us rewrite the above example to leverage the parallel execution of the `MAP_MERGE` operator. We need to transform the procedure into a table function, because `MAP_MERGE` only supports table functions as `<mapper_identifier>`.

Sample Code

```
CREATE FUNCTION mapper (IN a nvarchar(200))
RETURNS TABLE (col_a nvarchar(200))
AS
BEGIN
    ot = SELECT :a AS COL_A from dummy;
    RETURN :ot;
END;
```

After transforming the mapper procedure into a function, we can now replace the whole `FOR` loop by the `MAP_MERGE` operator.

Sequential FOR-Loop Version

```
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;
```

Parallel MAP_Merge Operator

```
DO (OUT ret_tab TABLE(col_a
nvarchar(200))=>?)
BEGIN
    t = SELECT * FROM tab;
    ret_tab = MAP_MERGE(:t,
mapper(:t.col_a));
END;
```

7.8 Hints

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

7.8.1 NO_INLINE and INLINE Hints

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

> **i 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;
```

7.8.2 ROUTE_TO Hint

The ROUTE_TO hint routes the query to the specified volume ID or service type.

Syntax

Code Syntax

```
<servicetype> ::= 'indexserver' | 'xsengine' | 'scriptserver' | 'dpserver' |
'computeserver'
<hint_with_parameters> ::= ROUTE_TO( <volume_id> [{, <volume_id> }] )
                           | ROUTE_TO( '<servicetype>' [{,
'<servicetype>' }] )
                           | NO_ROUTE_TO( <volume_id> [{, <volume_id> }] )
                           | NO_ROUTE_TO( '<servicetype>' [{,
'<servicetype>' }] )
                           | ROUTE_BY( <table_name> [{, <table_name>}] )
                           | ROUTE_BY_CARDINALITY( <table_name> [{,
<table_name>}] )
                           | DATA_TRANSFER_COST ({0 | 1})
```

Description

The ROUTE_TO hint can be used with either "volume ID", or "service type". If the "volume id" is provided, the statement is intended to be routed to the specified volume. But if the "service type" (a string argument that can have values like "indexserver", "computeserver" and so on) is provided within the hint, the statement can be routed to all nodes related to this service.

Example

Sample Code

```
create table mytab(a int);
insert into mytab values(1);

select * from mytab with hint(ROUTE_TO('indexserver'));
select * from mytab with hint(ROUTE_TO('indexserver','computeserver'));

select * from mytab with hint(NO_ROUTE_TO('indexserver'));
select * from mytab with hint(NO_ROUTE_TO('indexserver','computeserver'));

select preferred_routing_volumes, * from sys.m_sql_plan_cache_ where
statement_string like '%select * from mytab%';
```

7.9 SQLScript Variable Cache

Description

SQLScript data caching improves performance by exploiting cached intermediate result data corresponding to a table variable assigned to a SELECT query.

It is used mainly for storing an intermediate result fetched from distributed query processing in a compute-storage separation architecture.

Syntax

Syntax

```
CREATE PROCEDURE <procedure_name> [(<parameter_clause>)] [LANGUAGE <lang>]
    [SQL SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name>] [READS SQLDATA]
[<route_target_element>]
    [<variable_cache_option>] AS
```

```
    BEGIN [SEQUENTIAL EXECUTION] <procedure_body> END
CREATE FUNCTION <function_name> [(<parameter_clause>)]
    RETURNS <return_type> [LANGUAGE <lang>]
    [SQL SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name>]
[<route_target_element>]
    [<variable_cache_option>] AS
    BEGIN <function_body> END
```

Syntax Elements

```
<variable_cache_prefix> ::= VARIABLE CACHE ON
<enable_mode> ::= ENABLE | DISABLE | AUTOMATIC
<variable_list> ::= <variable_name> [, <variable_name> …]
<variable_list_clustered> ::= ( <variable_list> )
<variable_entry> ::= <variable_name> | <variable_list_clustered>
<variable_entry_with_mode> ::= <variable_entry> <enable_mode>
<variable_entry_without_mode> ::= <variable_entry>
<variable_entry_with_mode_list> ::= <variable_entry_with_mode>
[,<variable_entry_with_mode> …]
<variable_entry_without_mode_list> ::= <variable_entry_without_mode>
[,<variable_entry_without_mode> …]
<variable_cache_option> ::= <variable_cache_prefix>
<variable_entry_with_mode_list>|<variable_entry_without_mode_list>
<variable_cache_option_with_mode_mandatory> ::= <variable_cache_prefix>
<variable_entry_with_mode_list>
<variable_cache_option_plain> ::= <variable_cache_prefix> <variable_list>
<object_type> ::= PROCEDURE | FUNCTION
-- Add new variables
ALTER <object_type> <object_name> ADD <variable_cache_option>
-- Remove variables
ALTER <object_type> <object_name> DROP <variable_cache_option_plain>
-- Remove all variables
ALTER <object_type> <object_name> DROP VARIABLE CACHE ALL
-- Change enable mode of existing variables
ALTER <object_type> <object_name> ALTER
<variable_cache_option_with_mode_mandatory>
```

Configuration

Variable Cache Mode

All table variables, without a specified mode of caching, can be assigned to variable caching with the following configuration.

```
ALTER SYSTEM ALTER CONFIGURATION ('indexserver.ini', 'system') SET
('sqlscript','variable_cache_default_mode') = 'enable|disable|automatic' WITH
RECONFIGURE;
```

Automatic Specification in Variable Cache Mode

Table variables, whose mode of caching is automatic, are cached when the thresholds specified in the configuration in the format below are satisfied.

LOAD_TIME: Statement execution + ITAB (intermediate result) materialization

ITAB_SIZE: Size of materialized ITAB (intermediate result)

```
ALTER SYSTEM ALTER CONFIGURATION ('indexserver.ini', 'system') SET
('sqlscript','variable_cache_automatic_config') = ' {"LOAD_TIME": <microsecond>,
"ITAB_SIZE": <byte> }' WITH RECONFIGURE;
```

Example

Sample Code

```
create table mytab1(a int);
insert into mytab1 values(1);
create table mytab2(b int);
insert into mytab2 values(2);
create table mytab3(c int);
insert into mytab3 values(3);

drop procedure myproc;
create procedure myproc
as begin
    a = select * from mytab1;
    b = select * from mytab2;
    c = select * from mytab3;
    select * from :a, :b, :c;
end;

call myproc; -- disabled cache
alter procedure myproc add variable cache on A enable,B enable;
call myproc; -- 1st run, cache miss and store results for a and b
call myproc; -- 2nd run, cache hit a and b
insert into mytab1 values (11); -- invalidate cache for mytab1
call myproc; -- 3rd run, cache miss for a, and hit for b
alter procedure myproc drop variable cache on C;
call myproc; -- disabled cache
```

Supportability

System View / Monitoring View

SQLSCRIPT_VARIABLE_CACHE: View indicating which variables are to be cached

Column Name	Data Type	Description
SCHEMA _NAME	NVARCHAR(256)	Schema of the target object
OBJECT_ NAME	NVARCHAR(256)	Target object name
OBJECT_TYPE	VARCHAR(16)	Object type ("PROCEDURE" or "FUNC-TION")
VARIABLE _NAME	NVARCHAR(256)	Variable name to be cached
VARIABLE_TYPE	VARCHAR(16)	Type of variable ("TABLE")
ENABLE_MODE	VARCHAR(16)	Activation mode ("ENABLED", "DISA-BLED", or "AUTOMATIC")

M_SQLSCRIPT_ VARIABLE _CACHE : Monitoring view projecting statistics of currently cached variables

Column Name	Data Type	Description

HOST	VARCHAR(64)	Host of the node where the cached data is located
PORT	INTEGER	Port of the node where cached data is located
SCHEMA_NAME	NVARCHAR(256)	Schema of the target object
OBJECT_NAME	NVARCHAR(256)	Target object name
OBJECT_TYPE	VARCHAR(16)	Object type ("PROCEDURE" or "FUNC-TION")
VARIABLE_NAME	NVARCHAR(256)	Variable name to be cached
VARIABLE_TYPE	VARCHAR(16)	Type of variable ("TABLE")
SQLSCRIPT_PLAN_ID	INTEGER	ID of the SQLScript execution plan
SQLSCRIPT_OPERATOR_ID	INTEGER	SQLScript Operator ID
MEMORY_SIZE	INTEGER	The memory size of the cached variable
CACHE_TIMESTAMP	TIMESTAMP	Date and time when the latest cached data was generated

Scope

- Only caching table variable is supported. Scalar, array, and row type variables cannot be cached.
- Results from SELECT statements, referencing only persistent tables or views, can be cached.
- SELECT statements with following conditions **cannot** be cached:
    - the statement contains non-deterministic SQL functions: RAND, SYSUUID, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_CONNECTION, CURRENT_USER, SESSION_CONTEXT
    - the statement contains crypto functions: HASH_SHA256, HASH_MD5
    - the statement contains tables that can be updated in other statements in the current procedure, referenced by DDL/DML statements or by another CALL statement for a procedure not specified with READ SQL DATA
    - the statement contains other SQLScript variables
- The cache entry is invalidated when any related table is updated.

8 Imperative SQLScript Logic

This section focuses on imperative language constructs such as loops and conditionals. The use of imperative logic splits the logic between several data flows.

Related Information

Declarative SQLScript Logic [page 85]

8.1 Scalar Variables

Syntax

```
DECLARE <sql_identifier> [{,<sql_identifier> }...] [CONSTANT] <type> | AUTO [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 by using the 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 as described above by using `:var`.

→ Tip

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

Even though the `:=` operator is still available, SAP recommends that you use only the = operator in defining scalar variables.

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;
```

This examples shows various ways for making declarations and assignments.

i Note

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 by using an SQL statement:

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

Assign a scalar UDF to a scalar variable:

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

Assign a 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;
```

8.1.1 SELECT INTO with DEFAULT Values

The SELECT INTO statement is widely used for assigning a result set to a set of scalar variables. Since the statement does not accept an empty result set, it is necessary to define exit handlers in case an empty result set is returned. The introduction of DEFAULT values makes it possible to to handle empty result sets without the need of writing exit handlers to assign default values to the target variables when the result set is empty.

Syntax

Code Syntax

```
SELECT <select_list> INTO <var_name_list> [DEFAULT <scalar_expr_list>]
<from_clause>
[<where_clause>]
[<group_by_clause>]
[<having_clause>]
 [{<set_operator> <subquery>, ... }]
[<order_by_clause>]
 [<limit>] ;
[EXEC | EXECUTE IMMEDIATE] <string_expression>
[ INTO  <var_name_list> [DEFAULT <scalar_expr_list>] ]
  [ USING <scalar_expr_list> ]
```

Description

It is also possible to use a single array element as the result of SELECT INTO and EXEC INTO. The syntax of the INTO clause was extended as follows:

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

Sample Code

```
DROP TABLE T1;
CREATE TABLE T1 (A INT NOT NULL, B VARCHAR(10));

DO BEGIN
    DECLARE A_COPY INT ARRAY;
    DECLARE B_COPY VARCHAR(10) ARRAY;
    SELECT A, B INTO A_COPY[1], B_COPY[1] DEFAULT -2+1, NULL FROM T1;
    SELECT :A_COPY[1], :B_COPY[1] from dummy;
    --(A_COPY[1],B_COPY[1]) = (-1,?), use default value
    EXEC 'SELECT A FROM T1' INTO A_COPY[1] DEFAULT 2;
    SELECT :A_COPY[1], :B_COPY[1] from dummy;
    --(A_COPY[1]) = (2), exec into statement with default value
    INSERT INTO T1 VALUES (0, 'sample0');
    SELECT A, B INTO A_COPY[1], B_COPY[1] DEFAULT 5, NULL FROM T1;
    SELECT :A_COPY[1], :B_COPY[1] from dummy;
    --(A_COPY[1],B_COPY[1]) = (0,'sample0'), executed as-is
END;
```

Example

```
DO BEGIN
    DECLARE A_COPY INT;
    DECLARE B_COPY VARCHAR(10);
    CREATE ROW TABLE T1 (A INT NOT NULL, B VARCHAR(10));
    SELECT A, B INTO A_COPY, B_COPY DEFAULT -2+1, NULL FROM T1;
    --(A_COPY,B_COPY) = (-1,?), use default value
    EXEC 'SELECT A FROM T1' INTO A_COPY DEFAULT 2;
    --(A_COPY) = (2), exec into statement with default value
    INSERT INTO T1 VALUES (0, 'sample0');
    SELECT A, B INTO A_COPY, B_COPY DEFAULT 5, NULL FROM T1;
    --(A_COPY,B_COPY) = (0,'sample0'), executed as-is
END;
```

Related Information

EXEC [page 165]
EXECUTE IMMEDIATE [page 165]
CREATE PROCEDURE [page 19]

8.1.2 SQL in Scalar Expressions

Description

SQLScript now supports SELECT as an SQL query within scalar expressions.

If the SELECT statement returns a 1*1 result set (1 row and 1 column), that result set can be used directly as an expression.

The following use cases are possible:

- SQL sub-query within a scalar value assignment
- SQL sub-query within a condition.

Examples

Sample Code

```
x = (SELECT TOP 1 val from mytab) * 10; ...

IF (SELECT MAX(val) FROM mytab) > 100 THEN ...
```

The result set of the sub-query is expected to have a 1*1 size but if the result set has 0 records, a null value will be returned. In any other case, you will get an error message.

```
create table multiple_col_tab(i int, j int);
insert into multiple_col_tab values(1, 2);
do begin
  declare n int = (select * from multiple_col_tab) + 1; -- ERR-00269: too many
values
end;

create table multiple_row_tab(i int);
insert into multiple_row_tab values(1);
insert into multiple_row_tab values(2);
do begin
  declare n int = (select * from multiple_row_tab) + 1; -- ERR-01300: fetch
returns more than requested number of rows
end;

create table empty_tab(i int);
do begin
  declare n int = (select * from empty_tab) + 1; -- n has null value
end;
```

If the right-hand side of an assignment contains only a SELECT statement (even with parenthesizes, for example: `x = (SELECT * FROM tab)`), it will be always be treated as a table variable assignment. The workaround is to use SELECT INTO.

```
create table mytab(i int);
insert into mytab values(1);

do begin
  declare n int;
  n = (select i from mytab); -- ERR-01310: scalar type is not allowed: N
end;

do begin
  declare n int;
  select i into n from mytab; -- workaround
end;
```

Limitations

Auto type is not supported.

```
do begin
  declare n auto = (select 10 from dummy) + 1; -- ERR-00007: feature not
supported: subquery in auto type assignment
end;
```

8.2 Table Variables

Table variables are, as the name suggests, variables with a reference to tabular data structure. The same applies to tabular parameters, unless specified otherwise.

Related Information

Table Variable Type Definition [page 90]

8.2.1 Table Variable Operators

8.2.1.1 Index-Based Cell Access to Table Variables

The index-based cell access allows you random access (read and write) to each cell of a table variable.

```
<table_variable>.<column_name>[<index>]
```

For example, writing to a 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 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 -2 ([1-2147483646]) and that an SQL expression or a scalar user-defined functions (scalar UDF) that return a number also can be used as an index. Instead of using 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.

8.2.1.2 Modifying the Content of Table Variables

Apart from the index-based table cell assignment, SQLScript offers additional operations for directly modifying the content of a table variable, without having to assign the result of a statement to a new table variable. This, together with not involving the SQL layer, leads to performance improvement. On the other hand, such operations require data materialization, contrary to the declarative logic.

> **i Note**
>
> For all position expressions the valid values are in the interval from 1 to 2^31 -2 ([1-2147483646]).

Inserting Data Records into Table Variables

You can insert a new data record at a specific position in a table variable with the following syntax:

```
:<table_variable>.INSERT((<value1,…, <valueN), [, <index> ])
```

All existing data records at positions starting from the given index onwards, are moved to the next position. If the index is greater than the original table size, the records between the inserted record and the original last record are initialized with `NULL` values.

Sample Code

```
CREATE TABLE TAB(K VARCHAR(20), V INT);
INSERT INTO TAB VALUES('A', 7582);
INSERT INTO TAB VALUES('B', 4730);
INSERT INTO TAB VALUES('C', 1960);
INSERT INTO TAB VALUES('A', 8650);
INSERT INTO TAB VALUES('D', 1318);
INSERT INTO TAB VALUES('C', 3836);
INSERT INTO TAB VALUES('B', 8602);
INSERT INTO TAB VALUES('C', 3257);
CREATE PROCEDURE ADD_SUM(IN IT TAB, OUT OT TAB) AS
BEGIN
  DECLARE IDX INT = 0;
  DECLARE K VARCHAR(20) = '';
  DECLARE VSUM INT = 0;

  IF IS_EMPTY(:IT) THEN
    RETURN;
  END IF;

  OT = SELECT * FROM :IT ORDER BY K;
  WHILE :OT.K[IDX + 1] IS NOT NULL DO
    IDX = IDX + 1;
    IF :OT.K[IDX] <> K THEN
      IF K <> '' THEN
        :OT.INSERT(('Sum ' || K, VSUM), IDX);
        IDX = IDX + 1;
      END IF;
      :OT.INSERT(('Section ' || :OT.K[IDX], 0), IDX);
      IDX = IDX + 1;
      K = :OT.K[IDX];
      VSUM = 0;
    END IF;
    VSUM = VSUM + :OT.V[IDX];
  END WHILE;
```

```
    :OT.INSERT(('Sum ' || K, VSUM), IDX + 1);
END
CALL ADD_SUM(TAB, ?)
K                  V
------------------
Section A          0
A              7.582
A              8.650
Sum A         16.232
Section B          0
B              4.730
B              8.602
Sum B         13.332
Section C          0
C              1.960
C              3.836
C              3.257
Sum C          9.053
Section D          0
D              1.318
Sum D          1.318
```

If you do not specify an index (position), the data record will be appended at the end.

Sample Code

```
CREATE TABLE SOURCE(K VARCHAR(20), PCT DECIMAL(5, 2), V DECIMAL(10, 2));
CREATE TABLE TARGET(K VARCHAR(20), V DECIMAL(10, 2));
INSERT INTO SOURCE VALUES ('A', 5.99, 734.42);
INSERT INTO SOURCE VALUES ('A', 50.83, 422.26);
INSERT INTO SOURCE VALUES ('B', 75.07, 362.53);
INSERT INTO SOURCE VALUES ('C', 87.21, 134.53);
INSERT INTO SOURCE VALUES ('C', 80.72, 2722.49);
CREATE PROCEDURE SPLIT(IN IT SOURCE, OUT OT1 TARGET, OUT OT2 TARGET) AS
BEGIN
  DECLARE IDX INT;
  DECLARE MAXIDX INT = RECORD_COUNT(:IT);
  FOR IDX IN 1..MAXIDX DO
    DECLARE V1 DECIMAL(10, 2) = :IT.V[IDX] * :IT.PCT[IDX] / 100;
    DECLARE V2 DECIMAL(10, 2) = :IT.V[IDX] - V1;
    :OT1.INSERT((:IT.K[IDX], V1));
    :OT2.INSERT((:IT.K[IDX], V2));
  END FOR;
END;
CALL SPLIT(SOURCE, ?, ?);
OT1            OT2
K  V           K  V
------------------------
A     43,99    A  690,43
A    214,63    A  207,64
B    272,15    B   90,38
C    117,32    C   17,21
C  2.197,59    C  524,9
```

You can also provide values for a limited set of columns:

```
:<table_variable>.(<column1>,…, <column>).INSERT((<value1>,…, <valueN>),
[ <index> ])
```

Note

The values for the omitted columns are initialized with `NULL` values.

Inserting Table Variables into Other Table Variables

You can insert the content of one table variable into another table variable with one single operation without using SQL.

Code Syntax

```
:<target_table_var>[.(<column_list>)].INSERT(:<source_table_var>[,
<position>])
```

If no position is specified, the values will be appended to the end. The positions starts from 1 - NULL and all values smaller than 1 are invalid. If no column list is specified, all columns of the table are insertion targets.

Sample Code

Usage Example

```
:tab_a.insert(:tab_b);
:tab_a.(col1, COL2).insert(:tab_b);
:tab_a.INSERT(:tab_b, 5);
:tab_a.("a","b").insert(:tab_b, :index_to_insert);
```

The mapping which column of the source table is inserted into which column of the target table is done according to the column position. The source table has to have the same number of columns as the target table or as the number of columns in the column list.

If SOURCE_TAB has columns (X, A, B, C) and TARGET_TAB has columns (A, B, C, D), then `:target_tab.insert(:source_tab)` will insert X into A, A into B, B into C and C into D.

If another order is desired, the column sequence has to specified in the column list for the TARGET_TAB. for example `:TARGET_TAB.(D, A, B, C).insert(:SOURCE_TAB)` will insert X into D, A into A, B into B and C into C.

The types of the columns have to match, otherwise it is not possible to insert data into the column. For example, a column of type DECIMAL cannot be inserted in an INTEGER column and vice versa.

Sample Code

Iterative Result Build

```
CREATE COLUMN TABLE DATA(K VARCHAR, V INT);
INSERT INTO DATA VALUES('A', 123);
INSERT INTO DATA VALUES('B', 45);
INSERT INTO DATA VALUES('B', 67);
INSERT INTO DATA VALUES('C', 890);

CREATE PROCEDURE P(OUT OT DATA) AS
BEGIN
  DECLARE I INT;
  LT0 = SELECT DISTINCT K FROM DATA;
  FOR I IN 1..RECORD_COUNT(:LT0) DO
    DECLARE K VARCHAR = :LT0.K[I];
    LT1 = SELECT K, V + 1000 * :I AS V FROM DATA WHERE K = :K;
    :OT.INSERT(:LT1, 1);
  END FOR;
END;

CALL P(?)
```

```
K       V
--------
C   3.890
B   2.045
B   2.067
A   1.123
```

Updating Data Records in Table Variables

You can modify a data record at a specific position. There are two equivalent syntax options.

```
:<table_variable>.UPDATE((<value1>,…, <valueN), <index>)
<table_variable>[<index>] = (<value1>,…, <valueN>)
```

i Note

The index must be specified.

You can also provide values for a limited set of columns.

```
:<table_variable>.(<column1>,…, <column>).UPDATE((<value1>,…, <valueN>), <index>)
<table_variable>.(<column1>,…, <column>)[<index>] = (<value1>,…, <valueN>)
```

i Note

The values for the omitted columns remain unchanged.

Sample Code

```
CREATE TABLE TAB (V1 INT, V2 INT);
INSERT INTO TAB VALUES(599, 7442);
INSERT INTO TAB VALUES(5083, 4226);
INSERT INTO TAB VALUES(7507, 3253);
INSERT INTO TAB VALUES(8721, 1453);
INSERT INTO TAB VALUES(8072, 2749);
CREATE PROCEDURE MIRROR (IN IT TAB, OUT OT TAB) AS
BEGIN
  DECLARE IDX INT;
  DECLARE MAXIDX INT = RECORD_COUNT(:IT);
  FOR IDX IN 1..MAXIDX DO
    OT[MAXIDX-IDX+1] = (:IT.V2[:IDX], :IT.V1[:IDX]);
  END FOR;
END;
CALL MIRROR(TAB, ?);
V1        V2
------------
2.749  8.072
1.453  8.721
3.253  7.507
4.226  5.083
7.442    599
```

i Note

You can also set values at a position outside the original table size. Just like with `INSERT`, the records between the original last record and the newly inserted records are initialized with `NULL` values.

Deleting Data Records from Table Variables

You can delete data records from a table variable.

Deleting a Single Record

You can use the following syntax:

```
:<table_variable>.DELETE([ <index> ])
```

If no index (position) is specified, all records are deleted.

If the index is outside the table size, no operation is performed.

Sample Code

```
CREATE TABLE HIER(PARENT VARCHAR(30), CHILD VARCHAR(30));
INSERT INTO HIER VALUES ('root', 'A');
INSERT INTO HIER VALUES ('root', 'B');
INSERT INTO HIER VALUES ('A', 'C');
INSERT INTO HIER VALUES ('C', 'D');
INSERT INTO HIER VALUES ('A', 'E');
INSERT INTO HIER VALUES ('E', 'F');
INSERT INTO HIER VALUES ('E', 'G');
CREATE PROCEDURE CALC_LEVEL (IN IT HIER, IN ROOT VARCHAR(30), OUT OT_LEVEL
TABLE(NODE VARCHAR(30), L INT)) AS
BEGIN
  DECLARE STACK TABLE(NODE VARCHAR(30), L INT);

  STACK[1] = (ROOT, 1);
  WHILE NOT IS_EMPTY(:STACK) DO
    DECLARE I INT;
    DECLARE NUM_CHILDREN INT;
    DECLARE CURR_NODE VARCHAR(30) = :STACK.NODE[1];
    DECLARE CURR_LEVEL INT = :STACK.L[1];
    CHILDREN = SELECT CHILD FROM :IT WHERE PARENT = CURR_NODE;
    :OT_LEVEL.INSERT((CURR_NODE, CURR_LEVEL));
    NUM_CHILDREN = RECORD_COUNT(:CHILDREN);
    :STACK.DELETE(1);
    FOR I IN 1..NUM_CHILDREN DO
      :STACK.INSERT((:CHILDREN.CHILD[I], CURR_LEVEL + 1));
    END FOR;
  END WHILE;
END;
CALL CALC_LEVEL(HIER, 'root', ?)
NODE  L
-------
root  1
A     2
B     2
C     3
E     3
D     4
F     4
G     4
```

Deleting Blocks of Records from Table Variables

To delete blocks of records from table variables, you can use the following syntax:

```
:<table_variable>.DELETE(<start index>..<end index>)
```

If the starting index is greater than the table size, no operation is performed. If the end index is smaller than the starting index, an error occurs. If the end index is greater than the table size, all records from the starting index to the end of the table are deleted.

Sample Code

```
CREATE TABLE PROD_PER_DATE (PROD_NAME VARCHAR(20), PROD_DATE DATE, NUM_DELTA
INT);
INSERT INTO PROD_PER_DATE VALUES ('PC', '20170105', 100);
INSERT INTO PROD_PER_DATE VALUES ('PC', '20170106', 50);
INSERT INTO PROD_PER_DATE VALUES ('PC', '20170117', 200);
INSERT INTO PROD_PER_DATE VALUES ('Notebook', '20170320', 30);
INSERT INTO PROD_PER_DATE VALUES ('Notebook', '20170322', 310);
INSERT INTO PROD_PER_DATE VALUES ('Phone', '20170121', 20);
INSERT INTO PROD_PER_DATE VALUES ('Phone', '20170205', 50);
CREATE PROCEDURE TOTAL_NUM_EXCEEDS_CAPACITY (
  IN IT PROD_PER_DATE,
  IN CAPACITY INT,
  OUT OT_RESULT TABLE(PROD_NAME VARCHAR(20), PROD_DATE DATE, NUM_TOTAL INT)
) AS
BEGIN
  DECLARE IDX INT = 0;
  DECLARE NUM_TOTAL INT = 0;
  DECLARE INTERVALS TABLE(FROM_IDX INT, TO_IDX INT);
  DECLARE FROM_IDX INT = 1;
  DECLARE TO_IDX INT = 0;
  OT_RESULT = SELECT PROD_NAME, PROD_DATE, NUM_DELTA AS NUM_TOTAL
    FROM :IT ORDER BY PROD_NAME, PROD_DATE;
  WHILE :OT_RESULT.PROD_NAME[IDX + 1] IS NOT NULL DO
    IDX = IDX+1;
    IF IDX > 1 THEN
      IF :OT_RESULT.PROD_NAME[IDX] <> :OT_RESULT.PROD_NAME[IDX - 1] THEN
        IF TO_IDX = 0 THEN
          TO_IDX = IDX - 1;
        END IF;
        IF FROM_IDX <= TO_IDX THEN
          :INTERVALS.INSERT((FROM_IDX, TO_IDX));
        END IF;
        NUM_TOTAL = 0;
        FROM_IDX = IDX;
        TO_IDX = 0;
      END IF;
    END IF;
    NUM_TOTAL = NUM_TOTAL + :OT_RESULT.NUM_TOTAL[IDX];
    OT_RESULT.NUM_TOTAL[IDX] = NUM_TOTAL;
    IF NUM_TOTAL > CAPACITY AND TO_IDX = 0 THEN
      TO_IDX = IDX - 1;
    END IF;
  END WHILE;
  IF TO_IDX = 0 THEN
    TO_IDX = IDX;
  END IF;
  :INTERVALS.INSERT((FROM_IDX, TO_IDX));
  IDX = RECORD_COUNT(:INTERVALS);
  WHILE IDX > 0 DO
    :OT_RESULT.DELETE(:INTERVALS.FROM_IDX[IDX] .. :INTERVALS.TO_IDX[IDX]);
    IDX = IDX - 1;
  END WHILE;
END;
CALL TOTAL_NUM_EXCEEDS_CAPACITY(PROD_PER_DATE, 100, ?)
PROD_NAME  PROD_DATE   NUM_TOTAL
--------------------------------
Notebook   22.03.2017        340
PC         06.01.2017        150
PC         17.01.2017        350
```

ℹ Note

The algorithm works with positive delta values only.

Deleting Selected Records from a Table Variable

```
:<table_variable>.DELETE(<array_of_integers>)
```

The provided array expression contains indexes pointing to records which shall be deleted from the table variable. If the array contains an invalid index (for example, zero), an error occurs.

Sample Code

```
CREATE TABLE PROD_PER_DATE (PROD_NAME VARCHAR(20), PROD_DATE DATE, NUM_DELTA
INT);
INSERT INTO DATE_VALUES VALUES ('PC', '20170105', 100);
INSERT INTO DATE_VALUES VALUES ('PC', '20170106', -50);
INSERT INTO DATE_VALUES VALUES ('PC', '20170117', 200);
INSERT INTO DATE_VALUES VALUES ('Notebook', '20170320', 300);
INSERT INTO DATE_VALUES VALUES ('Notebook', '20170322', -10);
INSERT INTO DATE_VALUES VALUES ('Phone', '20170121', 20);
INSERT INTO DATE_VALUES VALUES ('Phone', '20170205', 50);

CREATE PROCEDURE TOTAL_NUM_EXCEEDS_CAPACITY (
  IN IT PROD_PER_DATE,
  IN CAPACITY INT,
  OUT OT_RESULT TABLE(PROD_NAME VARCHAR(20), PROD_DATE DATE, NUM_TOTAL INT)
) AS
BEGIN
  DECLARE IDX INT = 0;
  DECLARE NUM_TOTAL INT = 0;
  DECLARE DEL_IDX INT ARRAY;
  DECLARE ARR_IDX INT = 0;
  OT_RESULT = SELECT PROD_NAME, PROD_DATE, NUM_DELTA AS NUM_TOTAL
    FROM :IT ORDER BY PROD_NAME, PROD_DATE;
  WHILE :OT_RESULT.PROD_NAME[IDX+1] IS NOT NULL DO
    IDX = IDX+1;
    IF IDX > 1 THEN
      IF :OT_RESULT.PROD_NAME[IDX] <> :OT_RESULT.PROD_NAME[IDX - 1] THEN
        NUM_TOTAL = 0;
      END IF;
    END IF;
    NUM_TOTAL = NUM_TOTAL + :OT_RESULT.NUM_TOTAL[IDX];
    OT_RESULT.NUM_TOTAL[IDX] = NUM_TOTAL;
    IF NUM_TOTAL <= CAPACITY THEN
      ARR_IDX = ARR_IDX + 1;
      DEL_IDX[ARR_IDX] = IDX;
    END IF;
  END WHILE;
  :OT_RESULT.DELETE(:DEL_IDX);
END;
CALL TOTAL_NUM_EXCEEDS_CAPACITY(PROD_PER_DATE, 60, ?)
PROD_NAME  PROD_DATE   NUM_TOTAL
--------------------------------
Notebook   20.03.2017        300
Notebook   22.03.2017        290
PC         05.01.2017        100
PC         17.01.2017        250
Phone      05.02.2017         70
```

ℹ Note

This algorithm works also with negative delta values.

8.2.1.3 UNNEST Function

The UNNEST function combines one or many arrays and/or table variables. 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(<unnest_param> [ {, <unnest_param>} ...] )[WITH
ORDINALITY] [AS (<column_specifier> [ {, <column_specifier>}... ]) ]

<unnest_param> ::= :table_variable
                 | :array_variable
                 | :array_function

<column_specifier> ::= '*'
                     | '(' <projection_aliasing_list> ')'
                     | <column_name>

<projection_aliasing_list> ::= <column_name> [AS <column_name>] [,
<projection_aliasing_list>]
```

For example, the following statements convert the array `arr_id` of type INTEGER and the array `arr_name` of type VARCHAR(10) into a table and assign it to the tabular output parameter **`rst`**:

```
CREATE PROCEDURE ARRAY_UNNEST_SIMPLE(OUT rst TABLE(":ARR_ID" INT, ":ARR_NAME"
NVARCHAR(10)))
READS SQL DATA
AS BEGIN
  DECLARE arr_id INTEGER ARRAY = ARRAY(1, 2);
  DECLARE arr_name NVARCHAR(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
```

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
```

It is also possible to use table variables in the UNNEST function. While for arrays the associated column-specifier list entry needs to contain a single column name, the associated entry for a table variable must be either '*' or a projection aliasing list. '*' means that all columns of the input table should be included in the result. With the projection aliasing list, it is possible to specify a subset of the columns of the input table and to rename them in order to avoid name conflicts (a result must not contain multiple columns with the same name).

Sample Code

```
create column table tab0(a int);
insert into tab0 values(1);
insert into tab0 values(2);
insert into tab0 values(3);

do begin
  t0 = select * from tab0 order by a asc;
  t1 = select * from tab0 order by a desc;
  lt = unnest(:t0, :t1) as (*, (a as b));
  select * from :lt;
end;

-- expected result {1, 3}, {2, 2}, {3, 1}

do begin
  t0 = select * from tab0 order by a asc;
  t1 = select * from tab0 order by a desc;
  lt = unnest(:t0, :t1) as (*, (a as b, a as c));
  select * from :lt;
end;

-- expected result {1, 3, 3}, {2, 2, 2}, {3, 1, 1}
```

If the result table variable is declared explicitly, it may contain columns with NOT NULL types. Due to the fact that the columns are adjusted to the longest column, this scenario may lead to a run-time error. The following table shows the NOT NULL behavior:

Result	LHS Type	RHS Type
Potential run-time error	NOT NULL	NOT NULL

Result	LHS Type	RHS Type
Compile-time error	NOT NULL	Nullable
No error	Nullable	NOT NULL
No error	Nullable	Nullable

> **i Note**
>
> Array types are always nullable.

> **i Note**
>
> Default Column Names
>
> If there is no column specifier list, the column names for arrays and the ordinality column in the result table will be generated. A generated name always begins with "COL" and is followed by a number, which refers to the column index in the result table. For example, if the third column in the result table has a generated name, it is "COL3". However, if this name is already occupied because the input table variable contains a column with this name, the index number will be increased to generate an unoccupied column name (if "COL3" is used, "COL4" is the next candidate). This behavior is similar for the ordinality column. This column is named "ORDINALITY" (without index), if this name is available and "ORDINALITY" + INDEX (starting from 1), if "ORDINALITY" is already occupied.

8.2.1.4 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.

i Note

Note that the `IS_EMPTY` cannot be used in SQL queries or expressions.

8.2.1.5 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
```

i Note

RECORD_COUNT cannot be used in queries.

8.2.1.6 Search in Table Variables

This feature offers an efficient way to search by key value pairs in table variables.

Syntax

```
position = <tabvar>.SEARCH((<column_list>), (<value_list>) [, <start_position>])
```

Description

The size of the column list and the value list must be the same, columns and values are matched by their position in the list. The `<start_position>` is optional, the default is `1` (first position), which is equal to scanning all data.

The search function itself can be used in further expressions, but not directly in SQL statements.

The position of the first matching record is returned (or NULL, if no record matches). This result can be used in conjunction with other table variable operators (DELETE, UPDATE).

Example

Sample Code

```
DECLARE LT1 TABLE ("Key1"…, "Key2"…, "Val1"…);
LT1 = … – see Table LT1 Initial State
pos = :LT1.SEARCH (("Key1", "Key2"), ('I', 3)); – pos = NULL (not found)
:LT1.INSERT(('I', 3, 'X')); -- see Table LT1 after a Single Insert
pos = :LT1.SEARCH(("Key1", "Key2"), ('M', 3)); – pos = 5
:LT1.DELETE(pos);
val = :LT1."Val1"[:LT1.SEARCH(("Key1", "Key2"), ('E', 5))]; – val = 'V12'
```

LT1 Initial State

Key 1	Key 2	Val 1
A	1	V11
E	5	V12
B	6	V13
E	7	V14
M	3	V15

LT1 after a Single Insert

Key1	Key2	Val1
A	1	V11
E	5	V12
B	6	V13
E	7	V14
M	3	V15

Key1	Key2	Val1
I	3	X

LT1 after a Single Delete

Key1	Key2	Val1
A	1	V11
E	*5*	*V12*
B	6	V13
E	7	V14
I	3	X

8.2.2 SQL DML Statements on Table Variables

You can modify data in SQLScript table variables with SQL DML statements. The following statements are supported:

- INSERT
- UPDATE
- DELETE

The syntax of the statements is identical with that for manipulating persistent tables. The only difference is that you need to mark the variables by using a colon.

```
DECLARE lt TABLE (a INT, b VARCHAR(20));
INSERT INTO :lt VALUES (1, 'abc');
UPDATE :lt SET b = 'def' WHERE a = 1;
DELETE FROM :lt WHERE a = 1;
```

Constraints

The DML statements for table variables support the following constraint checks:

- Primary key
- NOT NULL

The constraints can be defined in both the user-defined table type and in the declaration, similarly to the persistent table definition.

```
CREATE TYPE tt AS TABLE (a INT PRIMARY KEY, b INT NOT NULL);
DECLARE lt1 tt; -- the variable has constraints defined by the table type
DECLARE lt2 TABLE (a INT, b INT, c INT NOT NULL, PRIMARY KEY(a, b));
```

Compatibility with Other Statements

For implementation reasons, it is not possible to combine DML statements with other table-variable related statements for the same table variable. If a table variable is manipulated by a DML statement, it can only be used in SQL statements: that includes queries and sub-calls, if the variable is bound to an input parameter. The variable cannot be the target of any assign statements and therefore cannot be bound to an output parameter of a sub-call.

```
DECLARE lt1 TABLE(a int);
DECLARE lt2 TABLE LIKE :lt1;
INSERT INTO :lt1 VALUES(1);
INSERT INTO :lt2 (SELECT * FROM :lt1); -- supported
SELECT * FROM :lt2; -- supported
CALL nested_proc(:lt2); -- supported only if the procedure parameter is IN
:lt1.INSERT(:lt2); -- not supported (INSERT operator)
lt2 = SELECT * FROM :lt1; -- not supported (assignment target)
```

Conversion

If you need to combine DML statements with other types of statements for one data set, you need to use multiple table variables. It is possible to convert data between a variable used in a DML statement and a variable not used in a DML statement in both directions.

The following example demonstrates the conversion in both directions:

```
DECLARE tab_without_dml TABLE (a INT);
DECLARE tab_with_dml TABLE LIKE :lt1;
--
tab_without_dml = SELECT * FROM mytab;
--
-- execute non-DML statements with tab_without_dml ...
--
INSERT INTO :tab_with_dml (SELECT * FROM :tab_without_dml); -- convert variable
without DML to variable with DML
--
-- execute DML statements with tab_with_dml ...
--
tab_without_dml = SELECT * FROM :tab_with_dml; -- convert variable with DML to
variable without DML
```

i Note

Both variables are declared the same way, that is at declaration time there is no difference between variables used in a DML statement and variables not used in a DML statement. In both directions, the conversion implies a data copy.

Use Cases

You can use DML statements if your scenario relies mainly on SQL statements, especially if you need to utilize a complex SQL logic for manipulation of your data, like:

- complex WHERE conditions for UPDATE or DELETE
- complex UPDATE statements
- constraint checks

In other cases, it is recommended to use the SQLScript table variable operators for manipulation of table variable data because they offer a better performance, can be combined with other table variable relevant statements and do not imply any restriction with regards to procedure or function parameter handling.

> **i Note**
>
> The primary key check can also be accomplished by using sorted table variables.

Limitations

DML statements on table variables cannot be used in autonomous transactions and parallel execution blocks.

Neither input, nor output procedure or function parameters can be manipulated with DML statements.

8.2.3 Sorted Table Variables

Introduction

Sorted table variables are a special kind of table variables designed to provide efficient access to their data records by means of a defined key. They are suitable for usage in imperative algorithms operating on mass data. The data records of sorted table variables are always sorted by a search key which is specified in the data type of the variable. When accessing the data via the SQLScript search operator, the efficient binary search is utilized, if possible.

Search Key

The search key can be any subset of the table variable columns. The order of the columns in the search key definition is important: the data records are first sorted by the first search key column, then by the second search key column and so on.

i Note

The table LT is sorted by columns B, A, C:

Position	A	B	C	D
1	0	1	10	100
2	2	1	15	200
3	1	2	3	150
4	1	2	5	30

To see how the search key is utilized, check the explanation below about the table variable search operator.

Sequence of Data Records

The sorting order is based on the data type of the search key. As the sorting is relevant only for the SQLScript table variable search operator, it is not guaranteed for all data types that the sorting will behave in exactly the same way as the ORDER BY specification in SQL statements. You can also not influence the sorting - in particular, you cannot specify an ascending or a descending order.

Primary Key

Sorted table variables also allow primary key specification. The primary key must consist exactly of the search key columns. The uniqueness of the primary key is checked in every operation on the table variable (table assignment, insert operator, and so on). If the uniqueness is violated, the corresponding error is thrown.

Data Type Definition

The search key can be specified as part of a user-defined table type:

```
CREATE TYPE <name> AS TABLE (<column list>) SQLSCRIPT SEARCH KEY(<key list>)
```

Variable Declaration

The search key can also be specified as part of a variable declaration:

```
DECLARE <name> TABLE(<column list>) SEARCH KEY(<key list>)
DECLARE <name> <table type> SEARCH KEY(<key list>)
```

In the second case, the table type must not include any search key definition.

Procedure or Function Parameters

The search key can also be specified as part of a parameter definition

```
CREATE PROCEDURE <proc> (IN <param> TABLE(<column list>) SEARCH KEY(<key list>))
CREATE PROCEDURE <proc> (IN <param> <table type> SEARCH KEY(<key list>))
```

In the second case, the table type must not include any search key definition.

The input sorted table variables are re-sorted on call, unless a sorted table variable with a compatible key was provided (in this case, no re-sorting is necessary).

Input sorted table variables cannot be modified within the procedure or the function.

For outermost calls, the result sets corresponding to output sorted table variables are sorted according to the search key, using the ORDER BY clause. Thus you can ensure that the output table parameters have a defined sequence of the data records.

For sub-calls, the sorted outputs can be assigned to any kind of table variable - unsorted, or sorted with another search key (this requires a copy and/or a resorting). The usual use case should be indeed an assignment to a sorted table variable with the same search key (this requires neither a copy nor a resorting).

Table Variable Search Operator And Binary Search

If you search by an initial part of the key or by the whole key, the binary search can be utilized. If you search by some additional fields, then first the binary search is applied to narrow down the search interval which is then scanned sequentially.

Examples based on the table LT above:

Search statement	Behavior
`:LT.SEARCH(B, 1)`	You search by column B. Binary search can be applied and the 1st data record is found.
`:LT.SEARCH((B, A), (1, 2))`	You search by columns B, A. Binary search can be applied and the 2nd data record is found.
`:LT.SEARCH((B, C), (1, 15))`	You search by columns B, C. Binary search can be applied only for column B (B = 1), because the column A, which would be the next search key column, is not provided. The binary search narrows down the search interval to 1..2 and this interval is searched sequentially for C = 200 and the 2nd data record is found.
`:LT.SEARCH(A, 1)`	You search by column A. Binary search cannot be applied at all because the first search key column B was not provided. The 3rd data record is found by sequential search.

Output of Table Search Operator

If there is a matching data record, the position of the 1st matching data record is returned. This is the same behavior as with unsorted table variables.

However, if you search by the complete search key (all search key columns are specified) and there is no matching record, a negative value is returned instead of NULL. The absolute value of the return value indicates the position where a data record with the specified key values would be inserted in to keep the sorting.

Examples based on the table LT above:

Search statement	Result
`:LT.SEARCH(B, 3)`	The full search key was not specified and there is no matching data record. The result is NULL.
`:LT.SEARCH((B, A, C), (1, 2, 20))`	The full search key was specified and there is no matching data record. The result is -3, because a data record having B = 1, A = 2, C = 20 would have to be inserted at position 3.

This allows you to insert a missing data record directly at the correct position. Otherwise the insert operator would have to search for this position once more.

Example:

Sample Code

```
DECLARE lt TABLE(key int, count int) SEARCH KEY(key);
DECLARE search_result int;
...
search_result = :lt.SEARCH(key, someval);
IF search_result > 0 THEN
  lt.count[search_result] = :lt.count[search_result] + 1;
ELSE
  :lt.INSERT((someval, 0), -search_result);
END IF;
```

Iterating over Records with the Same Key Value

The sorting allows you not only to access a single data record but also to iterate efficiently over data records with the same key value. Just as with the table variable search operator, you have to use the initial part of the search key or the whole search key.

Sample Code

A table variable has 3 search key columns and you iterate over data records having a specific key value combination for the first two search key columns.

```
DECLARE pos int;
DECLARE mytab TABLE (key1 int, key2 int, key3 int, value int) SEARCH
KEY(key1, key2, key3);
DECLARE keyval1, keyval2 int;
...
pos = :mytab.SEARCH((key1, key2), (keyval1, keyval2));
```

```
IF pos > 0 THEN
  WHILE :mytab.key1[pos] = keyval1 AND :mytab.key2[pos] = keyval2 DO
    -- do something with the record at position "pos"
    ...
    pos = pos + 1;
  END WHILE;
END IF;
```

SQLScript Table Variable Modification Operators

For sorted table variables, you can use all available table variable modification operators. However, on every modification, the system has to ensure that the sorting is not violated. This has the following consequences:

- Insert operator
  - The insert operator without explicit position specification inserts the data record(s) at the correct positions taking the sorting definition into account.
  - The insert operator with explicit position specification checks if the sorting would be violated. If so, an error is raised and no data is inserted.
  - When inserting a table variable into a sorted table variable with explicit position specification, the input table variable is not re-sorted, it must comply with the sorting definition.
  - The highest explicitly specified position for insertion is the current table variable size increased by one (otherwise, empty data records would be created, which may violate the sorting).
- Update operator/Table cell assignment
  - It is not allowed to modify a search key column
  - It is not allowed to modify not existing data records (this would lead to creation of new data records and possibly sorting violation).

As mentioned above, if a primary key is defined, then its uniqueness is checked as well.

Table Variable Assignments

You can use sorted table variables as assignment target just like unsorted table variables. The data records will always be re-sorted according to the search key. If a primary key is defined, the system checks if it is unique. Any ORDER BY clause in queries, the result of which is assigned to a sorted table variable, is irrelevant.

Limitations

- The following data types are not supported for the search key:
  - Spatial data types
  - LOB types
- Output of table functions cannot be defined as sorted table type.

8.3 Auto Type Derivation

Description

It is possible to declare a variable without specifying its type explicitly and let SQLScript determine the type automatically. This auto type derivation can be used for scalar variables, tables and arrays.

Syntax

Code Syntax

```
DECLARE <var> AUTO = <value>
```

Note

The default value is mandatory and cannot be omitted.

Note

The existing syntax for definition of scalar and table variables is expanded as follows:

Code Syntax

Local Auto Scalar Variables

```
DECLARE <sql_identifier> [{,<sql_identifier> }...] [CONSTANT] AUTO [NOT
NULL] <proc_default>
```

Code Syntax

Local Auto Table Variables

```
DECLARE <sql_identifier> [{,<sql_identifier> }...] [CONSTANT] AUTO
<proc_table_default>
```

Caution

Potential incompatibility

The new feature may introduce some problems with existing procedures or functions, since AUTO is now interpreted as a keyword with higher precedence than a table or a table type named `AUTO`. The workaround for this incompatibility is to use SCHEMA.AUTO or quoted "AUTO" to interpret it as table type.

Sample Code

Example of incompatibility

```
create table auto (a bigint);
declare tab1 auto = select 1 a, 2 b from dummy;
```

Sample Code

Workaround

```
-- assume that current schema is schema_x
create table auto (a bigint);
do begin
 declare tab1 "AUTO";
 declare tab2 schema_x.auto;
end;
```

Examples

Sample Code

```
declare var1 auto = 1.0;
declare arr1 auto = array(1, 2);
declare tab1 auto = select 1 as x from dummy;
```

Data Type Derivation

The derived type is determined by the type of the default value but is not always exactly same as the evaluated type of the default value in the assignment. If the type has a length, the maximum length will be used to improve flexibility.

Actual Type of Default Value	Derived Type for Auto Variable
VARCHAR(n)	VARCHAR(MAX_LENGTH)
NVARCHAR(n)	NVARCHAR(MAX_LENGTH)
ALHPANUM(n)	ALPHANUM(MAX_LENGTH)
VARBINARY(n)	VARBINARY(MAX_LENGTH)
DECIMAL(p, s)	DECIMAL
SMALLDECIMAL	DECIMAL

Scope and Limitations

Auto type can be used for SQLScript scalar and table variables with the following limitations:

- Auto type cannot be used inside a trigger
- Auto type cannot be used for row-type variables
- Auto type cannot be used, if the default value contains one of the following:
  - System variables
  - Scalar access of any table or auto-type table

8.4 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>;
<key> ::= <string_literal> | <string_variable>
<value> ::= <scalar_expression>
```

While `<key>` can only be a constant string or a scalar variable, `<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 illustrate how you can set the value of a session variable in a procedure:

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

```
CREATE PROCEDURE CHANGE_SESSION_VAR (IN NEW_VALUE NVARCHAR(50), IN KEY_NAME
NVARCHAR(50))
AS
BEGIN
                SET :key_name =   :new_value || ‘_suffix’;
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* on the SAP Help Portal.

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;
```

> **i Note**
>
> `SET <key> = <value>` cannot be used in functions and procedures flagged as READ ONLY (scalar and table functions are implicitly READ ONLY).

> **i Note**
>
> The maximum number of session variables can be configured with the configuration parameter `max_session_variables` under the section session (min=1, max=5000). The default is 1024.

> **i 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*.

8.5 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

> **i 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]
```

8.6 Control Structures

8.6.1 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 how to use 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, 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. No `ELSE`-branches or `ELSEIF`-branches are allowed after an else branch.

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.

The predicate `x [NOT] BETWEEN lower AND upper` can also be used within the expression <bool_expr1>. It works just like `[ NOT ] ( x >= lower AND x <= upper)`. For more information, see Example 4.

Examples

Example 1

You use the `IF` statement to implement the functionality of the `UPSERT` statement in SAP HANA database.

```
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;
```

Example 4

Use of the BETWEEN operator

```
CREATE FUNCTION between_01(x INT)
RETURNS result NVARCHAR(1) AS
BEGIN
  IF :x BETWEEN 0 AND 100 THEN
    result = 'X';
  ELSE
    result = 'O';
  END IF;
END;
```

Related Information

ins_msg_proc [page 321]

8.6.2 Loop

Description

You use LOOP to repeatedly execute a set of statements. LOOP is identical with an infinite loop and it is necessary to implement finite logic by using BREAK or RETURN.

Syntax

Code Syntax

```
LOOP [ SEQUENTIAL EXECUTION ]
    [ <proc_decl_list> ]
    [ <proc_handler_list> ]
    <proc_stmt_list>
```

```
    END LOOP;
```

Related Information

CREATE PROCEDURE [page 19]

8.6.3 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.

The predicate `x [NOT] BETWEEN lower AND upper` can also be used within the expression of the `<condition>`. It works just like `[ NOT ] ( x >= lower AND x <= upper)`. For more information, see Example 3.

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;
```

Example 3

```
CREATE FUNCTION between_03(x INT)
RETURNS result NVARCHAR(1) AS
BEGIN
  DECLARE idx INT = :x;
  result = 'O';

  WHILE :idx BETWEEN 5 AND 15 DO
    idx = :idx + 1;
    result = 'X';
  END WHILE;
END;
```

> ⚠ Caution
>
> No specific checks are performed to avoid infinite loops.

8.6.4 For Loop

Syntax:

FOR - IN Loop iterates over a set of data:

```
FOR <loop-var> IN [REVERSE] <start_value> .. <end_value> DO [SEQUENTIAL
EXECUTION][<proc_decl_list>] [<proc_handler_list>]
```

```
    <proc_stmts>
END FOR
```

FOR - EACH Loop iterates over all rows from a cursor:

```
FOR <loop-var> AS <loop-var> [<open_param_list>] DO [SEQUENTIAL EXECUTION]
[<proc_decl_list>] [<proc_handler_list>]
    <proc_stmts>
END FOR
<open_param_list> ::= ( <expression> [ { , <expression> }...] )
```

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> ::= <expression>
```

Defines the starting value of the loop.

```
<end_value> ::=  <expression>
```

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>` equals `<end_value>` .

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

Example

You can use scalar UDF in the loop boundary values, 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;
```

8.6.5 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 321]

8.6.6 Operators

8.6.6.1 IN Operator

Description

SQLScript supports the use of IN clauses as conditions in IF or WHILE statements. Just like in standard SQL, the condition can take one of the following forms:

- a list of expressions on the left-hand side and a list of lists of expressions on the right-hand side
- a list of expressions on the left-hand side and a subquery on the right-hand side

In both cases, the numbers and types of entries in each list of the respective row of the result set on the right-hand side must match the numbers and types of entries on the left-hand side.

Examples

Sample Code

Pseudo Code Examples

```
-- single expression on the left-hand side
IF :i IN (1, 2, 3, 6, 8, 11, 12, 100) THEN
[...]
END IF;


-- multiple expressions on the left-hand side
IF (:key, :val) NOT IN ((1, 'H2O'), (2, 'H2O'), (3, 'abc'), (5, 'R2D2'), (6,
'H2O'), (7, 'H2O')) THEN
[...]
END IF;


-- subquery on the right-hand side
IF :i NOT IN (SELECT a FROM mytable) THEN
[...]
END IF;


-- subquery using table variable
IF (:a, :b, :c) IN (SELECT id, city, date from :lt where :id < :d) THEN
[...]
END IF;

-- subquery using table function
FOR i IN 1 .. CARDINALITY(:arr) DO
  IF :arr[:i] IN (SELECT b FROM tfunc()) THEN
  [...]
```

```
  END IF;
END FOR;
```

Limitations

Floating-point numbers, variables, and expressions can be used but due to the implementation of these data types, the results of the calculations may be inaccurate. For more information, see the chapter Numeric Data Types in the SAP HANA SQL and System Views Reference.

8.6.6.2 EXISTS Operator

SQLScript supports the use of EXISTS clauses as conditions in IF and WHILE statements. Just like in standard SQL, it evaluates to true if the sub-query returns a non-empty result set, and to false in any other case.

```
IF EXISTS (SELECT * FROM mytab WHERE date = :d) THEN
...
END IF

--


IF NOT EXISTS (SELECT * FROM SYS.TABLES WHERE schema_name = :schema AND
table_name = :table) THEN
...
END IF


--
WHILE :i < 100 AND EXISTS (SELECT * FROM mytab WHERE a = :i) DO
  i = :i + 1;
 ...
END WHILE


--
WHILE NOT EXISTS (SELECT * FROM mytab WHERE a > sfunc(:z).r2) DO
...
END WHILE
```

8.6.6.3 BETWEEN Operator

The predicate `x [NOT] BETWEEN lower AND upper` can be used within the expression of the `<condition>` of a WHILE loop. It works just like `[ NOT ] ( x >= lower AND x <= upper)`.

Example

Sample Code

```
CREATE FUNCTION between_03(x INT)
RETURNS result NVARCHAR(1) AS
BEGIN
DECLARE idx INT = :x;
result = 'O';
WHILE :idx BETWEEN 5 AND 15 DO
idx = :idx + 1;
result = 'X';
END WHILE;
END;
```

Related Information

While Loop [page 144]

8.7 Cursors

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

8.7.1 Define Cursor

Syntax:

```
DECLARE CURSOR <cursor_name> [({<param_def>{,<param_def>} ...)] [<holdability>
HOLD]
           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 data type of the parameter.

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

Defines an SQL select statement. See SELECT.

Defines cursor holdability

```
<holdability> := WITH | WITHOUT
```

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;
```

> **Sample Code**
>
> Example for Cursor Holdability
>
> ```
> CREATE TABLE mytab (col INT);
> INSERT INTO mytab VALUES (10);
> CREATE PROCEDURE testproc AS BEGIN
>     DECLARE i INT;
>     DECLARE CURSOR mycur WITH HOLD FOR SELECT * FROM mytab;
>     OPEN mycur;
>     ROLLBACK;
>     FETCH mycur INTO i;
>     CLOSE mycur;
>     SELECT :i as i FROM DUMMY;
> END;
>
>
> CALL testproc; -- Expected Result: {10}
> ```

Related Information

SELECT Statement (Data Manipulation)

8.7.2 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. If the cursor definition contains parameters, the actual values for each of these parameters should be provided when the cursor is opened.

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

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');
```

8.7.3 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;
```

8.7.4 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 moves the cursor to the next row. It is assumed that the cursor was declared and opened before. You can use the cursor attributes to check if the cursor points to a valid row.

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 154]

8.7.5 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.

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 321]

8.7.6 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 displayed syntax.

Description

Opens a previously declared cursor and iterates over each row in the result set of the query, bound to the cursor. The statements in the body of the procedure are executed for each row in the result set. 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 that `<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 how to use 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 321]

8.7.7 Updatable Cursor

Syntax

```
UPDATE <target_table> [ [ AS ] <correlation_name> ]
    SET <set_clause_list>
    WHERE CURRENT OF <cursor_name>
DELETE FROM <target_table> [ [ AS ] <correlation_name> ]
    WHERE CURRENT OF <cursor_name>
```

Description

When you iterate over each row of a result set, you can use the updatable cursor to change a record directly on the row, to which the cursor is currently pointing. The updatable cursor is a standard SQL feature (ISO/IEC 9075-2:2011).

For more information, see sections 14.8 & 14.13 in the SQL standard documentation (ISO/IEC 9075-2:2011).

Restrictions

The following restrictions apply:

- The cursor has to be declared with a SELECT statement having the FOR UPDATE clause in order to prevent concurrent WRITE on tables (without FOR UPDATE, the cursor is not updatable)
- The updatable cursor may be used only for UPDATE and DELETE operations.
- Using an updatable cursor in a single query instead of SQLScript is prohibited.
- Only persistent tables (both ROW and COLUMN tables) can be updated with an updatable cursor.
- UPDATE or DELETE operations performed on a table by means of an updatable cursor are allowed only one time per row.

> **Note**
>
> Updating the same row multiple times is possible, if several cursors selecting the same table are declared within a single transaction.

Examples

Example for updating a single table by using an updatable cursor:

Sample Code

```
CREATE TABLE employees (employee_id INTEGER, employee_name VARCHAR(30));
INSERT INTO employees VALUES (1, 'John');
INSERT INTO employees VALUES (20010, 'Sam');
INSERT INTO employees VALUES (21, 'Julie');
INSERT INTO employees VALUES (10005, 'Kate');

DO BEGIN
    DECLARE CURSOR cur FOR SELECT * FROM employees FOR UPDATE;
    FOR r AS cur DO
        IF r.employee_id < 10000 THEN
            UPDATE employees SET employee_id = employee_id + 10000
            WHERE CURRENT OF cur;
        ELSE
            DELETE FROM employees WHERE CURRENT OF cur;
        END IF;
  END FOR;
END;
```

Example for updating or deleting multiple tables (currently COLUMN tables only supported) by means of an updatable cursor.

> **Note**
>
> In this case, you have to specify columns of tables to be locked by using the FOR UPDATE OF clause within the SELECT statement of the cursor. Keep in mind that DML execution by means of an updatable cursor is allowed only one time per row.

Sample Code

```
CREATE COLUMN TABLE employees (employee_id INTEGER, employee_name
VARCHAR(30), department_id INTEGER);
INSERT INTO employees VALUES (1, 'John', 1);
INSERT INTO employees VALUES (2, 'Sam', 2);
INSERT INTO employees VALUES (3, 'Julie', 3);
INSERT INTO employees VALUES (4, 'Kate', 4);

CREATE COLUMN TABLE departments (department_id INTEGER, department_name
VARCHAR(20));
INSERT INTO departments VALUES (1, 'Development');
INSERT INTO departments VALUES (2, 'Operation');
INSERT INTO departments VALUES (3, 'HR');
INSERT INTO departments VALUES (4, 'Security');

DO BEGIN
    DECLARE CURSOR cur FOR SELECT employees.employee_name,
departments.department_name
        FROM employees, departments WHERE employees.department_id =
departments.department_id
        FOR UPDATE OF employees.employee_id, departments.department_id;
    FOR r AS cur DO
        IF r.department_name = 'Development' THEN
            UPDATE employees SET employee_id = employee_id + 10000,
department_id = department_id + 100
            WHERE CURRENT OF cur;
            UPDATE departments SET department_id = department_id + 100
            WHERE CURRENT OF cur;
        ELSEIF r.department_name = 'HR' THEN
            DELETE FROM employees WHERE CURRENT OF cur;
            DELETE FROM departments WHERE CURRENT OF cur;
        END IF;
    END FOR;
END;
```

8.7.8 Cursor Holdability

Syntax

```
DECLARE CURSOR cursor_name [(<parameter>)] [<holdability> HOLD] FOR ...
<holdability> := WITH | WITHOUT HOLD
```

Description

It is now possible to use control features directly within SQLScript in order to control cursor holdability for specific objects instead of using a system configuration, as it was necessary before.

Expression	Description
DECLARE CURSOR cursor_name WITH HOLD FOR ...	Declares a cursor with holdability for both commit and roll-back
DECLARE CURSOR cursor_name WITHOUT HOLD FOR ...	Declares a cursor without holdability for both commit and rollback
DECLARE CURSOR cursor_name FOR ...	Declares a cursor with holdability for commit and without holdability for rollback

Controlling the cursor holdability by cursor declaration gets higher priority than system configuration:

Configuration	Declaration	Result
WITHOUT HOLD	WITH HOLD	WITH HOLD
WITH HOLD	WITHOUT HOLD	WITHOUT HOLD
WITHOUT HOLD	WITHOUT HOLD	WITHOUT HOLD
WITH HOLD	WITH HOLD	WITH HOLD

If a cursor is holdable for commit and not holdable for rollback, it will have holdability for rollback after commit. A not holdable cursor will be invalidated by transactional operations (commit or rollback), but not closed. It will return a null value for fetch operations rather than throwing an exception and an exception will be thrown by using an updatable cursor.

Example

Sample Code

```
CREATE TABLE mytab (col INT);
INSERT INTO mytab VALUES (10);
CREATE PROCEDURE testproc AS BEGIN
    DECLARE i INT;
    DECLARE CURSOR mycur WITH HOLD FOR SELECT * FROM mytab;
    OPEN mycur;
    ROLLBACK;
    FETCH mycur INTO i;
    CLOSE mycur;
    SELECT :i as i FROM DUMMY;
END;


CALL testproc; -- Expected Result: {10}
```

Restrictions

It is currently not possible to use an updatable cursor while the cursor is holdable on rollback, since DML operations using an updatable cursor after rollback may cause unexpected results.

8.8 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

> i 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.

8.9 Transactional Statements

8.9.1 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:

V	ID
0	1

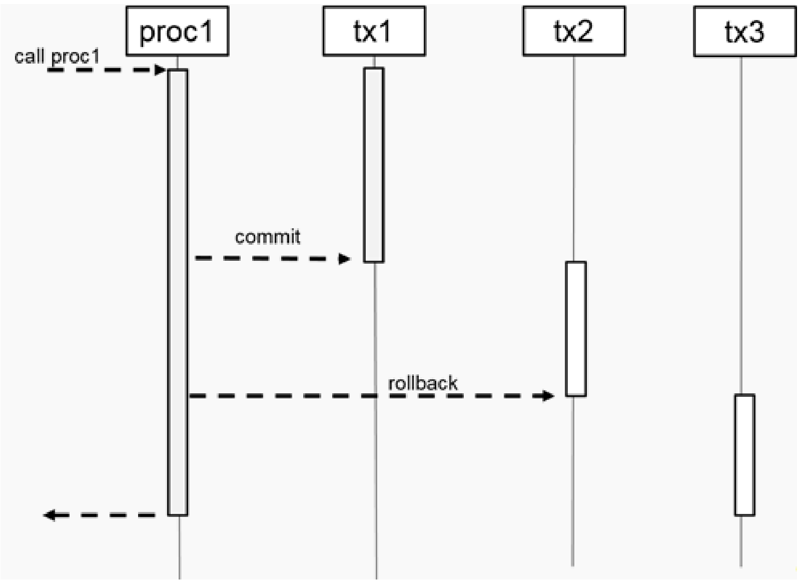
After you execute the PROC1 procedure, the B_TAB table is updated as follows:

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.

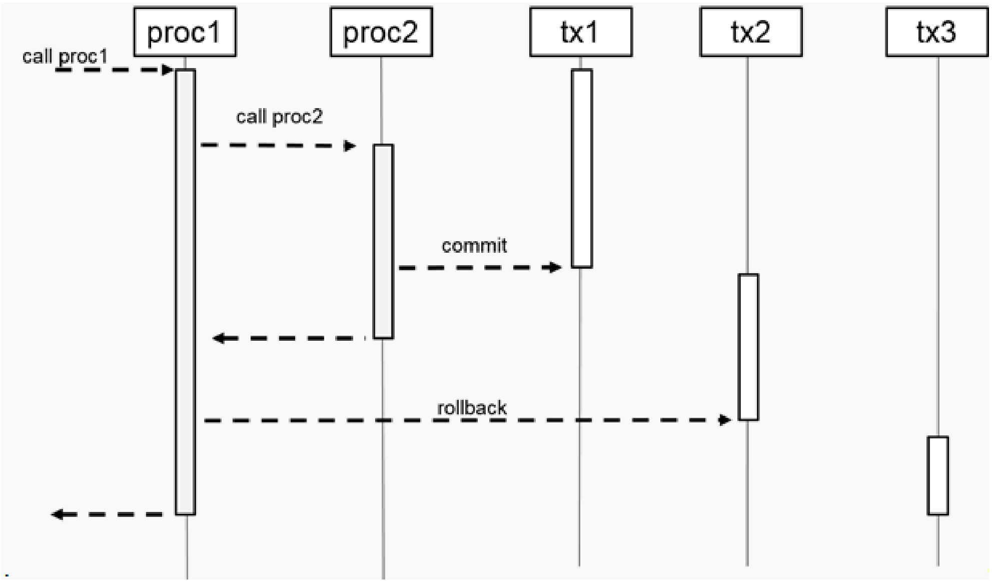
Example 2:

```
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.



> **i 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.

8.9.2 SAVEPOINT

SQLScript now supports transactional savepoints that allow the rollback of a transaction to a defined point. This includes:

- the definition of a SAVEPOINT: `SAVEPOINT <name>`
- the rollback to a specific SAVEPOINT: `ROLLBACK TO SAVEPOINT <name>`
- and the releasing of a SAVEPOINT: `RELEASE SAVEPOINT <name>`

Limitation

SAVEPOINT is a transactional statement, such as COMMIT or ROLLBACK. Therefore, the limitations of transactional statements apply to SAVEPOINT as well.

Example

```
drop table t1;
create table t1( i1 int );
create or replace procedure test
as begin
 insert into t1 values(1);
 SAVEPOINT save1;
 insert into t1 values(2);
 ROLLBACK TO SAVEPOINT save1;
 select * from t1;
 RELEASE SAVEPOINT save1;
end;
call test; -- result: {1}
select * from t1; -- result: {1}
```

8.10 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 may not be supported in SQLScript and provides more flexibility when creating SQL statements, it does have some disadvantages at run time:

- Opportunities for optimizations are limited.
- The statement is potentially recompiled every time the statement is executed.
- You must be very careful to avoid SQL injection bugs that might harm the integrity or security of the database.

> **i Note**
>
> You should avoid dynamic SQL wherever possible as it may have negative effects on security or performance.

8.10.1 EXEC

Syntax

```
EXEC '<sql-statement>' [INTO <var_name_list> [DEFAULT <scalar_expr_list>]]
[USING <expression_list>] [READS SQL DATA]
```

Description

EXEC executes the SQL statement `<sql-statement>` passed in a string argument. `EXEC` does not return any result set, if `<sql_statement>` is a `SELECT` statement. You have to use `EXECUTE IMMEDIATE` for that purpose.

Related Information

USING and INTO Clauses in DSQL [page 166]
EXECUTE IMMEDIATE [page 165]

8.10.2 EXECUTE IMMEDIATE

Syntax

```
EXECUTE IMMEDIATE '<sql-statement>' [INTO <var_name_list> [DEFAULT
<scalar_expr_list>]] [USING <expression_list>] [READS SQL DATA]
```

Description

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

You can also use the `INTO` and `USING` clauses to pass scalar and table values in or out. Result sets assigned to variables via INTO clause are not appended to the procedure result iterator.

When the suffix `READS SQL DATA` is attached, the statement is considered read-only. Since it is not possible to check at compile time whether the statement that is about to be executed is read-only, the operation returns a run-time error, if the executed statement is not read-only. The read-only declaration has the following advantages:

- DSQL can be used in a read-only context, for example read-only procedures and table user-defined functions
- read-only DSQL can be parallelized with other read-only operations thus improving the overall execution time.

To avoid the repetition of the suffix READS SQL DATA for every DSQL statement in a read-only procedure or a function, the DSQL will automatically be considered read-only, regardless of the suffix. However, it is still possible to add the suffix.

```
CREATE PROCEDURE Proc1(IN A NVARCHAR(12)) READS SQL DATA as
BEGIN
    EXEC 'SELECT * FROM ' || :A;
END
```

Example

You use dynamic SQL to delete the contents of the 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;
```

Related Information

EXEC [page 165]
USING and INTO Clauses in DSQL [page 166]

8.10.3 USING and INTO Clauses in DSQL

This feature introduces additional support for parameterized dynamic SQL. It is possible to use scalar variables, as well as table variable in USING and INTO clauses and CALL-statement parameters with USING and INTO clauses. You can use the INTO and USING clauses to pass in or out scalar or tabular values. Result sets, assigned to variables by means of the INTO clause, are not appended to the procedure result iterator.

Syntax

```
EXEC '<sql-statement>' [INTO <var_name_list>] [USING <expression_list>];
EXECUTE IMMEDIATE '<sql-statement>' [INTO <var_name_list>] [USING
<expression_list>];
<var_name_list> ::= <var_name> [{, <var_name>} ...]
<var_name> ::= <identifier>
<expression_list> ::= <expression> [{, <expression>} ...]
```

Description

EXEC executes the SQL statement `<sql-statement>` passed as a string argument. EXEC does not return a result set, if `<sql_statement>` is a SELECT-statement. You have to use EXECUTE IMMEDIATE for that purpose.

If the query returns result sets or output parameters, you can assign the values to scalar or table variables with the INTO clause.

When the SQL statement is a SELECT statement and there are table variables listed in the INTO clause, the result sets are assigned to the table variables sequentially. If scalar variables are listed in the INTO clause for a SELECT statement, it works like `<select_into_stmt>` and assigns the value of each column of the first row to a scalar variable when a single row is returned from a single result set. When the SQL statement is a CALL statement, output parameters represented as `':<var_name>'` in the SQL statement are assigned to the variables in the INTO clause that have the same names.

Examples

Sample Code

INTO Example 1

```
DO (IN tname NVARCHAR(10) => 'mytable')
BEGIN
  DECLARE tv TABLE (i INT);
  EXEC 'select col1 * 10 as i from ' || :tname INTO tv;
  SELECT * FROM :tv;
END;
```

Sample Code

INTO Example 2

```
DO (IN TNAME NVARCHAR(10)  =>'mytable',
    IN CNAME1 NVARCHAR(10) => 'I',
    IN CNAME2 NVARCHAR(10) => 'A',
    OUT K INT =>?, OUT J INT => ?)
BEGIN
  EXEC 'select max(' || :cname1 || ') as a, min(' ||:cname2 ||') as b from
'|| :TNAME INTO K, J ;
```

```
END;
```

Sample Code

INTO Example 3

```
CREATE PROCEDURE myproc (OUT i INT, OUT ot TABLE (i INT))
AS BEGIN
  ...
END;

DO (OUT a INT => ?, OUT tv TABLE (i INT) => ?)
BEGIN
  EXEC 'call myproc(:a, :tv)' INTO a, tv;
END;
```

You can also bind scalar or table values with the USING clause. When `<sql-statement>` uses `':<var_name>'` as a parameter, only variable references are allowed in the USING clause and variables with the same name are bound to the parameter `':<var_name>'`. However, when `<sql-statement>` uses `'?'` as a parameter (unnamed parameter bound), any expression is allowed in the USING clause and values are mapped to parameters sequentially. The unnamed parameter bound is supported when there are only input parameters.

Sample Code

USING Example 1

```
DO BEGIN
  DECLARE tv TABLE (col1 INT) = SELECT * FROM mytab;
  DECLARE a INT = 123;
  DECLARE tv2 TABLE (col1 INT);
  EXEC 'select col1 + :a as col1 from :tv' INTO tv2 USING :a, :tv;
  SELECT * FROM :tv2;
END;
```

Sample Code

USING Example 2

```
DO (IN TNAME NVARCHAR(10)  =>'mytable',
    IN CNAME1 NVARCHAR(10) => 'I',
    IN CNAME2 NVARCHAR(10) => 'A',
    OUT K INT =>?, OUT J INT => ?)
BEGIN
  DECLARE a INT = 2;
  DECLARE b INT = 3;
  EXEC 'select max(' || :cname1 || ') + ? * ? as a, min(' || :cname2 || ') as
b from ' || :TNAME INTO K, J  USING :a, :b;
END;
```

Sample Code

USING Example 3

```
CREATE PROCEDURE myproc (IN i INT, IN itv TABLE (col1 INT))
AS BEGIN
  ...
```

```
END;

DO BEGIN
  DECLARE tv TABLE (col1 INT) = SELECT * FROM mytab;
  DECLARE a INT = 123;
  EXEC 'call myproc(:a, :tv)' USING :a, :tv;
END;
```

Limitations

A table variable cannot be used in both an INTO-clause and a USING-clause.

The parameter '?' only works with scalar input parameters.

The parameter '?' and the variable reference ':<var_name>' cannot be used at the same time in an SQL statement.

8.10.4 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 to a table or a table variable. In terms of logic, it can be considered a partially dynamic SQL statement. The advantage of the function is that you can assign it to a table variable and that will not block SQL inlining.

Caution

The disadvantage of APPLY_FILTER is the missing parametrization capability. Using constant values always leads 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 spent for query preparation and potential cache flooding effects in fast-changing parameter value scenarios. To avoid this, we recommend to use EXEC with USING clause to make use of a parametrized WHERE-clause.

Sample Code

Before:

```
v_filter = :column || ' = ''' || :value || '''';
lt = APPLY_FILTER(:lt0, :v_filter);
```

Sample Code

After:

```
EXEC 'SELECT * FROM :lt0 WHERE (' || :column || ' = :value' INTO lt
USING :lt0, :value READS SQL DATA;
```

Examples

Example 1: Applying a filter to a persistent table

You create the following procedure

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

8.11 Exception Handling

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

8.11.1 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;
```

There are two system variables ::SQL_ERROR_CODE and ::SQL_ERROR_MESSAGE that can be used to get the error code and the error message, as 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, the code looks as follows:

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

The following 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.

Type	Description
SQL Error Code	Code strings starting with ERR_SQL_*
SQLScript error code	Code strings starting with ERR_SQLSCRIPT_*
Transactional error code	ERR_TX_ROLLBACK_LOCK_TIMEOUT ERR_TX_ROLLBACK_DEADLOCK ERR_TX_SERIALIZATION ERR_TX_LOCK_ACQUISITION_FAIL
User error code	User error code

When catching transactional errors, the transaction still lives inside the EXIT HANDLER. That allows the explicit use of COMMIT or ROLLBACK.

i Note

It is now possible to define an exit handler for the statement FOR UPDATE NOWAIT with the error code 146. For more information, see Supported Error Codes [page 182].

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;
```

For more information about declaring a condition, see DECLARE CONDITION [page 176].

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;
```

i Note

In the example above, in case of an unhandled exception the transaction will be rolled back. Thus the new row in the table `LOG_TABLE` will be gone as well. To avoid this, you can use an autonomous transaction. For more information, see Autonomous Transaction [page 160].

8.11.2 DECLARE CONTINUE HANDLER

Description

The EXIT handler in SQLScript already offers a way to process exception conditions in a procedure or a function during execution. The CONTINUE handler not only allows you to handle the error but also to continue with the execution after an exception has been thrown.

⚠ Caution

Triggers are not supported inside CONTINUE HANDLER.

Syntax

Code Syntax

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

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

Behavior

The behavior of the CONTINUE handler for catching and handling exceptions is the same as that of the EXIT handler with the following exceptions and extensions.

Continue After Handling

SQLScript execution continues with the statement following the exception-throwing statement right after catching and handling the exception.

Sample Code

```
DO BEGIN
    DECLARE A INT = 10;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION BEGIN -- Catch the exception
        SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
    END;
    A = 1 / 0; -- An exception will be thrown
    SELECT :A FROM DUMMY; -- Continue from this statement after handling the
exception
END;
```

In multilayer blocks, SQLScript execution continues with the next statement in the inner-most block after the exception-throwing statement.

Sample Code

```
DO BEGIN
    DECLARE A INT = 10;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY; -- Catch the
exception
    SELECT :A FROM DUMMY;
    BEGIN
        A = 1 / 0; -- An exception throwing
        A = :A + 1; -- Continue from this statement after handling the
exception
    END;
    SELECT :A FROM DUMMY; -- Result: 11
END;
```

Block Parallel Execution

It is difficult to determine which statement is the statement following an error-throwing statement in parallel execution blocks. Some of the statements may have already been executed before the exception occurs.

For this reason, implicit or explicit parallel execution is not supported within the scope of a continue handler.

Sample Code

```
CREATE PROCEDURE PROC READS SQL DATA AS BEGIN
    SELECT * FROM DUMMY;
END;


DO BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY; -- Catch the
exception
    BEGIN PARALLEL EXECUTION -- not supported
        CALL PROC;
        CALL PROC;
        CALL PROC;
    END;
END;
```

Handling of Conditional Statements

If there is an error in a conditional statement for an IF, a WHILE, or a FOR block, the whole block will be skipped after handling the error because the condition is no longer valid.

Sample Code

```
DO BEGIN
    DECLARE A INT = 0;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
    IF A = 1 / 0 THEN -- An error occurs
        A = 1;
    ELSE
        A = 2;
    END IF;
    SELECT :A FROM DUMMY; -- Continue from here, Result: 0
END;
```

Exit Handlers and Continue Handlers

EXIT handlers cannot be declared within the same scope or within a nested scope of a CONTINUE handler, but CONTINUE handlers can be declared in the nested scope of an EXIT handler.

Sample Code

```
DO BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY; -- OK
    BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY; -- Checker error
thrown
        DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
        BEGIN
```

```
            DECLARE EXIT HANDLER FOR SQLEXCEPTION
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY; -- Checker error
thrown
            SELECT 1 / 0 FROM DUMMY;
        END;
    END;
END;
```

Variable Values

The value of the variable remains as it was before the execution of the statement that returns an exception.

Sample Code

```
CREATE TABLE TAB (I INT);
DO BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION BEGIN END;
    INSERT INTO TAB VALUES (1);
    INSERT INTO TAB VALUES (1 / 0); -- An error thrown
    SELECT ::ROWCOUNT FROM DUMMY; -- 1, not 0
END;


DO BEGIN
    DECLARE CONTINUE HANDLER FOR SQL_ERROR_CODE 12346 BEGIN END;
    BEGIN
        DECLARE CONTINUE HANDLER FOR SQL_ERROR_CODE 12345 BEGIN
            SIGNAL SQL_ERROR_CODE 12346;
            SELECT ::SQL_ERROR_CODE FROM DUMMY; -- 12346, not 12345
        END;
        SIGNAL SQL_ERROR_CODE 12345;
    END;
END;


DO BEGIN
    DECLARE A INT = 10;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION BEGIN
        SELECT :A FROM DUMMY; -- Result: 10
    END;
    A = 1 / 0;
    SELECT :A FROM DUMMY; -- Result: 10
END;
```

8.11.3 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;
```

> **i 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 177].

8.11.4 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 176]. The used error code must be within the user-defined range of 10000 to 19999.

For example, to signal an SQL_ERROR_CODE 10000, proceed as follows:

```
SIGNAL SQL_ERROR_CODE 10000;
```

To raise a user-defined condition, for example invalid_input, as declared in the previous section (see DECLARE CONDITION [page 176]), use the following command:

```
SIGNAL invalid_input;
```

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

In both cases you get the following information in case the user-defined exception is 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`:

```
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 following 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;
```

If the procedures are called 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
```

For more information on how to handle the exception and continue with procedure execution, see Nested Block Exceptions in Exception Handling Examples [page 179].

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.

You can change the error message of an SQL error 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
```

You can get the original message via the system variable ::SQL_ERROR_MESSAGE. This is useful, if you still want to keep the original message, but would 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;
```

8.11.5 Exception Handling Examples

General Exception Handling

A general exception can be handled with an exception handler declared at the beginning of a statement that makes an explicit or an 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

You can declare an exception handler that catches exceptions with 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 by 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;
```

8.11.6 Supported Error Codes

The following is a list of the error codes supported by the exit handler.

Code	Type	Description
131	ERR_TX_ROLLBACK_LOCK_TIMEOUT	transaction rolled back by lock wait timeout
133	ERR_TX_ROLLBACK_DEADLOCK	transaction rolled back by detected deadlock
138	ERR_TX_SERIALIZATION	transaction serialization failure
256	ERR_SQL	sql processing error
257	ERR_SQL_PARSE	sql syntax error
258	ERR_SQL_INSUFF_PRIV	insufficient privilege
259	ERR_SQL_INV_TABLE	invalid table name
260	ERR_SQL_INV_COLUMN	invalid column name
261	ERR_SQL_INV_INDEX	invalid index name
262	ERR_SQL_INV_QUERY	invalid query name
263	ERR_SQL_INV_ALIAS	invalid alias name
264	ERR_SQL_INV_DATATYPE	invalid datatype
265	ERR_SQL_MISSING_EXP	expression missing
266	ERR_SQL_INCNST_DATATYPE	inconsistent datatype
267	ERR_SQL_LONG_LEN_TYPE	specified length too long for its data-type
268	ERR_SQL_AMBG_COLUMN	column ambiguously defined
269	ERR_SQL_MANY_VALUES	too many values
270	ERR_SQL_FEW_VALUES	not enough values
271	ERR_SQL_DPLC_ALIAS	duplicate alias
272	ERR_SQL_DPLC_COLUMN	duplicate column name
273	ERR_SQL_LONG_CHAR	not a single character string
274	ERR_SQL_INS_LARGE_VALUE	inserted value too large for column
275	ERR_SQL_NOT_FUNCTION	aggregate function not allowed
276	ERR_SQL_NOT_SINGLE_GROUP	missing aggregation or grouping

Code	Type	Description
277	ERR_SQL_NOT_GROUP_EXP	not a GROUP BY expression
278	ERR_SQL_NESTED_WO_GROUP	nested group function without GROUP BY
279	ERR_SQL_TOO_DEEP_NESTED	group function is nested
280	ERR_SQL_ORDER_EXCEED_NUM	ORDER BY item must be the number of a SELECT-list
281	ERR_SQL_OUTER_IN_OR	outer join not allowed in operand of OR or IN
282	ERR_SQL_OUTER_CROSS_JOIN	two tables cannot be outer-joined to each other
283	ERR_SQL_OUTER_MORE_TWO	a table may be outer joined to at most one other table
284	ERR_SQL_JOIN_NOT_MATCH	join field does not match
285	ERR_SQL_INV_JOIN_PRED	invalid join condition
286	ERR_SQL_LONG_IDENTIFIER	identifier is too long
287	ERR_SQL_NOT_NULL	cannot insert NULL or update to NULL
288	ERR_SQL_EXST_TABLE	cannot use duplicate table name
289	ERR_SQL_EXST_INDEX	cannot use duplicate index name
290	ERR_SQL_EXST_QUERY	cannot use duplicate query name
291	ERR_SQL_NOT_POS_ARGUMENT	argument identifier must be positive
292	ERR_SQL_FEW_ARGUMENT	wrong number of arguments
293	ERR_SQL_INV_ARGUMENT	argument type mismatch
294	ERR_SQL_MANY_PRIMARY_KEY	cannot have more than one primary key
295	ERR_SQL_LONG_MULTIKEY	too long multi key length
296	ERR_SQL_REP_TABLE_KEY	replicated table must have a primary key
297	ERR_SQL_REP_UPDATE_KEY	cannot update primary key field in replicated table
298	ERR_SQL_NOT_DDL_STORE	cannot store DDL
299	ERR_SQL_NOT_DROP_SYSIDX	cannot drop index used for enforcement of unique/primary key
300	ERR_SQL_ARG_OUT_OF_RANGE	argument index is out of range
301	ERR_SQL_UNIQUE_VIOLATED	unique constraint violated
302	ERR_SQL_INV_CHAR_VAL	invalid CHAR or VARCHAR value
303	ERR_SQL_INV_DATETIME_VAL	invalid DATE, TIME or TIMESTAMP value
304	ERR_SQL_DIV_BY_ZERO	division by zero undefined

Code	Type	Description
305	ERR_SQL_SINGLE_ROW	single-row query returns more than one row
306	ERR_SQL_INV_CURSOR	invalid cursor
307	ERR_SQL_NUM_OUT_OF_RANGE	numeric value out of range
308	ERR_SQL_EXST_COLUMN	column name already exists
309	ERR_SQL_SUBQ_TOP_ORDERBY	correlated subquery cannot have TOP or ORDER BY
310	ERR_SQL_IN_PROC	sql error in procedure
311	ERR_SQL_DROP_ALL_COLUMNS	cannot drop all columns in a table
312	ERR_SQL_SEQ_EXHAUST	sequence is exhausted
313	ERR_SQL_INV_SEQ	invalid sequence
314	ERR_SQL_OVERFLOW_NUMERIC	numeric overflow
315	ERR_SQL_INV_SYNONYM	invalid synonym
316	ERR_SQL_INV_NUM_ARG_FUNC	wrong number of arguments in function invocation
317	ERR_SQL_NOT_MATCH_PLAN_TABLE	\"P_QUERYPLANS\" not exists nor valid format
318	ERR_SQL_DECIMAL_PRECISION	decimal precision specifier is out of range
319	ERR_SQL_DECIMAL_SCALE	decimal scale specifier is out of range
320	ERR_SQL_LOB_INDEX	cannot create index on expression with datatype LOB
321	ERR_SQL_INV_VIEW	invalid view name
322	ERR_SQL_EXST_VIEW	cannot use duplicate view name
323	ERR_SQL_REP_DPLC_ID	duplicate replication id
324	ERR_SQL_EXST_SEQ	cannot use duplicate sequence name
325	ERR_SQL_ESC_SEQ	invalid escape sequence
326	ERR_SQL_SEQ_CURRVAL	CURRVAL of given sequence is not yet defined in this session
327	ERR_SQL_CANNOT_EXPLAIN	cannot explain plan of given statement
328	ERR_SQL_INV_FUNC_PROC	invalid name of function or procedure
329	ERR_SQL_EXST_FUNC_PROC	cannot use duplicate name of function or procedure
330	ERR_SQL_EXST_SYNONYM	cannot use duplicate synonym name
331	ERR_SQL_EXST_USER	user name already exists
332	ERR_SQL_INV_USER	invalid user name
333	ERR_SQL_COLUMN_NOT_AL-LOWED_HERE	column not allowed

Code	Type	Description
334	ERR_SQL_INV_PRIV	invalid user privilege
335	ERR_SQL_EXST_ALIAS	field alias name already exists
336	ERR_SQL_INV_DEFAULT	invalid default value
337	ERR_SQL_INTO_NOT_ALLOWED	INTO clause not allowed for this SE-LECT statement
338	ERR_SQL_ZERO_LEN_NOT_ALLOWED	zero-length columns are not allowed
339	ERR_SQL_INV_NUMBER	invalid number
340	ERR_SQL_VAR_NOT_BOUND	not all variables bound
341	ERR_SQL_UNDERFLOW_NUMERIC	numeric underflow
342	ERR_SQL_COLLATE_CONFLICT	collation conflict
343	ERR_SQL_INV_COLLATE_NAME	invalid collate name
344	ERR_SQL_LOADER_PARSE	parse error in data loader
345	ERR_SQL_NOT_REP_TABLE	not a replication table
346	ERR_SQL_INV_REP_ID	invalid replication id
347	ERR_SQL_INV_OPTION	invalid option in monitor
348	ERR_SQL_INV_DATETIME_FORMAT	invalid datetime format
349	ERR_SQL_CREATE_UNIQUE_INDEX	cannot CREATE UNIQUE INDEX; dupli-cate key found
350	ERR_SQL_DROP_COL_PRIMARY_KEY	cannot drop columns in the primary-key column list
351	ERR_SQL_DROP_MULTI_COL_UNIQUE	column is referenced in a multi-column constraint
352	ERR_SQL_CREATE_UNIQUE_IN-DEX_ON_CDX_TAB	cannot create unique index on cdx table
353	ERR_SQL_EXST_UPDATE_LOG_GROUP	update log group name already exists
354	ERR_SQL_INV_UP-DATE_LOG_GROUP_NAME	invalid update log group name
355	ERR_SQL_UPDATE_LOG_TABLE_KEY	the base table of the update log table must have a primary key
356	ERR_SQL_MAX_UPDATE_LOG_GROUP	exceed maximum number of update log group
357	ERR_SQL_BASE_TABLE_AL-READY_HAS_ULT	the base table already has a update log table
358	ERR_SQL_ULT_CAN_NOT_HAVE_ULT	update log table can not have a update log table
359	ERR_SQL_STR_LENGTH_TOO_LARGE	string is too long
360	ERR_SQL_VIEW_CHECK_VIOLATION	view WITH CHECK OPTION where-clause violation

Code	Type	Description
361	ERR_SQL_VIEW_UPDATE_VIOLATION	data manipulation operation not legal on this view
362	ERR_SQL_INV_SCHEMA	invalid schema name
363	ERR_SQL_MAX_NUM_INDEX_COL-UMN	number of index columns exceeds its maximum
364	ERR_SQL_INV_PARTIAL_KEY_SIZE	invalid partial key size
365	ERR_SQL_NO_MATCH-ING_UNIQUE_OR_PRIMARY_KEY	no matching primary key for this col-umn list
366	ERR_SQL_NO_PRIMARY_KEY	referenced table does not have a pri-mary key
367	ERR_SQL_MISMATCH_OF_COL-UMN_NUMBERS	number of referencing columns must match referenced columns
368	ERR_SQL_TEMP_TA-BLE_WITH_UNIQUE	unique constraint not allowed on tem-porary table
369	ERR_SQL_MAX_VIEW_DEPTH	exceed maximum view depth limit
370	ERR_SQL_DIRECT_IN-SERT_WITH_UNIQUE_INDEX	cannot perform DIRECT INSERT opera-tion on table with unique indexes
371	ERR_SQL_XML_PARSE	invalid XML document
372	ERR_SQL_XPATH_PARSE	invalid XPATH
373	ERR_SQL_INV_XML_DURATION	invalid XML duration value
374	ERR_SQL_INV_XML_FUNCTION	invalid XML function usage
375	ERR_SQL_INV_XML_INDEX_OPERA-TION	invalid XML index operation
376	ERR_SQL_PYTHON	Python buildin procedure error
377	ERR_SQL_JIT	JIT operation error
378	ERR_SQL_INV_COLUMN_VIEW	invalid column view
379	ERR_SQL_TABLE_SCHEMA_MIS-MATCH	table schema mismatch
380	ERR_SQL_RUN_LEVEL_CHANGE	fail to change run level
381	ERR_SQL_RESTART	fail to restart
382	ERR_SQL_COLLECT_ALL_VERSIONS	fail to collect all version garbage
383	ERR_SQL_INV_IDENTIFIER	invalid identifier
384	ERR_SQL_TOO_LONG_CONSTANT	string is too long
385	ERR_SQL_RESTORE_SESSION	could not restore session
386	ERR_SQL_EXST_SCHEMA	cannot use duplicate schema name
387	ERR_SQL_AMBG_TABLE	table ambiguously defined
388	ERR_SQL_EXST_ROLE	role already exists
389	ERR_SQL_INV_ROLE	invalid role name

Code	Type	Description
390	ERR_SQL_INV_USERTYPE	invalid user type
391	ERR_SQL_INV_USABLE_VIEW	invalidated view
392	ERR_SQL_CYCLIC_ROLES	can't assign cyclic role
393	ERR_SQL_NO_GRANT_OP-TION_FOR_ROLE	roles must not receive a privilege with grant option
394	ERR_SQL_CANT_REVOKE_ROLE	error revoking role
395	ERR_SQL_INV_USER_DEFINED_TYPE	invalid user-defined type name
396	ERR_SQL_EXST_USER_DE-FINED_TYPE	cannot use duplicate user-defined type name
397	ERR_SQL_INV_OBJ_NAME	invalid object name
398	ERR_SQL_MANY_ORDER_BY	cannot have more than one order by
399	ERR_SQL_TOO_DEEP_ROLE_TREE	role tree too deep
400	ERR_SQL_INSERT_ONLY_TA-BLE_WITH_PRIMARY_KEY	primary key not allowed on insert-only table
401	ERR_SQL_INSERT_ONLY_TA-BLE_WITH_UNIQUE	unique constraint not allowed on insert-only table
402	ERR_SQL_DROPPED_USER	the user was already dropped before query execution
403	ERR_SQL_INTERNAL_ERROR	internal error
404	ERR_SQL_INV_STRUCTURED_PRIVI-LEGE_NAME	invalid (non-existent) structured privi-lege name
405	ERR_SQL_DUP_STRUCTURED_PRIVI-LEGE_NAME	cannot use duplicate structured privi-lege name
406	ERR_SQL_CANT_UPDATE_GEN_COL	INSERT, UPDATE and UPSERT are dis-allowed on the generated field
407	ERR_SQL_INV_DATE_FORMAT	invalid date format
408	ERR_SQL_PASS_OR_PARAME-TER_NEEDED	password or parameter required for user
409	ERR_SQL_TOO_MANY_PARAME-TER_VALUES	multiple values for a parameter not supported
410	ERR_SQL_INV_PRIVILEGE_NAME-SPACE	invalid privilege namespace
411	ERR_SQL_INV_TABLE_TYPE	invalid table type
412	ERR_SQL_INV_PASSWORD_LAYOUT	invalid password layout
413	ERR_SQL_PASSWORD_REUSED	last n passwords can not be reused
414	ERR_SQL_ALTER_PASS-WORD_NEEDED	user is forced to change password
415	ERR_SQL_USER_DEACTIVATED	user is deactivated
416	ERR_SQL_USER_LOCKED	user is locked; try again later

Code	Type	Description
417	ERR_SQL_CANT_DROP_WITH-OUT_CASCADE	can't drop without CASCADE specifica-tion
418	ERR_SQL_INV_VIEW_QUERY	invalid view query for creation
419	ERR_SQL_CANT_DROP_WITH_RE-STRICT	can't drop with RESTRICT specification
420	ERR_SQL_ALTER_PASS-WORD_NOT_ALLOWED	password change currently not allowed
421	ERR_SQL_FULLTEXT_INDEX	cannot create fulltext index
422	ERR_SQL_MIXED_PRIVILEGE_NAME-SPACES	privileges must be either all SQL or all from one namespace
423	ERR_SQL_LVC	AFL error
424	ERR_SQL_INV_PACKAGE	invalid name of package
425	ERR_SQL_EXST_PACKAGE	duplicate package name
426	ERR_SQL_NUM_COLUMN_MISMATCH	number of columns mismatch
427	ERR_SQL_CANT_RESERVE_INDEX_ID	can not reserve index id any more
429	ERR_SQL_INTEGRITY_CHECK_FAILED	integrity check failed
430	ERR_SQL_INV_USABLE_PROC	invalidated procedure
433	ERR_SQL_NOT_NULL_CONSTRAINT	null value found
434	ERR_SQL_INV_OBJECT	invalid object ID
435	ERR_SQL_INV_EXP	invalid expression
436	ERR_SQL_SET_SYSTEM_LICENSE	could not set system license
437	ERR_SQL_ONLY_LICENSE_HANDLING	only commands for license handling are allowed in current state
438	ERR_SQL_INVALID_USER_PARAME-TER_VALUE	invalid user parameter value
439	ERR_SQL_COMPOSITE_ERROR	composite error
440	ERR_SQL_TABLE_TYPE_CONVER-SION_ERROR	table type conversion error
442	ERR_SQL_MAX_NUM_COLUMN	number of columns exceeds its maxi-mum
443	ERR_SQL_INV_CALC_SCENARIO	invalid calculation scenario name
444	ERR_SQL_PACKMAN	package manager error
445	ERR_SQL_INV_TRIGGER	invalid trigger name
446	ERR_SQL_EXST_TRIGGER	cannot use duplicate trigger name
447	ERR_SQL_BACKUP_FAILED	backup could not be completed
448	ERR_SQL_RECOVERY_FAILED	recovery could not be completed
449	ERR_SQL_RECOVERY_STRATEGY	recovery strategy could not be deter-mined

Code	Type	Description
450	ERR_SQL_UNSET_SYSTEM_LICENSE	failed to unset system license
451	ERR_SQL_NOT_AL-LOWED_SUBJ_TAB_ACCESS_TRIGGER	modification of subject table in trigger not allowed
452	ERR_SQL_INV_BACKUPID	invalid backup id
453	ERR_SQL_USER_WITHOUT_PASS-WORD	user does not have a password
455	ERR_SQL_READ_ONLY_SES-SION_VARIABLE	the predefined session variable cannot be set via SET command
456	ERR_SQL_NOT_ALLOWED_FOR_SPE-CIAL_ROLE	not allowed for this role
457	ERR_SQL_DPLC_CONSTRAINT	duplicate constraint name
458	ERR_SQL_UNSUPPORTED_FUNCTION	unsupported function included
459	ERR_SQL_INV_USABLE_FUNC	invalidated function
460	ERR_SQL_INV_PRIVILEGE_FOR_OB-JECT	invalid privilege for object
461	ERR_SQL_FK_NOT_FOUND	foreign key constraint violation
462	ERR_SQL_FK_ON_UPDATE_DE-LETE_FAILED	failed on update or delete by foreign key constraint violation
463	ERR_SQL_MAX_NUM_TABLE	number of tables exceeds its maximum
464	ERR_SQL_MAX_PARSE_TREE_DEPTH	SQL internal parse tree depth exceeds its maximum
465	ERR_SQL_INV_USABLE_TRIGGER	Cannot execute trigger, was invalidated by object change
466	ERR_SQL_CREDENTIAL_NOT_FOUND	no credential found
467	ERR_SQL_PARAM_VARIABLE	cannot use parameter variable
468	ERR_SQL_HINT	hint error
469	ERR_SQL_INV_SRC_DATATYPE	unsupported datatype on source, con-sider using a view
470	ERR_SQL_INV_DATA_SOURCE_CONF	invalid data source configuration
471	ERR_SQL_INV_DATA_SOURCE	invalid data source name
472	ERR_SQL_EXST_DATA_SOURCE	cannot use duplicate data source name
473	ERR_SQL_ADAPTER_CONFIGURATION	invalid adapter configuration
474	ERR_SQL_INV_ADAPTER	invalid adapter name
475	ERR_SQL_EXST_ADAPTER	cannot use duplicate adapter name
476	ERR_SQL_INV_REMOTE_OBJECT	invalid remote object name
477	ERR_SQL_CREDENTIAL_EXISTS	credential exists
478	ERR_SQL_UDF_RUNTIME	user defined function runtime error
479	ERR_SQL_INV_SPATIAL_ATTRIBUTE	invalid spatial attribute

Code	Type	Description
480	ERR_SQL_INV_SPATIAL_UNIT	invalid spatial unit of measure name
481	ERR_SQL_EXST_SPATIAL_UNIT	cannot use duplicate spatial unit of measure name
482	ERR_SQL_INV_SPATIAL_REF_SYS	invalid spatial reference system name
483	ERR_SQL_EXST_SPATIAL_REF_SYS	cannot use duplicate spatial reference system name
484	ERR_SQL_SESSION_GROUP_COM-MAND_FAILURE	invalid session group command
485	ERR_SQL_INV_STRUCTURED_PRIVI-LEGE_DEFINITION	invalid definition of structured privilege
487	ERR_SQL_IMPORT_PARTIALLY_FAILED	some of rows have failed to be imported
488	ERR_SQL_INV_DATABASE	invalid database name
489	ERR_SQL_INV_EPMMODEL	invalid EPM Model name
490	ERR_SQL_EXST_EPMMODEL	cannot use duplicate EPM Model name
491	ERR_SQL_INV_EPMMODEL_DEF	invalid EPM Model definition
492	ERR_SQL_INV_EPMQUERYSOURCE	invalid EPM Query Source name
493	ERR_SQL_EXST_EPMQUERYSOURCE	cannot use duplicate EPM Query Source name
494	ERR_SQL_INV_EPMQUERY-SOURCE_DEF	invalid EPM Query Source definition
498	ERR_SQL_IMPORT_FAIL_ON_MAX_RE-CORD_SIZE_CHECK	Memory for a record exceeds the limit
499	ERR_SQL_INV_C2C	invalid stacked column search
500	ERR_SQL_REQUIRE_PREDICATE	predicates are required in a where clause
501	ERR_SQL_SERIES_INVALID_SPEC	Invalid series data specification:
502	ERR_SQL_INV_TASK	invalid name of task
503	ERR_SQL_EXST_TASK	cannot use duplicate name of task
504	ERR_SQL_INV_ADAPTER_LOCATION	invalid adapter location
505	ERR_SQL_LAST_ADAPTER_LOCATION	cannot remove last location of adapter, use DROP ADAPTER statement
506	ERR_SQL_SYSTEM_ADAPTER	invalid create, alter or drop system adapter
507	ERR_SQL_INV_AGENT	invalid agent name
508	ERR_SQL_EXST_AGENT	cannot use duplicate agent name
509	ERR_SQL_INV_AGENT_PROPS	invalid agent properties
510	ERR_SQL_TEMP_TABLE_IN_USE	cannot alter global temporary table in use or create/alter/drop index on the table

Code	Type	Description
640	ERR_SQL_2	sql processing error
641	ERR_SQL_INV_REMOTE_SUBSCRIP-TION	invalid remote subscription name
642	ERR_SQL_EXST_REMOTE_SUBSCRIP-TION	cannot use duplicate remote subscrip-tion name
643	ERR_SQL_INV_REMOTE_SUBSCRIP-TION_DEF	invalid remote subscription definition
644	ERR_SQL_EXST_RE-MOTE_SOURCE_ADAPTER_LOCATION	remote source refers to the adapter lo-cation
645	ERR_SQL_EXST_RE-MOTE_SOURCE_ACTIVE_SUBSCRIP-TIONS	remote source has active remote sub-scriptions:
646	ERR_SQL_INV_USABLE_TASK	invalidated task
647	ERR_SQL_NOT_ALLOWED_SYN-TAX_FOR_TRIGGER	not supported syntax in trigger
648	ERR_SQL_TRIG-GER_AND_PROC_NEST-ING_DEPTH_EXCEEDED	nesting depth of trigger and procedure is exceeded
649	ERR_SQL_QUERY_PINNED_PLAN	Pinned plan error
650	ERR_SQL_QUERY_REMOVE_PIN-NED_PLAN	Remove pinned plan error
651	ERR_SQL_EXST_OBJECT	cannot use duplicate object name
652	ERR_SQL_AMBG_SCHEMA	schema ambiguously defined
653	ERR_SQL_SET_ROW_ORDER	row order already set on table
654	ERR_SQL_NO_ROW_ORDER	no row order on table set
655	ERR_SQL_LICENSING_RUNTIME	licensing error
656	ERR_SQL_LONG_PROPERTY	property value too long
657	ERR_SQL_CANCEL_TASK_TIME-OUT_REACHED	request to cancel task was sent but task did not cancel before timeout was reached
658	ERR_SQL_CANNOT_MUTATE_TA-BLE_DURING_FK_EXECUTION	cannot mutate the table during trigger or foreign key execution
659	ERR_SQL_EXST_WORKLOAD_CLASS	cannot use duplicate workload class name
660	ERR_SQL_INV_WORKLOAD_CLASS	invalid workload class name
661	ERR_SQL_EXST_WORKLOAD_MAP-PING	cannot use duplicate workload mapping name
662	ERR_SQL_INV_WORKLOAD_MAPPING	invalid workload mapping name
663	ERR_SQL_CONNECT_NOT_ALLOWED	user not allowed to connect from client
664	ERR_SQL_INV_AGENT_GROUP	invalid agent group name

Code	Type	Description
665	ERR_SQL_EXST_AGENT_GROUP	cannot use duplicate agent group name
666	ERR_SQL_AGENT_GROUP_NOT_EMPTY	agents are still set to this agent group.
667	ERR_SQL_TEXT_MINING_FAILURE	text mining error
668	ERR_SQL_2D_POINTS_SUPPORTED_ONLY	ST_Point columns support 2-dimensional points only
669	ERR_SQL_SPATIAL_ERROR	spatial error
670	ERR_SQL_PART_NOT_EXIST	part does not exist
671	ERR_SQL_EXST_LIBRARY	cannot use duplicate library name
672	ERR_SQL_DPLC_ASSOCIATION	duplicate association name
673	ERR_SQL_INV_GRAPH_WORKSPACE	invalid graph workspace name
675	ERR_SQL_EXST_GRAPH_WORKSPACE	cannot use duplicate graph workspace name
676	ERR_SQL_DUP_WORKLOAD_MAPPING	cannot use duplicate workload mapping to same combination of (user name, application user name, application name, client, application component name, application component type)
677	ERR_SQL_CHECK_CONSTRAINT_VIOLATION	check constraint violation
678	ERR_SQL_PLANSTABILIZER	plan stabilizer error
679	ERR_SQL_PLANSTABILIZER_NO_MANAGER	plan stabilizer error - manager not found: please check if Plan Stabilizer is enabled
680	ERR_SQL_PLANSTABILIZER_STORED_HINT	plan stabilizer stored hint error - statement hint table error
681	ERR_SQL_PLANSTABILIZER_STORED_HINT_COMMAND	plan stabilizer stored hint error - error while processing statement hint command
682	ERR_SQL_PLANSTABILIZER_STORED_HINT_TABLE_EMPTY	plan stabilizer stored hint error - statement hint table is empty
683	ERR_SQL_PLANSTABILIZER_STORED_HINT_MAP_LOAD_ERROR	plan stabilizer stored hint error - statement hint table is corrupt.
684	ERR_SQL_PLANSTABILIZER_STORED_HINT_RECORD_ALREADY_EXISTS	plan stabilizer stored hint error - statement hint record already exists
685	ERR_SQL_PLANSTABILIZER_STORED_HINT_RECORD_DOES_NOT_EXIST	plan stabilizer stored hint error - statement hint record does not exist
686	ERR_SQL_START_TASK_ERROR	start task error
687	ERR_SQL_EXCEED_LAG_TIME	exceed lag time of RESULT_LAG

Code	Type	Description
689	ERR_SQL_DUPLI-CATE_ROWID_MATCHED	Duplicate rowid matched during merge into
690	ERR_SQL_PLANSTABIL-IZER_STORED_PLAN	plan stabilizer stored plan error
691	ERR_SQL_PLANSTABIL-IZER_STORED_PLAN_COMMAND	plan stabilizer stored plan error - error while processing command
692	ERR_SQL_PLANSTABIL-IZER_STORED_PLAN_TABLE_EMPTY	plan stabilizer stored plan error - stored plan table is empty
693	ERR_SQL_PLANSTABIL-IZER_STORED_PLAN_MAP_LOAD_ER-ROR	plan stabilizer stored plan error - stored plan table is corrupt.
694	ERR_SQL_PLANSTABIL-IZER_STORED_PLAN_RECORD_AL-READY_EXISTS	plan stabilizer stored plan error - stored plan record already exists
695	ERR_SQL_PLANSTABIL-IZER_STORED_PLAN_RE-CORD_DOES_NOT_EXIST	plan stabilizer stored plan error - stored plan record does not exist
696	ERR_SQL_PLANSTABIL-IZER_STORED_PLAN_CANNOT_CON-VERT_ABSTRACT_PLAN	plan stabilizer stored plan error - can-not convert to abstract plan
697	ERR_SQL_PREACTIVE_KEY_EXISTS	Preactive key already exists
698	ERR_SQL_NO_PREACTIVE_KEY	No preactive key exists
699	ERR_SQL_EXST_DEPENDENCY_RULE	cannot use duplicate dependency rule name
700	ERR_SQL_SINGLE_COL-UMN_SEARCH_THROW_ERROR	no_stacked_column_search(throw_er-ror) error
701	ERR_SQL_EXST_USERGROUP	usergroup name already exists
702	ERR_SQL_INV_USERGROUP	invalid usergroup name
704	ERR_SQL_USERGROUP_DELE-TION_FAILED	usergroup cannot be dropped
705	ERR_SQL_CONCURRENT_GRANT	Two concurrent statements performed the same grant operation
706	ERR_SQL_INV_SYMMETRIC_CIPHER	currently only AES-256-CBC is sup-ported: invalid cipher
707	ERR_SQL_EXST_COLUMN_KEY	cannot use duplicate column key name
708	ERR_SQL_EXST_COLUMN_KEYCOPY	column keycopy already exists
709	ERR_SQL_EXST_KEYPAIR	keypair already exists
710	ERR_SQL_INV_ASYMMETRIC_CIPHER	currently only RSA-OAEP-2048 is sup-ported: invalid cipher
711	ERR_SQL_EXST_COLUMN_KEY_ID	cannot use duplicate column key id

Code	Type	Description
712	ERR_SQL_PLANSTABIL-IZER_STORED_PLAN_MIGRATION	plan stabilizer stored plan error - migra-tion error
713	ERR_SQL_NOT_OWN_KEYPAIR	keypair not owned by the creator of the column key
714	ERR_SQL_DROP_COLUMN_KEYCOPY	cannot drop the last key admin keycopy
715	ERR_SQL_EMPTY_WORKLOAD_MAP-PING	cannot use a workload mapping with no properties
716	ERR_SQL_STALE_STATEMENT	statement is stale, metadata or column encryption key of some columns have changed
717	ERR_SQL_INV_KEY_ID	invalid key id
1,280	ERR_SQLSCRIPT_2	sqlscript error
1,281	ERR_SQLSCRIPT_WRONG_PARAMS	wrong number or types of parameters in call
1,282	ERR_SQLSCRIPT_OUT_PARAM_VAR	output parameter not a variable
1,283	ERR_SQLSCRIPT_OUT_PARAM_DE-FAULT	OUT and IN OUT parameters may not have default expressions
1,284	ERR_SQLSCRIPT_DUP_PARAMETERS	duplicate parameters are not permitted
1,285	ERR_SQLSCRIPT_DUP_DECL	at most one declaration is permitted in the declaration section
1,286	ERR_SQLSCRIPT_CURSOR_SE-LECT_STMT	cursor must be declared by SELECT statement
1,287	ERR_SQLSCRIPT_ID_NOT_DECLARED	identifier must be declared
1,288	ERR_SQLSCRIPT_NOT_ASSIGN_TAR-GET	expression cannot be used as an as-signment target
1,289	ERR_SQLSCRIPT_NOT_INTO_TARGET	expression cannot be used as an INTO-target of SELECT/FETCH statement
1,290	ERR_SQLSCRIPT_LHS_CANNOT_AS-SIGNED	expression is inappropriate as the left hand side of an assignment statement
1,291	ERR_SQLSCRIPT_EXPR_WRONG_TYP E	expression is of wrong type
1,292	ERR_SQLSCRIPT_ILLE-GAL_EXIT_STMT	illegal EXIT statement, it must be ap-pear inside a loop
1,293	ERR_SQLSCRIPT_ID_EXCEP-TION_TYPE	identifier name must be an exception name
1,294	ERR_SQLSCRIPT_INTO_CLAUSE	an INTO clause is expected in SELECT statement
1,295	ERR_SQLSCRIPT_NOT_AL-LOWED_SQL_STMT	EXPLAIN PLAN and CALL statement are not allowed
1,296	ERR_SQLSCRIPT_NOT_CURSOR	identifier is not a cursor

Code	Type	Description
1,297	ERR_SQLSCRIPT_NUM_FETCH_VAL-UES	wrong number of values in the INTO list of a FETCH statement
1,298	ERR_SQLSCRIPT_UNHANDLED_EX-CEPTION	unhandled user-defined exception
1,299	ERR_SQLSCRIPT_NO_DATA_FOUND	no data found
1,300	ERR_SQLSCRIPT_FETCH_MANY_ROW S	fetch returns more than requested number of rows
1,301	ERR_SQLSCRIPT_VALUE_ERROR	numeric or value error
1,302	ERR_SQLSCRIPT_OUT_PARAM_IN_FU NCTION	parallelizable function cannot have OUT or IN OUT parameter
1,303	ERR_SQLSCRIPT_USER_DEFINED_EX-CEPTION	user-defined exception
1,304	ERR_SQLSCRIPT_CURSOR_AL-READY_OPEN	cursor is already opened
1,305	ERR_SQLSCRIPT_INVALID_RE-TURN_TYPE	return type is invalid
1,306	ERR_SQLSCRIPT_RETURN_TYPE_MIS-MATCH	return type mismatch
1,307	ERR_SQLSCRIPT_UNSUPPORTED_DA-TATYPE	unsupported datatype is used
1,308	ERR_SQLSCRIPT_INVALID_SIN-GLE_ASSIGNMENT	illegal single assignment
1,309	ERR_SQLSCRIPT_INVA-LID_USE_OF_TABLE_VARIABLE	invalid use of table variable
1,310	ERR_SQLSCRIPT_NOT_AL-LOWED_SCALAR_TYPE	scalar type is not allowed
1,311	ERR_SQLSCRIPT_NO_OUT_PARAM	Out parameter is not specified
1,312	ERR_SQLSCRIPT_AT_MOST_ONE_OUT _PARAM	At most one output parameter is al-lowed
1,313	ERR_SQLSCRIPT_OUT_PARAM_TABLE	output parameter should be a table or a table variable
1,314	ERR_SQLSCRIPT_INVALID_VARIA-BLE_NAME	inappropriate variable name: do not al-low \"\" or '_SYS_' prefix for the name of variable or parameter
1,315	ERR_SQLSCRIPT_RETURN_RE-SULT_SET_WITH_RESULTVIEW	Return result set from SELECT state-ment exist when result view is defined
1,316	ERR_SQLSCRIPT_NOT_AS-SIGNED_OUT_TABVAR	some out table variable is not assigned
1,317	ERR_SQLSCRIPT_FUNC-TION_NAME_MAX_LEN	Function name exceedes max. limit
1,318	ERR_SQLSCRIPT_BUILTIN_NOT_DE-FINED	Built-in function not defined

Code	Type	Description
1,319	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_TABLE_NAME	Parameter must be a table name
1,320	ERR_SQLSCRIPT_BUILTIN_PARAM_AT-TRIBUTE_WITH_SCHEMA	Parameter must be an attribute name without a table name upfront
1,321	ERR_SQLSCRIPT_BUILTIN_PARAM_AT-TRIBUTE_WITH_ALIAS	Parameter must be an attribute name without an alias
1,322	ERR_SQLSCRIPT_CALC_ATTR_NOT_A LLOWED	CE_CALC not allowed
1,323	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_COL_OR_AGGR_VE CTOR	Parameter must be a vector of columns or aggregations
1,324	ERR_SQLSCRIPT_BUILTIN_MISS-ING_JOIN_ATTR_IN_PROJECTION	Join attribute must be available in pro-jection list
1,325	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_SQLIDENT_VECTOR	Parameter must be a vector of sql iden-tifiers
1,326	ERR_SQLSCRIPT_DUPLICATE_ATTRIB-UTE_NAME	Duplicate attribute name
1,327	ERR_SQLSCRIPT_PARAM_UNSUP-PORTED_TYPE	Parameter has a non supported type
1,328	ERR_SQLSCRIPT_BUILTIN_MISS-ING_ATTRIBUTE_IN_PROJECTION	Attribute not found in column table
1,329	ERR_SQLSCRIPT_BUILTIN_DUPLI-CATE_COLUMN_NAME	Duplicate column name
1,330	ERR_SQLSCRIPT_BUILTIN_CAL-CATTR_EXPRESSION_SYNTAX	Syntax Error for calculated Attribute
1,331	ERR_SQLSCRIPT_BUILTIN_FILTER_EX-PRESSION_SYNTAX	Syntax Error in filter expression
1,332	ERR_SQLSCRIPT_BUIL-TIN_FIRST_PARAM_NOT_COL-UMN_TABLE	Parameter must be a valid column table or projection view on column tables
1,333	ERR_SQLSCRIPT_BUILTIN_JOIN-ATTR_NOT_FOUND_IN_VAR	Join attributes not found in variable
1,334	ERR_SQLSCRIPT_BUIL-TIN_IN_PARAM_NOT_SAME_TA-BLE_TYPE	Input parameters do not have the same table type
1,335	ERR_SQLSCRIPT_RUNTIME_CY-CLIC_DEPENDENCY	Cyclic dependency found in a runtime procedure
1,336	ERR_SQLSCRIPT_RUNTIME_UNEX-PECTED_EXCEPTION	Unexpected internal exception caught in a runtime procedure
1,337	ERR_SQLSCRIPT_VAR_DE-PENDS_ON_UNASSIGNED_VAR	Variable depends on an unassigned var-iable
1,338	ERR_SQLSCRIPT_CE_CONVER-SION_CUSTOM_TAB_MISSING	CE_CONVERSION: customizing table missing

Code	Type	Description
1,339	ERR_SQLSCRIPT_TOO_MANY_PAR-AMS	Too many parameters
1,340	ERR_SQLSCRIPT_NESTED_CALL_TOO_DEEP	The depth of the nested call is too deep
1,341	ERR_SQLSCRIPT_VERSION_VALIDA-TION_FAILED	Procedure version validation failed
1,342	ERR_SQLSCRIPT_CE_CALC_ATTRIB-UTE_AND_ALIAS_ARE_SAME	Attribute has the same name as the alias
1,343	ERR_SQLSCRIPT_RETRY_EXCEPTION	Retry Exception is occurred in a run-time procedure
1,344	ERR_SQLSCRIPT_NOT_ALLOWED_DY-NAMIC_SQL	Dynamic SQL or DDL is not allowed
1,345	ERR_SQLSCRIPT_NOT_AL-LOWED_CONCURRENT_WRITES	Concurrently two or more write opera-tions to the same object are not allowed
1,346	ERR_SQLSCRIPT_NOT_AL-LOWED_CONCUR-RENT_READ_AND_WRITE	Concurrently read and write operations to the same object are not allowed
1,348	ERR_SQLSCRIPT_LLANG_GET_LI-BRARY_IMPORT_LIST_FAILED	Failed to retrieve the list of imported li-braries from LLANG procedure
1,349	ERR_SQLSCRIPT_INITIAL_ASSIGN-MENT_REQUIRED_FOR_CON-STANT_TABLE	Assigning initial value is required for de-claring constant table variable
1,350	ERR_SQLSCRIPT_NOT_AL-LOWED_NON_DETERMINISTIC_FEA-TURE	Non-deterministic feature is not al-lowed
1,351	ERR_SQLSCRIPT_INVA-LID_PARSE_TREE	Invalid parse tree
1,352	ERR_SQLSCRIPT_ENCRYP-TION_NOT_ALLOWED	Not allowed for encrypted procedure or function
1,353	ERR_SQLSCRIPT_NOT_NULL_COL-UMN_IGNORED	NOT NULL constraints in explicit table types are ignored
1,354	ERR_SQLSCRIPT_CUR-SOR_NOT_OPENED	Cursor to be fetched has not been opened yet
1,355	ERR_SQLSCRIPT_INVALID_EX-TERN_LANG	Invalid external language
2,816	ERR_SQLSCRIPT	SqlScript Error
2,817	ERR_SQLSCRIPT_BUIL-TIN_TOO_MANY_RETURN_PARAM	SqlScript Builtin Function
2,818	ERR_SQLSCRIPT_FUNC-TION_NOT_FOUND	SqlScript
2,819	ERR_SQLSCRIPT_TEMPLATE_PARAM-ETER_NUMBER_WRONG	SqlScript

Code	Type	Description
2,820	ERR_SQLSCRIPT_VARIABLE_NOT_DE-CLARED	SqlScript
2,821	ERR_SQLSCRIPT_DUPLICATE_VARIA-BLE_NAME	SqlScript
2,822	ERR_SQLSCRIPT_SQL_EXECU-TION_FAILED	SqlScript
2,823	ERR_SQLSCRIPT_DROP_FUNC-TION_FAILED	SqlScript
2,824	ERR_SQLSCRIPT_LOAD_FUNC-TION_FAILED	SqlScript
2,825	ERR_SQLSCRIPT_SIGNATURE_MIS-MATCH_WITH_CATALOG	SqlScript
2,826	ERR_SQLSCRIPT_REGISTER_FUNC-TION_IN_CATALOG_FAILED	SqlScript
2,827	ERR_SQLSCRIPT_SCALAR_IN-PUT_PARAMS_NOT_SUPPORTED	SqlScript
2,828	ERR_SQLSCRIPT_LAN-GUAGE_NOT_SUPPORTED	SqlScript
2,829	ERR_SQLSCRIPT_DROP_FUNC-TION_FAILED_EXISTING_CALLER	SqlScript
2,830	ERR_SQLSCRIPT_LLANG_EX-ACTLY_ONE_OUTPUT_PARAM	SqlScript
2,831	ERR_SQLSCRIPT_BUIL-TIN_FIRST_PARAM_NOT_COL-UMN_TABLE	SqlScript
2,832	ERR_SQLSCRIPT_BUIL-TIN_PARAM_COUNT_NOT_IN_RANGE	SqlScript
2,833	ERR_SQLSCRIPT_BUIL-TIN_PARAM_COUNT_MISMATCH	SqlScript
2,834	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_INPUT	SqlScript
2,835	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_TABLE_NAME	SqlScript
2,836	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_VARIABLE	SqlScript
2,837	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_VARIABLE_VECTOR	SqlScript
2,838	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_SCALAR_VALUE	SqlScript
2,839	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_SQLIDENT_VECTOR	SqlScript
2,840	ERR_SQLSCRIPT_BUILTIN_PARAM_AT-TRIBUTE_WITH_SCHEMA	SqlScript

Code	Type	Description
2,841	ERR_SQLSCRIPT_BUILTIN_MISS-ING_ATTRIBUTE_IN_PROJECTION	SqlScript
2,842	ERR_SQLSCRIPT_BUILTIN_MISS-ING_JOIN_ATTR_IN_PROJECTION	SqlScript
2,843	ERR_SQLSCRIPT_TEMPL_FUNC-TION_CAN_NOT_BE_CALLED	SqlScript
2,844	ERR_SQLSCRIPT_PARAM_COUNT_MIS MATCH	SqlScript
2,845	ERR_SQLSCRIPT_PARAM_WRONG_TY PE	SqlScript
2,846	ERR_SQLSCRIPT_PARAM_WRONG_TY PE_COMPARED_TO_SIGNATURE	SqlScript
2,847	ERR_SQLSCRIPT_PARAM_WRONG_TA BLE_TYPE	SqlScript
2,848	ERR_SQLSCRIPT_PARAM_MODE_MIS-MATCH	SqlScript
2,849	ERR_SQLSCRIPT_PARAM_UNSUP-PORTED_TYPE	SqlScript
2,850	ERR_SQLSCRIPT_NO_OUT-PUT_PARAM	SqlScript
2,851	ERR_SQLSCRIPT_OUT-PUT_PARAM_NOT_TABLE_TYPE	SqlScript
2,852	ERR_SQLSCRIPT_BUILTIN_NOT_DE-FINED	SqlScript
2,853	ERR_SQLSCRIPT_VAR_DE-PENDS_ON_UNASSIGNED_VAR	SqlScript
2,854	ERR_SQLSCRIPT_VAR_CYCLIC_DE-PENDENCY	SqlScript
2,855	ERR_SQLSCRIPT_PARAM_NOT_INI-TIALIZED	SqlScript
2,856	ERR_SQLSCRIPT_PARAM_MIS-MATCH_TABLE_TYPE	SqlScript
2,857	ERR_SQLSCRIPT_CALL_OPEN_MISS-ING_CALL_CLOSE	SqlScript
2,858	ERR_SQLSCRIPT_BUIL-TIN_IN_PARAM_NOT_SAME_TA-BLE_TYPE	SqlScript
2,859	ERR_SQLSCRIPT_BUILTIN_JOIN-ATTR_NOT_FOUND_IN_VAR	SqlScript
2,860	ERR_SQLSCRIPT_FUNC-TION_NOT_NESTABLE	SqlScript
2,861	ERR_SQLSCRIPT_CALL_CLOSE_MISS-ING_CALL_OPEN	SqlScript

Code	Type	Description
2,862	ERR_SQLSCRIPT_TA-BLE_TYPE_NOT_DERIVABLE	SqlScript
2,863	ERR_SQLSCRIPT_MISS-ING_FTC_TYPE_MAPPING	SqlScript
2,864	ERR_SQLSCRIPT_INVALID_TA-BLE_TYPE_NAME	SqlScript
2,865	ERR_SQLSCRIPT_DUPLICATE_ATTRIB-UTE_NAME	SqlScript
2,866	ERR_SQLSCRIPT_FUNCTION_EXIST-ING	SqlScript
2,867	ERR_SQLSCRIPT_FUNC-TION_TYPE_NOT_SUPPORTED	SqlScript
2,868	ERR_SQLSCRIPT_FUNC-TION_NAME_MAX_LEN	SqlScript
2,869	ERR_SQLSCRIPT_BUILTIN_PARAM_AT-TRIBUTE_WITH_ALIAS	SqlScript
2,870	ERR_SQLSCRIPT_INTERNAL_ERR	SqlScript
2,871	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_AGGREGFUN_VEC-TOR	SqlScript
2,872	ERR_SQLSCRIPT_FUNC-TION_NAME_INVALID	SqlScript
2,873	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_PROJECTION_VEC-TOR	SqlScript
2,874	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_FILTER_EXPRES-SION	SqlScript
2,875	ERR_SQLSCRIPT_RLANG_EX-ACTLY_ONE_OUTPUT_PARAM	SqlScript
2,876	ERR_SQLSCRIPT_JSLANG_EX-ACTLY_ONE_OUTPUT_PARAM	SqlScript
2,877	ERR_SQLSCRIPT_SQLLANG_EX-ACTLY_ONE_OUTPUT_PARAM	SqlScript
2,878	ERR_SQLSCRIPT_GENERICLANG_EX-ACTLY_ONE_OUTPUT_PARAM	SqlScript
2,879	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_TABLE_TYPE	SqlScript
2,880	ERR_SQLSCRIPT_VARIABLE_NOT_TA-BLE_TYPE	SqlScript
2,881	ERR_SQLSCRIPT_BUILTIN_CAL-CATTR_EXPRESSION_SYNTAX	SqlScript

Code	Type	Description
2,882	ERR_SQLSCRIPT_BUILTIN_UN-EVEN_NR_OF_PARAMS	SqlScript
2,883	ERR_SQLSCRIPT_CALC_ATTR_NOT_A LLOWED	SqlScript
2,884	ERR_SQLSCRIPT_BUILTIN_DUPLI-CATE_COLUMN_NAME	SqlScript
2,885	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_KEY_VALUE_VEC-TOR	SqlScript
2,886	ERR_SQLSCRIPT_BUILTIN_CAL-CATTR_REFERENCED_FIELD_MISSING	SqlScript
2,887	ERR_SQLSCRIPT_BUILTIN_FIL-TER_REFERENCED_FIELD_MISSING	SqlScript
2,888	ERR_SQLSCRIPT_BUILTIN_FILTER_EX-PRESSION_SYNTAX	SqlScript
2,889	ERR_SQLSCRIPT_BUIL-TIN_PARAM_NOT_COL_OR_AGGR_VE CTOR	SqlScript
2,890	ERR_SQLSCRIPT_TABLE_INPUT_PAR-AMS_NOT_SUPPORTED	SqlScript
2,891	ERR_SQLSCRIPT_TABLE_INOUT_PAR-AMS_NOT_SUPPORTED	SqlScript
601	ERR_API_TOO_MANY_SESSION_VARI-ABLES	too many session variables are set
612	ERR_API_SESSION_VARIA-BLE_KEY_LENGTH_EXCEEDED	maximum length of key for session vari-able exceeded
146	ERR_TX_LOCK_ACQUISITION_FAIL	Resource busy and NOWAIT specified

8.12 Array Variables

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.

8.12.1 Declare a Variable of Type ARRAY

You declare a variable of type ARRAY 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 | VARBINARY | CLOB |
NCLOB  |BLOB
```

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

```
DECLARE arr INTEGER ARRAY;
```

Only unbounded arrays with a maximum cardinality of 2147483646, that is in the range between 1 and 2^31 -2 ([1-2147483646]), are supported. 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.

8.12.2 Set an Element of an Array

The syntax for setting a value to 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, where `<array_index>` can have any value from 1 to 2^31 -2 ([1-2147483646]). For example, the following statement stores the value 10 in the second element of the array **id**:

```
id[2] = 10;
```

Please note that all elements of the array that are not set, have the value NULL. In the given example id[1] is 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, the 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 can also be used as an index. For example, a scalar UDF that adds two values and returns the result

```
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;
```

i Note

The array starts with the index 1.

8.12.3 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 -2 ([1-2147483646]). 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;
```

8.12.4 ARRAY_AGG Function

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.

8.12.5 TRIM_ARRAY Function

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 the following:

```
ID
---
1
2
```

8.12.6 CARDINALITY Function

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;
```

8.12.7 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;
```

8.12.8 Array Parameters for Procedures and Functions

You can create procedures and functions with array parameters so that array variables or constant arrays can be passed to them.

The flowing scenarios are supported:

- Array input/output/inout parameter for procedures
- Array input parameter for SUDF/TUDF
- Array return type for SUDF
- Array parameter for library procedures/functions

- Array input parameter for anonymous block/embedded SQL function
- Array variables in DML/queries.

> **! Restriction**
>
> This feature supports array parameters only for server-side query parameters. It is not possible to use client-side array interfaces. Array parameters cannot be used in the outermost queries or calls. It is allowed to use array parameters only in nested queries or nested calls.

Syntax

Code Syntax

```
CREATE [OR REPLACE] PROCEDURE <proc_name> [(<parameter_clause>)] [LANGUAGE
<lang>] [SQL SECURITY <mode>] [DEFAULT SCHEMA <default_schema_name>]
[READS SQL DATA ] [WITH ENCRYPTION] AS
BEGIN [SEQUENTIAL EXECUTION]
   <procedure_body>
END
<parameter_clause> ::= <parameter> [{,<parameter>}...]
<parameter> ::= [IN | OUT | INOUT] <param_name> <param_type>
<param_type> ::= <sql_type> [ARRAY] | <table_type> | <table_type_definition>
```

Code Syntax

```
CREATE FUNCTION <func_name> [(<parameter_clause>)] RETURNS <return_type>
[LANGUAGE <lang>] [SQL SECURITY <mode>][DEFAULT SCHEMA <default_schema_name>
[DETERMINISTIC]] [WITH ENCRYPTION]
AS BEGIN
<function_body>
END
<parameter_clause> ::= <parameter> [{,<parameter>}...]
<parameter> ::= [IN] <param_name> <param_type>
<param_type> ::= <sql_type> [ARRAY] | <table_type> | <table_type_definition>
<return_type> ::= <return_parameter_list>
<return_parameter_list> ::= <return_parameter>[{, <return_parameter>}...]
<return_parameter>      ::= <parameter_name> <sql_type> [ARRAY]
```

Examples

Sample Code

```
create procedure my_l_proc_out(out c int array, in b int array) as
begin
  c = array(123456, 7890);
  c[3] = :b[1];
  c[4] = :b[2];
end;

do begin
  declare a int array;
```

```
  declare b int array = array(3, 4);
  call my_l_proc_out(:a, :b);
  select :a from dummy;
END;
```

Sample Code

```
create function my_sudf_arr (in a int array) returns b int array as
begin
  b = subarray(:a, 1, 2);
end;

do begin
  declare arr_var int array = array(1, 2, 3, 4);
  select my_sudf_arr(:arr_var) x from dummy;
end;
```

Sample Code

```
create function my_tudf_arr (in A int array) returns table(I int) as
begin
  B = unnest(:A);
  return select ":A" as I from :B;
end;

do begin
  declare arr_var int array = array(1, 2, 3, 4);
  select * from my_tudf_arr(:arr_var);
end;
```

i Note

For improving SQLScript usability, not only constant arrays but also array variables can be used in DML and queries. In addition, it is also possible to use array variables in the SELECT INTO clause.

Sample Code

```
create table tab1 (i int, a int array);

do begin
  declare a int array = array(1, 2, 3);
  declare b int array;
  insert into tab1 values (1, :a);
  select tab1.A into b from tab1;
  select array(1,2,3) into b from dummy;
  insert into tab1 values (1, array(1, 2, 3));
  select :a from dummy;
end;
```

i Note

The system view ELEMENT_TYPES now shows the element data type of the parameter, if it is an array type. The ELEMENT_TYPES view has the columns SCHEMA_NAME, OBJECT_NAME, ELEMENT_NAME, and DATA_TYPE_NAME.

Limitations

The following limitations apply:

- LOB type array parameter is not supported.
- DEFAULT VALUE for an array parameter is not supported.
- Using an array parameter in the USING clause of Dynamic SQL is not supported.

8.13 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
```

8.14 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;
```

i 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 performing independent INSERT operations 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;
```

i Note

Only the following statements are allowed in read-write procedures, which can be called within a parallel block:

- DML
- Imperative logic
- Autonomous transaction
- Implicit SELECT and SELECT INTO scalar variable

Restrictions and Limitations

The following restrictions apply:

- Updating the same table in different statements is not allowed
- Only concurrent reads on one table are allowed. Implicit SELECT and SELCT INTO scalar variable statements are supported.
- 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.

8.15 Recursive SQLScript Logic

Description

Before the introduction of SQLScript recursive logic, it was necessary to rewrite any recursive operation into an operation using iterative logic, if it was supposed to be used within an SQLScript procedure or a function. SQLScript now supports recursive logic that allows you to write a procedure or a function that calls itself within its body until the abort condition is met.

Example

Sample Code

```
create procedure factorial_proc(in i int, out j int) as begin
  if :i <= 1 then
    j = 1;
  else
    call factorial_proc(:i-1, j);
    j = :i * :j;
  end if;
end;

call factorial_proc(0, ?);
call factorial_proc(1, ?);
call factorial_proc(4, ?);
call factorial_proc(10, ?);

create function factorial_func(i int) returns j int as begin
  if :i <= 1 then
    j = 1;
  else
    j = :i * factorial_func(:i-1);
  end if;
end;

select factorial_func(0) from dummy;
select factorial_func(1) from dummy;
select factorial_func(4) from dummy;
select factorial_func(10) from dummy;
```

```
create function factorial_func2(i int) returns table(a int) as begin
  if :i <= 1 then
    return select 1 as a from dummy;
  else
    return select :i * a as a from factorial_func2(:i - 1);
  end if;
end;

select * from factorial_func2(0);
select * from factorial_func2(1);
select * from factorial_func2(4);
select * from factorial_func2(10);
```

Limitations

The following limitations apply:

- By default, the maximum depth of a procedure call is 32.
- User-defined functions do not have an explicit call-depth check, but the system will return a run-time error when no further evaluation is available.
- SQLScript Library member procedures and functions do not support recursion.

9 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.

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_vari able>,<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
CE_VERTICAL_UNION	CE_VERTICAL_UNION(<table_v ariable>, <projection_list>[{ ,<tab le_variable >, <projection_list >}...])	unnest (<table_variable> [{, <table_variable> } …]) as ((<projection_list>) [{, (<projection_list>) } …])

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.

9.1 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.

9.1.1 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.

> **i 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.

9.1.2 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.

9.1.3 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.

> **i 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.

9.1.4 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.

9.2 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.

9.2.1 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.

> **i 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.

9.2.2 CE_LEFT_OUTER_JOIN

Calculate the left outer join. Besides the function name, the syntax is the same as for CE_JOIN.

9.2.3 CE_RIGHT_OUTER_JOIN

Calculate the right outer join. Besides the function name, the syntax is the same as for `CE_JOIN`.

ℹ Note

CE_FULL_OUTER_JOIN is not supported.

9.2.4 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 224] 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.

9.2.5 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 rule is that unquoted identifiers are converted to 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 a 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:

Expression Functions

Name	Description	Syntax
Conversion Functions	Conversion between data types	

Name	Description	Syntax
float	Converts `arg` to a float data type.	float float(arg)
double	Converts `arg` to a double data type.	double double(arg)
decfloat	Converts `arg` to a decfloat data type.	decfloat decfloat(arg)
fixed	Converts `arg` to a fixed data type.	fixed fixed(arg, int, int)
string	Converts `arg` to a string data type.	string string(arg)
date	Converts `arg` to the `daydate` data type.[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 was 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)

Name	Description	Syntax
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)
ltrim	Removes a white space prefix from a string. The white space 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 white space characters).	• string ltrim(string) • string ltrim(string, string)
rtrim	Removes trailing white spaces from a string. The white space 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 white space characters).	• string rtrim(string) • string rtrim(string, string)
trim	Removes white space characters from the beginning and the end of a string. The following statements are allowed: • trim(s) = ltrim(rtrim(s)) • trim(s1, s2) = ltrim(rtrim(s1, s2), s2)	• string trim(string) • string trim(string, string)
lpad	Adds a white space character to the left of a string. A second string argument specifies the white space 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 a white space at the end of a string. A second string argument specifies the white space 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 mathematical functions described here generally operate on floating-point values; their inputs automatically convert to double, the output will also be a double.	

Name	Description	Syntax
	These functions have the same functionality as in the C programming language.	• 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)
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	Returns `arg`, if `arg` is positive or zero, `-arg` in any other case. `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 on time data	
utctolocal	Interprets `datearg` (a date, without timezone) as utc and converts it to the time zone named by `timezonearg` (a string).	iutctolocal(datearg, timezonearg)
localtoutc	Converts the local datetime `datearg` to the time zone specified by the string `timezonearg`, returns as a date.	localtoutc(datearg, timezonearg)
weekday	Returns the week day as an integer in the range 0..6. 0 is Monday.	weekday(date)
now	Returns the current date and time (local time of the server timezone) as a 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		

Name	Description	Syntax
if	Returns `arg2`, if `intarg` is considered true (not equal to zero), else returns `arg3`. Currently, no shortcut evaluation is implemented, which means that both `arg2` and `arg3` are evaluated in any case. This means that you cannot use `if` to avoid a division-by-zero error, which has the side effect of terminating expression evaluation when it occurs.	if(intarg, arg2, arg3)
case	Returns value1, if arg1 == cmp1, value2 if arg1 == cmp2 and so on. Returns the default, if there is no match.	• case(arg1, default) • case(arg1, cmp1, value1, cmp2, value2, ..., default)
isnull	Returns 1 (= true), if arg1 is set to null and null checking is on during the evaluator run.	isnull(arg1)
rownum	Returns the number of the row in the currently scanned table structure. The first row has the number 0.	rownum()

[1] Due to calendar variations with dates earlier than 1582, the use of the `date` data type is deprecated and you should use the `daydate` data type instead.

> **i 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, see the section *Using String Functions With Multi-Byte Character Encoding*. Note that this limitation does not exist for SQL functions of the SAP HANA database, which natively support unicode strings.

9.2.5.1 Using String Functions with Multi-Byte Character Encoding

To allow the use of the string functions of the Calculation Engine with multi-byte character encoding, you can use the `charpos` and `chars` functions. An example of this usage for the single-byte character function `midstr` follows below:

```
midstr(<input_string>, charpos(<input_string>, 32), 1)
```

Related Information

CE_CALC [page 224]

9.2.6 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.

> **i 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.

> i 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.

9.2.7 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.

9.3 Special Operators

In this section we discuss operators that have no immediate counterpart in SQL.

9.3.1 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.

.

9.3.2 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:

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.

Key	Values	Type	Mandatory	Default	Documentation
‘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', 'unconver-ted', 'converted', '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.
'reference_date'	Any	Constant	N	None	The default refer-ence date for any kind of conversion.
'source_unit_col-umn'	Column in input table	Column name	N	None	The name of the column containing the source unit in the input table.
'target_unit_col-umn'	Column in input table	Column name	N	None	The name of the column containing the target unit in the input table.
'refer-ence_date_col-umn'	Column in input table	Column name	N	None	The default refer-ence date for any kind of conversion.
'output_unit_col-umn'	Any	Column name	N	"OUTPUT_UNIT"	The name of the column containing the target unit in the output table.

For ERP conversion:

Key	Values	Type	Mandatory	Default	
'client'	Any	Constant		None	The client as stored in the ta-bles.
'conversion_type'	Any	Constant		'M'	The conversion type as stored in the tables.

Key	Values	Type	Mandatory	Default	
'schema'	Any	Schema name		Current schema	The default schema in which the conversion ta-bles should be looked up.

9.3.3 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.

Example:

You trace the content of variable `input` to a local temporary table.

```
out = TRACE(:input);
```

> ⚠ Caution
>
> This operator should not be used in production code as it will cause significant run-time overhead. Additionally, the naming conventions used to store the tracing information may change. This operator should only be used during development for debugging purposes.

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 needed for some SQL spatial functions.

The following example illustrates a small scenario for 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`.

The ':' in front of the variable `<first>` is 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     => ?);
```

returns 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 do not specify the SRID, you receive an error as demonstrated with the function `ST_GEOMFROMTEXT` in the following example. Here SRID 0 is used to specify 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 return an error because it is not allowed for the values in one column to have different SRID.

In addition, there is a consistency check for output table variables to ensure that all elements of a spatial column have the same SRID.

> **Note**
>
> The following functions are 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)`.

For more information on SQL spatial functions and their usage, see SAP HANA Spatial Reference available on the SAP HANA Platform.

11 System Variables

System variables are built-in variables in SQLScript that provide you with 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 either the number of updated rows of the previously executed DML, CALL and CREATE TABLE statement, or the number of rows returned from a SELECT statement. There is no accumulation of ::ROWCOUNT values from all previously executed statements. When the previous statement does not return a value, the previous value of ::ROWCOUNT is retained. When ::ROWCOUNT is used right after a PARALLEL EXECUTION block, the system variable stores only the value of the last statement in the procedure definition.

> **⚠ Caution**
>
> Until SAP HANA 2.0 SPS03, the system variable ::ROWCOUNT was updated only after DML statements. Starting with SAP HANA 2.0 SPS04, the behavior of ::ROWCOUNT changes, it is now also updated for SELECT, CALL and CREATE TABLE statements.

The following limitations apply:

- ::ROWCOUNT for a nested CALL statement is an aggregation of the number of updated rows and does not include the number of rows returned from SELECT statements.

- ::ROWCOUNT for a SELECT statement is supported for normal SELECT statements, SELECT INTO statements and table variable assignments that contain a SELECT statement. It does not include SELECT sub-queries as a part of DML or DDL.
- ::ROWCOUNT for SELECT statements with multiple result sets is not supported.

> **i Note**
>
> When ::ROWCOUNT is used after a SELECT statement, it requires to fetch entire rows from the result set to get the total number of selected rows. When the result from the SELECT statement is assigned to a table variable or scalar variable it has barely any effect on the performance. However, a SELECT statement that is returning a result set cannot avoid fetching all rows implicitly regardless of how many rows will be explicitly fetched from the result set.

The following examples demonstrate 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
```

Incompatible Behavior Change

> ⚠ Caution
>
> The update of ::ROWCOUNT in SAP HANA 2.0 SPS04 introduces an incompatible behavior change. Please refer to the following description for the details, workaround and supporting tools.

Since ::ROWCOUNT is now updated after SELECT, CALL and CREATE TABLE statements, the behavior of existing procedures may change, if the system variable ::ROWCOUNT is not used directly after a DML statement. Using ::ROWCOUNT directly after the target statement is recommended and can guarantee the same behavior between different versions.

To detect such cases, new rules were introduced in SQLScript Code Analyzer:

- RULE_NAMESPACE: 'SAP', RULE_NAME: 'ROW_COUNT_AFTER_SELECT', CATEGORY: 'BEHAVIOR'
- RULE_NAMESPACE: 'SAP', RULE_NAME: 'ROW_COUNT_AFTER_DYNAMIC_SQL', CATEGORY: 'BEHAVIOR'

Based on the result from the SQLScript Code Analyzer rule, you can update your procedures according to the new standard behavior.

The following scenario shows a simple example of the impact of the behavior changes.

> Sample Code
>
> Behavior Change Example

```
create table mytab (i int);
insert into mytab values (1);

create table mytab2 (i int);
insert into mytab2 values (2);

do begin
  insert into mytab select * from mytab2; -- ::ROWCOUNT = 1
  x = select * from mytab;                -- ::ROWCOUNT = 1 (retained,
SPS03), ::RWCOUNT = 2 (SPS04)
  select ::rowcount from dummy;           -- 1 in SPS03, 2 in SPS04
end;
```

Statement	::ROWCOUNT (SPS03)	::ROWCOUNT (SPS04)
DML	The number of updated rows	The number of updated rows
SELECT statement `select * from mytab;`	N/A (retain previous value)	The number of rows returned from the SELECT statement
Table variable statement with SELECT statement `tv = select * from mytab;`	N/A (retain previous value)	The number of rows returned from the SELECT statement
SELECT INTO statement `select i into a from mytab;`	N/A (retain previous value)	1 if the statement is executed successfully, retains the previous value otherwise.
SELECT INTO statement with default value `select i into a default 2 from mytab;`	N/A (retain previous value)	0 if the default values are assigned, 1 if the values are assigned from the SELECT statement, retains the previous value otherwise.

SELECT statement in dynamic SQL
`exec 'select * from mytab';`
`execute immediate 'select * from mytab';`
	0	The number of rows from the SELECT statement

EXEC INTO with SELECT statement
`exec 'select i, j from mytab' into s1, s2;`
`exec 'select * from mytab' into tv;`
	0	EXEC INTO with scalar variables works similar to SELECT INTO case.
		EXEC INTO with a table variable works similar to a table variable assign statement case.

Nested CALL statement `call proc_nested;`	N/A (retain previous value)	The number of updated rows.
CREATE TABLE statement `create table tab_a as (select * from mytab);`	N/A (retains previous value)	The number of updated rows

11.3 ::CURRENT_LINE_NUMBER

SQLScript procedures, functions and triggers can return the line number of the current statement via ::CURRENT_LINE_NUMBER.

Syntax

::CURRENT_LINE_NUMBER

Example

Sample Code

```
create procedure proc_inner(out o int) as
begin
   o = ::CURRENT_LINE_NUMBER;
end;
```

Sample Code

```
create procedure proc_outer as
begin
    declare a int;
    call proc_inner(a);
    select :a, ::CURRENT_LINE_NUMBER from dummy;
end;
call proc_outer;
-- Returns [3, 5]
```

Sample Code

```
do begin
    declare a int = ::CURRENT_LINE_NUMBER;
    select :a, ::CURRENT_LINE_NUMBER + 1 from dummy;
end;
-- Returns [2, 3 + 1]
```

12 Built-In Libraries

This section provides information about built-in libraries in SQLScript.

12.1 Built-in Library SQLSCRIPT_SYNC

In some scenarios you may need to let certain processes wait for a while (for example, when executing repetitive tasks). Implementing such waiting manually may lead to "busy waiting" and to the CPU performing unnecessary work during the waiting time. To avoid this, SQLScript offers a built-in library SYS.SQLSCRIPT_SYNC containing the procedures SLEEP_SECONDS and WAKEUP_CONNECTION.

Procedure SLEEP_SECONDS

This procedure puts the current process on hold. It has one input parameter of type DOUBLE which specifies the waiting time in seconds. The maximum precision is one millisecond (0.001), but the real waiting time may be slightly longer (about 1-2 ms) than the given time.

> **i Note**
>
> - If you pass 0 or NULL to SLEEP_SECONDS, SQLScript executor will do nothing (also no log will be written).
> - If you pass a negative number, you get an error.

Procedure WAKEUP_CONNECTION

This procedure resumes a waiting process. It has one input parameter of type INTEGER which specifies the ID of a waiting connection. If this connection is waiting because the procedure SLEEP_SECONDS has been called, the sleep is terminated and the process continues. If the given connection does not exist or is not waiting because of SLEEP_SECONDS, an error is raised.

If the user calling WAKEUP_CONNECTION is not a session admin and is different from the user of the waiting connection, an error is raised as well.

> **i Note**
>
> - The waiting process is also terminated, if the session is canceled (with ALTER SYSTEM CANCEL SESSION or ALTER SYSTEM DISCONNECT SESSION).
> - A session admin can wake up any sleeping connection.

- The sleeping process is listed in the monitoring view M_SERVICE_THREADS. Its LOCK_WAIT_NAME starts with 'SQLScript/SQLScript_Sync/Sleep/'.

Limitations

The library cannot be used in functions (neither in scalar, nor in tabular ones) and in calculation views.

Examples

Sample Code

Monitor

```
CREATE PROCEDURE MONITOR AS
BEGIN
  USING SQLSCRIPT_SYNC AS SYNCLIB;
  WHILE 1 = 1 DO
    IF RECORD_COUNT(OBSERVED_TABLE) > 100000 THEN
      INSERT INTO LOG_TABLE VALUES (CURRENT_TIMESTAMP, 'Table size exceeds 
100000 records');
    END IF;
    CALL SYNCLIB:SLEEP_SECONDS(300);
  END WHILE;
END
```

Sample Code

Resume all sleeping processes

```
CREATE PROCEDURE RESUME_ALL AS
BEGIN
  USING SQLSCRIPT_SYNC AS SYNCLIB;
  DECLARE CURSOR WAITING_CONNECTIONS FOR SELECT CONNECTION_ID FROM 
M_SERVICE_THREADS
    WHERE LOCK_WAIT_NAME LIKE 'SQLScript/SQLScript_Sync/Sleep/%';
  FOR C AS WAITING_CONNECTIONS DO
    CALL SYNCLIB:WAKEUP_CONNECTION(C.CONNECTION_ID);
  END FOR;
END
```

12.2 Built-in Library SQLSCRIPT_STRING

The SQLSCRIPT_STRING library offers a handy and simple way for manipulating strings. You can split libraries with given delimiters or regular expressions, format or rearrange strings, and convert table variables into the already available strings.

Syntax

Code Syntax

```
CREATE LIBRARY SYS.SQLSCRIPT_STRING LANGUAGE SQLSCRIPT AS BUILTIN
BEGIN
    FUNCTION SPLIT(IN VALUE NVARCHAR(5000), IN SEPARATOR NVARCHAR(5000), IN
MAXSPLIT INT DEFAULT -1) RETURNS ...;
    FUNCTION SPLIT_TO_TABLE(IN VALUE NVARCHAR(5000), IN SEPARATOR
NVARCHAR(5000), IN MAXSPLIT INT DEFAULT -1) RETURNS TABLE(RESULT
NVARCHAR(5000));
    FUNCTION SPLIT_TO_ARRAY(IN VALUE NVARCHAR(5000), IN SEPARATOR
NVARCHAR(5000), IN MAXSPLIT INT DEFAULT -1) RETURNS RESULTS NVARCHAR(5000)
ARRAY;

    FUNCTION SPLIT_REGEXPR(IN VALUE NVARCHAR(5000), IN REGEXPR
NVARCHAR(5000), IN MAXSPLIT INT DEFAULT -1) RETURNS ...;
    FUNCTION SPLIT_REGEXPR_TO_TABLE(IN VALUE NVARCHAR(5000), IN REGEXPR
NVARCHAR(5000), IN MAXSPLIT INT DEFAULT -1) RETURNS TABLE(RESULT
NVARCHAR(5000));
    FUNCTION SPLIT_REGEXPR_TO_ARRAY(IN VALUE NVARCHAR(5000), IN REGEXPR
NVARCHAR(5000), IN MAXSPLIT INT DEFAULT -1) RETURNS RESULTS NVARCHAR(5000)
ARRAY;

    FUNCTION FORMAT(IN FORMAT NVARCHAR(5000), IN ...) RETURNS RESULT
NVARCHAR(8388607);
    FUNCTION FORMAT_TO_TABLE(IN FORMAT NVARCHAR(5000), IN TABLE(...)) RETURNS
TABLE(RESULT NVARCHAR(8388607));
    FUNCTION FORMAT_TO_ARRAY(IN FORMAT NVARCHAR(5000), IN TABLE(...)) RETURNS
RESULTS NVARCHAR(8388607) ARRAY;

    FUNCTION TABLE_SUMMARY(IN TABLE TABLE(...), IN ROWS INT DEFAULT 100)
RETURNS RESULT NVARCHAR(8388607);
 END;
```

SPLIT Family Functions

SPLIT / SPLIT_REGEXPR

The SPLIT(_REGEXPR) function returns multiple variables depending on the given parameters.

- If MAXSPLIT is -1, there is no limit on the number of splits.
- If MAXSPLIT is specified, at most MAXSPLIT splits are made.
- Empty string as input returns an empty string as result.
- String without separators as input returns the whole given string.

- String with N-1 separators as input returns N separated strings.

SPLIT_TO_ARRAY / SPLIT_ REGEXPR TO_ARRAY

The SPLIT_TO_ARRAY(REGEXPR) returns a NVARCHAR(5000) array with N separated strings

- Empty string as input returns an array of null values.
- String without separators as input returns an array with the whole given string in the first element.
- String with N-1 separator as input returns an array of N separated strings.

SPLIT_TO_TABLE / SPLIT_REGEXPR_TO_TABLE

The SPLIT_TO_TABLE(_REGEXPR) returns a single column table with table type (WORD NVARCHAR(5000))

- Empty string as input returns a single column table with 0 rows.
- String without separators as input returns a single column table with a whole given string in the first row
- String with N-1 separator as input returns a single column table with N separated strings in N rows.
- This function can be interpreted as UNNEST(SPLIT_TO_ARRAY(val, sep)) AS ("WORD").

Sample Code

```
DO BEGIN
    SQLSCRIPT_STRING AS LIB;
    DECLARE a1, a2, a3 INT;
    (a1, a2, a3) = LIB:SPLIT('10, 20, 30', ', '); --(10, 20, 30)
END;

DO BEGIN
    USING SQLSCRIPT_STRING AS LIB;
    DECLARE first_name, last_name STRING;
    DECLARE area_code, first_num, last_num INT;
    first_name = LIB:SPLIT('John Sutherland', ','); --('John Sutherland')
    (first_name, last_name) = LIB:SPLIT('John Sutherland', ' '); --
('John','Sutherland')
    first_name = LIB:SPLIT('Brian', ' '); --('Brian')
    (first_name, last_name) = LIB:SPLIT('Brian', ' '); -- throw SQL_FEW_VALUES
    (first_name, last_name) = LIB:SPLIT('Michael Forsyth Jr', ' ');--throw
SQL_MANY_VALUES
    (first_name, last_name) = LIB:SPLIT('Michael Forsyth Jr', ' ', 1); --
('Michael', 'Forsyth Jr')
    (area_code, first_num, last_num) = LIB:SPLIT_REGEXPR('02)2143-5300', '\(|
\)|-'); --(02, 2143, 5300)
END;

DO BEGIN
    USING SQLSCRIPT_STRING AS LIB;
    DECLARE arr INT ARRAY;
    DECLARE arr2 STRING ARRAY;
    DECLARE tv, tv2 TABLE(RESULT NVARCHAR(5000));

    arr = LIB:SPLIT_TO_ARRAY('10,20,30,40,50',','); --array(10,20,30,40,50)
    arr2 = LIB:SPLIT_REGEXPR_TO_ARRAY('Blake Kelly; Fred Randall; Bell Walsh;
Leonard Quinn; Chris McDonald', '\s*;\s*'); --array('Blake Kelly', 'Fred
Randall', 'Bell Walsh', 'Leonard Quinn', 'Chris McDonald')
    tv = LIB:SPLIT_TO_TABLE('10,20,30,40,50',','); --table[(10),(20),(30),
(40),(50)]
    tv2 = LIB:SPLIT_REGEXPR_TO_TABLE('10+20/30*40-50', '\+|\/|\*|-'); --
table[(10),(20),(30),(40),(50)]
END;
```

i Note

The SPLIT_TO_TABLE function currently does not support implicit table variable declaration.

```
CREATE PROCEDURE SPLIT_TO_TABLE_TEST AS BEGIN
    USING SQLSCRIPT_STRING AS lib;
    DECLARE tv TABLE(RESULT NVARCHAR(5000)); --Needs explicit table variable
declaration
    tv = LIB:SPLIT_TO_TABLE('a,b',',');
    SELECT * FROM :tv;
END;

CALL SPLIT_TO_TABLE_TEST(); -- [(a), (b)]
```

FORMAT Family Functions

FORMAT String

FORMAT functions support a new Python-style formatting.

Code Syntax

```
replacement_field :=  "{" [field_name] [":"format_spec] "}"
field_name :=  [column_name | integer]
format_spec :=  [sign][0][width][.precision][type]
sign := "+" | "-" | " "
width := integer
precision :=  integer
type := "s" | "b" | "c" | "d" | "o" | "x" | "X" | "e" | "E" | "f" | "F" | "g"
| "G"
```

String Representation Types

Type	Meaning
's'	String format
None	The same as 's'

Integer Representation Types

Type	Meaning
'b'	Binary format
'c'	Character
'd'	Decimal Integer
'o'	Octal format
'x'	HEX format. Using lower-case letters in the result
'X'	HEX format. Using upper-case letters in the result
None	The same as 'd'

Floating Point and Decimal Value Representation Types

Type	Meaning
'e'	Exponent notation. The default precision is 6.
'E'	Exponent notation. Using upper case 'E' in the result.
'f'	Fixed point. The default precision is 6.
'F'	Fixed point. Use NAN for nan and INF for inf in the result.
'g'	General format. The default precision is 6. Type 'e' with precision *p-1,* the number has exponent *exp* If *-4* <= *exp* < *p*, the same as 'f' and the precision is *p-1-exp* Else, the same as 'e' and precision is *p - 1*
'G'	General format. Using upper case 'E' in the result.
None	Similar to 'g'. The default precision is as high as needed to represent the number.

Example

Type	Example
Basic	FORMAT('{} {}', 'one', 'two') => 'one two' FORMAT('{1} {0}', 1, 2) => '2 1'
Truncating long strings	FORMAT('{:.5}', 'xylophone') =>'xylop' FORMAT('{:10.5}', 'xylophone') => 'xylop '
Numbers	FORMAT('{:d}', 42) => '42' FORMAT('{:f}', 3.141592653589793) => '3.141593' FORMAT('{:g}', 123456) => '123456' FORMAT('{:g}', 1234567) => '1.23456e+06' FORMAT('{:g}', 0.000123456) => '0.000123456' FORMAT('{:g}', 0.0000123456) => '1.23456e-05'
Padding Numbers	FORMAT('{:4d}', 42) => ' 42' FORMAT('{:06.2f}', 3.141592653589793) => '003.14' FORMAT('{:04d}', 42) => '0042'
Signed Numbers	FORMAT('{:+d}', 42) => '+42' FORMAT('{: d}', -23) => '-23' FORMAT('{: d}', 42) => ' 42'

Type	Example
Column Names	`tv = select 1 as first, 2 as last from dummy;` `FORMAT_TO _TABLE('{first} {last}', :tv) => [('1 2')]` `FORMAT_TO _TABLE('{first:04d} {last:02d}', :tv) => [('0001 02')]`

FORMAT

Returns a single formatted string using a given format string and additional arguments. Two type of additional arguments are supported: scalar variables and a single array. The first argument type accepts only scalar variables and should have a proper number and type of arguments. With the second argument type is allowed only one array that should have a proper size and type.

FORMAT_TO_TABLE/FORMAT_TO_ARRAY

Returns a table or an array with N formatted strings using a given table variable. FORMAT STRING is applied row by row.

Sample Code

```
DO BEGIN
    USING SQLSCRIPT_STRING AS LIB;
    DECLARE your_name STRING = LIB:FORMAT('{} {}', 'John', 'Sutherland');
--'John Sutherland'
    DECLARE name_age STRING = LIB:FORMAT('{1} {0}', 30, 'Sutherland');
--'Sutherland 30'
    DECLARE pi_str STRING = LIB:FORMAT('PI: {:06.2f}', 3.141592653589793);
--'PI: 003.14'
DECLARE ts STRING = LIB:FORMAT('Today is {}', TO_VARCHAR (current_timestamp,
'YYYY/MM/DD')); --'Today is 2017/10/18'
    DECLARE scores double ARRAY = ARRAY(1.4, 2.1, 40.3);
    DECLARE score_str STRING = LIB:FORMAT('{}-{}-{}', :scores);
--'1.4-2.1-40.3'
END;

DO BEGIN
    USING SQLSCRIPT_STRING AS LIB;
    DECLARE arr NVARCHAR(5000) ARRAY;
    declare tv table(result NVARCHAR(5000));
    --tt: [('John', 'Sutherland', 1988), ('Edward','Stark',1960)]
    DECLARE tt TABLE (first_name NVARCHAR(100), last_name NVARCHAR(100),
birth_year INT);
    tt.first_name[1] = 'John';
    tt.last_name[1] = 'Sutherland';
    tt.birth_year[1] = 1988;

    tt.first_name[2] = 'Edward';
    tt.last_name[2] = 'Stark';
    tt.birth_year[2] = 1960;

    arr = LIB:FORMAT_TO_ARRAY('{first_name} {last_name} was born in
{birth_year}', :tt);
    --['John Sutherland was born in 1988', 'Edward Stark was born in 1960']

    tv = LIB:FORMAT_TO_TABLE('{first_name} {last_name} was born in
{birth_year}', :tt);
    --tv: [('John Sutherland was born in 1988'), ('Edward Stark was born in
1960')]
END;
```

TABLE_SUMMARY

TABLE_SUMMARY converts a table variable into a single formatted string. It serializes the table into a human-friendly format, similar to the current result sets in the client. Since the table is serialized as a single string, the result is fetched during the PROCEDURE execution, not at the client-side fetch time. The parameter MAX_RECORDS limits the number of rows to be serialized. If the size of the formatted string is larger than NVARCHAR(8388607), only the limited size of the string is returned.

By means of SQLScript FORMAT functions, the values in the table are be formatted as follows:

- Integer types: formatted with SQLScript FORMAT string “d”.
- String types: formatted with SQLScript FORMAT string “s”.
- LOB types: formatted with SQLScript FORMAT string ".32s" (maximum 32 characters)
- Float types: formatted with SQLScript FORMAT string “.2f” (2 digit floating point value)
- Fixed types: formatted with SQLScript FORMAT string "" (default: preserve original precision + scale)

Sample Code

```
CREATE TABLE SAMPLE1(NAME nvarchar(32), AGE INT);
INSERT INTO SAMPLE1 VALUES ('John Bailey', 28);
INSERT INTO SAMPLE1 VALUES ('Kevin Lawrence', 56);
INSERT INTO SAMPLE1 VALUES ('Leonard Poole', 31);
INSERT INTO SAMPLE1 VALUES ('Vanessa Avery', 16);

DO
BEGIN
  USING SQLSCRIPT_STRING AS STRING;
  USING SQLSCRIPT_PRINT AS PRINT;
  T1 = SELECT * FROM SAMPLE1;
  LIB:PRINT_LINE(STRING:TABLE_SUMMARY(:T1, 3));
END;


------------------------
NAME,AGE
John Bailey,28
Kevin Lawrence,56
Leonard Poole,31
```

12.3 Built-in Library SQLSCRIPT_PRINT

Syntax

Code Syntax

```
CREATE LIBRARY SYS.SQLSCRIPT_PRINT LANGUAGE SQLSCRIPT AS BUILTIN
BEGIN
    PROCEDURE PRINT_LINE(IN VALUE NVARCHAR(8388607));
    PROCEDURE PRINT_TABLE(IN TAB TABLE(...), IN MAX_RECORDS INT DEFAULT 100);
END;
```

Description

The PRINT library makes it possible to print strings or even whole tables. It is especially useful when used together with the STRING library. The PRINT library procedures produce a server-side result from the parameters and stores it in an internal buffer. All stored strings will be printed in the client only after the end of the PROCEDURE execution. In case of nested execution, the PRINT results are delivered to the client after the end of the outermost CALL execution. The traditional result-set based results are not mixed up with PRINT results.

The PRINT library procedures can be executed in parallel. The overall PRINT result is flushed at once, without writing it on a certain stream for each request. SQLScript ensures the order of PRINT results, based on the description order in the PROCEDURE body, not on the order of execution.

> **Note**
>
> The built-in library SQLSCRIPT_PRINT is only supported in SAP HANA HDBSQL.

PRINT_LINE

This library procedure returns a string as a PRINT result. The procedure accepts NVARCHAR values as input, but also most other values are possible, as long as implicit conversion is possible (for example, INTEGER to NVARCHAR). Hence, most of the non-NVACHAR values can be used as parameters, since they are supported with SQLScript implicit conversion. Users can freely introduce string manipulation by using either a concatenation operator (||), a TO_NVARCHAR() value formatting, or the newly introduced SQLSCRIPT_STRING built-in library.

PRINT_TABLE

This library procedure takes a table variable and returns a PRINT result. PRINT_TABLE() parses a table variable into a single string and sends the string to the client. The parameter MAX_RECORDS limits the number of rows to be printed. PRINT_TABLE() is primarily used together with TABLE_SUMMARY of the STRING library.

Example

Sample Code

```
DO
BEGIN
  USING SQLSCRIPT_PRINT as LIB;
  LIB:PRINT_LINE('HELLO WORLD');
  LIB:PRINT_LINE('LINE2');
  LIB:PRINT_LINE('LINE3');
END;

DO
BEGIN
USING SQLSCRIPT_PRINT as LIB1;
  USING SQLSCRIPT_STRING as LIB2;
  LIB1:PRINT_LINE('HELLO WORLD');
  LIB1:PRINT_LINE('Here is SAMPLE1');
  T1 = SELECT * FROM SAMPLE1;
  LIB1:PRINT_LINE(LIB2:TABLE_SUMMARY(:T1));
  LIB1:PRINT_LINE('Here is SAMPLE2');
  T2 = SELECT * FROM SAMPLE2;
  LIB1:PRINT_TABLE(:T2);
  LIB1:PRINT_LINE('End of PRINT');
END;
```

12.4 Built-In Library SQLSCRIPT_LOGGING

SQLSCRIPT_LOGGING supports user level tracing for various types of SQLScript objects including procedures, table functions and SQLScript libraries.

Interface

Code Syntax

```
CREATE LIBRARY SQLSCRIPT_LOGGING AS BUILTIN BEGIN
    PUBLIC VARIABLE LEVEL_FATAL CONSTANT VARCHAR(5) =  'fatal';
    PUBLIC VARIABLE LEVEL_ERROR CONSTANT VARCHAR(5) = 'error';
    PUBLIC VARIABLE LEVEL_WARNING CONSTANT VARCHAR(7) = 'warning';
    PUBLIC VARIABLE LEVEL_INFO CONSTANT VARCHAR(4) = 'info';
    PUBLIC VARIABLE LEVEL_DEBUG CONSTANT VARCHAR(5) = 'debug';

    PUBLIC PROCEDURE CREATE_CONFIGURATION(CONFIGURATION_NAME VARCHAR(32));
    PUBLIC PROCEDURE DROP_CONFIGURATION(CONFIGURATION_NAME VARCHAR(32));

    PUBLIC PROCEDURE SET_OUTPUT_TABLE(CONFIGURATION_NAME VARCHAR(32),
SCHEMA_NAME NVARCHAR(256), TABLE_NAME NVARCHAR(256));
    PUBLIC PROCEDURE SET_LEVEL(CONFIGURATION_NAME VARCHAR(32), LEVEL
VARCHAR(7));

    PUBLIC PROCEDURE START_LOGGING(CONFIGURATION_NAME VARCHAR(32));
    PUBLIC PROCEDURE STOP_LOGGING(CONFIGURATION_NAME VARCHAR(32));
```

```
    PUBLIC PROCEDURE ADD_SQLSCRIPT_OBJECT(CONFIGURATION_NAME VARCHAR(32),
SCHEMA_NAME NVARCHAR(256), OBJECT_NAME NVARCHAR(256));
    PUBLIC PROCEDURE REMOVE_SQLSCRIPT_OBJECT(CONFIGURATION_NAME VARCHAR(32),
SCHEMA_NAME NVARCHAR(256), OBJECT_NAME NVARCHAR(256));

    PUBLIC PROCEDURE SET_FILTER(CONFIGURATION_NAME VARCHAR(32), TYPE
VARCHAR(16), ...);
    PUBLIC PROCEDURE ADD_FILTER(CONFIGURATION_NAME VARCHAR(32), TYPE
VARCHAR(16), ...);
    PUBLIC PROCEDURE REMOVE_FILTER(CONFIGURATION_NAME VARCHAR(32), TYPE
VARCHAR(16), ...);
    PUBLIC PROCEDURE UNSET_FILTER(CONFIGURATION_NAME VARCHAR(32), TYPE
VARCHAR(16));

    PUBLIC PROCEDURE LOG(LEVEL VARCHAR(7), TOPIC VARCHAR(32), MESSAGE
NVARCHAR(5000), ...);
END;
```

Description

Logging

An SQLScript object with LOG() is called a logging object. A log message can be categorized by its topic.

Procedure	Description
LOG (LEVEL, TOPIC, MESSAGE, ...)	A formatted log message is inserted in the output table if there is a configuration that enables the log. The invoking user should have the SQLSCRIPT LOGGING privilege for the current object. Saving log messages requires a configuration, otherwise the logging will be ignored. **! Restriction** Not available inside scalar user-defined functions and autonomous transaction blocks.

Configuration

A configuration is an imaginary object designed for logging settings. It is not a persistence object and lasts only until the end of the execution of the outermost statement. All settings for logging can be controlled by configurations. At least 1 configuration is required to save the log messages and up to 10 configurations can exist at a time.

Procedure	Description
CREATE_CONFIGURATION (CONFIGURATION_NAME)	A constructor to create a configuration with the given name. The CONFIGURATION_NAME should be unique during the whole execution.

Procedure	Description
DROP_CONFIGURATION (CONFIGURATION_NAME)	A destructor to remove the configuration with the given name. All configurations are destructed automatically when the outermost statement finishes its execution.
SET_LEVEL (CONFIGURATION_NAME, LEVEL)	This is a mandatory configuration setting. The Logging Library writes logs with higher (less verbose level) or equal level. The levels (from less verbose to more verbose) are: fatal, error, warning, info, debug

SQLScript Objects

SQLSCRIPT_LOGGING supports procedures, table functions and SQLScript libraries. SQLScript objects need to be registered to a configuration in order to collect logs from the objects. Only object-wise configurations are supported, a member-wise setting for libraries is not available.

Procedure	Description
ADD_SQLSCRIPT_OBJECT (CONFIGURATION_NAME, SCHEMA_NAME, OBJECT_NAME)	Opt-in for collecting logs from the object. It requires SQLSCRIPT LOGGING privilege for the object. Up to 10 objects can be added to a single configuration.
REMOVE_SQLSCRIPT_OBJECT (CONFIGURATION_NAME, SCHEMA_NAME, OBJECT_NAME)	Opt-out for collecting logs from the object

Output Table

Log messages from logging objects are inserted into an output table.

Procedure	Description
SET_OUTPUT_TABLE (CONFIGURATION_NAME, SCHEMA_NAME, TABLE_NAME)	Sets which table should be used as an output table. Only a single output table is supported. The table type must match SQLSCRIPT_LOGGING_TABLE_TYPE. This is a mandatory configuration setting

Filters

You can focus on specific messages by using filters. The OR operator is applied in case of multiple filter values:

```
call SET_FILTER('conf1', 'topic', 'sqlscript', 'compiler')
```

will be evaluated as

```
topic==‘sqlscript’ || topic == ‘compiler’
```

> **i Note**
>
> Currently only the type `'topic'` is supported.

Procedure	Description
SET_FILTER (CONFIGURATION_NAME, TYPE, ...)	Sets a filter for logging. Supports open-ended parameter for multiple filter values.
ADD_FILTER (CONFIGURATION_NAME, TYPE, ...)	Adds filter values to the filter type
REMOVE_FILTER (CONFIGURATION_NAME, TYPE, ...)	Remove filter values from the filter type
UNSET_FILTER (CONFIGURATION_NAME, TYPE)	Reset filter value to default (no filters)

Starting and Stopping the Logging

SQLSCRIPT_LOGGING requires to explicitly start the logging before calling an object. The logging is stopped implicitly when the outermost statement execution is finished but can also be stopped explicitly.

Procedure	Description
START_LOGGING (CONFIGURATION_NAME)	Start to collect logs for the given configuration. Throws an error if the output table or level are not set.
STOP_LOGGING (CONFIGURATION_NAME)	Stop collecting logs for the given configuration.

Configuration Steps

1. Create a log table for records by using SYS.SQLSCRIPT_LOGGING_TABLE_TYPE.
2. Create a procedure or call an anonymous block with following content:
    1. Define one or more configuration settings.
    2. Set up the logging level and the output table created in step 1.
    3. Add one ore more SQLScript objects (a procedure, a function, a library) to the configuration.
    4. (Optional) Set a filter by using a filter type and value.
    5. Start logging.
    6. Call the SQLScript object added to the configuration in step C.
    7. (Optional) Stop logging.
3. Call the procedure created in step 2.

Example

Sample Code

```
create function tudf1() returns table(a int) as begin
    using SQLSCRIPT_LOGGING as LIB;
    call LIB:LOG('debug', 'all', 'start tudf1');
    s = select 1 as a from dummy;
    call LIB:LOG('debug', 'all', 'this is tudf1');
    call LIB:LOG('debug', 'all', 'end tudf1');
    return :s;
end;

create function tudf2() returns table(a int) as begin
```

```
    using SQLSCRIPT_LOGGING as LIB;
    begin sequential execution
        call LIB:LOG('debug', 'all', 'start tudf2');
        call LIB:LOG('debug', 'all', 'tudf2 calls tudf1');
        s = select * from tudf1();
        call LIB:LOG('debug', 'all', 'end tudf2');
    end;
    return :s;
end;

create table t1 like sys.sqlscript_logging_table_type;
create table t2 like sys.sqlscript_logging_table_type;
create table t_all like sys.sqlscript_logging_table_type;

DO BEGIN
    using SQLSCRIPT_LOGGING as LIB;
    -- conf1
    call LIB:CREATE_CONFIGURATION('conf1');
    call LIB:ADD_SQLSCRIPT_OBJECT('conf1', current_schema, 'TUDF1');
    call LIB:SET_OUTPUT_TABLE('conf1', current_schema, 'T1');
    call LIB:SET_LEVEL('conf1', 'debug');
    call LIB:START_LOGGING('conf1');

    -- conf2
    call LIB:CREATE_CONFIGURATION('conf2');
    call LIB:ADD_SQLSCRIPT_OBJECT('conf2', current_schema, 'TUDF2');
    call LIB:SET_OUTPUT_TABLE('conf2', current_schema, 'T2');
    call LIB:SET_LEVEL('conf2', 'debug');
    call LIB:START_LOGGING('conf2');

    -- all
    call LIB:CREATE_CONFIGURATION('conf_all');
    call LIB:ADD_SQLSCRIPT_OBJECT('conf_all', current_schema, 'TUDF1');
    call LIB:ADD_SQLSCRIPT_OBJECT('conf_all', current_schema, 'TUDF2');
    call LIB:SET_OUTPUT_TABLE('conf_all', current_schema, 'T_ALL');
    call LIB:SET_LEVEL('conf_all', 'debug');
    call LIB:START_LOGGING('conf_all');

    select * from tudf2();
END;

create user sqlscript_logging_user_a password Dummy1234 NO
FORCE_FIRST_PASSWORD_CHANGE;

connect sqlscript_logging_user_a password Dummy1234;
create procedure p1 sql security invoker as begin
    using SQLSCRIPT_LOGGING as LIB;
    call LIB:LOG('error', 'sqlscript', 'hello world');
end;

grant execute, sqlscript logging on p1 to SYSTEM;

connect SYSTEM password manager;

DO BEGIN
    using SQLSCRIPT_LOGGING as LIB;
    call LIB:CREATE_CONFIGURATION('conf1');
    call LIB:SET_OUTPUT_TABLE('conf1', current_schema, 'T1');
    call LIB:SET_LEVEL('conf1', 'debug');
    call LIB:ADD_SQLSCRIPT_OBJECT('conf1', 'SQLSCRIPT_LOGGING_USER_A', 'P1');
    call LIB:START_LOGGING('conf1');
    call SQLSCRIPT_LOGGING_USER_A.p1;
    call LIB:STOP_LOGGING('conf1');
END;
```

Related Information

SQLSCRIPT_LOGGING Privilege [page 257]
SQLSCRIPT_LOGGING_TABLE_TYPE [page 258]

12.4.1 SQLSCRIPT_LOGGING Privilege

SQLSCRIPT LOGGING privilege is required to collect logs for a SQLScript object. A logging user can be different from the procedure owner and the owner can expose log messages to other users selectively by using this privilege.

Syntax

Code Syntax

```
<schema_privilege> ::= ALL PRIVILEGES |...| SQLSCRIPT LOGGING
<object_privilege> ::= ALL PRIVILEGES |...| SQLSCRIPT LOGGING
```

Example

Sample Code

```
connect sqlscript_logging_user_a password Dummy1234;
create procedure p1 sql security invoker as begin
    using SQLSCRIPT_LOGGING as LIB;
    call LIB:LOG('error', 'sqlscript', 'hello world');
end;

grant execute, sqlscript logging on p1 to SYSTEM;
```

Related Information

Built-In Library SQLSCRIPT_LOGGING [page 252]

12.4.2 SQLSCRIPT_LOGGING_TABLE_TYPE

SQLSCRIPT_LOGGING:LOG can only write logs to a table with a predefined table type. You can create an output table using the type SYS.SQLSCRIPT_LOGGING_TABLE_TYPE or the public synonym SQLSCRIPT_LOGGING_TABLE_TYPE.

Definition

```
CREATE TYPE SYS.SQLSCRIPT_LOGGING_TABLE_TYPE AS TABLE ( HOST VARCHAR(64) NOT
NULL, PORT INTEGER NOT NULL, THREAD_ID BIGINT NOT NULL,
    CONNECTION_ID INTEGER NOT NULL, TRANSACTION_ID INTEGER NOT NULL, TIMESTAMP
TIMESTAMP NOT NULL, LEVEL VARCHAR(7) NOT NULL, USER_NAME NVARCHAR(256) NOT NULL,
TOPIC VARCHAR(32) NOT NULL,
    DATABASE_NAME NVARCHAR(256), SCHEMA_NAME NVARCHAR(256), OBJECT_NAME
NVARCHAR(256), MEMBER_NAME NVARCHAR(256),
    SOURCE_LINE INTEGER NOT NULL, MESSAGE NVARCHAR(5000));

CREATE PUBLIC SYNONYM SQLSCRIPT_LOGGING_TABLE_TYPE FOR
SYS.SQLSCRIPT_LOGGING_TABLE_TYPE;
```

Example

Sample Code

```
create table mytab like sys.sqlscript_logging_table_type;
```

Related Information

Built-In Library SQLSCRIPT_LOGGING [page 252]

13 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 that 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 (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 decisions of the optimizer.

Sample Code

```
CALL PROC (var1=>1, var2=>2, mytab=>?)
```

14 Supportability

14.1 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:

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 compiled 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 (microseconds)
STATEMENT_EXECUTION_TIME	BIGINT	Elapsed time for executing statement (microseconds)
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
STATEMENT_MATERIALIZATION_TIME	BIGINT	Specifies the ITAB materialization time.
STATEMENT_MATERIALIZATION_MEMORY_SIZE	BIGINT	Specifies the memory size of the ITAB materialization.

STATEMENT_EXECUTION_MEMORY_SIZE	BIGINT	Shows the peak memory (in bytes) used for executing a statement. In case of distributed execution, it is a sum of the local peak memories of multiple servers.
		i Note
		By default this column shows '-1'. You need to perform the following configurations to enable the statistics.
		`global.ini: ('resource_tracking', 'enable_tracking') = 'true’`
		`global.ini: ('resource_tracking', 'memory_tracking') = 'true'`
		The value is filled only after the execution is complete. During the execution, it shows -1.

M_ACTIVE_PROCEDURES is also helpful for analyzing long-running procedures and for determining their current status. You can run the following query from another session to find out 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 an INI-configuration `monitoring_level` to control the granularity of monitoring level:

Level	Description
0	Disables profiling information, such as STATE-MENT_START_TIME and STATEMENT_END_TIME.
1	Default mode. Enables profiling information, but still disables the collection of STATEMENT_PARAMTER values.
2	Full information for the monitoring view

To prevent flooding of the memory with irrelevant data, the number of records is limited. If the record count exceeds the given threshold, the first record is deleted irrespective of its status. The limit can be adjusted the INI-parameter `execution_monitoring_limit`, for example `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).

The default behavior of M_ACTIVE_PROCEDURES is to keep the records of completed internal statements until the parent procedure is complete. 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 are retained after execution and `RETENTION_PERIOD_FOR_SQLSCRIPT_CONTEXT` defines how long the result should be kept in M_ACTIVE_PROCEDURES. The following options are possible:

- 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 is set: no records are kept.

> **i Note**
>
> All configuration parameters need to be defined in the section `sqlscript`.

14.2 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.

14.2.1 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 on the SAP Help Portal.

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 on the SAP Help Portal.

> **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.

14.3 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 and for the length or precision/scale information. This is particularly important for signature variables because 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 The SAP HANA SQL and System Views Reference for additional details on which data types conversion are supported).

If the derived type is larger (for example, `BIGINT`) than the expected type (for example, `INTEGER`) this can lead to errors, as illustrated 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, for example 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 illustrated 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 the 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 return 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:

Level	Output	Description
silent	--	Ignore potential type error
warn	general warning: Declared type "VARCHAR(2)" of attribute "A" not same as assigned type "VARCHAR(10)"	Print warning in case of type mismatch(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

> **i Note**
>
> Both configuration parameters need to be defined in the `sqlscript` section.

14.4 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.

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	-

[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

14.4.1 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.

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.

> **i 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).

> **i 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 on the SAP Help Portal.

14.4.2 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 you can define an arbitrary number of conditions. 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.

> **i Note**
>
> If a variable value changes to `NULL`, the debugger will not break since it cannot evaluate the expression anymore.

14.4.3 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 a function occurs. User-defined exceptions are also handled.

The debugger stops on the line, where the exception is thrown, and allows access to the current value of all local variables, the call stack and a short information about the error. After that, the execution can continue and you can step into the exception handler or into further exceptions (fore example, on a CALL statement).

14.4.4 Save Table

Save Table allows you to store the result set of a table variable into a persistent table in a predefined schema in a debugging session.

14.5 EXPLAIN PLAN for Call

Syntax

```
EXPLAIN PLAN [SET STATEMENT_NAME = <statement_name>] FOR <explain_plan_entry>
```

Syntax Elements

Syntax Element	Description
`<statement_name> ::= <string_literal>`	Specifies the name of a specific execution plan in the output table for a given SQL statement
`<explain_plan_entry>`	Specifies the entry to explain

Syntax Element	Description

`<explain_plan_entry> ::= <call_statement> | SQL PLAN CACHE ENTRY <plan_id>`
 	
`<plan_id> ::= <integer_literal>`
	`<plan_id>` specifies the identifier of the entry in the SQL plan cache to be explained. Refer to the M_SQL_PLAN_CACHE monitoring view to find the `<plan_id>` for the desired cache entry.
	
	`<call_statement>` specifies the procedure call to explain the plan for. For more information about subqueries, see the CALL statement.

i Note

The `EXPLAIN PLAN [SET STATEMENT_NAME = <statement_name>] FOR SQL PLAN CACHE ENTRY <plan_id>` command can only be run by users with the OPTIMIZER_ADMIN privilege.

Description

`EXPLAIN PLAN` provides information about the compiled plan of a given procedure. It inserts each piece of information into a system global temporary table named `EXPLAIN_CALL_PLANS`. The result is visible only within the session where the `EXPLAIN PLAN` call is executed.

`EXPLAIN PLAN` generates the plan information by using the given SQLScript Engine Plan structure. It traverses the plan structure and records each information corresponding to the current SQLScript Engine Operator.

In the case of invoking another procedure inside of a procedure, `EXPLAIN PLAN` inserts the results of the invoked procedure (callee) under the invoke operator (caller), although the actual invoked procedure is a sub-plan which is not located under the invoke operator.

Another case is the `else` operator. `EXPLAIN PLAN` generates a dummy `else` operator to represent alternative operators in the condition operator.

Example

```
CREATE PROCEDURE proc_p1(a int) as
begin
    declare i int default 0;
    create table tab1 (attr1 int);
    if a > 0 then
        select 5 from dummy;
    else
        select 10 from dummy;
    end if;
    while i < 10 do
        insert into tab1 values (1);
        i := i + 1;
    end while;
    drop table tab1;
end;
```

```
EXPLAIN PLAN SET STATEMENT_NAME = 'test' FOR CALL proc_p1(1);
```

You can retrieve the result by selecting from the table `EXPLAIN_CALL_PLANS`.

```
SELECT * FROM EXPLAIN_CALL_PLANS WHERE STATEMENT_NAME = 'test';
```

The EXPLAIN PLAN FOR select query deletes its temporary table by HDB client but in the case of EXPLAIN PLAN FOR call, it is not yet supported. To delete rows in the table, execute a delete query from EXPLAIN_CALL_PLANS table or close the current session.

> **i Note**
>
> Client integration is not available yet. You need to use the SQL statement above to retrieve the plan information.

14.6 EXPLAIN PLAN for Table User-Defined Functions

Syntax

```
EXPLAIN PLAN [ SET STATEMENT_NAME = <statement_name> ] FOR <explain_plan_entry>
```

Description

To improve supportability, SQLScript now provides more detailed information on the SQLScript Table User-Defined Function (TUDF) native operator in EXPLAIN PLAN.

TUDF is automatically unfolded when applicable. If unfolding is blocked, the cause is listed in EXPLAIN PLAN. This feature automatically applies to EXPLAIN PLAN FOR select statements under the following conditions:

- the SELECT query uses a TUDF
- the TUDF cannot be unfolded.

If the two conditions are met, an SQL PLAN is automatically generated along with an SQLScript Engine Plan of the TUDF.

Behavior

EXPLAIN PLAN for SQLScript TUDF native operator provides the following compiled plans:

- EXPLAIN PLAN FOR select statement from SQL PLAN. The result is retrievable from the table EXPLAIN_PLAN_TABLE.
- EXPLAIN PLAN FOR CALL from SQLScript Plan. The result is retrievable from the table EXPLAIN_CALL_PLANS.

EXPLAIN_PLAN_TABLE	EXPLAIN_CALL_PLANS
OPERATOR_PROPERTIES field: • lists the detailed reasons why the SQLScript TUDF is not unfolded (see the table below) • contains a comma-separated list of objects used within the TUDF	The internal SQLScript plan of the outermost TUDF is explained. It is automatically generated along with EXPLAIN_PLAN_TABLE with the same STATEMENT_NAME.

List of Possible Reasons Why TUDF Is Not Unfolded

Reasons	Explanation
NOT UNFOLDED BECAUSE FUNCTION BODY CANNOT BE SIMPLIFIED TO A SINGLE STATEMENT	Multiple statements in TUDF body cannot be simplified into a single statement.
NOT UNFOLDED DUE TO ANY TABLE	TUDF uses ANY TABLE type.
NOT UNFOLDED DUE TO BINARY TYPE PARAMETER	TUDF has a binary type as its parameter.
NOT UNFOLDED DUE TO DEV_NO_SQLSCRIPT_SCENARIO HINT	The caller of TUDF disables unfolding with the DEV_NO_PREPARE_SQLSCRIPT_SCENARIO hint.
NOT UNFOLDED DUE TO DEBUGGING SESSION	TUDF is executed in debugging session.
NOT UNFOLDED DUE TO ENCRYPTED PROCEDURE OR FUNCTION	TUDF is an encrypted function.
NOT UNFOLDED DUE TO IMPERATIVE LOGICS	TUDF has an imperative logic, including SQLScript IF, FOR,WHILE, or LOOP statements.
NOT UNFOLDED DUE TO INTERNAL SQLSCRIPT OPERATOR	TUDF unfolding is blocked by an internal SQLScript operator.
NOT UNFOLDED DUE TO INPUT PARAMETER TYPE MISMATCH	The type of the input argument does not match the defined type of the TUDF input parameter.
NOT UNFOLDED DUE TO JSON OR SYSTEM FUNCTION	TUDF uses JSON or system function.
NOT UNFOLDED DUE TO NATIVE SQLSCRIPT OPERATOR	TUDF has a SQLScript native operator, which does not have an appropriate SQL counterpart.
NOT UNFOLDED DUE TO NO CALCULATION VIEW UNFOLDING	The caller of TUDF disables *Calculation View* unfolding.
NOT UNFOLDED DUE TO PRIMARY KEY CHECK	TUDF has a primary key check.
NOT UNFOLDED DUE TO RANGE RESTRICTION	Table with RANGE RESTRICTION is used within the TUDF.
NOT UNFOLDED DUE TO RECURSION	The TUDF has a recursive call.
NOT UNFOLDED DUE TO SEQUENCE OBJECT	A SEQUENCE variable is used within the TUDF.
NOT UNFOLDED DUE TO SEQUENTIAL EXECUTION	TUDF is executed with SEQUENTIAL EXECUTION clause.
NOT UNFOLDED DUE TO SPATIAL TYPE PARAMETER	TUDF has a spatial type as its parameter.
NOT UNFOLDED DUE TO TIME TRAVEL OPTION	TUDF uses a history table OR the time travel option is used.
NOT UNFOLDED DUE TO WITH CLAUSE	TUDF uses a WITH clause.

Reasons	Explanation
NOT UNFOLDED DUE TO WITH HINT	TUDF uses a WITH HINT clause that cannot be unfolded.
NOT UNFOLDED DUE TO WITH PARAMETERS CLAUSE	TUDF uses a WITH PARAMETERS clause.
NOT UNFOLDED DUE TO XML CLAUSE	TUDF has an XML clause.

Example

Sample Code

```
create function func() returns table (a int)
as begin
    declare k int = 0;
    declare x int = 0;
    for x in 1..4 do -- imperative logic
        k := :k + :x;
    end for;
    return select :k as a from dummy;
end;
```

Sample Code

```
explain plan set statement_name = 'TUDF_PLAN' for select * from func();
```

You can retrieve the SQL Plan from the EXPLAIN_PLAN_TABLE.

Sample Code

```
select statement_name, operator_name, operator_details, operator_properties,
schema_name, table_name from explain_plan_table where statement_name
='TUDF_PLAN';
```

STATEMENT_NAME	OPERATOR_NAME	OPERATOR_DETAILS	OPERATOR_PROPERTIES	SCHEMA_NAME	TABLE_NAME
TUDF_PLAN	COLUMN SEARCH	FUNC.A	LATE MATERIALIZATION, ENUM_BY: CS_TABLE	?	?
TUDF_PLAN	TABLE FUNCTION		NOT UNFOLDED DUE TO IMPERATIVE LOGICS, ACCESSED_OBJECT_NAMES: SYS.DUMMY, PUBLIC.DUMMY	SYSTEM	FUNC

You can retrieve the SQL Plan from the table EXPLAIN_CALL_PLANS.

Sample Code

```
select statement_name, operator_name, operator_string, procedure_name,
execution_engine from explain_call_plans where statement_name ='TUDF_PLAN';
```

STATEMENT_NAME	OPERATOR_NAME	OPERATOR_STRING	PROCEDURE_NAME	EXECUTION_ENGINE
TUDF_PLAN	Function	select * from func()	FUNC	SQLScript
TUDF_PLAN	Sequential Op		FUNC	SQLScript
TUDF_PLAN	Initial Op		FUNC	SQLScript
TUDF_PLAN	Expression Op		FUNC	SQLScript
TUDF_PLAN	Range Op		FUNC	SQLScript
TUDF_PLAN	Assign		FUNC	SQLScript
TUDF_PLAN	Loop Op		FUNC	SQLScript
TUDF_PLAN	Sequential Op		FUNC	SQLScript
TUDF_PLAN	Range Op		FUNC	SQLScript
TUDF_PLAN	Range Op		FUNC	SQLScript
TUDF_PLAN	Expression Op		FUNC	SQLScript
TUDF_PLAN	Assign		FUNC	SQLScript
TUDF_PLAN	Continue		FUNC	SQLScript
TUDF_PLAN	Assign	select __typed_Integer__($1) as a from dummy	FUNC	SQLScript, EAPI
TUDF_PLAN	Return		FUNC	SQLScript
TUDF_PLAN	Terminal Op		FUNC	SQLScript

Limitations

- EXPLAIN PLAN is generated once per statement. It will not be regenerated regardless of configuration changes. To regenerate EXPLAIN PLAN, the SQL PLAN CACHE should be cleared via `ALTER SYSTEM CLEAR SQL PLAN CACHE`.
- EXPLAIN_CALL_PLAN accumulates execution plans over time. That content is not be automatically deleted.

14.7 SQLScript Code Coverage

Description

SAP HANA stores the results of a code coverage session in the M_SQLSCRIPT_CODE_COVERAGE_RESULTS monitoring view and stores the definitions of objects that were used during a code coverage session in the M_SQLSCRIPT_CODE_COVERAGE_OBJECT_DEFINITIONS monitoring view.

Syntax

To start SQLScript code coverage:

```
ALTER SYSTEM START SQLSCRIPT CODE COVERAGE
 [ FOR DEBUG TOKEN <token_id> ]
 [ FOR USER <user_id> ]
 [ FOR APPLICATION USER <application_user_id> ]
 [ FOR SESSION <session_id> ]
```

To stop SQLScript code coverage:

```
ALTER SYSTEM STOP SQLSCRIPT CODE COVERAGE
```

Syntax Elements

`<token_id>`: specifies the token that the code coverage applies to.

`<user_id>`: specifies the database user ID that the code coverage applies to.

`<application_user_id>`: specifies the ID of the application user that the code coverage applies to.

`<session_id>`: specifies the ID of the session that the code coverage applies to.

Select from the monitoring views at any time, and from any column, you are interested in after starting code coverage. However, the full content of code coverage run is visible only after the query triggered in the second session (which is being covered) finishes (described in the second example, below).

The content in the monitoring views is overwritten in these views each time you stop a SQLScript code coverage session and start a new one. Since the data is temporary, copy or export the content from these views to retain data recorded by a SQLScript code coverage session before executing ALTER SYSTEM STOP SQLSCRIPT CODE COVERAGE.

You must have at least two connections for code coverage. In the first session you execute the codes on which you run code coverage, and in the second session you start the code coverage for a specific connection ID to record the coverage.

> ⚠ Caution
>
> You must have the EXECUTE, DEBUG, and ATTACH_DEBUGGER privileges to perform code coverage.

Example

SAP HANA requires two sessions to perform the code coverage. The examples below use session A to execute the code on which you run code coverage, and session B starts the code coverage for a specific connection ID to record the coverage.

1. In either session, create the `limitedLoop` and `dummy_proc` procedures:

```
CREATE PROCEDURE limitedLoop() AS
BEGIN
DECLARE i BIGINT := 0;
LOOP
i := i + 1;
IF :i > 27 THEN
BREAK;
END IF;
END LOOP;
END;
CREATE PROCEDURE dummy_proc() AS
BEGIN
SELECT * FROM DUMMY;
CALL limitedLoop();
END;
```

2. From session A, issue this to determine the connection ID:

```
SELECT SESSION_CONTEXT('CONN_ID') FROM DUMMY;
```

3. In session B, start code coverage by using the connection ID of the user who is executing the code in session A (this example uses a connection ID of 203247):

```
ALTER SYSTEM START SQLSCRIPT CODE COVERAGE FOR SESSION '203247';
```

4. From session A, call the `dummy_proc` procedure:

```
CALL dummy_proc();
```

5. From session B, view the code coverage by querying the M_SQLSCRIPT_CODE_COVERAGE_RESULTS and M_SQLSCRIPT_CODE_COVERAGE_OBJECT_DEFINITIONS monitoring views

```
SELECT * FROM M_SQLSCRIPT_CODE_COVERAGE_RESULTS;
SELECT * FROM M_SQLSCRIPT_CODE_COVERAGE_OBJECT_DEFINITIONS;
```

   If required, store the contents of the monitoring views for future reference (this can be a regular or a local temporary table):

```
CREATE LOCAL TEMPORARY TABLE "#SomeTableName" AS (SELECT * FROM
M_SQLSCRIPT_CODE_COVERAGE_RESULTS) WITH DATA;
```

6. From session B, disable the code coverage (this also clears the existing code coverage):

```
ALTER SYSTEM STOP SQLSCRIPT CODE COVERAGE;
```

14.8 SQLScript Code Analyzer

The SQLScript Code Analyzer consists of two built-in procedures that scan CREATE FUNCTION and CREATE PROCEDURE statements and search for patterns indicating problems in code quality, security or performance.

Interface

The view SQLSCRIPT_ANALYZER_RULES listing the available rules is defined in the following way:

Column Name	Type
RULE_NAMESPACE	VARCHAR(16)
RULE_NAME	VARCHAR(64)
CATEGORY	VARCHAR(16)
SHORT_DESCRIPTION	VARCHAR(256)
LONG_DESCRIPTION	NVARCHAR(5000)
RECOMMENDATION	NVARCHAR(5000)

Procedure ANALYZE_SQLSCRIPT_DEFINITION

The procedure ANALYZE_SQLSCRIPT_DEFINITION can be used to analyze the source code of a single procedure or a single function that has not been created yet. If not yet existing objects are referenced, the procedure or function cannot be analyzed.

Sample Code

```
CREATE PROCEDURE ANALYZE_SQLSCRIPT_DEFINITION(IN OBJECT_DEFINITION NCLOB,
                                              IN RULES TABLE(RULE_NAMESPACE
VARCHAR(16), RULE_NAME VARCHAR(64), CATEGORY VARCHAR(16)),
                                              OUT FINDINGS
TABLE(RULE_NAMESPACE VARCHAR(16), RULE_NAME VARCHAR(64), CATEGORY VARCHAR(16),

SHORT_DESCRIPTION NVARCHAR(256), START_POSITION INT, END_POSITION INT)

                                            ) AS BUILTIN
```

Parameter	Description
OBJECT_DEFINITION	Contains the DDL string of the SQLScript function or procedure that should be analyzed
RULES	Rules to be used for the analysis. Available rules can be retrieved from the view SQLSCRIPT_ANALYZER_RULES

Parameter	Description
FINDINGS	Lists potential problems found during the analysis

Procedure ANALYZE_SQLSCRIPT_OBJECTS

The procedure ANALYZE_SQLSCRIPT_OBJECTS can be used to analyze the source code of multiple already existing procedures or functions.

Sample Code

```
CREATE PROCEDURE ANALYZE_SQLSCRIPT_OBJECTS(IN OBJECTS_TO_ANALYZE
TABLE(SCHEMA_NAME NVARCHAR(256), OBJECT_NAME NVARCHAR(256)),
                                           IN RULES TABLE(RULE_NAMESPACE
VARCHAR(16),RULE_NAME VARCHAR(64), CATEGORY VARCHAR(16)),
                                           OUT OBJECT_DEFINITIONS
TABLE(OBJECT_DEFINITION_ID INT, SCHEMA_NAME NVARCHAR(256), OBJECT_NAME
NVARCHAR(256), OBJECT_DEFINITION NCLOB),
                                           OUT FINDINGS
TABLE(OBJECT_DEFINITION_ID INT, RULE_NAMESPACE VARCHAR(16), RULE_NAME
VARCHAR(64), CATEGORY VARCHAR(16),

SHORT_DESCRIPTION NVARCHAR(256), START_POSITION INT, END_POSITION INT)) AS
BUILTIN
```

Parameter	Description
OBJECTS	A list of existing SQLScript procedures and functions that should be analyzed
RULES	Rules that should be used for the analysis. Available rules can be retrieved from the view SQLSCRIPT_ANALYZER_RULES.
OBJECT_DEFINITIONS	Contains the names and definitions of all objects that were analyzed, including those without any findings
FINDINGS	Lists potential problems found by the analysis. Affected objects are identified by their OBJECT_DEFINITION_ID, which is also used in OBJECT_DEFINITIONS

Rules

The following rules, provided by SAP, are currently available:

Rule Name	Category
UNNECESSARY_VARIABLE [page 281]	CONSISTENCY
UNUSED_VARIABLE_VALUE [page 281]	CONSISTENCY

Rule Name	Category
UNCHECKED_SQL_INJECTION_SAFETY [page 281]	SECURITY
SINGLE_SPACE_LITERAL [page 281]	CONSISTENCY
COMMIT_OR_ROLLBACK_IN_DYNAMIC_SQL [page 281]	STYLE
USE_OF_SELECT_IN_SCALAR_UDF [page 282]	PERFORMANCE
USE_OF_UNASSIGNED_SCALAR_VARIABLE [page 282]	CONSISTENCY
DML_STATEMENTS_IN_LOOPS [page 283]	PERFORMANCE
USE_OF_CE_FUNCTIONS [page 284]	PERFORMANCE
USE_OF_DYNAMIC_SQL [page 284]	PERFORMANCE
ROW_COUNT_AFTER_SELECT [page 284]	BEHAVIOR
ROW_COUNT_AFTER_DYNAMIC_SQL [page 284]	BEHAVIOR

UNNECESSARY_VARIABLE

For each variable, it is tested if it is used by any output parameter of the procedure or if it influences the outcome of the procedure. Statements relevant for the outcome could be DML statements, implicit result sets, conditions of control statements.

UNUSED_VARIABLE_VALUE

If a value, assigned to a variable, is not used in any other statement, the assignment can be removed. In case of default assignments in DECLARE statements, the default is never used.

UNCHECKED_SQL_INJECTION_SAFETY

Parameters of type string should always be checked for SQL injection safety, if they are used in dynamic SQL. This rule checks if the function `is_sql_injection_safe` is called for every parameter of that type.

For a simple conditional statement like `IF is_sql_injection_safe(:var) = 0 THEN...`, the control flow in the true branch is checked. The procedure should either end (by returning or by throwing an error) or the unsafe parameter value should be escaped with the functions `escape_single_quotes` or `escape_double_quotes`, depending on where the value is used.

If the condition is more complex (for example, more than one variable is checked in one condition), a warning will be displayed because it is only possible to check if any execution of the dynamic SQL has passed the SQL injection check.

SINGLE_SPACE_LITERAL

This rule searches for string laterals consisting of only one space. If ABAP VARCHAR MODE is used, such string literals are treated as empty strings. In this case CHAR(32) can be used instead of ' '.

COMMIT_OR_ROLLBACK_IN_DYNAMIC_SQL

This rule detects dynamic SQL that uses the COMMIT or ROLLBACK statements. It is recommended to use COMMIT and ROLLBACK directly in SQLScript, thus eliminating the need of dynamic SQL.

This rule has some limitations in terms of analyzing dynamic SQL:

- It can only check dynamic SQL that uses a constant string (for example, EXEC 'COMMIT';). It cannot detect dynamic SQL that evaluates any expression (for example, EXEC 'COM' || 'MIT';)

- It can only detect simple strings containing COMMIT or ROLLBACK and whitespaces, as well as simple comments. More complex strings might not be detected by this rule.

USE_OF_SELECT_IN_SCALAR_UDF

This rule detects and reports SELECT statements in scalar UDFs. SELECT statements in scalar UDFs can affect performance. If table operations are really needed, procedures or table UDFs should be used instead.

Sample Code

USE_OF_SELECT_IN SCALAR_UDF

```
DO BEGIN
  tab = SELECT RULE_NAMESPACE, RULE_NAME, category FROM
SQLSCRIPT_ANALYZER_RULES where rule_name = 'USE_OF_SELECT_IN_SCALAR_UDF';
  CALL ANALYZE_SQLSCRIPT_DEFINITION('
    CREATE FUNCTION f1(a INT) RETURNS b INT AS
    BEGIN
      DECLARE x INT;
      SELECT count(*) into x FROM _sys_repo.active_object;
      IF :a > :x THEN
        SELECT count(*) INTO b FROM _sys_repo.inactive_object;
      ELSE
        b = 100;
      END IF;
    END;', :tab, res);
  SELECT * FROM :res;
END;
```

The following findings will be reported in this example:

RULE_NAME-SPACE	RULE_NAME	Category	SHORT_DESCRIP-TION	START_POSITION	END_POSITION
SAP	USE_OF_SE-LECT_IN_SCA-LAR_UDF	PERFORMANCE	Found SELECT statement in Sca-lar UDF	186	240
SAP	USE_OF_SE-LECT_IN_SCA-LAR_UDF	PERFORMANCE	Found SELECT statement in Sca-lar UDF	97	149

USE_OF_UNASSIGNED_SCALAR_VARIABLE

The rule detects variables which are used but were never assigned explicitly. Those variables still have their default value when used, which might be undefined. It is recommended to assign a default value (that can be NULL) to be sure that you get the intended value when you read from the variable. If this rule returns a warning or an error, check in your code if have not assigned a value to the wrong variable. Always rerun this rule after changing code, since it is possible that multiple errors trigger only a single message and the error still persists.

For every DECLARE statement this rule returns one of the following:

- <nothing>: if the variable is always assigned before use or not used. Everything is correct.
- Variable <variable> may be unassigned: if there is at least one branch, where the variable is unassigned when used, even if the variable is assigned in other branches.
- Variable <variable> is used but was never assigned explicitly: if the variable will never have a value assigned when used.

DML_STATEMENTS_IN_LOOPS

The rule detects the following DML statements inside loops - INSERT, UPDATE, DELETE, REPLACE/UPSERT. Sometimes it is possible to rewrite the loop and use a single DML statement to improve performance instead.

In the following example a table is updated in a loop. This code can be rewritten to update the table with a single DML statement.

Sample Code

DML Statements in Loops

```
DO BEGIN
tab = select rule_namespace, rule_name, category from
sqlscript_analyzer_rules;
call analyze_sqlscript_definition('

Create procedure example() AS
BEGIN
    declare i int = 0;
    declare size int;
    declare olda int;
    declare newa int;
    CREATE TABLE T1 (a INT);
    INSERT INTO T1 VALUES(1);
    INSERT INTO T1 VALUES(-2);
    INSERT INTO T1 VALUES(-1);
    INSERT INTO T1 VALUES(3);
    T2 = SELECT * FROM T1;
    SELECT COUNT(*) INTO size FROM T1;
    FOR i IN 1 .. :size DO
        olda = :T2.A[:i];
        newa = :olda;
        if :olda < 0 then
            newa = 0;
           end if;
        UPDATE T1 SET A= :newa WHERE A = :olda;
    END FOR;
    SELECT * FROM T1;
END;

    ', :tab, res);
select * from :res;
end;

// Optimized version

drop procedure example2;
Create procedure example2() AS
BEGIN
    declare i int = 0;
    declare size int;
    declare olda int;
    declare newa int;
    CREATE TABLE T1 (a INT);
    INSERT INTO T1 VALUES(1);
    INSERT INTO T1 VALUES(-2);
    INSERT INTO T1 VALUES(-1);
    INSERT INTO T1 VALUES(3);
    UPDATE T1 SET A = 0 WHERE A < 0;
    SELECT * FROM T1;
END;

DROP TABLE T1;
CALL EXAMPLE2();
```

USE_OF_CE_FUNCTIONS

The rule checks whether Calculation Engine Plan Operators (CE Functions) are used. Since they make optimization more difficult and lead to performance problems, they should be avoided. For more information and how to replace them using only plain SQL, see Calculation Engine Plan Operators [page 216]

USE_OF_DYNAMIC_SQL

The rule checks and reports, if dynamic SQL is used within a procedure or a function.

ROW_COUNT_AFTER_SELECT

The rule checks, if the system variable ::ROWCOUNT is used after a SELECT statement.

ROW_COUNT_AFTER_DYNAMIC_SQL

The rule checks, if the system variable ::ROWCOUNT is used after the use of dynamic SQL.

Examples

Sample Code

```
DO BEGIN
  tab = SELECT rule_namespace, rule_name, category FROM
SQLSCRIPT_ANALYZER_RULES; -- selects all rules
  CALL ANALYZE_SQLSCRIPT_DEFINITION('
      CREATE PROCEDURE UNCHECKED_DYNAMIC_SQL(IN query NVARCHAR(500)) AS
      BEGIN
          DECLARE query2 NVARCHAR(500) = ''SELECT '' || query || '' from
tab'';
          EXEC :query2;
          query2 = :query2; --unused variable value
      END', :tab, res);
    SELECT * FROM :res;
END;
```

Sample Code

```
DO BEGIN
  tab = SELECT rule_namespace, rule_name, category FROM
SQLSCRIPT_ANALYZER_RULES;
  to_scan = SELECT schema_name, procedure_name object_name, definition
            FROM sys.procedures
            WHERE procedure_type = 'SQLSCRIPT2' AND schema_name
IN('MY_SCHEMA','OTHER_SCHEMA')
            ORDER BY  procedure_name;
  CALL analyze_sqlscript_objects(:to_scan, :tab, objects, findings);
  SELECT t1.schema_name, t1.object_name, t2.*, t1.object_definition
  FROM :findings t2
  JOIN :objects t1
  ON t1.object_definition_id = t2.object_definition_id;
END;
```

Manual Rule Suppression

Due to the nature of static code analysis, the SQLScript Code Analyzer may produce false positives. To avoid confusion when analyzing large procedures with many findings, and potentially many false positives, the Code Analyzer offers a way to manually suppress these false positives.

You can use SQLScript Pragmas to define which rules should be suppressed. The pragma name is `AnalyzerSuppress` and it must at least one argument describing which rule should be suppressed.

Sample Code

```
create procedure proc as begin
  @AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY')
  declare a int;
end
```

Related Information

Limitations in the SQLScript Code Analyzer [page 285]

14.8.1 Limitations in the SQLScript Code Analyzer

Limited Support for Continue Handler

The Code Analyzer has limited support for Continue Handler. The Continue Handler blocks are currently not analyzed as a normal part of a procedure. Consider the following example:

Sample Code

```
create procedure wrong_proc(in tablename nvarchar(50)) as begin
  declare fallbackquery nvarchar(100) = 'select * from "' ||
escape_double_quotes(tablename) || '" where a > 5';
  declare continue handler for sqlexception exec :fallbackquery;
  -- do some computations
  select 1/0 from dummy;
end
```

The Code Analyzer will return a finding that the parameter `'tablename'` is used within DSQL, although the example is safe against injections.

If you look into the following example, you will see that the the handler block is analyzed on its own:

Sample Code

```
create procedure proc(in tablename nvarchar(50)) as begin
  declare continue handler for sqlexception
  begin
    declare fallbackquery nvarchar(100) = 'select * from "' ||
escape_double_quotes(tablename) || '" where a > 5';
    exec :fallbackquery;
  end;
  --do some computations
  select 1/0 from dummy;
end
```

In this case the Code Analyzer will not return a finding because the injection handling is performed in the handler block itself.

Library Variables Not Supported

Sample Code

```
create library libraryZ language sqlscript as begin
  public variable var2 varchar(10);
  public procedure callee_internal(in query1 varchar(20)) as begin
    var2 = 'i am not used';
    var2 = :query1 || :query1;
    select var2 from dummy;
  end;
end
```

In this case it is expected that the Code Analyzer will return a finding stating that that the value of `'var2'` is not used. However, currently most checks related to library member variables are not supported, including the following scenario:

Sample Code

```
create library libraryY language sqlscript as begin
  public variable var2 varchar(10);
  public procedure callee_internal(in query1 varchar(20)) as begin
    var2 = :query1;
    exec var2;
  end;
end
```

In this case the Code Analyzer does not return a warning stating that `'query1'` is used in dynamic SQL without being checked.

Limitations of UNCHECKED_SQL_INJECTION_SAFETY

The following issues are limited only to the UNCHECKED_SQL_INJECTION_SAFETY rule:

1. Pure SQL queries are not analyzed. This means that expressions inside those queries are not taken into consideration, for example validators for SQL injection.

   ### Sample Code

   Validator in pure SQL

   ```
   create procedure safe_dynamic_sql(in query nvarchar(500)) as begin
     declare escaped_query nvarchar(550);
     select escape_single_quotes(:query) into escaped_query from dummy;
     exec escaped_query;
   end
   ```

   The example above returns a finding even though the procedure is injection safe.
   If a SQLScript variable is used within a query, the Code Analyzer assumes that it is contained in the result.

   ### Sample Code

   SQLScript variable as input for pure SQL

   ```
   create procedure safe_dynamic_sql(in query nvarchar(500)) as begin
     declare some_value nvarchar(550);
     select b into some_value from some_tabe where :query = a;
     exec some_value;
   end
   ```

   In the example above `'query'` is not contained in `'some_value'` but is considered unsafe. There is no further analysis whether the output of the query possibly contains (parts of) the SQLScript variable inputs.
2. Nested procedure calls are also not analyzed.

   ### Sample Code

   Nested Procedure Call Example

   ```
   create procedure escape_proc(in query nvarchar(500), out escaped_query
   nvarchar(600)) as begin
       escaped_query = escape_single_quotes(query);
   end

   create procedure safe_dynamic_sql(in query nvarchar(500)) as begin
     declare escaped_value nvarchar(550);
     call escape_proc(query, escaped_value);
     exec escaped_value;
   end
   ```

   In example above, the Code Analyzer also returns a finding because it does not analyze the inner procedure `'escape_proc'`.
3. There are also limitations for structured types, like array variables, row variables or table variables.
   A variable of structured type is considered one unit. It is either affected by an unchecked input completely, or not at all.

   ### Sample Code

   Container Example

   ```
   create procedure row_type_injection(in query nvarchar(500)) as begin
     declare r row(a nvarchar(500), b nvarchar(650));
     r.a = query;
   ```

```
  r.b = escape_double_quotes(query);
  exec :r.b;
end
```

In the example above, the Code Analyzer will return a finding because the row variable `'r'` is considered one unit. Because the in parameter `'query'` is assigned directly (without escaping) to `'r.a'`, the variable `'r'` as a whole is considered affected by the input variable. Thus every operation that uses any part of `'r'` is assumed to use the unescaped version of `'query'`.

Related Information

SQLScript Code Analyzer [page 279]

14.9 SQLScript Plan Profiler

SQLScript Plan Profiler is a new performance analysis tool designed mainly for the purposes of stored procedures and functions. When SQLScript Plan Profiler is enabled, a single tabular result per call statement is generated. The result table contains start time, end time, CPU time, wait time, thread ID, and some additional details for each predefined operation. The predefined operations can be anything that is considered of importance for analyzing the engine performance of stored procedures and functions, covering both compilation and execution time. The tabular results are displayed in the new monitoring view M_SQLSCRIPT_PLAN_PROFILER_RESULTS in HANA.

> **i Note**
>
> Currently, only stored procedures are supported.

Starting the Profiler

There are two ways to start the profiler and to check the results.

ALTER SYSTEM

You can use the ALTER SYSTEM command with the following syntax:

Code Syntax

```
ALTER SYSTEM <command> SQLSCRIPT PLAN PROFILER [<filter>]
<command> := START | STOP | CLEAR
<filter>  := FOR SESSION <session_id> | FOR PROCEDURE <procedure_name>
```

i Note

You cannot filter by both session ID and procedure name.

The commands behave as follows:

- START
When the START command is executed, the profiler checks if the exact same filter has already been applied and if so, the command is ignored. You can check the status of enabled profilers in the monitoring view M_SQLSCRIPT_PLAN_PROFILERS. Results are available only after the procedure execution has finished. If you apply a filter by procedure name, only the outermost procedure calls are returned.

Sample Code

```
a) ALTER SYSTEM START SQLSCRIPT PLAN PROFILER FOR SESSION 111111;
b) ALTER SYSTEM START SQLSCRIPT PLAN PROFILER FOR SESSION 222222;
c) ALTER SYSTEM START SQLSCRIPT PLAN PROFILER FOR SESSION 222222; --
ignored because the profiler has already been started for session 222222
d) ALTER SYSTEM START SQLSCRIPT PLAN PROFILER FOR PROCEDURE P1;
e) ALTER SYSTEM START SQLSCRIPT PLAN PROFILER FOR PROCEDURE S1.P1; -- not
ignored, the filter is not the same (P1 != S1.P1)
f) ALTER SYSTEM START SQLSCRIPT PLAN PROFILER ; -- every procedures will
be profiled
```

- STOP
When the STOP command is executed, the profiler disables all started commands, if they are included in the filter condition (no exact filter match is needed). The STOP command does not affect the results that are already profiled.

```
<continued from the example above>
g) ALTER SYSTEM STOP SQLSCRIPT PLAN PROFILER FOR SESSION 222222; -- only b)
will be disabled
h) ALTER SYSTEM STOP SQLSCRIPT PLAN PROFILER FOR PROCEDURE P1; -- both d) and
e) will be disabled
i) ALTER SYSTEM STOP SQLSCRIPT PLAN PROFILER; -- both a) and f) will be
disabled
```

- CLEAR
The CLEAR command is independent of the status of profilers (running or stopped). The CLEAR command clears profiled results based on the PROCEDURE_CONNECTION_ID, PROCEDURE_SCHEMA_NAME, and PROCEDURE_NAME in M_SQLSCRIPT_PLAN_PROFILER_RESULTS. If the results are not cleared, the oldest data will be automatically deleted when the maximum capacity is reached.

```
j) ALTER SYSTEM CLEAR SQLSCRIPT PLAN PROFILER FOR SESSION 222222; -- deletes
records with PROCEDURE_CONNECTION_ID = 222222
k) ALTER SYSTEM CLEAR SQLSCRIPT PLAN PROFILER FOR PROCEDURE S1.P1; -- delete
records with PROCEDURE_SCHEMA_NAME = S1 and PROCEDURE_NAME = P1
l) ALTER SYSTEM CLEAR SQLSCRIPT PLAN PROFILER; -- deletes all records
```

i Note

The `<filter>` does not check the validity or existence of `<session id>` or `<procedure_id>`.

SQL Hint

You can use the SQL HINT command to start the profiler with the following syntax:

Code Syntax

```
CALL <procedure name> WITH HINT(SQLSCRIPT_PLAN_PROFILER);
```

SQL Hint is the most convenient way to enable the profiler. In that way, the profiling result is returned as an additional result set. If the profiler has already been enabled by means of the ALTER SYSTEM command, the result will be also visible in the monitoring view.

Currently both hint and system commands can be used to enable the SQLScript Plan Profiler for anonymous blocks.

Sample Code

Example using SQL Hint

```
DO BEGIN
    select * from dummy;
END WITH HINT(SQLSCRIPT_PLAN_PROFILER); -- returns additional result set
```

Sample Code

Example using system command

```
ALTER SYSTEM START SQLSCRIPT PLAN PROFILER FOR SESSION <SESSION_ID>;
DO BEGIN
    select * from dummy;
END; -- profiling result can be checked in m_sqlscript_plan_profiler_results
```

Checking Status and Results

You can check the status of the profiler by using the following command:

```
SELECT * FROM M_SQLSCRIPT_PLAN_PROFILERS;
```

You can check the results by using the following command:

```
SELECT * FROM M_SQLSCRIPT_PLAN_PROFILER_RESULTS;
```

Sample Code

Example

```
ALTER SYSTEM START SQLSCRIPT PLAN PROFILER;
CALL P1;
CALL P2;
SELECT * FROM M_SQLSCRIPT_PLAN_PROFILER_RESULTS WHERE PROCEDURE_NAME = 'P1'
OR PROCEDURE_NAME = 'P2';
```

14.9.1 M_SQLSCRIPT_PLAN_PROFILER_RESULTS View

The M_SQLSCRIPT_PLAN_PROFILER_RESULTS view contains the following columns:

Name	Data Type	Description
PROCEDURE_DATABASE_ID	INTEGER	Connection ID of the outermost procedure
PROCEDURE_DATABASE_NAME	NVARCHAR(256)	Database name of outermost procedure
PROCEDURE_SCHEMA_NAME	NVARCHAR(256)	Schema name of outermost procedure
PROCEDURE_LIBRARY_NAME	NVARCHAR(256)	Library name of outermost procedure
PROCEDURE_NAME	NVARCHAR(256)	Name of outermost procedure
RESULT_ID	INTEGER	Profile result ID
OPERATOR	VARCHAR(5000)	Name of operation
OPERATOR_STRING	NCLOB	Operator string
OPERATOR_DETAILS	NCLOB	Operation details
START_TIME	TIMESTAMP	Start time of the operation
END_TIME	TIMESTAMP	End time of the operation
DURATION	BIGINT	Clock time in microseconds between START_TIME and END_TIME
ACTIVE_TIME_SELF	BIGINT	Clock time in microseconds spent in the operation itself, excluding its children
ACTIVE_TIME_CUMULATIVE	BIGINT	Total clock time in microseconds spent in the operation itself and its children
CPU_TIME_SELF	BIGINT	CPU time in microseconds spent in the operation itself, excluding its children
CPU_TIME_CUMULATIVE	BIGINT	Total CPU time in microseconds spent in the operation itself and its children
CONNECTION_ID	INTEGER	Connection ID used for the operation
TRANSACTION_ID	INTEGER	Transaction ID used for the operation
STATEMENT_ID	VARCHAR(20)	Statement ID used for the operation
THREAD_ID	BIGINT	Thread ID used for the operation
OPERATOR_DATABASE_NAME	NVARCHAR(256)	Database name of the procedure or function where operator is defined
OPERATOR_SCHEMA_NAME	NVARCHAR(256)	Schema name of the procedure or function where operator is defined
OPERATOR_LIBRARY_NAME	NVARCHAR(256)	Library name of procedure/function where operator is defined

Name	Data Type	Description
OPERATOR_PROCEDURE_NAME	NVARCHAR(256)	Name of procedure/function where operator is defined
OPERATOR_LINE	INTEGER	SQL line of operator
OPERATOR_COLUMN	INTEGER	SQL column of operator
OPERATOR_POSITION	INTEGER	SQL position of operator
OPERATOR_HOST	VARCHAR(64)	Host where the operation occurred
OPERATOR_PORT	INTEGER	Port where the operation occurred
OPERATOR_ID	INTEGER	ID of operation (cannot be joined to any other views having the same name)
PARENT_OPERATOR_ID	INTEGER	ID of parent operation (cannot be joined to any other views having the same name)
PROCEDURE_HOST	VARCHAR(64)	Name of the host where the outermost procedure started
PROCEDURE_PORT	INTEGER	Port where the outermost procedure has started
USED_MEMORY_SIZE_SELF	BIGINT	Memory used in the operation itself, excluding its children (in bytes)
USED_MEMORY_SIZE_CUMULATIVE	BIGINT	Total memory used in the operation itself and its children (in bytes)

Memory Usage

Description

The following columns are used to track the memory usage of each operator (similarly to CPU times and ACTIVE times):

- USED_MEMORY_SIZE_SELF: Memory used in the operation itself, excluding its children (in bytes)
- USED_MEMORY_SIZE_CUMULATIVE: Total memory used in the operation itself and its children (in bytes)

Those columns show the memory usage of each SQL statement, such as STATEMENT_EXECUTION_MEMORY_SIZE and STATEMENT_MATERIALIZATION_MEMORY_SIZE in M_ACTIVE_PROCEDURES. For entries whose memory consumption is not collected or not calculated, the value displayed is '-1'.

The following two configurations must be enabled to activate the resource tracking:

```
alter system alter configuration ('global.ini', 'system') set
('resource_tracking', 'enable_tracking') = 'true' with reconfigure;
```

```
alter system alter configuration ('global.ini', 'system') set
('resource_tracking', 'memory_tracking') = 'true' with reconfigure;
```

Example

Sample Code

```
do begin
       v1 = select * from small_table with hint(no_inline);
       v2 = select * from big_table with hint(no_inline);
       select * from :v1 union all select * from :v2;
end with hint(sqlscript_plan_profiler);
```

Simplified result in M_SQLSCRIPT_PLAN_PROFILER_RESULTS:

OPERATOR	OPERATOR_STRING	OPERATOR_DETAILS	USED_MEMORY_SELF	USED_MEMORY_CUMULATIVE
Do			-1	4084734
Execute SePlan			-1	4084734
Sequential Op			-1	4084734
Initial Op			-1	-1
Parallel Op			-1	4084734
Parallel Evaluation			-1	4084734 (<a> + <b> + <c> + <d> + <e>)
Table Assign Op	select * from big_table with hint(no_inline)		-1	4035899
Execute SQL Statement		..., statement execution memory: <a>, itab size: <b>	4035899 (<a> + <b>)	4035899
Table Assign Op	select * from small_table with hint(no_inline)		-1	16067
Execute SQL Statement		..., statement execution memory: <c>, itab size: <d>	16067 (<c> + <d>)	16067
Select Op	select * from $$_SS_SE_TAB_VAR_V1_2$$ "V1" union all ...		-1	32768
Execute SQL Statement		..., statement execution memory: <e>	32768 (<e>)	32768
Flow Control Op			-1	-1
Terminal Op			-1	-1

Nested Calls

Description

The following columns provide more detailed information about nested calls:

- OPERATOR_DATABASE_NAME
- OPERATOR_SCHEMA_NAME
- OPERATOR_LIBRARY_NAME
- OPERATOR_PROCEDURE_NAME
- OPERATOR_LINE
- OPERATOR_COLUMN
- OPERATOR_POSITION

Example

Sample Code

The example illustrates the content of the columns above.

```
create or replace procedure p2(out o table(a int))
as begin
       insert into t1 values (2);
       o = select * from t1;
end;
create or replace procedure p1
as begin
       call p2(v) with hint(no_inline);
       select * from :v;
end;
call p1 with hint(sqlscript_plan_profiler);
```

The table below shows a simplified result output.

PROCEDURE_SCHEMA_NAME	PROCEDURE_NAME	OPERATOR	OPERATOR_STRING	OPERATOR_SCHEMA_NAME	OPERATOR_PROCEDURE_NAME	OPERATOR_LINE	OPERATOR_COLUMN	OPERATOR_POSITION
SYSTEM	P1	Call	call p1					
SYSTEM	P1	Compile						
SYSTEM	P1	Execute Se-Plan						
SYSTEM	P1	Initial Op						
SYSTEM	P1	Call Op	call p2(v) with hint(no_inline)	SYSTEM	P1	3	2	32
SYSTEM	P1	Compile						

PROCEDURE_SCHEMA_NAME	PROCEDURE_NAME	OPERATOR	OPERATOR_STRING	OPERATOR_SCHEMA_NAME	OPERATOR_PROCEDURE_NAME	OPERATOR_LINE	OPERATOR_COLUMN	OPERATOR_POSITION
SYSTEM	P1	Execute Se-Plan						
SYSTEM	P1	Initial Op						
SYSTEM	P1	DML Op	insert into t1 values (2)	SYSTEM	P2	3	2	52
SYSTEM	P1	Table Assign Op	select * from t1	SYSTEM	P2	4	2	81
SYSTEM	P1	Terminal Op						
SYSTEM	P1	Get Element Op		SYSTEM	P1	3	10	40
SYSTEM	P1	Select Op	select * from :v	SYSTEM	P1	4	2	67
SYSTEM	P1	Terminal Op						

14.10 SQLScript Pragmas

With pragmas SQLScript offers a new way for providing meta information. Pragmas can be used to annotate SQLScript code, but they do not have a function themselves and only affect other statements and declarations. Pragmas are clearly distinct syntax elements similar to comments, but while comments provide information to the reader of the code, pragmas provide information to the compiler and the code analyzer.

Syntax

Code Syntax

Procedure Head

```
<parameter_clause> ::= <parameter_with_pragma> [{',' <parameter_with_pragma>}]

<parameter_with_pragma> ::= {<single_pragma>} <parameter>

<single_pragma> ::= '@' <identifier> '(' [<single_pragma_parameter_clause>]
')'

<single_pragma_parameter_clause> ::= <string_literal> [{',' <string_literal>}]

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

Code Syntax

Procedure Body

```
<proc_decl_list> ::= <proc_decl_or_pragma> [{, <proc_decl_or_pragma>}]

<proc_decl_or_pragma> ::= <proc_decl> | <single_pragma> | <pragma_scope>

<pragma_scope> ::= '@' PUSHSCOPE '(' <single_pragma> [{',' <single_pragma>}]
')'
                 | '@' POPSCOPE '(' ')'

<proc_stmt_list> ::= {<proc_stmt_or_pragma>}

<proc_stmt_or_pragma> ::= <proc_stmt> | <single_pragma> | <pragma_scope>
```

i Note

The keywords **pushscope** and **popscope** are not case sensitive. PuShScopE is equal to pushscope and PUSHSCOPE.

Semantics

While the exact semantics depend on the specific pragma type, there are rules that apply to pragmas in general. The identifier is case insensitive, which means that **pragma** and **PrAgMa** are recognized as the same pragma. However, pragma arguments are case sensitive.

Pragma scopes affect all declarations or statements between one `pushscope` and the next `popscope` with all the pragmas that are specified in the pushscope.

Sample Code

```
do begin
  @pushscope(@AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY'))
  declare a int;
  declare b nvarchar(500);
  @popscope()
  declare c date;
  select :c from dummy;
end
```

In the example above the declarations for **a** and **b** will be affected by the pragma **'AnalyzerSuppress'**, while the declaration for **c** and the SELECT statement, are not affected.

Pragma scopes are independent of the logical structure of the code. This means that irrespective of which parts of the code are executed, the pragma scopes always affect the same statements and declarations.

Sample Code

```
create procedure proc(in a int, in b int) as
begin

@pushscope(@AnalyzerSuppress('SAP.USE_OF_UNASSIGNED_SCALAR_VARIABLE.CONSISTENC
Y'))
  if a < b then
```

```
    declare c date;
    select :c from dummy;
    @popscope()
  end if;
  a = :b; -- line 9
end
```

In this example, the assignment on line 9 will never be affected by the pragma. The SELECT statement, on the other hand, will always be affected by the pragma.

When using both pushscopes and single pragmas before declarations or statements, all pushscopes must precede the first single pragma. It is not allowed to mix pushscopes and single pragmas arbitrarily. For more information, see the examples in the section Limitations.

Single pragmas affect the next statement or declaration. This includes everything that is contained by the statement or declaration.

Sample Code

```
do begin
  @AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY')
  declare a, b, c int;
  @AnalyzerSuppress('SAP.USE_OF_UNASSIGNED_SCALAR_VARIABLE.CONSISTENCY')
  a = :b + 1;
end
```

In this example the single pragma on line 2 will affect the declarations of the three variables **a**, **b** and **c**. The single pragma on line 4 will affect the assignment and all parts of it. This also includes the expression **:b + 1** on the right hand side.

There is an exception for statements that contain blocks, that is basic blocks, loops and conditionals. The pragmas that are attached to a basic block, a loop or a conditional will not affect the declarations and statements within those blocks.

Sample Code

```
do begin
  @AnalyzerSupress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY')
  begin
    declare a nvarchar(50);
    select * from dummy;
  end;
end
```

In this example neither the declaration of **a**, nor the SELECT statement are affected by the pragma. Since such blocks belong to the normal SQLScript code, you can add a pragma or pragma scopes directly.

Available Pragmas

AnalyzerSuppress('NAME_SPACE.RULE_NAME.CATEGORY', ...)

Limitations

Single pragmas may not be followed directly by pragma scopes.

Sample Code

```
do begin /* NOT allowed*/
  @AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY')
  @pushScope(@AnalyzerSuppress('SAP.UNUSED_VARIABLE_VALUE.CONSISTENCY'))
  declare a, b int = 5;
  @popscope()
end

do begin /* NOT allowed*/
  @AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY')
  @pushScope(@AnalyzerSuppress('SAP.UNUSED_VARIABLE_VALUE.CONSISTENCY'))
  @someOtherPragma()
  declare a, b int = 5;
  @popscope()
end

do begin /*allowed*/
  @pushScope(@AnalyzerSuppress('SAP.UNUSED_VARIABLE_VALUE.CONSISTENCY'))
  @AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY')
  declare a, b int = 5;
  @popscope()
end

do begin /*allowed*/
  @pushScope(@AnalyzerSuppress('SAP.UNUSED_VARIABLE_VALUE.CONSISTENCY'))
  declare a int;
  @AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY')
  declare b int = 5;
  @popscope()
end

do begin /*allowed*/
  @pushScope(@AnalyzerSuppress('SAP.UNUSED_VARIABLE_VALUE.CONSISTENCY'))
  declare a int;
  @AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY')
  @someOtherPragma()
  declare b int = 5;
  @popscope()
end
```

It is not allowed to use pragma scopes within the parameter declaration list and in the declaration list before the initial begin of a procedure.

Sample Code

```
-- not allowed
create procedure
wrong_proc(@pushscope(@AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY'
)) in a int, in b nvarchar @popscope())
as begin
  select * from dummy;
end

-- not allowed
create procedure wrong_proc as
   @pushscope(@AnalyzerSuppress('SAP.UNNECESSARY_VARIABLE.CONSISTENCY'))
   a int;
   b nvarchar;
```

```
    @popscope()
begin
  select * from dummy;
end
```

Related Information

SQLScript Code Analyzer [page 279]

14.11 End-User Test Framework in SQLScript

The already existing mechanism of using libraries in SQLScript is re-used for the purposes of writing end-user tests. The language type SQLSCRIPT TEST has been introduced to specify that a library contains end-user tests. Currently, this language type can be only used for libraries.

> **i Note**
>
> To ensure a clear separation between productive and test-only coding, libraries of that language type cannot be used in any function, procedure or library that does not utilize the language type SQLSCRIPT TEST.

```
CREATE LIBRARY LIB_TEST LANGUAGE SQLSCRIPT TEST AS BEGIN <body> END;
```

Within the body of such a test library, you can use some of the SQLScript pragmas to mark a library member procedure as a test or test-related coding: `@Test()`, `@TestSetup()`, `@TestTeardown()`, `@TestSetupConfig('ConfigName')`, `@TestTeardownConfig('ConfigName')`, `@TestSetupLibrary()` as well as `@TestTearDownLibrary()`. Those pragmas are supported only for library member procedures and the procedures may not have any parameters.

> **i Note**
>
> All of these pragmas are optional and not required by default within an SQLSCRIPT TEST library. But to enable a library member procedure to be invoked as end-user test by the SQLScript Test Framework, at least the `@Test()` pragma is required.
>
> **Sample Code**
>
> ```
> CREATE LIBRARY LIB_TEST LANGUAGE SQLSCRIPT TEST AS
> BEGIN
>     @TestSetUpLibrary()
>     public procedure SetUpLibrary() as
>     begin
>         select 'SetUpLibrary' from dummy;
>     end;
>     @TestTearDownLibrary()
>     public procedure TearDownLibrary() as
>     begin
>         select 'whatever' from dummy;
> ```

```
    end;

     @TestClassification('FAST','base')
    @TestSetUpConfig('config1')
    public procedure SetUpConfig1() as
    begin
        truncate table tab_test;
        insert into tab_test values(1, 'first entry');
        insert into tab_test values(2, 'second entry');
        insert into tab_test values(3, 'third entry');
    end;
    @TestSetUpConfig('config2')
    public procedure SetUpConfig2() as
    begin
        truncate table tab_test;
        insert into tab_test values(5, 'fifth entry');
        insert into tab_test values(6, 'sixth entry');
        insert into tab_test values(7, 'seventh entry');
    end;
    @TestSetUpConfig('config3')
    public procedure SetUpConfig3() as
    begin
        truncate table tab_test;
        insert into tab_test values(5, 'some pattern string');
    end;
    @TestTearDownConfig('config1', 'config2', 'config3')
    public procedure TearDownConfig() as
    begin
        truncate table tab_test;
    end;
    @TestSetUpTest()
    public procedure SetUpTest() as
    begin
        using sqlscript_test as testing;
        declare num_entries int = record_count(tab_test);
        testing:expect_ne(0, num_entries);
    end;
    @TestTearDownTest()
    public procedure TearDownTest() as
    begin
        select 'whatever' from dummy;
    end;

     @TestClassification('SLOW')
    @Test()
    public procedure TestA as
    begin
        using sqlscript_test as testing;
        tab1 = select 'A1' as A from dummy;
        tab2 = select 'A2' as A from dummy;
        testing:expect_table_eq(:tab1, :tab2);
    end;
    @Test()
    public procedure TestC as
    begin
        using sqlscript_test as testing;
        declare str nclob;
        call proc_test(:str);
        testing:expect_eq('some replaced string', :str);
    end;
END;
```

To run the example SQLSCRIPT TEST library above, you would also need an object to be tested, for example the following procedure:

Sample Code

```
CREATE TABLE TAB_TEST(A INT, B NCLOB);
CREATE PROCEDURE PROC_TEST(OUT result VARCHAR(20)) AS
BEGIN
    DECLARE str STRING;
    SELECT B INTO str FROM TAB_TEST WHERE A = 5;
    IF LOCATE(:str, 'pattern') <> 0 THEN
        result = REPLACE(:str, 'pattern', 'replaced');
    ELSE
        result = :str;
    END IF;
END;
```

When invoking end-user tests, the SQLScript Test Framework considers member procedures of the SQLSCRIPT TEST library, marked with one of the pragmas mentioned above. It is, however, still possible to have additional member functions or procedures in such a library without any pragmas. These could then serve as helpers or be used to separate common coding.

The order of execution of library member procedures having these pragmas is defined as follows:

```
1.    @TestSetupLibrary()
2.      @TestSetupConfig('Config1')
3.        @TestSetup()
4.          @Test()
5.        @TestTeardown()
6.        @TestSetUp()
7.          @Test()
8.        @TestTeardown()
9.        [...]
10.     @TestTeardownConfig('Config1')
11.     @TestSetupConfig('Config2')
12.       @TestSetup()
13.         @Test()
14.       @TestTeardown()
15.       @TestSetUp()
16.         @Test()
17.       @TestTeardown()
18.       [...]
19.     @TestTeardownConfig('Config2')
20.     [...]
21.   @TestTeardownLibrary()
```

Note

In case the execution of a library member procedure having one of the `SetUp` pragmas fails, the corresponding `TearDown`, as well as the tests, will not be executed. With the `@TestClassification(…)` pragma, `SetUpLibrary`, `SetUpConfiguration` and `Test` procedures can be assigned additional tags that can be used in test filters.

Related Information

Invoking End-User Tests [page 302]
Listing End-User Tests [page 306]
Matchers for End-User Tests [page 307]

14.11.1 Invoking End-User Tests

The entry point of the end-user test framework in SQLScript is the built-in procedure SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA.

> **i Note**
>
> As the name of the procedure indicates, the tests are run on the existing data in the system. You need to pay special attention when writing tests that change or delete objects or data in the system because others may be influenced by these changes. Tests themselves may also be influenced by other tests running in parallel on the same system.

Users do not have the EXECUTE privilege for the built-in procedure SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA by default. You need to get this privilege granted (for example, by a SYSTEM user).

To invoke end-user tests in the SQLScript test framework, the following CALL statement has to be executed.

```
CALL SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA('<json_string>', ?, ?, ?)
```

The first parameter of SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA specifies the test plan to be executed and has to be provided in JSON format. The test plan specifies which tests and with what configuration shall be run. It also contains information about which test libraries are to be executed by the test framework.

> **i Note**
>
> Wildcards can be used to specify values in the JSON string ('*' for multiple wildcard characters, '?' for exactly one wildcard character).

```
CALL
SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA('{"schema":"MY_SCHEMA","library":"*"}',
?, ?, ?)
CALL
SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA('{"schema":"MY_SCHEMA","library":"LIB*TE
ST"}', ?, ?, ?)
CALL
SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA('[{"schema":"MY_SCHEMA","library":"SOME_
PREFIX_*"},{"schema":"OTHER_SCHEMA","library":"*_SOME_SUFFIX"}]', ?, ?, ?)
```

The first call to SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA will run all tests (in all their configurations respectively) of all libraries with language type SQLSCRIPT TEST in the schema MY_SCHEMA. The second call will do the same but applies a filter to the libraries that are to be executed. Here, only SQLSCRIPT TEST libraries having a name starting with 'LIB' and ending with 'TEST' will be executed by the test framework. For the third call, also libraries with language type SQLSCRIPT TEST in the schema OTHER_SCHEMA will be executed but their name has to end with '_SOME_SUFFIX'.

The complete definition of what can be provided in the JSON string of the test plan is described below.

```
<test_plan> ::= <lib_spec> || <lib_spec_list>

<lib_spec_list> ::= '[' <lib_spec> [',' <lib_spec>] ']'

<lib_spec> ::= '{' ["schema":"' <wildcard_pattern> '",] "library":"'
<wildcard_pattern> '"' [', "classifications":' <wildcard_pattern_list>] [',
```

```
"exclude-classifications":' <wildcard_pattern_list>] [',  "run":'
<run_spec_list>] '}'

<run_spec_list> ::= '[' <run_spec> [',' <run_spec>] ']'

<run_spec> ::= '{' <run_spec_member> [',' <run_spec_member] '}'

<run_spec_member> ::= [ '"tests":[' <wildcard_pattern_list> ']' ||
'"configurations":[' <wildcard_pattern_list> ']' || '"exclude-tests":['
<wildcard_pattern_list> ']' || '"exclude-configurations":['
<wildcard_pattern_list> ']' ]

<wildcard_pattern_list> ::= '"' <wildcard_pattern> '"' [', "' <wildcard_pattern>
'"']

<wildcard_pattern> ::= letter_or_digit_or_asterisk+
```

Note

`<wildcard_pattern>` is always case-sensitive.

Examples:

Sample Code

```
[{
    "schema":"MY_SCHEMA",
    "library":"*"
},
{
    "library": "MY_LIB",
    "run": [{
        "exclude-tests": ["A", "B"],
        "configurations": ["config1", "config3"]
    },
    {
        "tests": ["A", "B"],
        "exclude-configurations": ["config2"]
    }]
},
{
    "schema": "MY_SCHEMA",
    "library": "*",
    "run": [{
        "tests": ["*TEST*KERNEL*"],
        "exclude-tests": ["DISABLED_*"],
        "exclude-configurations": ["*SCALE_OUT*"]
    },
    {
        "configurations": ["*SINGLE_NODE*", "*SCALE_OUT*"],
        "exclude-configurations": ["*STRESS_TEST*"]
    }]
}]
```

Behavior

- Invalid syntax or semantics result in an error.
- Unknown properties produce a warning.

- The property `library` is mandatory but there are default values for other properties:
  - If "schema" is not specified, current session schema will be used.
  - If "run" is not specified, all configurations and tests will be selected. That is identical to "run": `[{ "tests": [ "*" ], "configurations": [ "*" ] }]`.
- When "tests" and "exclude-tests" match exactly the same values, an error will be thrown. The same applies to "configurations" and "exclude-configurations".
- When both "exclude-tests" and "tests" are given, "exclude-tests" will always have higher precedence. The same applies to "exclude-configurations" and "configurations".
- If a library or a configuration does not contain any tests (after applying the filter), neither the setup, nor the teardown of this library or configuration will be executed.
- An empty test plan will be generated if the input does not match any tests and no error will be thrown. Also no entries will be added to the output tables.

> **i Note**
>
> Each entry in <run_spec_list> will cause a separate list of tests and configurations to be added to the test plan depending on the values of the inner <run_spec_member> entries. In that way some tests as well as configurations of the same library may be executed repeatedly by the test framework.

Classifications and Exclude-Classifications

Classifications can be specified on multiple levels and the filtering based on classifications also needs to be performed on multiple levels.

For exclude-classifications this means the following:

- If a classification specifier of a library member (the classification specified with the pragma) matches a pattern in the exclude-specification, this member and everything it includes will not be executed. For example, if a `SetUpLibrary` matches an exclude-classification, nothing in this library will be executed. For a `config` it means that no test will be executed in this `config`. And for a test it just means that this test is not executed.
- If the classifications specifier does not match the exclude-specification, the library, the configuration or the test is executed.

For classifications this means the following:

- If a classifications specifier of a library member matches a pattern in the specification this member and everything it includes, it will be executed unless an exclude specification matches.
- If the classification specifier does not match the specification, only the members included that match the specification will be executed.
- If tests do not match, they will not be executed.

Consider the following example:

> **Sample Code**
>
> ```
> CREATE LIBRARY LIB_TEST LANGUAGE SQLSCRIPT TEST AS BEGIN
>     @TestClassification('clas0')
>     @TestSetUpLibrary()
>     PUBLIC PROCEDURE SETUPLIB AS BEGIN END;
> ```

```
    @TestClassification('clas1')
    @TestSetUpConfig('A')
    PUBLIC PROCEDURE SETUPCONFIGA AS BEGIN END;
    @TestSetUpConfig('B')
    PUBLIC PROCEDURE SETUPCONFIGB AS BEGIN END;
    @TestClassification('clas2')
    PUBLIC PROCEDURE TESTA AS BEGIN END;

    PUBLIC PROCEDURE TESTB AS BEGIN END;
END
```

If classification 'clas0' is included, everything will be executed. If classification 'clas1' is included, everything in configuration 'A' will be executed. If classification 'clas2' is included, only 'TESTA' will be executed but in both configurations - 'A' and 'B'.

If classification 'clas0' is included and 'clas1' excluded, only the configuration 'B' will be executed (with both tests). If classification 'clas0' is included and 'clas2' is excluded, only 'TESTB' will be executed but in both configurations - 'A' and 'B'. If classification 'clas1' is included and 'clas2' excluded, only 'TESTB' in configuration 'A' will be executed.

If classification 'clas2' is included and 'clas0' excluded, nothing will be executed. If classification 'clas2' is included and 'clas1' excluded, only 'TESTA' will be executed and only in configuration 'B'. If classification 'clas1' is included and 'clas0' excluded, nothing will be executed.

Output

The three output parameters of SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA have the following table structures.

Results

Column Name	Type	Description
SCHEMA_NAME	NVARCHAR(256)	Schema name
LIBRARY_NAME	NVARCHAR(256)	Library name
CONFIGURATION_NAME	NVARCHAR(256)	Configuration name
TEST_NAME	NVARCHAR(256)	Test name
TEST_EXECUTION_ID	BIGINT	Unique identifier for look-up in details output table
TEST_EXECUTION_TIME	BIGINT	Duration in microseconds
TEST_EXECUTION_MEMORY_SIZE	BIGINT	Memory size used during test execution (cf. M_ACTIVE_PROCEDURES)
TEST_EXECUTION_RESULT_STATE	VARCHAR(16)	Test result PASSED | FAILED | ERROR | SKIPPED | CANCELLED
TEST_COMMENTS	NVARCHAR(5000)	User-defined comment defined for corresponding member in test library

Details

Column Name	Type	Description
TEST_EXECUTION_ID	BIGINT	Identifier for particular test run
RESULT_DETAIL_ID	BIGINT	Unique identifier for look-up in call stacks output table
RESULT_DETAIL	NCLOB	Long text describing what failed / which error occurred during test run

Call Stacks

Column name	Type	Description
RESULT_DETAIL_ID	BIGINT	Identifier for a particular call stack
FRAME_LEVEL	INTEGER	Level of the call stack frame
DATABASE_NAME	NVARCHAR(256)	Database name
SCHEMA_NAME	NVARCHAR(256)	Schema name
OBJECT_NAME	NVARCHAR(256)	Object name
MEMBER_NAME	NVARCHAR(256)	Library member name
LINE	INTEGER	SQL line number
COLUMN	INTEGER	SQL column value
POSITION	INTEGER	SQL position value

14.11.2 Listing End-User Tests

For checking which tests and configurations will be invoked by the test framework when providing a JSON string as test plan description, the built-in library SYS.SQLSCRIPT_TEST contains two additional procedures. LIST_TESTS returns every test that would be executed at least once. LIST_CONFIGURATIONS returns every configuration that would execute at least one test. The result set will not contain any duplicates.

```
CALL SYS.SQLSCRIPT_TEST:LIST_TESTS('<json_string>')
CALL SYS.SQLSCRIPT_TEST:LIST_CONFIGURATIONS('<json_string>')
```

Sample Code

Examples

```
CALL SYS.SQLSCRIPT_TEST:LIST_TESTS('{"schema":"MY_SCHEMA","library":"*"}', ?)
CALL
SYS.SQLSCRIPT_TEST:LIST_TESTS('{"schema":"MY_SCHEMA","library":"LIB*TEST"}', ?
)
CALL
SYS.SQLSCRIPT_TEST:LIST_CONFIGURATIONS('[{"schema":"MY_SCHEMA","library":"SOME
_PREFIX_*"},{"schema":"OTHER_SCHEMA","library":"*_SOME_SUFFIX"}]', ?)
```

Both will return the following tabular output.

Column Name	Type	Description
SCHEMA_NAME	NVARCHAR(256)	Schema name
LIBRARY_NAME	NVARCHAR(256)	Library name
CONFIGURATION/TEST_NAME	NVARCHAR(256)	Configuration/Test name
COMMENTS	NCLOB	Description

14.11.3 Matchers for End-User Tests

Within the SQLSCRIPT TEST libraries, certain procedures of the built-in library SYS.SQLSCRIPT_TEST can be used to verify results within end-user tests.

Currently, there are several matchers for scalar variables, one matcher for table variables and one that aborts the execution of the current test. The matchers for scalar variables are:

Matcher Name	Description
EXPECT_EQ	Checks if the inputs are equal
EXPECT_NE	Checks if the inputs are not equal
EXPECT_GE	Checks if the first input is greater than or equal to the second input
EXPECT_GT	Checks if the first input is greater than the second input
EXPECT_LE	Checks if the first input is less than or equal to the second input
EXPECT_LT	Checks if the first input is less than the second input
EXPECT_NULL	Checks if the input is null

All scalar matchers, except EXPECT_NULL, take exactly two scalar input arguments. The data types of these two inputs must be comparable in SQLScript. Most of the data types can be categorized in three classes: string types, numeric types and date types. While all types within the same class are comparable to each other, it is not possible to compare date and numeric types. String types can be compared to every other data type but will be converted to a non-string type prior to the comparison. Whenever two different data types are compared, at least one of the inputs will be converted. When the conversion fails, it is considered a normal execution error instead of reporting a matcher failure.

The table matcher (**EXPECT_TABLE_EQ**) has three input arguments. Besides the two table variables that should be compared, there is a third optional input - IGNORE_ORDER. This parameter is TRUE by default and will compare the table variables without considering the order of rows. For example row 2 of the first input might match row 5 of the second input. However, every row will be matched at most to one row in the other table variable. The two input table variables must have an equal number of columns and the columns must have same names. The data types of the columns have to be comparable as well. If the types of the table columns are different, one of the columns will be converted before the comparison. Unlike in scalar comparisons, this will not lead to a run-time error if such a conversion fails. Instead, the row will always be considered a mismatch. One additional difference to scalar matchers is the handling of NULL values. For scalar matchers, anything compared to NULL is false (even NULL). The table matcher assumes that NULL is equal to NULL.

In case a matcher fails, a human-readable output will be added to the *Details* output table of the built-in procedure SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA. A call stack is also generated for such a matcher failure to make it possible to determine its exact code location. The table matcher will report a maximum of 100 row mismatches for the sake of readability.

The built-in library SQLSCRIPT_TEST also contains a procedure named **FAIL**. This procedure will (similarly to a matcher) add an entry to the *Details* output table of SYS.SQLSCRIPT_RUN_TESTS_ON_ORIGINAL_DATA whereby the error message that was provided as an input argument to the procedure FAIL will be included as a message. After that, this procedure will abort the execution of the current test. The subsequent tests will still be executed.

15 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.
- 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 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.

15.1 Reduce the 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 requires either the definition of a temporary view (using WITH), or the multiple repetition of a sub-query. The two statements above break the complex query into two simpler SQL statements that are linked by table variables. This query is much easier to understand 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.

15.2 Identify Common Sub-Expressions

The query examined in the previous topic contained common sub-expressions. Such common sub-expressions might introduce expensive repeated computation that should be avoided.

It is very complicated for query optimizers to detect common sub-expressions in SQL queries. If you break up a complex query into logical subqueries 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 SAP HANA database handles a particular statement.

15.3 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.

For example:

```
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 must typically 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;
```

15.4 Reduce Dependencies

One of the most important methods for speeding up processing in the SAP HANA database is through massively parallelized query execution.

Parallelization is exploited at multiple levels of granularity. For example, the requests of different users can be processed in parallel, and single relational operators within a query can also be executed on multiple cores in parallel. It is also possible to execute different statements of a single SQLScript procedure 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.

As an SQLScript developer, you can support the database engine in its attempt to parallelize execution by avoiding unnecessary dependencies between separate SQL statements, and by using declarative constructs if possible. The former means avoiding variable references, and the latter means avoiding imperative features, such as cursors.

15.5 Avoid Using Cursors

While the use of cursors is sometime required, they also imply row-by-row processing. Consequently, opportunities for optimizations by the SQL engine are missed. You should therefore consider replacing cursors with loops in SQL statements.

Read-Only Access

For read-only access to a cursor, consider using simple selects or joins:

```
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 following:

```
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 the SAP HANA database. It also potentially benefits from internal optimizations like buffering and 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;
```

Like updates and deletes, computing this statement in the SQL engine reduces the calls through the runtime stack of the SAP HANA database. It also potentially benefits from internal optimizations like buffering and parallel execution.

15.6 Avoid Using Dynamic SQL

Dynamic SQL is a powerful way to express application logic. It allows SQL statements to be constructed at the execution time of a procedure. However, executing dynamic SQL is slow because compile-time checks and query optimization must be performed each time the procedure is called. 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, like an SQL injection, for example. Using variables in SQL statements prevents these problems because type checks are performed at compile time and parameters cannot inject arbitrary SQL code.

The table below summarizes potential use cases for dynamic SQL:

Dynamic SQL Use Cases

Feature	Proposed Solution
Projected attributes	Dynamic SQL
Projected literals	SQL + variables
`FROM` clause	SQL + variables; result structure must remain unchanged
`WHERE` clause – attribute names and Boolean operators	`APPLY_FILTER`

16 Developing Applications with SQLScript

This section contains information about creating applications with SQLScript for SAP HANA.

16.1 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.

16.2 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

Window Functions and the Window Specification

16.3 Calling SQLScript From Clients

The following chapters discuss 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. This section shows examples of how this can be done.

16.3.1 Calling SQLScript from ABAP

Using ABAP Managed Database Procedures (AMDP)

The best way to call SQLScript from ABAP is by means of the AMDP framework. That framework manages the lifecycle of SQLScript objects and embeds them as ABAP objects (classes). The development, maintenance, and transport is performed on the ABAP side. A call of an AMDP corresponds to a class method call in ABAP. The AMDP framework takes care of generating and calling the corresponding database objects.

For more information, see ABAP - Keyword Documentation → ABAP - Reference → Processing External Data → ABAP Database Accesses → AMDP - ABAP Managed Database Procedures.

Using CALL DATABASE PROCEDURE

> → Tip
>
> You can call SQLScript from ABAP by using a procedure proxy that can be natively called from ABAP by using the built-in command `CALL DATABASE PROCEDURE`. However, it is recommended to use AMDP.

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.

For more information, see ABAP - Keyword Documentation → ABAP - Reference → Processing External Data → ABAP Database Accesses → ABAP and SAP HANA → Access to Objects in SAP HANA XS → Access to SAP HANA XSC Objects → Database Procedure Proxies for SQLScript Procedures in XSC → CALL DATABASE PROCEDURE.

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
```

Related Information

CALL DATABASE PROCEDURE

AMDP - ABAP Managed Database Procedures

16.3.2 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();
}
```

16.3.3 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 test_pro1(?,?)";
                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);
            }
        }
    }
}
```

17 Appendix

17.1 Example code snippets

The examples used throughout this manual make use of various predefined code blocks. These code snippets are presented below.

17.1.1 ins_msg_proc

This code is used in the examples of this reference manual to store outputs, so that you can see the way the examples work. It simply stores text along with a time stamp 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

For information about the capabilities available for your license and installation scenario, refer to the Feature Scope Description for SAP HANA.

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.
About the icons:

- Links with the icon : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.
The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Bias-Free Language

SAP supports a culture of diversity and inclusion. Whenever possible, we use unbiased language in our documentation to refer to people of all cultures, ethnicities, genders, and abilities.

www.sap.com/contactsap

THE BEST RUN