PUBLIC

SAP HANA Platform SPS 09
Document Version: 1.1 – 2015-02-16

SAP HANA SQLScript Reference

Content

1 About SAP HANA SQLScript

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

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

2 Backus Naur Form Notation

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

Symbols used in BNF

Table 1:

Symbol	Description
< >	Angle brackets are used to surround the name of a syntactic element (BNF non-terminal) of the SQL language.
::=	The definition operator is used to provide definitions of the element appeared 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 specifying that 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 will be 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
<string_literal> ::= <double_quote><string_content><double_quote> |
<single_quote><string_content><single _quote>
<string_content> = { <letter> | <digit> }...
<identifier> ::= <letter> { <letter> | <digit> }...
<password> ::= [{ <letter> | <digit> }...]
<sign> ::= + | -
<period> ::= .
<unsigned_integer> ::= <digit>...
<signed_integer> ::= [<sign>] <unsigned_integer>
<signed_numeric_literal> ::= [<sign>] <unsigned_numeric_literal>
<unsigned_numeric_literal> ::= <exact_numeric_literal> |
<approximate_numeric_literal>
<exact_numeric_literal> ::= <unsigned_integer> [<period> [<unsigned_integer>]]
                          | <period> <unsigned_integer>
<approximate_numeric_literal> ::= <mantissa> E <exponent>
<mantissa> ::= <exact_numeric_literal>
```

```
<exponent> ::= <signed_integer>
```

3 What is SQLScript?

The motivation for SQLScript is to embed data-intensive application logic into the database. As of today, applications only offload very limited functionality into the database using SQL, most of the application logic is normally executed in an application server. This has the effect that data to be operated upon needs to be copied from the database into the application server and vice versa. When executing data intensive logic, this copying of data is 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 such as SQL.

The SAP HANA database is optimized for modern technology trends and takes advantage of modern hardware, for example, by having data residing in main-memory and allowing massive-parallelization on multi-core CPUs. The goal of the SAP HANA database is to optimally support application requirements by leveraging such hardware. To this end, 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 that allow developers to push data intensive logic into 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 for leveraging sophisticated parallel execution strategies of the database. SQLScript addresses the following problems:

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

Related Information

Exploit Underlying Engine [page 113]

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 dataand 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`, `EXECUTE IMMEDIATE`, and `APPLY_FILTER`. 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.

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 use dynamic SQL to create an escape from the procedure in the code.

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

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

Related Information

SAP HANA Security Guide
SAP HANA SQL and System Views Reference

3.2 SQLScript Processing Overview

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

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

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

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

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

With SQLScript one can implement applications both using imperative orchestration logic and (functional) declarative logic, and this is also reflected in the way SQLScript processing works for both coding styles. Imperative logic is executed sequentially and declarative logic is executed by exploiting the internal architecture of the SAP HANA database utilizing its potential for parallelism.

3.2.1 Orchestration-Logic

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

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

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

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

3.2.1.1 Example for Orchestration-Logic

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

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

Related Information

ins_msg_proc [page 124]

3.2.2 Declarative-Logic

Declarative logic is used for efficient execution of data-intensive computations. This logic is internally represented as data flows which can be executed in parallel. As a consequence, operations in a dataflow graph have to be free of side effects. This means they must not change any global state either in the database or in the application. The first condition is ensured by only allowing changes on the dataset that is passed as input to the operator. The second condition is achieved by only allowing a limited subset of language features to express the logic of the operator. Given these prerequisites, the following kinds of operators are available:

- SQL SELECT Statement
- Custom operators provided by SAP

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

4 Datatype Extension

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

4.1 Scalar Datatypes

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

Table 2:

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

> **Note**
>
> This is the same as for SQL statements, excluding the TEXT and SHORTTEXT types.

See SAP HANA SQL and System Views Reference, Data Types section, for further details on scalar types.

4.2 Table Types

SQLScript's datatype extension also allows the definition of table types. These table types are used to define parameters for a procedure that represent tabular results.

4.2.1 CREATE TYPE

Syntax

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

Syntax Elements

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

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

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

Defines a table column.

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

The available data types. For more information on data types, see Scalar Datatypes [page 13]

Description

The CREATE TYPE statement creates a user-defined type.

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

Example

You create a table type called tt_publishers.

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

You create a table type called tt_years.

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

4.2.2 DROP TYPE

Syntax

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

Syntax Elements

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

The identifier of the table type 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 drop only the specified type, dependent objects of the type will be invalidated but not dropped.

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

Description

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

Example

You create a table type called my_type.

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

You drop the my_type table type.

```
DROP TYPE my_type;
```

5 Logic Container

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

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

5.1 Procedures

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

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

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

- The calculation and transformations described in procedures can be parameterized and reused in other procedures.
- The user is able to 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.
- It is easy to define common sub-expressions. The query optimizer decides if a materialization strategy (which avoids recomputation of expressions) or other optimizing rewrites are best to apply. In any case, it eases the task to detect common sub-expressions and improves the readability of the SQLScript code.
- Scalar variables or imperative language features are also available and can be used if they are required.

5.1.1 CREATE PROCEDURE

You use this SQL statement to create a procedure.

Syntax

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

Syntax Elements

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

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

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

The input and output parameters of the procedure.

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

A procedure parameter with associated data type.

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

Default: IN

Each parameter is marked using the keywords IN/OUT/INOUT. Input and output parameters must be explicitly typed (i.e. no un-typed tables are supported).

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

The variable name for a parameter.

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

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 scalar type.

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

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

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

A table type previously defined with the CREATE TYPE command, see CREATE TYPE [page 14].

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

A table type implicitly defined within the signature.

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

Default: SQLSCRIPT

Defines the programming language used in the procedure. It is good practice to define the language in all procedure definitions.

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

Default: DEFINER

Specifies the security mode of the procedure.

```
DEFINER
```

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

```
INVOKER
```

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

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

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

```
READS SQL DATA
```

Marks the procedure as being read-only, side-effect free i.e. 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 the procedure only calls other read-only procedures. The advantage of using this parameter is that certain optimizations are available for read-only procedures.

```
 WITH RESULT VIEW <view_name>
 <view_name> ::= <identifier>
```

Specifies the result view to be used as the output of a read-only procedure.

When a result view is defined for a procedure, it can be called by an SQL statement in the same way as a table or view. See *Example 2 - Using a result view* below.

```
SEQUENTIAL EXECUTION
```

This statement will force sequential execution of the procedure logic. No parallelism takes place.

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

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

```
<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>
<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>
```

Condition handler declaration.

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

Declares exception handlers to catch SQL exceptions.

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

One or more condition values.

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

You can use a specific error code number or condition name declared on condition variable.

```
 <proc_stmt_list> ::= {<proc_stmt>}...
 <proc_stmt> ::= <proc_block>
               | <proc_assign>
               | <proc_single_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>
```

Procedure body statements.

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

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

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

Assign 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.

```
<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>
```

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 "SQLScript reference".

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

```
 <proc_ce_call> ::= TRACE ( <variable_name> ) ;
                  | CE_LEFT_OUTER_JOIN ( <table_variable> , <table_variable> ,
'[' <expr_alias_comma_list> ']' [ <expr_alias_vector>]  ) ;
                  | CE_RIGHT_OUTER_JOIN ( <table_variable> , <table_variable> ,
'[' <expr_alias_comma_list> ']' [ <expr_alias_vector>] ) ;
                  | CE_FULL_OUTER_JOIN ( <table_variable> , <table_variable> ,
'[' <expr_alias_comma_list> ']' [ <expr_alias_vector>]  );
                  | CE_JOIN ( <table_variable> , <table_variable> , '['
<expr_alias_comma_list> ']' [<expr_alias_vector>]  ) ;
                  | CE_UNION_ALL ( <table_variable> , <table_variable> ) ;
                  | CE_COLUMN_TABLE ( <table_name> [ <expr_alias_vector>]  ) ;
                  | CE_JOIN_VIEW ( <table_name> [ <expr_alias_vector>] ) ;
                  | CE_CALC_VIEW ( <table_name> [ <expr_alias_vector>] ) ;
                  | CE_OLAP_VIEW ( <table_name> [ <expr_alias_vector>] ) ;
                  | CE_PROJECTION ( <table_variable> , '['
<expr_alias_comma_list> ']' <opt_str_const> ) ;
                  | CE_PROJECTION ( <table_variable> <opt_str_const> ) ;
                  | CE_AGGREGATION ( <table_variable> , '['
<agg_alias_comma_list> ']' [ <expr_alias_vector>] );
                  | CE_CONVERSION ( <table_variable> , '['
<proc_key_value_pair_comma_list> ']' [ <expr_alias_vector>] ) ;
                  | CE_VERTICAL_UNION ( <table_variable> , '['
<expr_alias_comma_list> ']' <vertical_union_param_pair_list> ) ;
                  | CE_COMM2R ( <table_variable> , <int_const> , <str_const> ,
<int_const> , <int_const> , <str_const> ) ;
 <table_name>  ::= [<schema_name>.]<identifier>
 <schema_name> ::= <identifier>
```

For more information about the CE-Operators, see Calculation Engine Plan Operators [page 92].

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

APPLY_FILTER defines a dynamic WHERE condition <variable_name> that will be applied during runtime. For more information about APPLY_FILTER please see the "SQLScript reference".

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

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

```
WITH ORDINALTIY
```

Appends an ordinal column to the return values.

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

Specifies the column names of the return table.

```
 <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 IF - THEN - ELSE IF to control execution flow with conditionals.

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

You use loop to repeatedly execute a set of statements.

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

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

```
 <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 - IN loops to iterate over 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> }...] )
```

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

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

Terminates a loop.

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

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

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

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

```
<proc_resignal> ::= RESIGNAL [<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.

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

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

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

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

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

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.

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

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

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

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

Cursor operations.

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

You use EXEC to make dynamic SQL calls.

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

Return a value from a procedure.

Description

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

Examples

Example 1 - Creating an SQL 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.

Example 2 - Using a result view

You create a procedure using a result view ProcView to return its results.

```
CREATE PROCEDURE ProcWithResultView(IN id INT, OUT o1 CUSTOMER)
LANGUAGE SQLSCRIPT
READS SQL DATA WITH RESULT VIEW ProcView AS
BEGIN
  o1 = SELECT * FROM CUSTOMER WHERE CUST_ID = :id;
END;
```

You call this procedure from an SQL statement as follows.

```
SELECT * FROM ProcView PLACEHOLDER."$$id$$"=>'5');
```

> **Note**
>
> Procedures and result views produced by procedures are not connected from the security perspective and therefore do not inherit privileges from each other. The security aspects of each object must be handled separately. For example, you must grant the `SELECT` privilege on a result view and `EXECUTE` privilege on a connected procedure.

5.1.2 DROP PROCEDURE

Syntax

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

Syntax Elements

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

The name of the procedure 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 procedure, dependent objects of the procedure 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 procedure and dependent objects.

```
RESTRICT
```

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

Description

Drops a procedure created using CREATE PROCEDURE from the database catalog.

Examples

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

```
DROP PROCEDURE my_proc;
```

5.1.3 ALTER PROCEDURE RECOMPILE

Syntax

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

Syntax Elements

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

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

```
WITH PLAN
```

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

Description

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

Example

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

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

5.1.4 Procedure Calls

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

5.1.4.1 CALL

Syntax

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

Syntax Elements

```
<proc_name> ::= [<schema_name>.]<identifier>
<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 6] and Scalar Datatypes [page 13].

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

```
 WITH OVERVIEW
```

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

Calling a procedure WITH OVERVIEW will return 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 17].

CALL conceptually returns 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 as for query results. SQL statements that are not assigned to any table variable in the procedure body will be added as result sets at the end of the list of result sets. The type of the result structures will be determined during compilation time but will not be visible in the signature of the procedure.

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

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

Examples

For the examples, consider the following procedure signature:

```
CREATE PROCEDURE proc(
            IN value integer,IN currency nvarchar(10),OUT outTable typeTable,
```

```
            OUT valid integer)
AS
BEGIN
    …
END;
```

Calling the `proc` procedure:

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

Calling the `proc` procedure in debug mode:

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

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

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

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

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

In this example, udf() is a scalar user-defined function. For more information about scalar user-defined functions, see CREATE FUNCTION [page 39]

5.1.4.2 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>
```

Parmeters can be of table or scalar type.

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

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

Specifies a procedure input parameter.

> **Note**
>
> Please note the use of a colon in-front of the identifier name.

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

Specifies a procedure input parameter.

Description:

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

Example:

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

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

Related Information

CALL

5.1.4.3 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 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 will produce the same result.

5.1.5 Procedure Parameters

Parameter Modes

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

Table 3: Parameter modes

Mode	Description
IN	An input parameter
OUT	An output parameter
INOUT	Specifies a parameter that will both pass-in and return data to and from the procedure. **Note** This is only supported for Scalar values.

Supported Parameter Types

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

Related Information

Datatype Extension [page 13]

5.1.5.1 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)
```

> **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")
```

Do not use the following syntax:

```
CALL test_table ('tab')
```

5.1.5.2 Default values for procedure parameters

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

```
 CREATE PROCEDURE MYPROC(IN P1 INT, IN P2 INT DEFAULT 1, OUT out1 DUMMY) AS
BEGIN
  out1 = SELECT :P1 + :P2 AS DUMMY FROM DUMMY;
END;
```

You can see that the second parameter has a default value of `1`.

To use the default values in the procedure signature, you need to pass in procedure parameters using *Named Parameters*. For more information see Named Parameters.

You can call this procedure in the following ways:

With all input values specified:

```
CALL MYPROC(3, 4,?);
```

Using the default value via named parameters:

```
CALL MYPROC(P1 => 3, out1 => ?)
```

Related Information

Call with Named Parameters [page 30]

5.1.6 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 like tables, see the SQL Reference documentation for details. For more information see Data Import Export Statements.

The system views for procedures are summarized below:

Related Information

SAP HANA SQL and System Views Reference

5.1.6.1 SYS.PROCEDURES

Available stored procedures

Structure

Table 4:

Column name	Data type	Description
SCHEMA_NAME	NVARCHAR(256)	Schema name of the stored procedure
PROCEDURE_NAME	NVARCHAR(256)	Name of the stored procedure
PROCEDURE_OID	BIGINT	Object ID of the stored procedure
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'
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'

5.1.6.2 SYS. PROCEDURE_PARAMETERS

Parameters of stored procedures

Structure

Table 5:

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

5.1.6.3 SYS.OBJECT_DEPENDENCIES

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

Structure

Table 6:

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
DEPENDENCY_TYPE	INTEGER	Type of dependency between base and dependent object

5.1.6.3.1 Object Dependencies View Examples

In this section we explore 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

Firstly you will find all the (direct and indirect) base objects of the procedure DEPS.MY_PROC. You execute the following statement.

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

The result obtained is as follows:

Table 7:

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

Let's examine the DEPENDENCY_TYPE column in more detail. As you obtained the results in the table above via a select on all the base objects of the procedure, the objects show include both persistent and transient objects. You can distinguish between these object dependency types using the DEPENDENCY_TYPE column, as shown below:

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.

Now you will obtain only the base objects that are used in DEPS.MY_PROC. You execute the following statement.

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

The result obtained is as follows:

Table 8:

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

Finally you find all the dependent objects that are using DEPS.MY_PROC. You execute the following statement.

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

The result obtained is as follows:

Table 9:

BASE_SCHEMA_NAME	BASE_OBJECT_NAME	BASE_OBJECT_TYPE	DEPENDENT_SCHEMA_NAME	DEPENDENT_OBJECT_NAME	DEPENDENT_OBJECT_TYPE	DEPENDENCY_TYPE
DEPS	MY_PROC	PROCEDURE	DEPS	MY_CALLER	PROCEDURE	1

5.2 User Defined Function

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

Table 10:

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

5.2.1 CREATE FUNCTION

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

Syntax

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

Syntax Elements

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

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

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

The input parameters of the function.

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

A function parameter with associated data type.

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

The variable name for a parameter.

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

Scalar UDF only supports primitive SQL types as input, whereas Table UDF also supports table types as input. Currently the following primitive SQL types are allowed in scalar UDF:

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

Table UDF allows a wider range of primitive SQL types:

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

To look at a table type previously defined with the `CREATE TYPE` command, see CREATE TYPE [page 14].

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

A table type implicitly defined within the signature.

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

Table UDF 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>
```

Defines the output parameters

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

Defines the structure of the returned table data.

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

Default: SQLSCRIPT

Defines the programming language used in the function.

> **Note**
>
> Only SQLScript UDF 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> ::= <identifier>
```

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

```
 <function_body>        ::= <scalar_function_body>|<table_function_body>
 <scalar_function_body> ::= [DECLARE <func_var>]
```

```
                              <proc_assign>
 <table_function_body>  ::= [<func_block_decl_list>]
                            [<func_handler_list>]
                             <func_stmt_list>
                             <func_return_statement>
```

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

For the definition of <proc_assign>, see CREATE PROCEDURE [page 17].

```
 <func_block_decl_list> ::= DECLARE { <func_var>|<func_cursor>|<func_condition> }
 <func_var>             ::= <variable_name_list> [CONSTANT] { <sql_type>|
<array_datatype> } [NOT NULL][<func_default>];
 <array_datatype>       ::= <sql_type> ARRAY [ := <array_constructor> ]
 <array_constructor>    ::= ARRAY ( <expression> [{,<expression>}...] )
 <func_default>         ::= { DEFAULT | := } <func_expr>
 <func_expr>            ::= !!An element of the type specified by <sql_type>
```

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

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

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

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

See CREATE PROCEDURE [page 17].

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

For further information of the definitions in <func_stmt>, see CREATE PROCEDURE [page 17]..

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

A table function must contain a return statement.

Example

You create a table UDF with the following definition.

```
 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;
```

You use the scale function like a table. See the following example:

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

You also create a scalar UDF with the following definition.

```
 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;
```

You use the `func_add_mul` function like a built-in function. See the following example:

```
 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
```

You create a function `func_mul` which is assigned to a scalar variable in the `func_mul_wrapper` function.

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

5.2.2 DROP FUNCTION

Syntax

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

Syntax Elements

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

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

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

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

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

```
CASCADE
```

Drops the function and dependent objects.

```
RESTRICT
```

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

Description

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

Examples

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

```
DROP FUNCTION my_func;
```

5.2.3 Function Parameters

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

Table 11:

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

5.2.4 Function Metadata

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

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

5.2.4.1 SYS.FUNCTIONS

A list of available functions

Structure

Table 12:

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

Column name	Data type	Description
FUNCTION_OID	BIGINT	Object ID of 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
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'

5.2.4.2 SYS.FUNCTIONS_PARAMETERS

A list of parameters of functions

Structure

Table 13:

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

Column name	Data type	Description
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
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 Declarative SQLScript Logic

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

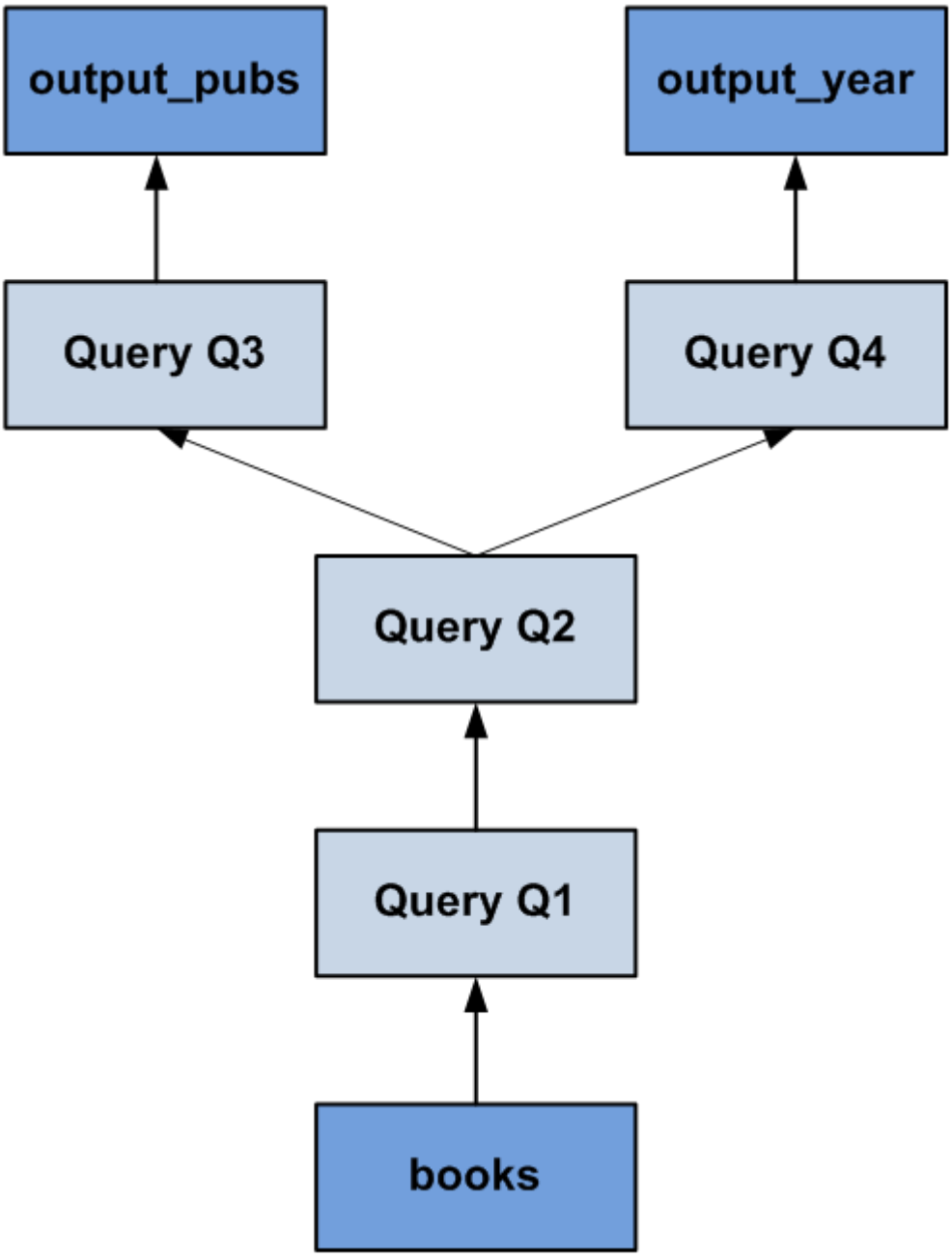
This approach leads to data flows which are free of side effects. The declarative nature to define business logic might require some deeper thought when specifying an algorithm, but it gives the SAP HANA database freedom to optimize the data flow which may result in better performance.

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

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

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

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



6.1 Table Parameter

Syntax

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

Description

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

Example

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

Defines the tabular structure directly in the signature.

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

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

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

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

6.2 Local Table Variables

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

6.3 Table Variable Type Definition

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

If the table variable derived its type from the SQL Query, the SQLScript compiler determines the type from the first assignments of that variable. This provides a great deal of flexibility. One disadvantage however, is that it also lead to many type conversions in the background. This is because sometimes the derived table type does not match the typed table parameters at the signature. This can lead to additional conversion costs, which are unnecessary.

To avoid this unnecessary cost, you can declare the type of a table variable explicitly.

Signature

```
DECLARE <sql_identifier> [{,<sql_identifier> }...] TABLE
(<column_list_definition>)
```

Description

Local table variables are declared using the `DECLARE` keyword. A table variable `var` can be referenced by using `:var`. For more information, see Referencing Variables [page 52]. The `<sql_identifier>` must be unique among all other scalar variables and table variables in the same code block. You can, however, use names that are identical to the name of another variable in a different code block. Additionally, you can reference these 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 as follows:

```
 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 as follows:

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

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

Another difference, compared to derived types, is that a reference to a table variable without assignment leads to an error during compile time.

```
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 sends an error because table variable <a> is unassigned if <:num> is not 4. In comparison the derived table variable type approach would send an error at runtime, but only if <:num> is not 4.

The following table shows the differences:

Table 14:

	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	Pass compile phase even though the variable can be unassigned in some cases. Error when unassigned variable is used in execution time. (Success if the variable was assigned in execution time)	Error in compile phase if there's possibility of reference to unassigned table variable. But check only when a table variable is used. If a procedure passed compile phase, there is no possibility of a null referencing error during execution.

> **Note**
>
> The following declarations are not supported (compare with scalar variable declaration): CONSTANT, NOT NULL option, Default Value.

6.4 Binding Table Variables

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

6.5 Referencing Variables

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

Example:

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

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

6.6 Column View Parameter Binding

Syntax

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

Syntax Elements

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

The name of the column view.

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

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

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

Defines the parameter used to refer to the given expression.

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

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

Description

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

Examples:

Example 1 - Basic example

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

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

Example 2 - Using a Hierarchical View

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

```
CREATE PROCEDURE "EXTEND_EXPRESSION"(
    IN in_expr nvarchar(20),
    OUT out_result "TTY_HIER_OUTPUT")
LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
```

```
    DECLARE expr VARCHAR(256) := 'leaves(nodes())';
    IF :in_expr <> '' THEN
        expr := 'leaves(' || :in_expr || ')';
    END IF;
    out_result = SELECT query_node, result_node FROM h_proc ("expression"
=> :expr ) as h order by h.result_node;
END;
```

You call this procedure as follows.

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

7 Imperative SQLScript Logic

In this section we will focus on imperative language constructs such as loops and conditionals. The use of imperative logic splits the logic among several dataflows. For additional information, see Orchestration-Logic [page 11] and Declarative SQLScript Logic [page 47].

7.1 Local Scalar Variables

Syntax

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

Syntax Elements

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

Default value expression assignment.

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

The value to be assigned to the variable.

Description

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

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

Example

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

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

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

> **Note**
>
> Before the SAP HANA SPS 08 release, scalar UDF assignment to the scalar variable was not supported. If you wanted to get the result value from a scalar UDF and consume it in a procedure, the scalar UDF had to be used in a `SELECT` statement, even if this were expensive.
>
> It is now possible to assign a scalar UDF to scalar variable with the following limitation.
>
> **Limitation:** It is only supported with a scalar UDF that has 1 output. See the following code examples.
>
> Consuming the result using an SQL statement:
>
> ```
> DECLARE i INTEGER DEFAULT 0;
> SELECT SUDF_ADD(:input1, :input2) into i from dummy;
> ```
>
> Assign the scalar UDF to the scalar variable:
>
> ```
> DECLARE i INTEGER DEFAULT 0;
> i := SUDF_ADD(:input1, :input2);
> ```

7.2 Variable Scope Nesting

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

Consider the following code:

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

```
            a := 3;
        END;
        val := a;
    END;
END;
```

When you call this procedure the result is:

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

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

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

Now when you call this modified procedure the result is:

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

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

Local Variables in Control Structures

Conditionals

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

```
        END IF;
        v := 1 / (4-:inval);
    END IF;
    v := 1 / (5-:inval);
END;
call nested_block_if(1, ?)
-->OUT:[1]
call nested_block_if(2, ?)
-->OUT:[2]
call nested_block_if(3, ?)
-->OUT:[3]
call nested_block_if(4, ?)
--> OUT:[2]
call nested_block_if(5, ?)
--> OUT:[1]
```

While Loop

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

For Loop

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

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

Loop

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

```
 CREATE PROCEDURE nested_block_loop(IN inval INT, OUT val INT) LANGUAGE
SQLSCRIPT READS SQL DATA AS
BEGIN
    DECLARE a1 int;
    DECLARE a2 int;
    DECLARE a3 int;
    DECLARE v1 int default 1;
    DECLARE v2 int default 1;
    DECLARE v3 int default 1;
    DECLARE CURSOR C FOR SELECT * FROM mytab1;
    OPEN C;
    FETCH C into a1;
    CLOSE C;
    LOOP
        DECLARE CURSOR C FOR SELECT * FROM mytab2;
        OPEN C;
        FETCH C into a2;
        CLOSE C;
        LOOP
            DECLARE CURSOR C FOR SELECT * FROM mytab3;
            OPEN C;
            FETCH C INTO a3;
            CLOSE C;
            IF :v2 = 1 THEN
                BREAK;
            END IF;
        END LOOP;
        IF :v1 = 1 THEN
            BREAK;
        END IF;
    END LOOP;
    IF :inval = 1 THEN
        val := :a1;
    ELSEIF :inval = 2 THEN
        val := :a2;
    ELSEIF :inval = 3 THEN
        val := :a3;
    END IF;
END;
call nested_block_loop(1, ?)
--> OUT:[1]
call nested_block_loop(2, ?)
--> OUT:[2]
call nested_block_loop(3, ?)
--> OUT:[3]
```

7.3 Control Structures

7.3.1 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`.

> **Note**
>
> `NULL` is the default value for all local variables.

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

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

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

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

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

Description:

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

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

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

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

Examples:

Example 1

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

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

Example 2

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

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

Example 3

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

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

Related Information

ins_msg_proc [page 124]

7.3.2 While Loop

Syntax:

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

Syntax elements:

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

Defines a Boolean expression which evaluates to true or false.

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

Description:

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

Example 1

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

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

Example 2

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

```
CREATE PROCEDURE proc (in input1 INTEGER, out output1 TYPE1)
AS
BEGIN
```

```
    DECLARE i INTEGER DEFAULT :input1;
    DECLARE cnt INTEGER DEFAULT 0;
    WHILE SUDF(:i) > 0 DO
        cnt := :cnt + 1;
        i := :i - 1;
    END WHILE;
    output1 = SELECT value FROM T1 where id = :cnt ;
END;
```

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

7.3.3 For Loop

Syntax:

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

Syntax elements:

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

Defines the variable that will contain the loop values.

```
REVERSE
```

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

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

Defines the starting value of the loop.

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

Defines the end value of the loop.

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

Defines the procedural statements that will be looped over.

Description:

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

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

Example 1

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

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

Example 2

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

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

7.3.4 Break and Continue

Syntax:

```
BREAK
CONTINUE
```

Syntax elements:

```
BREAK
```

Specifies that a loop should stop being processed.

```
CONTINUE
```

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

Description:

These statements provide internal control functionality for loops.

Example:

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

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

Related Information

ins_msg_proc [page 124]

7.4 Cursors

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

7.4.1 Define Cursor

Syntax:

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

Syntax elements:

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

Specifies the name of the cursor.

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

Defines an optional SELECT parameter.

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

Defines the variable name of the parameter.

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

Defines the datatype of the parameter.

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

Defines an SQL select statement. See SELECT.

Description:

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

> **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.

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

Related Information

SELECT

7.4.2 Open Cursor

Syntax:

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

Syntax elements:

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

Specifies the name of the cursor to be opened.

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

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

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

Specifies a scalar value to be passed to the cursor.

Description:

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

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

Example:

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

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

7.4.3 Close Cursor

Syntax:

```
CLOSE <cursor_name>
```

Syntax elements:

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

Specifies the name of the cursor to be closed.

Description:

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

Example:

You close the cursor `c_cursor1`.

```
CLOSE c_cursor1;
```

7.4.4 Fetch Query Results of a Cursor

Syntax:

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

Syntax elements:

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

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

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

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

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

Specifies the identifier of a variable.

Description:

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

Example:

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

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

Related Information

Attributes of a Cursor [page 68]

7.4.5 Attributes of a Cursor

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

Table 15: 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 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 124]

7.4.6 Looping over Result Sets

Syntax:

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

Syntax elements:

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

Defines an identifier to contain the row result.

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

Specifies the name of the cursor to be opened.

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

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

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

Specifies a scalar value to be passed to the cursor.

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

Defines the procedural statements that will be looped over.

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

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

Description:

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

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

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

Example:

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

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

Related Information

ins_msg_proc [page 124]

7.5 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 ) 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;
          SELECT 1/:p 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

- DDL
- Cursor
- Table assignments

> ⚠ Caution
>
> `COMMIT` and `ROLLBACK` are only available inside an `AUTONONMOUS TRANSACTION` block. Outside of the autonomous transaction block they are not supported.

7.6 Dynamic SQL

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

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

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

7.6.1 EXEC

Syntax:

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

Description:

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

Example:

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

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

7.6.2 EXECUTE IMMEDIATE

Syntax:

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

Description:

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

Example:

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

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

7.6.3 APPLY_FILTER

Syntax

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

Syntax Elements

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

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

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

You can use APPLY_FILTER with persistent tables and table variables.

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

The name of the table that is to be filtered.

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

The name of the table variable to be filtered.

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

The filter command to be applied.

> **Note**
>
> The following constructs are not supported in the filter string <filter_variable_name>:
>
> - • sub-queries, for example: CALL GET_PROCEDURE_NAME(' PROCEDURE_NAME in (SELECT object_name FROM SYS.OBJECTS), ?);
> - • fully-qualified column names, for example: CALL GET_PROCEDURE_NAME(' PROCEDURE.PROCEDURE_NAME = 'DSO', ?);

Description

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

Examples

Example 1 - Apply a filter on a persistent table

You create the following procedure

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

You call the procedure with two different filter variables.

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

```
CALL GET_PROCEDURE_NAME(' SCHEMA_NAME = ''SYS''', ?);
```

Example 2 - Using a table variable

```
CREATE TYPE outtype AS TABLE (SCHEMA_NAME NVARCHAR(255), PROCEDURE_NAME
NVARCHAR(255));
CREATE PROCEDURE GET_PROCEDURE_NAME (IN filter NVARCHAR(100), OUT procedures
outtype)
AS
BEGIN
    temp_procedures = SELECT SCHEMA_NAME, PROCEDURE_NAME FROM SYS.PROCEDURES;
    procedures = APPLY_FILTER(:temp_procedures,:filter);
END;
```

7.7 Exception Handling

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

7.7.1 DECLARE EXIT HANDLER

Syntax

```
<proc_handler>
          ::= DECLARE EXIT HANDLER FOR <proc_condition_value_list> <proc_stmt>
```

> **Note**
>
> This is a syntax fragment from the CREATE PROCEDURE statement. For the full syntax see, `CREATE PROCEDURE`.

Description

The DECLARE EXIT HANDLER parameter allows you to define exception handlers to process exception conditions in your procedures. You can explicitly signal an exception and completion condition within your code using SIGNAL and RESIGNAL.

Related Information

CREATE PROCEDURE [page 17]
SIGNAL and RESIGNAL [page 76]

7.7.2 DECLARE CONDITION

Syntax

```
DECLARE <condition name> CONDITION [ FOR <sqlstate value> ]
```

> **Note**
>
> This is a syntax fragment from the CREATE PROCEDURE statement. For the full syntax see, `CREATE PROCEDURE`.

Description

You use the DECLARE CONDITION parameter to name exception conditions, and optionally, their associated SQL state values.

Related Information

CREATE PROCEDURE [page 17]

7.7.3 SIGNAL and RESIGNAL

Syntax

```
  SIGNAL <signal value> [<set signal information>]
RESIGNAL [<signal value>] [<set signal information>]
```

> **Note**
>
> This is a syntax fragment from the CREATE PROCEDURE statement. For the full syntax see, `CREATE PROCEDURE`.

Description

You use the SIGNAL and RESIGNAL directives in your code to trigger exception states.

You can use SIGNAL or RESIGNAL with specified error code in user-defined error code range. A user-defined exception can be handled by the handler declared in the procedure. Also it can be also handled by the caller which can be another procedure or client.

Related Information

CREATE PROCEDURE [page 17]

7.7.4 Exception Handling Examples

General exception handling

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

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

Error code exception handling

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

```
CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
```

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

```
CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
CREATE PROCEDURE MYPROC AS
BEGIN
    DECLARE myVar INT;
    DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 1299
        BEGIN
               SELECT 0 INTO myVar FROM DUMMY;
                SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
                SELECT :myVar FROM DUMMY;
        END;
    SELECT I INTO myVar FROM MYTAB; --NO_DATA_FOUND exception
    SELECT 'NeverReached_noContinueOnErrorSemantics' FROM DUMMY;
END;
CALL MYPROC;
```

Conditional Exception Handling

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

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

Signal an exception

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

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

```
CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
CREATE PROCEDURE MYPROC AS
BEGIN
    DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 10001;
```

```
    DECLARE EXIT HANDLER FOR MYCOND SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
FROM DUMMY;
    INSERT INTO MYTAB VALUES (1);
    SIGNAL MYCOND SET MESSAGE_TEXT = 'my error';
    -- will not be reached
END;
CALL MYPROC;
```

Resignal an exception

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

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

Nested block exceptions.

You can declare exception handlers for nested blocks.

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

7.8 ARRAY

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

7.8.1 ARRAY CONSTRUCTOR

Syntax

```
ARRAY(<value_expression> [{, <value_expression>}...])
```

Syntax Elements

```
<value_expression> ::= <string_literal> | <number>
```

The array can contain strings or numbers.

Description

The ARRAY function returns an array whose elements are specified in the list of value expressions.

Examples

You define an integer array that contains the numbers 1,2 and 3.

```
array_id  INTEGER ARRAY[] := ARRAY(1, 2, 3);
```

7.8.2 DECLARE ARRAY-TYPED VARIABLE

Syntax

```
<array_name> <type> ARRAY  [:= <array_constructor>]
```

Syntax Elements

```
<array_variable> ::= <identifier>
```

The variable that will contain the array.

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

The data type for the array elements.

```
<array_constructor> ::= ARRAY(<value_expression> [{, <value_expression>}...])
```

Defines the array elements. For more information, see **ARRAY CONSTRUCTOR**

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

Description

Declare an array variable whose element type is <type>, which represents one of the *SQL* types.

Currently only an unbounded `ARRAY` is supported with a maximum cardinality of 2^31. You cannot define a static-size for an array.

> **Note**
>
> Note you cannot use TEXT or SHORTTEXT as the array type.

Examples

Example 1

You define an empty array of type `INTEGER`.

```
array_int INTEGER ARRAY;
```

Example 2

You define an `INTEGER` array with values 1,2 and 3.

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

Related Information

ARRAY CONSTRUCTOR [page 80]

7.8.3 SET AN ELEMENT OF AN ARRAY

Syntax

```
:<array_variable>'[' <array_index> ']' := <value_expression>
```

Syntax Elements

```
<array_variable> ::= <identifier>
```

The array to be operated upon.

```
<array_index> ::= <unsigned_integer>
```

The index of the element in the array to be modified. `<array_index>` can be any value from 1 to 2^31.

> **Note**
>
> The array index starts with the index `1`

```
<value_expression> ::= <string_literal> | <number>
```

The value to which the array element should be set.

Description

The array element specified by `<array_index>` can be set to `<value_expression>`.

Examples

You create an array with the values `1,2,3`. You add `10` to the first element in the array.

```
id Integer ARRAY[] := ARRAY(1, 2, 3);
id[1] := :id[1] + 10;
```

7.8.4 RETURN AN ELEMENT OF AN ARRAY

Syntax

```
 <scalar_variable> := <array_variable> "[" <array_index>"]"
```

Syntax Elements

```
<scalar_variable> :: = <identifier>
```

The variable where the array element will be assigned.

```
<array_variable> ::= <identifier>
```

The target array where the element is to be obtained from.

```
<array_index> ::= <unsigned_integer>
```

The index of the element to be returned. <array_index> can be any value from 1 to 2,147,483,646.

Description

The value of the array element specified by <array_index given_index> can be returned. The array element can be referenced in *SQL* expressions.

Example

You create and call the following procedure.

```
CREATE PROCEDURE ReturnElement (OUT output INT) AS
BEGIN
   DECLARE id  INTEGER ARRAY := ARRAY(1, 2, 3);
   DECLARE n   INTEGER := 1;
   output := :id[:n];
END;
call ReturnElement(?) ;
```

The results is as follows.

```
Out(1)
------
1
```

7.8.5 UNNEST

Syntax

```
UNNEST(:<array_variable> [ {, array_variable} ...] )[WITH ORDINALITY]
       [AS <return_table_specification>)]
```

Syntax Elements

```
<array_variable> ::= <identifier>
```

The array to be operated upon.

```
WITH ORDINALITY
```

Specifies that an ordinal column will be appended to the returned table. When you use this, you must explicitly specify an alias for the ordinal column. For more information, see *Example 2* where "SEQ" is specified as the alias.

```
<return_table_specification> ::= (<column_name> [ {, column_name}… ])
```

The column names of the returned table.

```
<column_name> ::= <identifier>
```

The name of a column in the returned table.

Description

The UNNEST function converts an array into a table. UNNEST returns a table including a row for each element of the array specified. If there are multiple arrays given, 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 the elements of the arrays are filled with NULL values.

> **Note**
>
> The UNNEST function cannot be referenced directly in FROM clause of a SELECT statement.

Examples

Example 1

You use UNNEST to obtain the values of an ARRAY id and name in which the cardinality differs.

```
CREATE PROCEDURE ARRAY_UNNEST_SIMPLE()
LANGUAGE SQLSCRIPT SQL SECURITY INVOKER AS
BEGIN
     DECLARE id  INTEGER ARRAY;
     DECLARE name  VARCHAR(10) ARRAY;
     id[1] := 1;
     id[2] := 2;
     name[1] := 'name1';
     name[2] := 'name2';
     name[3] := 'name3';
     name[5] := 'name5';
     rst = UNNEST(:id, :name) AS ("ID", "NAME");
     SELECT * FROM :rst;
END;
CALL ARRAY_UNNEST_SIMPLE();
```

The result of calling this procedure is as follows.

```
ID   NAME
----------
1    name1
2    name2
?    name3
```

```
?      ?
?    name5
```

Example 2

You use UNNEST with the WITH ORDINALITY directive to generate a sequence column along with the results set .

```
CREATE PROCEDURE ARRAY_UNNEST()
LANGUAGE SQLSCRIPT SQL SECURITY INVOKER AS
BEGIN
    DECLARE amount  INTEGER     ARRAY := ARRAY(10, 20);

    rst = UNNEST( :amount) WITH ORDINALITY AS ( "AMOUNT", "SEQ");
    select SEQ,  AMOUNT from :rst;
END;
CALL ARRAY_UNNEST();
```

The result of calling this procedure is as follows.

```
SEQ     AMOUNT
----------------
1     10
2     20
```

7.8.6 ARRAY_AGG

Syntax

```
ARRAY_AGG”(“:<table_variable>.<column_name> [<order_by_clause>]”)”
```

Syntax Elements

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

The name of the table variable to be converted.

```
<column_name>::= <identifier>
```

The name of the column, within the table variable, to be converted.

```
<order_by_clause> ::= ORDER BY { <order_by_expression>, ... }
```

The `ORDER BY` clause is used to sort records by expressions or positions.

```
<order_by_expression> ::= <expression> [ ASC | DESC ] [ NULLS FIRST | NULLS
LAST ]
```

Specifies the ordering of data.

```
ASC | DESC
```

`ASC` sorts records in ascending order. `DESC` sorts records in descending order. The default is ASC.

```
NULLS FIRST | NULLS LAST
```

Specifies where in the results set `NULL` values should appear. By default for ascending ordering `NULL` values are returned first, and for descending they are returned last. You can override this behavior using `NULLS FIRST` or `NULLS LAST` to explicitly specify `NULL` value ordering.

Description

The `ARRAY_AGG` function converts a column of a table into an array.

> **Note**
>
> `ARRAY_AGG` function does not support using value expressions instead of table variables.

Examples

You create the following table and procedure.

```
CREATE TABLE tab1 (a INT, b INT, c INT);
INSERT INTO tab1 VALUES (1, 4, 1);
INSERT INTO tab1 VALUES (2, 3, 2);
CREATE PROCEDURE ARRAY_AGG_TEST()
LANGUAGE SQLSCRIPT SQL SECURITY INVOKER AS
BEGIN
   DECLARE id  Integer ARRAY;
   tab = SELECT * FROM tab1;
   id := ARRAY_AGG(:tab.a ORDER BY c, b DESC);
   rst = UNNEST(:id);
   SELECT * FROM :rst;
END;
CALL ARRAY_AGG_TEST();
```

The result of calling this procedure is as follows.

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

7.8.7 TRIM_ARRAY

Syntax

```
TRIM_ARRAY”(“:<array_variable>, <trim_quantity>”)”
```

Syntax Elements

```
<array_variable> ::= <identifier>
```

The array to be operated upon.

```
<trim_quantity> ::= <unsigned_integer>
```

The number of elements to be removed.

Description

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>.

Examples

You create the following procedure.

```
CREATE PROCEDURE ARRAY_TRIM()
LANGUAGE SQLSCRIPT SQL SECURITY INVOKER AS
BEGIN
   DECLARE array_id  Integer ARRAY := ARRAY(0, 1, 2);
   array_id := TRIM_ARRAY(:array_id, 1);
   rst1 = UNNEST(:array_id) as ("ID");
   select * from :rst1 order by "ID";
END;
```

The result of calling this procedure is as follows.

```
ID
---
0
1
```

7.8.8 CARDINALITY

Syntax

```
CARDINALITY(:<array_variable>)
```

Syntax Elements

```
<array_variable> ::= <identifier>
```

The array to be operated upon.

Description

The CARDINALITY function returns the number of elements in the array <array_variable>. It returns N (>= 0) if the index of the N-th element is the largest among the indices.

Example

Example 1

You create and call the following procedure.

```
CREATE PROCEDURE CARDINALITY_1() AS
BEGIN
   DECLARE array_id  Integer ARRAY := ARRAY(1, 2, 3);
   DECLARE n Integer;
   n := CARDINALITY(:array_id);
   select :n as card from dummy;
END;
CALL CARDINALITY_1();
```

The result of calling this procedure is as follows.

```
CARD
----
3
```

Example 2

You create and call the following procedure.

```
CREATE PROCEDURE CARDINALITY_2() AS
BEGIN
   DECLARE array_id  Integer ARRAY;
   DECLARE n Integer;
   n := CARDINALITY(:array_id);
   select :n as card from dummy;
END;
CALL CARDINALITY_2();
```

The result of calling this procedure is as follows.

```
CARD
----
0
```

Example 3

You create and call the following procedure.

```
CREATE PROCEDURE CARDINALITY_3() AS
BEGIN
   DECALRE array_id  Integer ARRAY;
   DECLARE   n Integer;
   array_id[20] := NULL;
   n := CARDINALITY(:array_id);
   select :n as card from dummy;
END;
CALL CARDINALITY_3();
```

The result of calling this procedure is as follows.

```
CARD
----
20
```

7.8.9 CONCATENATE TWO ARRAYS

Syntax

```
:<array_variable_left> "||" :<array_variable_right>
or
CONCAT'('<array_variable_left> , <array_variable_right> ')'
```

Syntax Elements

```
<array_variable_left> ::= <identifier>
```

The first array to be concatenated.

```
<array_variable_right> ::= <identifier>
```

The second array to be concatenated.

Description

The concat function concatenates two arrays. It returns the new array that contains a concatenation of <array_variable_left> and <array_variable_right>.

Examples

You create and call the following procedure.

```
CREATE PROCEDURE ARRAY_COMPLEX_CONCAT3()
LANGUAGE SQLSCRIPT SQL SECURITY INVOKER AS
BEGIN
    DECLARE id1  INTEGER ARRAY;
    DECLARE id2  INTEGER ARRAY;
    DECLARE id3  INTEGER ARRAY;
    DECLARE id4  INTEGER ARRAY;
    DECLARE id5  INTEGER ARRAY;
    DECLARE 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");
    SELECT "seq", "id" FROM :rst1 ORDER BY "seq";
END;
CALL ARRAY_COMPLEX_CONCAT3();
```

8 Calculation Engine Plan Operators

➔ Recommendation

SAP recommends that you use SQL rather than Calculation Engine Plan Operators with SQLScript.

The execution of Calculation Engine Plan Operators currently is bound to processing within the calculation engine and does not allow a possibility to use alternative execution engines, such as L native execution. As most Calculation Engine Plan Operators are converted internally and treated as SQL operations, the conversion requires multiple layers of optimizations. This can be avoided by direct SQL use. Depending on your system configuration and the version you use, mixing Calculation Engine Plan Operators and SQL can lead to significant performance penalties when compared to to plain SQL implementation.

Table 16: Overview: Mapping between CE_* Operators and SQL

CE Operator	CE Syntax	SQL Equivalent
CE_COLUMN_TABLE	CE_COLUMN_TABLE(<table_name>[,<attributes>])	SELECT [<attributes>] FROM <table_name>
CE_JOIN_VIEW	CE_JOIN_VIEW(<column_view_name>[,<attributes>]) out = CE_JOIN_VIEW("PRODUCT_SALES", ["PRODUCT_KEY", "PRODUCT_TEXT", "SALES"]);	SELECT [<attributes>] FROM <column_view_name> out = SELECT product_key, product_text, sales FROM product_sales;
CE_OLAP_VIEW	CE_OLAP_VIEW (<olap_view_name>[,<attributes>]) out = CE_OLAP_VIEW("OLAP_view", ["DIM1", SUM("KF")]);	SELECT [<attributes>] FROM <olap_view_name> out = select dim1, SUM(kf) FROM OLAP_view GROUP BY dim1;
CE_CALC_VIEW	CE_CALC_VIEW(<calc_view_name>,[<attributes>]) out = CE_CALC_VIEW("TESTCECTABLE", ["CID", "CNAME"]);	SELECT [<attributes>] FROM <calc_view_name> out = SELECT cid, cname FROM "TESTCECTABLE";

CE Operator	CE Syntax	SQL Equivalent
CE_JOIN	CE_JOIN(<left_table>,<right_table>,<join_attributes>[<projection_list>]) ot_pubs_books1 = CE_JOIN (:lt_pubs, :it_books, ["PUBLISHER"]);	SELECT [<projection_list>] FROM <left_table>,<right_table> WHERE <join_attributes> ot_pubs_books1 = SELECT P.publisher AS publisher, name, street,post_code, city, country, isbn, title, edition, year, price, crcy FROM :lt_pubs AS P, :it_books AS B WHERE P.publisher = B.publisher;
CE_LEFT_OUTER_JOIN	CE_LEFT_OUTER_JOIN(<left_table>,<right_table>,<join_attributes>[<projection_list>])	SELECT [<projection_list>] FROM <left_table> LEFT OUTER JOIN <right_table> ON <join_attributes>
CE_RIGHT_OUTER_JOIN	CE_RIGHT_OUTER_JOIN(<left_table>,<right_table>,<join_attributes>[<projection_list>])	SELECT [<projection_list>] FROM <left_table> RIGHT OUTER JOIN <right_table> ON <join_attributes>
CE_PROJECTION	CE_PROJECTION(<table_variable>,<projection_list>[,<filter>]) ot_books1 = CE_PROJECTION (:it_books, ["TITLE","PRICE", "CRCY" AS "CURRENCY"], '"PRICE" > 50');	SELECT <projection_list> FROM <table_variable> where [<filter>] ot_book2= SELECT title, price, crcy AS currency FROM :it_b ooks WHERE price > 50;
CE_UNION_ALL	CE_UNION_ALL(<table_variable1>,<table_variable2>) ot_all_books1 = CE_UNION_ALL (:lt_books, :it_audiobooks);	SELECT * FROM <table_variable1> UNION ALL SELECT * FROM <table_variable2> ot_all_books2 = SELECT * FROM :lt_books UNION ALL SELECT * FROM :it_audiobooks;
CE_CONVERSION	CE_CONVERSION(<table_variable>,<conversion_params>, [<rename_clause>])	SQL-Function CONVERT_CURRENCY

CE Operator	CE Syntax	SQL Equivalent
CE_AGGREGATION	CE_AGGREGATION(<table_variable>,<aggregate_list> [,<group_columns>]) ot_books1 = CE_AGGREGATION (:it_books, [COUNT ("PUBLISHER") AS "CNT"], ["YEAR"]);	SELECT <aggregate_list> FROM <table_variable> [GROUP BY <group_columns>] ot_books2 = SELECT COUNT (publisher) AS cnt, year FROM :it_books GROUP BY year;
CE_CALC	CE_CALC('<expr>', <result_type>) TEMP = CE_PROJECTION(:table_var, ["ID" AS "KEY", CE_CALC('rownum()', INTEGER) AS "T_ID"]);	SQL Function TEMP = SELECT "ID" AS "KEY", ROW_NUMBER() OVER () AS "T_ID" FROM :table_var

Calculation engine plan operators encapsulate data-transformation functions and can be used in the definition of a procedure or a table user-defined function. They constitute a no longer recommended alternative to using SQL statements. Their logic is directly implemented in the calculation engine, which is the execution environments of SQLScript.

There are different categories of operators.

- Data Source Access operators that bind a column table or a column view to a table variable.
- Relational operators that allow a user to bypass the SQL processor during evaluation and to directly interact with the calculation engine.
- Special extensions that implement functions.

8.1 Data Source Access Operators

The data source access operators bind the column table or column view of a data source to a table variable for reference by other built-in operators or statements in a SQLScript procedure.

8.1.1 CE_COLUMN_TABLE

Syntax:

```
CE_COLUMN_TABLE(<table_name> [<attributes>])
```

Syntax Elements:

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

```
<schema_name>  ::= <identifier>
```

Identifies the table name of the column table, with optional schema name.

```
<attributes>  ::= '[' <attrib_name>[{, <attrib_name> }…] ']'
<attrib_name> ::= <string_literal>
```

Restricts the output to the specified attribute names.

Description:

The `CE_COLUMN_TABLE` operator provides access to an existing column table. It takes the name of the table and returns its content bound to a variable. Optionally a list of attribute names can be provided to restrict the output to the given attributes.

Note that many of the calculation engine operators provide a projection list for restricting the attributes returned in the output. In the case of relational operators, the attributes may be renamed in the projection list. The functions that provide data source access provide no renaming of attributes but just a simple projection.

> **Note**
>
> Calculation engine plan operators that reference identifiers must be enclosed with double-quotes and capitalized, ensuring that the identifier's name is consistent with its internal representation.
>
> If the identifiers have been declared without double-quotes in the `CREATE TABLE` statement (which is the normal method), they are internally converted to upper-case letters. Identifiers in calculation engine plan operators must match the internal representation, that is they must be upper case as well.
>
> In contrast, if identifiers have been declared with double-quotes in the `CREATE TABLE` statement, they are stored in a case-sensitive manner. Again, the identifiers in operators must match the internal representation.

8.1.2 CE_JOIN_VIEW

Syntax:

```
CE_JOIN_VIEW(<column_view_name>[{,<attributes>,}...])
```

Syntax elements:

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

Identifies the column view, with optional schema name.

```
<attributes>  ::= '[' <attrib_name>[{, <attrib_name> }…] ']'
<attrib_name> ::= <string_literal>  [AS <column_alias>]
```

Specifies the name of the required columns from the column view.

```
column_alias ::= <string literal>
```

A string representing the desired column alias.

Description:

The `CE_JOIN_VIEW` operator returns results for an existing join view (also known as Attribute View). It takes the name of the join view and an optional list of attributes as parameters of such views/models.

8.1.3 CE_OLAP_VIEW

Syntax:

```
CE_OLAP_VIEW(<olap_view_name>, '['<attributes>']')
```

Syntax elements:

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

Identifies the olap view, with optional schema name.

```
<attributes> ::= <aggregate_exp> [{, <dimension>}…] [{, <aggregate_exp>}…]
```

Specifies the attributes of the OLAP view.

> **Note**
>
> Note you must have at least one <aggregation_exp> in the attributes.

```
<aggregate_exp> ::= <aggregate_func>(<aggregate_column> [AS <column_alias>])
```

Specifies the required aggregation expression for the key figure.

```
<aggregate_func> ::= COUNT | SUM | MIN | MAX
```

Specifies the aggregation function to use. Supported aggregation functions are:

- `count("column")`
- `sum("column")`
- `min("column")`
- `max("column")`
- use `sum("column") / count("column")` to compute the average

```
<aggregate_column> ::= <string_literal>
```

The identifier for the aggregation column.

```
<column_alias> ::= <string_literal>
```

Specifies an alias for the aggregate column.

```
<dimension> ::= <string_literal>
```

The dimension on which the OLAP view should be grouped.

Description:

The `CE_OLAP_VIEW` operator returns results for an existing OLAP view (also known as an Analytical View). It takes the name of the OLAP view and an optional list of key figures and dimensions as parameters. The OLAP cube that is described by the OLAP view is grouped by the given dimensions and the key figures are aggregated using the default aggregation of the OLAP view.

8.1.4 CE_CALC_VIEW

Syntax:

```
CE_CALC_VIEW(<calc_view_name>, [<attributes>])
```

Syntax elements:

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

Identifies the calculation view, with optional schema name.

```
<attributes>  ::= '[' <attrib_name>[{, <attrib_name> }…] ']'
<attrib_name> ::= <string_literal>
```

Specifies the name of the required attributes from the calculation view.

Description:

The `CE_CALC_VIEW` operator returns results for an existing calculation view. It takes the name of the calculation view and optionally a projection list of attribute names to restrict the output to the given attributes.

8.2 Relational Operators

The calculation engine plan operators presented in this section provide the functionality of relational operators that are directly executed in the calculation engine. This allows exploitation of the specific semantics of the calculation engine and to tune the code of a procedure if required.

8.2.1 CE_JOIN

Syntax:

```
CE_JOIN (<left_table>, <right_table>, <join_attributes> [<projection_list>])
```

Syntax elements:

```
<left_table>  ::= :<identifier>
```

Identifies the left table of the join.

```
<right_table> ::= :<identifier>
```

Identifies the right table of the join.

```
<join_attributes> ::= '[' <join_attrib>[{, <join_attrib> }…] ']'
<join_attrib>     ::= <string_literal>
```

Specifies a list of join attributes. Since `CE_JOIN` requires equal attribute names, one attribute name per pair of join attributes is sufficient. The list must at least have one element.

```
<projection_list> ::= '[' {, <attrib_name> }… ']'
<attrib_name>     ::= <string_literal>
```

Specifies a projection list for the attributes that should be in the resulting table.

> **Note**
>
> If the optional projection list is present, it must at least contain the join attributes.

Description:

The `CE_JOIN` operator calculates a natural (inner) join of the given pair of tables on a list of join attributes. For each pair of join attributes, only one attribute will be in the result. Optionally, a projection list of attribute names can be given to restrict the output to the given attributes. Finally, the plan operator requires each pair of join attributes to have identical attribute names. In case of join attributes having different names, one of them must be renamed prior to the join.

8.2.2 CE_LEFT_OUTER_JOIN

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

8.2.3 CE_RIGHT_OUTER_JOIN

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

> **Note**
>
> CE_FULL_OUTER_JOIN is not supported.

8.2.4 CE_PROJECTION

Syntax:

```
CE_PROJECTION(<var_table>, <projection_list>[, <filter>])
```

Syntax elements:

```
<var_table> ::= :<identifier>
```

Specifies the table variable which is subject to the projection.

```
<projection_list> ::= '[' <attrib_name>[{, <attrib_name> }…] ']'
<attrib_name>     ::= <string_literal>  [AS <column_alias>]
<column_alias>    ::= <string_literal>
```

Specifies a list of attributes that should be in the resulting table. The list must at least have one element. The attributes can be renamed using the SQL keyword `AS`, and expressions can be evaluated using the `CE_CALC` function.

```
<filter> ::= <filter_expression>
```

Specifies an optional filter where Boolean expressions are allowed. See CE_CALC [page 100] for the filter expression syntax.

Description:

Restricts the columns of the table variable `<var_table>` to those mentioned in the projection list. Optionally, you can also rename columns, compute expressions, or apply a filter.

With this operator, the `<projection_list>` is applied first, including column renaming and computation of expressions. As last step, the filter is applied.

> ⚠ Caution
>
> Be aware that **`<filter>`** in `CE_PROJECTION` can be vulnerable to SQL injection because it behaves like dynamic SQL. Avoid use cases where the value of **`<filter>`** is passed as an argument from outside of the procedure by the user himself or herself, for example:
>
> ```
> create procedure proc (in filter nvarchar (20), out output ttype)
> begin
> tablevar = CE_COLUMN_TABLE(TABLE);
> output = CE_PROJECTION(:tablevar,
>          ["A", "B"], '"B" = :filter );
> end;
> ```
>
> It enables the user to pass any expression and to query more than was intended, for example: `'02 OR B = 01'`.
>
> SAP recommends that you use plain SQL instead.

8.2.5 CE_CALC

Syntax:

```
CE_CALC ('<expr>', <result_type>)
```

Syntax elements:

```
<expr> ::= <expression>
```

Specifies the expression to be evaluated. Expressions are analyzed using the following grammar:

- b --> b1 ('or' b1)*
- b1 --> b2 ('and' b2)*
- b2 --> 'not' b2 | e (('<' | '>' | '=' | '<=' | '>=' | '!=') e)*
- e --> '-'? e1 ('+' e1 | '-' e1)*
- e1 --> e2 ('*' e2 | '/' e2 | '%' e2)*
- e2 --> e3 ('**' e2)*
- e3 --> '-' e2 | id ('(' (b (',' b)*)? ')')? | const | '(' b ')'

Where terminals in the grammar are enclosed, for example 'token' (denoted with id in the grammar), they are like SQL identifiers. An exception to this is that unquoted identifiers are converted into lower-case. Numeric constants are basically written in the same way as in the *C* programming language, and string constants are enclosed in single quotes, for example, 'a string'. Inside string, single quotes are escaped by another single quote.

An example expression valid in this grammar is: `"col1" < ("col2" + "col3")`. For a full list of expression functions, see the following table.

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

Specifies the result type of the expression as an SQL type

Description:

`CE_CALC` is used inside other relational operators. It evaluates an expression and is usually then bound to a new column. An important use case is evaluating expressions in the `CE_PROJECTION` operator. The `CE_CALC` function takes two arguments:

The following expression functions are supported:

Table 17: Expression Functions

Name	Description	Syntax
Conversion Functions	Convert between data types	
float	convert arg to float type	float float(arg)
double	convert arg to double type	double double(arg)
decfloat	convert arg to decfloat type	decfloat decfloat(arg)

Name	Description	Syntax
fixed	convert arg to fixed type	fixed fixed(arg, int, int)
string	convert arg to string type	string string(arg)
date	convert arg to date type daydate[1]	daydate(stringarg), daydate daydate(fixedarg)
String Functions	Functions on strings	
charpos	returns the one-based position of the nth character in a string. The string is interpreted as using a UTF-8 character encoding	charpos(string, int)
chars	returns the number of characters in a UTF-8 string. In a CESU-8 encoded string this function returns the number of 16-bit words utilized by the string, just the same as if the string is encoded using UTF-16.	chars(string)
strlen	returns the length of a string in bytes, as an integer number[1]	int strlen(string)
midstr	returns a part of the string starting at arg2, arg3 bytes long. arg2 is counted from 1 (not 0) [2]	string midstr(string, int, int)
leftstr	returns arg2 bytes from the left of the arg1. If arg1 is shorter than the value of arg2, the complete string will be returned. [1]	string leftstr(string, int)
rightstr	returns arg2 bytes from the right of the arg1. If arg1 is shorter than the value of arg2, the complete string will be returned. [1]	string rightstr(string, int)
instr	returns the position of the first occurrence of the second string within the first string (>= 1) or 0, if the second string is not contained in the first. [1]	int instr(string, string)
hextoraw	converts a hexadecimal representation of bytes to a string of bytes. The hexadecimal string may contain 0-9, upper or lowercase a-f and no spaces between the two digits of a byte; spaces between bytes are allowed.	string hextoraw(string)
rawtohex	converts a string of bytes to its hexadecimal representation. The output will contain only 0-9 and (upper case) A-F, no spaces and is twice as many bytes as the original string.	string rawtohex(string)

Name	Description	Syntax
ltrim	removes a whitespace prefix from a string. The Whitespace characters may be specified in an optional argument. This functions operates on raw bytes of the UTF8-string and has no knowledge of multi byte codes (you may not specify multi byte whitespace characters).	• string ltrim(string) • string ltrim(string, string)
rtrim	removes trailing whitespace from a string. The Whitespace characters may be specified in an optional argument. This functions operates on raw bytes of the UTF8-string and has no knowledge of multi byte codes (you may not specify multi byte whitespace characters).	• string rtrim(string) • string rtrim(string, string)
trim	removes whitespace from the beginning and end of a string. The following statements are functionally: • trim(s) = ltrim(rtrim(s)) • trim(s1, s2) = ltrim(rtrim(s1, s2), s2)	• string trim(string) • string trim(string, string)
lpad	adds whitespace to the left of a string. A second string argument specifies the whitespace which will be added repeatedly until the string has reached the intended length. If no second string argument is specified, chr(32) (' ') will be added.	• string lpad(string, int) • string lpad(string, int, string)
rpad	adds whitespace to the end of a string. A second string argument specifies the whitespace which will be added repeatedly until the string has reached the intended length. If no second string argument is specified, chr(32) (' ') will be added.	• string rpad(string, int) • string rpad(string, int, string)
Mathematical Functions	The math functions described here generally operate on floating point values; their inputs will automatically convert to double, the output will also be a double.	

Name	Description	Syntax
• double log(double) • double exp(double) • double log10(double) • double sin(double) • double cos(double) • double tan(double) • double asin(double) • double acos(double) • double atan(double) • double sinh(double) • double cosh(double) • double floor(double) • double ceil(double)	These functions have the same functionality as in the *C*programming language.	
sign	sign returns -1, 0 or 1 depending on the sign of its argument. Sign is implemented for all numeric types, date, and time.	• int sign(double), etc. • int sign(date) • int sign(time)
abs	Abs returns arg, if arg is positive or zero, -arg else. Abs is implemented for all numeric types and time.	• int abs(int). • double abs(double) • decfloat abs(decfloat) • time abs(time)
Date Functions	Functions operating on date or time data	
utctolocal	interpret datearg (a date, without timezone) as utc and convert it to the timezone named by timezonearg (a string)	iutctolocal(datearg, timezonearg)
localtoutc	convert the local datetime datearg to the timezone specified by the string timezonearg, return as a date	localtoutc(datearg, timezonearg)
weekday	returns the weekday as an integer in the range 0..6, 0 is Monday.	weekday(date)
now	returns the current date and time (localtime of the server timezone) as date	now()
daysbetween	returns the number of days (integer) between date1 and date2. This is an alternative to date2 - date1	daysbetween(date1, date2)
Further Functions		
if	return arg2 if intarg is considered true (not equal to zero), else return arg3. Currently, no shortcut evaluation is implemented, meaning that both arg2 and arg3 are evaluated in any case. This means you cannot use if to avoid a divide by zero error which has the side effect of terminating expression evaluation when it occurs.	if(intarg, arg2, arg3)

Name	Description	Syntax
case	return value1 if arg1 == cmp1, value2 if arg1 == cmp2 etc, default if there no match	• case(arg1, default) • case(arg1, cmp1, value1, cmp2, value2, ..., default)
isnull	return 1 (= true), if arg1 is set to null and null checking is on during evaluator run	isnull(arg1)
rownum	returns the number of the row in the currently scanned table structure. The first row has number 0	rownum()

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

> **Note**
>
> `date` is based on the proleptic Gregorian calendar. `daydate` is based on the Gregorian calendar which is also the calendar used by SAP HANA SQL.

[2] These Calculation Engine string functions operate using single byte characters. To use these functions with multi-byte character strings please see section: Using String Functions with Multi-byte Character Encoding below. Note, this limitation does not exist for the SQL functions of the SAP HANA database which support Unicode encoded strings natively.

8.2.5.1 Using String Functions with Multi-byte Character Encoding

To allow the use of the string functions of Calculation Engine with multi-byte character encoding you can use the `charpos` and `chars` (see table above for syntax of these commands) functions. An example of this usage for the single byte character function `midstr` follows below:-

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

8.2.6 CE_AGGREGATION

Syntax:

```
CE_AGGREGATION (<var_table>, <aggregate_list> [, <group_columns>]);
```

Syntax elements:

```
<var_table>  ::= :<identifier>
```

A variable of type table containing the data that should be aggregated.

> **Note**
>
> `CE_AGGREGATION` cannot handle tables directly as input.

```
<aggregate_list> ::= '['<aggregate_exp>[{, <aggregate_exp>}] ']'
```

Specifies a list of aggregates. For example, `[SUM ("A"), MAX("B")]` specifies that in the result, column "A" has to be aggregated using the SQL aggregate `SUM` and for column B, the maximum value should be given.

```
<aggregate_exp> ::= <aggregate_func>(<aggregate_column>[AS <column_alias>])
```

Specifies the required aggregation expression.

```
<aggregate_func> ::= COUNT | SUM | MIN | MAX
```

Specifies the aggregation function to use. Supported aggregation functions are:

- `count("column")`
- `sum("column")`
- `min("column")`
- `max("column")`
- use `sum("column") / count("column")` to compute the average

```
<aggregate_column> ::= <string_literal>
```

The identifier for the aggregation column.

```
<column_alias> ::= <string_literal>
```

Specifies an alias for the aggregate column.

```
<group_columns> ::= '['<group_column_name> [{,<group_column_name>}...]']'
```

Specifies an optional list of group-by attributes. For instance, `["C"]` specifies that the output should be grouped by column C. Note that the resulting schema has a column named C in which every attribute value from the input table appears exactly once. If this list is absent the entire input table will be treated as a single group, and the aggregate function is applied to all tuples of the table.

```
<group_column_name> ::= <identifier>
```

Specifies the name of the column attribute for the results to be grouped by.

> **Note**
>
> `CE_AGGREGATION` implicitly defines a projection: All columns that are not in the list of aggregates, or in the group-by list, are not part of the result.

Description:

Groups the input and computes aggregates for each group.

The result schema is derived from the list of aggregates, followed by the group-by attributes. The order of the returned columns is defined by the order of columns defined in these lists. The attribute names are:

- For the aggregates, the default is the name of the attribute that is aggregated.
- For instance, in the example above `([SUM("A"),MAX("B")])`, the first column is called A and the second is B.
- The attributes can be renamed if the default is not appropriate.
- For the group-by attributes, the attribute names are unchanged. They cannot be renamed using `CE_AGGREGATION`.

> **Note**
>
> Note that `count(*)` can be achieved by doing an aggregation on any integer column; if no group-by attributes are provided, this counts all non-null values.

8.2.7 CE_UNION_ALL

Syntax:

```
CE_UNION_ALL (<var_table1>, :var_table2)
```

Syntax elements:

```
<var_table1> ::= :<identifier>
<var_table2> ::= :<identifier>
```

Specifies the table variables to be used to form the union.

Description:

The `CE_UNION_ALL` function is semantically equivalent to `SQL UNION ALL` statement. It computes the union of two tables which need to have identical schemas. The `CE_UNION_ALL` function preserves duplicates, so the result is a table which contains all the rows from both input tables.

8.3 Special Operators

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

8.3.1 CE_VERTICAL_UNION

Syntax:

```
CE_VERTICAL_UNION(<var_table>, <projection_list> [{,<var_table>,
<projection_list>}...])
```

Syntax elements:

```
<var_table> ::= :<identifier>
```

Specifies a table variable containing a column for the union.

```
<projection_list> ::= '[' <attrib_name>[{, <attrib_name> }…] ']'
<attrib_name>     ::= <string_literal>  [AS <column_alias>]
<column_alias>    ::= <string_literal>
```

Specifies a list of attributes that should be in the resulting table. The list must at least have one element. The attributes can be renamed using the SQL keyword `AS`.

Description:

For each input table variable the specified columns are concatenated. Optionally columns can be renamed. All input tables must have the same cardinality.

> ⚠ Caution
>
> The vertical union is sensitive to the order of its input. SQL statements and many calculation engine plan operators may reorder their input or return their result in different orders across starts. This can lead to unexpected results.

.

8.3.2 CE_CONVERSION

Syntax:

```
CE_CONVERSION(<var_table>, <conversion_params>, [<rename_clause>])
```

Syntax elements:

```
<var_table> ::= :<identifier>
```

Specifies a table variable to be used for the conversion.

```
<conversion_params> ::= '['<key_val_pair>[{,<key_val_pair>}...]']'
```

Specifies the parameters for the conversion. The `CE_CONVERSION`operator is highly configurable via a list of key-value pairs. For the exact conversion parameters permissible, see the *Conversion parameters* table.

```
<key_val_pair> ::= <key> = <value>
```

Specify the key and value pair for the parameter setting.

```
<key> ::= <identifier>
```

Specifies the parameter key name.

```
<value> ::= <string_literal>
```

Specifies the parameter value.

```
<rename_clause> ::= <rename_att>[{,<rename_att>}]
```

Specifies new names for the result columns.

```
<rename_att>     ::= <convert_att> AS <new_param_name>
<convert_att>    ::= <identifier>
<new_param_name> ::= <identifier>
```

Specifies the new name for a result column.

Description:

Applies a unit conversion to input table **<var_table>** and returns the converted values. Result columns can optionally be renamed. The following syntax depicts valid combinations. Supported keys with their allowed domain of values are:

Table 18: Conversion parameters

Key	Values	Type	Mandatory	Default	Documentation
'family'	'currency'	key	Y	none	The family of the conversion to be used.
'method'	'ERP'	key	Y	none	The conversion method.
'error_handling'	'fail on error', 'set to null', 'keep unconverted'	key	N	'fail on error'	The reaction if a rate could not be determined for a row.
'output'	combinations of 'input', 'unconverted', 'converted', 'passed_through', 'output_unit', 'source_unit', 'target_unit', 'reference_date'	key	N	'converted, passed_through, output_unit'	Specifies which attributes should be included in the output.
'source_unit'	Any	Constant	N	None	The default source unit for any kind of conversion.
'target_unit'	Any	Constant	N	None	The default target unit for any kind of conversion.

Key	Values	Type	Mandatory	Default	Documentation
'reference_date'	Any	Constant	N	None	The default reference date for any kind of conversion.
'source_unit_column'	column in input table	column name	N	None	The name of the column containing the source unit in the input table.
'target_unit_column'	column in input table	column name	N	None	The name of the column containing the target unit in the input table.
'reference_date_column'	column in input table	column name	N	None	The default reference date for any kind of conversion.
'output_unit_column'	Any	column name	N	"OUTPUT_UNIT"	The name of the column containing the target unit in the output table.

For ERP conversion specifically:

Table 19:

Key	Values	Type	Mandatory	Default	
'client'	Any	Constant		None	The client as stored in the tables.
'conversion_type'	Any	Constant		'M'	The conversion type as stored in the tables.
'schema'	Any	schema name		current schema	The default schema in which the conversion tables should be looked-up.

8.3.3 TRACE

Syntax:

```
TRACE(<var_input>)
```

Syntax elements:

```
<var_input>  ::= :<identifier>
```

Identifies the SQLScript variable to be traced.

Description:

The TRACE operator is used to debug SQLScript procedures. It traces the tabular data passed as its argument into a local temporary table and returns its input unmodified. The names of the temporary tables can be retrieved from the `SYS.SQLSCRIPT_TRACE` monitoring view. See SQLSCRIPT_TRACE below.

Example:

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

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

> **Note**
>
> This operator should not be used in production code as it will cause significant runtime overhead. Additionally, the naming conventions used to store the tracing information may change. Thus, this operator should only be used during development for debugging purposes.

Related Information

SQLSCRIPT_TRACE

9 Best Practices for Using SQLScript

So far this document has introduced the syntax and semantics of SQLScript. This knowledge is sufficient for mapping functional requirements to SQLScript procedures. However, besides functional correctness, non-functional characteristics of a program play an important role for user acceptance. For instance, one of the most important non-functional characteristics is performance.

The following optimizations all apply to statements in SQLScript. The optimizations presented here cover how dataflow exploits parallelism in the SAP HANA database.

- Reduce Complexity of SQL Statements: Break up a complex SQL statement into many simpler ones. This makes a SQLScript procedure easier to comprehend.
- Identify Common Sub-Expressions: If you split a complex query into logical sub queries it can help the optimizer to identify common sub expressions and to derive more efficient execution plans.
- Multi-Level-Aggregation: In the special case of multi-level aggregations, SQLScript can exploit results at a finer grouping for computing coarser aggregations and return the different granularities of groups in distinct table variables. This could save the client the effort of reexamining the query result.
- Understand the Costs of Statements: Employ the explain plan facility to investigate the performance impact of different SQL queries.
- Exploit Underlying Engine: SQLScript can exploit the specific capabilities of the OLAP- and JOIN-Engine by relying on views modeled appropriately.
- Reduce Dependencies: As SQLScript is translated into a dataflow graph, and independent paths in this graph can be executed in parallel, reducing dependencies enables better parallelism, and thus better performance.
- Avoid Mixing Calculation Engine Plan Operators and SQL Queries: Mixing calculation engine plan operators and SQL may lead to missed opportunities to apply optimizations as calculation engine plan operators and SQL statements are optimized independently.
- Avoid Using Cursors: Check if use of cursors can be replaced by (a flow of) SQL statements for better opportunities for optimization and exploiting parallel execution.
- Avoid Using Dynamic SQL: Executing dynamic SQL is slow because compile time checks and query optimization must be done for every invocation of the procedure. Another related problem is security because constructing SQL statements without proper checks of the variables used may harm security.

9.1 Reduce Complexity of SQL Statements

Best Practices: Reduce Complexity of SQL Statements

Variables in SQLScript enable you to arbitrarily break up a complex SQL statement into many simpler ones. This makes a SQLScript procedure easier to comprehend. To illustrate this point, consider the following query:

```
 books_per_publisher = SELECT publisher, COUNT (*) AS cnt
FROM :books GROUP BY publisher;
largest_publishers = SELECT * FROM :books_per_publisher
WHERE cnt >= (SELECT MAX (cnt)
```

```
FROM :books_per_publisher);
```

Writing this query as a single SQL statement either requires the definition of a temporary view (using WITH) or repeating a sub query multiple times. The two statements above break the complex query into two simpler SQL statements that are linked via table variables. This query is much easier to comprehend because the names of the table variables convey the meaning of the query and they also break the complex query into smaller logical pieces.

The SQLScript compiler will combine these statements into a single query or identify the common sub-expression using the table variables as hints. The resulting application program is easier to understand without sacrificing performance.

9.2 Identify Common Sub-Expressions

Best Practices: Identify Common Sub-Expressions

The query examined in the previous sub section contained common sub-expressions. Such common sub-expressions might introduce expensive repeated computation that should be avoided. For query optimizers it is very complicated to detect common sub-expressions in SQL queries. If you break up a complex query into logical sub queries it can help the optimizer to identify common sub-expressions and to derive more efficient execution plans. If in doubt, you should employ the EXPLAIN plan facility for SQL statements to investigate how the HDB treats a particular statement.

9.3 Multi-level Aggregation

Best Practices: Multi-level Aggregation

Computing multi-level aggregation can be achieved using grouping sets. The advantage of this approach is that multiple levels of grouping can be computed in a single SQL statement.

```
SELECT publisher, name, year, SUM(price)
 FROM :it_publishers, :it_books
WHERE publisher=pub_id AND crcy=:currency
GROUP BY GROUPING SETS ((publisher, name, year), (year))
```

To retrieve the different levels of aggregation the client typically has to examine the result repeatedly, for example by filtering by NULL on the grouping attributes.

In the special case of multi-level aggregations, SQLScript can exploit results at a finer grouping for computing coarser aggregations and return the different granularities of groups in distinct table variables. This could save the client the effort of reexamining 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;
```

It is a matter of developer choice and also the application requirements as to which alternative is the best fit for the purpose.

9.4 Understand the Costs of Statements

It is important to keep in mind that even though the SAP HANA database is an in-memory database engine and that the operations are fast, each operation has its associated costs and some are much more costly than others.

As an example, calculating a UNION ALL of two result sets is cheaper than calculating a UNION of the same result sets because of the duplicate elimination the UNION operation performs. The calculation engine plan operator CE_UNION_ALL (and also UNION ALL) basically stacks the two input tables over each other by using references without moving any data within the memory. Duplicate elimination as part of UNION, in contrast, requires either sorting or hashing the data to realize the duplicate removal, and thus a materialization of data. Various examples similar to these exist. Therefore it is important to be aware of such issues and, if possible, to avoid these costly operations.

You can get the query plan from the view SYS.QUERY_PLANS. The view is shared by all users. Here is an example of reading a query plan from the view.

```
EXPLAIN PLAN [ SET PLAN_ID = <plan_id> ] FOR <dml_stmt>
SELECT lpad(' ', level) || operator_name AS operator_name,
       operator_details, object_name, subtree_cost,
       input_cardinality, output_cardinality, operator_id,
       parent_operator_id, level, position
       FROM sys.query_plans
       WHERE PLAN_ID = <plan_id> ORDER BY operator_id;
```

Sometimes alternative formulations of the same query can lead to faster response times. Consequently reformulating performance critical queries and examining their plan may lead to better performance.

The SAP HANA database provides a library of application-level functions which handle frequent tasks, e.g. currency conversions. These functions can be expensive to execute, so it makes sense to reduce the input as much as possible prior to calling the function.

9.5 Exploit Underlying Engine

Best Practices: Exploit Underlying Engine

SQLScript can exploit the specific capabilities of the built-in functions or SQL statements. For instance, if your data model is a star schema, it makes sense to model the data as an Analytic view. This allows the SAP HANA database to exploit the star schema when computing joins producing much better performance.

Similarly, if the application involves complex joins, it might make sense to model the data either as an Attribute view or a Graphical Calculation view. Again, this conveys additional information on the structure of the data which is exploited by the SAP HANA database for computing joins. When deciding to use Graphical Calculation

views involving complex joins refer to SAP note 1857202 for details on how, and under which conditions, you may benefit from SQL Engine processing with Graphical Calculation views.

Using CE functions only, or alternatively SQL statements only, in a procedure allows for many optimizations in the underlying database system. However when SQLScript procedures using imperative constructs are called by other programs, for example predicates to filter data early, can no longer be applied. The performance impact of using these constructs must be carefully analyzed when performance is critical.

Finally, note that not assigning the result of an SQL query to a table variable will return the result of this query directly to the client as a result set. In some cases the result of the query can be streamed (or pipelined) to the client. This can be very effective as this result does not need to be materialized on the server before it is returned to the client.

9.6 Reduce Dependencies

Best Practices: Reduce Dependencies

One of the most important methods for speeding up processing in the SAP HANA database is a massive parallelization of executing queries. In particular, parallelization is exploited at multiple levels of granularity: For example, the requests of different users can be processed in parallel, and also single relational operators within a query are executed on multiple cores in parallel. It is also possible to execute different statements of a single SQLScript in parallel if these statements are independent of each other. Remember that SQLScript is translated into a dataflow graph, and independent paths in this graph can be executed in parallel.

From an SQLScript developer perspective, we can support the database engine in its attempt to parallelize execution by avoiding unnecessary dependencies between separate SQL statements, and also by using declarative constructs if possible. The former means avoiding variable references, and the latter means avoiding imperative features, for example cursors.

9.7 Avoid Mixing Calculation Engine Plan Operators and SQL Queries

Best Practices: Avoid Mixing Calculation Engine Plan Operators and SQL Queries

The semantics of relational operations as used in SQL queries and calculation engine operations are different. In the calculation engine operations will be instantiated by the query that is executed on top of the generated data flow graph.

Therefore the query can significantly change the semantics of the data flow graph. For example consider a calculation view that is queried using attribute publisher (but not year) that contains an aggregation node (CE_AGGREGATION) which is defined on publisher and year. The grouping on year would be removed from the grouping. Evidently this reduces the granularity of the grouping, and thus changes the semantics of the model. On the other hand, in a nested SQL query containing a grouping on publisher and year this aggregation-level would not be changed if an enclosed query only queries on publisher.

Because of the different semantics outlined above, the optimization of a mixed data flow using both types of operations is currently limited. Hence, one should avoid mixing both types of operations in one procedure.

9.8 Avoid Using Cursors

Best Practices: Avoid Using Cursors

While the use of cursors is sometime required, they imply row-at-a-time processing. As a consequence, opportunities for optimizations by the SQL engine are missed. So you should consider replacing the use of cursors with loops, by SQL statements as follows:

Read-Only Access

For read-only access to a cursor consider using simple selects or join:

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS
Reads SQL DATA
BEGIN
    DECLARE val decimal(34,10) := 0;
    DECLARE CURSOR c_cursor1 FOR
    SELECT isbn, title, price FROM books;
    FOR r1 AS c_cursor1 DO
    val := :val + r1.price;
    END FOR;
END;
```

This sum can also be computed by the SQL engine:

```
SELECT sum(price) into val FROM books;
```

Computing this aggregate in the SQL engine may result in parallel execution on multiple CPUs inside the SQL executor.

Updates and Deletes

For updates and deletes, consider using the

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE val INT := 0;
    DECLARE CURSOR c_cursor1 FOR
    SELECT isbn, title, price FROM books;
    FOR r1 AS c_cursor1 DO
        IF r1.price > 50 THEN
            DELETE FROM Books WHERE isbn = r1.isbn;
        END IF;
    END FOR;
END;
```

This delete can also be computed by the SQL engine:

```
DELETE FROM Books
WHERE isbn IN (SELECT isbn FROM books WHERE price > 50);
```

Computing this in the SQL engine reduces the calls through the runtime stack of HDB and potentially benefits from internal optimizations like buffering or parallel execution.

Insertion into Tables

```
CREATE PROCEDURE foreach_proc LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE val INT := 0;
    DECLARE CURSOR c_cursor1 FOR SELECT isbn, title, price FROM books;
    FOR r1 AS c_cursor1 DO
        IF r1.price > 50
        THEN
            INSERT INTO ExpensiveBooks VALUES(..., r1.title, ...);
        END IF;
    END FOR;
END;
```

This insertion can also be computed by the SQL engine:

```
 SELECT ..., title, ... FROM Books WHERE price > 50
INTO ExpensiveBooks;
```

Similar to updates and deletes, computing this statement in the SQL engine reduces the calls through the runtime stack of the SAP HANA database, and potentially benefits from internal optimizations like buffering or parallel execution.

9.9 Avoid Using Dynamic SQL

Best Practices: Avoid using Dynamic SQL

Dynamic SQL is a powerful way to express application logic. It allows for constructing SQL statements at execution time of a procedure. However, executing dynamic SQL is slow because compile time checks and query optimization must be done for every start up of the procedure. So when there is an alternative to dynamic SQL using variables, this should be used instead.

Another related problem is security because constructing SQL statements without proper checks of the variables used can create a security vulnerability, for example, SQL injection. Using variables in SQL statements prevents these problems because type checks are performed at compile time and parameters cannot inject arbitrary SQL code.

Summarizing potential use cases for dynamic SQL are:

Table 20: Dynamic SQL Use Cases

Feature	Proposed Solution
Projected attributes	Dynamic SQL
Projected literals	SQL + variables
`FROM` clause	SQL + variables; result structure must remain unchanged

Feature	Proposed Solution
`WHERE` clause – attribute names & Boolean operators	`APPLY_FILTER`

10 Developing Applications with SQLScript

This section contains information about creating applications with SQLScript for SAP HANA.

10.1 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.

10.2 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

10.3 Calling SQLScript From Clients

In this document we have discussed the syntax for creating SQLScript procedures and calling them. Besides the SQL command console for invoking a procedure, calls to SQLScript will also be embedded into client code. In this section we present examples how this can be done.

10.3.1 Calling SQLScript from ABAP

Using CALL DATBASE PROCEDURE

The best way to call *SQLScript* from *ABAP* is to create a procedure proxy which can be natively called from *ABAP* by using the built in command `CALL DATABASE PROCEDURE`.

The *SQLScript* procedure has to be created normally in the SAP HANA Studio with the HANA Modeler. After this a procedure proxy can be creating using the ABAP Development Tools for Eclipse. In the procedure proxy the type mapping between ABAP and HANA data types can be adjusted. The procedure proxy is transported normally with the ABAP transport system while the HANA procedure may be transported within a delivery unit as a *TLOGO* object.

Calling the procedure in ABAP is very simple. The example below shows calling a procedure with two inputs (one scalar, one table) and one (table) output parameter:

```
CALL DATABASE PROCEDURE z_proxy
EXPORTING   iv_scalar = lv_scalar
            it_table  = lt_table
IMPORTING   et_table1 = lt_table_res.
```

Using the connection clause of the `CALL DATABASE PROCEDURE` command, it is also possible to call a database procedure using a secondary database connection. Please consult the ABAP help for detailed instructions of how to use the `CALL DATABASE PROCEDURE` command and for the exceptions may be raised.

It is also possible to create procedure proxies with an ABAP API programmatically. Please consult the documentation of the class `CL_DBPROC_PROXY_FACTORY` for more information on this topic.

Using ADBC

```
REPORT  ZRS_NATIVE_SQLSCRIPT_CALL.
PARAMETERS:
  con_name TYPE dbcon-con_name default 'DEFAULT'.
TYPES:
  BEGIN OF result_t,
    key     TYPE i,
    value   TYPE string,
  END OF result_t.
```

```
DATA:
  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,
  d_ref      TYPE REF TO DATA,
  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( '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'' )' ).
    CATCH cx_sql_exception.
    ENDTRY.
* Create test procedure
    TRY.
      stmt_ref->execute_ddl( 'DROP PROCEDURE zrs_testproc' ).
    CATCH cx_sql_exception.
    ENDTRY.
    TRY.
      stmt_ref->execute_ddl( 'DROP VIEW zrs_testproc_view' ).
    CATCH cx_sql_exception.
    ENDTRY.
    stmt_ref->execute_ddl( 'CREATE PROCEDURE zrs_testproc( OUT t1
zrs_testproc_tab ) READS SQL DATA WITH RESULT VIEW zrs_testproc_view AS BEGIN t1
= select * from zrs_testproc_tab; end' ).
* 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' ).
      stmt_ref->execute_ddl( 'CREATE GLOBAL TEMPORARY COLUMN TABLE
zrs_testproc_p1( key int, value NVARCHAR(255) )' ).
    CATCH cx_sql_exception.
    ENDTRY.
************************************
** Execution time
************************************
    PERFORM execute_with_transfer_table.
    PERFORM execute_with_result_view.
    con_ref->close( ).
  CATCH cx_sql_exception INTO sqlerr_ref.
     PERFORM handle_sql_exception USING sqlerr_ref.
  ENDTRY.
FORM execute_with_result_view.
  clear result_tab.
* execute procedure call by selecting from the result view
* additional input parameters have to be passed in via the WITH PARAMETERS clause
    res_ref = stmt_ref->execute_query( 'SELECT * FROM zrs_testproc_view' ).
* set output table
    GET REFERENCE OF result_tab INTO d_ref.
    res_ref->set_param_table( d_ref ).
* get the complete result set in the internal table
    row_cnt = res_ref->next_package( ).
```

```
    write: / 'EXECUTE WITH RESULT VIEW: row count: ', row_cnt.
ENDFORM.
FORM execute_with_transfer_table.
  clear result_tab.
* 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' ).
* set output table
    GET REFERENCE OF result_tab INTO d_ref.
    res_ref->set_param_table( d_ref ).
* get the complete result set in the internal table
    row_cnt = res_ref->next_package( ).
    write: / 'EXECUTE WITH TRANSFER TABLE: row count: ', row_cnt.
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.
```

10.3.2 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();
}
```

10.3.3 Calling SQLScript from C#

Given procedure:

```
CREATE PROCEDURE TEST_PRO1(IN strin NVARCHAR(100), OUT SorP NVARCHAR(100))
language sqlscript AS
BEGIN
    select 10 from dummy;
    SorP := N'input str is ' || strin;
END;
```

This procedure can be called as follows:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.Common;
using ADODB;
using System.Data.SqlClient;
namespace NetODBC
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                DbConnection conn;
                DbProviderFactory _DbProviderFactoryObject;
                String connStr = "DRIVER={HDBODBC32};UID=SYSTEM;PWD=<password>;
                                  SERVERNODE=<host>:<port>;DATABASE=SYSTEM";
                String ProviderName = "System.Data.Odbc";
                _DbProviderFactoryObject =
DbProviderFactories.GetFactory(ProviderName);
                conn = _DbProviderFactoryObject.CreateConnection();
                conn.ConnectionString = connStr;
                conn.Open();
                System.Console.WriteLine("Connect to HANA database
successfully");
                DbCommand cmd = conn.CreateCommand();
                //call Stored Procedure
                cmd = conn.CreateCommand();
                cmd.CommandText = "call SqlScriptDocumentation.scalar_proc (?)";
                DbParameter inParam = cmd.CreateParameter();
                inParam.Direction = ParameterDirection.Input;
                inParam.Value = "asc";
                cmd.Parameters.Add(inParam);
                DbParameter outParam = cmd.CreateParameter();
                outParam.Direction = ParameterDirection.Output;
                outParam.ParameterName = "a";
                outParam.DbType = DbType.Integer;
                cmd.Parameters.Add(outParam);
```

```
                reader = cmd.ExecuteReader();
                System.Console.WriteLine("Out put parameters = " +
outParam.Value);
                reader.Read();
                String row1 = reader.GetString(0);
                System.Console.WriteLine("row1=" + row1);
            }
            catch(Exception e)
            {
                System.Console.WriteLine("Operation failed");
                System.Console.WriteLine(e.Message);
            }
        }
    }
}
```

11 Appendix

11.1 Example code snippets

The examples used throughout this manual make use of various predefined code blocks. These code snippets are presented below.

11.1.1 ins_msg_proc

This code is used in the examples in this reference manual to store outputs so the action of the examples can be seen. It simple stores some text along with a timestamp of the entry.

Before you can use this procedure you must create the following table.

```
CREATE TABLE message_box (p_msg VARCHAR(200), tstamp TIMESTAMP);
```

You can create the procedure as follows.

```
CREATE PROCEDURE ins_msg_proc (p_msg VARCHAR(200)) LANGUAGE SQLSCRIPT AS
BEGIN
    INSERT INTO message_box VALUES (:p_msg, CURRENT_TIMESTAMP);
END;
```

To view the contents of the message_box you select the messages in the table.

```
select * from message_box;
```

Important Disclaimer for Features in SAP HANA Options

There are several types of licenses available for SAP HANA. Depending on the license type of your SAP HANA installation, some of the features and tools that are described in the SAP HANA platform documentation may only be available via the SAP HANA options, which may be released independently of an SAP HANA Platform Support Package Stack (SPS). Although various features included in SAP HANA options are cited in the SAP HANA platform documentation, customers who only purchased the license for the base edition of the SAP HANA platform do not have the right to use features included in SAP HANA options, because these features are not included in the license of the base edition of the SAP HANA platform. For customers to whom these license restrictions apply, the use of features included in SAP HANA options in a production system requires purchasing the corresponding software license(s) from SAP. The documentation for the SAP HANA optional components is available in SAP Help Portal at http://help.sap.com/hana_options. For more information, see also SAP Note 2091815 - SAP HANA Options If you have additional questions about what your particular license provides, or wish to discuss licensing features available in SAP HANA options, please contact your SAP account team representative.

Important Disclaimers and Legal Information

Coding Samples

Any software coding and/or code lines / strings ("Code") included in this documentation are only examples and are not intended to be used in a productive system environment. The Code is only intended to better explain and visualize the syntax and phrasing rules of certain coding. SAP does not warrant the correctness and completeness of the Code given herein, and SAP shall not be liable for errors or damages caused by the usage of the Code, unless damages were caused by SAP intentionally or by SAP's gross negligence.

Accessibility

The information contained in the SAP documentation represents SAP's current view of accessibility criteria as of the date of publication; it is in no way intended to be a binding guideline on how to ensure accessibility of software products. SAP in particular disclaims any liability in relation to this document. This disclaimer, however, does not apply in cases of wilful misconduct or gross negligence of SAP. Furthermore, this document does not result in any direct or indirect contractual obligations of SAP.

Gender-Neutral Language

As far as possible, SAP documentation is gender neutral. Depending on the context, the reader is addressed directly with "you", or a gender-neutral noun (such as "sales person" or "working days") is used. If when referring to members of both sexes, however, the third-person singular cannot be avoided or a gender-neutral noun does not exist, SAP reserves the right to use the masculine form of the noun and pronoun. This is to ensure that the documentation remains comprehensible.

Internet Hyperlinks

The SAP documentation may contain hyperlinks to the Internet. These hyperlinks are intended to serve as a hint about where to find related information. SAP does not warrant the availability and correctness of this related information or the ability of this information to serve a particular purpose. SAP shall not be liable for any damages caused by the use of related information unless damages have been caused by SAP's gross negligence or willful misconduct. All links are categorized for transparency (see: http://help.sap.com/disclaimer).

www.sap.com/contactsap