SAP HANA Database – SQLScript Guide

- SAP HANA Appliance Software SPS 03

Target Audience

- Consultants
- Administrators

Public

Document version 1.0 – 2011/12/22

THE BEST-RUN Bbusinesses RUN SAP

SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf
Germany
T +49/18 05/34 34 24
F +49/18 05/34 34 20
www.sap.com

warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

Disclaimer

Some components of this product are based on Java™. Any code change in these components may cause unpredictable and severe malfunctions and is therefore expressively prohibited, as is any decompilation of these components.

Any Java™ Source Code delivered with this product is only to be used by SAP's Support Services and may not be modified or altered in any way.

Documentation in the SAP Service Marketplace

You can find this documentation at the following address:
`http://service.sap.com/hana`

Terms for Included Open Source Software

This SAP software contains also the third party open source software products listed below. Please note that for these third party products the following special terms and conditions shall apply.

1. This software was developed using ANTLR.

2. gSOAP

Part of the software embedded in this product is gSOAP software. Portions created by gSOAP are Copyright (C) 2001-2004 Robert A. van Engelen, Genivia inc. All Rights Reserved.

THE SOFTWARE IN THIS PRODUCT WAS IN PART PROVIDED BY GENIVIA INC AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

3. SAP License Agreement for STLport SAP License Agreement for STLPort between SAP Aktiengesellschaft Systems, Applications, Products in Data Processing Neurottstrasse 16 69190 Walldorf, Germany (hereinafter: SAP) and you (hereinafter: Customer)

a) Subject Matter of the Agreement

A) SAP grants Customer a non-exclusive, non-transferrable, royalty-free license to use the STLport.org C++ library (STLport) and its documentation without fee.

B) By downloading, using, or copying STLport or any portion thereof Customer agrees to abide by the intellectual property laws, and to all of the terms and conditions of this Agreement.

C) The Customer may distribute binaries compiled with STLport (whether original or modified) without any royalties or restrictions.

D) Customer shall maintain the following copyright and permissions notices on STLport sources and its documentation unchanged:

Copyright 2001 SAP AG

E) The Customer may distribute original or modified STLport sources, provided that:

o The conditions indicated in the above permissions notice are met;

o The following copyright notices are retained when present, and conditions provided in accompanying permission notices are met:

Coypright 1994 Hewlett-Packard Company

Copyright 1996,97 Silicon Graphics Computer Systems Inc.

Copyright 1997 Moscow Center for SPARC Technology.

Copyright 1999,2000 Boris Fomitchev

Copyright 2001 SAP AG

Permission to use, copy, modify, distribute and sell this software and its documentation for any purposes is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purposes is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Moscow Center for SPARC makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

Boris Fomitchev makes no representations about the suitability of this software for any purpose. This material is provided "as is", with absolutely no warranty expressed or implied.

Any use is at your own risk. Permission to use or copy this software for any purpose is hereby granted without fee, provided the above notices are retained on all copies.

Permission to modify the code and to distribute modified code is granted, provided the above notices are retained, and a notice that the code was modified is included with the above copyright notice.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purposes is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. SAP makes no representations about the suitability of this software for any purpose. It is provided with a limited warranty and liability as set forth in the License Agreement distributed with this copy.

SAP offers this liability and warranty obligations only towards its customers and only referring to its modifications.

b) Support and Maintenance SAP does not provide software maintenance for the STLport. Software maintenance of the STLport therefore shall be not included.

All other services shall be charged according to the rates for services quoted in the SAP List of Prices and Conditions and shall be subject to a separate contract.

c) Exclusion of warranty

As the STLport is transferred to the Customer on a loan basis and free of charge, SAP cannot guarantee that the STLport is error-free, without material defects or suitable for a specific application under third-party rights. Technical data, sales brochures, advertising text and quality descriptions produced by SAP do not indicate any assurance of particular attributes.

d) Limited Liability

A) Irrespective of the legal reasons, SAP shall only be liable for damage, including unauthorized operation, if this (i) can be compensated under the Product Liability Act or (ii) if caused due to gross negligence or intent by SAP or (iii) if based on the failure of a guaranteed attribute.

B) If SAP is liable for gross negligence or intent caused by employees who are neither agents or managerial employees of SAP, the total liability for such damage and a maximum limit on the scope of any such damage shall depend on the extent to which its occurrence ought to have anticipated by SAP when concluding the contract, due to the circumstances known to it at that point in time representing a typical transfer of the software.

C) In the case of Art. 4.2 above, SAP shall not be liable for indirect damage, consequential damage caused by a defect or lost profit.

D) SAP and the Customer agree that the typical foreseeable extent of damage shall under no circumstances exceed EUR 5,000.

E) The Customer shall take adequate measures for the protection of data and programs, in particular by making backup copies at the minimum intervals recommended by SAP. SAP shall not be liable for the loss of data and its recovery, notwithstanding the other limitations of the present Art. 4 if this loss could have been avoided by observing this obligation.

F) The exclusion or the limitation of claims in accordance with the present Art. 4 includes claims against employees or agents of SAP.

4. Adobe Document Services Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and / or other countries. For information on Third Party software delivered with Adobe document services and Adobe LiveCycle Designer, see SAP Note 854621

SAP HANA Database SQLScript Guide

TABLE OF CONTENTS

Abstract

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.

This document describes SQLScript for SAP HANA appliance SPS 03 (denoted as SQLScript V3). With version 3 of the language various syntax constructs deprecated in SQLScript V2 have been removed and replaced. This document also contains an update with new language constructs and more details on debugging SQLScript.

NOTE: There are syntactical changes introduced with SQLScript V3. The old syntax will still be supported until SAP HANA SPS 04, but will lead to a deprecated warning in the server log files and also in applications through client interfaces. Moreover, syntax that was deprecated in SPS2 will now lead to compiler or runtime errors. Please make sure you update your code syntax accordingly.

This SQLScript Guide is related to the following SAP HANA Guides:

Topic	**Guide**	**Quick Link**
SAP HANA Landscape Deployment & Installation	SAP HANA Knowledge Center on SAP Service Marketplace	https://service.sap.com/hana • SAP HANA Master Guide • SAP HANA Installation Guide
SAP HANA Administration & Security	SAP HANA Knowledge Center on SAP Service Marketplace	https://service.sap.com/hana • Security Guide
SAP HANA SQL Reference	SAP HANA Knowledge Center on SAP Service Marketplace	https://service.sap.com/hana • SAP HANA SQL Reference Guide
SAP HANA Modeling Guide	SAP HANA Knowledge Center on SAP Service Marketplace	https://service.sap.com/hana • SAP HANA Modeling Guide
SAP HANA Development Guide	SAP HANA Knowledge Center on SAP Service Marketplace	https://service.sap.com/hana • SAP HANA Development Guide

Motivation

The motivation for SQLScript is to embed data-intensive application logic into the database. As of today, ABAP applications only offload very limited functionality into the database using SQL. Most of the application logic is executed in the ABAP application server which means 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 (e.g. looping over rows in a table). However, these algorithms are hard to optimize and parallelize compared to declarative set-oriented languages such as SQL.

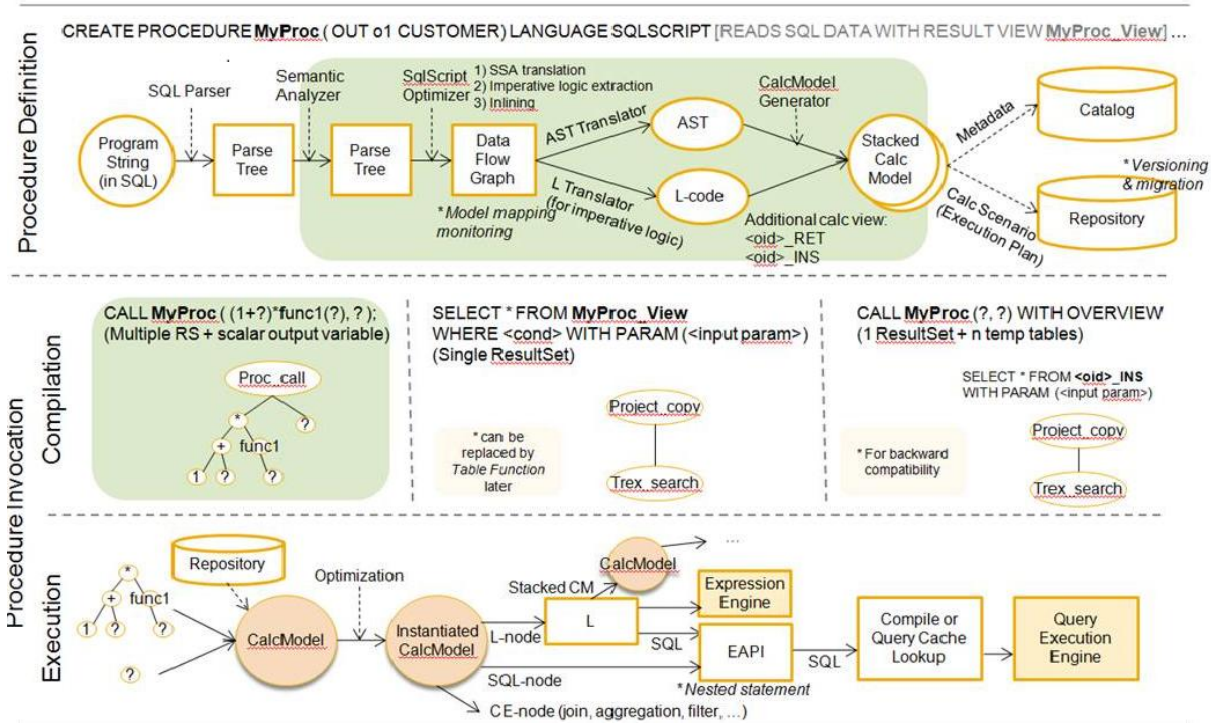
The SAP HANA database is optimized for modern technology trends (e.g. data resides in main-memory, massive-parallelization on multi-core CPUs). The goal of the SAP HANA database is to optimally support application requirements by leveraging such hardware. To this end, the SAP HANA database exposes a very sophisticated interface to the application consisting of different languages. The expressiveness of these languages far exceeds that of OpenSQL. The set of SQL extensions for the SAP HANA database which 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 shall be used in cases where other modeling constructs of SAP HANA, e.g. analytic views or attribute views are not sufficient. We refer to the SAP HANA Database – Development Guide for an extensive treatment of this topic.

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 embedded in other SQL statements.
- SQL queries do not have features to express business logic (e.g. 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, e.g. by iterative approximation algorithms. Thus it is possible to mix imperative constructs known from stored procedures with declarative ones.

SQLScript Processing Overview

In order to better understand the features of SQLScript, and their impact on the execution time of a procedure, it is instructive to understand how SQLScript procedures are processed in the SAP HANA database, see the picture below.



When a user defines a new procedure, e.g. using the CREATE PROCEDURE statement discussed later in this document, the SAP HANA database query compiler processes the statement in a similar way to an SQL statement. In the figure above, the compilation steps are represented by the arrows, and the data structures used in this process are shown as nodes in the process flow. 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.
- Code generation - For declarative logic the compiler creates calculation models to represent the data flow defined by the SQLScript code. It will be optimized further by the calculation engine, when it is instantiated. For imperative logic the code blocks are translated into L nodes[1].
- The calculation models generated in the previous step are combined into a stacked calculation model.
- The compiled procedure creates content in the database catalog and the repository.

Invocation of a procedure can be divided into two phases:

[1] „L“ is a new language developed by SAP for executing safe procedural code in SAP HANA Database.

1) Compilation - During compilation the call to the procedure is rewritten for processing by the calculation engine.
2) Execution - The execution commences with binding actual parameters to the calculation models generated n the definition phase. 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.

Orchestration-Logic

Orchestration logic is used to implement data flow and control flow logic using DDL-, DML- and SQL-Query statements as well as imperative language constructs like 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, i.e. 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 data flow 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 for an internal L-node. 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, i.e. 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.

Example for Orchestration-Logic:

```
CREATE PROCEDURE orchestrationProc LANGUAGE SQLSCRIPT AS

  v_id BIGINT;
  v_name VARCHAR(30);
  v_pmnt BIGINT;
  v_msg VARCHAR(200);
  CURSOR c_cursor1 (p_payment BIGINT) FOR
    SELECT id, name, payment FROM control_tab
```

```
    WHERE payment > :p_payment
    ORDER BY id ASC;

BEGIN

     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.

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 those prerequisites the following kinds of operators are available:

- SQL SELECT Statement
- Calculation engine plan operators (see page 22 ff)
- Custom operators provided by SAP

Logically each operator represents a node in the data flow graph. The first two kinds of operators are available with SQLScript V2. Custom operators have to be manually implemented, e.g. by SAP.

Datatype Extension

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

Scalar Datatypes

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

- `TINYINT, SMALLINT, INTEGER, BIGINT`
- `DECIMAL(p, s), REAL, FLOAT, DOUBLE`
- `VARCHAR, NVARCHAR, CLOB, NCLOB`
- `VARBINARY, BLOB`
- `DATE, TIME, TIMESTAMP`

See the SQL reference listed at the beginning of this document for further details on scalar types.

Table Type Definition

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.

NOTE:

With SQLScript V3 syntax for creating table types was deprecated. See section *Changes in Previous Versions of SQLScript* in this guide.

CREATE TYPE

Syntax:

```
CREATE TYPE {schema.}name AS TABLE (name1 type1 {, name2 type2,...})
```

Description:

The syntax for defining table types follows the SQL syntax for defining new types – also see the SQL reference for details. In order to create a table type in a different schema than the current default schema, the schema has to be provided as a prefix (e.g. `myschema.mytable` for a table type). The table type is specified using a list of attribute names and primitive data types. For each table type, attributes must have unique names.

In contrast to definitions of database tables, table types do not have an instance that is created when the table type is created (i.e., no Data Manipulation Language (DML) statements (INSERT, UPDATE, DELETE) are supported on table types).

A table type can be dropped using the DROP TYPE statement.

Example:

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

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

DROP TYPE

Syntax:

```
DROP TYPE {schema.}name {CASCADE}
```

Description:

Drop a table type created by CREATE TYPE statement – also see the SQL reference on dropping types. Dependent objects are invalidated by default. If the optional CASCADE is used, dependent objects will also be dropped.

By default, the drop statement invalidates dependent objects. For example, if a procedure uses the table type in its parameters, it will be invalidated. When a table type with the same name is created again, then previously invalidated objects become valid again. When cascade is specified, dependent objects will be dropped too.

Functional Extension

Basic Concept

In this section we discuss the declarative sublanguage of SQLScript. It allows users to describe complex dataflow logic using side-effect free procedures. Such procedures are marked as read-only procedures, and they may only use features described in this section. We will call procedures using only the functional extension (i.e. no imperative extension) as *functional-style procedures* as they follow the functional programming paradigm.

- Procedures can have multiple input parameters and output parameters (which can be of scalar types or table types).
- Procedures 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 SQLScript functional-style procedures has some advantages compared to the use of pure SQL.

- The calculation and transformations described in procedures can be parameterized and reused inside 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 will decide if a materialization strategy (which avoids re-computation 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.
- If scalar variables are required or imperative language features are required, then procedural extensions are also required.

Create Procedure

Overview

In this section we discuss the statements to create and drop SQLScript procedures. Unlike previous versions of SQLScript, the statement CREATE FUNCTION is no long available to create an SQLScript

procedure. This statement is now reserved to support creation of scalar UDFs (please contact SAP for more details).

A procedure has an arbitrary number (including zero) of output parameters of any type (see section *Datatype Extension* in this guide) to return it results. It is a callable statement, and so it can be called using a CALL statement. Read-only procedures with one table output parameter support the additional creation of a result view.

Illustration

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

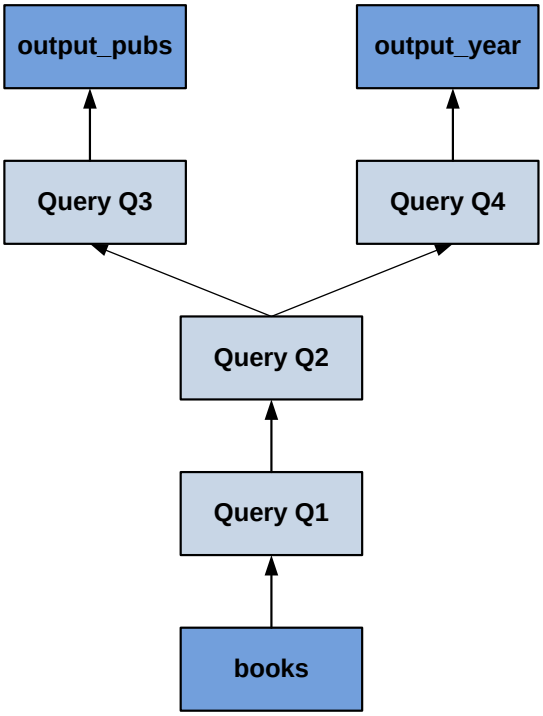
```
CREATE PROCEDURE getOutput( IN cnt INTEGER, IN currency VARCHAR(3),
                            OUT output_pubs tt_publishers,
                            OUT output_year tt_years)
     LANGUAGE SQLSCRIPT READS SQL DATA AS

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;
```

The SQLScript above defines a read-only procedure which has 2 scalar input parameters and 2 output parameters of type table. As different languages are supported in the body of the procedure, the implementation language is defined to be SQLScript. Moreover, the procedure is a read-only procedure. Hence it is free of side effects and can be called by other read-only procedures.

The first line contains a SQL query Q1 which identifies big publishers based on the number of books they have published (using the input parameter `cnt`). Afterwards, detailed information about these publishers along with their corresponding books is determined in query Q2. Finally, this information is aggregated in two 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 picture below. Similar to SQL queries, the graph is analyzed and optimized before execution. It is also possible to call a table function from within

another table function. In terms of the dataflow graph such an inner table function can be seen as a sub-graph which consumes intermediate results and returns its output to the subsequent nodes. For optimization, the sub-graph of the called function is merged with the graph of the calling function, and the resulting graph is then optimized. The optimization applies similar rules as a SQL optimizer uses for its logical optimization (e.g. filter pushdown). Afterwards, the plan is translated into a physical plan which consists of physical database operations (e.g. hash joins). The translation into a physical plan involves further optimizations using a cost model as well as heuristics.



CREATE PROCEDURE

Syntax:

```
CREATE PROCEDURE {schema.}name {({IN|OUT|INOUT}
                                 param_name data_type {,...})}
      {LANGUAGE <LANG>} {SQL SECURITY <MODE>}
      {READS SQL DATA {WITH RESULT VIEW <view_name>}} AS
BEGIN
  ...
END
```

Description:

Creates an SQLScript procedure. Any procedure in SQLScript can have a list of input and output parameters. Each parameter is marked using the keywords IN/OUT/INOUT; where IN is used if nothing is declared. Input and output parameters must be explicitly typed (i.e. no un-typed tables are supported). The input and output parameters of a procedure can have any of the primitive SQL type or a table type. INOUT parameters can only be of scalar type.

The implementation language is by default SQLSCRIPT. It is good practice to define the language in all procedure definitions. Other implementation languages are supported but are not covered in this guide.

You can specify the security mode. Privileges are always checked with the privileges of the definer of a procedure when the procedure is created. With security mode “definer”, which is the default, execution of the procedure is then performed with the privileges of the definer of the procedure. The other alternative is mode “invoker”. In this case, privileges are checked at runtime with the privileges of the caller of the function. Please note that analytical privileges are checked regardless of the security mode. See the SAP HANA Security Guide referenced at the beginning of this document for details.

Optionally, a procedure can be tagged as read only procedure using `READS SQL DATA`. This marks a procedure as being free of side-effects. One factor to be considered here is that neither DDL nor DML statements are allowed in its body, and only other read-only procedures can be called by the procedure. The advantage to this definition is that certain optimizations are only available for read-only procedures.

If a read-only procedure has exactly one table output parameter a `RESULT VIEW` can be specified. The name of the result view can be any valid SQL identifier. When a result view is defined for a procedure, it can be called from a SQL statement like a table or view reference as shown below.

Example:

```
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;
```

Normally, procedures can only be executed via the call statement. By using `WITH RESULT VIEW` it is possible to query the result of a procedure as part of a SQL statement. Please notice that no variable references are supported in the WITH PARAMETERS clause and that all constants are passed as string constants (i.e. enclosed in single quotes). Also, parameter names are lower case independent from the original capitalization in the procedure signature.

Example:

```
SELECT * FROM ProcView WITH PARAMETERS
                                    ('placeholder' = ('$$id$$', '5'))
```

You must use DROP and CREATE to modify the definition of a procedure.

NOTE:

With SAP HANA SPS 03 the name resolution of unqualified schema elements (e.g. tables or views) will be aligned with the SQL standard. Until SPS 03, unqualified schema elements were looked up in the default schema as it was defined when the procedure was created. With SPS 03 these elements will be looked up in the procedure’s schema. See section *Syntax Deprecated with SPS 03* for details.

DROP PROCEDURE

Syntax:

```
DROP PROCEDURE {schema.}name {CASCADE}
```

Description:

Drops a procedure created by CREATE PROCEDURE from the database catalog. If a cascading drop is defined, dependent objects will also be dropped. If a cascading drop is not defined, dependent objects will be invalidated. After a non-cascading drop, a procedure with the same name can be created which can make the previously invalidated dependent objects valid again.

ALTER PROCEDURE

Syntax:

```
ALTER PROCEDURE {schema.}name RECOMPILE {WITH PLAN}
```

Description:

This statement manually triggers a recompilation of a procedure by generating an updated execution plan. If the optional WITH PLAN argument is provided then internal debug information is created. Details on debugging SQLScript procedures are provided later in this document. For production code the procedure shall be compiled without WITH PLAN option to avoid any overhead during compilation and execution of the procedure.

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.

CALL - Procedure Called From Client

Syntax:

```
CALL {schema.}name (param1 {, ...}) {IN DEBUG MODE}
```

Description:

On the outer-most level actual parameters passed to a procedure are scalar constants and can be passed as either 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 either null or ?.

CALL conceptually returns list of result sets with one entry for every tabular result. An iterator can be used to iterate over this results set. For each result set one can iterate over the result table as is done 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 this list of result sets. The type of these 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, e.g. 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.

If the optional IN DEBUG MODE clause is used then additional debug information will be created. This information can be used to debug the instantiation of the internal execution plan of the called procedure. Note that this type of call implies additional runtime overhead. Thus, it should only be used for debugging purposes. Further details on debugging SQLScript procedures is provided later in this document.

Example:

```
CALL getOutput (1000, 'EUR', NULL, NULL);
```

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

CALL…WITH OVERVIEW From Client

Syntax:

```
CALL {schema.}name (param1 {, ...}) WITH OVERVIEW {IN DEBUG MODE}
```

Description:

A common requirement is to write the result of a procedure call directly into a physical table. For this purpose a procedure can be called using the CALL … WITH OVERVIEW syntax. This flavor of calling a procedure returns one result set that holds the information of which table contains the result of a particular table's output variable. Scalar outputs will be represented as temporary tables with only one cell. CALL … WITH OVERVIEW is the replacement for the former CALLS statement. When passing existing tables to the output parameters CALL … WITH OVERVIEW will insert the result set tuples of the procedure into the given tables. When passing NULL to the output parameters, temporary tables holding the result sets will be generated. These tables will be dropped automatically once the session is closed. The semantics of the optional IN DEBUG MODE clause is the same as for the call without WITH OVERVIEW clause.

Example:

```
CALL getOutput (1000, 'EUR', ot_publishers, NULL) WITH OVERVIEW;
```

Returns the following table:

Variable	Table
OUPPUT_PUBS	"SYSTEM"."OT_PUBLISHERS"
OUTPUT_YEAR	"SYSTEM"."OUTPUT_YEAR_4CF5FA0BF9397F88E10000000A12737F"

NOTE: Until SAP HANA SPS 03 the interpretation of scalar parameter values differed depending on the context. With SPS 03 a consistent behavior is available where string delimiters will always be

removed before parameter values are used in the statements. See section *Syntax Deprecated with SPS 03* for details.

CALL - Internal Procedure Call

Syntax:

```
CALL {schema.}name (:in_param1, out_param {, ...})
```

Description:

For an internal procedure, i.e. calls of one procedure by 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.

Implementing Functional Logic

The body of a procedure consists of a sequence of statements separated by semicolons:

```
CREATE PROCEDURE {schema.}name({IN|OUT|INOUT} param_name data_type
{,...}) LANGUAGE SQLSCRIPT ... AS
BEGIN
      statement1;
      statement2;
      {...}
END
```

Statements make up the logic in the body of a procedure. Each statement specifies a transformation of some data (e.g. by means of classical relational operators such as selection, projection) and binds the result 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 dataflow of a procedure, statements bind new variables that are referenced elsewhere in the body of the procedure.

Please note that 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.

We will discuss imperative constructs that cannot be optimized very well later in this document.

Table Variables

Table variables are variables with table type, i.e. they are bound to the value of a physical table, SQL query, or a calculation engine plan operator. Table variables can also be used in the signature of a procedure as IN or OUT variables.

A table variable is bound to tabular value when a procedure is called and a tabular IN parameter is bound to the value of its argument. The other alternative is in the body of a procedure, when a tabular expression is bound to a variable using an assignment. While IN and OUT parameters have to be typed explicitly, the type of a table variable in the body of a procedure is derived from the expression they are bound to at the compile time of the procedure.

Binding Table Variables

Table variables are bound using the equality operator. This operator binds the result of a valid SELECT statement or a call to a calculation engine plan operator on the right hand 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 (i.e. recursion is not possible).

Scalar Variables

In SQLScript it is possible to define scalar parameters as IN, OUT, or INOUT. They can be referenced anywhere in the procedure. We discuss assignment to scalar variables and local scalar variables later in this document (using the assignment operator :=). Scalar variables are referenced in the same way as table variables.

Referencing Variables

Bound variables are referenced by their name (e.g. *var*), i.e. in the variable reference the variable name is prefixed by : such as *:var.* A table function describes a dataflow graph using its statements and the variables that connect the statements. The order in which statements are written in a function 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 of the procedure. 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. Variable `:it_books` 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.

In addition to results of SELECT statements the result of a calculation engine plan operator, provided by the calculation engine, is also assigned to a variable. Calculation engine plan operators can be used to assign variable as follows:

```
ot_sales = CE_UNION_ALL(:lt_on_sale, :lt_cheap_books);
```

Here, `ot_sales` is bound, `:lt_on_sale` and `:lt_cheap_books` are variables that were assigned earlier in the body of the procedure.

Calculation Engine Plan Operators

Calculation engine plan operators encapsulate data-transformation functionality and can be used in the definition of functions. They constitute an alternative to using SQL statements as their logic is directly implemented in the calculation engine, i.e. the execution environment 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 implement, e.g., crucial business functions inside the database kernel.

Direct use of the calculation engine allows implementers to influence the execution of a procedure which, in some cases, is more efficient. However, you must be careful with the different execution semantics of SQL engine and calculation engine; this issue is discussed in more detail later in this document. A more extensive description of choosing between calculation engine plan operators and SQL statements is provided in the "SAP HANA Development Guide" document, you can find a link to this guide at the beginning of this document.

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.

CE_COLUMN_TABLE

Syntax:

```
CE_COLUMN_TABLE ("table_name"{, ["attrib_name", ...]})
```

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.

Example:

```
ot_books1 = CE_COLUMN_TABLE("BOOKS");

ot_books2 = CE_COLUMN_TABLE("BOOKS", ["TITLE","PRICE","CRCY"]);
```

This example only works on a column table and does not invoke the SQL processor. It is semantically equivalent to the following:

```
ot_books3 = SELECT * FROM books;
```

```
ot_books4 = SELECT title, price, crcy FROM books;
```

NOTE:

Calculation engine plan operators that reference identifiers have to enclose them with double-quotes and use capitalization of the identifier's name 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. Identifiers in calculation engine plan operators must match the internal representation, i.e. 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 case sensitive. Again, the identifiers in operators must match the internal representation.

CE_JOIN_VIEW

Syntax:

```
CE_JOIN_VIEW("join_view_name"{, ["attrib_name", ...]})
```

Description:

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

Example:

```
out = CE_JOIN_VIEW("PRODUCT_SALES",
                   ["PRODUCT_KEY", "PRODUCT_TEXT", "SALES"]);
```

Retrieves the attributes PRODUCT_KEY, PRODUCT_TEXT, and SALES from the join view PRODUCT_SALES.

It is equivalent to the following SQL:

```
out = SELECT product_key, product_text, sales FROM product_sales;
```

CE_OLAP_VIEW

Syntax:

```
CE_OLAP_VIEW("OLAP_view_name"{, ["DIM", "KEY_FIG", ... ]})
```

Description:

The CE_OLAP_VIEW operator returns results for an existing OLAP view (also denoted as 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.

Example:

```
out = CE_OLAP_VIEW("OLAP_view", ["DIM1", "KF"]);
```

Is equivalent to the following SQL:

```
out = select dim1, SUM(kf) FROM OLAP_view GROUP BY dim1;
```

CE_CALC_VIEW

Syntax:

```
CE_CALC_VIEW ("CALC_VIEW_NAME"{, ["attrib_name", ...]})
```

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.

Example:

```
out = CE_CALC_VIEW("_SYS_SS_CE_TESTCECTABLE_RET", ["CID", "CNAME"]);
```

Semantically equivalent to the following SQL:

```
out = SELECT cid, cname FROM "_SYS_SS_CE_TESTCECTABLE_RET";
```

Relational Operators

The calculation engine plan operators presented in this section provide the functionality of relational operators which 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 needed.

CE_JOIN

Syntax:

```
CE_JOIN (:var1_table, :var2_table, [join_attr, ...]{,
         [attrib_name , ...]})
```

Syntactically, the plan operator CE_JOIN takes four parameters as input

1. One table variable representing the left argument to be joined.
2. One table variable representing the right argument to be joined.
3. 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.
4. Optionally, a projection list can be provided specifying the attributes that should be in the resulting table. If this 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. If a projection list is provided, it must include the join 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.

Example:

```
ot_pubs_books1 = CE_JOIN (:lt_pubs, :it_books,["PUBLISHER"]);
ot_pubs_books2 = CE_JOIN (:lt_pubs, :it_books,["PUBLISHER"],
                          ["TITLE","NAME","PUBLISHER","YEAR"]);
```

This example is semantically equivalent to the following SQL but does not invoke the SQL processor.

```
ot_pubs_books3 = 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;

ot_pubs_books4 = SELECT title, name, P.publisher AS publisher, year
                 FROM :lt_pubs AS P, :it_books AS B
                 WHERE P.publisher = B.publisher;
```

CE_LEFT_OUTER_JOIN

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

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.

CE_PROJECTION

Syntax:

```
CE_PROJECTION (:var_table, [param_name [AS new_param_name], ...]{,
               [Filter]})
```

Description:

Restricts the columns in the schema of table variable `var_table` to those mentioned in the projection list. Optionally renames columns, computes expressions, or applies a filter.

The calculation engine plan operator CE_PROJECTION takes three parameters as input:

1. A variable of type table which is subject to the projection. Like `CE_JOIN`, `CE_PROJECTION` cannot handle tables directly as input.
2. A list of attributes which 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.
3. An optional filter where Boolean expressions are allowed, as defined for the CE_CALC operator below.

In this operator, the projection in parameter 2 is applied first – including column renaming and computation of expressions. As last step, the filter is applied.

Example:

```
ot_books1 = CE_PROJECTION (:it_books,["TITLE","PRICE",
                           "CRCY" AS "CURRENCY"], '"PRICE" > 50');
```

Semantically equivalent to the following SQL:

```
ot_books2= SELECT title, price, crcy AS currency
           FROM :it_books WHERE price > 50;
```

CE_CALC

Syntax:

```
CE_CALC ('<expr>', <result type>)
```

Description:

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

1. The expression enclosed in single quotes. The grammar for the expression is given below.
2. The result type of the expression as a SQL type.

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, i.e. *'token'*. Identifiers, denoted with *id* in the grammar are like SQL identifiers with the exception 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, e.g. 'a string'. Inside string single quotes are escaped by another single quote.

An example expression valid in this grammar is: "col1" < ("col2" + "col3").

The following functions are supported:

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	date date(stringarg), date date(fixedarg)
Sting Functions	*Functions on strings*	
strlen	return the length of a string in bytes, as an integer number	int strlen(string)
midstr	return a part of the string starting at arg2, arg3 bytes long. arg2 is counted from 1 (not 0)	string midstr(string, int, int)
leftstr	return arg2 bytes from the left of the arg1. If arg1 is shorter than the value of arg2, the complete string will be returned.	string leftstr(string, int)
rightstr	return arg2 bytes from the right of the arg1. If arg1 is shorter than the value of arg2, the complete string will be returned.	string rightstr(string, int)
instr	return 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.	int instr(string, string)
hextoraw	convert 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	convert a string of bytes to its hexadecimal representation. The output will contain only 0-9 and (upper case) A-F, no spaces and is twice as many bytes as the original string.	string rawtohex(string
ltrim	remove 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	remove 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	remove whitespace from the beginning and end of a string. These should be equivalent: • trim(s) = ltrim(rtrim(s)) • trim(s1, s2) = ltrim(rtrim(s1, s2), s2)	string trim(string) string trim(string, string)
lpad	add 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	add whitespace to the end of a string. A second string argument specifies the whitespace which will be added repeatedly until the	string rpad(string, int)

Name	Description	Syntax
	string has reached the intended length. If no second string argument is specified, chr(32) (' ') will be added.	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.*	
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 meaning as in C	
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	return the weekday as an integer in the range 0..6, 0 is monday.	weekday(date)
now	return the current date and time (localtime of the server timezone)	now()

Name	Description	Syntax
	as date	
daysbetween	return 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)
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()

Example:

```
with_tax = CE_PROJECTION(:product, ["CID", "CNAME", "OID", "SALES",
                                    CE_CALC('"SALES" * :vat_rate',
                                    decimal(10,2)) AS "SALES_VAT"],
                                    '"CNAME" = :cname');
```

Semantically equivalent to the following SQL:

```
with_tax2 = SELECT cid, cname, oid, sales,
                   sales * :vat_rate as sales_vat
            FROM :product WHERE cname = ':cname';
```

Notice, that all columns used in the `CE_CALC` have to be included in the projection list.

Another frequent use case of CE_CALC is computing row numbers:

```
CREATE PROCEDURE ceGetRowNum(IN it_books books,
                             OUT ranked_books ot_ranked_books)
    LANGUAGE SQLSCRIPT READS SQL DATA AS
BEGIN
  ordered_books = SELECT title, price, crcy
                  FROM :it_books ORDER BY price DESC;
  ranked_books = CE_PROJECTION(:it_books, ["TITLE", "PRICE",
                           CE_CALC('rownum()', integer) AS "RANK",
                           "CRCY" AS "CURRENCY"]);
END;
```

CE_AGGREGATION

Syntax:

```
CE_AGGREGATION (:var_table,
                [aggregate ("column") {AS "renamed_col"}] {,
                ["column", ...]});
```

Description:

Groups the input and computes aggregates for each group.

Syntactically, the aggregation operator takes three input parameters:

1. A variable of type table containing the data that should be aggregated. CE_AGGREGATION cannot handle tables directly as input.
2. A list of aggregates. For instance, [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.
3. An optional list of group-by attributes. For instance, ["C"] specifies that the output should be grouped by column C, i.e. 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 should be treated as a single group, and the aggregate function is applied to all tuples.

The final parameter is optional. Note that 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.

Supported aggregation functions are:

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

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

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.

Example:

```
ot_books1 = CE_AGGREGATION (:it_books,
                            [COUNT ("PUBLISHER") AS "CNT"], ["YEAR"]);
```

Semantically equivalent to the following SQL:

```
ot_books2 = SELECT COUNT (publisher) AS cnt, year
            FROM :it_books GROUP BY year;
```

CE_UNION_ALL

Syntax:

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

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, i.e. the result is a table which contains all the rows from both input tables.

Example:

```
ot_all_books1 = CE_UNION_ALL (:lt_books, :it_audiobooks);
```

Semantically equivalent to the following SQL:

```
ot_all_books2 = SELECT * FROM :lt_books
                UNION ALL
                SELECT * FROM :it_audiobooks;
```

Special Operators

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

CE_VERTICAL_UNION

Syntax :

```
CE_VERTICAL_UNION(:input1, [project_att1 {AS new_param_name}, ...],
              :input2, [project_att1 {AS new_param_name}, ...], ...)
```

Description:

For each input, applies the concatenation of their columns. Optionally columns can be renamed. Clearly, all input tables must have the same cardinality. The arguments of the operator are a list of pairs consisting of:

1. A table-typed variable
2. A projection list

Example:

```
out = CE_VERTICAL_UNION(:firstname,
                        ["ID", "FIRSTNAME" AS "GIVENNAME"],
                        :lastname, ["LASTNAME" AS "FAMILYNAME"]);
```

WARNING:

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

CE_CONVERSION

Syntax:

```
CE_CONVERSION(:input1, [key1 = value1, "key2" = :value, ...,
                        key3 = 'value3', "key_n" = "value_n", ...],
                        [convert_att1 {AS new_param_name}, ...])
```

Description:

Applies a unit conversion to input table 'input1' and returns the converted values. Result columns can optionally be renamed. The operator is highly configurable via a list of key-value pairs. The syntax above outlines valid combinations. Supported keys with their allowed domain of values are:

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'	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
'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

Key	Values	Type	Mandatory	Default	Documentation
					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

And for ERP-Conversion in particular also:

Key	Values	Type	Mandatory	Default	Documentation
'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

Example:

```
conv_tab = CE_CONVERSION(:input, [family = 'currency',
                         method = 'ERP', client = '004',
                         conversion_type = 'M', target_unit = 'EUR',
                         source_unit_column = "WAERK",
                         reference_date_column = "ERDAT",
                         output_unit_column = "TRGCUR"]);
```

There is no immediate SQL-equivalent of the above call to the plan operator CE_CONVERSION.

TRACE

Syntax :

```
TRACE(:input1)
```

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 view.

Example:

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

See the discussion on tracing and debugging later in this document for further details

NOTICE:
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.

Imperative Extension

Until now our focus was on SQLScript procedures that are free of side effects. This style of programming can be used to create procedures for basically any transformation of data into output data. However, in some cases, applications have to update the database content and by doing so create side effects. In this section we will be focusing on procedures that implement this kind of application logic. These procedures can be more easily implemented if application state can be maintained and updated, and so by definition they are not side-effect free.

Scalar Variables

Description:

In this section, the description of scalar variables for table functions is expanded upon. As an extension scalar parameters are supported as IN, OUT and INOUT parameters. Additionally, scalar variables can also be defined as local variables. Furthermore, this section illustrates how multiple values can be assigned to the same scalar variable.

Example:

```
CREATE PROCEDURE sql_proc() LANGUAGE SQLSCRIPT AS
    v_count   INT := 0;
    v_isbn    VARCHAR(20);
    v_title   VARCHAR(50) := '';
    v_price   decimal(5,2) := 0;
    v_crcy    VARCHAR(3) := 'XXX';

BEGIN

    v_isbn := '978-3-8266-1664-8';
    SELECT isbn, title, price, crcy
      INTO v_isbn, v_title, v_price, v_crcy
      FROM books WHERE isbn = :v_isbn;

    SELECT COUNT(*) INTO v_count FROM books;
    DELETE FROM books;
    SELECT COUNT(*) INTO v_count FROM books;
```

```
    v_isbn := '978-3-642-19362-0';
    v_title := 'In-Memory Data Management';
    v_price := 42.75;

    INSERT into books
      VALUES (v_isbn, :v_title, 1, 1, '2011', :v_price, 'EUR');
    SELECT isbn, title, price, crcy
      INTO v_isbn, v_title, v_price, v_crcy
      FROM books WHERE isbn = :v_isbn;

    UPDATE books SET price = price * 1.1 WHERE isbn = :v_isbn;
    SELECT isbn, title, price, crcy
      INTO v_isbn, v_title, v_price, v_crcy
      FROM books WHERE isbn = :v_isbn;

END;
```

In this procedure, local variables are declared directly after the signature. Local variables can optionally be initialized with their declaration. By default variables are initialized with NULL. A scalar variable – `var` – can be referenced the same way as described above using `:var`.

Assignment (in contrast to binding table variables) is possible multiple times overwriting the previous value stored in the scalar variable or table variable. Assignment is performed using the operator `:=`.

Note that in the above example that the result of some SQL queries are not assigned to a table variable. The result of these queries is directly returned to the client as a result set, see section *Procedure Calls* in this guide for details. I some cases this is a very effective way to return results to the client because the result can be streamed to the client. In contrast to that for the outermost call the result of table variables is materialized on the server side before it is returned to the client.

Control Structures

Conditionals

Syntax:

```
IF <bool-expr1>
THEN
   {then-stmts1}
{ELSEIF <bool-expr2>
THEN
   {then-stmts2}}
{ELSE
   {else-stmts3}}
END IF
```

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. In most cases this branch starts with `ELSE`. 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` another boolean expression – bool-expr2 – is evaluated. If it evaluates to true, the statements – then-stmts2 – is executed. In this fashion 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.

Example:

The following example illustrates the use of the `IF`-statement by implementing the functionality of the SAP HANA database`s UPSERT statement.

```
CREATE PROCEDURE upsert_proc (IN v_isbn VARCHAR(20)) LANGUAGE
SQLSCRIPT AS

    found INT := 1;

BEGIN

    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;
```

Test for Null-Values

Syntax:

```
IF <var> IS {NOT} NULL
THEN ... ELSE ... END IF
```

Description:

An important special case of the conditional statement is the test if a scalar variable – var – contains a NULL value. Notice that NULL is the default for local variables.

Example:

The following example illustrates the use of the `IF`-statement to check for a NULL value.

```
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;
```

While Loop

Syntax:

```
WHILE <bool-stmt> DO
   {stmts}
END WHILE
```

Description:

The while loop executes the statements – stmts – in the body of the loop as long as the boolean expression at the beginning – bool-stmt – of the loop evaluates to true.

Example:

The example illustrates how nested loops can be used.

```
v_index1 := 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;
```

Notice, that no specific checks are performed to avoid infinite loops.

For Loop

Syntax:

```
FOR <loop-var> IN {REVERSE} <start> .. <end> DO
   {stmts}
END FOR
```

Description:

The for loop iterates a range of numeric values – denoted by start and end in the syntax – and binds the current value to a variable (loop-var) in ascending order. Iteration starts with value start and is incremented by one until the loop-var is larger than end. Hence, if start is larger than end, the body loop will not be evaluated. For each enumerated value of the loop variable the statements in the body of the loop are evaluated. The optional keyword `REVERSE` specifies to iterate the range in descending order.

Example:

The example illustrates how nested loops are used. In this example, another procedure is called in the inner-most loop which traces the current values of the loop variables by appending them to a table.

```
FOR v_index1 IN -2 .. 2 DO
   FOR v_index2 IN REVERSE 0 .. 5 DO
      CALL ins_msg_proc('Here is ' ||:v_index1 || '~'
                    ||:v_index2 || '.');
    END FOR;
END FOR;
```

Break And Continue

Syntax:

```
BREAK

CONTINUE
```

Description:

This is internal control functionality for loops. You can use break to immediately leave the loop and continue to immediately resume with the next iteration.

Example:

```
FOR x IN 0 .. 10 DO
    IF :x < 3 THEN
        CONTINUE;
    END IF;
    ins_msg_proc('Inside loop, after CONTINUE: x = ' || :x);
    IF :x = 5 THEN
        BREAK;
    END IF;
END FOR;
```

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 query to which the cursor is bound.

Define Cursor

Syntax:

```
CURSOR <cursor-name> {(<param-name> <param-type>, ...)} FOR
       <SQL-SELECT-Stmt>
```

Description:

Cursors are defined like local variables, i.e. after the signature of the procedure and before the procedure's body. The cursor is defined with a name, optionally a list of parameters, and a 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.

Open Cursor

Syntax:

```
OPEN <cursor-name>{(arg1, ...)}
```

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 that one can fetch the result rows of a query.

Close Cursor

Syntax:

```
CLOSE <cursor-name>
```

Description:

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

Fetch Query Results of a Cursor

Syntax:

```
FETCH <cursor-name> INTO <var1, ... , var_n>
```

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.

Attributes of a Cursor

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

Attribute	Description
cur::ISCLOSED	Is true if cursor cur is closed, true otherwise.
cur::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.
cur::ROWCOUNT	Returns the number of rows in the result set of the query bound to the cursor. This value is available after the first FETCH operation.

Example:

```
CREATE PROCEDURE cursor_proc LANGUAGE SQLSCRIPT AS

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

BEGIN

    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
```

Looping over Result Sets

Syntax:

```
FOR <row-var> AS <cursor-name>{(arg1, ...)} DO
   <stmts>
   <row-var>.<column>    -- attribute access
END FOR
```

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 was visited, the loop is exited and the cursor is closed. 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 *row* and the iterated data contains a column *name*, then the value of this column can be accessed using *row.name*.

Example:

```
CREATE PROCEDURE foreach_proc() LANGUAGE SQLSCRIPT AS
    v_isbn    VARCHAR(20) := '';
    CURSOR c_cursor1 (v_isbn VARCHAR(20)) FOR
       SELECT isbn, title, price, crcy FROM books
       ORDER BY isbn;

BEGIN

    FOR cur_row as c_cursor1 DO
        CALL ins_msg_proc('book title is: ' || cur_row.title);
    END FOR;

END;
```

Dynamic SQL

With dynamic SQL it is possible to construct SQL statements at runtime of a SQLScript procedure.

Syntax:

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

Description:

Executes the SQL statement passed in a string argument. This statement allows for constructing an SQL statement at execution time of a procedure. Thus, on the one hand dynamic SQL allows to use variables where they might not be supported in SQLScript or more flexibility in creating SQL statements. On the other hand, dynamic SQL comes with an additional cost at runtime:

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

Note: It is recommended to avoid dynamic SQL because it might have a negative impact on security or performance.

Example:

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

Catalog Information

When a procedure or function is created, this will be reflected in the database catalog. One can use this information for debugging purposes. For procedures information are stored in system views summarized below.

View SYS.PROCEDURES:

Column	Description
SCHEMA_NAME	The schema in which the procedure was created
PROCEDURE_NAME	The name of the procedure
PROCEDURE_OID	The object ID, i.e. catalog-wide unique identifier
OWNER_OID	The object ID of the owner of the procedure
INPUT_PARAMETER_COUNT	The number of IN parameters
OUTPUT_PARAMETER_COUNT	The number of OUT parameters
INOUT_PARAMETER_COUNT	The number of INOUT parameters
IS_UNICODE	TRUE if the source isunicode, FALSE otherwise
DEFINITION	The statement used to create this procedure; currently this is always stored in plain text.
PROCEDURE_TYPE	The implementation language of the procedure, e.g. SQLSCRIPT2 or BUILTIN

NOTE:

Currently the code of a procedure is stored in plain text. Thus, every user with rights to read these system tables or views can read the code of a procedure.

View SYS. PROCEDURE_PARAMAMETERS:

Column	Description
SCHEMA_NAME	The schema of the procedure
PROCEDURE_NAME	The name of the procedure
PROCEDURE_OID	The unique identifier of the procedure
PARAMETER_NAME	The name of the procedure parameter
DATA_TYPE_ID	The id for the primitive data type; maps directly to a data type name, NULL for table types
DATA_TYPE_NAME	The name of the data type

LENGTH	The length information for the data type, if applicable
SCALE	The scale information for the data type, if applicable
POSITION	the position of the parameter in the signature
TABLE_TYPE_SCHEMA	The schema of the table type
TABLE_TYPE_NAME	The type name of the table type
PARAMETER_TYPE	IN, OUT, or INOUT
IS_NULLABLE	TRUE if NULL can be passed to the parameter, FALSE otherwise

These views are accessible by every user. Depending on the granted privileges a user will see all procedures or functions (CATALOG READ or DATA ADMIN privilege) or only some (schema owner, or execute privilege).

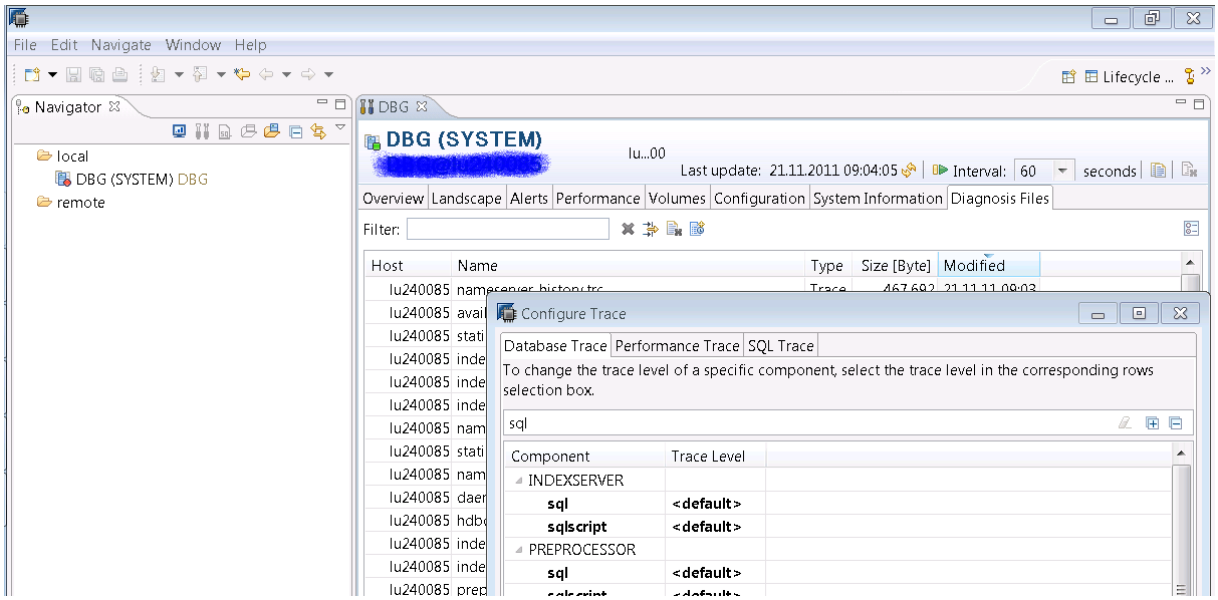
Procedures can be exported and imported like tables, see the SQL Reference documentation for details.

Tracing and Debugging SQLScript

In this section we summarize the capabilities to typically required to understand errors returned by the SQLScript compiler or unexpected runtime behavior of SQLScript procedures.

Tracing

To examine problems when creating SQLScript procedures, it is possible to increase the trace level for the *indexserver*. In this case the SAP HANA database will not only trace error messages but will produce more detailed information. In the SAP HANA studio this tracing information is available via the Administration perspective that can be accessed via the “tools” button or via the menu View → Open Perspective → Administration Console. In this perspective tracing information is available in the panel called “Diagnosis Files”.



Trace Levels for Debugging Information

In its default setup the SAP HANA database will only trace errors into trace files. These trace files are stored on the server in subdirectory <hostname>/trace. The error trace for the *indexserver* is called indexserver_alert.trc.

To trace additional information, e.g. for debugging purposes, please select the panel “Diagnosis Files” in the Administration perspective and click on the button “Configure Trace…” on top of the diagnosis files at the right border of the view. In the dialog select the tab “Database Trace”. Activate the check box “Show all Components”. Now you can select the component to trace. For SQLScript the following trace information in component “Indexserver” are relevant:

- **sqlscript**: information related to the SQLScript compiler, i.e. the early phases of compiling SQLScript into calculation models
- **calcengineinstantiate**: information collected during instantiation of a calculation model, i.e. the final phase of compiling SQLScript into calculation models and when the SQLScript procedure is invoked.
- **calcengine**: information collected by the calculation engine at runtime. This is relevant for the parts of SQLScript that are translated into calculation models.
- **llang**: information collected by the L component. This is relevant for all aspect of SQLScript the relate to L, i.e. compilation into L and execution of L code.

For each of the components one can set the trace level to NONE, FATAL, ERROR, WARNING, INFO or DEBUG. After this, SAP HANA database traces information further information into the file indexserver_<host>.<port>.<xyz>.trc.

```
[9245][0000000000] 2011-11-01 13:50:37.136194 e PersistenceM PersistenceManagerImpl.cpp(01604) : Cleaning
up DTX error: volume 1 not in known DTX volume set. Adding to DTX volume set.
[9245][0000000000] 2011-11-01 13:50:41.703824 e indexserver  TREXIndexServer.cpp(00593) : error while drop
public synonym SQLSCRIPT_TRACE
[9247][0000000000] 2011-11-01 13:53:33.749726 e Metadata     catalog.cc(00335) : failed to revoke select on
SYS.DUAL from PUBLIC
```

The above snippet is taken from an error trace. Each entry contains the date and time of the entry, an indicator of the severity (e.g. e for error and w for warning). After this the component is provided, e.g. Metadata for the last entry in the example above. It then resumes with the detailed trace information.

Performance Trace

To analyze the performance characteristics of a given workload it is possible to collect performance information. It allows specialists to associate long-running SQLScript or SQL features with code fragments. Based on this information rewrites of the SQL / SQLScript code can be proposed or performance issues in the database engine can be identified.

To trace the performance characteristics of a given workload, please select the panel “Diagnosis Files” in the Administration perspective and click on the button “Configure Trace…” on top of the diagnosis files at the right-hand border of the view. In the dialog select the tab “Performance Trace”. In this tab you can filter statistics for different users. More details are collected if the check boxes “Trace Execution Plans” and “Activate Function Profiler” are enabled. You can specify the filename of the file that will store the profiled data. As with other trace files, the performance statistics are stored on the server in directory <host>/trace. To confirm the settings and start collecting data you click on “Start Tracing”. Performance tracing stops after the defined period or by manually stopping the tracing. You can load the performance trace in the import feature in the Performance tab of the Administration perspective.

Tracing SQL Statements

During the compilation of SQLScript multiple SQL statements may be combined into larger ones, or statements may be reordered for better performance. Because of this, it is sometimes useful to trace the SQL statements executed by the SAP HANA database for debugging purposes. Another reason for using this feature would be to examining the performance of certain SQL statements.

To trace the execution of SQL statement, please select the panel “Diagnosis Files” in the Administration perspective and click on the button “Configure Trace…” on top of the diagnosis files at the right border of the view. In the dialog select the tab “SQL Trace”. In this tab one can define the trace level which defines the SQL statements to be traced (e.g. ALL or ALL WITH RESULT). Additionally, one can filter the statements to certain users, tables, views or types of statements. The executed SQL commands are written into the file sqltrace_<host>_<port>_<xyz>.py. When the connection settings are adjusted correctly, the python file can be executed directly against the SAP HANA database.

Further Tracing Information

Further tracing functionality relevant to SQLScript is available via the server configuration files. In the tab “Configuration” in the Administration perspective you can set the configuration parameters for the *indexserver*. More precisely, the indexserver.ini configuration file contains further settings for detailed information on the instantiated calculation engine models which are the runtime representation for SQLScript.

- Section: calcengine
  - Entry: tracemodels = yes
    If set to yes the json model before, and after, optimization is printed to trace file

  - Entry: show_intermediate_results = yes or topXXX
    TopXXX will write the first XXX rows of the result set, yes writes all results
- Section: trace
  - Entry: row_engine = warning
    This entry sets the trace level for the row engine of the SAP HANA database which enables warnings for the SQLScript compiler including deprecation warnings. The default is error.

Generate Debug Information for SQLScript

In this section basic sources are described to collect debugging information as they are available in the SAP HANA database for SPS3. These concepts may change in future versions of the SAP HANA database.

Tracing for SQLScript Compiler

For better debugging support during development of SQLScript procedures it is possible to generate additional information on how SQLScript code in the editor maps to operators in the Calculation Engine. As the SQLScript compiler and the Calculation Engine perform non-trivial rewrites in the compilation process, e.g. merging or reordering operations, this is often needed to understand error messages returned by the compiler.

Error messages displayed after a procedure is called may refer to internal data structures of a plan. To be able to map the error messages to the source code, the procedure must be recompiled with debug information:

```
alter procedure getOutput RECOMPILE WITH PLAN;
```

As the debug information implies an additional overhead, procedures should be recompiled again after all needed debugging information has been collected:

```
alter procedure getOutput RECOMPILE;
```

After generating the debug information the SAP HANA database provides two temporary tables that provide compile-time debug information :

```
select * from #PROCEDURE_DATAFLOWS_
```

The temporary table `#PROCEDURE_DATAFLOWS_` can be used to identify the data flows into which the SQLScript compiler translated the SQLScript source. In our example, the compiler creates a single dataflow, which is optimal for optimizing the script into an efficient plan.

```
select * from #PROCEDURE_MAPPING_
```

DEFINER_...	SCHEMA_NAME	PROCEDURE...	STATEMENT_ID	TYPE	LINE	COLUMN	STATEMENT	RENAMED_STATEME...	DATAFLOW_NA...	VARIABLE_NAME	SCENARIO_NAME	NODE_NAME
SYSTEM	SQLSCRIPTDOC...	GETOUTPUT	46997053699...	DECLARE	0	0			143020	BIG_PUB_BOOK...	_SYS_SS_CE_1430...	$$big_pub_books_1$$
SYSTEM	SQLSCRIPTDOC...	GETOUTPUT	46997053699...	DECLARE	0	0			143020	BIG_PUB_IDS_1	_SYS_SS_CE_1430...	$$big_pub_ids_1$$
SYSTEM	SQLSCRIPTDOC...	GETOUTPUT	46997053693...	ASSIGN_OUTPARA...	0	0		OUTPUT_PUBS = OUT...	143020	OUTPUT_PUBS	_SYS_SS_CE_1430...	$$output_pubs$$
SYSTEM	SQLSCRIPTDOC...	GETOUTPUT	46997053693...	ASSIGN_OUTPARA...	0	0		OUTPUT_YEAR = OUT...	143020	OUTPUT_YEAR	_SYS_SS_CE_1430...	$$output_year$$
SYSTEM	SQLSCRIPTDOC...	GETOUTPUT	46997053694...	ASSIGN	7	3	big_pub_id...	BIG_PUB_IDS_2 = SE...	143020	BIG_PUB_IDS_2	_SYS_SS_CE_1430...	$$big_pub_ids_2$$
SYSTEM	SQLSCRIPTDOC...	GETOUTPUT	46997053695...	ASSIGN	12	3	big_pub_b...	BIG_PUB_BOOKS_2 = ...	143020	BIG_PUB_BOOK...	_SYS_SS_CE_1430...	$$big_pub_books_2$$
SYSTEM	SQLSCRIPTDOC...	GETOUTPUT	46997053696...	ASSIGN	18	3	output_pu...	OUTPUT_PUBS_1 = S...	143020	OUTPUT_PUBS	_SYS_SS_CE_1430...	$$output_pubs$$
SYSTEM	SQLSCRIPTDOC...	GETOUTPUT	46997053696...	ASSIGN	22	3	output_yea...	OUTPUT_YEAR_1 = S...	143020	OUTPUT_YEAR	_SYS_SS_CE_1430...	$$output_year$$

As shown above, the second temporary table maps internally used identifiers (which are used in error messages) into lines in the source code. The columns have the following semantics:

Column	Description
DEFINER_NAME	The user that created the procedure
SCHEMA_NAME	The schema in which the procedure was created
PROCEDURE_NAME	The name of the procedure
STATEMENT_ID	The identifier for the statement in the procedure
TYPE	DECLARE for IN parameters or local variables, ASSIGN_OUTPARAM for OUT parameters, and ASSIGN for assignments to table variables.
LINE and COLUMN	the position in the procedure; the signature is in column 0.
STATEMENT	the original statement
RENAMED_STATEMENT	as the single static assignment form is used to compile the procedure, variables are renamed. This statement shows the result of this renaming
DATAFLOW_NAME	the dataflow
VARIABLE_NAME	the assigned variable, internal variables have a suffix with a number.
SCENARIO_NAME	the name of the calculation scenario created for the procedure during compilation
NODE_NAME	the name of the node (or operator) in the calculation scenario.

Tracing SQLScript Invocation

As pointed out at the beginning of this document, when SQLScript procedures are invoked their corresponding calculation scenario(s) is/are instantiated. In this process, arguments are used to further optimize the plan generated for a SQLScript procedure by creating different calculation models. After this optimization step, the optimized calculation models are executed.
For debugging purposes, it is possible to collect information during instantiation and execution of a procedure. This is done adding “IN DEBUG MODE” to the call of the procedure.

```
CALL getOutput(0, 'EUR', NULL, NULL) WITH OVERVIEW IN DEBUG MODE;
```

Similar to the compile-time debugging information, internal tables are filled with further debugging information which are useful in understand issues that occur when a procedure is invoked.

```
select * from M_CE_DEBUG_NODE_MAPPING;
```

SCENARIO_NAME	NODE_NAME	NODE_TYPE	SUCC_SCENARIO_NAME	SUCC_NODE_NA...	RUNTIME_NODE_NAME
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_INS"					$REQUEST$
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_INS"	142536				142536_0
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_INS"	_SYS_SS_CE_142536_TMP_CALL	TEMPLATE	"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_T...	output_pubs	
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_INS"	_SYS_SS_CE_142536_TMP_CALL	TEMPLATE	"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_T...	output_year	
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_TMP"	output_pubs				output_pubs_SYS_SS_CE_142536_TMP_CALL_0
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_TMP"	output_year				output_year_SYS_SS_CE_142536_TMP_CALL_0
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_TMP"	$$output_pubs$$				$$output_pubs$$_SYS_SS_CE_142536_TMP_CALL
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_TMP"	$$output_year$$				$$output_year$$_SYS_SS_CE_142536_TMP_CALL_
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_TMP"	big_pub_books_2				big_pub_books_2_SYS_SS_CE_142536_TMP_CALL
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_TMP"	big_pub_ids_2				big_pub_ids_2_SYS_SS_CE_142536_TMP_CALL_0

Column	Description
SCENARIO_NAME	the calculation scenario representing the compiled SQLScript code.
NODE_NAME	the name of the node (or operator) in the calculation scenario.
NODE_TYPE	TEMPLATE_NODE represents a call to another calculation scenario; an empty type represents an operator inside a calaculation scenario.
SUCC_SCENARIO_NAME	for TEMPLATE_NODES identifies the called calculation scenario.

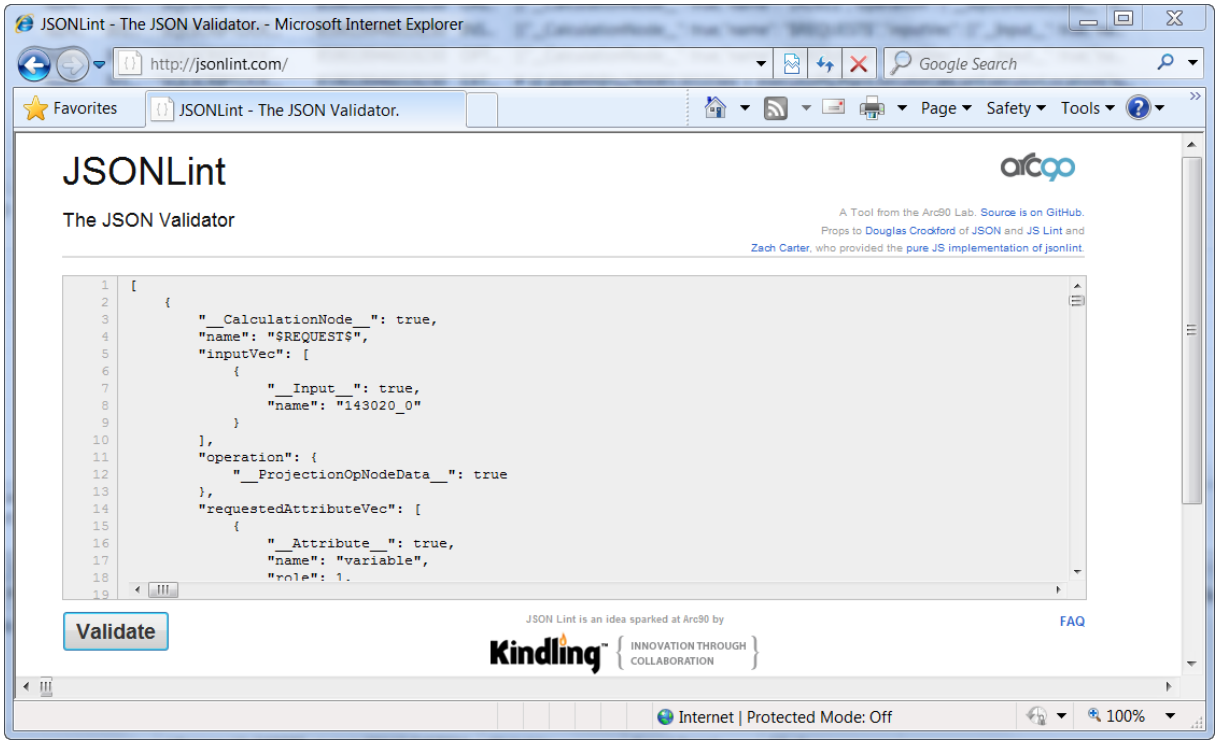
SUCC_NODE_NAME	identifies the nodes from which the result is consumed. This reference indicates the dataflow among calculation scenarios
RUNTIME_NODE_NAME	the node names in the optimized calculation model

More detailed information are available in the temporary table `M_CE_DEBUG_JSONS`:

```
select * from M_CE_DEBUG_JSONS
```

SCENARIO_NAME	QUERY_ID	TYPE	JSON
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_INS"	858999003949713	ORIGINAL	[{"__CalculationNode__": true,"name": "_SYS_SS_CE_142536_TMP_CALL","op...
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_TMP"	858999003949713	ORIGINAL	[{"__CalculationNode__": true,"name": "141948","operation": {"__SqlDSNode...
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_INS"	858999003949713	INSTANTIATED	[{"__CalculationNode__": true,"name": "$REQUEST$","inputVec": [{"__Input__...
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_INS"	858999003949713	OPTIMIZED	[{"__CalculationNode__": true,"name": "$REQUEST$","inputVec": [{"__Input__...
"SQLSCRIPTDOCUMENTATION"."_SYS_SS_CE_142536_INS"	858999003949713	EXTRACE	# id plan526@lu240085:30103ex = executorPy.PlanExecutor()ex.setExecuto...

This temporary table provides access to the calculation models generated when a SQLScript procedures is invoked. The TYPE column indicates the steps of the instantiation of the procedure where TYPE = EXTRACE actually provides the runtime information including the SQL statements executed. For better readability, one can use JSONLint to generate a readable format of the JSON string.



Manually Tracing Intermediate Results

The debug information presented so far can be used to understand how the SAP HANA database will process the SQLScript procedure. Nevertheless, the actual data passed to or returned by certain expressions may be important to debug a SQLScript procedure. This can be achieved by adding TRACE-operators into the SQLScript code as illustrated below.

```
CREATE PROCEDURE getOutput2( IN cnt INTEGER, IN currency VARCHAR(3),
                             OUT output_pubs tt_publishers,
```

```
                              OUT output_year tt_years)
    LANGUAGE SQLSCRIPT READS SQL DATA AS

BEGIN

  t_big_pub_ids = SELECT publisher AS pid FROM books     -- Query Q1
                  GROUP BY publisher HAVING COUNT(isbn) > :cnt;
  big_pub_ids = TRACE(:t_big_pub_ids);
  t_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;
  big_pub_books = TRACE(:t_big_pub_books);
  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;
```

In the example above we have extended the dataflow of the procedure by two additional TRACE operators. These will trace the intermediate result at this point into local temporary tables. Note that as a consequence the trace information is only available via the same connection and session. Execute the following command:

```
select * from sqlscript_trace;
```

This view contains 4 columns:

Column	Description
PID	the process id
TID	the transaction is
VARNAME	the internal variable name used for the variable to which the TRACE operator is assigned.
TABLENAME	the name of the local temporary table containing the table result.

If you want to clear the content you may call the following built-in procedure.

```
call truncate_sqlscript_trace;
```

Please keep in mind that this tracing functionality adds significant runtime overhead. So only use the tracing functionality for debugging purpose.

Developing Applications with SQLScript

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.

Below, we briefly cover some best practices which were part of this reference in previous versions. For an in-depth discussion we now refer to the SAP HANA Database – Development Guide.

- **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.

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.

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
```

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.

Calling SQLScript from ABAP

```
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.
```

Calling SQLScript from Java

For an explanation for setting up the JDBC driver we refer to the SAP HANA Development Guide.

```
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();
}
```

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("Output 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);
            }
        }
    }
}
```

Integration with SAP HANA Modeler

All statements discussed can be submitted to SAP HANA database like regular SQL statements in the SQL Editor. In the SAP HANA studio a user-friendly editor to create SQLScript procedures is available. It supports creating read-only SQLScript procedures. The basic workflow for creating a procedure is described in the documentation (see the SAP HANA Modeling Guide referenced at the beginning of this document). More precisely, the following restrictions apply for procedures created in the information modeler:

- Read-only procedures can be created.
- IN variables can be of scalar or table type.
- OUT parameters must have table type.
- Result views cannot be defined.
- Table types required for the signature are generated automatically when the procedure is activated, and so reuse of table types is not yet available.
- Procedure names are derived from the name of the object, and procedures are created in schema _SYS_BIC.

When a procedure is stored, it is not immediately available in the catalog to be called. Instead it is stored in internal tables as a design time object. The procedure can be activated when the user has finished editing. During activation the procedure is compiled and can then be called from other procedures. More specifically:

- A procedure with name <proc> will create a repository object with the same name used when the procedure is stored
- Activation of a procedure with name <proc> in package <package> creates
    - a SQLScript procedure with name ”_SYS_BIC”.”<package-name>/<proc>”.

- an input or output parameter of type table with name <param> will be generated and has the name "_SYS_BIC"."<package-name>/<proc>/tabletype/<param>"
- during activation the procedure is compiled and existing procedures with the same name will be replaced; activation is aborted if any compile errors occur. Error messages are provided in the activation log.
- Unqualified type references are by default mapped to the current default schema. Such unqualified references can be mapped to a target schema during activation

This approach has the advantage that procedures can be transported using the transport mechanism available via the modeler, i.e. an export and import facility is available; see the documentation SAP HANA Modeling Guide for details. Notice that import and export also works for procedures in the database catalog:

```
EXPORT <proc-name> AS BINARY INTO '<path>'
      {WITH REPLACE} {THREADS <n>}

IMPORT <proc-name> AS BINARY FROM '<path>'
      {WITH REPLACE} {THREADS <n>}
```

Changes in Previous Versions of SQLScript

Syntax Removed in SPS 03

Feature	Category	Old Syntax	Availability in SAP HANA 1.0 GA	Availability in SAP HANA SPS 02	Availability in SAP HANA SPS 03	New Syntax
Create table type	SqlScript v1 syntax	CREATE TABLE TYPE <name> <definition>	available	deprecated	removed	CREATE TYPE <name> AS TABLE <definition>
	SqlScript v1 syntax	ALTER TABLE TYPE <name> <definition>	available	deprecated	removed	DROP / CREATE
Drop table type	SqlScript v1 syntax	DROP TABLE TYPE <name> <definition>	available	deprecated	removed	DROP TYPE <name>
Create function	SqlScript v1 syntax	CREATE FUNCTION <name> (<params>) [TYPE TABLE] BEGIN … END	available	deprecated	removed	CREATE PROCEDURE <name> (<param list>) LANGUAGE SQLSCRIPT READS SQL DATA [WITH RESULT VIEW <view name>] AS BEGIN … END
External language body	SqlScript v1 syntax	''' <external language body> '''	available	deprecated	removed	BEGIN … END
Replace function	SqlScript v1 syntax	REPLACE FUNCTION	available	deprecated	removed	DROP / CREATE
Invocation (external)	SqlScript v1 syntax	CALLS/CALL FUNCTION <func name> (<params>)	available	deprecated	removed	CALL <proc name> (<params>) / CALL <proc name> (<params>) WITH

						OVERVIEW
Invocation (internal)	SqlScript v1 syntax	CALLS/CALL FUNCTION <func name> (<params>)	available	deprecated	removed	CALL <proc name> (<params>)
Parameter declaration	Old procedure syntax	<param name> [IN|OUT|IN OUT] <data type>	removal announced	removal announced	removed	[IN|OUT |INOUT]
	Old procedure syntax	IN OUT	removal announced {	removal announced	will be removed	INOUT
	SqlScript v1 syntax	Support for internal data types, e.g. String"	removal announced	removal announced	removed	Use standard SQL types, e.g. VARCHAR
Cursor declaration	Old procedure syntax	CURSOR <cursor name> <cursor param list> IS <spec>	removal announced	removal announced	removed	CURSOR <cursor name> <cursor param list> FOR <spec>
Cursor	Old procedure syntax	<cursor name> % ISOPEN, <cursor name> % NOTFOUND, <cursor name> % ROWCOUNT	removal announced	removal announced	removed	<cursor name> :: ISCLOSED, <cursor name> :: NOTFOUND, <cursor name> :: ROWCOUNT
Statement	Old procedure syntax	SQL % ROWCOUNT	removal announced	removal announced	removed	:: ROWCOUNT
Control statement	Old procedure syntax	ELSIF <condition> THEN <stmt list>	removal announced	removal announced	removed	ELSEIF <condition> THEN <stmt list>
	Old procedure syntax	WHILE <condition> LOOP <stmt list> END LOOP	removal announced	removal announced	removed	WHILE <condition> DO <stmt list> END WHILE
	Old procedure syntax	FOR <cursor name> IN [REVERSE] <num> .. <num> LOOP <stmt list> END LOOP	removal announced	removal announced	removed	FOR <cursor name> IN [REVERSE] <num> .. <num> DO <stmt list> END FOR
	Old procedure syntax	FOREACH <iterator name> IN <cursor name> [<param list>] LOOP <stmt list> END LOOP	removal announced	removal announced	removed	FOR <iterator name> AS <cursor name> [<param list>] DO <stmt list> END FOR
	Old procedure syntax	EXIT or LEAVE (for exit), EXIT WHEN <condition>	removal announced	removal announced	removed	BREAK
	Old procedure syntax	CONTINUE WHEN <condition>	removal announced	removal announced	removed	IF <condition> CONTINUE
Exception handling	Old procedure syntax	EXCEPTION WHEN … THEN	removal announced	removal announced	removed	N/A
Variable access	SqlScript v1 syntax	@ var @	available	deprecated	removed	:var or var

Create function/procedure		CREATE FUNCTION/PROCEDURE … BEGIN … END;	available	deprecated	removed	No ; at the end needed

Syntax Deprecated with SPS 03

The following constructs are deprecated with SPS 03. There are ways you can continue to use old syntax for the time being, but it is strongly recommended to update your code. In a future release of the SAP HANA database – most likely SPS 04 – the deprecated syntax will result in compiler or runtime errors.

Handling of Literals Passed as Scalar Parameter

Behavior deprecated with SAP HANA SPS 03: The handling of scalar parameter values is currently different when used in:

1) SQL statements that are assigned to a table variable
2) SQL statements that are not assigned to table variables + CE operators (CE_CALC [expression for calculated attribute], CE_PROJECTION [expression for filter].

Consistent behavior since SAP HANA SPS 03: string delimiters will always be removed before parameter values are used in the statements:

- CALL myProc('name')
  - usage as attribute with the value name
  - executed sql statement: SELECT * FROM tab WHERE COLUMN = NAME
- CALL myProc(' "name"')
  - usage as attribute with the value "name"
  - executed sql statement: SELECT * FROM tab WHERE COLUMN = "name"
- CALL myProc('''name '')
  - usage as string literal with the value 'name'
  - executed sql statement: SELECT * FROM tab WHERE COLUMN = 'name'

This behavior is governed by the parameters in the indexserver.ini file in section 'calcengine'. If the parameter 'use_prepared_statement_quoting' is enabled (‘= on’), then the behavior of quoted strings is like SQL prepared statements as described above. Otherwise the behavior used prior SPS 03 is active.

Name Resolution

Before SAP HANA SPS 03 unqualified schema elements (e.g. tables, views, or procedures) were looked up in the default schema defined at the time when the procedure was created. With SPS 03 this semantics is deprecated and unqualified schema elements are looked up in the procedure’s schema. This is illustrated below

```
SET SCHEMA "ERP_SCHEMA"

CREATE PROCEDURE "PROC_SCHEMA".myProc AS
BEGIN

   SELECT * FROM "SOME_TABLE_OR_VIEW";
   SELECT * FROM "CRM_SCHEMA"."SOME_TABLE_OR_VIEW";
```

```
END;
```

The name resolution is defined by the parameter 'use_old_name_resolution' in section 'sqlscript' of the indexserver.ini file as indicated in the following table.

Parameter	Name resolution during compile time
use_old_name_resolution = true (current default schema)	`SELECT * FROM "ERP_SCHEMA"."SOME_TABLE_OR_VIEW"; SELECT * FROM "CRM_SCHEMA"."SOME_TABLE_OR_VIEW";`
use_old_name_resolution = false (schema of procedure)	`SELECT * FROM "PROC_SCHEMA"."SOME_TABLE_OR_VIEW"; SELECT * FROM "CRM_SCHEMA"."SOME_TABLE_OR_VIEW";`