SAP NetWeaver
How-To Guide

How To...Use SQL-Script for Planning Functions in PAK

Applicable Releases:

SAP NetWeaver BW 7.30, SP10; BW 7.31, SP 9; BW 7.40 SP5 and higher

SAP HANA 1.0

Topic Area:

Business Information Management

Version 1.4

September 2015

THE BEST-RUN BUSINESSES RUN SAP™

© Copyright 2015 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft, Windows, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.

IBM, DB2, DB2 Universal Database, OS/2, Parallel Sysplex, MVS/ESA, AIX, S/390, AS/400, OS/390, OS/400, iSeries, pSeries, xSeries, zSeries, z/OS, AFP, Intelligent Miner, WebSphere, Netfinity, Tivoli, Informix, i5/OS, POWER, POWER5, OpenPower and PowerPC are trademarks or registered trademarks of IBM Corporation.

Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either trademarks or registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Oracle is a registered trademark of Oracle Corporation.

UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group.

Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Sun Microsystems, Inc.

JavaScript is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

MaxDB is a trademark of MySQL AB, Sweden.

SAP, R/3, mySAP, mySAP.com, xApps, xApp, SAP NetWeaver, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. These materials are provided by SAP AG and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

These materials are provided "as is" without a warranty of any kind, either express or implied, including but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement.

SAP shall not be liable for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials.

SAP does not warrant the accuracy or completeness of the information, text, graphics, links or other items contained within these materials. SAP has no control over the information that you may access through the use of hot links contained in these materials and does not endorse your use of third party web pages nor provide any warranty whatsoever relating to third party web pages.

SAP NetWeaver "How-to" Guides are intended to simplify the product implementation. While specific product features and procedures typically are explained in a practical business context, it is not implied that those features and procedures are the only approach in solving a specific business problem using SAP NetWeaver. Should you wish to receive additional information, clarification or support, please refer to SAP Consulting.

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

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.

THE BEST-RUN BUSINESSES RUN SAP™

Document History

Document Version	Description
1.00	First official release of this guide
1.10	Remark regarding message table added (chapter 3.9)
1.20	New note mentioned necessary for displaying messages (chapter 3.9)
1.30	Note added regarding side-effect free SQL-Script (chapter 3.2)
1.40	Small corrections in chapter SQL-Script Debugging (chapter 3.8)

THE BEST-RUN BUSINESSES RUN SAP™

Typographic Conventions

Type Style	Description
Example Text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation
Example text	Emphasized words or phrases in body text, graphic titles, and table titles
`Example text`	File and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example text`	User entry texts. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE TEXT`	Keys on the keyboard, for example, `F2` or `ENTER`.

Icons

Icon	Description
⚠	Caution
💡	Note or Important
✲	Example
⬆	Recommendation or Tip

Table of Contents

1. Scenario

The Planning Applications Kit is capable to execute calculation used in planning functions in memory and thus greatly accelerates the execution. Unfortunately not all planning functions can be executed in memory (please have a look at the list in note 1637199). Especially custom defined planning function (ABAP exits) cannot be executed in SAP HANA. Instead implementing the planning function logic in ABAP it is now possible to write a SQL-Script procedure that changes the plan data as desired and that can be executed in SAP HANA. SQL-Script is an extension of SQL and is a proprietary programming language in SAP HANA. As SQL-Script is not well known yet in the planning world we want to give some examples of planning function implementations that should be usable as a first introduction into SQL-Script. In our examples we try to show some common patterns generally used in planning functions, especially custom defined planning functions (looping over tables, local programming variables, accessing external data, using local tables, inserting and changing data in tables, using modularization/subroutines etc.).

2. General Description of the Solution

For a full introduction into SQL-Script we recommend to have a look at the HANA development guide (http://help.sap.com/saphelp_hana/sap_hana_developer_guide_en.pdf).

Still we try to explain how our planning function logic is built up and why we do it in this way.

When using SQL-Script for the implementation of planning function logic we only implement the calculation logic in SQL-Script. We still need some orchestration logic that is implemented in ABAP. In order to use a SQL-Script procedure you have to create a planning function type and the corresponding ABAP class that is used in the planning function type and implements the orchestration logic.

If you need parameters in the planning function they have to be created in the planning function type. Please note that only elementary planning function parameters are passed to SAP HANA execution.

In this guide we will not go into all details how the framework for a planning function type SQL-Script is to be set up. This is explained in the documentation. There is also a report that generates templates that can be used to create the necessary objects (RSPLS_SQL_SCRIPT_TOOL, please check SAP notes 1887040, 1897239, 1945449, 1976522). We will rather concentrate on the planning logic inside the SQL-Script procedure.

Still we will give some hints on which objects are necessary for a SQL-Script planning function. An step by step procedure on how such a planning function can be implemented as an AMDP (ABAP Managed Database Procedure) can be found in the following How to paper:

How to… Use Data from Another Aggregation Level in a PAK SQL-Script procedure – Implemented as an AMDP Example (https://scn.sap.com/docs/DOC-53376).

As you should also be able to test the procedure in your own system we will include some SQL-Script that can be used to create a test scenario.

Please note that even if it is possible to use procedural logic in SQL-Script the system can only use all its capabilities for speeding up the execution of a procedure (such as parallelization) if only declarative logic is used. A typical example for a programming technique that results in procedural programming is a loop over a table. This is a typical concept when you are using ABAP programs. In SQL-Script such a loop can also be realized by using so-called cursors (a cursor can be seen as a work area you use for looping over a table) but if you use those cursors then SAP HANA cannot parallelize the execution of the procedure anymore. Thus you should try to limit the use of such cursor to the absolute minimum. We also try to avoid cursors and other procedural programming in our examples. This is why the logic looks a bit unfamiliar when you are coming from an ABAP background. Nevertheless we will also have an example where we have to use cursors.

3. Examples for Planning Functions

3.1 Create a Test Environment

Our SQL-Script procedure receives the data in the plan buffer (restricted by the filter selection) and returns the changed plan data (in the ABAP class you can set whether an after image or a table with the delta records should be returned).

In the appendix you will find some coding that generates a table and fills it with records the table corresponds to the aggregation level. Our data model is very simple. We are using the following characteristics:

0CALYEAR

0CALMONTH2

Z_PROD

We use two key figures:

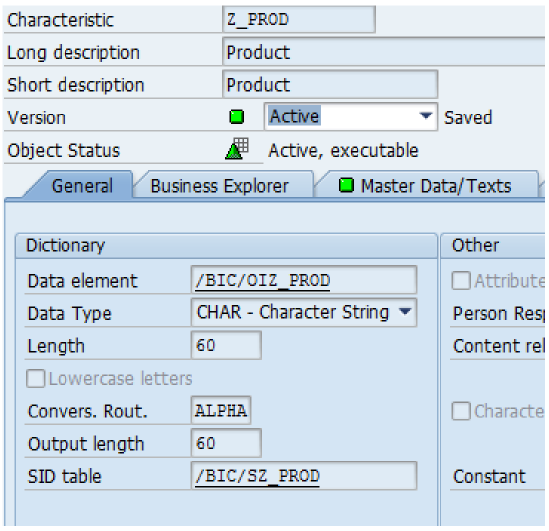
Z_QUAN (with a fixed unit ‘PC’)

Z_AMOUNT (with a fixed currency ‘EUR’).

You can easily create the characteristic and the two key figures in your system.

Use the following settings for the InfoObjects:

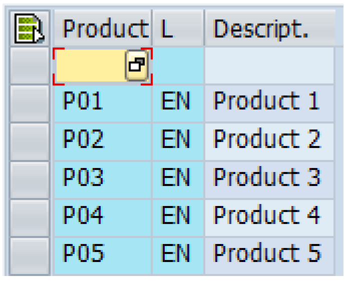
Z_PROD:



Characteristic	Z_PROD
Long description	Product
Short description	Product
Version	🟩 Active Saved
Object Status	Active, executable

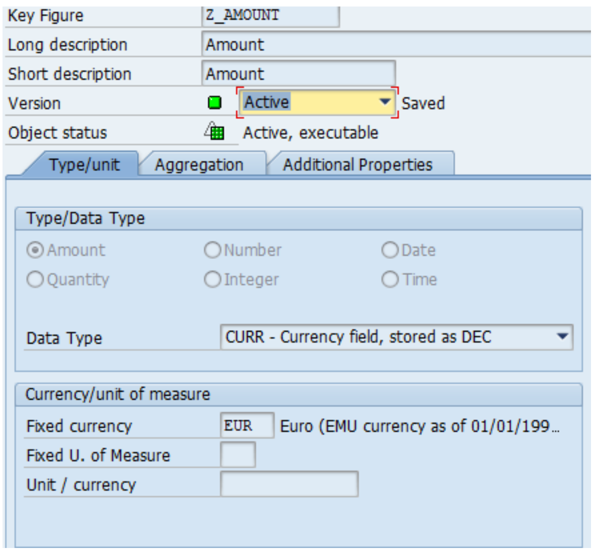
Data element	/BIC/OIZ_PROD
Data Type	CHAR - Character String
Length	60
☐ Lowercase letters	
Convers. Rout.	ALPHA
Output length	60
SID table	/BIC/SZ_PROD

Master Data for Z_PROD:



Product	L	Descript.
P01	EN	Product 1
P02	EN	Product 2
P03	EN	Product 3
P04	EN	Product 4
P05	EN	Product 5

Z_AMOUNT



Key Figure	Z_AMOUNT
Long description	Amount
Short description	Amount
Version	Active Saved
Object status	Active, executable

Type/unit | Aggregation | Additional Properties

Type/Data Type

[✓ Amount] Number Date

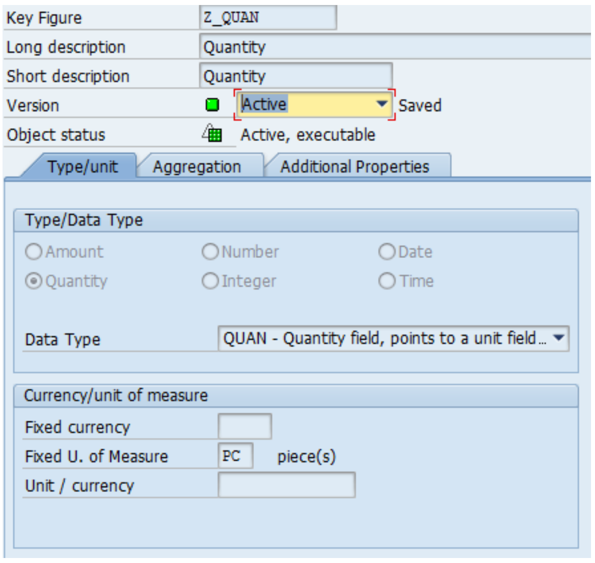
Quantity Integer Time

Data Type: CURR - Currency field, stored as DEC

Currency/unit of measure

Fixed currency	EUR Euro (EMU currency as of 01/01/199...
Fixed U. of Measure	
Unit / currency	

Z_QUAN:



Key Figure	Z_QUAN
Long description	Quantity
Short description	Quantity
Version	Active Saved
Object status	Active, executable

Type/unit | Aggregation | Additional Properties

Type/Data Type

Amount Number Date

[✓ Quantity] Integer Time

Data Type: QUAN - Quantity field, points to a unit field...

Currency/unit of measure

Fixed currency	
Fixed U. of Measure	PC piece(s)
Unit / currency	

Now create an InfoCube in order to test the planning functions also in an InfoCube and an aggregation level containing all characteristics and key figures of the InfoCube.

In order to create the test environment just open HANA Studio and open a SQL console. Copy and paste the coding from appendix 1 into the console. Exchange the schema name ‘SAPXYZ’ by the name of your default BW schema (usually a concatenation of ‘SAP’ and you three letter ABAP system name) and press F8. If you want to have a look at the table in HANA Studio expand the name of your system, then expand the node ‘Catalog’. Now look for your default schema name (SAP + name of your ABAP system) and expand this node. The tables can be found under the entry ‘Tables’. As BW creates all its tables in this schema you will see a large number of tables. It is best to search for your table with the context menu.

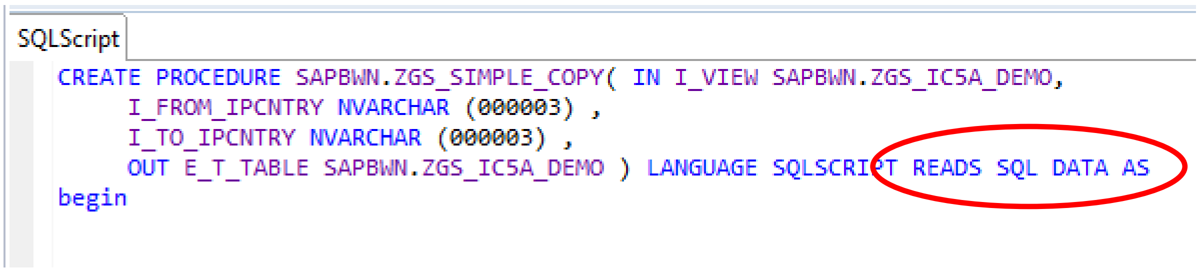
Please note that you could also first fill the InfoCube with some data (create a manual step in a planning sequence and enter the new data records) and then use the How to Paper: How to… Easily Create a Test Environment for a SQL-Script Planning Function in PAK (https://scn.sap.com/docs/DOC-53377) in order to create and fill the test data environment.

We need a second table as we want to have some ‘price’ table to do some price calculations later. Again we will create this table directly in HANA (this time we will have no InfoCube corresponding to this table so we cannot use the option described in the How to Paper above). Open a new SQL console in HANA Studio, copy and paste the coding from appendix 2 into the console, adapt the name of the schema, and press F8.

As a last step we need a table type corresponding to our aggregation level. Our SQL-Script procedures have to have an ‘in’-table containing the existing data in the selection and an ‘out’-table with the changed data. These tables need to have a type that matches the aggregation level. You can either use report RSPLS_SQL_SCRIPT_TOOL in order to generate the type or just use the SQL-Script from appendix 3. Again you have to adapt the schema name before executing the SQL-Script. In the HANA Studio you will find the table type in the catalog under you schema name and then ‘Procedures->Table Types’.

3.2 Side-Effect Free SQL-Script Procedures

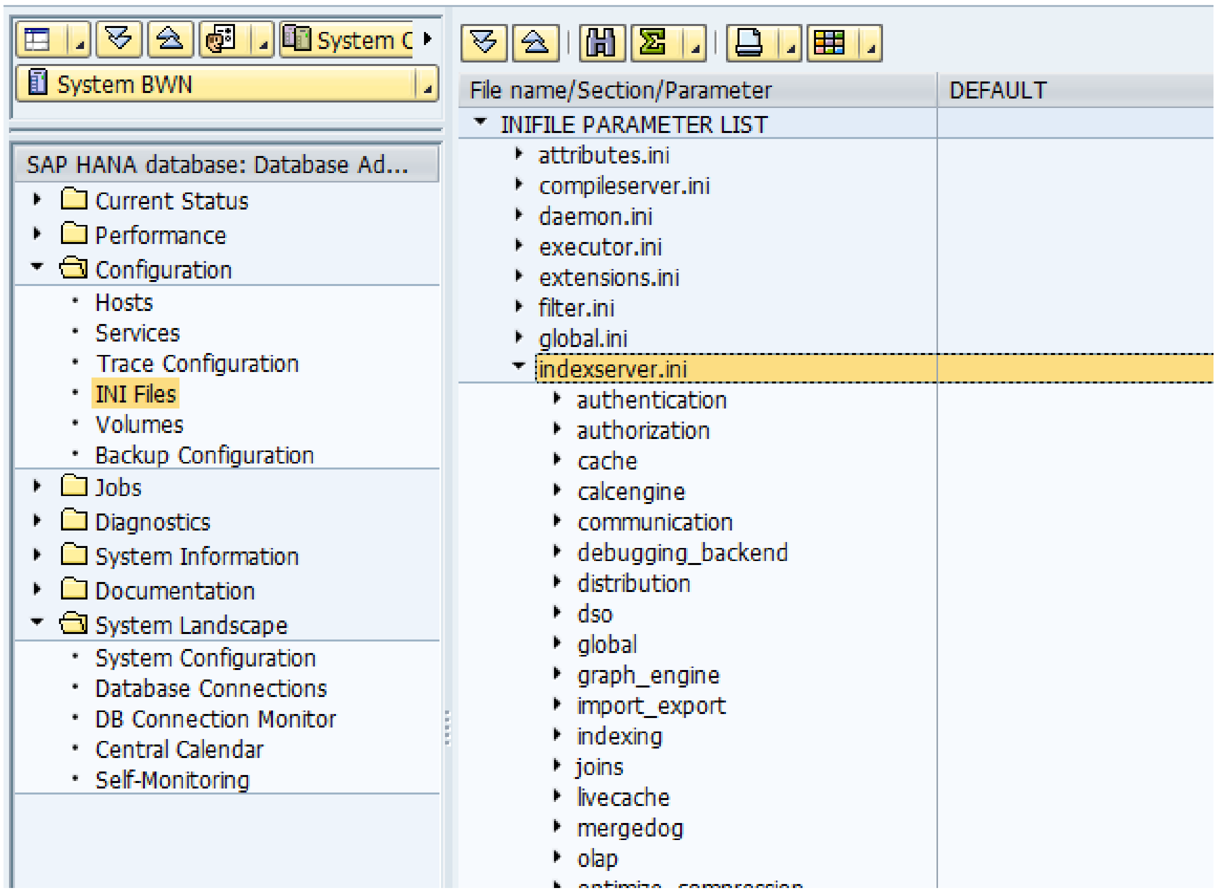
By default in planning only those SQL-Script procedures are allowed that do not change any data other than the ‘out-going’ table. Those SQL-Script procedures are called side-effect free as they do not change any table that is not declared as an outgoing table in the interface of the procedure. Thus it can be assured that the SQL-Script does not have un-wanted effects. In addition those procedures have to explicitly declare that they are side-effect free by adding the statement ‘READS SQL DATA’ at the beginning of the procedure:



```
CREATE PROCEDURE SAPBWN.ZGS_SIMPLE_COPY( IN I_VIEW SAPBWN.ZGS_IC5A_DEMO,
     I_FROM_IPCNTRY NVARCHAR (000003) ,
     I_TO_IPCNTRY NVARCHAR (000003) ,
     OUT E_T_TABLE SAPBWN.ZGS_IC5A_DEMO ) LANGUAGE SQLSCRIPT READS SQL DATA AS
begin
```

On the other hand in some cases it is necessary to change data base tables via a SQL-Script planning function or to use local temporary tables to hold intermediate results. Such procedures are NOT side effect free. The system has to specifically allow such procedures if they are necessary.

The setting can be either set from the ABAP server in transaction DBACOCKPIT or directly in HANA. When using DBACOCKPIT start the transaction, expand the entry ‘Configuration’ and double-click on ‘INI Files’. On the right hand side expand the node ‘indexserver.ini’ and look for the entry ‘planningengine’.



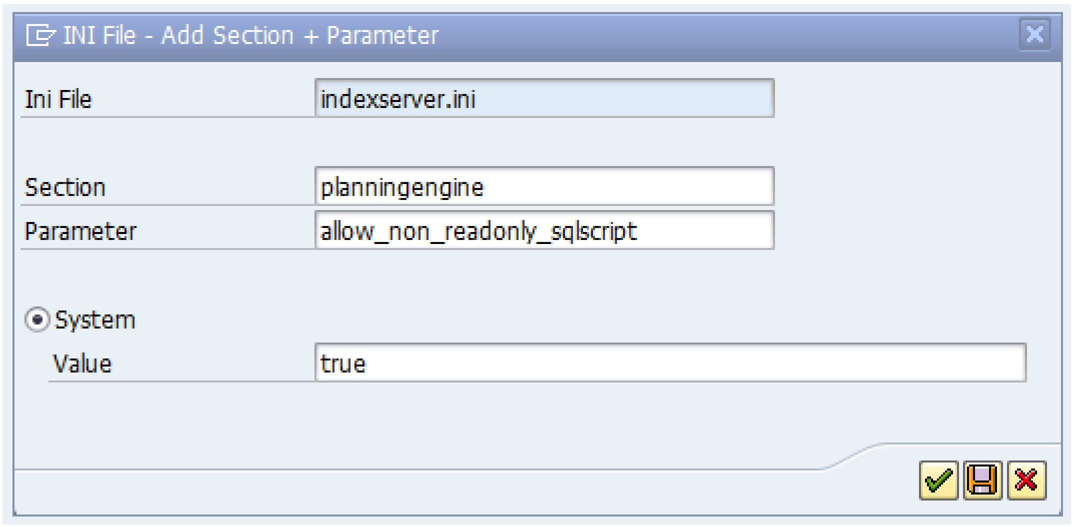
System BWN

SAP HANA database: Database Ad...

- Current Status
- Performance
- Configuration
  - Hosts
  - Services
  - Trace Configuration
  - INI Files
  - Volumes
  - Backup Configuration
- Jobs
- Diagnostics
- System Information
- Documentation
- System Landscape
  - System Configuration
  - Database Connections
  - DB Connection Monitor
  - Central Calendar
  - Self-Monitoring

File name/Section/Parameter	DEFAULT
INIFILE PARAMETER LIST	
attributes.ini	
compileserver.ini	
daemon.ini	
executor.ini	
extensions.ini	
filter.ini	
global.ini	
indexserver.ini	
authentication	
authorization	
cache	
calcengine	
communication	
debugging_backend	
distribution	
dso	
global	
graph_engine	
import_export	
indexing	
joins	
livecache	
mergedog	
olap	

If the entry ‘planningengine’ is not available you have to create the entry. Double-click on the entry ‘indexserver.ini’ and enter the following entries in the popup.



INI File - Add Section + Parameter

Ini File	indexserver.ini
Section	planningengine
Parameter	allow_non_readonly_sqlscript
● System	
Value	true

Confirm the popup by pressing the ‘save’ symbol.

If the parameter is already there and has the entry ‘false’ or ‘no’ you have to change the parameter value to ‘true’ (or ‘yes’).

Even if most of our following examples are side-effect free we have allowed non side-effect free procedures for our examples below and thus do not add the statement ‘READS SQL DATA’ in all of the sample procedures.

3.3 Create a Simple Copy Function

In our first example we want to create a simple copy function in a SQL-Script procedure. We want to copy data from one year to another year. It should be able to specify the source and target year from the planning function.

We start with the body for the procedure. The simplest body for a procedure looks like follows:

```
Create procedure SAPBWN.ZCOPY_SIMP(
     IN I_VIEW SAPBWN.ZSQL1,
    OUT E_T_TABLE SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS

begin

/* some coding*/
end;
```

As we already explained above the procedure has an ‘in’-table containing the selected data and an ‘out’-table containing the after image or delta (in the implementing ABAP class this has to be specified).

As we want to specify the from- and the to-value for the year in the planning function we need two more parameters and our procedure body looks like this:

```
Create procedure SAPBWN.ZCOPY_SIMP(
     IN I_VIEW SAPBWN.ZSQL1,
        I_FROM_YEAR NVARCHAR (4),
        I_TO_YEAR  NVARCHAR (4),
    OUT E_T_TABLE SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS
begin

/* some coding*/

end;
```

Please note that the report RSPLS_SQL_SCRIPT_TOOL can also create a template for such a procedure body. If you have specified the planning function type (with the parameters) it even creates the parameters in addition to the in- and out-table.

Now we can concentrate on the function logic. What should the planning function do? It should first read the existing data in the table i_view– but only such data where the year is equal to the ‘from’ year. In SQL-Script this is written as:

```
Select * FROM :i_view WHERE CALYEAR = :I_FROM_YEAR
```

Now the year should be set to the ‘to’ year. As we have to change the value of one result field in the select statement we cannot use ‘*’ anymore but have to specify all fields:

```
Select    CALMONTH2,:I_TO_YEAR as CALYEAR, Z_PROD, Z_AMOUNT, Z_QUAN
FROM :i_view WHERE CALYEAR = :I_FROM_YEAR
```

The result of this select should be returned in the out-table. Thus we have our final SQL-Script as:

```
Create procedure SAPBWN.ZCOPY_SIMP(
     IN I_VIEW SAPBWN.ZSQL1,
        I_FROM_YEAR NVARCHAR (4),
        I_TO_YEAR  NVARCHAR (4),
    OUT E_T_TABLE SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS

begin

E_T_TABLE = select
    CALMONTH2,
    :I_TO_YEAR as CALYEAR,
    Z_PROD,
    Z_AMOUNT,
    Z_QUAN
FROM :i_view
WHERE CALYEAR = :I_FROM_YEAR
;

end
;
```

If you want to test your logic just copy and paste the following coding to your SQL console and execute it – the function is copying data from 2014 to 2015:

```
call SAPBWN.ZCOPY_SIMP(ZTSQL_DATA, '2014', '2015', ?)
```

Please note that if you press F8 then you will execute all coding in the console. If you want to execute only single statement then mark the statement and press F8.

If you just press F8 (without marking the procedure call) then the system will send you an error message as the procedure already exists as it tries to create it again.

Please also note that you cannot change a procedure directly. The easiest way to change a procedure is to delete and create it again. The correct statement for deleting a procedure is:

```
drop procedure SAPBWN.ZCOPY_SIMP;
```

If you put a ‘drop procedure’ before your ‘create procedure’ you are on the save side and can execute the entire SQL-Script in your console.

(see also appendix 4)

Remark: We have assumed for our planning function that the filter contains the source and the target data of the planning function. A copy function could also read the source data as reference data. If this should be implemented then the planning function type needs to have the option ‘with reference data’ set and then our procedure will have a second in-table: i_ref_view. This table contains the data in the reference data selection. The table has the same table type as the i_view table and you can access the table in the very same way.

Please keep in mind that our procedure generates deltas. That means that in the out-table we do not have to insert the existing records for 2014 and that our copy function adds the new value for 2015 to (potentially) existing records for 2015. If you want to overwrite existing values then you have to clear the existing value. Here is an example how this can be done:

In addition to all the copied records the out-table needs to contain a negative value of all the existing records for the target year. The SQL-statement for receiving all negative values from the in-table for a given year is:

```
select
    CALMONTH2,
    CALYEAR,
    Z_PROD,
    ((-1) * Z_AMOUNT) as Z_AMOUNT,
    ((-1) * Z_QUAN) as Z_QUAN
FROM :i_view
WHERE CALYEAR = :I_TO_YEAR
```

As we now need the added results of both select statements in our result table we have to use the ‘union’ statement in SQL-Script and our formula looks like:

```
Create procedure SAPBWN.ZCOPY_SIMP2(
     IN I_VIEW SAPBWN.ZSQL1,
        I_FROM_YEAR NVARCHAR (4),
        I_TO_YEAR  NVARCHAR (4),
    OUT E_T_TABLE SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS

begin

E_T_TABLE = select
    CALMONTH2,
    :I_TO_YEAR as CALYEAR,
    Z_PROD,
    Z_AMOUNT,
    Z_QUAN
FROM :i_view WHERE CALYEAR = :I_FROM_YEAR
union
select
    CALMONTH2,
    CALYEAR,
    Z_PROD,
    ((-1) * Z_AMOUNT) as Z_AMOUNT,
    ((-1) * Z_QUAN) as Z_QUAN
FROM :i_view WHERE CALYEAR = :I_TO_YEAR
;
end
;
```

If you test the procedure like above copying 2014 to 2015 we will see the same result as before (as we do not have any data for 2015 in our table. If you copy from 2013 to 2014 you will see that all 2014 data has been reversed. As we are working with deltas this will result in a deletion of the existing records when the procedure is used in a planning function in PAK.

(See appendix 5 for the coding)

3.4 Create a Function accessing External Data

In PAK/BW-IP custom defined planning functions or Fox formulas with ABAP function module calls are often used when ‘external’ data needs to be accessed – that is either data in an arbitrary database table or data in another aggregation level of the InfoCube or DataStore Object. We want to show how easy such a scenario can be realized with SQL-Script.

Remember that we have created a price table in chapter 3.1. For each product we now want to calculate the amount (=revenue) as a product of the quantity and the price. If we do not find a price for a product then nothing should be done. Thus we have to join the two tables: the data table and the price table. For each product we have to look for the price so the join will be done over the product. In our procedure we use an additional feature – we can define abbreviations for the involved tables so we do not have to use the full table name. Our data table will be ‘a’, our price table will be ‘b’.

Please keep in mind that we return (in this case) delta values and not an after image. We do not want to change the key figure Z_QUAN so we have to return a (delta of) 0 for this key figure. We want to set the amount to the new value so we have to subtract the existing value to receive the correct delta.

```
CREATE PROCEDURE SAPBWN.ZCALC_REV( IN I_VIEW SAPBWN.ZSQL1,
    OUT E_T_TABLE SAPBWN.SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS
begin E_T_TABLE = select
    a.CALMONTH2,
    a.CALYEAR,
    a.Z_PROD,
   ( (a.Z_QUAN * b.PRICE) - a.Z_AMOUNT)  as Z_AMOUNT,
    0 as Z_QUAN
from :i_view as a
inner join SAPBWN.ZTSQL_PRICE as b on a.Z_PROD = b.Z_PROD
;

end
;
```

(see also appendix 6)

Using data from another aggregation level within a procedure is now possible (from SAP BW 7.40, SP06 , BW 7.30 SP08 and BW 7.31 SP06 on - notes 1976514 and 1976522 are also necessary). The procedure receives the data from the additional aggregation level(s) in additional importing tables. Thus the tables can be accessed in the very same way as the data table i_view. There is also a how to paper available showing how to set up such a scenario:

How to… Use Data from Another Aggregation Level in a PAK SQL-Script procedure – Implemented as an AMDP Example (https://scn.sap.com/docs/DOC-53376).

3.5 Create a Simple Distribution Function

In this chapter we want to build a distribution function and want to introduce some more programming concepts.

We want to evenly (re-)distribute the amounts within a year and month to all products.

First of all we need to know what the total values are. Remember that we do not want to distribute from one month/year to another but want the values to stay within their month/year. In a standard planning function we would solve this by only using the product as a characteristic to be changed. The system form one package for each month/year and would execute the Fox coding for each package. In our procedure we work with one package only so we have to pay attention on the year and month.

So in the first step we need the totals value for each month and year. We can use a simple SQL-query (result of a select statement) for this and later keep on working with the result. Thus we do not have to use any local table (we will show later how this is done). You can use those results to further select from them and to combine them via unions or joins with other results or tables. You cannot do any table operation like inserting data (via insert…) or deleting data from this result set [if you pass the content of such a table on to a subroutine then you can access the records directly in the subroutine – see below).

Please note that our out-table also is filled as such a result of a SQL-query. That means that you cannot use insert statements in order to write data into the out-table. If you need such a concept then you will have to use a local temporary table (see below).

Our SQL-Script for a result containing the totals for each month and year looks like this:

```
total_vals = select
         CALMONTH2,
         CALYEAR,
         SUM(Z_AMOUNT) as total
     from :i_view group by CALMONTH2, CALYEAR;
```

Please note that in order to calculate the total value over a number of records in ABAP you would have to loop over the entire table and add up the values. In SQL-Script we can obtain the result with a single statement.

Now we have to know the number of products per month and year so we can calculate amount each product will receive.

```
count_prod = select
        CALMONTH2,
        CALYEAR,
        COUNT(Z_PROD) as count
        from :i_view
        group by CALMONTH2, CALYEAR;
```

So the value each product (per month and year) will receive is:

```
/* calculate the value*/
 dist_values = select
        a.CALMONTH2,
        a.CALYEAR,
        (a.total / b.count) as dist_value
        from :total_vals as a
        inner join
        :count_prod as b on
        a.CALMONTH2 = b.CALMONTH2 and
        a.CALYEAR = b.CALYEAR;
```

As you can see we can access the results of our SQL-queries (total_vals and count_prod) in the same way as our data table i_view.

The last steps could also be less steps and a more compact way. But this makes the coding harder to read. The system will parse the SQL-Script anyway and will optimize the coding. Thus we can introduce some intermediate steps without harming the performance.

Now we can fill the out-table. Please remember that we decided to hand out deltas so we have to revert the existing values for amount as well.

```
  E_T_TABLE = select
   a.CALMONTH2,
   a.CALYEAR,
   a.Z_PROD,
   (b.dist_value - a.Z_AMOUNT) as Z_AMOUNT,
   0 as Z_QUAN
   from :i_view as a
   inner join
   :dist_values as b on
   a.CALMONTH2 = b.CALMONTH2 and
   a.CALYEAR = b.CALYEAR
   ;
```

(also see appendix 7 for the coding)

If you want to check the correctness of the calculation you can also hand out an after image by the statement:

```
  E_T_TABLE = select
   a.CALMONTH2,
   a.CALYEAR,
   a.Z_PROD,
   b.dist_value as Z_AMOUNT,
   a.Z_QUAN
   from :i_view as a
   inner join
   :dist_values as b on
   a.CALMONTH2 = b.CALMONTH2 and
   a.CALYEAR = b.CALYEAR
   ;
```

3.6 Create a Complex Distribution Function 1

In this chapter we will program a more complex planning function. Our aim is to show you how to handle local temporary tables. Programming variables and cursors are used (even if the logic might also be implemented with declarative logic).

Our planning function should do a distribution of the key figure Z_QUAN. For each month and year it should distribute the value on '#' to the other products in the same ratio as defined by the current ratio. To add to the complexity we assume that we want to distribute INTEGER numbers (as we are working with pieces which should usually be integers…). Obviously we cannot stick to the exact distribution rate but have to use some rounding. But here it is getting tricky.

Imagine we have 4 products and the following (more simplifies records):

Product	Quantity
#	4
P1	1
P2	1
P3	1
Total	7

If we first distribute and then round we get the following case: for each product the new value would be 2 1/3 which rounds to 2. So after the execution of the algorithm the data looks like this:

P1	2
P2	2
P3	2
Total	6

We have a distribution that matches the distribution factors correctly but we have 'lost' one piece. Obviously we have to find an algorithm that tries to stay as close as possible to the existing distribution but does a cleverer rounding.

The algorithm that we choose works as follows: we calculate the delta value for our first product. This is:

(value on '#') * (original value on P1) / (total value over all products)

In our case the delta is 1 1/3. We NOW do the rounding. So P1 gets a delta of 1. We conclude that we now can calculate:

(new value to be distributed to remaining products) = (value on '#') – (delta for P1)

But in order to calculate the proper proportion for the next product we also have to use a new total:

(new total) = (original total) – (original value on P1).

(there are several similar options how to design the algorithm)

Obviously we have a recursive algorithm and we cannot realize this function in a procedure with declarative logic. We want to implement the algorithm using local temporary tables and loops over these tables using a cursor.

As in our planning function above we do the same logic for each month and year.

We first define the cursor and some programming variables. In our algorithm we will go step by step over the products and for each product will calculate the delta. The current remainder (per month and year) that is to be distributed and the current total (again per month and year) used to calculate the ratio is kept in a table. Thus we need a cursor to loop over all products in the data table except the product '#' which has an internal value ' '.

```
v_init_prod nvarchar(12) := ' ';
 cursor c_cursor1 for select * from :i_view where Z_PROD <> :v_init_prod;
 v_sum decimal(17,3);
 v_rem decimal(17,3);
 v_value decimal(17,3);
```

In the next step we calculate the totals per month and year, but this time without the '#' value (as we do not want to distribute the the '#' value).

```
total_vals = select
         CALMONTH2,
         CALYEAR,
         SUM(Z_QUAN) as total
     from :i_view
     where Z_PROD <> :v_init_prod
     group by CALMONTH2, CALYEAR;
```

We now need two temporary tables: one table should hold the values of the remainders to be distributed and the remaining total (for the ratio) per month and year. In each step we will update the table with the new remainder and total.

As we are working in a loop we need to insert our newly calculated deltas for the quantity for each product into a table. As the out-table does not allow any inserts we need an intermediate table we can insert record in and from which we can later fill the out-table. Please note the naming conventions for temporary local tables (starting with '#').

```
CREATE
 LOCAL TEMPORARY
     TABLE
     #L_REM
    ( CALMONTH2 NVARCHAR(2),
    CALYEAR NVARCHAR(4),
    sum decimal(17,3),
    rem DECIMAL(000017, 000003) ) ;

CREATE LOCAL TEMPORARY TABLE #E_TMP (
     CALMONTH2 NVARCHAR(3),
    CALYEAR NVARCHAR(4),
    Z_PROD NVARCHAR(60),
    Z_QUAN DECIMAL(17,3) ) ;
```

Before we can start our loop we need to fill our remainder table with the starting values. At the beginning the total is the total over all products unequal to '#' (we have calculated that in total_vals) and the remainder that is to be distributed is the full value on '#'.

```
insert into #l_REM
    select
    a.CALMONTH2,
    a.CALYEAR,
    a.total as sum,
    b.Z_QUAN as rem
 from :total_vals as a
 inner join
      :i_view as b on
      a.CALMONTH2 = b.CALMONTH2 and
      a.CALYEAR = b.CALYEAR

      where b.Z_PROD = :v_init_prod;
```

We now are ready to loop over the data table and for the month, year, and product that is in the work area we read the corresponding record with the current remainder and total.

```
for r as c_cursor1 do

select sum, rem into v_sum, v_rem from #L_REM where CALMONTH2 = r.CALMONTH2 and
CALYEAR = r.CALYEAR;
```

After a safety check that the total is not zero we calculate the current delta value and round it.

```
if :v_sum <> 0 then
/* calculate the value*/
v_value := r.Z_QUAN * :v_rem / :v_sum;
v_value := round( :v_value, 0, ROUND_HALF_UP);
```

Now we can subtract the value from the remainder and calculate the new total.

```
v_sum := :v_sum - r.Z_QUAN;
v_rem := :v_rem - :v_value;
```

Finally we update the remainder and the total in the table and insert our delta record into the table of intermediate results.

```
/* update the remainder table*/
update #L_REM set sum = :v_sum, rem = :v_rem where CALMONTH2 = r.CALMONTH2 and
CALYEAR = r.CALYEAR;
/* update the data table with delta!*/
insert into #E_TMP values (r.CALMONTH2,r.CALYEAR, r.Z_PROD, :v_value);
end if;

end for;
```

Now we can fill the out-table from the table of the intermediate results. As we return deltas we have to also return a negative value for all records for the product '#'. For the other records we already have correctly calculated the delta.

```
e_t_table = select
  CALMONTH2,
  CALYEAR,
  Z_PROD,
  0 as Z_AMOUNT,
  Z_QUAN
 from #E_TMP
  union select
  CALMONTH2,
  CALYEAR,
  Z_PROD,
  0 as Z_AMOUNT,
  (-1) * Z_QUAN as Z_QUAN
  from :i_view where Z_PROD = :v_init_prod;
```

Finally we have to delete the local temporary table as otherwise we will get an error once we re-run the procedure.

```
drop table  #l_rem;
drop table #E_TMP;
```

(Have a look at appendix 8 for the complete coding)

3.7 Create a Complex Distribution Function 2

We want to stick to our distribution function and now give you an example how you can use subroutines in SQL-Script. A subroutine is just a SQL-Script procedure. Thus if we want to use tables in the interface of the subroutine we have to define a table type.

In our example we will show you that it is possible to change the result of a SQL-query – not directly but in a subroutine. In the subroutine you have to specify the structure of the in-table which must match the structure of the SQL-query.

Let us first create the type for our sub routine. It matches the type we have used for our local temporary table in the example above.

```
CREATE TYPE "SAPBWN"."ZREM" AS TABLE
  ( "CALMONTH2" NVARCHAR(2),
    "CALYEAR" NVARCHAR(4),
    "SUM" DECIMAL(17,3),
    "REM" DECIMAL(17,3))
```

Now we create the procedures for reading and updating the remainder table.

```
CREATE PROCEDURE SAPBWN.ZREAD_REM(
    IN   I_REM SAPBWN.ZREM,
    IN   I_CALMONTH2 nvarchar(2),
    IN   I_CALYEAR nvarchar(4),
   OUT  E_SUM decimal(17,3),
   OUT  e_rem decimal(17,3) ) LANGUAGE SQLSCRIPT AS

cursor c_cursor (v_CALMONTH2 NVARCHAR(2), v_CALYEAR NVARCHAR(4))
                  for select SUM, REM from :i_REM
                  where CALMONTH2 = :v_CALMONTH2 and CALYEAR = :v_CALYEAR;

begin
```

```
open c_cursor(:I_CALMONTH2, :i_CALYEAR);

fetch c_cursor into e_sum, e_rem;
close c_cursor;

end;
```

Please note that we use a cursor with two variables. Such a cursor can simulate a ‘loop at… where…’ construct in ABAP.

```
CREATE PROCEDURE SAPBWN.ZUPDATE_REM(
    IN I_REM_tab SAPBWN.ZREM,
    IN   I_CALMONTH2 nvarchar(2),
    IN   I_CALYEAR nvarchar(4),
    IN   i_sum decimal(17,3),
    IN   i_rem decimal(17,3),
   OUT E_REM SAPBWN.ZREM) LANGUAGE SQLSCRIPT AS

begin

e_rem =
Select * from :i_rem_tab where CALMONTH2 <> :I_CALMONTH2 or CALYEAR <> :I_CALYEAR
  union
  select
    CALMONTH2,
    CALYEAR,
    :i_sum,
    :i_rem
  from :i_rem_tab where CALMONTH2 = :I_CALMONTH2 and CALYEAR = :I_CALYEAR;

end;
```

We can now implement our procedure. It is very similar to our first distribution function. The only difference is that we do not use a local temporary table for our remainders table but a result of an SQL-query. We read and update the table via procedures. Other than that the coding is identical. Please have a look at appendix 9 for the coding itself.

3.8 SQL-Script Debugging

Now when you have written your SQL-Script procedure you probably want to debug the application in order to test it and find errors.

Before SAP HANA SP7 it is not possible to debug the SQL-Script procedure directly in the schema. You have to create a project and get a local copy of the procedure in order to debug the procedure. With SAP HANA SP7 it is now also possible to debug a procedure that is defined in a schema and it is also possible to debug AMDPs (ABAP managed database procedures) which is started from ABAP.

You have two options to debug your procedure. You can use the direct debugging in the schema and use the test data you have created above for the input tables. This is probably the easier option because you will probably use less data and because you do not have to jump between technologies. Debugging the AMDP has the advantage that you do not have to create a test data environment and that you can test and debug the procedure with the ‘real’ data in the InfoCube/planning buffer.

You can find videos about the two options in SDN:

- Debugging a catalogue (schema) procedure: http://www.saphana.com/docs/DOC-4473
- Debugging an AMDP: http://scn.sap.com/docs/DOC-51790

When you are using the first option you have to keep in mind that you cannot directly debug a procedure with a tabular input. But as we have our test data this problem can be easily solved. We create a new procedure that is used to call the procedure we want to debug.

```
create procedure zdebug_int2 ( ) LANGUAGE SQLSCRIPT AS

begin

view = select * from "SAPBWN"."ZTSQL_DATA";

create local temporary table #tmp like "SAPBWN"."ZTSQL_DATA";

call SAPBWN.ZDIST_INT2 (:view, #tmp);

drop table #tmp;

end;
```

(see also appendix 10)

Before you start the debugging you can set a break-point in the ‘debug’-procedure and a break-point at the beginning of the procedure you finally want to debug (in the above case ZDIST_INT2). Please note that we do not have a single step in SQL-Script debugging (yet). You can use F8 to jump to the next break-point. You can set break-points also from within the debugger. In the debugger you can see the content of each programming variable and can also see the content of tables (as SQL-queries, temporary local tables etc.).

3.9 Sending Messages from SQL-Script

In some cases it is necessary to return some information from your custom defined planning functions – for example if something went wrong when executing the function logic or some information that a certain check was performed successfully. This information should be visible in the normal message log displayed after the execution of a planning function. In a custom defined planning function defined in ABAP you can return a message table containing this information. In a planning function defined in SQL-Script we have a very similar technique: you can add another ‘out-‘table (E_T_MESG) that contains the messages you want to send. You can find more information on sending messages and also on how to fill the message table in SAP Note 2011847.

Because of a bug in order to successfully send messages it is necessary to implement note 2111553.

4. Appendix

4.1 SQL-Script for Creating the Data Table

```
drop table "SAPBWN"."ZTSQL_DATA";

CREATE COLUMN TABLE "SAPBWN"."ZTSQL_DATA" (
     "CALMONTH2" NVARCHAR(2),
     "CALYEAR" NVARCHAR(4),
     "Z_PROD" NVARCHAR(60),
     "Z_AMOUNT" DECIMAL(17,2),
     "Z_QUAN" DECIMAL(17,3) ) ;

INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('01','2014',' ',10.00 ,10.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('01','2014','P01',100.00 ,10.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('01','2014','P02',200.00 ,10.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('01','2014','P03',300.00 ,10.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('01','2014','P04',400.00 ,10.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('01','2014','P05',500.00 ,10.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('02','2014',' ',2.00 ,2.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('02','2014','P01',10.00 ,1.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('02','2014','P02',20.00 ,1.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('02','2014','P03',30.00 ,1.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('02','2014','P04',40.00 ,1.000 );
INSERT INTO SAPBWN.ZTSQL_DATA VALUES ('02','2014','P05',50.00 ,1.000 );
SELECT  * FROM "SAPBWN"."ZTSQL_DATA"
```

4.2 SQL-Script for Creating the Price Table

```
drop table "SAPBWN"."ZTSQL_PRICE";

CREATE COLUMN TABLE "SAPBWN"."ZTSQL_PRICE" (
      "Z_PROD" NVARCHAR(60),
     "PRICE" DECIMAL(17,2) ) ;

INSERT INTO SAPBWN.ZTSQL_PRICE VALUES ('P01',11.00 );
INSERT INTO SAPBWN.ZTSQL_PRICE VALUES ('P02',12.00 );
INSERT INTO SAPBWN.ZTSQL_PRICE VALUES ('P03',13.00 );
INSERT INTO SAPBWN.ZTSQL_PRICE VALUES ('P04',14.00 );
INSERT INTO SAPBWN.ZTSQL_PRICE VALUES ('P05',15.00 );
SELECT  * FROM "SAPBWN"."ZTSQL_PRICE"
```

4.3 SQL-Script for Creating the Table Types

```
CREATE TYPE SAPBWN.ZSQL1 AS TABLE (
  CALMONTH2 NVARCHAR (000002),
  CALYEAR NVARCHAR (000004),
  Z_PROD NVARCHAR (000060),
  Z_AMOUNT DECIMAL (000017, 000002),
  Z_QUAN DECIMAL (000017, 000003))
```

4.4 SQL-Script for a Simple Copy Function

```
drop procedure SAPBWN.ZCOPY_SIMP;

Create procedure SAPBWN.ZCOPY_SIMP(
     IN I_VIEW SAPBWN.ZSQL1,
        I_FROM_YEAR NVARCHAR (4),
        I_TO_YEAR  NVARCHAR (4),
    OUT E_T_TABLE SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS

begin

E_T_TABLE = select
    CALMONTH2,
    :I_TO_YEAR as CALYEAR,
    Z_PROD,
    Z_AMOUNT,
    Z_QUAN
FROM :i_view
WHERE CALYEAR = :I_FROM_YEAR
;

end
;


call SAPBWN.ZCOPY_SIMP(ZTSQL_DATA, '2014', '2015', ?)
```

4.5 SQL-Script for a Simple Copy Function with Overwrite

```
drop procedure SAPBWN.ZCOPY_SIMP2;

Create procedure SAPBWN.ZCOPY_SIMP2(
     IN I_VIEW SAPBWN.ZSQL1,
        I_FROM_YEAR NVARCHAR (4),
        I_TO_YEAR  NVARCHAR (4),
    OUT E_T_TABLE SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS

begin

E_T_TABLE = select
    CALMONTH2,
    :I_TO_YEAR as CALYEAR,
    Z_PROD,
    Z_AMOUNT,
    Z_QUAN
FROM :i_view WHERE CALYEAR = :I_FROM_YEAR
union
select
    CALMONTH2,
    CALYEAR,
    Z_PROD,
    ((-1) * Z_AMOUNT) as Z_AMOUNT,
    ((-1) * Z_QUAN) as Z_QUAN
FROM :i_view WHERE CALYEAR = :I_TO_YEAR
;
end
;


call SAPBWN.ZCOPY_SIMP2(ZTSQL_DATA, '2013', '2014', ?)
```

4.6 SQL-Script for a Price Calculation

```
drop procedure SAPBWN.ZCALC_REV;


CREATE PROCEDURE SAPBWN.ZCALC_REV( IN I_VIEW SAPBWN.ZSQL1,
    OUT E_T_TABLE SAPBWN.SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS
begin E_T_TABLE = select
    a.CALMONTH2,
    a.CALYEAR,
    a.Z_PROD,
   ( (a.Z_QUAN * b.PRICE) - a.Z_AMOUNT)  as Z_AMOUNT,
    0 as Z_QUAN
from :i_view as a
inner join SAPBWN.ZTSQL_PRICE as b on a.Z_PROD = b.Z_PROD
;

end
;

call SAPBWN.ZCALC_REV (ZTSQL_DATA, ?)
```

4.7 SQL-Script for a Simple Distribution

```
drop procedure SAPBWN.ZDIST_SIMP;

CREATE PROCEDURE SAPBWN.ZDIST_SIMP( IN I_VIEW SAPBWN.ZSQL1,
    OUT E_T_TABLE SAPBWN.SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS
begin


/* get the sum of all values per month and year*/
total_vals = select
         CALMONTH2,
         CALYEAR,
         SUM(Z_AMOUNT) as total
     from :i_view group by CALMONTH2, CALYEAR;

/* get the number of products per month and year*/
count_prod = select
        CALMONTH2,
        CALYEAR,
        COUNT(Z_PROD) as count
        from :i_view
        group by CALMONTH2, CALYEAR;

/* calculate the value*/
 dist_values = select
        a.CALMONTH2,
        a.CALYEAR,
        (a.total / b.count) as dist_value
        from :total_vals as a
        inner join
        :count_prod as b on
        a.CALMONTH2 = b.CALMONTH2 and
        a.CALYEAR = b.CALYEAR;

/* Set the value for each product*/
  E_T_TABLE = select
   a.CALMONTH2,
   a.CALYEAR,
   a.Z_PROD,
   (b.dist_value - a.Z_AMOUNT) as Z_AMOUNT,
   0 as Z_QUAN
   from :i_view as a
   inner join
   :dist_values as b on
   a.CALMONTH2 = b.CALMONTH2 and
   a.CALYEAR = b.CALYEAR
   ;

end;

call SAPBWN.ZDIST_SIMP (ZTSQL_DATA, ?)
```

4.8 SQL-Script for a Complex Distribution

```
drop procedure SAPBWN.ZDIST_INT;

CREATE PROCEDURE SAPBWN.ZDIST_INT( IN I_VIEW SAPBWN.ZSQL1,
    OUT E_T_TABLE SAPBWN.SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS

 v_init_prod nvarchar(12) := ' ';
 cursor c_cursor1 for select * from :i_view where Z_PROD <> :v_init_prod;
 v_sum decimal(17,3);
 v_rem decimal(17,3);
 v_value decimal(17,3);


begin


/* get the sum of all values <> '#' per month and year*/
total_vals = select
         CALMONTH2,
         CALYEAR,
         SUM(Z_QUAN) as total
     from :i_view
     where Z_PROD <> :v_init_prod
     group by CALMONTH2, CALYEAR;


/* local table for remaining value and total*/
 drop table  #l_rem;

 CREATE
 LOCAL TEMPORARY
     TABLE
     #L_REM
    ( CALMONTH2 NVARCHAR(2),
    CALYEAR NVARCHAR(4),
    sum decimal(17,3),
    rem DECIMAL(000017, 000003) ) ;


/* create local table for values*/
 drop table #E_TMP;
 CREATE LOCAL TEMPORARY TABLE #E_TMP (
     CALMONTH2 NVARCHAR(3),
    CALYEAR NVARCHAR(4),
    Z_PROD NVARCHAR(60),
    Z_QUAN DECIMAL(17,3) ) ;


/* fill in the initial remainders*/
insert into #l_REM
    select
    a.CALMONTH2,
    a.CALYEAR,
    a.total as sum,
    b.Z_QUAN as rem
 from :total_vals as a
 inner join
      :i_view as b on
      a.CALMONTH2 = b.CALMONTH2 and
      a.CALYEAR = b.CALYEAR
      where b.Z_PROD = :v_init_prod;



/* loop over the existing records*/
for r as c_cursor1 do

select sum, rem into v_sum, v_rem from #L_REM where CALMONTH2 = r.CALMONTH2 and
CALYEAR = r.CALYEAR;

if :v_sum <> 0 then
/* calculate the value*/

v_value := r.Z_QUAN * :v_rem / :v_sum;
v_value := round( :v_value, 0, ROUND_HALF_UP);

v_sum := :v_sum - r.Z_QUAN;
v_rem := :v_rem - :v_value;

/* update the remainder table*/
```

```
update #L_REM set sum = :v_sum, rem = :v_rem where CALMONTH2 = r.CALMONTH2 and
CALYEAR = r.CALYEAR;


/* update the data table with delta!*/
insert into #E_TMP values (r.CALMONTH2,r.CALYEAR, r.Z_PROD, :v_value);

end if;

end for;


e_t_table = select
  CALMONTH2,
  CALYEAR,
  Z_PROD,
  0 as Z_AMOUNT,
  Z_QUAN
 from #E_TMP
  union select
  CALMONTH2,
  CALYEAR,
  Z_PROD,
  0 as Z_AMOUNT,
  (-1) * Z_QUAN as Z_QUAN
  from :i_view where Z_PROD = :v_init_prod;


drop table  #l_rem;
drop table #E_TMP;

end;


call SAPBWN.ZDIST_INT (ZTSQL_DATA, ?)
```

4.9 SQL-Script for a Complex Distribution 2

```
drop type "SAPBWN"."ZREM";

CREATE TYPE "SAPBWN"."ZREM" AS TABLE
  ( "CALMONTH2" NVARCHAR(2),
     "CALYEAR" NVARCHAR(4),
     "SUM" DECIMAL(17,3),
     "REM" DECIMAL(17,3));


drop procedure SAPBWN.ZREAD_REM;


CREATE PROCEDURE SAPBWN.ZREAD_REM(
    IN   I_REM SAPBWN.ZREM,
    IN   I_CALMONTH2 nvarchar(2),
    IN   I_CALYEAR nvarchar(4),
   OUT  E_SUM decimal(17,3),
   OUT  e_rem decimal(17,3) ) LANGUAGE SQLSCRIPT AS
```

```
cursor c_cursor (v_CALMONTH2 NVARCHAR(2), v_CALYEAR NVARCHAR(4))
                 for select SUM, REM from :i_REM
                 where CALMONTH2 = :v_CALMONTH2 and CALYEAR = :v_CALYEAR;

begin

open c_cursor(:I_CALMONTH2, :i_CALYEAR);

fetch c_cursor into e_sum, e_rem;
close c_cursor;

end;


drop procedure SAPBWN.ZUPDATE_REM;


CREATE PROCEDURE SAPBWN.ZUPDATE_REM(
    IN I_REM_tab SAPBWN.ZREM,
    IN   I_CALMONTH2 nvarchar(2),
    IN   I_CALYEAR nvarchar(4),
    IN   i_sum decimal(17,3),
    IN   i_rem decimal(17,3),
   OUT E_REM SAPBWN.ZREM) LANGUAGE SQLSCRIPT AS

begin

e_rem =
Select * from :i_rem_tab where CALMONTH2 <> :I_CALMONTH2 or CALYEAR <> :I_CALYEAR
  union
  select
    CALMONTH2,
    CALYEAR,
    :i_sum,
    :i_rem
  from :i_rem_tab where CALMONTH2 = :I_CALMONTH2 and CALYEAR = :I_CALYEAR;

end;



drop procedure SAPBWN.ZDIST_INT2;

CREATE PROCEDURE SAPBWN.ZDIST_INT2( IN I_VIEW SAPBWN.ZSQL1,
    OUT E_T_TABLE SAPBWN.SAPBWN.ZSQL1 ) LANGUAGE SQLSCRIPT AS

 v_init_prod nvarchar(12) := ' ';
 cursor c_cursor1 for select * from :i_view where Z_PROD <> :v_init_prod;
 v_sum decimal(17,3);
 v_rem decimal(17,3);
 v_value decimal(17,3);


begin
```

```
/* get the sum of all values <> '#' per month and year*/
total_vals = select
         CALMONTH2,
         CALYEAR,
         SUM(Z_QUAN) as total
     from :i_view
     where Z_PROD <> :v_init_prod
     group by CALMONTH2, CALYEAR;


/* create local table for values*/
 drop table #E_TMP;

 CREATE LOCAL TEMPORARY TABLE #E_TMP (
     CALMONTH2 NVARCHAR(3),
    CALYEAR NVARCHAR(4),
    Z_PROD NVARCHAR(60),
    Z_QUAN DECIMAL(17,3) ) ;


/* fill in the initial remainders*/
l_t_rem = select
    a.CALMONTH2,
    a.CALYEAR,
    a.total as sum,
    b.Z_QUAN as rem
 from :total_vals as a
 inner join
      :i_view as b on
      a.CALMONTH2 = b.CALMONTH2 and
      a.CALYEAR = b.CALYEAR
      where b.Z_PROD = :v_init_prod;

/* add to single value: new = round (value * rem / sum) .
   new sum = sum - new
   new remainder rem = rem - new  */

/* loop over the existing records*/
for r as c_cursor1 do


/* read from the remainder table*/
call SAPBWN.ZREAD_REM ( :L_t_REM, r.CALMONTH2, r.CALYEAR, v_sum, v_rem);


if :v_sum <> 0 then
/* calculate the value*/

v_value := r.Z_QUAN * :v_rem / :v_sum;
v_value := round( :v_value, 0, ROUND_HALF_UP);

v_sum := :v_sum - r.Z_QUAN;
v_rem := :v_rem - :v_value;

/* update the remainder table*/
call SAPBWN.ZUPDATE_REM ( :L_T_REM, r.CALMONTH2, r.CALYEAR, v_sum, v_rem,
:l_T_rem);
```

```
/* update the data table with delta!*/
insert into #E_TMP values (r.CALMONTH2,r.CALYEAR, r.Z_PROD, :v_value);

end if;

end for;


e_t_table = select
  CALMONTH2,
  CALYEAR,
  Z_PROD,
  0 as Z_AMOUNT,
  Z_QUAN
 from #E_TMP
  union select
  CALMONTH2,
  CALYEAR,
  Z_PROD,
  0 as Z_AMOUNT,
  (-1) * Z_QUAN as Z_QUAN
  from :i_view where Z_PROD = :v_init_prod;

drop table #E_TMP;

end;

call SAPBWN.ZDIST_INT2 (ZTSQL_DATA, ?)
```

4.10 Debug Procedure

```
create procedure zdebug_int2 ( ) LANGUAGE SQLSCRIPT AS

begin

view = select * from "SAPBWN"."ZTSQL_DATA";

create local temporary table #tmp like "SAPBWN"."ZTSQL_DATA";

call SAPBWN.ZDIST_INT2 (:view, #tmp);

drop table #tmp;

end;
```

www.sdn.sap.com/irj/sdn/howtoguides

THE BEST-RUN BUSINESSES RUN SAP™