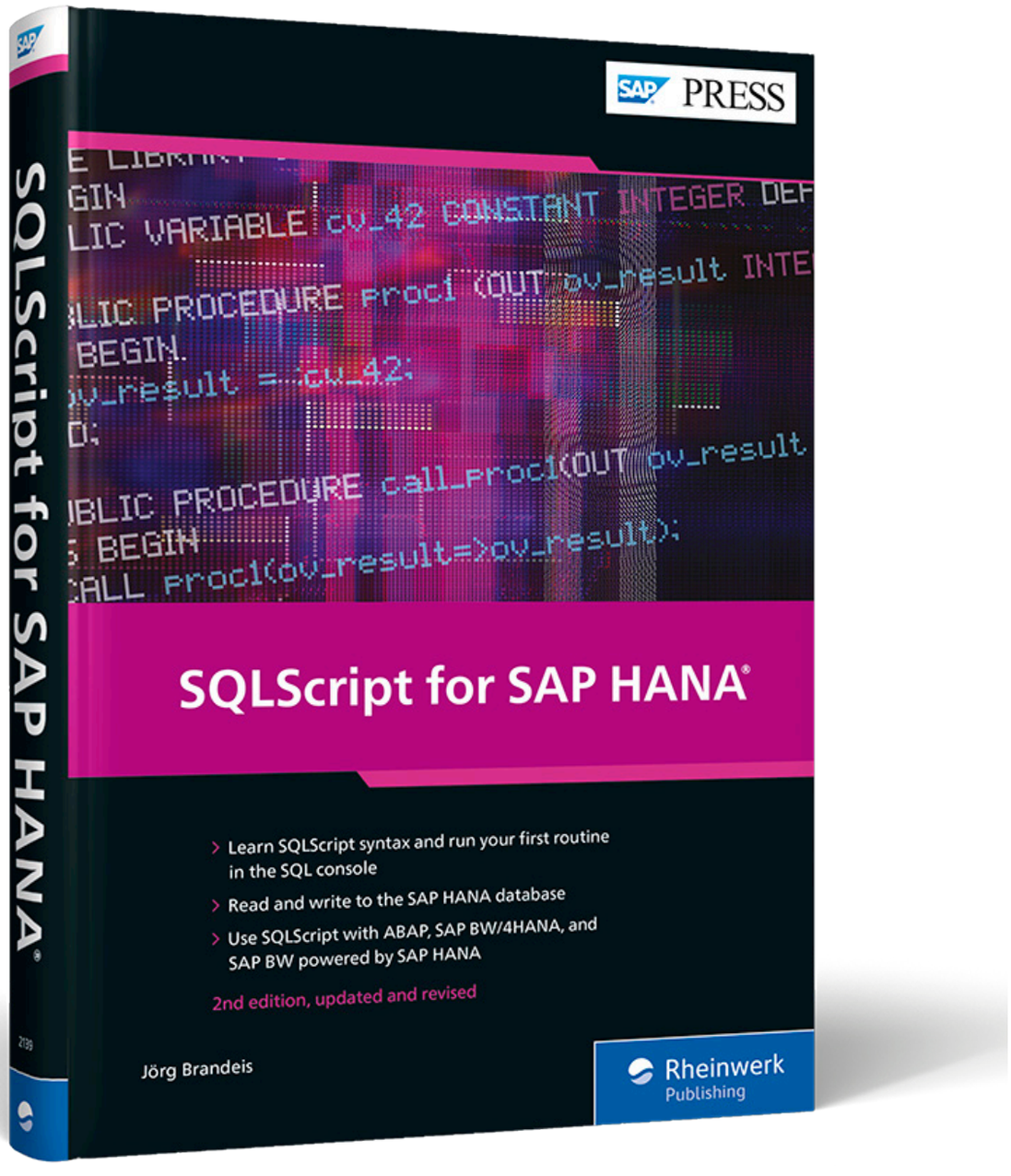


Browse the Book

This sample chapter demonstrates the techniques for using SQLScript procedures for ABAP programs. It starts with the most important concept, AMDP, followed by the other development objects of the AMDP framework. The end of this chapter briefly shows you an alternative to AMPD and give some basic recommendations for using database procedures in ABAP programs.

Jörg Brandeis

SQLScript for SAP HANA

387 pages, 2nd, updated and revised edition 2021, $79.95
ISBN978-1-4932-2139-4

www.sap-press.com/5336

Chapter 8
SQLScript in ABAP Programs

To enable the use of SQLScript from within ABAP programs, various options are available, for example, the ABAP-managed database procedures (AMDP) framework or table functions. In this chapter, we'll provide you an overview of these techniques.

In previous chapters, you learned how to write effective SQLScript source code. You can use various options to call this source code from within the SAP NetWeaver ABAP platform. From the ABAP perspective, SQLScript basically is *Native SQL*, which is how all database-specific SQL dialects are described. The opposite of Native SQL is *Open SQL*, which is part of the ABAP programming language. Open SQL is database-independent and is translated into the corresponding Native SQL at runtime.

Consider Other Database Systems

If you use *Native SQL* in ABAP, the corresponding programs and classes are restricted for use in a particular database system. To be compatible for multiple database systems, you must develop along several tracks. Accordingly, you should create different implementations for the various SQL variants. In this chapter, we'll focus on following a two-track approach in our examples, using both techniques:

- SQLScript for SAP HANA databases
- Open SQL for all other databases

In the following section, you will learn the techniques for using SQLScript procedures for ABAP programs. We'll start with the most important concept, *AMDP*, followed by the other development objects of the AMDP framework. And at the end of this chapter, we will briefly show you an alternative to AMPD and give some basic recommendations for using database procedures in ABAP programs.

8.1 AMDP Procedures

From the point of view of an ABAP developer, an AMDP procedure is a *method* of an ABAP class, which is implemented in the SQLScript

programming language. Being a method solves several problems, such as the following:

- The SQL procedure is called as a method call of an ABAP class. As a result, the procedure call is perfectly integrated in ABAP.
- The *transport* of the database procedure is performed using the ABAP class rather than requiring several transport mechanisms that must be kept in sync.
- At the time of development, the developer does not need direct access to the SAP HANA database with the appropriate authorizations. Working with the usual ABAP developer authorizations with the ABAP development tools is sufficient.

In the following sections, the terms AMDP and AMDP procedure are used interchangeably. If not explicitly mentioned, the information doesn't refer to AMDP functions.

Before we jump into using AMDP, however, we'll first provide you with an initial introduction to some of the concepts of AMDP.

8.1.1 Introduction to AMDP

AMDP

The term *ABAP-managed database procedure (AMDP)* stands for the framework recommended by SAP for using SQLScript code in ABAP programs. This concept serves as the basis for creating, managing, and calling database procedures and functions. The basic idea behind AMDP is that the source code of SAP HANA database objects can be wrapped in an implementation of methods of an ABAP class. The corresponding SAP HANA database object is then generated from this implementation when required.

Three different objects can be created with the AMDP framework:

- *AMDP procedures* can be called like methods of an ABAP class. These procedures are implemented in SQLScript. The caller of the method doesn't know that the procedure is actually a database procedure.
- *Core data services (CDS) table functions* are AMDP functions that are encapsulated by a CDS object and can therefore be called from ABAP or Open SQL like a normal database view via a `SELECT` query.
- *AMDP functions for AMDP methods* can't be called directly from ABAP or Open SQL. However, these functions can be used when implementing other AMDP objects in SQLScript source code.

Table 8.1 compares the most important properties of the three objects.

	AMDP Procedure	CDS Table Function	AMDP Function
Call from ABAP	Like an ABAP method	In a `SELECT` statement	Not possible
Implementation	In a static or instance method	In `PUBLIC`, static method of a static class	In a static or instance method
Two-track development	Inheritable	Possible by case distinction	Not relevant, since only callable from other AMDP methods
Type of data access	Read and write	Read-only	Read-only
Where are the parameters defined?	When you define the method	When you define the CDS object	When you define the method
Type of parameters	Any `IMPORTING`, `EXPORT`, and `CHANGE`	Any scalar `IMPORTING` and exactly one table-like `RETURNING` parameter	Any scalar `IMPORTING` and exactly one table-like `RETURNING` parameter

Table 8.1 Comparison of the Three AMDP Objects

AMDP Framework

With the release of SAP NetWeaver 7.4 Support Package Stack (SPS) 05, AMDPs were introduced as a framework for managing and calling database procedures in the SAP HANA database.

With the release of SAP NetWeaver 7.5, AMDP functions have been added to this concept. Although these functions are technically not procedures, but functions, the name AMDP is also used for them.

AMDP classes are classes that implement the interface `IF_AMDP_MARKER_HDB` and that contain at least one AMDP method.

AMDP methods can be either AMDP procedures or AMDP functions. AMDP functions either implement a CDS table function, or AMDP functions are available internally for other AMDP methods.

Tools

The AMDP framework provides ABAP developers with extensive options for implementation according to the *code-to-data paradigm*. Direct access to the SAP HANA database is not absolutely necessary since all develop-

ment objects can be implemented via the *ABAP development tools* in the Eclipse development environment. You can also use a debugger for the SQLScript code in the AMDP methods. This AMDB debugger is described in more detail in Chapter 11, Section 11.2.3.

If SAP HANA data models such as *calculation views* are modeled or if procedures and functions are developed outside the AMDP framework, direct database access with corresponding authorizations will be necessary. The Eclipse plug-ins from SAP HANA Studio must also be installed.

Benefits of AMDP

When does using SQLScript in an ABAP program make sense? No general answer exists for this question. Basically, the use of AMDPs makes sense above all when significant performance improvements can be achieved, which is particularly the case in the following scenarios:

- If using SQLScript prevents the transport of large amounts of data between the database and SAP NetWeaver Application Server (AS).
- If the SQLScript procedure is programmed declaratively and can therefore be easily parallelized and optimized by the SAP HANA database.
- If transformations with routines are to be executed directly in the SAP HANA database in SAP Business Warehouse (SAP BW). Data transfer processes will be accelerated considerably, regardless of whether the previous ABAP code worked with database access or not.

Disadvantages

But the use of AMDPs also has a few disadvantages. You must always consider whether you want to accept these disadvantages for the expected performance improvements. The following reasons, among others, speak against the use of AMDPs:

- The application developer must be proficient in several programming languages. Sometimes, a well-implemented ABAP method is better than a poorly implemented AMDP.
- SAP GUI is not suitable for developing AMDPs.
- The application logic is distributed between the database server and SAP NetWeaver Application Server (AS).
- Performance is not improved in all cases.
- AMDP developments can't be used in other database systems. For a suitable procedure for parallel development, still leveraging the advantages of SAP HANA with AMDPs, Section 8.1.5.

Alternatives

You should always evaluate the alternatives before using AMDPs, including, for example, CDS views and the extended features of Open SQL of the latest SAP NetWeaver releases. According to the SAP documentation (*http://s-prs.co/v533626*), AMDPs should only be used if database-specific functions, such as the SQL function for currency conversion, is required but not available in Open SQL, or if the transport of large amounts of data between the database server and SAP NetWeaver Application Server (AS) must be avoided.

SPACE and the Empty Character String

In ABAP, a global constant `SPACE`, of length 1, contains exactly one blank character. This constant is often used for selecting data, for example, in the following `WHERE` clause:

```
WHERE recordmode = SPACE
```

In fact, however, the database doesn't store the space character, but an empty string of length 0. ABAP therefore does not distinguish between an empty string and a single space character.

However, SQLScript does distinguish between these two strings. For use in an ABAP system, SAP has introduced the `ABAPVARCHAR` mode. This mode converts all string literals with exactly one blank character into an empty string in SQLScript as well, as in the following query:

```
SELECT session_context( 'ABAPVARCHARMODE' ) FROM DUMMY
```

Furthermore, you can test whether this mode is switched on. Since this mode is controlled by the session variable `ABAPVARCHAR`, you can also turn this mod on and off in the code, as shown in Listing 8.1.

```
DO BEGIN
  SET 'ABAPVARCHARMODE' = 'true';
  SELECT length(' '),
         '->' || ' ' || '<-' FROM dummy;

  SET 'ABAPVARCHARMODE' = 'false';
  SELECT length(' '),
         '->' || ' ' || '<-' FROM dummy;
END;
```

Listing 8.1 Testing the ABAP VARCHAR mode

The result depends on the mode set. If you really need exactly one space as a string in SQLScript, use the expression `CHAR(32)`.

8.1.2 Creating AMDP Procedures

IF_AMDP_MARKER_HDB

You can create an AMDP method only in a *global class*, which implements the marker interface `IF_AMDP_MARKER_HDB`. The interface itself has no methods; it serves exclusively to mark AMDP classes.

An AMDP class can include both normal ABAP methods and AMDP methods. Whether a method is implemented as AMDP is not specified in the method declaration. Listing 8.2 shows an example.

```
CLASS zcl_amdp_demo DEFINITION
  PUBLIC
  CREATE PUBLIC .

  PUBLIC SECTION.
    INTERFACES if_amdp_marker_hdb.
    TYPES gty_tt_countries TYPE TABLE OF t005t .
    METHODS get_countries
      IMPORTING
        VALUE(iv_langu)   TYPE  langu
      EXPORTING
        VALUE(et_country) TYPE  gty_tt_countries
      CHANGING
        VALUE(cv_subrc)   TYPE sy-subrc.
ENDCLASS.

CLASS zcl_amdp_demo IMPLEMENTATION.
  METHOD get_countries
     BY DATABASE PROCEDURE FOR HDB LANGUAGE SQLSCRIPT
     USING t005t.

    et_country = select *
                 from t005t
                 where spras = :iv_langu;

    SELECT CASE
              WHEN COUNT(*) > 0
              THEN 0
              ELSE 4
              END AS subrc
           INTO cv_subrc
           FROM :et_country;
  ENDMETHOD.
ENDCLASS.
```

Listing 8.2 Example of a Simple AMDP Method

Restrictions on the method signature

An SQLScript procedure is generated at a later time from the source code of an AMDP method. However, these database procedures are more restrictive

with regard to parameters than ABAP methods. Accordingly, the following restrictions must of course also apply to AMDP methods:

- First, all parameters must be completely typed. Generic data types such as `TYPE TABLE` or `REF TO DATA` are not permitted.
- Since SQLScript doesn't know any structures, only scalar data types and table types may be used as parameters. The table types can only use scalar data types in their row structure.
- All parameters must be transferred as *value parameters* (call by value); you can't use any *reference parameters* (call by reference). The reason for this restriction is obvious when you consider that the application server is running on a different system than the database. Accordingly, no shared memory area exists that both systems reference.
- You can only use `IMPORTING`, `EXPORTING`, and `CHANGING` parameters with AMDP procedures. The use of `RETURNING` parameters is not possible.
- Only AMDP exception classes can be declared in the signature of the method. These exception classes are the subclasses of `CX_AMDP_ERROR`. If these exception classes are declared, the caller of the AMDP method can handle these exceptions. However, if not declared, these errors result in a *dump*.

Implementing an AMDP

AMDP procedures are developed in the SQLScript language, which is indicated to the system by the addition `BY DATABASE PROCEDURE FOR HDB LANGUAGE SQLSCRIPT`.

If an AMDP implementation only reads data, you can optionally specify the addition `OPTIONS READ-ONLY`.

USING

The optional addition `USING` enables you to tell an AMDP implementation that you want to use the corresponding tables, views, AMDP functions, or AMDP procedures from the default database schema of the SAP NetWeaver system. Thus, you can use this name without having to explicitly specify the relevant database schema in the SQLScript source code. The objects are only separated by whitespaces and listed after the keyword `USING`. Note that this code is still ABAP code. If, for example, you specify a table generated by SAP BW from the generation namespace, `/BIC/` or `/BIO/`, you do not need to use any quotation marks, for example:

```
...USING /BIO/PPLANT.
```

If you access this table in SQLScript code, however, you must use special notation, since the slash is not permitted in simple notation, as in the following:

```
SELECT ... FROM "/BIO/PPLANT".
```

USING for AMDP objects

The use of AMDP procedures and functions is declared in the `USING` clause via the name of the associated ABAP method using the notation `CLASS=>METHOD`. Listing 8.3 contains an example of the keywords in the `METHOD` clause of the method implementation. Everything after the period in line 5 up to the keyword `ENDMETHOD` is interpreted as SQLScript.

```
1 METHOD get_countries
2    BY DATABASE PROCEDURE FOR HDB
3    LANGUAGE SQLSCRIPT
4    OPTIONS READ-ONLY
5    USING t005t.
6
7   <SQLScript-Code>
8 ENDMETHOD.
```

Listing 8.3 Keywords for Implementing an AMDP Method

A General Concept for a Specific Database

Basically, all instructions and declarations in connection with AMDP are designed in such a way that AMDPs can be implemented in different database systems and in different languages. Accordingly, the implementation of AMDP methods also includes the mandatory entries `FOR <databasesystem>` and `LANGUAGE <querylanguage>`.

However, to date (SAP NetWeaver 7.51), only implementations for the SAP HANA database with the SQLScript language are possible. The language L may only be used within SAP.

The other database systems under SAP NetWeaver each have their own programming languages, such as Oracle's PL/SQL language. However, these languages can't be used to implement AMDPs.

8.1.3 Generated Objects of an AMDP Method

An ABAP class serves as an envelope for the code of the AMDP procedure. Some SAP HANA database objects are automatically generated from this class. Figure 8.1 shows the objects generated by the procedure found in Listing 8.2.

Note that a table type, two procedures, a database table, and a view have been generated for an AMDP method. Each object has its own special tasks, which we'll explain next.

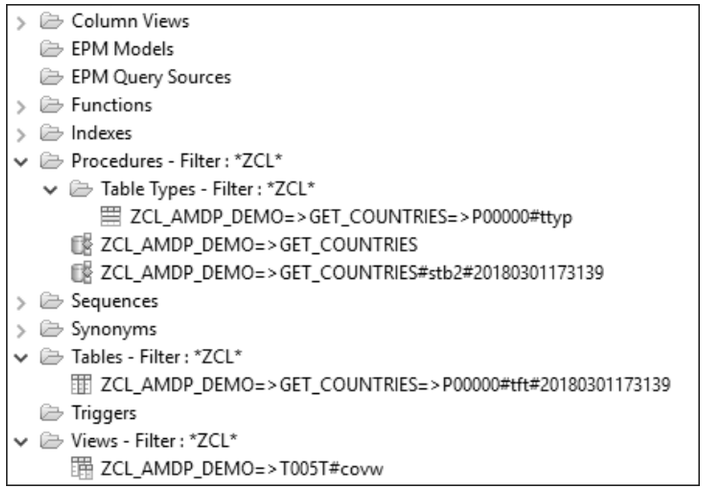


Figure 8.1 Generated Objects of an AMDP Procedure

The Generated AMDP Procedure

The procedure `ZCL_AMDP_DEMO=>GET_COUNTRIES` contains the actual SQLScript source code. However, the code has been slightly modified, as shown in Listing 8.4.

```
CREATE PROCEDURE
  "ZCL_AMDP_DEMO=>GET_COUNTRIES"
(
  IN   "IV_LANGU" NVARCHAR (000001),
  OUT  "EV_SUBRC" INTEGER,
  IN   "CT_COUNTRY__IN__" "ZCL_AMDP_DEMO=>GET_COUNTRIES=>P00000#
ttyp",
  OUT  "CT_COUNTRY" "ZCL_AMDP_DEMO=>GET_COUNTRIES=>P00000#ttyp"
)
LANGUAGE sqlscript  SQL SECURITY INVOKER
AS BEGIN
"CT_COUNTRY" = select * from  :CT_COUNTRY__IN__ ;
BEGIN
    lt_countries = SELECT land1 FROM :CT_COUNTRY;
    et_country = SELECT t5.*
                 FROM "ZCL_AMDP_DEMO=>T005T#covw" AS t5
                 INNER JOIN :LT_COUNTRIES AS countries
                 ON t5.land1 = countries.land1
                 WHERE t5.spras = :IV_LANGU;

    SELECT CASE
              WHEN COUNT(*) > 0
```

```
                THEN 0
                ELSE 4
                END AS subrc
            INTO ev_subrc
            FROM :CT_COUNTRY;
END;

END;
```

Listing 8.4 Source Code of Procedure ZCL_AMDP_DEMO=>GET_COUNTRIES

Parameter list

The relevant passages in the source code are highlighted in bold type. First, you'll see the parameter list. Notice that the CHANGING parameter CT_COUNTRY occurs twice. This repetition occurs because, in SQLScript, an INOUT parameter must have a scalar type. Thus, in addition to the OUT parameter CT_COUNTRY, the IN parameter CT_COUNTRY__IN__ is created. In the first line after AS BEGIN, the parameter CT_COUNTRY is filled with the contents of CT_COUNTRY__IN__, which simulates the behavior of a CHANGING parameter.

Table Types for the Table Parameters

In the parameter interface of an SQLScript procedure, all parameters must be typed. In the case of table parameters, corresponding table types are created for AMDP procedures. In our example, CT_COUNTRY is typed using the generated table type ZCL_AMDP_DEMO=>GET_COUNTRIES=>P00000#ttyp.

USING Views

Each access to a table declared via USING is encapsulated by a view. In our example, this view is the database view ZCL_AMDP_DEMO=>T005T#covw, which replaces direct access to table T005T. The task of the view is to compensate for any field sequences that differ between the ABAP Dictionary and the database object.

Stub Procedure

In addition to the actual AMDP procedure, a second procedure is generated, which is named according to the name pattern <AMDPprocedure>#stb2#<timestamp>. Accordingly, in our example, the procedure name is ZCL_AMDP_DEMO=>GET_COUNTRIES#stb2#20180301173139.

This procedure serves as a *stub* for the call from SAP NetWeaver. The timestamp (Section 8.1.4) is used for versioning purposes in case the object changes.

The source code shown in Listing 8.5 involves only two parameters: IV_LANGU and EV_SUBRC. The CHANGING table parameter CT_COUNTRIES is not part of the parameter definition of the procedure.

Return of the table

The return of the table CT_COUNTRIES is not carried out via an explicitly defined table parameter but via the *result set* of the procedure. After the call of the AMDP procedure, a SELECT query is executed to the table CT_COUNTRIES.

```
CREATE PROCEDURE
  "ZCL_AMDP_DEMO=>GET_COUNTRIES#stb2#20180301173139"
(
  IN    "IV_LANGU" NVARCHAR (000001),
  OUT   "EV_SUBRC" INTEGER
)
LANGUAGE sqlscript SQL SECURITY INVOKER AS BEGIN

  "CT_COUNTRY__IN__" = SELECT * FROM "ZCL_AMDP_DEMO=>GET_COUNTRIES=
>P00000#tft#20180301173139" ;
  CALL "ZCL_AMDP_DEMO=>GET_COUNTRIES" (
    "CT_COUNTRY__IN__" => :CT_COUNTRY__IN__ ,
    "IV_LANGU" => :IV_LANGU ,
    "EV_SUBRC" => :EV_SUBRC ,
    "CT_COUNTRY" => :CT_COUNTRY
  );
  SELECT * FROM :CT_COUNTRY;

END;
```

Listing 8.5 Source Code of the Stub Procedure

Input Tables as Global Temporary Tables

The values of *table parameters* are transferred into the stub procedure via generated, global temporary tables. Before the actual AMDP procedure is called, the table variables are initialized from the global temporary table. In our example, CT_COUNTRIES is filled from the global temporary table in the following way:

```
"CT_COUNTRY__IN__" = SELECT * FROM "ZCL_AMDP_DEMO=>GET_COUNTRIES=>
P00000#tft#20180301173139"
```

Here again, a timestamp is part of the name.

8.1.4 Lifecycle of the Generated Objects

The lifecycle of the AMDP classes and methods is not identical to the lifecycle of their corresponding SAP HANA objects. In the following sections, we'll describe the effects of various actions carried out in the SAP NetWeaver Application Server (AS) on the corresponding SAP HANA objects:

Object lifecycles

- **Creating and activating an AMDP class**
  Creating and activating an AMDP class with an AMDP method doesn't generate any database objects. Thus, you can transport AMDP classes to SAP NetWeaver systems without an SAP HANA database. If an SAP HANA database is available as the central database system, the database objects are temporarily created and immediately deleted again for the syntax check.
- **Executing an AMDP method**
  Database objects, introduced in the previous section, are generated during the first execution of the AMDP method. The *timestamp* in the object's name refers to the activation time of the class.

  Even if a class contains several AMDP methods, only the objects for the executed methods are generated.
- **Repeated editing and activation of an AMDP class**
  The stub procedures for all AMDP procedures are deleted when the class is *reactivated*. The same applies to all generated AMDP procedures that have changed. The generated global temporary table will remain for the time being.

  The next time the program is executed, the objects are created again with the new timestamp. Since global temporary tables are not deleted immediately, some objects with different timestamps can still exist at the same time.
- **Deleting an AMDP class**
  The generated objects on SAP HANA are deleted upon the deletion of the AMDP class.

The late generation of SAP HANA objects means that transporting AMDP classes in a mixed system landscape can run smoothly. Versioning of changes is carried out using timestamps to ensure that the correct version of a procedure is always called.

8.1.5 Two-Track Development

There are different reasons why you would want both an ABAP and an AMDP implementation of a method. From a technical point of view, developing

both is not a problem, since whether a method is written in SQLScript or ABAP is only determined when the method is implemented and doesn't affect the definition of the method. In the following sections, we'll look at a few typical use cases where more than one implementation exists.

Clean Code: Separation of Concerns

The principle of the separation of concerns was formulated in 1974 by Edsger W. Dijkstra. Under this principle, different tasks should also be performed by different components in a system. In our case, separation of concerns means that database access should not be mixed with other application code. Instead, different classes should be created for each aspect.

If you adhere to this principle, two-track development of AMDP and ABAP implementations is also relatively simple. Classes for database access can be easily exchanged.

Support of Different Database Systems

If the application needs to run on different database systems, you can have one implementation in Open SQL and one in AMDP. Of course, you can also create special implementations for each of the other databases.

Switching between implementations

Switching between the various implementations should take place automatically at runtime. For this purpose, you can use the *factory design pattern*. As shown in Figure 8.2, we'll use a static factory method, `GET_INSTANCE`, to generate the instances. Based on the `SY-DBSYS` field, this method decides which class implementation should be used.

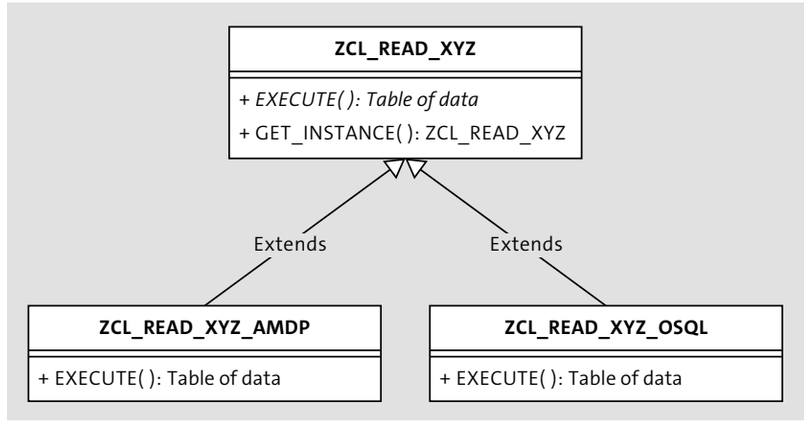


Figure 8.2 UML Diagram for Two Implementations of Database Access with a Static Factor Method

Listing 8.6 shows a simple implementation of the method GET_INSTANCE. Of course, other database systems could also be considered.

```
METHOD get_instance.
  DATA lv_classname TYPE classname.
  CASE sy-dbsys.
    WHEN 'HDB'.
      lv_classname = 'ZCL_READ_XYZ_AMDP'.
    WHEN OTHERS.
      lv_classname = 'ZCL_READ_XYZ_OSQL'.
  ENDCASE.

  CREATE OBJECT ro_instance TYPE (lv_classname).
ENDMETHOD.
```

Listing 8.6 Static Factory Method GET_INSTANCE in ABAP

Comparison between ABAP and AMDP

If you want to test the differences between the implementations in ABAP and AMDP in runtime and result, inheritance is rather useful concept as in, for example, the previous section where we leveraged inheritance from a shared superclass. However, you'll need a suitable switch that allows you the flexibility to select the relevant implementation. You can create these switches, for example, by using entries in a Customizing table or by defining *user parameters*. These switches can then be queried in a corresponding factory method.

Retroactive Implementation as AMDP

Let's say you implemented a method in ABAP and later discover performance problems; you can implement the method as an AMDP. To preserve the original implementation for performance and result comparisons, creating a subclass is a good idea. The corresponding method is then implemented as an AMDP in this subclass.

Refactoring

Due to restrictions on AMDPs regarding access to instance attributes and parameters, this procedure must be taken into account when creating your methods. Alternatively, you can carry out a refactoring, as shown in Figure 8.3, by following this approach:

1. Outsource the logic into a new method that meets the requirements of an AMDP: no access to class and instance data, no RETURNING parameters, no parameters with structures.
2. Create a subclass.
3. Redefine the corresponding method as an AMDP.

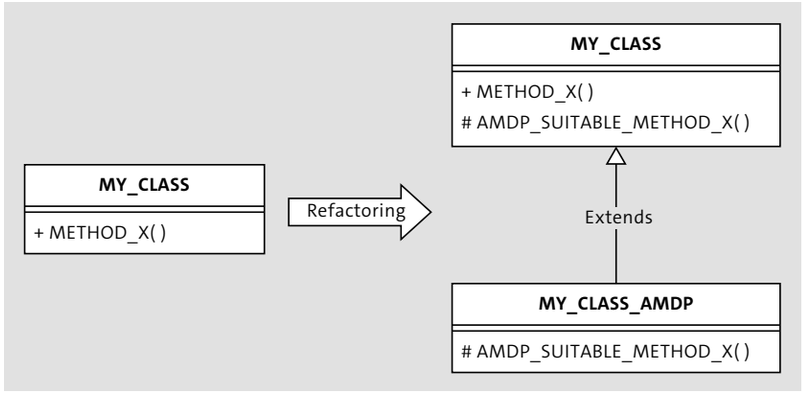


Figure 8.3 Outsourcing the Logic to an AMDP-Compatible Method and Redefinition

To switch between the two versions of the class, you can use a factory method again.

ABAP Unit Tests

A mock object is often used to insert data into a class to be tested in a unit test. This object has the same external interface as a real object but provides predefined data that is independent of the database state. Classes that contain an AMDP can, for example, be replaced for unit tests by a local subclass in which the AMDP method has been replaced by a corresponding ABAP implementation. For such tests, you can use the *dependency injection design pattern*.

Only in global classes

Note that AMDP implementation is only possible in global classes. In local classes, only an implementation in ABAP is allowed.

8.1.6 Using AMDP Procedures in Other AMDP Procedures

An AMDP method generates a corresponding database procedure the first time it is executed (see also Section 8.1.4). This database procedure can of course also be called directly in any SQLScript code, especially in other AMDP procedures. Thus, a certain modularization of the programs is possible without direct development access to the database.

Example

Listing 8.7 shows a simple example of calling an AMDP procedure in another AMDP method. Note that a CHANGING parameter in the AMDP method has the following two associated parameters in the corresponding database table, as described in Section 8.1.3:

- CT_PRICE
- CT_PRICE__IN__

In this case, you must assign these parameters yourself in order to call the procedure correctly.

[+]

Avoid CHANGING Table Parameters

In the database procedure, the CHANGING table parameters are implemented in two parameters. As shown in Listing 8.7, the code needed for a direct call of this procedure can look confusing to untrained readers because the CT_PRICES__IN__ parameter was not defined anywhere in the ABAP class. If you use these two parameters instead, the code will be more readable:

- IT_PRICES for entering the table
- ET_PRICES for the output of the table

If the generated procedures are only called using ABAP methods, whether or not you use CHANGING table parameters doesn't matter.

```
CLASS zcl_amdp_call DEFINITION PUBLIC.

  PUBLIC SECTION.
    TYPES: BEGIN OF ty_s_price,
             item        TYPE numc4,
             net_price   TYPE wertv9,
             gross_price TYPE wertv9,
             vat         TYPE wertv9,
             curr        TYPE waers,
           END OF ty_s_price.

    TYPES ty_t_price TYPE STANDARD TABLE OF ty_s_price.

    INTERFACES if_amdp_marker_hdb.

    METHODS calculate_vat
      IMPORTING
        VALUE(iv_vat)   TYPE int1
      CHANGING
        VALUE(ct_price) TYPE ty_t_price.

    METHODS calculate_gross_price
      IMPORTING
```



```
        VALUE(iv_vat)   TYPE int1
      CHANGING
        VALUE(ct_price) TYPE ty_t_price.

ENDCLASS.

CLASS zcl_amdp_call IMPLEMENTATION.
  METHOD calculate_gross_price BY DATABASE PROCEDURE
                       FOR HDB LANGUAGE SQLSCRIPT
                       USING zcl_amdp_call=>calculate_vat.
    CALL "ZCL_AMDP_CALL=>CALCULATE_VAT"(
                             iv_vat => :iv_vat,
                             ct_price => :ct_price,
                             ct_price__in__ => :ct_price );

    ct_price = SELECT item,
                      net_price,
                      net_price + vat as gross_price,
                      vat,
                      curr,
                      FROM :ct_price;
  ENDMETHOD.

  METHOD calculate_vat BY DATABASE PROCEDURE
                        FOR HDB LANGUAGE SQLSCRIPT.
    ct_price = SELECT item,
                      net_price,
                      gross_price,
                      net_price * :iv_vat / 100 as vat,
                      curr,
                      FROM :ct_price;

  ENDMETHOD.

ENDCLASS.
```

Listing 8.7 Calling an AMDP Procedure from Another AMDP Method

All AMDP procedures used must be declared in an AMDP method via USING, which ensures that the necessary database objects are generated as required. Note that the class constructor of the AMDP classes used is called before the method is executed.

USING

8.2 CDS Table Functions

A *CDS table function* is an AMDP function that provides table data and can be consumed from ABAP in Open SQL, like a database view.

Implementation

The implementation of a CDS table function is similar to an AMDP procedure as a method of an ABAP class. A corresponding user-defined function is then generated from this method in the SAP HANA database. Since user-defined functions can't be consumed directly in Open SQL, the AMDP functions are encapsulated by a CDS object. An AMDP function can therefore also be considered a programmed *CDS view*. The CDS object can then be used normally, such as a database view, in Open SQL.

Core Data Services

The classic *ABAP Dictionary* manages database objects such as data types (data elements, structures, table types); database tables; and views. The definitions of these objects are stored and managed in a *database-neutral* form in the associated database tables (DD*) at the ABAP application server level. Depending on the database system used, the appropriate database objects are then generated from the stored definition. The classic ABAP Dictionary, which can be called via Transaction SE11, only allows the limited use of the SQL options, especially for views.

CDS are a common concept that exists both directly in the SAP HANA database and on the SAP NetWeaver AS ABAP. With CDS, database objects such as views or table functions can be defined with the help of their own language, which is based on the data definition language (DDL) of SQL. These data models can be enriched with semantic information using annotations.

8.2.1 Creating a CDS Table Function

To create a CDS table function, two related objects must be created:

- The *CDS table function* is defined in a *DDL source* with the parameters and the table type of the return table. This step implicitly defines the signature of the associated AMDP method.
- The *AMDP function* is implemented in a method of a static ABAP class. This step contains the SQLScript source code for the table function.

Interaction

Figure 8.4 shows the interaction between a CDS table function and an AMDP function.

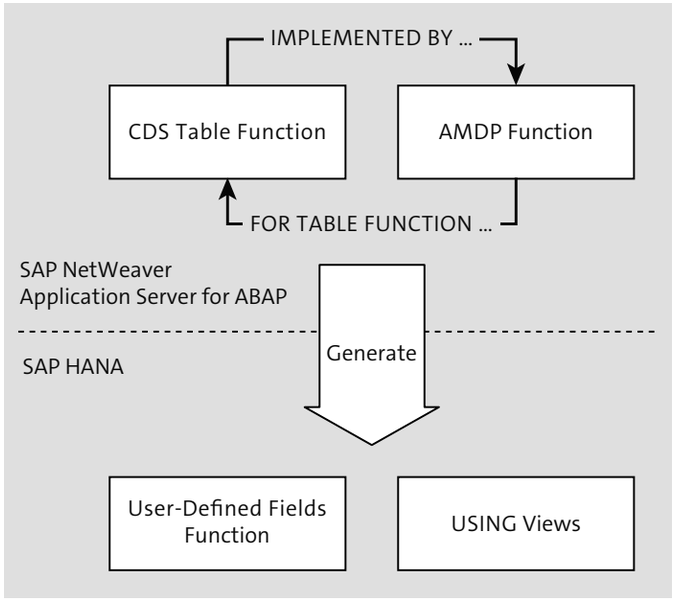


Figure 8.4 Objects Involved in a CDS Table Function

CDS Table Function

CDS table functions wizard

You can use a wizard to create CDS table functions from the ABAP development tools of the Eclipse development environment. Start by clicking **New • Other ABAP Repository Object** in the context menu of the development package and then choose **Core Data Services • Data Definition** in the following dialog. Figure 8.5 shows the corresponding wizard. To create a CDS table function, you must select the **Define Table Function with Parameters** template.

The wizard then generates a basic structure of the corresponding source code of the CDS table function, as shown in Listing 8.8.

```
@EndUserText.label: 'Example for a table function'
define table function Z_CDS_TF
with parameters parameter_name : parameter_type
returns {
  client_element_name : abap.clnt;
  element_name : element_type;
}
implemented by method class_name=>method_name;;
```

Listing 8.8 Basic Structure of the Definition of a CDS Table Function

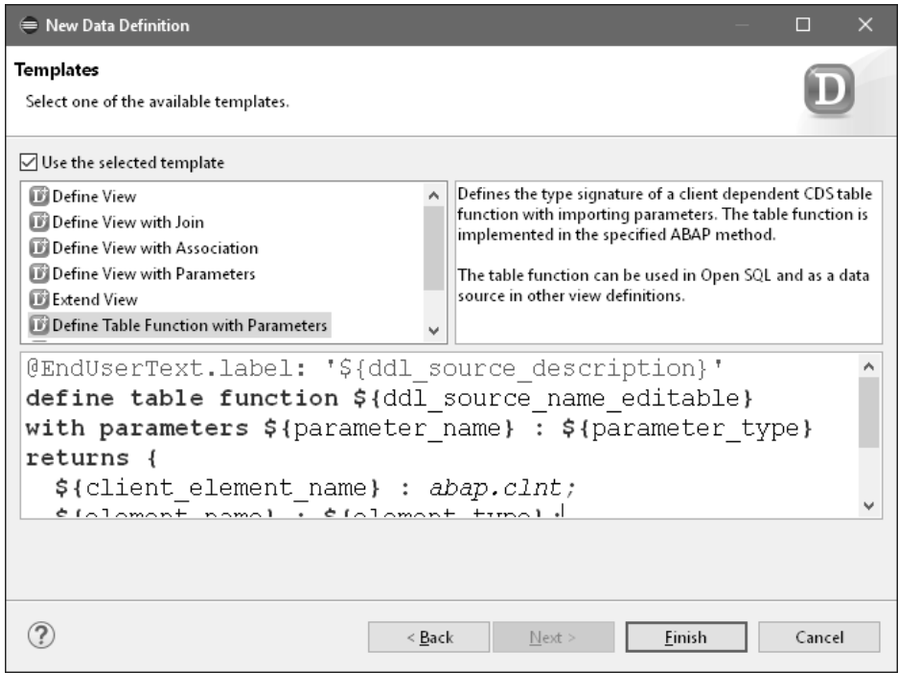


Figure 8.5 Wizard for Creating CDS Objects

In this section, we'll describe only a rudimentary syntax for defining CDS objects, which are not the focus of the book. For more information on CDS objects, we recommend the SAP document "SAP HANA Core Data Services (CDS) Reference," at *http://s-prs.co/v533627*.

Components

The definition of the CDS table function consists of the following components:

- `@EndUserText.label`
  This annotation contains the descriptive text for the CDS table function previously specified in the wizard.
- `define table function <CDS_entity>`
  This component indicates the start of the definition of the CDS table function. The name of the CDS entity is in the same namespace as the data types of the ABAP Dictionary. Therefore, a CDS entity must not have the same name as, for example, a database table.
- `with parameters <parameterlist>`
  If the CDS table function will have parameters, these parameters can be defined in this component. The use of parameters is optional. The client is often specified as a parameter, which will allow you to select data by client in the function, as shown in Listing 8.10.

- `returns { <fieldlist> }`
  The field list defines the structure of the return table. If the CDS table function is client-dependent, the first field must have the ABAP Dictionary type `CLNT`. For example, you can use the data element `MANDT`.
- `implemented by method <classname>=><methodname>`
  This component tells the CDS table function by which AMDP method the implementation is performed. The CDS entity can be activated if the associated method doesn't exist yet.

AMDP Function

The actual implementation of the CDS table function occurs an AMDP method in a *static*, *global* ABAP class, which requires the implementation of the marker interface `IF_AMDP_MARKER_HDB`. Unlike AMDP procedures, AMDP functions for CDS table functions are static methods. The parameters are not defined individually in the method definition. Instead, you'll use the addition `FOR TABLE FUNCTION <CDS_table_function_name>` to establish the reference to the associated CDS table function. The parameters defined in the CDS entity are then available in the function implementation.

The method implementation must be structured as shown in Listing 8.9.

```
METHOD <method_name> BY DATABASE FUNCTION
                     FOR HDB LANGUAGE SQLSCRIPT
                     OPTIONS READ-ONLY
                     [USING <usages>].
  <SQLScript-Code>
ENDMETHOD.
```

Listing 8.9 Structure of the Implementation of an AMDP Function for a CDS Table Function

The additions in the `METHOD` statement are all mandatory except for the `USING` specification. The use of `USING` in this context is similar as in AMDP procedures.

Example of a CDS Table Function

For a functioning CDS table function, the definition of the CDS entity and the implementation of the AMDP function belong together.

Definition

Listing 8.10 and Listing 8.11 contain some simple examples to show the basics of CDS table functions using texts for countries read from table `T005T`. In practice, your CDS table functions will be used for much more complex scenarios that utilize multiple features of the SAP HANA database and SQLScript.

```
@EndUserText.label: 'Country texts'
define table function z_country_text
with parameters
@Environment.systemField: #CLIENT mandt:mandt,
@Environment.systemField: #SYSTEM_LANGUAGE sy_langu:langu
returns {
  mandt:mandt;
  country:land1;
  text:landx50;
}
implemented by method zjb_cl_country=>get_country_text;
```

Listing 8.10 Example of Defining a CDS Table Function

The client and language are taken from the system fields via the annotation `@Environment.systemField`.

RETURN statement

The implementation shown in Listing 8.11 consists of only a `RETURN` statement, which returns the result of a `SELECT` query.

```
CLASS zjb_cl_country DEFINITION PUBLIC.
  PUBLIC SECTION.
    INTERFACES if_amdp_marker_hdb.
    CLASS-METHODS get_country_text
                  FOR TABLE FUNCTION z_country_text.
ENDCLASS.

CLASS zjb_cl_country IMPLEMENTATION.
  METHOD get_country_text BY DATABASE FUNCTION
                          FOR HDB LANGUAGE SQLSCRIPT
                          OPTIONS READ-ONLY
                          USING t005t.
    RETURN SELECT mandt,
                  land1 AS country,
                  landx50 AS text
                  FROM t005t
                  WHERE spras = :sy_langu
                    AND mandt = :mandt;
  ENDMETHOD.
ENDCLASS.
```

Listing 8.11 Example of Implementing a CDS Table Function

Testing

You can test the CDS table function using a simple ABAP program, as shown in Listing 8.12.


```
REPORT zjb_test .

SELECT *
    FROM z_country_text
    INTO TABLE @DATA(lt_country)
    ##db_feature_mode[amdp_table_function].

cl_demo_output=>display(  lt_country ).
```

Listing 8.12 ABAP Program for Testing the CDS Table Function from the Example

Notice that the CDS table function can be read with a `SELECT` query, much like a database table or a view.

Two-Track Implementation

You cannot create alternative implementations in ABAP or Open SQL for CDS table functions. However, if your program needs to run on SAP HANA as well as on other database systems, the caller of the CDS table function must take care of an alternative implementation. To query at runtime whether CDS table functions are available, you can use the `CL_ABAP_DBFEATURES` class. As shown in Listing 8.13, this class can be queried for the existence of the used feature `AMDP_TABLE_FUNCTION`.

```
IF cl_abap_dbfeatures=>use_features(
  VALUE #( ( cl_abap_dbfeatures=>amdp_table_function ) ) ).
* Implementation with CDS table function
ELSE.
* Alternative implementation without SAP HANA features
ENDIF.
```

Listing 8.13 Checking the Availability of CDS Table Functions

The syntax check also indicates with a warning that database-dependent functions are used when querying CDS table functions. You can use the following pragma to disable the warning, as shown in Listing 8.12.

```
##db_feature_mode[amdp_table_function]
```

This pragma signals to the syntax check that the developer is aware that this query will lead to errors in other database systems.

8.2.2 Generated Objects of a CDS Table Function

User-defined function

As with AMDP procedures, database objects can also be generated in SAP HANA for CDS table functions. First, a user-defined function is assigned the

same name as the associated static method of the AMDP function, as shown in Listing 8.14.

```
create function
  "ZJB_CL_COUNTRY=>GET_COUNTRY_TEXT"
(
   "MANDT" NVARCHAR (000003),
   "SY_LANGU" NVARCHAR (000001)
)
returns table
(
  "MANDT" NVARCHAR (000003) ,
  "COUNTRY" NVARCHAR (000003) ,
  "TEXT" NVARCHAR (000050)
)
language sqlscript  sql security invoker  as begin

    RETURN SELECT mandt,
                  land1 AS country,
                  landx50 AS text
                  FROM "ZJB_CL_COUNTRY=>T005T#covw"
                  WHERE spras = :SY_LANGU
                    AND mandt = :MANDT;

end;
```

Listing 8.14 Generated User-Defined Function for the AMDP Function

As with AMDP procedures, a view was also generated for the table specified via `USING`.

8.2.3 Implicit Client Handling of CDS Table Functions

By default, the implicit *client handling* of CDS table functions is enabled. Thus, the first field in the field list must be of dictionary type `CLNT`. Accordingly, the developer must also ensure that this field is filled out correctly with the source client of the data. In a `SELECT` query to the CDS table function, all data with other clients are then implicitly filtered out.

To improve performance, you should filter out the data of other clients in the AMDP function. To create a filter, you'll define an input parameter for the client, which is then automatically assigned the correct client via the annotation `@Environment.systemField: #CLIENT`. The examples shown in Listing 8.10 and Listing 8.11 illustrate the correct client handling.

Note the Client in the Join Condition [!]

As an ABAP developer, you may not be accustomed to taking the client into account since this task is usually undertaken by Open SQL. In SQLScript, however, the client is only an ordinary database field. Be sure you make the correct selection yourself, in particular for join conditions. If you don't select the client in this situation, the cartesian product of the data is formed over all existing clients.

If implicit client handling is not desired, you can disable it using the annotation `@ClientDependent: false`. Nevertheless, you can still have the client as a field in the return structure of the CDS table function and select it as for any other field.

Client independent

Use Implicit Client Handling If Possible [+]

Application and Customizing data are typically client specific. Thus, you should always use the implicit the client handling feature of the CDS table function.

8.3 AMDP Functions for AMDP Methods

If AMDP functions are only to be used in SQLScript source code by other AMDP methods, defining an associated CDS table function is not necessary. The definition and implementation of an AMDP function for AMDP methods differs in the following aspects:

- The AMDP method can be a static method or an instance method. Therefore, the associated AMDP classes don't have to be static.
- The method can also be declared in the private or protected visibility area.
- The parameters of the method are defined when the method is defined.
- The addition `FOR TABLE FUNCTION` in the method definition is omitted.

8.3.1 AMDP Table Functions

AMDP table functions have been available in ABAP since SAP NetWeaver 7.40. They are defined with a fully typed, table-like `RETURNING` parameter. In the implementation, the result table must then be returned with the `RETURN <table expression>` statement.

AMDP functions in different AMDP methods

Listing 8.15 contains an example using an AMDP table function in a different AMDP method. The direct call of the method GET_COUNTRY_TEXT in an ABAP method of the class ZCL_AMDP_FUNC would not be possible.

```
CLASS zcl_amdp_func DEFINITION PUBLIC.

  PUBLIC SECTION.
    TYPES: BEGIN OF ty_s_country,
             mandt   TYPE mandt,
             country TYPE land1,
             text    TYPE landx50,
           END OF ty_s_country.
    TYPES ty_t_country TYPE STANDARD TABLE OF ty_s_country
                       WITH DEFAULT KEY.

    INTERFACES if_amdp_marker_hdb.
    METHODS test_amdp_table_function
      IMPORTING VALUE(iv_langu)   TYPE langu
                VALUE(iv_mandt)   TYPE mandt
      EXPORTING VALUE(et_country) TYPE ty_t_country.

  PRIVATE SECTION.
    METHODS get_country_text
      IMPORTING VALUE(iv_langu)   TYPE langu
                VALUE(iv_mandt)   TYPE mandt
      RETURNING VALUE(rt_country) TYPE ty_t_country.
ENDCLASS.

CLASS zcl_amdp_func IMPLEMENTATION.

  METHOD test_amdp_table_function BY DATABASE PROCEDURE
                      FOR HDB LANGUAGE SQLSCRIPT
                      OPTIONS READ-ONLY
                      USING zcl_amdp_func=>get_country_text.
    et_country = select *
                    from "ZCL_AMDP_FUNC=>GET_COUNTRY_TEXT"
                    ( iv_langu => :iv_langu,
                      iv_mandt => :iv_mandt);
  ENDMETHOD.

  METHOD get_country_text BY DATABASE FUNCTION
                          FOR HDB LANGUAGE SQLSCRIPT
                          OPTIONS READ-ONLY
                          USING t005t.
```



```
    RETURN SELECT mandt,
                  land1 AS country,
                  landx50 AS text
                  FROM t005t
                  WHERE spras = :iv_langu
                    AND mandt = :iv_mandt;
  ENDMETHOD.

ENDCLASS.
```

Listing 8.15 Example of Using an AMDP Table Function in a Different AMDP Method

8.3.2 Scalar AMDP Functions

With SAP_BASIS component release 753, scalar functions can also be implemented in the AMDP framework. In the definition, these functions differ only in the scalar return type of the function. In the implementation, the result is then not returned with the RETURN statement, but the return parameter is simply assigned.

We explored the structure of an AMDP function in connection with CDS table functions earlier in Listing 8.9. For scalar AMDP functions, you can also specify the DETERMINISTIC option after OPTIONS if the same output is always generated for the same input. This step allows the results of the scalar function to be buffered from the database.

Listing 8.16 shows an example of a scalar function that cleans up values for InfoObjects that are not allowed in an SAP BW system. Note that, although this function finds and replaces most characters disallowed in SAP BW systems, other characters may cause problems.

```
CLASS zjb_bw_tools_amdp DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.
    INTERFACES if_amdp_marker_hdb.
    METHODS replace_unallowed_characters
      IMPORTING VALUE(iv_input)  TYPE rschavl60
      RETURNING VALUE(rv_output) TYPE rschavl60.
ENDCLASS.

CLASS zjb_bw_tools_amdp IMPLEMENTATION.
```

```
  METHOD replace_unallowed_characters BY DATABASE FUNCTION
                                      FOR HDB LANGUAGE SQLSCRIPT
                                      OPTIONS READ-ONLY.

    rv_output = CASE WHEN LEFT( :iv_input, 1 ) = '!'
                       THEN replace( iv_input ,'!' , '$')
                     WHEN :iv_input = '#'
                       THEN ''
                     ELSE :iv_input
                END;

    rv_output = replace_regexpr( '[[:cntrl:]]'
                                      IN :rv_output WITH '' );
    rv_output = replace( :rv_output, nchar( '0130'), '');
  ENDMETHOD.
ENDCLASS.
```

Listing 8.16 Example of a Scalar AMDP Function

Performance of Scalar Functions

In general, scalar functions run the risk of having a negative impact on performance. With the function shown in Listing 8.16, you can easily clean up dozens of fields in a transformation routine in the SAP BW system. However, large amounts of data can quickly lead to performance problems. Before using any scalar function, therefore, you should test them with realistic data volumes to determine whether the scalar function is sufficiently fast.

8.4 Alternatives to AMDP for Calling SQLScript Code from ABAP Programs

With the AMDP framework, calling SQLScript can be elegantly integrated into the ABAP language and Open SQL. A person reading an ABAP program may not initially recognize that AMDP objects are actually hidden behind a method call or a presumed table.

Alternative techniques

However, with alternative techniques, you can execute SQLScript statements from ABAP programs. These techniques may be necessary if secondary database connections will be used and can be helpful when executing dynamic SQL code.

However, you can call SQLScript code from ABAP without using AMDPs in several alternative ways:

- **ABAP Database Connectivity (ADBC)**
  ADBC is a class-based framework for running Native SQL. Individual SQL statements, database connections, and result sets are represented as ABAP objects. ADBC can be regarded as a successor to the static embedding of Native SQL using the `EXEC SQL` statement in ABAP code. In more than name only, the ADBC framework is reminiscent of the similarly structured Java Database Connectivity (JDBC) from the Java world.
- **`EXEC SQL <Native SQL> ENDEXEC`**
  This statement enables you to embed Native SQL statically into the ABAP source code. However, SAP no longer recommends this approach. Instead, ADBC is referenced for the use of Native SQL.
- **`CALL DATABASE PROCEDURE`**
  This ABAP statement allows you to call any database procedure on SAP HANA via an associated proxy object. However, SAP recommends using AMDPs instead, as long as the procedure will be called via the primary database connection.

8.5 Recommendations

In this chapter, you learned some ways to directly access the objects of an SAP HANA database in your programs. Even if trying out these new techniques is tempting, let's pause a moment to review some final considerations:

- If possible, you should continue to use Open SQL, as long as no significant performance improvements are foreseeable. The ABAP language has been extended in recent releases to include powerful functions, such as the use of expressions and the option of combining multiple queries via `UNION`. More complex queries can also be mapped using CDS views.
- However, if, for performance reasons or other considerations, executing SQLScript code directly from ABAP programs still seems necessary, the techniques of the AMDP framework on the SAP HANA database are the weapon of choice for executing static SQLScript code.
- If you need dynamic SQLScript code, ADBC is the right technique. However, dynamic SQL code is always problematic. On one hand, error analysis is often more difficult, and on the other hand, you must guard against security gaps and vulnerability to SQL injection attacks.
- If you're working with secondary database connections, you should consider using a `CALL DATABASE PROCEDURE` statement. Further, the autonomous transactions we discussed in Chapter 6, Section 6.6.2, can also solve some classic use cases for secondary database connections more elegantly.

Regardless of the technology used, the clean encapsulation of the database access is always recommended in both AMDP and Open SQL. Only if you bundle and encapsulate the database access in separate classes will you have the necessary flexibility to carry out optimizations later without fear of negative side effects.

8.6 Summary

Code execution on the SAP HANA database system can be extremely fast, which you can effectively take advantage of with the concepts learned in this chapter. With the AMDP framework, you have in your hands a tool that allows you to easily create and call database procedures and functions. In the next chapter, you'll learn how this framework is used by SAP BW to implement transformation routines in SQLScript.

Contents

Appendices 391

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

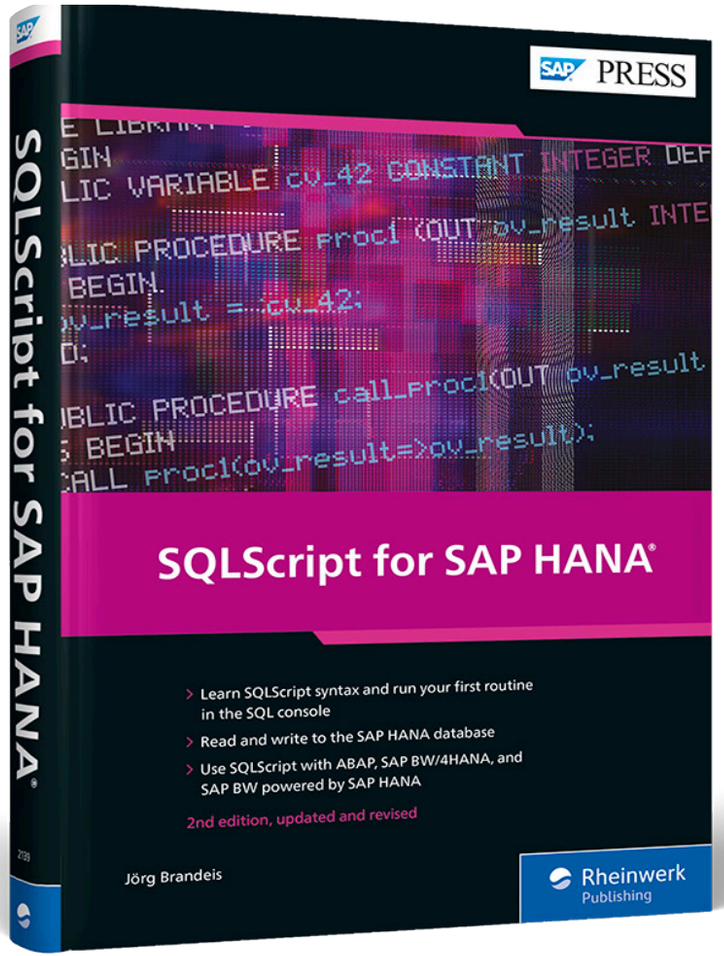
T

U

V

W

SAP PRESS
First-hand knowledge.
Rheinwerk Publishing



Jörg Brandeis is the managing director of Brandeis Consulting GmbH in Mannheim, which offers training and consulting for SAP BW/4HANA, SAP HANA and SQLScript. Until mid-2015, Mr. Brandeis worked as head of development at zetVisions AG in Heidelberg, where he was responsible for the development and architecture of the SAP NetWeaver-based products zetVisions CIM and SPoT. In this role, he worked extensively with agile development methods and clean code.

Jörg Brandeis

SQLScript for SAP HANA

387 pages, 2nd, updated and revised edition 2021, $69.95
ISBN 978-1-4932-2139-4

www.sap-press.com/5336