





## Browse the Book

ABAP-managed database procedures can be leveraged to optimize nonnative, ABAP-based applications such as programs, forms, and interfaces by leveraging the code pushdown features supported by SAP HANA database. In this chapter you will learn to define and implement AMDP methods.

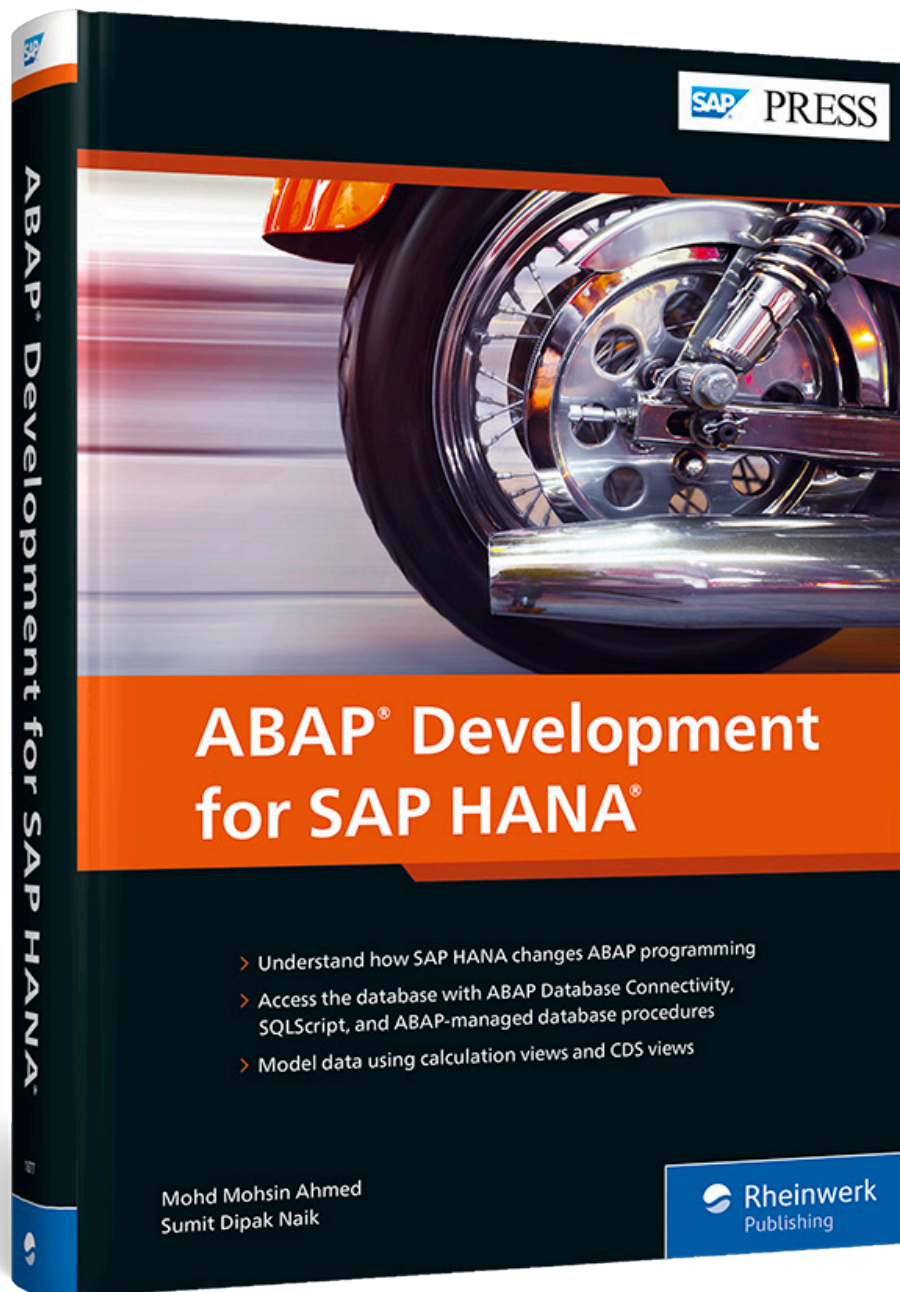
-  **“ABAP-Managed Database Procedures”**
-  **Contents**
-  **Index**
-  **The Authors**

Mohd Mohsin Ahmed, Sumit Dipak Naik

### ABAP Development for SAP HANA

643 Pages, 2021, \$89.95  
ISBN 978-1-4932-1877-6

 [www.sap-press.com/4954](http://www.sap-press.com/4954)



## Chapter 6

# ABAP-Managed Database Procedures

*ABAP-managed database procedures can be leveraged to optimize non-native, ABAP-based applications such as programs, forms, and interfaces by leveraging the code pushdown features supported by SAP HANA database. In this chapter you will learn to define and implement AMDP methods.*

In Chapter 2, we briefly introduced you to the code pushdown paradigm, which helps developers optimize business applications on the SAP HANA database. This chapter will include an emphasis on ABAP-managed database procedures (AMDPs), one of the recommended approaches to achieve code pushdown functionalities.

In this chapter, we'll start by covering essential aspects like the motivation behind introducing AMDPs, including how they are created and consumed in business applications in Section 6.1. We'll then show you how to create AMDP classes in Section 6.2 and also explore the concepts behind enhancements using AMDPs in Section 6.3. Next, we'll turn to exception handling in Section 6.4 and explore different debugging techniques to analyze your procedures in Section 6.5. We'll conclude this chapter on AMDPs tools in Section 6.6.

### 6.1 Introduction

Let's briefly recap some information you learned earlier in this book about code pushdown. In the classic style of coding, as shown in Figure 6.1, developers used to design applications by retrieving all the data at once using database array operations such as `FOR ALL ENTRIES`, `JOIN`, or ABAP Dictionary views. This approach reduced loads on database servers by limiting data transfer requests between the application and the database server.

Subsequent data processing operations are performed on the application server's internal table to achieve the desired results. However, this approach's drawback was the formation of complex SQL queries to retrieve data. Performance issues arose because unnecessary data was retrieved, filtered, and processed in the application layer.

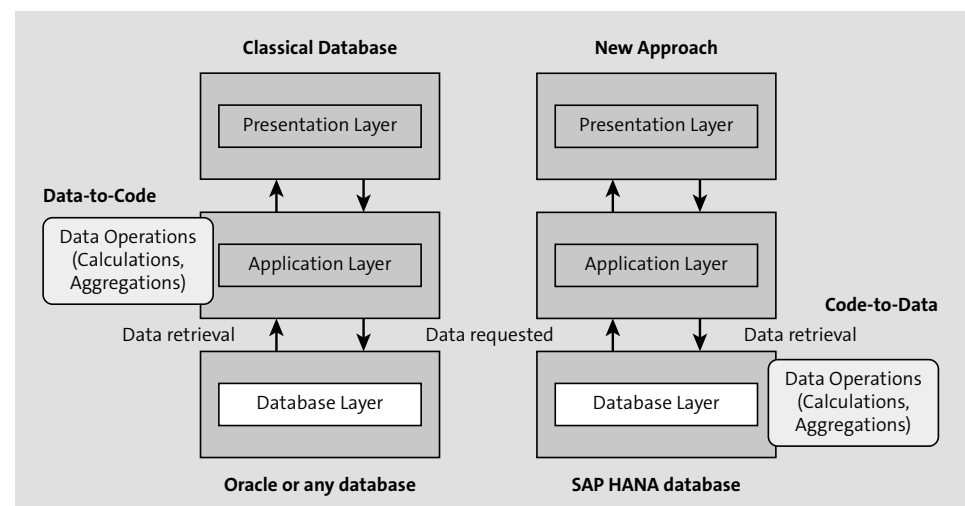


Figure 6.1 New Programming Approach

This fundamental change to the ABAP programming model in favor of pushing code and data processing to the database is called the *code-to-data paradigm*, shown in Figure 6.1, instead of the classic *data-to-code approach*.

In SAP HANA, several code-to-data techniques are available for implementing data-intensive calculations in the database layer. SAP HANA performs data-intensive calculations in the database layer using SAP HANA views or procedures. These artifacts are later utilized in ABAP applications using several code-to-data techniques. Whether or not you should use these techniques depends on the SAP NetWeaver Application Server for ABAP (AS ABAP) release used in your landscape.

As shown in Figure 6.2, if you are using SAP NetWeaver AS ABAP 7.4 SPO2 or lower, SAP HANA repository objects, such as SAP HANA views or procedures, are directly accessed in ABAP applications using *native SQL*. With SAP NetWeaver AS ABAP 7.4 SPO2 or higher, these objects are accessed in ABAP applications using ABAP proxy objects such as *external views* or *database procedure proxies* to overcome the limitations of using native SQL.

The techniques used before SAP NetWeaver AS ABAP 7.4 SPO5 are known as bottom-up approaches because SAP HANA views or procedures are created in the database layer. The views are created using a database user in the **Modeler** perspective and later consumed in the ABAP layer using Native SQL or proxy objects. Although these techniques offer the benefit of performing the data processing in the database layer, one limitation of using a bottom-up approach is its complex lifecycle management requirements. For example, handling SAP HANA repository objects must occur separately through delivery units and ABAP proxy objects by using the SAP HANA transport containers to import objects to other systems. Additionally, developers must ensure proper synchronization of all SAP HANA artifacts in all the environments whenever any procedure or

view changed. These shortcomings were later handled with the release of SAP NetWeaver AS ABAP 7.4 SPO5 by introducing progressive code pushdown techniques like ABAP-managed database procedures (AMDPs) and core data services (CDS). These techniques are referred to as top-down approaches because the entire lifecycle management is conducted by the ABAP layer. SAP HANA artifacts such as views and procedures are automatically created in the database.

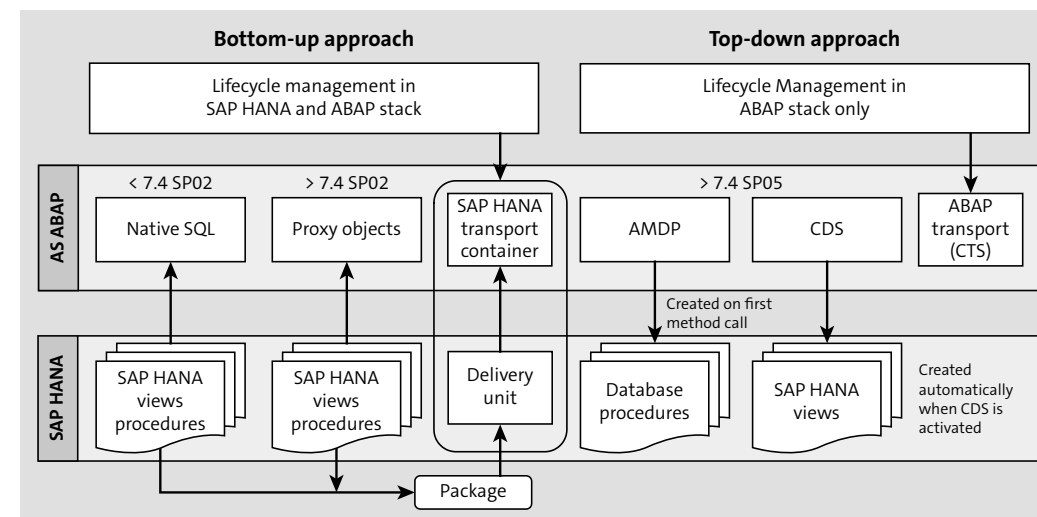


Figure 6.2 Code-to-Data Approach

For example, when an AMDP class and method are defined, a procedure is created in the database before an AMDP method is called for the first time in the calling program and updated on the subsequent calls if it has been changed. In contrast, in the case of a CDS view, the SAP HANA view is created when the CDS view is activated.

### 6.1.1 ABAP-Managed Database Procedure Framework

AMDPs are a recommended technique for achieving code pushdown functionality if your underlying database is SAP HANA. The framework uses a top-down approach to create and manage database procedures in the ABAP environment. The AMDP framework guarantees that the complete lifecycle of the AMDP procedure—from creating, changing, activating, and transporting the procedure—occurs in the application layer by the ABAP runtime environment.

As a developer, you'll write procedures in an AMDP method implementation of an AMDP class. In contrast to the traditional ABAP method, an AMDP method is a unique method and implements database-specific programming languages, such as SQLScript, native SQL, and L (used internally by SAP). The keyword `LANGUAGE` specifies the database-specific language for implementing the procedure.

As shown in Listing 6.1, the usage of the keyword `BY DATABASE PROCEDURE` in the implementation section of the AMDP method helps differentiate whether the method uses ABAP or any other language to implement the procedure.

AMDP Class Definition

```
CLASS zcl_amdp_example DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC.

  PUBLIC SECTION.
```

AMDP Marker Interface

```
INTERFACES if_amdp_marker_hdb.
```

AMDP Method

```
CLASS-METHODS amdp_method
Importing parameter defined using Dictionary type
  IMPORTING VALUE(im_input) TYPE matnr
Importing parameter defined using ABAP type
  EXPORTING VALUE(ex_output) TYPE i
Importing parameter defined using TABLE type
  CHANGING VALUE(ch_param) TYPE ttyp_d.

  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.
```

AMDP Class Implementation

```
CLASS zcl_amdp_example IMPLEMENTATION.
```

AMDP Method Implementation

```
METHOD amdp_method BY DATABASE PROCEDURE
  FOR HDB
  LANGUAGE SQLSCRIPT.
--Implement SQLScript Code ( Database specific code )
ENDMETHOD.
ENDCLASS.
```

#### Listing 6.1 AMDP Framework Definition

In general, you should consider the following points before creating or consuming an AMDP:

- Currently, the AMDP framework only supports database procedures for SAP HANA databases. However, SAP has designed the framework to support stored procedures for other databases.
- You can only create or edit an AMDP using the Eclipse-based ABAP Development Tools (ADT). Thus, the classic SAP GUI-based class builder (Transaction SE24) is not suitable for managing an AMDP class and its methods, as only the display function is supported in Transaction SE24.
- Developers classified as standard ABAP users with appropriate authorizations can manage database procedures using an AMDP class, and Transaction SICK can detect missing permissions.

#### Benefits of Using AMDPs

Let's discuss a few benefits of using AMDPs:

- A standard ABAP user can manage an AMDP, unlike in bottom-up techniques like SAP HANA views and procedures, where both ABAP and SAP HANA database users are required.
- The AMDP framework is responsible for communicating with the database and automatically creating the database procedures as SAP HANA repository catalog objects.
- The entire lifecycle management to synchronize, create, change, activate, and transport procedures is performed in the ABAP layer.
- The **ABAP** perspective within ADT serves as a development environment for writing and managing your SQL scripts.
- The framework supports full integration of SQLScript syntax check and debugging into the ABAP environment.
- Even though an AMDP might be implemented using a database-specific language, such as native SQL or SQLScript, the ABAP environment still evaluates source code for any syntax errors.
- Procedures are automatically created in the SAP HANA database by the ABAP runtime environment before the first AMDP method call.
- You can extend an AMDP using Business Add-Ins (BAdIs) if it has an extension provisioned by the software provider.
- An AMDP is not a replacement for database procedure proxies, which are still considered in sidecar scenarios with secondary database connections to access SQL-Script procedures in a different SAP HANA database.

#### 6.1.2 Development Environment for AMDP

Using ABAP Development Tools (ADT) is mandatory for creating and changing AMDPs, as the classic SAP GUI-based Transaction SE24 only supports the display function. ADT

also delivers additional features for developers to work effectively with AMDPs and improve developer productivity and efficiency.

Some features delivered by ADT for managing ADMPs include the following:

- Code completion functionality for ABAP, accessed by pressing **[Ctrl] + [Space]**
- SQLScript syntax check available in the ABAP environment
- Highlighting of syntax errors in SQLScript
- Analysis of AMDP methods via the debugging functionality
- Highlighted usage of embedded language to distinguish between database-specific language and ABAP code

Let's take a look at some of the features supported by ADT next.

As shown in Figure 6.3, the SAP GUI-based class builder (Transaction SE24) does not allow you to edit an AMDP class and only supports display function; thus, ADT is the preferred development environment for AMDPs.

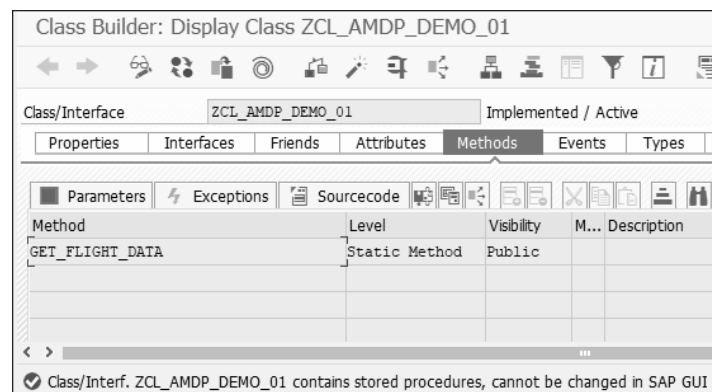


Figure 6.3 SAP GUI-Based Class Builder (Transaction SE24)

In ADT, the appearance of the form-based editor can be changed by the developer to highlight syntax errors and to differentiate between the embedded (database-specific) language and the ABAP language. To enable syntax highlighting, navigate to **Windows • Preferences**, as shown in Figure 6.4.

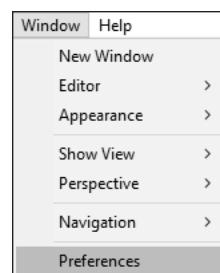


Figure 6.4 Preferences for the Form-Based Editor

Select **General • Appearance • Colors and Fonts • ABAP**, as shown in Figure 6.5. Then, select **Embedded language** under **Syntax Coloring** and click **Edit...** to modify the color according to your preferences.

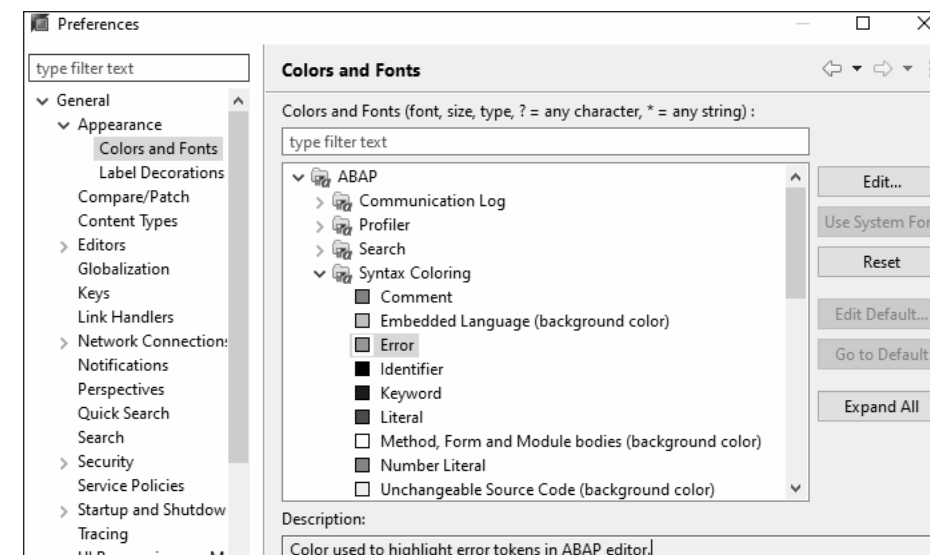


Figure 6.5 Modifying Colors and Fonts of the Form-Based Editor

In ADT, SQLScript syntax error is fully supported and integrated into ABAP. This can be seen by toggling the cursor on the error marker on the right-hand side of the form-based editor. The detailed SQLScript syntax errors can be seen by toggling the cursor on the error, as shown in Figure 6.6.

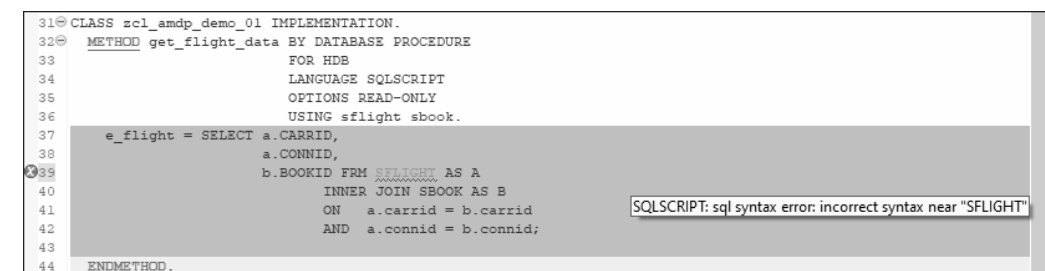


Figure 6.6 SQLScript Syntax Errors

In ADT, you can also highlight the SQLScript syntax errors at the point where they occur. As shown in Figure 6.7, the syntax error statement **SFLIGHT** is emphasized in amber color. Additionally the detailed syntax error description can also be seen on the right-hand side of the source code editor.



```

35 CLASS zcl_amdp_demo_01 IMPLEMENTATION.
36 METHOD get_flight_data BY DATABASE PROCEDURE
37     FOR HDB
38     LANGUAGE SQLSCRIPT
39     OPTIONS READ-ONLY
40     USING sflight sbook.
41
42     -- Data selection from data sources using SQLscript
43     E_FLIGHT = SELECT a.carrid,
44                     a.connid,
45                     sum(loccuram) AS bookamt,
46                     b.loccurkey
47     FROM sflight AS a INNER JOIN
48         sbook AS b
49     ON a.carrid = b.carrid
50     AND a.connid = b.connid
51     WHERE a.mandt = :iv_client
52     GROUP BY A.carrid, a.connid, b.loccurkey;
53
54     -- Filter based on the Selection screen criteria
55     E_FLIGHT = APPLY_FILTER( :E_FLIGHT, :iv_filters );
56
57 ENDMETHOD.
58 ENDClass.

```

**Figure 6.7** Emphasizing SQLScript Errors and Differentiating between ABAP and Database Specific Code

In the AMDP method, you can also set the background color of the embedded language to differentiate between ABAP and database specific code such as SQLScript. Set the color using the same process we discussed earlier in Figure 6.4: navigate to **Windows • Preferences • General • Appearance • Colors and Fonts • ABAP • Embedded language**. Figure 6.7 you can see that the background color for database-specific syntax is emphasized in gray color.

## 6.2 Creating AMDP Classes

A global class must be defined in the class library using ADT to create an AMDP procedure. A class is categorized as an AMDP class if its definition contains one or more tag interfaces. The tag interfaces are prefixed with `IF_AMDP_MARKER` and end in a suffix indicating database system for which the procedure is implemented.

In the following sections, we will understand several prerequisites that you should consider while defining and implementing an AMDP method. You will also learn to consume AMDP and check if current database (or a database specified using a database connection) supports the AMDP features in the ABAP applications.

### Example

The marker interface `IF_AMDP_MARKER_HDB` is relevant for SAP HANA database, where HDB indicates that the procedure is intended for an SAP HANA database.

### 6.2.1 Prerequisites

An AMDP class can be comprised of one or more traditional and AMDP methods. It can also contain AMDPs for each database system specified by a tag interface. The source

code shown in Listing 6.2 illustrates a simple AMDP definition implementing all the required prerequisites. These prerequisites are as follows:

- An AMDP class definition should contain a marker interface `IF_AMDP_MARKER_HDB`, as it implements an AMDP method for the SAP HANA database.
- In the class definition, the AMDP method parameter types should be a dictionary, ABAP (for example, integer or character), or table types. For parameters with table types, the line types should contain elementary components because nested tables are not supported.
- An AMDP method can only contain importing, exporting, and changing parameters. An AMDP method cannot have return parameters.
- Similar to remote function call (RFC) parameters, all method parameters should be defined as *pass by value*. *Pass by reference* in the method definition is not permitted.
- An AMDP method can be defined in the `PUBLIC SECTION`, `PRIVATE SECTION`, or `PROTECTED SECTION` of the class. However, if the AMDP methods of other classes do not call the method, you must declare the method as `PRIVATE`.

```

CLASS zcl_amdp_demo_01 DEFINITION
PUBLIC
FINAL
CREATE PUBLIC.
PUBLIC SECTION.

```

#### AMDP Marker Interface

```
INTERFACES if_amdp_marker_hdb.
```

#### Type Declaration

```

TYPES: BEGIN OF d_flight,
        carrid TYPE s_carr_id,
        connid TYPE s_conn_id,
        bookamt TYPE s_f_cur_pr,
        loccurkey TYPE s_currcode,
        END OF d_flight.
TYPES: tt_flight TYPE STANDARD TABLE OF d_flight.

```

#### AMDP Method Definition

```

CLASS-METHODS get_flight_data
IMPORTING
        VALUE(iv_filters) TYPE string
        VALUE(iv_client) TYPE sy-mandt
EXPORTING
        VALUE(e_flight) TYPE tt_flight

```

```

    CHANGING
        VALUE(c_return) TYPE i

    RAISING cx_amdp_no_connection
           cx_amdp_execution_error.
PROTECTED SECTION.
PRIVATE SECTION.
ENDCLASS.

```

### Listing 6.2 AMDP Class Definition

#### 6.2.2 Implementing AMDP Methods

An AMDP method is a unique method that optimizes ABAP applications by implementing code pushdown from the application server layer to the database layer. This method is wrapped in a global class and can be defined as either a static method or instance method. Even though you can define an AMDP method as an instance method, it will always be executed as a static method call.

Two types of AMDP methods exist:

- An AMDP procedure without a return code is defined by a method using the addition `BY DATABASE PROCEDURE`.
- An AMDP function with a return code is defined by a method using the addition `BY DATABASE FUNCTION`.

Any regular method within an AMDP class can be transformed into an AMDP method by using either the `BY DATABASE PROCEDURE` or `BY DATABASE FUNCTION` addition at the start of the method statement in the method implementation part, followed by the database system for which the procedure is implemented, the language in which the business logic is written, and the mandatory ABAP objects (which may include transparent tables, views, and other AMDPs that are used as data sources).

Additionally, you can mark an AMDP method as `READ ONLY` using the addition `OPTIONS`, which is optional.

The body of an AMDP method, shown in Listing 6.3 uses database-specific language such as SQLScript or native SQL. The source code shown in Listing 6.3 illustrates an AMDP class and method with database-specific logic using SQLScript to summarize sales by airline and flight code. The procedure also filters records based on the selection screen using the `FILTER` keyword. To implement business logic, you can use the full SQLScript reference, except for calculation engine (CE) functions such as the following:

- `CE_LEFT_OUTER_JOIN`
- `CE_COLUMN_TABLE`
- `CE_UNION_ALL`

```

CLASS zcl_amdp_demo_01 IMPLEMENTATION.
    METHOD get_flight_data BY DATABASE PROCEDURE
        FOR HDB
        LANGUAGE SQLSCRIPT
        OPTIONS READ-ONLY
        USING sflight sbook.

```

#### Data selection from data sources using SQLScript

```

E_FLIGHT = SELECT a.carrid,
                 a.connid,
                 sum(loccuram) AS bookamt,
                 b.loccurkey
FROM sflight AS a INNER JOIN
     sbook AS b
ON   a.carrid = b.carrid
AND  a.connid = b.connid
WHERE a.mandt = :iv_client
GROUP BY A.carrid, a.connid, b.loccurkey;

```

#### Filter based on the selection screen criteria

```

E_FLIGHT = APPLY_FILTER( :E_FLIGHT, :iv_filters );

```

```

ENDMETHOD.
ENDCLASS.

```

### Listing 6.3 AMDP Method Implementation Example

Listing 6.3 contains the following elements:

- A global AMDP class implementation contains the AMDP method implementation. In our example, our AMDP method will summarize total flight sales by airline and flight code.
- The method `GET_FLIGHT_DATA` is implemented as an AMDP method since the method is defined with the addition `BY DATABASE PROCEDURE`.
- What follows is the database addition `HDB` to implement the procedure for the SAP HANA database. The AMDP framework only supports SAP HANA database; however, the framework is designed to work with other databases.
- Further, the database-specific language to be used is specified. In this case, SQLScript will be used in the AMDP method to implement the business logic.
- All database objects, such as dictionary tables, views, and other AMDP methods used as data sources within the method body, must be declared explicitly with the keyword `USING`. These objects can be accessed directly, that is, without the need to prefix these objects with `SAP<SID>` (*Schema*). However, for nested AMDP calls, that is, for

AMDP methods called inside an AMDP body, you should specify objects by their full names, that is, with the class they belong to and the method name in uppercase and closed in double quotation marks.

- You must ensure that any objects that are not part of *SAP<SID>* schema are available at runtime since these objects are not managed and therefore are not included in the USING clause.
- Finally, the SQLScript language is used within the AMDP body to write the business logic that is executed in the database layer.
- The procedure results are filtered using the SQLScript function `APPLY_FILTER` based on the selection criteria provided as an importing parameter to the AMDP method.
- The procedure written in the AMDP body is highlighted in gray to differentiate between ABAP-specific language and database-specific language. Refer to Figure 6.4 to see how to set the color for the editor's background.

You should consider the following restrictions when implementing an AMDP method:

- Data definition language (DDL) such as `Create`, `Alter`, or `Delete` are not allowed to create, change, or delete any database objects.
- You cannot access local temporary data objects, such as internal tables, or variables defined in the class definition in the method implementation.
- Statements like database commits and rollbacks are not allowed in the method body. Also, to avoid data inconsistencies between procedures, you should handle logical units of work separately in the ABAP program.
- You cannot extend AMDP methods using implicit enhancement options, as these methods are directly executed on the database, and implicit options are not available within an AMDP method.
- While using data manipulation language (DML), such as `INSERT`, `UPDATE`, `MODIFY`, `DELETE`, etc., write access to buffered tables is not allowed.

### 6.2.3 Calling AMDP Methods in Applications

An AMDP method is called in an ABAP application similar to any other regular method, using an Eclipse-based form editor in ADT or through SAP GUI-based transactions. These methods are always executed as static method calls, even if defined as instance methods.

You can call an AMDP method in an ABAP application in several ways. In the Eclipse-based **ABAP** perspective (see Figure 6.8), you can use the code completion template by pressing `Ctrl` + `Space` to call an AMDP method. In SAP GUI-based editors (see Figure 6.9), you can use ABAP-based patterns by pressing `Ctrl` + `F6`.

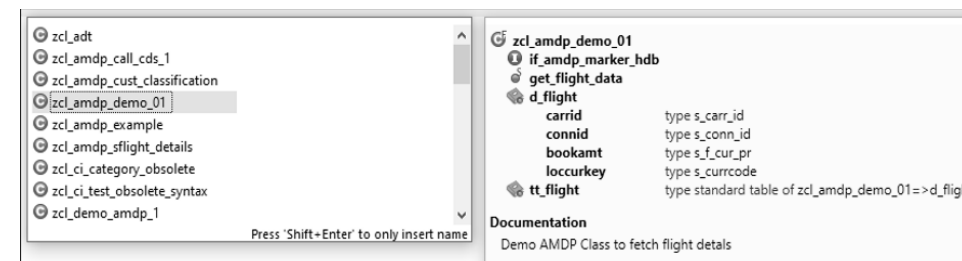


Figure 6.8 Consuming AMDP Method in ABAP (Eclipse)

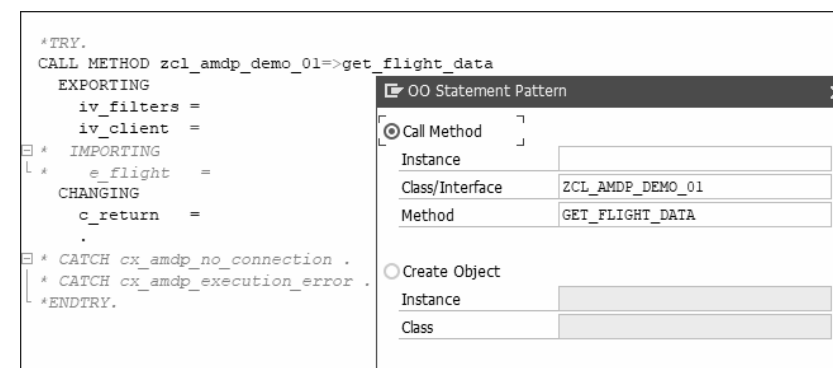


Figure 6.9 Consuming AMDP Method in ABAP (SAP GUI)

However, to consume an AMDP, the SAP NetWeaver AS ABAP's central database should be managed by the database system for which the AMDP method is implemented. If not the case, then the procedure call results in a runtime error.

As shown in Figure 6.10, before the first method call, the ABAP runtime environment creates the procedure implemented in the AMDP method in the database system or updates any existing database procedure if the AMDP has changed, as shown in Figure 6.11. Once the method is called, the execution is performed in the database system. Parameters of the interface are passed from the native SQL interface to the database system or are applied by the database system.

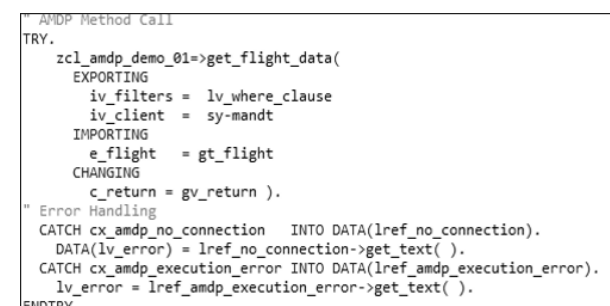


Figure 6.10 AMDP Method Called in Application



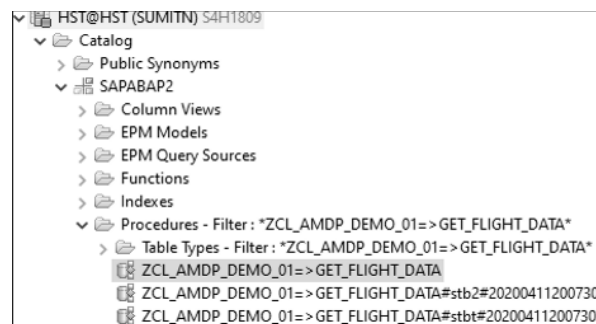


Figure 6.11 AMDP Procedure Created on the Database

Once a database procedure managed using AMDP has been created (ZCL\_AMDP\_DEMO\_01=>GET\_FLIGHT\_DATA) on the database schema, SAPABAP2, as show in Figure 6.11, this procedure can be called from other database procedures using the database syntax, provided that the database permits this access, including AMDP procedures (or database procedures) that are not managed by AMDP. If an AMDP procedure calls another procedure, this procedure must be specified in the calling method with the addition USING.

In general, we recommend that AMDP procedure implementations that are not called from AMDP methods of other classes be created as private methods of an AMDP class and that they be called in regular ABAP methods.

#### Note

In database systems that do not support AMDP, a traditional method can be created using an alternative implementation in Open SQL or native SQL.

As shown in Listing 6.4, an ABAP application can call an AMDP procedure to display flight booking information based on a user's selection. The example also illustrates the use of SELECT-OPTIONS to filter data records.

```
REPORT zcl_amdp_demo_call_01.
```

```
" Data declaration
DATA: gwa_sflight TYPE sflight.

" Select Options
SELECT-OPTIONS: s_carrid FOR gwa_sflight-carrid,
               s_connid FOR gwa_sflight-connid.

" Types Declaration
TYPES: BEGIN OF d_flight,
       carrid TYPE s_carr_id,
       connid TYPE s_conn_id,
```

```
       bookamt TYPE s_f_cur_pr,
       loccurkey TYPE s_currcode,
END OF d_flight.

" Internal table
DATA: gt_flight TYPE STANDARD TABLE OF d_flight.

" Variable
DATA: gv_return TYPE i.

" Build dynamic where clause, and pass it to the AMDP method
TRY.
    DATA(lv_where_clause) = cl_shdb_seltab=>combine_seltabs
        ( it_named_seltabs = VALUE #(
          ( name = 'CARRID' dref = REF #( s_carrid[] ) )
          ( name = 'CONNID' dref = REF #( s_connid[] ) ) ) ).
    CATCH cx_shdb_exception INTO DATA(lref_shdb_exception).
    DATA(lv_meesage) = lref_shdb_exception->get_text( ).
ENDTRY.

" AMDP Method Call
TRY.
    zcl_amdp_demo_01=>get_flight_data(
        EXPORTING
            iv_filters = lv_where_clause
            iv_client = sy-mandt
        IMPORTING
            e_flight = gt_flight
        CHANGING
            c_return = gv_return ).
" Error Handling
CATCH cx_amdp_no_connection INTO DATA(lref_no_connection).
    DATA(lv_error) = lref_no_connection->get_text( ).
CATCH cx_amdp_execution_error INTO DATA(lref_amdp_execution_error).
    lv_error = lref_amdp_execution_error->get_text( ).
ENDTRY.

" Display results
IF lv_error IS INITIAL.
cl_demo_output=>display_data(
    EXPORTING
        value = gt_flight
        name = 'Flight Booking information').
```

```
" Error Handling
ELSE.
  WRITE: lv_error.
ENDIF.
```

#### Listing 6.4 AMDP Method Call in ABAP Application

### 6.2.4 Using Multiple Selection Criteria

In ABAP reports, defining a selection screen is essential to empowering business users so they can filter data based on the desired elements. Selection criteria ensure that applications can process data faster by filtering out unwanted data in the database layer. But, to filter the data, you must define selection criteria using parameters, SELECT-OPTIONS, or a combination of both.

The purpose of parameters is to filter the records based on a single value, whereas with SELECT-OPTIONS, you can define complex selection criteria to filter out records. Developers can then use these selection screen elements directly in a WHERE clause of an Open SQL statement to filter the data. These selection criteria are then converted into the SQL WHERE conditions by the ABAP application server.

However, suppose you want to use these selection screen elements in an AMDP procedure. In this case, you can use parameters directly in the AMDP method, but this approach is not valid with SELECT-OPTIONS.

Because you cannot pass SELECT-OPTIONS directly to an AMDP method, this limitation of using an AMDP must be kept in mind. To pass SELECT-OPTIONS to an AMDP method, you must first transform the selection criteria into a filter string and then pass the string as an IMPORTING parameter to the AMDP method. To convert the SELECT-OPTIONS (selection tables or range tables) into a dynamic SQL WHERE clause, you can use the static method COMBINE\_SELTTABS( ) of the new class CL\_SHDB\_SELTTAB.

This generated condition can then be used in SQLScript to filter the data source using the SQLScript function APPLY\_FILTER in the AMDP method implementation. This function can be applied to database tables, views, and SAP HANA views, however this function cannot be used with analytical or table variables.

The APPLY\_FILTER function expects two parameters. The first parameter is the data source to which you want to apply the filter, and the second parameter is the generated WHERE clause, which is passed as a string argument.

The CL\_SHDB\_SELTTAB class is not available with SAP NetWeaver AS ABAP 7.4 and should be imported by following the steps described in SAP Note 2124672. SAP NetWeaver AS ABAP 7.4 SPO8 or higher is required to apply this SAP Note.

#### Note

The class CL\_LIB\_SELTTAB and its methods are obsolete.

An ABAP report can consume an AMDP method, as shown in Listing 6.5, filtering data based on the user selection via parameters or based on SELECT-OPTIONS to display bookings for all airline codes by a specific date and customer category.

```
REPORT zamdp_sflight_details.
```

#### \* Data declaration

```
DATA: gwa_sflight TYPE sflight.
```

#### \* Selection screen

```
PARAMETERS: p_date TYPE s_date.
```

```
SELECT-OPTIONS: s_carrid FOR gwa_sflight-carrid,
                s_connid FOR gwa_sflight-connid.
```

#### \* Build dynamic where clause

```
TRY.
  DATA(lv_where_clause) = cl_shdb_seltab=>combine_seltabs(
    it_named_seltabs = VALUE #(
      ( name = 'CARRID' dref = REF #( s_carrid[] ) )
      ( name = 'CONNID' dref = REF #( s_connid[] ) ) )
    iv_client_field = 'MANDT' ).
  CATCH cx_shdb_exception INTO DATA(lref_shdb_exception).
    DATA(lv_meesage) = lref_shdb_exception->get_text( ).
ENDTRY.
```

#### \* AMDP Method call to summarize booking amount by flight, airline code, date, and customer type

```
zcl_amdp_sflight_details=>get_data(
  EXPORTING
    iv_client = sy-mandt
    iv_date = p_date
    iv_filters = lv_where_clause
  IMPORTING
    et_results = DATA(gt_results) ).
```

#### \* Display results

```
cl_demo_output=>display_data(
  EXPORTING
    value = gt_results
    name = 'Flight Booking information').
```

#### Listing 6.5 ABAP Report: Handling SELECT-OPTIONS

The AMDP method shown in Listing 6.6 is filtering the data based on parameters and SELECT-OPTIONS passed from the application program shown earlier in Listing 6.5 using the SQLScript function APPLY\_FILTER.

```

CLASS zcl_amdp_sflight_details DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC.

  PUBLIC SECTION.
* AMDP Marker Interface
  INTERFACES: if_amdp_marker_hdb.
* Data declaration
  TYPES: BEGIN OF d_sflight,
          carrid TYPE s_carr_id,
          connid TYPE s_conn_id,
          fldate TYPE s_date,
          type TYPE string,
          total TYPE s_l_cur_pr,
        END OF d_sflight,

        tty_sflight TYPE STANDARD TABLE OF d_sflight.
* AMDP Method
  CLASS-METHODS get_data
  IMPORTING
    VALUE(iv_client) TYPE sy-mandt
    VALUE(iv_date) TYPE s_date
    VALUE(iv_filters) TYPE string
  EXPORTING
    VALUE(et_results) TYPE tty_sflight.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zcl_amdp_sflight_details IMPLEMENTATION.
  METHOD get_data BY DATABASE PROCEDURE
    FOR HDB
    LANGUAGE SQLSCRIPT
    OPTIONS READ-ONLY
    USING sflight sbook.

    ET_RESULTS = SELECT a.carrid, a.connid, b.fldate,
                      CASE b.custtype
                        WHEN 'B' then 'Business Customer'

```

```

                        WHEN 'P' then 'Private Customer'
                        ELSE 'Others'
                      END AS "TYPE",
                      SUM(b.loccuram) AS TOTAL
    from sflight as a INNER JOIN
      sbook as b on a.carrid = b.carrid
    and a.connid = b.connid
    WHERE a.mandt = :iv_client --Parameters
    AND b.fldate = :iv_date --Parameters
    GROUP BY a.carrid, a.connid, b.fldate, b.custtype ;

```

#### \* Filter based on Selection screen (Select options)

```

    ET_RESULTS = APPLY_FILTER( :ET_RESULTS, :iv_filters );
  ENDMETHOD.
ENDCLASS.

```

Listing 6.6 AMDP: Filtering Using SELECT-OPTIONS and Parameters

### 6.2.5 Feature Support Check Using Global Classes

We recommend checking whether the current database or a database specified using a database connection supports AMDP features and if it can be used at runtime in the ABAP applications.

The method USE\_FEATURE of the global class CL\_ABAP\_DBFEATURES can be used to check support for the database-specific feature. Several constants are provided to check database-specific features and can be passed to the USE\_FEATURE method in an internal table. The method returns the value of ABAP\_TRUE if the feature is supported by the database, whereas unsupported values raise an exception from the class CX\_ABAP\_INVALID\_PARAM\_VALUE and can be handled within the application program to avoid runtime errors. The database-specific features listed in Table 6.1 can be validated using the global class CL\_ABAP\_DBFEATURES.

Database Feature	Constant Name	Value
External views	EXTERNAL_VIEWS	2
Maximum number of key fields > 16 (120)	TABLE_KEYCNT_MAX1	3
Maximum width of key fields > 900 bytes (up to 2000)	TABLE_KEYLEN_MAX1	4
Maximum width of table or view > 4030 bytes (up to 16293)	TABLE_LEN_MAX1	5
AMDP table functions	AMDP_TABLE_FUNCTION	6

Table 6.1 Database-Specific Features

Database Feature	Constant Name	Value
AMDP methods are supported	CALL_AMDP_METHOD	8
CALL DATABASE PROCEDURE is supported	CALL_DATABASE_PROCEDURE	7
Internal table as the source in the FROM clause (from release 7.52)	ITABS_IN_FROM_CLAUSE	9
Limit/offset in subselect or common table expressions (CTEs)	LIMIT_IN_SUBSELECT_OR_CTE	10
CTE used in a correlated subquery	CTE_IN_CORRELATED_SUBQUERIES	11
MODIFY FROM SELECT	MODIFY_FROM_SELECT	12
Hierarchies	HIERARCHIES	13
GROUPING SETS	GROUPING_SETS	14

Table 6.1 Database-Specific Features (Cont.)

The AMDP-specific constants `CALL_AMDP_METHOD` and `AMDP_TABLE_FUNCTION` can be passed to the importing parameters of the method `USE_FEATURES` to validate if the underlying database supports the AMDP procedure, as shown in Listing 6.7.

```

TRY.
  DATA(lv_supported) = cl_abap_dbfeatures=>use_
  features( EXPORTING requested_features =
  VALUE #( ( cl_abap_dbfeatures=>call_amdp_method )
  ( cl_abap_dbfeatures=>amdp_table_function ) ) ).
CATCH cx_abap_invalid_param_value INTO DATA(lref_invalid_value).
  DATA(lv_message) = lref_invalid_value->get_text( ).
ENDTRY.
IF lv_supported IS NOT INITIAL."ABAP_TRUE
  WRITE:'Database specific feature', cl_abap_dbfeatures=>call_amdp_
  method, 'is supported'.
ELSE.
  WRITE lv_message.
ENDIF.

```

Listing 6.7 Database Specific Feature Check

You can also use the standard program `DEMO_DBFEATURES` to validate if the current database supports any database features before using them.

To validate database-specific features, follow these steps:

1. Execute the standard SAP program `DEMO_DBFEATURES` using Transaction `SE38` (see Figure 6.12).

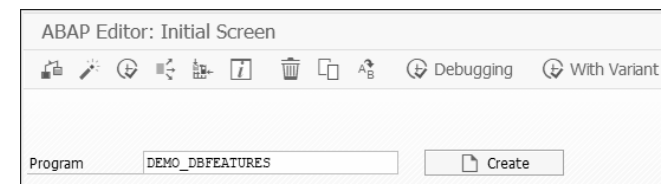


Figure 6.12 Execute DEMO\_DBFEATURES in Transaction SE38

2. You can choose the database features to validate for the SAP HANA database version (see Figure 6.13). Click on **Enter** to view the results.

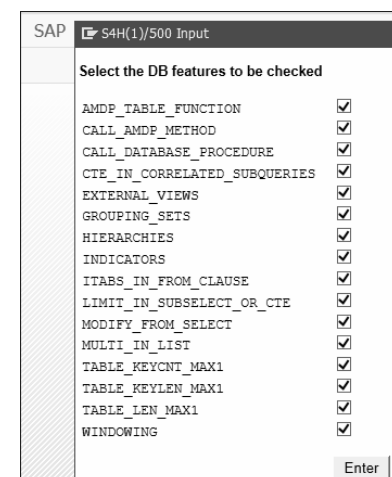


Figure 6.13 Select Database Features for AMBP Supportability Check

3. The report displays the list of supported and unsupported features for the underlying database (see Figure 6.14).

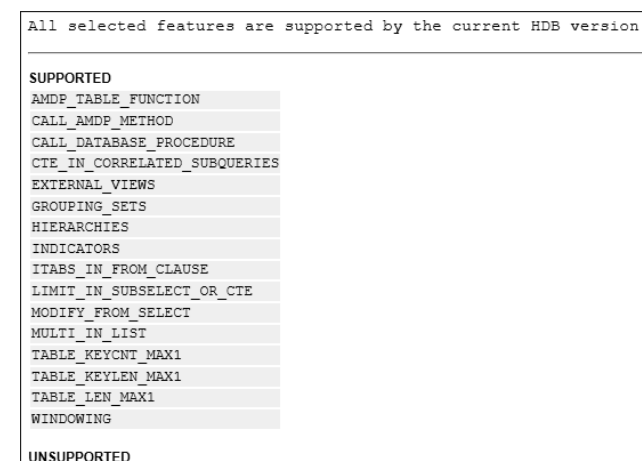


Figure 6.14 Resulting List of Supported and Unsupported AMDP Features

## 6.3 Enhancements

Similar to classic ABAP extensions, where several enhancement techniques like user exits, customer exits, business transaction events (BTE), and business add-ins (BAI), enhancement frameworks are available to perform modification-free extensions to SAP applications. These enhancements frameworks include implicit and explicit enhancements or are BAI-managed using enhancements spots.

In the following sections, you'll learn how to define, implement, and invoke AMDP BAI calls within other AMDP methods to extend standard business functionality.

### 6.3.1 AMDP BAI Overview

You can also extend an AMDP procedure if the software or extension provides for this extensibility. As described in Table 6.2, AMDP BAIs were introduced with SAP NetWeaver AS ABAP 7.4 SPO8 to allow for modification-free extensions. As shown in Figure 6.15, you could then consume these extensions to add or modify a business requirement in the procedure.

Software or Extension Provider	Customer or Partner Extension Consumer
<ul style="list-style-type: none"> <li>Responsible for creating an enhancement spot in Transaction SE20</li> <li>Responsible for creating a BAI definition categorized as an AMDP BAI, defines the BAI interface, and implements the fallback class</li> <li>Responsible for integrating the enhancement spot with the application</li> </ul>	<ul style="list-style-type: none"> <li>Responsible for providing an implementation class for the AMDP BAI</li> <li>Responsible for creating an active BAI implementation with SQLScript code to extend or add a business requirement</li> </ul>

Table 6.2 Modification-Free Extensions

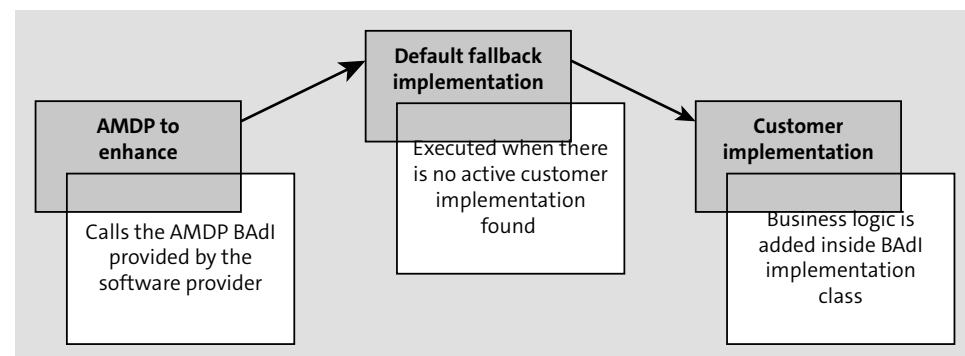


Figure 6.15 AMDP BAI Framework

In addition to implementing the methods of a normal BAI as AMDP methods and making these methods callable using CALL BAI, you can also create a special AMDP BAI.

An AMDP BAI is created in Transaction SE20 and is later called within an AMDP implementation, similar to other AMDPs. An AMDP BAI is a BAI that is categorized accordingly in the BAI Builder and meets the following prerequisites, which are shown in Figure 6.16:

- SAP has provided an **Enhancement Spot** (ES\_PPH\_READ\_BAI, in our example) and a **BAI Definition** (PPH\_AMDP\_READ\_MRP\_BAI, in our example).
- An AMDP BAI does not currently have any filters, as they are not supported, indicated by the unchecked **Limited filter use** box in the **Usability** section.
- In its definition, the BAI is categorized as an **AMDP BAI**, as you can see in the **Usability** section.
- The software provider has provided the mandatory **Fallback Class**. In this example, the fallback class is CL\_PPH\_AMDP\_READ\_MRP\_BAI. Only an AMDP class can be provided as a fallback class or implementation class.

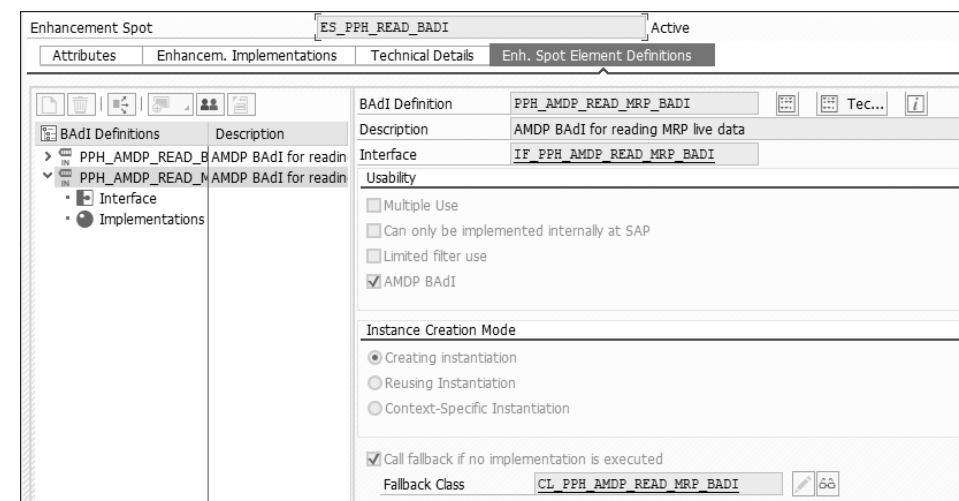


Figure 6.16 AMDP BAI Prerequisites

- Additionally, every BAI method of an AMDP BAI must be an AMDP method and must be implemented for the same database platform (only SAP HANA currently supported). This is shown in Figure 6.17 where the addition FOR DATABASE PROCEDURE FOR HDB is used to specify the database platform in the AMDP method IF\_PPH\_AMDP\_READ\_MRP\_BAI~MDPSX\_CHANGE. The addition HDB indicates this procedure is only relevant for the SAP HANA database.





To extend the ADMP method shown in Figure 6.20, the software provider needs to provision an AMDP BAdI. In this case, BAdI definition ZBADI\_RECLASSIFY\_CUSTOMERS is available and encapsulated in the enhancement spot ZES\_RECLASSIFY\_CUSTOMER. You can view the AMDP BAdI definition in Transaction SE18. Let's look at the definition more deeply, especially the following aspects:

1. An AMDP BAdI definition ZBADI\_RECLASSIFY\_CUSTOMERS is provided by SAP to extend the procedure and is encapsulated in the **Enhancement Spot** ZES\_RECLASSIFY\_CUSTOMER, as shown in Figure 6.22.

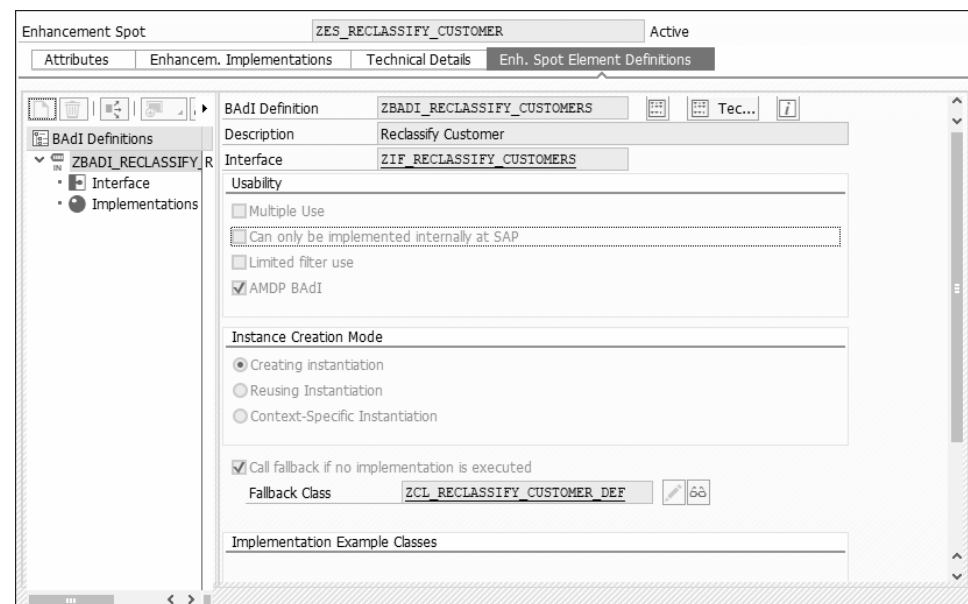


Figure 6.22 AMDP BAdI Definition

2. If no active implementation exists for the AMDP BAdI under **Implementations** section, the default implementation of the fallback class ZCL\_RECLASSIFY\_CUSTOMER\_DEF will be executed, as shown in Figure 6.23.

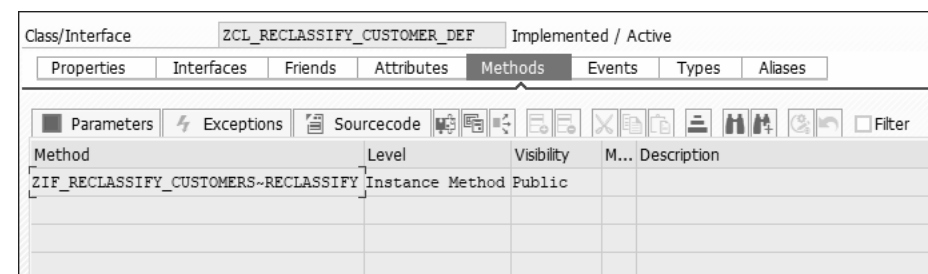


Figure 6.23 AMDP BAdI Method

3. Our AMDP BAdI method ZIF\_RECLASSIFY\_CUSTOMERS~RECLASSIFY will encapsulate the business logic written in SQLScript language, as shown in Figure 6.24, to classify and determine the customer category.

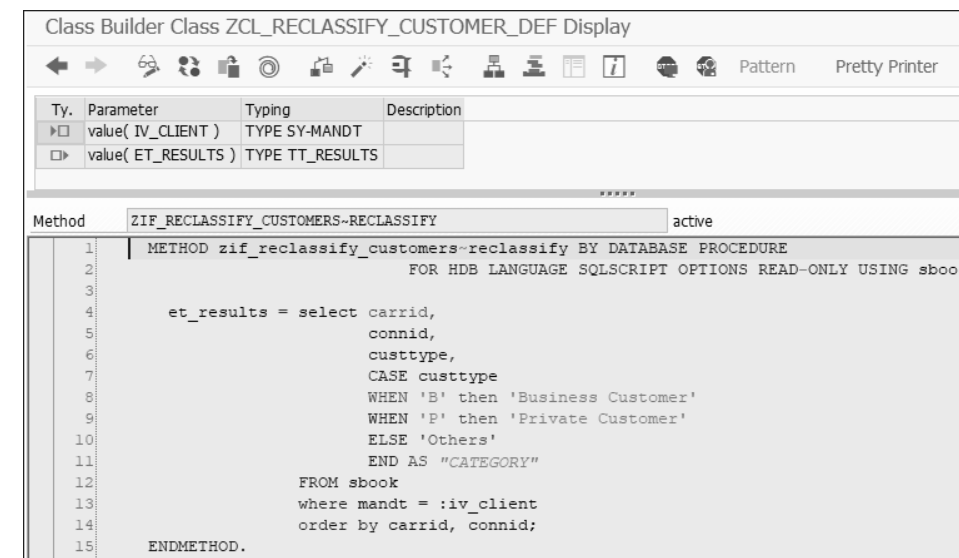


Figure 6.24 AMDP BAdI Method with SQLScript Logic

### 6.3.2 AMDP BAdI Implementation

To extend an AMDP method ZIF\_RECLASSIFY\_CUSTOMERS~RECLASSIFY shown in Figure 6.24, open the BAdI definition in Transaction SE18 and follow these steps to implement the BAdI definition:

1. After opening the BAdI definition in Transaction SE18, right-click on the BAdI definition name ZBADI\_RECLASSIFY\_CUSTOMERS and select on **Create BAdI Implementation**, as shown in Figure 6.25.

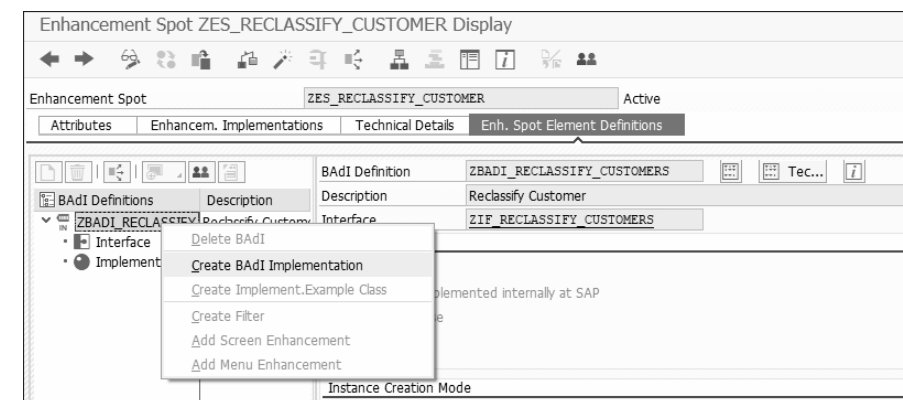




Figure 6.25 Create BAdI Implementation

2. Create the enhancement implementation by providing a name and a short text and then clicking on **OK**, as shown in Figure 6.26.

Enhancement Implementation	ZBADI_RECLASSIFY_CUSTOMERS_IMP
Short Text	Implementation to redetermine customers
Composite Enhancement Implementation	 



 

Figure 6.26 Create Enhancement Implementation

3. Now, create a BAdI implementation by providing the BAdI implementation a name and specifying the class to extend the AMDP method, as shown in Figure 6.27. Then, click on **OK**.

TRM(1)/200 Create BAdI Implementation	
BAdI Implementation	ZBADI_RECLASSIFY_CUSTOMERS_IMP
Description	Implementation to redertime category
Implementing Class	ZCL_RECLASSIFY_CUSTOMER_IMP



 

Figure 6.27 Name and Describe BAdI Implementation and Specify Implementing Class

4. Click on **Save** or use the shortcut (**Ctrl + S**) to save the BAdI implementation.
5. Click on the **Implementing Class** (see Figure 6.28) to extend the AMDP method RECLASSIFY and incorporate the customer requirement using database-specific language (SQLScript). The customer category will be determined in this step, with 'Privilege Customer' for customer type B and 'General Customer' for customer type P, as shown in Figure 6.29.
6. Click on **Save** and **Activate** to activate the implementation. The active implementation is shown in Figure 6.30.

Enhancement Implementation	ZBADI_RECLASSIFY_CUSTOMERS_IMP	Inactive								
Properties History Technical Details Enh. Implementation Elements										
<table border="1"> <tr> <th>Implementing Class</th> <th>Interface</th> <th>Method</th> <th>Short Description</th> </tr> <tr> <td>ZCL_RECLASSIFY_CUSTOMER_IMP</td> <td>ZIF_RECLASSIFY_CUSTOMERS</td> <td>ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY</td> <td></td> </tr> </table>			Implementing Class	Interface	Method	Short Description	ZCL_RECLASSIFY_CUSTOMER_IMP	ZIF_RECLASSIFY_CUSTOMERS	ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY	
Implementing Class	Interface	Method	Short Description							
ZCL_RECLASSIFY_CUSTOMER_IMP	ZIF_RECLASSIFY_CUSTOMERS	ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY								

Figure 6.28 Select the Implementing Class to Extend the AMDP Method

```

class ZCL_RECLASSIFY_CUSTOMER_IMP definition
  public
  final
  create public .

public section.

  interfaces IF_AMDP_MARKER_HDB .
  interfaces IF_BADI_INTERFACE .
  interfaces ZIF_RECLASSIFY_CUSTOMERS .
protected section.
private section.
ENDCLASS.

CLASS ZCL_RECLASSIFY_CUSTOMER_IMP IMPLEMENTATION.

  method ZIF_RECLASSIFY_CUSTOMERS~RECLASSIFY BY DATABASE PROCEDURE
    FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY USING sbook.

    et_results = select carrid,
                    connid,
                    custtype,
                    CASE custtype
                      WHEN 'B' then 'Privilege Customer'
                      WHEN 'P' then 'General Customer'
                      ELSE 'Others'
                    END AS "CATEGORY"
    FROM sbook
    where mandt = :iv_client
    order by carrid, connid;

  endmethod.
ENDCLASS.

```

Figure 6.29 Extend AMDP Method

Enhancement Spot	ZES_RECLASSIFY_CUSTOMER	Active
Attributes Enhancem. Implementations Technical Details Enh. Spot Element Definitions		
Enhancement implementations exist for this enhancement spot		
Enhancement Implementation	Version	
ZBADI_RECLASSIFY_CUSTOMERS_IMP	A	

Figure 6.30 Active Enhancement Implementation

7. After implementing the BAdI definition, the business requirement is extended in the implementing class, to redetermine the business category as 'Privilege Customer' or 'General Customer' based on the customer type, as shown in Figure 6.31.
8. Execute **F8** the application that calls the AMDP BAdI (see Figure 6.32) to display the customer classification data, as shown in Figure 6.33. The application program reclassifies the customer category to 'Privilege Customer' or 'General Customer' category using BAdI extensions.



- Choose the object type **Enhancement Spot** and click on **Next** to create the enhancement spot, as shown in Figure 6.36.

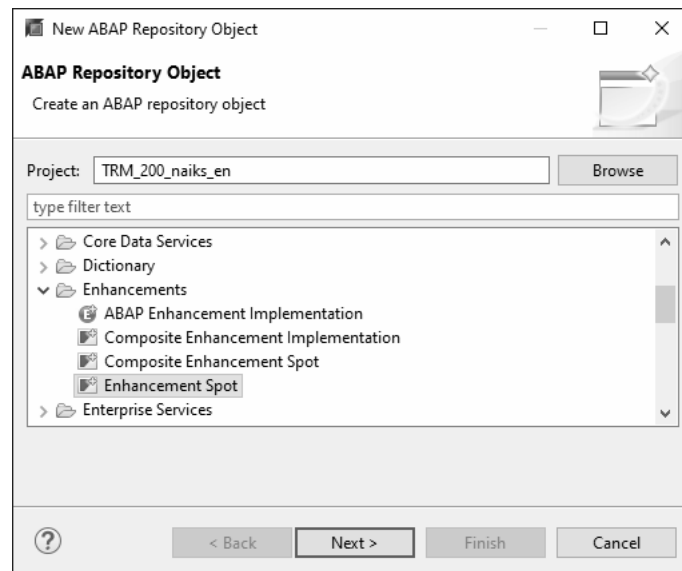


Figure 6.36 Create an Enhancement Spot using ADT in SAP HANA Studio

- To create the enhancement spot, maintain the **Object Name** field and click on **Next**, as shown in Figure 6.37.

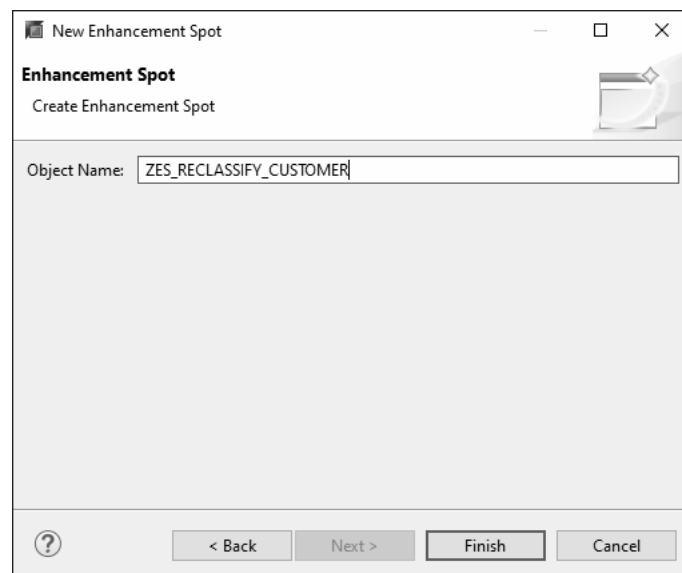


Figure 6.37 Specify Object Name for the Enhancement Spot using ADT in SAP HANA Studio

- Alternatively, you can also use Transaction SE20 to create the BAdI definition (see Figure 6.38).

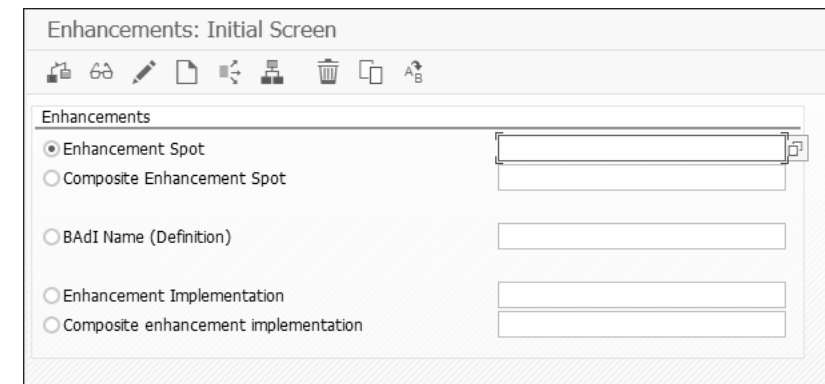


Figure 6.38 Create an Enhancement Spot using Transaction SE20

- Next, specify a name for the enhancement spot and maintain the **Short Text** and **Technology** fields, as shown in Figure 6.39. To create the enhancement spot, click on **OK**.

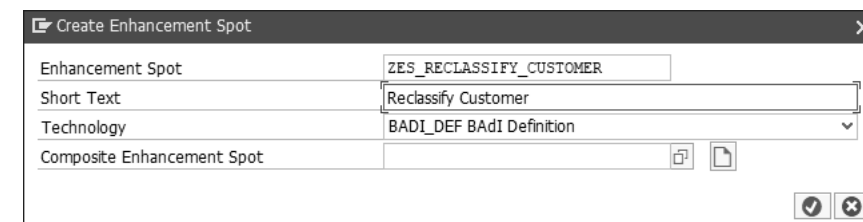


Figure 6.39 Specify Name and Short Text for the Enhancement Spot Using Transaction SE20

- Under the **Enh. Spot Element Definitions** tab, as shown in Figure 6.40, click on **Create BAdI** to create the definition.

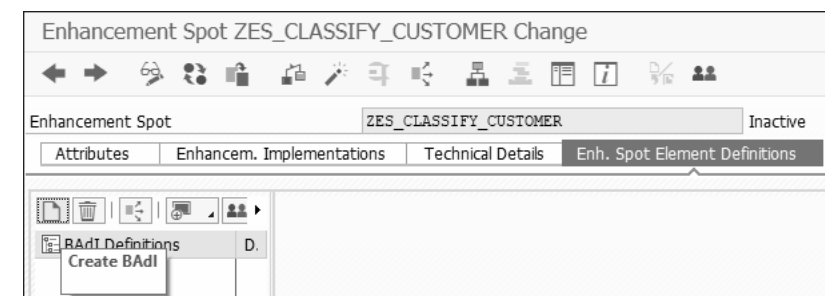


Figure 6.40 Create a BAdI Definition



7. Provide the BAdI a name and a short description and click on **OK** to create the definition, as shown in Figure 6.41.

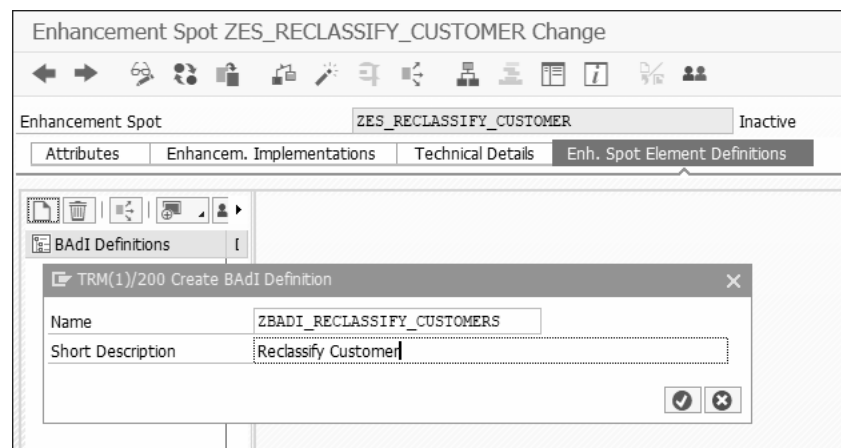


Figure 6.41 Specify BAdI Name and Short Description

8. Under the **Usability** section, categorize the BAdI definition as an AMDP BAdI by selecting the **AMDP BAdI** checkbox, as shown in Figure 6.42. You cannot classify the BAdI as filter dependent as the AMDP BAdI does not support filter functionality.

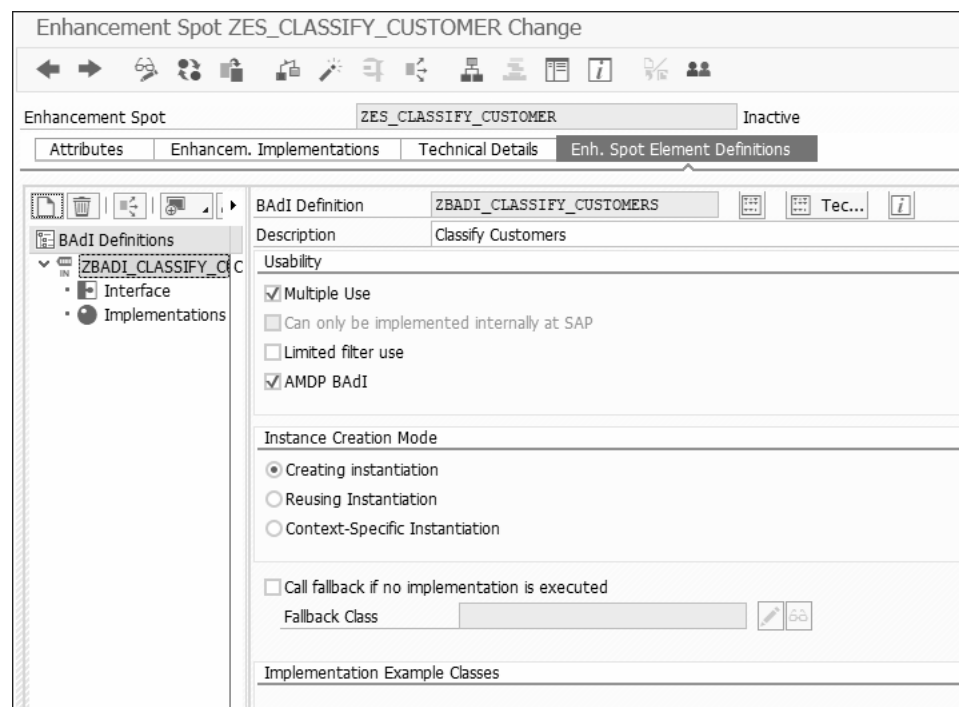


Figure 6.42 Categorize BAdI Definition as AMDP BAdI

9. Click on the **Interface** node under **BAdI Definitions** and specify the BAdI interface name ZIF\_RECLASSIFY\_CUSTOMERS and then click on **Yes** to create the BAdI interface, as shown in Figure 6.43.

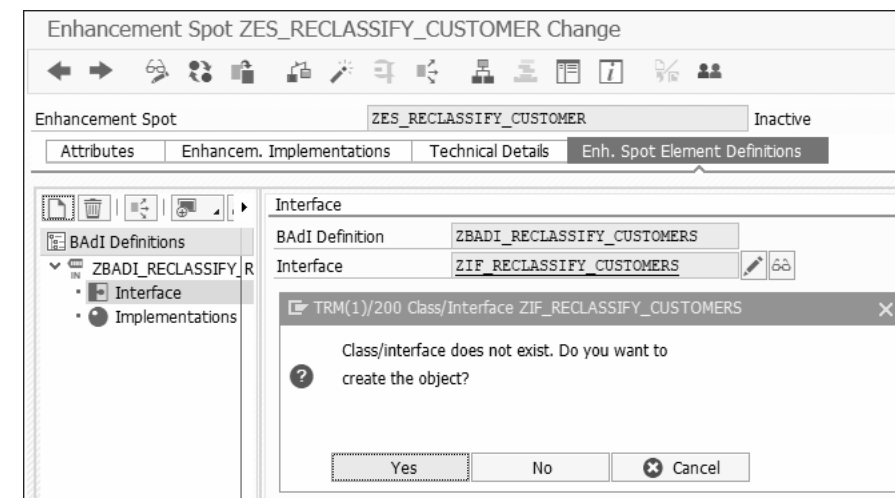


Figure 6.43 Create a BAdI Interface

10. Click on the interface name ZIF\_RECLASSIFY\_CUSTOMERS to define the AMDP method RECLASSIFY in the BAdI interface, as shown in Figure 6.44.

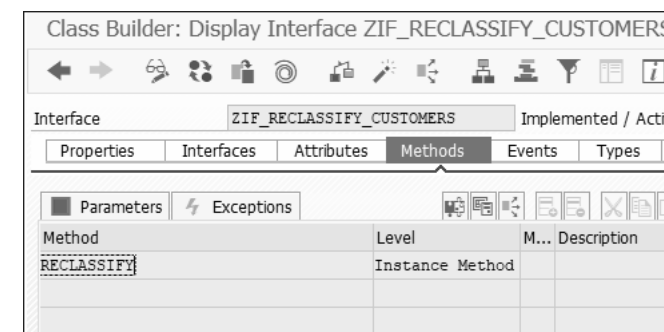


Figure 6.44 Define AMDP Method in BAdI Interface

11. In the BAdI interface, as shown in Figure 6.45, include the AMDP marker interface IF\_AMDP\_MARKER\_HDB for the database for which the procedure is to be created, in our case, this is specific to the SAP HANA database.

```

1 INTERFACE zif_reclassify_customers
2   PUBLIC .
3
4   INTERFACES if_badi_interface .
5   INTERFACES if_amdp_marker_hdb.
6
7   TYPES: BEGIN OF d_results,
8           carrid TYPE s_carr_id,
9           connid TYPE s_conn_id,
10          custtype TYPE s_custtype,
11          category TYPE string,
12          END OF d_results,
13
14          tt_results TYPE STANDARD TABLE OF d_results.
15
16   METHODS: reclassify
17             IMPORTING
18               VALUE(iv_client) TYPE sy-mandt
19             EXPORTING
20               VALUE(et_results) TYPE tt_results.
21 ENDINTERFACE.

```

Figure 6.45 AMDP Method Definition

- Finally, click on the fallback class ZCL\_RECLASSIFY\_CUSTOMER\_DEF. Click **Yes** to create the fallback class, as shown in Figure 6.46. Then, encapsulate the business logic in the AMDP method ZIF\_RECLASSIFY\_CUSTOMERS~RECLASSIFY, as shown in Figure 6.47. If no active implementation is created for the BAdI definition, then the runtime environment calls the active implementation in the fallback class when the AMDP BAdI is called.

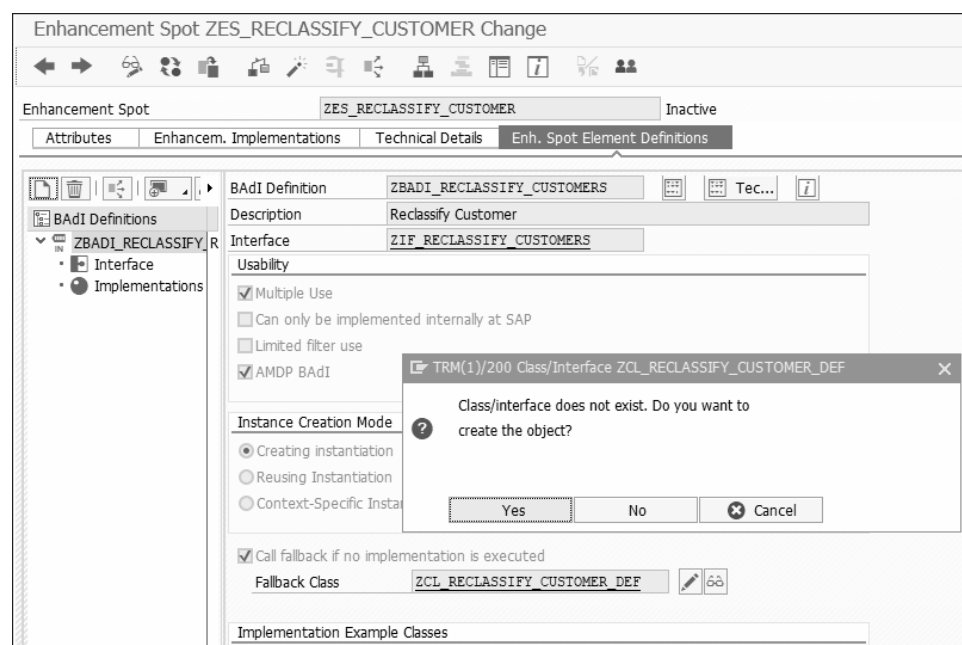


Figure 6.46 Create a Fallback Class

```

Class Builder Class ZCL_RECLASSIFY_CUSTOMER_DEF Display
-----
Ty. Parameter Typing Description
value( iv_client ) TYPE SY-MANDT
value( et_results ) TYPE TT_RESULTS

Method ZIF_RECLASSIFY_CUSTOMERS-RECLASSIFY active
-----
1 METHOD zif_reclassify_customers-reclassify BY DATABASE PROCEDURE
2   FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY USING sbook.
3
4   et_results = select carrid,
5                   connid,
6                   custtype,
7                   CASE custtype
8                     WHEN 'B' then 'Business Customer'
9                     WHEN 'P' then 'Private Customer'
10                    ELSE 'Others'
11                    END AS "CATEGORY"
12   FROM sbook
13   where mandt = :iv_client
14   order by carrid, connid;
15 ENDMETHOD.

```

Figure 6.47 Encapsulate Business Logic in AMDP Method

Listing 6.8 illustrates an AMDP interface ZIF\_RECLASSIFY\_CUSTOMERS definition where an AMDP Method RECLASSIFY has been defined, the AMDP method RECLASSIFY is implemented in the fallback class ZCL\_RECLASSIFY\_CUSTOMER\_DEF to encapsulate the business logic written in database specific SQL language to reclassify customers.

\*AMDP Interface Definition

```

INTERFACE zif_reclassify_customers
PUBLIC .
INTERFACES if_badi_interface .
INTERFACES if_amdp_marker_hdb.

```

```

TYPES: BEGIN OF d_results,
        carrid TYPE s_carr_id,
        connid TYPE s_conn_id,
        custtype TYPE s_custtype,
        category TYPE string,
        END OF d_results,

        tt_results TYPE STANDARD TABLE OF d_results WITH EMPTY KEY.

```

```

METHODS: reclassify
IMPORTING
  VALUE(iv_client) TYPE sy-mandt
EXPORTING
  VALUE(et_results) TYPE tt_results.
ENDINTERFACE.

```

\*AMDP Method Implementation in the fallback Class ZCL\_RECLASSIFY\_CUSTOMER\_DEF

```

METHOD zif_reclassify_customers~reclassify BY DATABASE PROCEDURE

```

```

                FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY
USING sbook.

    et_results = select carrid,
                  connid,
                  custtype,
                  CASE custtype
                    WHEN 'B' then 'Business Customer'
                    WHEN 'P' then 'Private Customer'
                    ELSE 'Others'
                  END AS "CATEGORY"
    FROM sbook
    where mandt = :iv_client
    order by carrid, connid;

ENDMETHOD.

```

Listing 6.8 BADI Interface and AMDP Method Implementation

### 6.3.4 AMDP BADI Calls

Software or extension providers usually call the AMDP method of an AMDP BADI interface in another AMDP method or an application program to support modification-free extensions to AMDPs. The AMDP framework then generates a database procedure for each AMDP BADI. An AMDP BADI's name consists of the BADI name and the interface method with the => separator.

If you want to call an AMDP BADI in an AMDP method, its usage must be first defined in the AMDP method implementation with the USING clause. Inside the AMDP method body, the BADI call is invoked by specifying the BADI definition and method name in uppercase and with the => separator. The BADI call is enclosed in double quotation marks, and furthermore, importing, exporting, and changing parameters are passed to the interface method. To define the usage of an AMDP BADI, follow these steps:

1. In the AMDP method implementation, the usage of the BADI definition is specified in the USING clause, as shown in Figure 6.48.

```

27 CLASS zcl_amdp_cust_classification IMPLEMENTATION.
28 METHOD execute BY DATABASE PROCEDURE
29     FOR HDB
30     LANGUAGE SQLSCRIPT
31     OPTIONS READ-ONLY
32 * BADI usage
33     USING zbadi_reclassify_customers=>reclassify.
34
35 -- BADI call
36 CALL "ZBADI_RECLASSIFY_CUSTOMERS=>RECLASSIFY" (:IV_CLIENT, :ET_RESULTS );
37
38 ENDMETHOD.
39 ENDClass.

```

Figure 6.48 BADI Definition Usage in an AMDP Method

2. The BADI call is invoked by specifying both the BADI definition and the method name in uppercase and with the => separator. As shown in Figure 6.49, the BADI call is enclosed in double quotation marks, and furthermore, importing, exporting, and changing parameters are passed to the interface method.

```

CLASS zcl_amdp_cust_classification IMPLEMENTATION.
  METHOD execute
    BY DATABASE PROCEDURE FOR HDB LANGUAGE SQLSCRIPT OPTIONS READ-ONLY
  * BADI usage
    USING zbadi_reclassify_customers=>reclassify.
  -- BADI call
  CALL "ZBADI_RECLASSIFY_CUSTOMERS=>RECLASSIFY" (:IV_CLIENT, :ET_RESULTS );
  ENDMETHOD.
ENDCLASS.

```

Figure 6.49 BADI Call in an AMDP Method

3. Finally, as shown in Figure 6.50, the AMDP BADI call is invoked in an ABAP application to derive the customer category classification.

```

DATA: gref_flights_check TYPE REF TO zbadi_reclassify_customers,
      gt_results         TYPE zif_reclassify_customers=>tt_results.

* AMDP BADI Handle
GET BADI gref_flights_check .

* AMDP BADI Call
CALL BADI gref_flights_check->reclassify
  EXPORTING
    iv_client = sy-mandt
  IMPORTING
    et_results = gt_results.

cl_demo_output=>display_data(
  EXPORTING
    value = gt_results
    name = 'Customer Category Classification').

```

Figure 6.50 AMDP BADI Call Invoked in ABAP Application

The source code shown in Listing 6.9 illustrates a BADI AMDP method call in another AMDP method.

```

CLASS zcl_amdp_cust_classification DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC.
  PUBLIC SECTION.

  * Marker Interface
  INTERFACES if_amdp_marker_hdb.

  * Type Definition
  TYPES: BEGIN OF d_results,
          carrid TYPE s_carr_id,

```

```

        connid TYPE s_conn_id,
        custtype TYPE s_custtype,
        category TYPE string,
    END OF d_results,
* Table Type
    tt_results TYPE STANDARD TABLE OF d_results WITH EMPTY KEY.
* AMDP Method
    METHODS: execute
    IMPORTING
        VALUE(iv_client) TYPE sy-mandt
    EXPORTING
        VALUE(et_results) TYPE tt_results.
    PROTECTED SECTION.
    PRIVATE SECTION.
ENDCLASS.

CLASS zcl_amdp_cust_classification IMPLEMENTATION.
    METHOD execute BY DATABASE PROCEDURE
        FOR HDB
        LANGUAGE SQLSCRIPT
        OPTIONS READ-ONLY
* Specify BAdI Usage before Call
        USING zbadi_reclassify_customers=>reclassify.
-- BAdI Call
    CALL "ZBADI_RECLASSIFY_CUSTOMERS=>RECLASSIFY"
        (:IV_CLIENT, :ET_RESULTS );
    ENDMETHOD.

```

Listing 6.9 BAdI Method Call in an AMDP Method

As shown in Listing 6.10, an AMDP BAdI method call in any application is similar to a kernel BAdI call using a CALL BAdI or GET BAdI statement.

```
REPORT zamdp_cust_classification.
```

```
DATA: gref_flights_check TYPE REF TO zbadi_reclassify_customers,
      gt_results         TYPE zif_reclassify_customers=>tt_results.
```

#### \* AMDP BAdI Handle

```
GET BADI gref_flights_check.
```

#### \* AMDP BAdI Call

```
CALL BADI gref_flights_check->reclassify
    EXPORTING
        iv_client = sy-mandt
```

```

IMPORTING
    et_results = gt_results.

cl_demo_output=>display_data(
    EXPORTING
        value = gt_results
        name = 'Customer Category Classification').

```

Listing 6.10 BAdI Method Call in an Application Program

## 6.4 Exception Handling

Like any other traditional method call, an AMDP method call may produce ABAP runtime errors. Even though these calls are executed directly in the SAP HANA database, runtime errors can be traced and analyzed in Transaction ST22. This transaction provides additional AMDP-relevant information in the **Database Procedure (AMDP) Information** area under **ABAP Developer View**.

These errors can occur for several reasons, such as a version mismatch with the stored procedure, while creating or executing the method. Other reasons include missing authorizations or database connectivity errors.

Exceptions are raised when any such error occurs either by the runtime environment or by the method definition's RAISE exception statement. Therefore, exceptions should be handled in the calling program to avoid runtime errors during program execution.

Figure 6.51 shows the exception classes that can be handled within an AMDP method to avoid runtime errors.

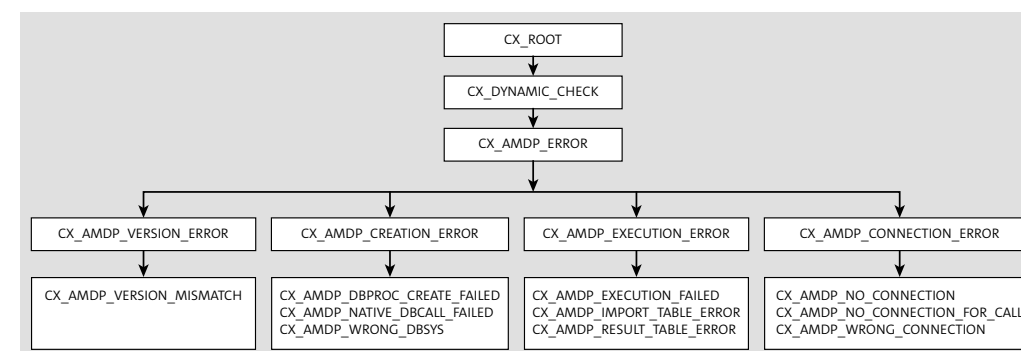


Figure 6.51 AMDP Exception Classes

The methods are prefixed with CX\_AMDP, indicating these classes are exception classes for handling errors during AMDP calls. These exception classes belong to the CX\_DYNAMIC\_CHECK category and must be declared explicitly using the RAISING addition in the definition of an AMDP method and handled when called in the application program.

# Contents

Acknowledgments .....	17
<b>1 SAP HANA Overview</b> .....	<b>19</b>
<b>1.1 Features</b> .....	<b>19</b>
1.1.1 Main Memory and CPU Innovations .....	21
1.1.2 Storage Innovations .....	22
1.1.3 Data Compression .....	23
1.1.4 Data Partitioning .....	25
<b>1.2 Architecture</b> .....	<b>27</b>
<b>1.3 Platform Capabilities</b> .....	<b>29</b>
<b>1.4 Use Cases</b> .....	<b>30</b>
1.4.1 Side-by-Side Scenario .....	30
1.4.2 Fully Integrated Scenarios .....	32
1.4.3 New Applications .....	32
<b>1.5 Summary</b> .....	<b>33</b>
<b>2 Code-to-Data Paradigm</b> .....	<b>35</b>
<b>2.1 What Is the Code-to-Data Paradigm?</b> .....	<b>35</b>
<b>2.2 Bottom-Up Approach</b> .....	<b>38</b>
2.2.1 Using Native SQL .....	39
2.2.2 Using Proxy Objects .....	40
<b>2.3 Top-Down Approach</b> .....	<b>41</b>
2.3.1 Using Transparent Optimization .....	42
2.3.2 Using ABAP SQL .....	43
2.3.3 Using ABAP-Managed Database Procedures .....	45
2.3.4 Using Core Data Services .....	45
<b>2.4 Performance Benefits of Code-to-Data Techniques</b> .....	<b>46</b>
<b>2.5 Summary</b> .....	<b>49</b>



<b>3</b>	<b>Development Environments</b>	51
<b>3.1</b>	<b>Evolution of Programming Languages and Development Tools</b>	51
<b>3.2</b>	<b>SAP HANA Studio</b>	54
3.2.1	Compatibility Checks	56
3.2.2	Download	60
3.2.3	Installation	64
3.2.4	Basic Navigation and Actions	67
3.2.5	ABAP Perspective	72
3.2.6	Modeler Perspective	75
3.2.7	Administration Perspective	83
<b>3.3</b>	<b>SAP HANA Client</b>	87
<b>3.4</b>	<b>ABAP Development Tools</b>	88
3.4.1	Overview	88
3.4.2	Installation	90
<b>3.5</b>	<b>SAP Business Technology Platform, ABAP Environment</b>	99
<b>3.6</b>	<b>Summary</b>	101
<b>4</b>	<b>Native SQL</b>	103
<b>4.1</b>	<b>Executing Native SQL Statements</b>	104
4.1.1	Literals and Host Variables	105
4.1.2	Statement for Cursor Processing	105
4.1.3	Database Procedure Calls	106
4.1.4	Statements for Establishing Database Connections	107
4.1.5	Data Type Compatibility	108
<b>4.2</b>	<b>ABAP Database Connectivity</b>	109
4.2.1	Querying the Database	110
4.2.2	DDL and DML Operations	115
4.2.3	Secondary Connections	118
4.2.4	Precautions While Using ABAP Database Connectivity	120
<b>4.3</b>	<b>Summary</b>	121

<b>5</b>	<b>SQLScript Programming</b>	123
<b>5.1</b>	<b>What Is SQLScript?</b>	123
5.1.1	SQL versus SQLScript	124
5.1.2	Types of SQLScript Statements	125
5.1.3	Prerequisites	125
<b>5.2</b>	<b>SQL Query Template</b>	128
5.2.1	Creating a Schema	128
5.2.2	Creating Tables	129
<b>5.3</b>	<b>Stored Procedures</b>	131
5.3.1	Creating Procedures	131
5.3.2	Deleting Procedures	133
<b>5.4</b>	<b>User-Defined Functions</b>	133
5.4.1	Joins and GROUP BY Clauses	135
5.4.2	Subqueries	137
<b>5.5</b>	<b>Constructs</b>	139
5.5.1	Local Scalar Variables	139
5.5.2	Local Table Variables	139
5.5.3	Conditional Statements	140
5.5.4	FOR Loops and WHILE Loops	141
5.5.5	Operators	141
5.5.6	Emptiness Check for Tables and Table Variables	143
5.5.7	Getting the Number of Records for Tables and Table Variables	144
<b>5.6</b>	<b>Cursors</b>	144
5.6.1	Example Cursor	145
5.6.2	Looping over Cursors	146
<b>5.7</b>	<b>Transactional Statements</b>	147
<b>5.8</b>	<b>Dynamic SQL</b>	148
<b>5.9</b>	<b>Exception Handling</b>	149
5.9.1	Continue after Handling	149
5.9.2	Block Parallel Execution	150
<b>5.10</b>	<b>Arrays</b>	150
5.10.1	Creating an Array	150
5.10.2	Accessing the Array	151
5.10.3	Concatenating Arrays	151
5.10.4	Converting a Table into an Array	151
5.10.5	Unpacking an Array into a Table	152

5.10.6	Deleting or Trimming an Array .....	152
5.10.7	Cardinality in Arrays .....	153
<b>5.11</b>	<b>SQL Injection Prevention Functions .....</b>	<b>153</b>
<b>5.12</b>	<b>Explicit Parallel Execution .....</b>	<b>154</b>
<b>5.13</b>	<b>System Variables .....</b>	<b>155</b>
<b>5.14</b>	<b>Debugging SQLScript .....</b>	<b>156</b>
<b>5.15</b>	<b>Summary .....</b>	<b>158</b>
<b>6</b>	<b>ABAP-Managed Database Procedures .....</b>	<b>159</b>
<b>6.1</b>	<b>Introduction .....</b>	<b>159</b>
6.1.1	ABAP-Managed Database Procedure Framework .....	161
6.1.2	Development Environment for AMDP .....	163
<b>6.2</b>	<b>Creating AMDP Classes .....</b>	<b>166</b>
6.2.1	Prerequisites .....	166
6.2.2	Implementing AMDP Methods .....	168
6.2.3	Calling AMDP Methods in Applications .....	170
6.2.4	Using Multiple Selection Criteria .....	174
6.2.5	Feature Support Check Using Global Classes .....	177
<b>6.3</b>	<b>Enhancements .....</b>	<b>180</b>
6.3.1	AMDP BAdI Overview .....	180
6.3.2	AMDP BAdI Implementation .....	185
6.3.3	AMDP BAdI Definition .....	189
6.3.4	AMDP BAdI Calls .....	196
<b>6.4</b>	<b>Exception Handling .....</b>	<b>199</b>
<b>6.5</b>	<b>Debugging .....</b>	<b>202</b>
6.5.1	AMDP Debugging before SAP NetWeaver AS ABAP 7.5 .....	203
6.5.2	AMDP Debugging with SAP NetWeaver AS ABAP 7.5 .....	208
6.5.3	AMDP Dump Analysis Using Transaction ST22 .....	209
<b>6.6</b>	<b>Tools .....</b>	<b>214</b>
6.6.1	AMDP Dependencies for ABAP and SAP HANA .....	214
6.6.2	AMDP Precondition Checks .....	215
6.6.3	AMDP Consistency Check for AMDP Table Functions .....	216
6.6.4	Delete Obsolete AMDPs .....	218
<b>6.7</b>	<b>Summary .....</b>	<b>218</b>

<b>7</b>	<b>Modeling .....</b>	<b>219</b>
<b>7.1</b>	<b>What Is Modeling for SAP HANA? .....</b>	<b>219</b>
7.1.1	Measures and Attributes .....	220
7.1.2	Dimensions .....	220
7.1.3	Fact Tables .....	221
7.1.4	Star Schemas .....	222
7.1.5	Hierarchies .....	223
7.1.6	Semantics .....	224
7.1.7	Joins .....	224
<b>7.2</b>	<b>Information Views .....</b>	<b>229</b>
7.2.1	Features .....	230
7.2.2	Information View Types .....	231
<b>7.3</b>	<b>Calculation Views .....</b>	<b>232</b>
7.3.1	Creating Dimension Calculation Views .....	233
7.3.2	Creating Calculated Attributes .....	235
7.3.3	Time Dimension-Based Calculated Views .....	237
7.3.4	Base Tables Aliases .....	238
7.3.5	Hidden Columns .....	240
7.3.6	Labels and Hidden Attributes .....	241
7.3.7	Using Measures in Calculation Views .....	241
<b>7.4</b>	<b>Modeling Functions .....</b>	<b>243</b>
7.4.1	Using a Hierarchy .....	243
7.4.2	Restricted Columns .....	250
7.4.3	Calculated Columns .....	252
7.4.4	Filtering Data .....	256
7.4.5	Using Variables and Input Parameters .....	258
7.4.6	Currency Conversion .....	263
7.4.7	Decision Tables .....	265
<b>7.5</b>	<b>SQL in Information Models .....</b>	<b>270</b>
7.5.1	Query a Modeled Hierarchy Using SQLScript .....	271
7.5.2	Creating and Using Functions in Information Views .....	271
7.5.3	Procedures in Calculation Views .....	272
<b>7.6</b>	<b>Virtual Data Models .....</b>	<b>275</b>
7.6.1	SAP HANA Live .....	275
7.6.2	SAP S/4HANA Embedded Analytics and Core Data Services Views .....	280
<b>7.7</b>	<b>Optimizing Data Models .....</b>	<b>284</b>
7.7.1	Tools to Check Model Performance .....	284
7.7.2	Good Modeling Practices .....	286
<b>7.8</b>	<b>Summary .....</b>	<b>290</b>

<b>8</b>	<b>Core Data Services</b>	291
<b>8.1</b>	<b>The Evolution of Core Data Services</b>	291
<b>8.2</b>	<b>Introduction</b>	295
8.2.1	Domain-Specific Languages	295
8.2.2	Availability and Features	296
8.2.3	Integration with ABAP	297
<b>8.3</b>	<b>Defining CDS Views</b>	298
8.3.1	Using Templates	300
8.3.2	Structure Definition	304
8.3.3	Creating CDS Views in ADT	307
8.3.4	Syntax and Naming Guidelines	310
<b>8.4</b>	<b>Built-In Functions and Expressions</b>	312
8.4.1	Numeric Built-In Functions	312
8.4.2	String Functions	315
8.4.3	COALESCE Function	321
8.4.4	Conversion Functions	323
8.4.5	Date and Time Functions	334
<b>8.5</b>	<b>Annotations</b>	349
8.5.1	Scope of Annotations	349
8.5.2	Types of Annotations	351
<b>8.6</b>	<b>Associations</b>	351
8.6.1	Defining Associations	352
8.6.2	Cardinality	354
8.6.3	Types of Associations	355
<b>8.7</b>	<b>Consuming CDS Views</b>	358
8.7.1	Using ABAP SQL	359
8.7.2	Using SAP List Viewer with Integrated Data Access	360
<b>8.8</b>	<b>Enhancements</b>	364
8.8.1	CDS View Extensions	367
8.8.2	Limiting CDS View Extensibility	368
8.8.3	Finding CDS Extensions to Enhance CDS View Definitions	369
<b>8.9</b>	<b>Authorization Concept</b>	370
8.9.1	Data Control Language Overview	371
8.9.2	Data Control Language Syntax and Access Conditions	372
8.9.3	Data Control Language Source Definition	379
8.9.4	Data Control Language Source Annotations	384
<b>8.10</b>	<b>CDS Table Functions</b>	386
8.10.1	AMDP Functions	387

8.10.2	Introduction to CDS Table Functions	389
8.10.3	Defining CDS Table Functions	391
<b>8.11</b>	<b>Summary</b>	399
<b>9</b>	<b>Open SQL and ABAP SQL</b>	401
<b>9.1</b>	<b>What Is Open SQL?</b>	401
<b>9.2</b>	<b>Features of Open SQL</b>	403
<b>9.3</b>	<b>ABAP SQL</b>	410
9.3.1	Code Pushdown in ABAP SQL	412
9.3.2	Joins	423
<b>9.4</b>	<b>Open SQL versus ABAP SQL Statements</b>	430
9.4.1	Major Syntax Changes in the SELECT Query	430
9.4.2	Arithmetic Operations and String Operations	431
9.4.3	Joins and Unions	431
<b>9.5</b>	<b>ABAP SQL versus CDS Views and AMDPs</b>	432
9.5.1	CDS Views	433
9.5.2	ABAP-Managed Database Procedures	433
<b>9.6</b>	<b>Summary</b>	434
<b>10</b>	<b>Business Object Processing Framework</b>	435
<b>10.1</b>	<b>Introduction to Business Object Processing Framework</b>	435
<b>10.2</b>	<b>Elements of BOPF</b>	438
10.2.1	Nodes	439
10.2.2	Attributes	441
10.2.3	Alternate Keys	442
10.2.4	Actions	442
10.2.5	Determinations	445
10.2.6	Associations	447
10.2.7	Validations	448
10.2.8	Queries	448
<b>10.3</b>	<b>BOPF API</b>	449
10.3.1	BOPF Interface Objects	449
10.3.2	Business Object Key	451
10.3.3	BOPF Constants Interface	451

<b>10.4</b>	<b>CRUD Operations Using the BOPF API</b>	453
10.4.1	Creating Business Object Instances	453
10.4.2	Searching for Business Object Instances	456
10.4.3	Updating and Deleting Business Object Instances	457
<b>10.5</b>	<b>Advanced BOPF API Features</b>	458
10.5.1	Consistency Checks	458
10.5.2	Triggering Actions	459
10.5.3	Action Validations	460
10.5.4	Transaction Management	461
<b>10.6</b>	<b>Enhancement Techniques</b>	462
10.6.1	Enhancement Workbench	463
10.6.2	Enhancing the Business Object Data Model	464
<b>10.7</b>	<b>Summary</b>	469
 <b>11 Performance and Optimization</b>		471
<b>11.1</b>	<b>Runtime Statistics Records</b>	472
<b>11.2</b>	<b>Runtime and Statistical Analysis</b>	475
11.2.1	Transaction SAT	476
11.2.2	ABAP Profiling Perspective	484
11.2.3	SQL Performance Trace	496
11.2.4	Single Transaction Analysis (Transaction ST12)	498
11.2.5	Explain Plan	499
11.2.6	SAP HANA Plan Visualizer	505
<b>11.3</b>	<b>ABAP Code Analysis</b>	509
11.3.1	SAP Global Check Variants	509
11.3.2	Using the ABAP Test Cockpit	510
11.3.3	ABAP Test Cockpit Administration	516
<b>11.4</b>	<b>System-Wide Analysis</b>	521
11.4.1	Database Administrator Cockpit	522
11.4.2	SQL Monitor	524
<b>11.5</b>	<b>SQL Performance Optimization</b>	526
11.5.1	Guided Performance Analysis	527
11.5.2	SQL Performance Tuning Worklist	528
11.5.3	Recommended Static Checks	532
<b>11.6</b>	<b>Summary</b>	533

 <b>12 SAP Business Technology Platform, ABAP Environment</b>		535
<b>12.1</b>	<b>Introduction to Application Development in the Cloud with SAP</b>	535
<b>12.2</b>	<b>ABAP Environment in the Cloud</b>	540
12.2.1	Architecture	541
12.2.2	Creating an ABAP Trial Instance	542
12.2.3	Connecting SAP HANA Studio to SAP BTP, Cloud Foundry Environment	546
12.2.4	ABAP Restrictions in the Cloud	549
12.2.5	Migrating ABAP Applications to the Cloud	552
<b>12.3</b>	<b>Creating ABAP Repository Objects in the Cloud</b>	555
12.3.1	Creating ABAP Packages in the Cloud	556
12.3.2	Creating ABAP Database Tables in the Cloud	560
12.3.3	Creating ABAP Classes in the Cloud	564
12.3.4	Creating ABAP CDS Views in the Cloud	569
<b>12.4</b>	<b>Summary</b>	572
 <b>Appendices</b>		575
<b>A</b>	<b>Programming Guidelines</b>	577
<b>B</b>	<b>Code Migration</b>	589
<b>C</b>	<b>The Authors</b>	637
 Index		639

# Index

## A

ABAP	36
classes (cloud)	564
cloud restrictions	549
core data services view	283, 569
database table (cloud)	560
database tables	560
migrating applications	552
profile configuration	486
profiler trace	487
profiling perspective	484
readiness	601
repository objects	555
runtime measurements	485
SQL statements	430
trace request	489
trial	542
ABAP annotations	351
ABAP code analysis	509
ABAP database connectivity	103, 109, 579
API classes	110
example	114
precautions	120
secondary database	118
ABAP database connectivity class	116
ABAP development	123
ABAP Development Tools	40, 55, 88, 163, 514
install Eclipse components	94
install package	96
links	96
platform supportability	91
readiness checks	92
software supportability	91
ABAP Dictionary	309, 578
core data service views	427
limitations	293
ABAP dictionary	116
ABAP Open SQL	
advantages	412
joins	424
multiple joins	425
runtime	411
ABAP package	556
ABAP perspective	72
ABAP profiler, debugger	492
ABAP runtime analysis	476
ABAP SQL	43, 401
joins	431
unions	431
ABAP Test Cockpit	44, 55, 509, 510, 514, 589
administration	516, 517
checks	512
cloud readiness	553
local check	617
remote check run	611
static check	532
statistics	511
ABAP trace	491
ABAP trace parameters	491
ABAP-managed database procedure	587
dump analysis	209
execution failure	210
ABAP-managed database procedures	37, 45, 159, 433
benefits	163
best practices	213
business add-in call	196
business add-in definition	184, 189
business add-in framework	180
business add-in implementation	185
business add-in method	184
business add-in prerequisites	181
business add-ins	180
call methods	170
class definition	168, 182
consistency check	216
create business add-in	
implementation	187
create classes	166
database-specific features	177
debug	202
debug process	205
debug users	203
debugging with SAP NetWeaver AS	
ABAP 7.5	208
delete obsolete AMDP	218
dependencies	214
development environment	163
enhancements	180
exception classes	199
exception handling	199, 200
extend method	184, 186
fallback class	195
form-based editor	164



- ABAP-managed database procedures (Cont.)
    - framework* ..... 161
    - framework definition* ..... 162
    - global classes* ..... 177
    - method definition* ..... 194
    - method implementation* ..... 183
    - methods* ..... 168, 169
    - precondition checks* ..... 215
    - prerequisites* ..... 166
    - tools* ..... 214
  - Action validations ..... 460
  - Ad-hoc associations ..... 355
  - Advanced Open SQL
    - code pushdown* ..... 412
    - comma-separated field lists* ..... 412
    - compute columns* ..... 416
    - escaping host variables* ..... 412
    - sample code* ..... 415
    - SQL expressions* ..... 418
  - Alternate keys ..... 442
  - Analytic privileges ..... 273
  - Analytic views ..... 255
  - Architecture ..... 27
    - index server* ..... 28
  - Array ..... 150
    - access* ..... 151
    - cardinality* ..... 153
    - concatenating* ..... 151
    - create* ..... 150
    - delete* ..... 152
    - unpack* ..... 152
  - Array trim ..... 152
  - Associations ..... 447
    - cardinality* ..... 354
    - define* ..... 352
- B**
- Block parallel execution ..... 150
  - BOPF
    - API* ..... 449
    - consumer layer* ..... 437
    - elements* ..... 438
    - enhancement techniques* ..... 462
    - interface objects* ..... 449
    - nodes* ..... 439
    - persistence layer* ..... 438
    - runtime layer* ..... 438
    - transaction layer* ..... 437
    - trigger actions* ..... 459
  - BOPF → Business Object Processing
    - Framework* ..... 436
  - Business object ..... 436
    - enhancement* ..... 464
  - Business object instance ..... 453
    - delete* ..... 457
    - update* ..... 457
  - Business object key ..... 451
  - Business Object Processing
    - Framework* ..... 435–437
  - Business rules ..... 435
  - Business transaction events ..... 180
- C**
- Calculated column ..... 252
    - aggregation* ..... 254
    - persistence* ..... 254
  - Calculation view ..... 232
    - attributes* ..... 241
    - base table alias* ..... 238
    - calculated attribute* ..... 235
    - cube calculation view* ..... 233
    - default view* ..... 233
    - dimension view* ..... 233
    - hidden column* ..... 240
    - labels* ..... 241
    - measures* ..... 241
    - parameters* ..... 258
    - procedures* ..... 272
    - time dimension-based calculation view* ..... 237
  - Central check system ..... 609
  - Check results ..... 520
  - Check run ..... 519
    - frequency* ..... 519
  - Cloud Foundry ..... 548
  - Coalesce ..... 321
  - Code Inspector ..... 509
    - static check* ..... 532
  - Code migration ..... 589
  - Code pushdown matrix ..... 48
  - Code pushdown techniques ..... 586
  - Code-to-data
    - performance benefits* ..... 46
  - Code-to-data paradigm ..... 35, 160
    - bottom-up approach* ..... 38
    - top-down approach* ..... 41
  - Column pruning ..... 288
  - Component annotations ..... 351
  - Conditional statements ..... 140
  - Consistency check ..... 458
  - Consistency validation ..... 466
  - Constants interface ..... 451

- Construct ..... 139
  - Conversion functions ..... 323
  - Core data services ..... 37, 45, 161, 291, 433
    - ABAP integration* ..... 297
    - ABAP SQL* ..... 359
    - annotation types* ..... 349
    - annotations* ..... 349
    - association types* ..... 355
    - associations* ..... 351
    - authorizations* ..... 370
    - built-in functions* ..... 312
    - consume views* ..... 358
    - create views* ..... 307
    - define* ..... 298
    - define table functions* ..... 391
    - enhancements* ..... 364
    - expressions* ..... 312
    - limit view extensibility* ..... 368
    - naming guidelines* ..... 310
    - structure definition* ..... 304
    - syntax* ..... 310
    - table function definition* ..... 390
    - table functions* ..... 387
    - view entity* ..... 307
    - view extensions* ..... 367
  - CPU innovations ..... 21
  - Create, read, update, and delete ..... 453
  - CRUD ..... 453
    - save operation* ..... 456
  - CRUD → Create, read, update, and delete
  - Currency conversion ..... 263, 323
    - SAP HANA* ..... 264
  - Cursor ..... 144
    - example* ..... 145
    - loop* ..... 146
  - Cursor technique ..... 105
  - Custom code
    - analysis* ..... 604
    - evaluation* ..... 605
    - functional modifications* ..... 595
    - remote check* ..... 608
    - SAP S/4HANA readiness* ..... 598
    - SQL performance optimization* ..... 596
  - Custom code adaption ..... 617
  - Custom code migration ..... 589, 616
  - Custom query ..... 469
- D**
- Data and time functions ..... 334
  - Data compression ..... 23
  - Data control language .. 125, 296, 371, 401, 577
    - access conditions* ..... 372
    - annotation values* ..... 385
    - create new source* ..... 380
    - source annotation* ..... 384
    - source definition* ..... 379
    - syntax* ..... 372
  - Data definition language ..... 103, 125, 295, 296, 401, 577
    - statement* ..... 115
  - Data definition, create ..... 299
  - Data engine functions ..... 28
  - Data manipulation language ..... 103, 125, 401, 577
  - Data mart ..... 31
  - Data model
    - attribute* ..... 220
    - dimension* ..... 220
    - fact table* ..... 221
    - hierarchy* ..... 223
    - measure* ..... 220
    - optimizing* ..... 284
    - semantics* ..... 224
    - star schema* ..... 222
  - Data modeling ..... 219
    - best practices* ..... 286
  - Data modification language
    - statement* ..... 116
  - Data partitioning ..... 25
  - Data types ..... 108
  - Database administrator cockpit ..... 522
  - Database procedure ..... 106
  - Database schema ..... 128
  - Data-to-code approach ..... 160
  - Date functions ..... 335
  - Debug query mode ..... 285
  - Decimal shift ..... 329
  - Decision table ..... 265
    - create* ..... 266
    - name* ..... 267
    - sample* ..... 266
    - SQLScript* ..... 270
  - Design-time object ..... 231
  - Determinations ..... 445, 465
  - Development environments ..... 51
  - Development tools ..... 51
  - Direct Extractor Connect ..... 82
  - Domain ..... 257
  - Domain fix values ..... 258
  - Domain-specific languages ..... 295

**E**

Embedded analytics .....	280
Emptiness check .....	143
Enhanced SQL expressions .....	420
Enhancement workbench .....	463
Exception handling .....	149
Explain plan .....	499
Exposed associations .....	357
Extract, load, and transform .....	81

**F**

Filtering data .....	256
----------------------	-----

**G**

GitHub .....	87
Guided performance analysis .....	527

**H**

Hierarchy .....	243
<i>level</i> .....	243
<i>level hierarchy creation</i> .....	244
<i>node styles</i> .....	245
<i>parent-child</i> .....	243
<i>parent-child creation</i> .....	246
<i>time-based</i> .....	249
Host variables .....	105

**I**

Information models with SQL .....	270
Information views .....	219, 229
<i>functions</i> .....	271
<i>types</i> .....	231
Infrastructure-as-a-service .....	536
In-memory database .....	20
Internet of Things .....	540

**J**

Java Database Connectivity .....	87
Joins .....	224
<i>inner join</i> .....	224
<i>left outer join</i> .....	225
<i>referential join</i> .....	227
<i>right outer join</i> .....	226
<i>star join</i> .....	228
<i>temporal join</i> .....	228
<i>text join</i> .....	227

**L**

Lightweight Directory Access Protocol .....	88
Literals .....	105
Local scalar variable .....	139
Local table variable .....	139

**M**

Measurement restriction .....	477, 480
Microsoft Excel .....	75
Modeling functions .....	243

**N**

Name server .....	29
Native SQL .....	39, 103, 401, 434, 579
<i>data type compatibility</i> .....	108
<i>executing statements</i> .....	104
<i>interface</i> .....	104
<i>secondary connections</i> .....	118
Node actions .....	442
Node attribute .....	441

**O**

Open Database Connectivity .....	87
Open SQL .....	401, 430, 579
<i>arithmetic operations</i> .....	431
<i>best practices</i> .....	404
<i>database guidelines</i> .....	580
<i>enhanced</i> .....	410
<i>features</i> .....	403
<i>joins</i> .....	423
<i>limitations</i> .....	292
<i>string operations</i> .....	431
<i>usage</i> .....	580

**P**

Parallel execution .....	154
Parameters .....	258
Partitioning .....	289
Perforce .....	87
Performance analysis mode .....	284
Performance optimization .....	472, 580
Persistence layer .....	28
Persistent node .....	441
Plan analysis .....	506
Platform-as-a-service .....	536
Preprocessor server .....	29

**Procedure**

<i>call</i> .....	132
<i>create</i> .....	131
<i>delete</i> .....	133
<i>language</i> .....	132
<i>parameters</i> .....	131
Procedure alter .....	133
Procedure call .....	274
Procedure schema .....	132
Programming guidelines .....	577
Proxy objects .....	40
Pseudo comments .....	632
Python .....	87

**Q**

Queries .....	448
Query language .....	295, 296
Query unfolding .....	287

**R**

Readiness check 2.0 .....	607
Restricted columns .....	250
Root node .....	455
Row-based storage .....	23
Run series parameters .....	518
Runtime analysis .....	475, 479, 481
<i>overview</i> .....	483
<i>variant</i> .....	480
Runtime object .....	231
Runtime statistics records .....	472

**S**

SAP Adaptive Server Enterprise .....	81
SAP Ariba .....	33, 536
SAP Business Explorer .....	219
SAP Business One, version for SAP HANA .....	32
SAP Business Suite powered by	
SAP HANA .....	32
SAP Business Suite powered by	
SAP S/4HANA .....	32
SAP Business Technology Platform .....	535, 537
<i>ABAP environment</i> .....	541
<i>architecture</i> .....	541
<i>Cloud Foundry environment</i> .....	547
<i>cockpit</i> .....	540
<i>trial</i> .....	538
SAP Business Technology Platform,	
ABAP environment .....	99
<i>setup</i> .....	100

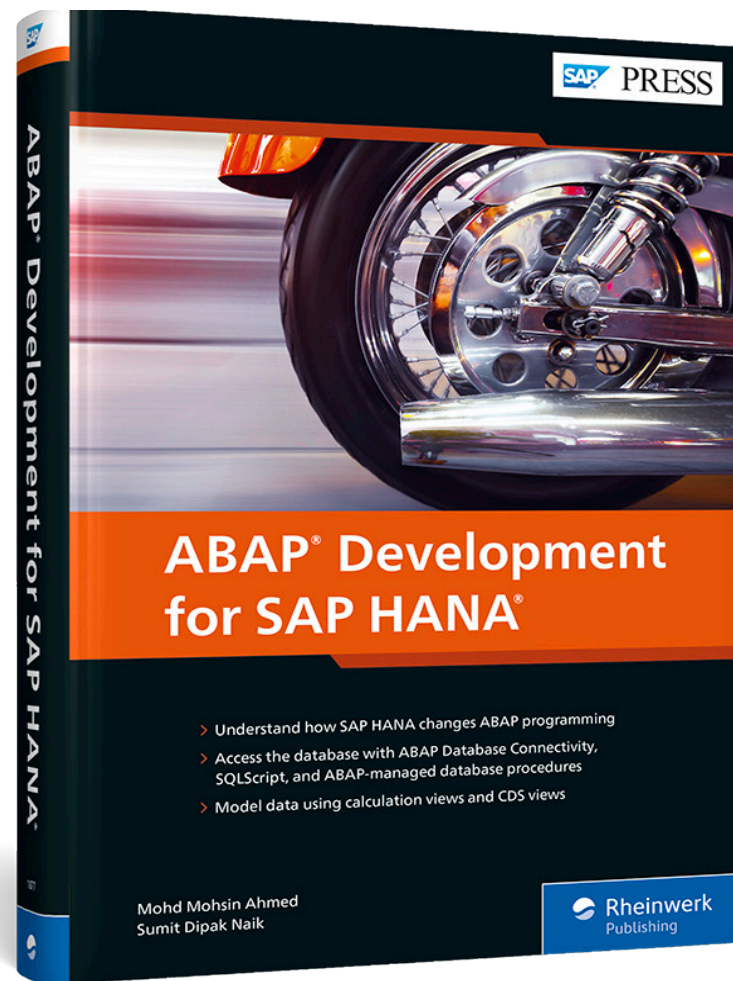
SAP Business Warehouse .....	82
SAP BusinessObjects Business Intelligence ..	75
SAP BusinessObjects Web Intelligence .....	219
SAP BW powered by SAP BW/4HANA .....	32
SAP BW powered by SAP HANA .....	32
SAP BW/4HANA .....	277
SAP Cash Management .....	33
SAP Cloud Platform .....	537, 594
SAP Concur .....	536
SAP Crystal Reports .....	219, 280
SAP Customer Experience .....	536
SAP Data Services .....	82
SAP Fieldglass .....	536
SAP global check variants .....	509
SAP HANA .....	19, 51, 589
<i>code migration</i> .....	592
<i>compression</i> .....	25
<i>features</i> .....	19
<i>repository</i> .....	28
<i>services</i> .....	30
<i>storage</i> .....	22
SAP HANA accelerators .....	31
SAP HANA Application Function Library .....	75
SAP HANA client .....	87
<i>compatibility</i> .....	87
SAP HANA Cloud .....	33
SAP HANA cockpit .....	86
SAP HANA cockpit 2.0 .....	86
SAP HANA extended application services ..	29
<i>platform</i> .....	51
SAP HANA Live .....	275, 277
<i>customization</i> .....	280
<i>deployment</i> .....	276
<i>private views</i> .....	279
<i>query views</i> .....	278
<i>reuse views</i> .....	279
<i>value help views</i> .....	279
SAP HANA Live Browser .....	279
SAP HANA Live Rapid Development .....	280
SAP HANA Live virtual data model .....	277
SAP HANA plan visualizer .....	505, 506
<i>analysis</i> .....	508
<i>execute</i> .....	507
SAP HANA platform edition .....	62
SAP HANA Predictive Analytics Library .....	28
SAP HANA Studio .....	54, 219, 546
<i>ABAP perspective</i> .....	623
<i>ABAP project</i> .....	75
<i>actions</i> .....	67
<i>administration console</i> .....	83
<i>catalog</i> .....	77
<i>check database version</i> .....	57

SAP HANA Studio (Cont.)	
<i>compatibility</i>	56
<i>compatibility issues</i>	59
<i>content</i>	79
<i>custom perspective</i>	69
<i>download</i>	60
<i>enhancement spot</i>	190
<i>installation</i>	64
<i>modeler perspective</i>	75
<i>navigation</i>	67
<i>perspective</i>	67
<i>perspectives</i>	55
<i>provisioning</i>	81
<i>security</i>	80
<i>views</i>	71
SAP HANA Business Function Library	28
SAP Landscape Transformation	
Replication Server	82
SAP List Viewer	360
SAP Lumira	219
SAP NetWeaver	88
SAP NetWeaver Application Server	
for ABAP	37, 401
SAP Note 1935918	595
SAP Note 2339297	473
SAP Note 2375176	58
SAP Note 2436955	498
SAP Precision Marketing	33
SAP product evolution	53
SAP S/4HANA	
<i>code migration</i>	591
<i>readiness check</i>	600
SAP S/4HANA Embedded Analytics	280
SAP Smart Meter Analytics	33
SAP Software Download Center	61
SAP SuccessFactors	33, 536
SAP Lumira	75
Scalar function	272
Scheduled check run	517
Selection criteria e	174
Service manager interface	449
Session-dependent function	257
Single transaction analysis	498
Software-as-a-service	536
SourceForge	87
Special type conversion function	331
SQL	124
<i>dynamic</i>	148
<i>performance optimization</i>	526
<i>view definition</i>	307
SQL console	505
SQL injection prevention function	153
SQL modeler	286
SQL monitor	524
SQL performance trace	496
SQL performance tuning worklist	529
<i>execution</i>	530
SQL performance tuning worklist	528
SQL query	
<i>execution</i>	111
<i>template</i>	128
SQL security mode	132
SQL statement, methods	117
SQL trace	500
SQLScript	28, 123
<i>debug</i>	156
<i>loop</i>	141
<i>operators</i>	141
<i>syntax error</i>	165
<i>system variables</i>	155
<i>table</i>	129
Statistical records	474
<i>analysis</i>	475
Statistical server	29
Stored procedure	131
String functions	315
Structured Query Language (SQL)	103
Subquery	137
System-wide analysis	521

## T

Table	
<i>array conversion</i>	151
<i>buffering</i>	585
<i>field</i>	129
<i>inserting data</i>	130
<i>select query</i>	130
<i>type</i>	130
TCUR schema	264
Templates	300
Time function	338
Time zone functions	343
Timestamp functions	339
Transaction	
ATC	517
DB59	523
DBACOCKPIT	78, 107, 119, 523
DBCO	107, 119
LC10	523
SAT	476
SE11	291
SE38	89
SE80	53, 511

Transaction (Cont.)	
SICK	216
STO4	500
STO5	497, 500
ST22	199, 212
STAD	473
SWLT	528
SYCM	600
Transaction manager	461
<i>interface</i>	450
Transactional statement	147
Transient node	441
Transparent optimization	42
User-defined function (Cont.)	
<i>join</i>	135
<i>scalar</i>	134
<i>table</i>	134
<b>V</b>	
Validations	448
Variables	258
Variant configuration	
<i>duration</i>	478
<i>program parts</i>	479
<i>statements</i>	479
Virtual data model	275
<b>W</b>	
Web Dynpro	55
WHERE clause	256
Whitelisted objects	602
Unit conversion	326
User-defined function	133
<i>alter function</i>	134
<i>drop function</i>	135



**Mohd Mohsin Ahmed** is an SAP HANA certified professional with more than 14 years of experience as an SAP technical consultant. He currently specializes in implementing SAP S/4HANA solutions with a variety of methodologies, across various industry sectors, including manufacturing, retail, food and beverage, life science, energy and utilities, public sector, and IT. He holds a degree in computer science engineering from Jawaharlal Nehru Technological University in Hyderabad, India.



**Sumit Dipak Naik** is an experienced ABAP and SAP HANA certified professional. He has more than 16 years of technical consulting, solutioning, and management experience. His focus is in implementing SAP ERP and SAP S/4HANA solutions, across various industry sectors, including telecommunications, food and beverages, manufacturing, retail, life sciences, energy and utilities, public sector, and IT. He has extensive experience with different implementation methodologies, approaches, and accelerators.

Mohd Mohsin Ahmed, Sumit Dipak Naik

## ABAP Development for SAP HANA

643 Pages, 2021, \$89.95

ISBN 978-1-4932-1877-6

 [www.sap-press.com/4954](http://www.sap-press.com/4954)

*We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.*