BC414

Programming Database Updates

SAP NetWeaver

Date _____

Training Center _____

Instructors _____

_____

Education Website _____

Participant Handbook

Course Version: 62
Course Duration: 2 Day(s)
Material Number: 50085749

An SAP course - use it to learn, reference it for work

Internal Use SAP Partner Only

Copyright

Trademarks

Disclaimer

Internal Use SAP Partner Only

About This Handbook

This handbook is intended to complement the instructor-led presentation of this course, and serve as a source of reference. It is not suitable for self-study.

Typographic Conventions

American English is the standard used in this handbook. The following typographic conventions are also used.

Type Style	Description
Example text	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths, and options. Also used for cross-references to other documentation both internal (in this documentation) and external (in other locations, such as SAPNet).
Example text	Emphasized words or phrases in body text, titles of graphics, and tables
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example SELECT and INCLUDE.
`Example text`	Screen output. This includes file and directory names and their paths, messages, names of variables and parameters, and passages of the source text of a program.
`Example text`	Exact user entry. These are words and characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

Icons in Body Text

The following icons are used in this handbook.

Internal Use SAP Partner Only

Icon	Meaning
	For more information, tips, or background
	Note or further explanation of previous point
	Exception or caution
	Procedures
	Indicates that the item is displayed in the instructor's presentation.

Contents

Course Overview

This course provides information about executing change transactions.

Target Audience

This course is intended for the following audiences:

- Experienced ABAP developers
- Consultant
- Project team members

Course Prerequisites

Required Knowledge

- Practical experience of the ABAP development environment (ABAP Workbench)
- Programming User Dialogs (Course BC410)

Recommended Knowledge

- ABAP Dictionary (Course BC430)

Course Goals

This course will prepare you to:

- executing transactions that make changes to the database.

Course Objectives

After completing this course, you will be able to:

- use the Open SQL statements for database updates
- implement the SAP locking concept for database updates
- implement different update techniques for database changes

Internal Use SAP Partner Only

Unit 1

Database Updates with Open SQL

Unit Overview

- Open SQL
- Single Record Operations
- Set Operations

Unit Objectives

After completing this unit, you will be able to:

- using Open SQL commands to execute database changes.

Unit Contents

Internal Use SAP Partner Only

Lesson: Database Updates with Open SQL

Lesson Overview

- Open SQL
- Single Record Operations
- Set Operations

Lesson Objectives

After completing this lesson, you will be able to:

- using Open SQL commands to execute database changes.

Business Example

You want to use Open SQL commands to execute database changes.

Open SQL: Introduction

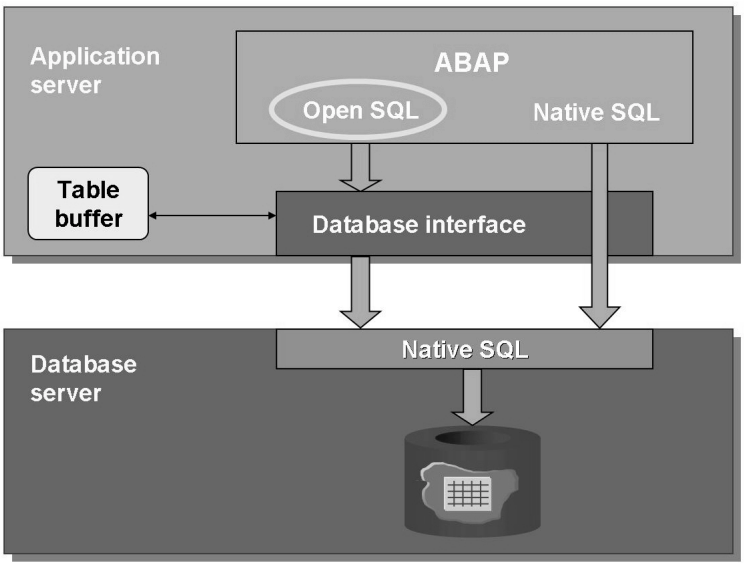


Figure 1: Overview: SQL terms

In ABAP, you have both the **Open SQL** commands as well as the respective database-specific **Native SQL** commands at your disposal for making database changes.

Accessing the database with Native SQL enables you to use database-specific commands. This requires detailed knowledge of the syntax in question. Programs that use Native SQL commands need additional programming after they have

been transported to different system environments (different database systems), since the syntax of the SQL commands generally needs to be adjusted on a database-specific basis.

Open SQL commands are not database-specific; they are automatically converted into the respective SQL statements by the database interface and passed to the database. An ABAP program that operates with Open SQL is therefore not database-specific and you can use it in any R/3 System without having to adjust it.

A further advantage of using Open SQL is that you can buffer SAP tables locally on the application server for quicker read access. This also means that the database load is reduced. The data is read from the buffer automatically after the respective table settings have been made.

The Open SQL set of commands only comprises operations for Data Manipulation Language (DML), not for Data Definition Language (DDL) since these are integrated in the ABAP Dictionary.

You should implement database accesses using Native SQL only if a particular Native SQL function that is not available in Open SQL must be used.

For more information about Native and Open SQL, refer to the ABAP Editor keyword documentation for the term **SQL**.

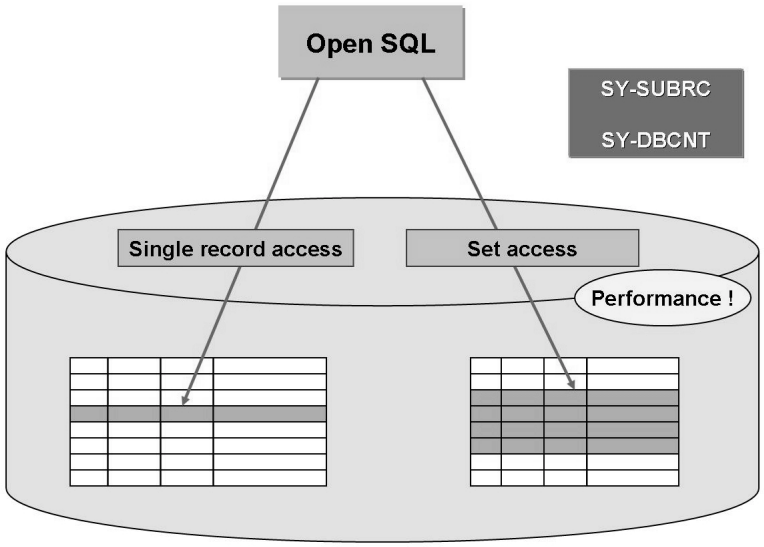


Figure 2: Target Quantity and Return Values

You can limit the target quantity on the database using all the Open SQL commands discussed here.

One or more rows can be processed using an SQL command. Commands that process several lines usually give better performance than corresponding single-set accesses (exception: mass data change using MODIFY).

Also, there is a syntax variant available for the change operation that you can use to change individual fields in a line.

If you have masked field selections (WHERE <field> LIKE “<search_mask>”), note that “_ ” masks an individual character and “**%** ” masks a character string of any length (in line with the SQL standard).

All the Open SQL commands provide you with a return message about the success or failure of the database operation performed. The message is issued in the form of a return code in the system field sy-subrc. Return code “0” (zero) always means that the operation has been completed successfully. All other values mean that errors have occurred. For further details, refer to the keyword documentation for the command in question.

In addition, the sy-dbcnt system field displays the number of records for which the desired database operation was actually carried out.

Note that Open SQL commands do not perform any automatic authorization checks. You need to execute these explicitly in your program (refer to the unit entitled *Authorization Checks*).

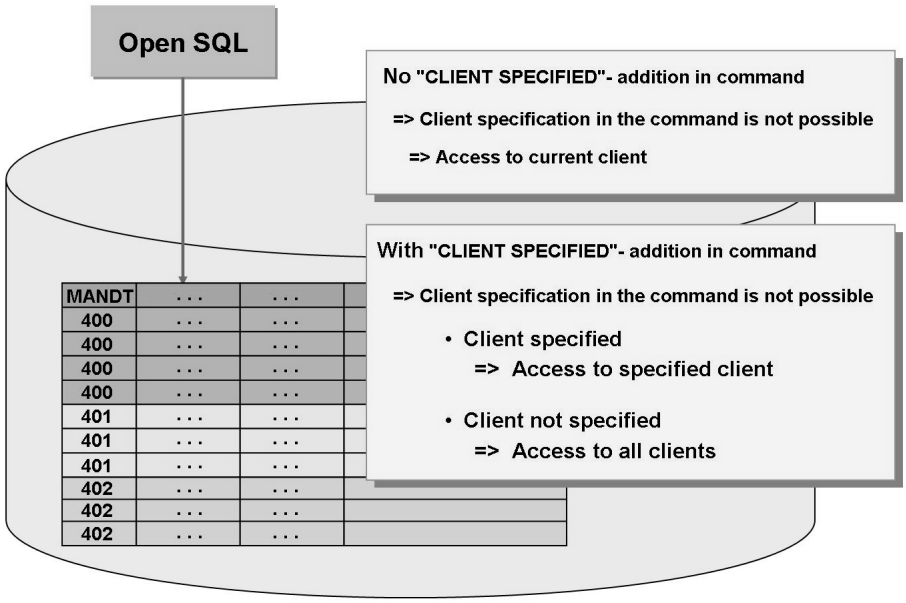


Figure 3: Accessing Client-Specific Tables

If the addition “CLIENT SPECIFIED” is not specified in an Open SQL command, no client specification is allowed in the respective WHERE clause, and the corresponding records of the current execution client are accessed.

If you wish to process data from other clients, you must specify the addition “CLIENT SPECIFIED” in the Open SQL command and the respective client(s) in the appropriate WHERE clause.

Make sure that an Open SQL command accesses all (!) clients if it contains the addition “CLIENT SPECIFIED” without a client specification.

Open SQL: Syntax

```
INSERT INTO <dbtab> [CLIENT SPECIFIED] VALUES <wa>.
```

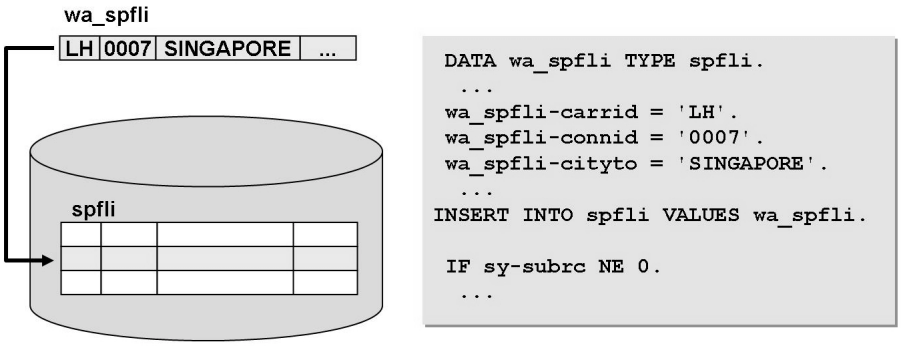


Figure 4: Creating a Single Record

To insert a new row in a database table, enter the command “INSERT INTO <dbtab> VALUES <wa>.”. For this purpose, you must place the line to be inserted in the structure <wa> before the command is called. This structure must be the same as for the lines in the database table concerned.

The client field that may possibly exist in the structure <wa> will only be taken into consideration if the “CLIENT SPECIFIED” addition is specified. If there is no “CLIENT SPECIFIED” addition, the current execution client applies.

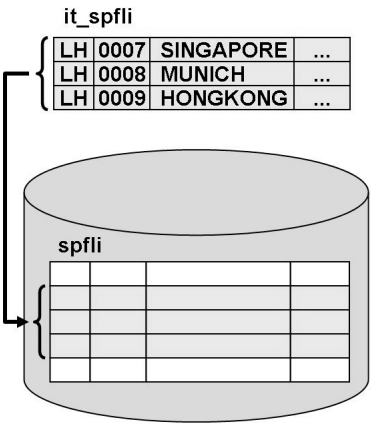
Rows can also be inserted using views. However, the view must already be created in the ABAP Dictionary with the maintenance status “*read and change*” and must only contain fields from a table.

This INSERT variant has the following return codes:

- 0: Line inserted successfully.
- 4: Line could not be inserted because a line with the same key already exists.

Alternative syntax : INSERT <dbtab> [CLIENT SPECIFIED] FROM <wa>.

```
INSERT <dbtab> [CLIENT SPECIFIED] FROM TABLE <itab>.
```




```
DATA:
  it_spfli TYPE STANDARD TABLE OF spfli,
  wa_itab  LIKE LINE OF it_spfli.
   ...
 wa_itab-carrid = 'LH'.
 wa_itab-connid = '0009'.
 wa_itab-cityto = 'HONGKONG'.
  ...
 APPEND wa_itab TO it_spfli.

    .
    .
 APPEND wa_itab TO it_spfli.

 INSERT spfli FROM TABLE it_spfli.

 IF sy-subrc NE 0.
  ...
```

Figure 5: Creating Several Records

You can use the command “INSERT <dbtab> FROM TABLE <itab>.” to create several rows in a database table. The internal table <itab> that you should specify here must have the same line structure as the corresponding databse table and the new data records.

The client field that may possibly exist in the interne Tabelle <itab> will only be taken into consideration if the “CLIENT SPECIFIED” addition is specified. If there is no “CLIENT SPECIFIED” addition, the current execution client applies.

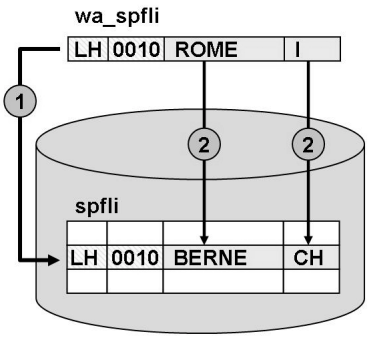
If it is possible to create all the lines, sy-subrc is automatically set to zero.

If, however, even one data record cannot be created, the system triggers a runtime error. This means that the entire insertion operation is discarded (database rollback). If, in such a case, you wish to have records that can be inserted actually inserted, use the command addition “ACCEPTING DUPLICATE KEYS”. This addition has the effect that, if there is an error, the runtime error (and thus also the database rollback) is suppressed, sy-subrc is set to 4, and all the records without errors are inserted.

The sy-dbcnt system field contains the number of rows that were successfully inserted in the database.

```
1  UPDATE <dbtab> [CLIENT SPECIFIED] FROM <wa>.

2  UPDATE <dbtab> [CLIENT SPECIFIED]
                  SET <f1> = <g1>  ...  <fn> = <gn>
                  WHERE <full_qualified_key>.
```




```
  DATA wa_spfli TYPE spfli.
   ...
  wa_spfli-carrid  = 'LH'.
  wa_spfli-connid    = '0010'.
  wa_spfli-cityto    = 'ROME'.
  wa_spfli-countryto = 'I'.
   ...
  UPDATE spfli FROM wa_spfli.
* UPDATE spfli
*    SET  cityto    = wa_spfli-cityto
*         countryto = wa_spfli-countryto
*   WHERE carrid = wa_spfli-carrid
*    AND  connid = wa_spfli-connid.
   IF sy-subrc NE 0.
   ...
```

Figure 6: Changing a Single Record

Using the two variants of the UPDATE command discussed above, you can change a specific line within a database table.

In the case of variant 1, the database record that contains the key in <wa> is overwritten by <wa>. However, the key field “MANDT” that may possibly exist in <wa> is only taken into consideration if the “CLIENT SPECIFIED” addition is specified as well (otherwise the current execution client applies). Logically, <wa> must have the same structure as the database record to be changed.

In the case of variant 2, the record specified in the WHERE clause is changed. However, only the fields specified in the “SET” addition are overwritten on the database side with the values specified. In this syntax version, you must define the record to be changed in the WHERE clause by exactly specifying all the key field evaluations. For details on specification of a possibly existing MANDT field, refer to the slide “Accessing Client-Specific Tables”.

You can specify simple calculation operations as evaulation for numeric database fields in the “SET” addition: f = g , f = f + g , f = f - g .

Rows can also be changed using views. However, the view must already be created in the ABAP Dictionary with the maintenance status “*read and change*” and must only contain fields from a table.

These two UPDATE variants have the following return codes :

- 0: Line was changed.
- 4: Line could not be changed because, for example, the specified key does not exist.

```
UPDATE <dbtab> [CLIENT SPECIFIED]
               SET <f1> = <g1>  ...  <fn> = <gn>
               WHERE <condition>.
```

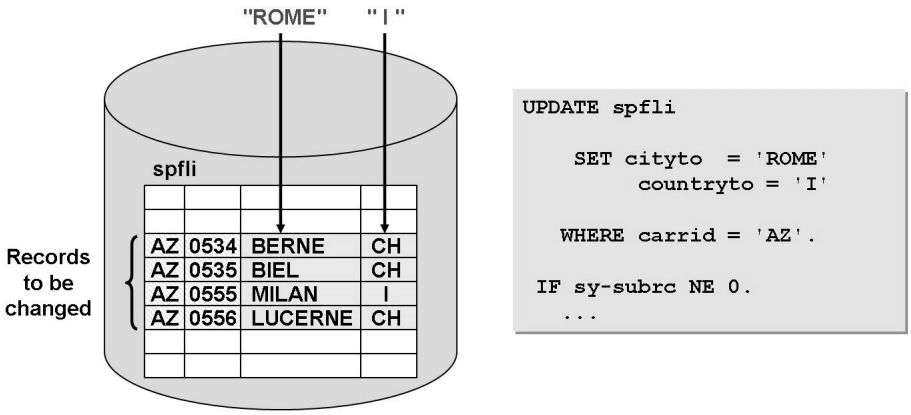


Figure 7: Changing Several Records (Through Condition)

If identical changes are to be made to the same fields in several rows of a database table, use the syntax specified on the slide.

Using the WHERE clause you define which lines are to be changed. For details on specification of a possibly existing MANDT field, refer to the slide “Accessing Client-Specific Tables”.

In the SET addition you specify which fields in these records are to be changed. The following calculations are also possible here for the numeric fields to be changed:

f = g , f = f + g , f = f - g .

This UPDATE variant has the following return codes:

- 0: At least one line has been changed.
- 4: No line was changed because, for example, no such line exists.

The sy-dbcnt field contains the number of updated rows.

```
UPDATE <dbtab> [CLIENT SPECIFIED] FROM TABLE <itab>.
```

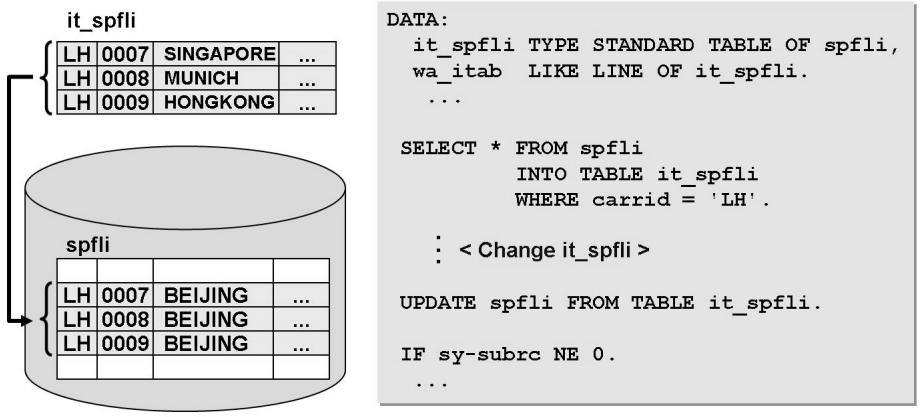


Figure 8: Changing Several Records (Through Internal Table)

You can perform a mass data change by specifying an internal table that has the same structure as the corresponding database table and the records to be changed.

The “MANDT” field that possibly exists in the specified internal table is only taken into consideration if the “CLIENT SPECIFIED” addition is specified (otherwise the current execution client applies).

This UPDATE variant has the following return codes:

- 0: All the specified lines were changed successfully.
- 4: At least one of the specified lines could not be changed (because, for example, it does not exist); the other lines were changed.

The sy-dbcnt system field contains the number of updated rows.

```
MODIFY <dbtab> [CLIENT SPECIFIED] FROM <wa>.
```

```
MODIFY <dbtab> [CLIENT SPECIFIED] FROM TABLE <itab>.
```

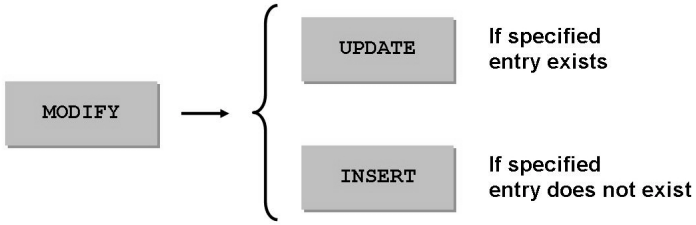


Figure 9: Modifying Single Record / Several Records

The MODIFY command is SAP-specific. It covers the two commands UPDATE and INSERT :

- If the data record specified in the MODIFY statement exists, this record is updated (→ UPDATE).
- If the data record specified in the MODIFY statement does not exist, this record is inserted (→ INSERT).

Using the various syntax variants you can process single records and several records (same as the syntax of UPDATE and INSERT)

The operation can also be carried out on views. However, the view must already be created in the ABAP Dictionary with the maintenance status "*read and change*" and must only contain fields from a table.

This command has the following return codes:

- 0: Specified record or all specified records were processed (updated/inserted).
- 4: The specified record or at least one of the specified records could not be processed - for example, because the record does not exist in the database and inserting it would destroy a **unique** secondary index. If you have chosen mass data change, the other records were processed.

The sy-dbcnt field contains the number of processed rows.

```
DELETE FROM <dbtab> [CLIENT SPECIFIED]
                    WHERE <full_qualified_key>.
```

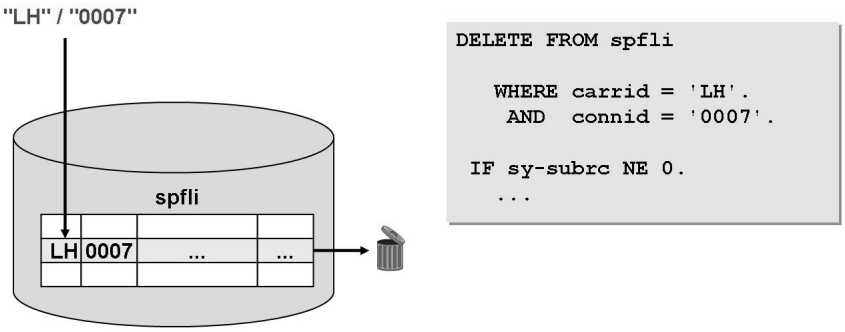


Figure 10: Deleting a Single Record

The syntax specified above for the DELETE command enables you to delete a single row in a database table. In this syntax version, you must define the record to be changed in the WHERE clause by exactly specifying all the key field evaluations. For details on specification of a possibly existing MANDT field, refer to the slide “Accessing Client-Specific Tables”.

A row can also be deleted from views. However, the view must already be created in the ABAP Dictionary with the maintenance status “*read and change*” and must only contain fields from a table.

This DELETE variant has the following return codes:

- 0: Line was deleted.
- 4: Line could not be deleted because, for example, it does not exist in the database.

Alternative syntax : DELETE <dbtab> [CLIENT SPECIFIED] FROM <wa>. With this syntax version, the structure <wa> must have the same structure as the records in the respective database table and must be filled with the key fields of the records to be deleted before the command is called. However, the key field “MANDT”, which possibly exists in <wa>, is only taken into consideration if the “CLIENT SPECIFIED” addition is specified (otherwise the execution client applies).

```
DELETE FROM <dbtab> [CLIENT SPECIFIED]
                   WHERE <condition>.
```

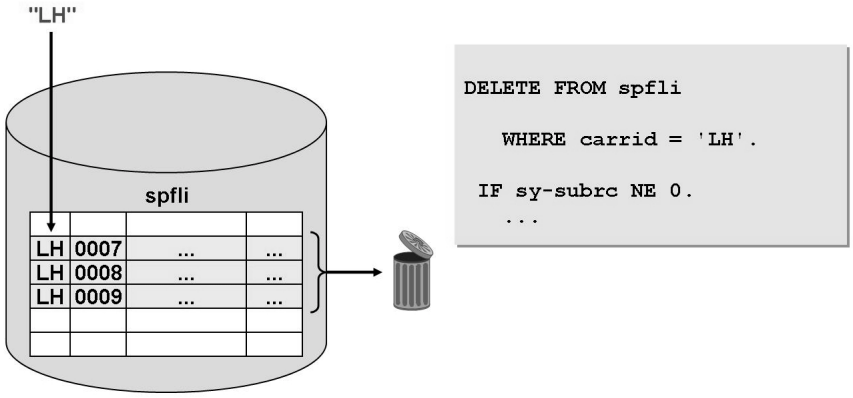


Figure 11: Deleting Several Records (Through Condition)

This syntax variant of the DELETE command enables you to delete several lines in a database table. Here, you can specify the lines that are to be deleted in the WHERE clause. For details on specification of a possibly existing MANDT field, refer to the slide “Accessing Client-Specific Tables”.

You can delete **all the lines** in a **cross-client** database table using the following syntax: DELETE FROM <dbtab> WHERE <field> LIKE “%”. Here <field> is an arbitrary table field.

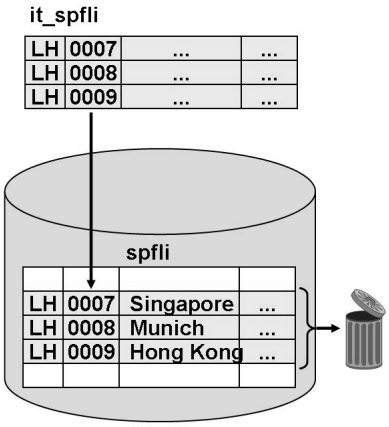
If you wish to delete **all the lines of the execution client** from a **client-specific** database table, you can use the same syntax : DELETE FROM <dbtab> WHERE <field> LIKE “%”. If you wish to delete **all the existing lines** from a **client-specific** database table, you must also specify the “CLIENT SPECIFIED” addition in the command: DELETE FROM <dbtab> CLIENT SPECIFIED WHERE <field> LIKE “%”.

Deleting data records using conditions produces the following return codes:

- 0: At least one line was deleted.
- 4: No line was deleted because, for example, the specified lines do not even exist.

The system field sy-dbcnt contains the number of lines that have been deleted from the database.

```
DELETE <dbtab> [CLIENT SPECIFIED] FROM TABLE <itab>.
```




```
DATA:
  it_spfli TYPE STANDARD TABLE OF spfli,
  wa_itab  LIKE LINE OF it_spfli.
   ...
wa_itab-carrid = 'LH'.
wa_itab-connid = '0007'.

APPEND wa_itab TO it_spfli.
   .
   .
APPEND wa_itab TO it_spfli.

DELETE spfli FROM TABLE it_spfli.

IF sy-subrc NE 0.
 ...
```

Figure 12: Deleting Several Records (Through Internal Table)

If you wish to delete several records from a database table, you can specify them first in an internal table that has the same structure as the respective database table and then use the above syntax of the DELETE command. To do so, all you need to do is specify the key part of the records to be deleted in the internal table. If the key field “MANDT” exists, then it will only be taken into consideration if the “CLIENT SPECIFIED” addition is specified (otherwise the current execution client applies).

This DELETE variant has the following return codes:

- 0: All the lines specified in the internal table were deleted.
- 4: At least one line could not be deleted (for example, because it does not exist); the other records were deleted.

The number of lines deleted from the database is shown in the system field sy-dbcnt.

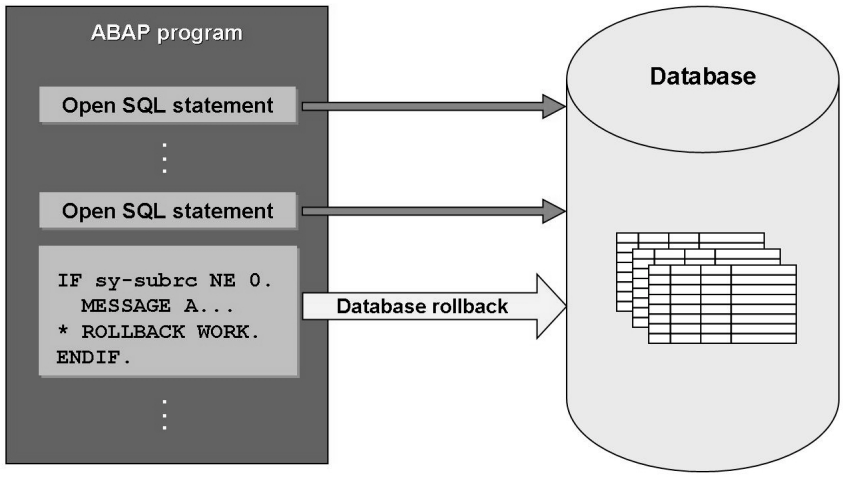


Figure 13: Restoring Previous Database Status

If an Open SQL statement that executes a change to the database returns a return code different than zero, you should make sure that the database is returned to the same status as before you attempted to make the respective change. You achieve this by performing a database rollback, which reverses all changes to the current database LUW (see next unit).

There are two ways of causing a database rollback:

- Sending a termination dialog message (**A-Message**)
- Using the **ROLLBACK WORK** ABAP statement

The transmission of an **A Message** causes a database rollback and terminates the program. All other message types (E,W, I) also involve a dialog but **do not** trigger a database rollback.

The ABAP statement **ROLLBACK WORK**, on the other hand, causes a database rollback without terminating the program. In this case, you should be careful since the **context has not been reset** in the current program.

Navigating in the Exercises

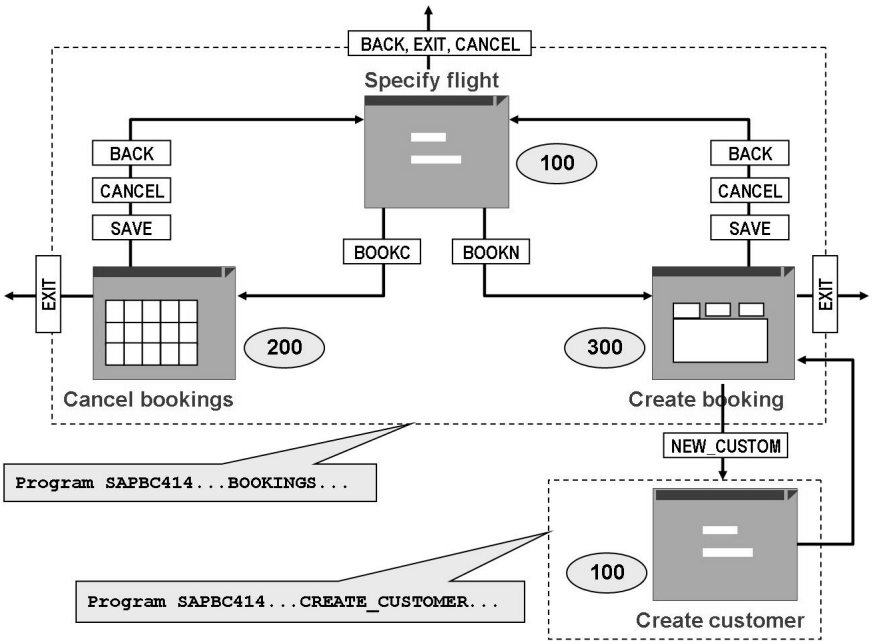


Figure 14: Navigating in the Exercises

Exercise 1: Changing a Single Record

Exercise Objectives

After completing this exercise, you will be able to:

- Insert single records in database tables

Business Example

You want to execute direct database changes from your program.

Task: Changing a Single Record

Program: SAPMZ##_CUSTOMER1

Transaction: Z##_CUSTOMER1

Copy template: SAPBC414**T**_CREATE_CUSTOMER_01

Model Solution: SAPBC414**S**_CREATE_CUSTOMER_01

1. Copy the template **SAPBC414T_CREATE_CUSTOMER_01** with **all** subobjects to **SAPMZ##_CUSTOMER1** (## = group number). Assign transaction code **Z##_CUSTOMER1** to the program.
2. The prgram you copied allows you to enter new customer data using screen 100. Extend the program to include the database dialog:

   After the function code *SAVE* is triggered (for example, by clicking the *Save* icon), the customer data is to be written to the database table SCUSTOM.

   The ABAP statements for the database dialog are to be encapsulated in the subroutine **SAVE_SCUSTOM**. The subroutine has already been created (and is empty).

Continued on next page

Insert the new customer data record in the database table **SCUSTOM**. If the operation is successful, message **S015** should be issued; if the operation is not successful, termination message **A048** should be issued.

Hint: There are SCUSTOM fields on the input screen.

On the internal program side, the newly entered customer data is retained in the SCUSTOM structure defined in the TABLES-statement.

The call already implemented from the NUMBER_GET_NEXT subroutine places a new customer number into the field SCUSTOM-ID so that all the data of the new customer is completely available in the SCUSTOM structure.

The BC414 message class is declared as a program default in the PROGRAM statement and therefore valid throughout the programs.

Solution 1: Changing a Single Record

Task: Changing a Single Record

Program: SAPMZ##_CUSTOMER1

Transaction: Z##_CUSTOMER1

Copy template: SAPBC414**T**_CREATE_CUSTOMER_01

Model Solution: SAPBC414**S**_CREATE_CUSTOMER_01

1. Copy the template **SAPBC414T_CREATE_CUSTOMER_01** with **all** subobjects to **SAPMZ##_CUSTOMER1** (## = group number). Assign transaction code **Z##_CUSTOMER1** to the program.

   a) -

2. The prgram you copied allows you to enter new customer data using screen 100. Extend the program to include the database dialog:

   After the function code *SAVE* is triggered (for example, by clicking the *Save* icon), the customer data is to be written to the database table SCUSTOM.

   The ABAP statements for the database dialog are to be encapsulated in the subroutine **SAVE_SCUSTOM**. The subroutine has already been created (and is empty).

   Insert the new customer data record in the database table **SCUSTOM**. If the operation is successful, message **S015** should be issued; if the operation is not successful, termination message **A048** should be issued.

   Hint: There are SCUSTOM fields on the input screen.

   On the internal program side, the newly entered customer data is retained in the SCUSTOM structure defined in the TABLES-statement.

   The call already implemented from the NUMBER_GET_NEXT subroutine places a new customer number into the field SCUSTOM-ID so that all the data of the new customer is completely available in the SCUSTOM structure.

   The BC414 message class is declared as a program default in the PROGRAM statement and therefore valid throughout the programs.

   a) see the model solution below

Continued on next page

Result

Hint: Tips on Model Solutions for this Course

The model solutions reproduce the statements for screen flow logic and ABAP program parts that you need to include. The exercises for course BC414 are designed to expand on two larger programs accompanying the contents of the unit in question. To make the examples clearer, we do not reproduce all the code necessary for every model solution. The following procedure is used instead:

- The model solution for the activity in which a program is edited for the first time is displayed completely.
- All subsequent model solutions contains only changed and new program parts.
- A complete version of both programs is provided in the appendix in the training folder.

Model Solution SAPBC414S_CREATE_CUSTOMER_01

Module Pool

```
*&---------------------------------------------------------------------*
*& Modulpool         SAPBC414S_CREATE_CUSTOMER_01                    *
*&---------------------------------------------------------------------*
INCLUDE BC414S_CREATE_CUSTOMERTOP.
INCLUDE BC414S_CREATE_CUSTOMERO01.
INCLUDE BC414S_CREATE_CUSTOMERI01.
INCLUDE BC414S_CREATE_CUSTOMER_01F01.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE status_0100.

PROCESS AFTER INPUT.
  MODULE exit AT EXIT-COMMAND.
  MODULE save_ok_code.
  FIELD: scustom-name MODULE mark_changed ON REQUEST.
  MODULE user_command_0100.
```

Continued on next page

Internal Use SAP Partner Only

TOP Include

```
*&---------------------------------------------------------------------*
*& Include BC414S_CREATE_CUSTOMERTOP                                   *
*&---------------------------------------------------------------------*
PROGRAM  sapbc414s_create_customer MESSAGE-ID bc414.

DATA: answer, flag.
DATA: ok_code LIKE sy-ucomm, save_ok LIKE ok_code.
TABLES: scustom.
```

PBO Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERO01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
MODULE STATUS_0100 OUTPUT.
  SET PF-STATUS 'DYN_0100'.
  SET TITLEBAR 'DYN_0100'.
ENDMODULE.                 " STATUS_0100  OUTPUT
```

PAI Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERI01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&---------------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'EXIT'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes on screen 100
        LEAVE PROGRAM.
      ELSE.
```

Continued on next page


```
        PERFORM ask_save USING answer.
        CASE answer.
          WHEN 'J'.
            ok_code = 'SAVE&EXIT'.
          WHEN 'N'.
            LEAVE PROGRAM.
          WHEN 'A'.
            CLEAR ok_code.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
    WHEN 'CANCEL'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes on screen 100
        LEAVE TO SCREEN 0.
      ELSE.
        PERFORM ask_loss USING answer.
        CASE answer.
          WHEN 'J'.
            LEAVE TO SCREEN 0.
          WHEN 'N'.
            CLEAR ok_code.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT
*&---------------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&---------------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
ENDMODULE.                             " SAVE_OK_CODE  INPUT

*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
    WHEN 'SAVE&EXIT'.
      PERFORM save.
      LEAVE PROGRAM.
    WHEN 'SAVE'.
      IF flag IS INITIAL.
```

Continued on next page

```
        SET SCREEN 100.
      ELSE.
        PERFORM save.
        SET SCREEN 0.
      ENDIF.
    WHEN 'BACK'.
      IF flag IS INITIAL.
        SET SCREEN 0.
      ELSE.
        PERFORM ask_save USING answer.
        CASE answer.
          WHEN 'J'.
            PERFORM save.
            SET SCREEN 0.
          WHEN 'N'.
            SET SCREEN 0.
          WHEN 'A'.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&---------------------------------------------------------------------*
*&      Module  MARK_CHANGED  INPUT
*&---------------------------------------------------------------------*
MODULE mark_changed INPUT.
* set flag to mark changes were made on screen 100
  flag = 'X'.
ENDMODULE.                             " MARK_CHANGED  INPUT
```

FORM Routines

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMER_01F01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  NUMBER_GET_NEXT
*&---------------------------------------------------------------------*
*      -->P_WA_SCUSTOM  text
```

Continued on next page


```
*---------------------------------------------------------------------*
FORM number_get_next USING p_scustom LIKE scustom.
  DATA: return TYPE inri-returncode.
* get next free number in the number range '01'
* of number range object 'SBUSPID'
  CALL FUNCTION 'NUMBER_GET_NEXT'
       EXPORTING
            nr_range_nr = '01'
            object      = 'SBUSPID'
       IMPORTING
            number      = p_scustom-id
            returncode  = return
       EXCEPTIONS
            OTHERS      = 1.
  CASE sy-subrc.
    WHEN 0.
      CASE return.
        WHEN 1.
* number of remaining numbers critical
          MESSAGE s070.
        WHEN 2.
* last number
          MESSAGE s071.
        WHEN 3.
* no free number left over
          MESSAGE a072.
      ENDCASE.
    WHEN 1.
* internal error
      MESSAGE a073 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " NUMBER_GET_NEXT

*&---------------------------------------------------------------------*
*&      Form  ASK_SAVE
*&---------------------------------------------------------------------*
*       -->P_ANSWER  text
*---------------------------------------------------------------------*
FORM ask_save USING p_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
       EXPORTING
            textline1 = 'Data has been changed.'(001)
            textline2 = 'Save before leaving transaction?'(002)
            titel     = 'Create Customer'(003)
       IMPORTING
```

Continued on next page


```
            answer    = p_answer.
ENDFORM.                              " ASK_SAVE

*&---------------------------------------------------------------------*
*&      Form  ASK_LOSS
*&---------------------------------------------------------------------*
*      -->P_ANSWER  text
*---------------------------------------------------------------------*
FORM ask_loss USING p_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_LOSS_OF_DATA'
       EXPORTING
            textline1 = 'Continue?'(004)
            titel     = 'Create Customer'(003)
       IMPORTING
            answer    = p_answer.
ENDFORM.                              " ASK_LOSS


*&---------------------------------------------------------------------*
*&      Form  ENQ_SCUSTOM
*&---------------------------------------------------------------------*
FORM enq_scustom.
  CALL FUNCTION 'ENQUEUE_ESCUSTOM'
       EXPORTING
            id             = scustom-id
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e060.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                              " ENQ_SCUSTOM


*&---------------------------------------------------------------------*
*&      Form  DEQ_ALL
*&---------------------------------------------------------------------*
FORM deq_all.
  CALL FUNCTION 'DEQUEUE_ALL'.
ENDFORM.                              " DEQ_ALL
```

Continued on next page


```
*&---------------------------------------------------------------------*
*&      Form  SAVE
*&---------------------------------------------------------------------*
FORM save.
* get SCUSTOM-ID from number range object SBUSPID
  PERFORM number_get_next USING scustom.
* save new customer
  PERFORM save_scustom.
ENDFORM.                               " SAVE


*&---------------------------------------------------------------------*
*&      Form  SAVE_SCUSTOM
*&---------------------------------------------------------------------*
FORM save_scustom.
  INSERT INTO scustom VALUES scustom.
  IF sy-subrc <> 0.
* insertion of dataset in DB-table not possible
    MESSAGE a048.
  ELSE.
* insertion successfull
    MESSAGE s015 WITH scustom-id.
  ENDIF.
ENDFORM.                               " SAVE_SCUSTOM
```


Internal Use SAP Partner Only

Exercise 2: Changing Several Data Records (optional)

Exercise Objectives

After completing this exercise, you will be able to:

- Change several data records, with respect to performance, in database tables.

Business Example

You want to make direct changes to several data records from your program.

Task: Changing Several Data Records (optional)

Program: SAPMZ##_UPDATE_STRAVELAG

Copy template: SAPBC414**T**_UPDATE_STRAVELAG

Model Solution: SAPBC414**S**_UPDATE_STRAVELAG

1. Copy the template **SAPBC414T_UPDATE_STRAVELAG** with **all** subobjects to **SAPMZ##_UPDATE_STRAVELAG** (## = group number). As this is a type 1 program, a transaction code is not required.
2. The copied program generates a list that presents the data of the travel agencies maintained in the **STRAVELAG** table. By selecting one or more lines, the user can change the data of the corresponding travel agencies on the subsequent screen (100).

   Extend the program to include the database dialog. The changed data is to be saved to the **STRAVELAG** database table by clicking the *Save* icon (function code SAVE) on screen 100.

   To do so, the sub program **SAVE_CHANGES** is called in the **USER_COMMAND_0100** PAI module of screen 100. This encapsulates the database dialog. This subroutine has already been created (empty).

Continued on next page

From the sub program, write the **changed** address data back to the STRAVELAG database table. When doing so, note the performance aspects. If the save is successful, the message **S030** is to be sent; if not, use the message **I048** to inform the user.

Hint: The travel agency data is buffered in the internal table **ITAB_TRAVEL** (work area WA_TRAVEL). The line structure in the internal table has the same structure as that in STRAVELAG, with the exception of the additional field MARK_CHANGED (C(1)).

If the address data on the screen 100 has been changed, MARK_CHANGED has the value “X”. Otherwise it is initial.

Solution 2: Changing Several Data Records (optional)

Task: Changing Several Data Records (optional)

Program: SAPMZ##_UPDATE_STRAVELAG

Copy template: SAPBC414**T**_UPDATE_STRAVELAG

Model Solution: SAPBC414**S**_UPDATE_STRAVELAG

1. Copy the template **SAPBC414T_UPDATE_STRAVELAG** with **all** subobjects to **SAPMZ##_UPDATE_STRAVELAG** (## = group number). As this is a type 1 program, a transaction code is not required.

   a) -

2. The copied program generates a list that presents the data of the travel agencies maintained in the **STRAVELAG** table. By selecting one or more lines, the user can change the data of the corresponding travel agencies on the subsequent screen (100).

   Extend the program to include the database dialog. The changed data is to be saved to the **STRAVELAG** database table by clicking the *Save* icon (function code SAVE) on screen 100.

   To do so, the sub program **SAVE_CHANGES** is called in the **USER_COMMAND_0100** PAI module of screen 100. This encapsulates the database dialog. This subroutine has already been created (empty).

   From the sub program, write the **changed** address data back to the STRAVELAG database table. When doing so, note the performance aspects. If the save is successful, the message **S030** is to be sent; if not, use the message **I048** to inform the user.

   Hint: The travel agency data is buffered in the internal table **ITAB_TRAVEL** (work area WA_TRAVEL). The line structure in the internal table has the same structure as that in STRAVELAG, with the exception of the additional field MARK_CHANGED (C(1)).

   If the address data on the screen 100 has been changed, MARK_CHANGED has the value “X”. Otherwise it is initial.

   a) see the model solution below

Continued on next page

Result

Model Solution SAPBC414S_UPDATE_STRAVELAG

Module Pool

```
*&---------------------------------------------------------------------*
*& Modulpool         SAPBC414S_UPDATE_STRAVELAG                      *
*&---------------------------------------------------------------------*
INCLUDE bc414s_update_stravelagtop.
INCLUDE bc414s_update_stravelagf01.
INCLUDE bc414s_update_stravelago01.
INCLUDE bc414s_update_stravelagi01.
INCLUDE bc414s_update_stravelage01.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.
* fill table control (only agencies, marked on list)
  LOOP AT ITAB_TRAVEL INTO WA_TRAVEL WITH CONTROL TC_STRAVELAG.
    MODULE TRANS_TO_DYNPRO.
  ENDLOOP.
*

PROCESS AFTER INPUT.
  MODULE EXIT AT EXIT-COMMAND.
  LOOP AT ITAB_TRAVEL.
    CHAIN.
     FIELD: STRAVELAG-STREET, STRAVELAG-POSTBOX, STRAVELAG-POSTCODE,
            STRAVELAG-CITY, STRAVELAG-COUNTRY, STRAVELAG-REGION,
            STRAVELAG-TELEPHONE, STRAVELAG-URL, STRAVELAG-LANGU.
* mark datasets, that were changed in table control (subset of all
* agencies, thet were shown on table control)
      MODULE SET_MARKER ON CHAIN-REQUEST.
    ENDCHAIN.
  ENDLOOP.
  MODULE SAVE_OK_CODE.
  MODULE USER_COMMAND_0100.
```

Continued on next page

Internal Use SAP Partner Only

TOP Include

```
*&---------------------------------------------------------------------*
*& Include BC414S_UPDATE_STRAVELAGTOP                                  *
*&---------------------------------------------------------------------*
PROGRAM  sapbc414s_update_stravelag NO STANDARD PAGE HEADING
                                    LINE-SIZE 120
                                    LINE-COUNT 10
                                    MESSAGE-ID bc414.

* Line type definition for internal table itab_travel
TYPES: BEGIN OF stravel_type.
        INCLUDE STRUCTURE stravelag.
TYPES:            mark_changed,
       END OF stravel_type.

* Standard internal table for travel agency data buffering and
* corresponding workarea
DATA: itab_stravelag LIKE STANDARD TABLE OF stravelag
      WITH NON-UNIQUE KEY agencynum,
      wa_stravelag TYPE stravelag.

* Workarea for transport of field values from/to screen 100
TABLES: stravelag.

* Transport function code from screen 100
DATA: ok_code TYPE sy-ucomm, save_ok LIKE ok_code.

* Table control structure on screen 100
CONTROLS: tc_stravelag TYPE TABLEVIEW USING SCREEN '0100'.

* Internal table to collect marked list entries, corresponding
* workarea
DATA: itab_travel TYPE STANDARD TABLE OF stravel_type
      WITH NON-UNIQUE KEY agencynum,
      wa_travel TYPE stravel_type.

* Mark field displayed as checkbox on list
DATA: mark.

* Flags:
DATA: flag,         "changes performed on table control
      modify_list.  "modification of list buffer is neccessary

* Positions of fields on list
```

Continued on next page

```
CONSTANTS: pos1 TYPE i VALUE   1,
           pos2 TYPE i VALUE   3,
           pos3 TYPE i VALUE  14,
           pos4 TYPE i VALUE  40,
           pos5 TYPE i VALUE  71,
           pos6 TYPE i VALUE  82,
           pos7 TYPE i VALUE 108.
```

PBO Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_UPDATE_STRAVELAGO01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'DYNPRO'.
  SET TITLEBAR 'DYNPRO'.
ENDMODULE.                              " STATUS_0100  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  TRANS_TO_DYNPRO  OUTPUT
*&---------------------------------------------------------------------*
MODULE trans_to_dynpro OUTPUT.
* Field transport to screen
  MOVE-CORRESPONDING wa_travel TO stravelag.
ENDMODULE.                              " TRANS_TO_DYNPRO  OUTPUT
```

PAI Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_UPDATE_STRAVELAGI01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
```

Continued on next page

```
  CASE save_ok.
    WHEN 'SAVE'.
      IF flag IS INITIAL.
* enries on table control not changed.
        SET SCREEN 0.
      ELSE.
* at least one field on table control changed
        PERFORM save_changes.
        SET SCREEN 0.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&---------------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&---------------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR: ok_code.
ENDMODULE.                             " SAVE_OK_CODE  INPUT


*&---------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&---------------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes performed on screen
        LEAVE TO SCREEN 0.
      ELSE.
* at least one field on table control changed.
        PERFORM popup_to_confirm_loss_of_data.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT


*&---------------------------------------------------------------------*
*&      Module  SET_MARKER  INPUT
*&---------------------------------------------------------------------*
MODULE set_marker INPUT.
  MOVE-CORRESPONDING stravelag TO wa_travel.
```

Continued on next page


```
  wa_travel-mark_changed = 'X'.
* mark datasets in internal table as modified
  MODIFY TABLE itab_travel FROM wa_travel.
* at least one dataset is modified in table control
  flag = 'X'.
ENDMODULE.                              " SET_MARKER  INPUT
```

Events

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_UPDATE_STRAVELAGE01                                *
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&   Event START-OF-SELECTION
*&---------------------------------------------------------------------*
START-OF-SELECTION.
* Read data from STRAVELAG into internal table ITAB_STRAVELAG
  PERFORM read_data USING itab_stravelag.
* Write data from ITAB_STRAVELAG on list
  PERFORM write_data.


*&---------------------------------------------------------------------*
*&   Event TOP-OF-PAGE
*&---------------------------------------------------------------------*
TOP-OF-PAGE.
* Write page title and page heading
  PERFORM write_header.


*&---------------------------------------------------------------------*
*&   Event END-OF-SELECTION
*&---------------------------------------------------------------------*
END-OF-SELECTION.
* Set PF-Status and Title of list
  SET PF-STATUS 'LIST'.
  SET TITLEBAR 'LIST'.


*&---------------------------------------------------------------------*
*&   Event AT USER-COMMAND
```

Continued on next page

```
*&---------------------------------------------------------------------*
AT USER-COMMAND.
  CLEAR: modify_list, flag, itab_travel.
* Collect data corresponding to marked lines into internal table
  PERFORM loop_at_list USING itab_travel.
* Call screen if any line on list was marked
  CHECK NOT itab_travel IS INITIAL.
  PERFORM call_screen.
* Modify list buffer if database table was modified -> submit report
  CHECK NOT modify_list IS INITIAL.
  SUBMIT (sy-cprog).
```

FORM Routines

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_UPDATE_STRAVELAGF01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  READ_DATA
*&---------------------------------------------------------------------*
*      -->P_ITAB_STRAVELAG  text
*---------------------------------------------------------------------*
FORM read_data USING p_itab_stravelag LIKE itab_stravelag.
  SELECT * FROM stravelag
           INTO CORRESPONDING FIELDS OF TABLE p_itab_stravelag.
ENDFORM.                                " READ_DATA


*&---------------------------------------------------------------------*
*&      Form  WRITE_DATA
*&---------------------------------------------------------------------*
FORM write_data.
  LOOP AT itab_stravelag INTO wa_stravelag.
    WRITE AT: /pos1 mark AS CHECKBOX,
                pos2 wa_stravelag-agencynum COLOR COL_KEY,
                pos3 wa_stravelag-name,
                pos4 wa_stravelag-street,
                pos5 wa_stravelag-postcode,
                pos6 wa_stravelag-city,
                pos7 wa_stravelag-country.
    HIDE: wa_stravelag.
  ENDLOOP.
```

Continued on next page


```
ENDFORM.                               " WRITE_DATA


*&---------------------------------------------------------------------*
*&      Form  WRITE_HEADER
*&---------------------------------------------------------------------*
FORM write_header.
  WRITE: / 'Travel agency data'(007), AT sy-linsz sy-pagno.
  ULINE.
  FORMAT COLOR COL_HEADING.
  WRITE AT: /pos2 'Agency'(001),
             pos3 'Name'(002),
             pos4 'Street'(003),
             pos5 'Postal Code'(004),
             pos6 'City'(005),
             pos7 'Country'(006).
  ULINE.
ENDFORM.                               " WRITE_HEADER

*&---------------------------------------------------------------------*
*&      Form  LOOP_AT_LIST
*&---------------------------------------------------------------------*
*      -->P_ITAB_AGNECYNUM  text
*----------------------------------------------------------------------*
FORM loop_at_list USING p_itab_travel LIKE itab_travel.
  DO.
    CLEAR: mark.
    READ LINE sy-index FIELD VALUE mark.
    IF sy-subrc <> 0.
      EXIT.
    ENDIF.
    CHECK NOT mark IS INITIAL.
    APPEND wa_stravelag TO p_itab_travel.
  ENDDO.
ENDFORM.                               " LOOP_AT_LIST


*&---------------------------------------------------------------------*
*&      Form  CALL_SCREEN
*&---------------------------------------------------------------------*
FORM call_screen.
* Initialize table control on screen
  REFRESH CONTROL 'TC_STRAVELAG' FROM SCREEN '0100'.
* Show screen in modal dialog box.
  CALL SCREEN 100 STARTING AT  5  5
```

Continued on next page


```
                  ENDING   AT 80 15.
ENDFORM.                              " CALL_SCREEN


*&---------------------------------------------------------------------*
*&      Form  POPUP_TO_CONFIRM_LOSS_OF_DATA
*&---------------------------------------------------------------------*
FORM popup_to_confirm_loss_of_data.
  DATA answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_LOSS_OF_DATA'
       EXPORTING
            textline1 = 'Cancel processing of travel agencies?'(008)
            titel     = 'Cancel processing'(009)
       IMPORTING
            answer    = answer.
  CASE answer.
    WHEN 'J'.
      LEAVE TO SCREEN 0.
    WHEN 'N'.
      LEAVE TO SCREEN '0100'.
  ENDCASE.
ENDFORM.                              " POPUP_TO_CONFIRM_LOSS_OF_DATA

*&---------------------------------------------------------------------*
*&      Form  SAVE_CHANGES
*&---------------------------------------------------------------------*
FORM save_changes.
* declare internal table and workarea of same linetype as DB table
  DATA: itab TYPE STANDARD TABLE OF stravelag,
        wa LIKE LINE OF itab.
* search for datasets changed on the screen
  LOOP AT itab_travel INTO wa_travel
       WHERE mark_changed = 'X'.
* fill workarea fitting to DB table
    MOVE-CORRESPONDING wa_travel TO wa.
* fill corresponding internal table
    APPEND wa TO itab.
  ENDLOOP.
* mass update on stravelag -> best performance
  UPDATE stravelag FROM TABLE itab.
* check success
  IF sy-subrc = 0.
* all datasets are successfully updated
    MESSAGE s030.
  ELSE.
```

Continued on next page


```
* at least one dataset from the internal table could not be updated
* on the database table
    MESSAGE i048.
  ENDIF.
* Flag: List does not show correct data any more
  modify_list = 'X'.
ENDFORM.                               " SAVE_CHANGES
```

Lesson Summary

You should now be able to:

- using Open SQL commands to execute database changes.

Unit Summary

You should now be able to:

- using Open SQL commands to execute database changes.

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 2

LUWs and Client/Server Architecture

Unit Overview

- SAP LUW
- Database LUW
- Consequences of the Client/Server Architecture

Unit Objectives

After completing this unit, you will be able to:

- Explain the meaning of the terms database LUW and SAP LUW
- Explain why you need to bundle changes to database tables in the client/server architecture of the R/3 System

Unit Contents

Lesson: LUWs in the SAP Client Server Architecture

Lesson Overview

- SAP LUW
- Database LUW
- Consequences of the Client/Server Architecture

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the meaning of the terms database LUW and SAP LUW
- Explain why you need to bundle changes to database tables in the client/server architecture of the R/3 System

Business Example

You want to bundle corresponding database changes.

LUWs in the SAP Client Server Architecture

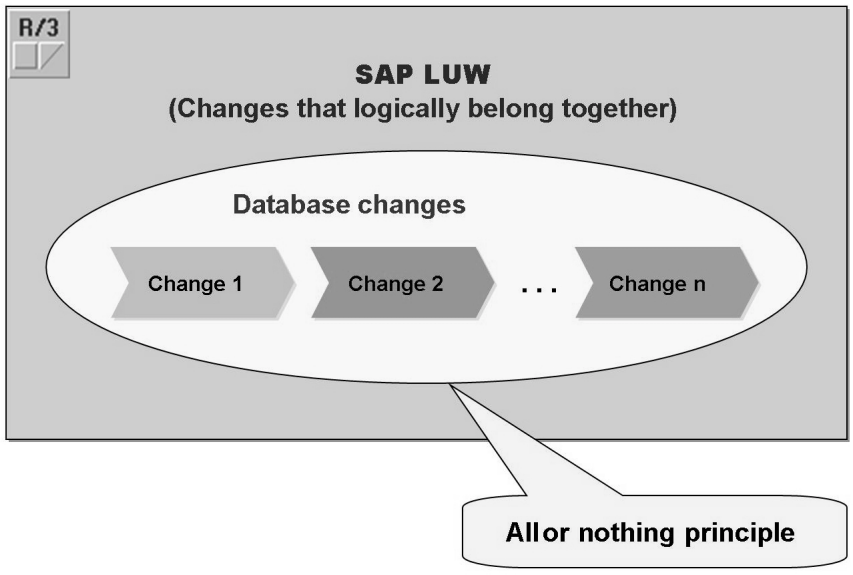


Figure 15: SAP Logical Unit of Work (SAP LUW):

An **SAP-LUW** (Logical Unit of Work) consists of changes in the R/3 System that belong together from a logical point of view. These changes are either carried out in full or they are not carried out at all (“all or nothing principle”).

In general, a business transaction is not processed by a single SAP LUW. The entire process from receipt of a customer order up to the issue of an invoice, for example, is split up into individual, logical parts. Each part corresponds to an SAP LUW. The definition of SAP LUWs depends on the entire process and its modelling.

For further information, refer to the ABAP Editor keyword documentation for the **term transaction processing**.

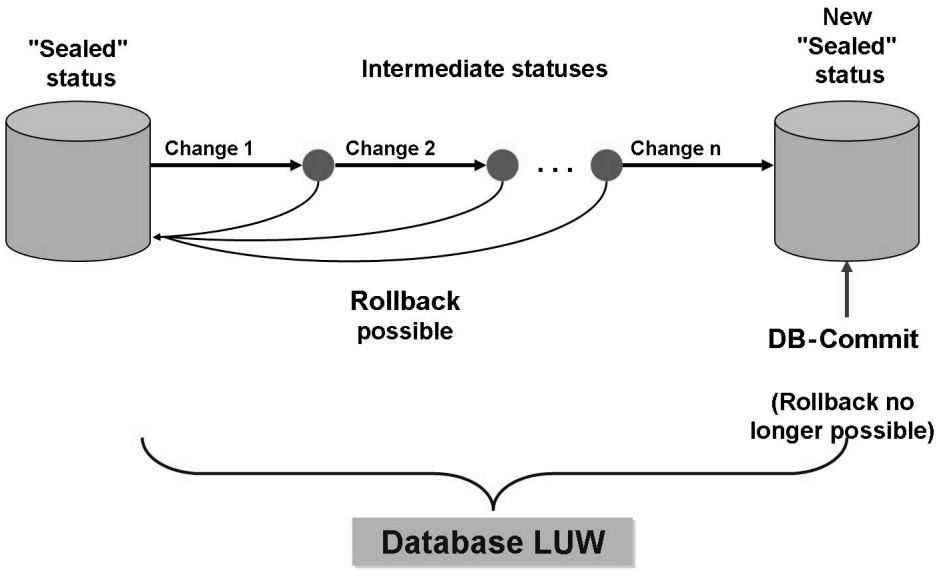


Figure 16: Database LUW

A **database LUW** consists of changes that are executed up until the database status is “sealed” (DB Commit).

Within a database LUW, it is always possible to discard all the changes that have taken place up to that point (DB rollback), in which case the database is reset to the status it had before the current database LUW. You use this DB rollback function in order to restore the previous (consistent) database status if an error has occurred.

The current database status is “sealed” by a DB Commit. After this point, it is no longer possible to discard the current database LUW.

Using the ABAP statements `ROLLBACK WORK` and `COMMIT WORK`, you can explicitly implement a DB rollback or DB Commit. There are also situations where a DB Commit is triggerred implicitly (see following figures!)

For further information, refer to the ABAP Editor keyword documentation for the **term transaction processing**.

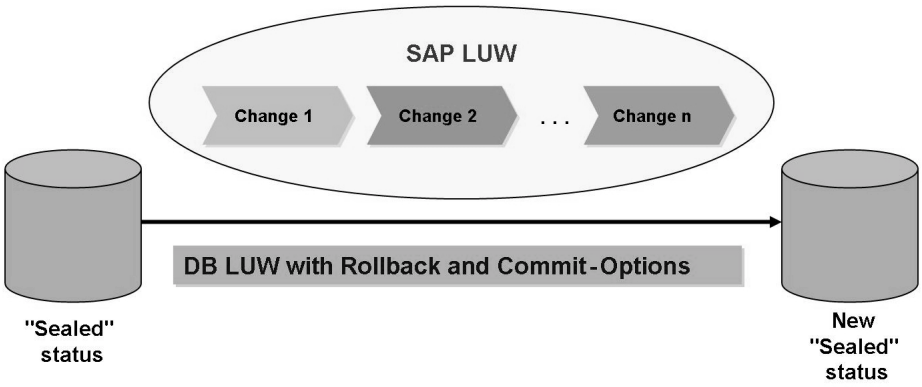


Figure 17: Implementing a SAP LUW Within a DB LUW

If there is an error during processing of an SAP-LUW, it should be possible to return to a consistent database status that existed before the beginning of the SAP-LUW. So that this is possible, the SAP-LUW must be processed within a DB-LUW.

This is not a trivial matter - due to the client-server architecture of the R/3 System - because an R/3 transaction usually has several screens for entering change data and, whenever there is a screen change, the system automatically triggers an implicit DB Commit (see the following slides). However, it should be possible within a transaction to bundle user entries that form an SAP-LUW within a DB-LUW and write them to the database.

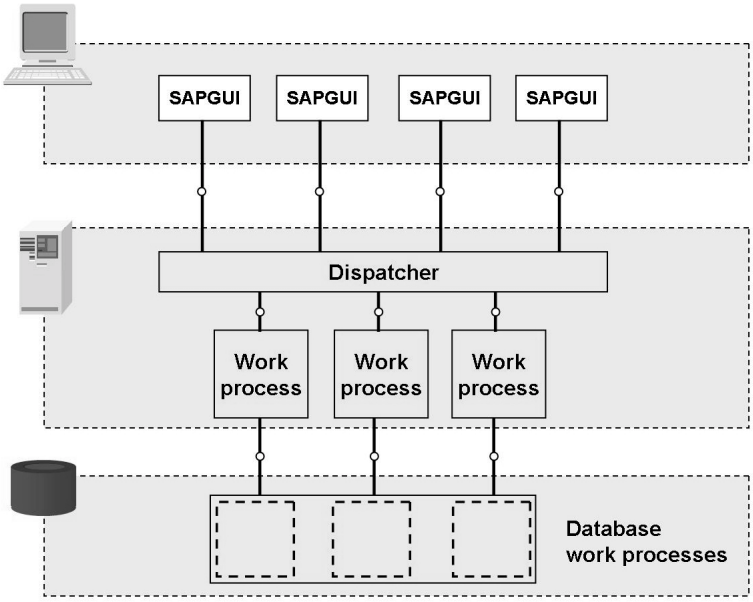


Figure 18: Client-Server Architecture of the R/3 System

The SAP R/3 System is based on a client/server architecture with three levels; database, application server, and presentation server.

This architecture, along with the distribution of users' requests (**user dispatching**), leads to a **highly-efficient and cost-effective multi-user system**.

The three-tier architecture means that a large number of users with **low-cost** desktop computers (with low performance) can be mapped to a small number of **high-performance (and considerably more expensive)** work processes on application servers. Each work process on an application server is assigned a work process on a high-performance database server.

Distributing user requests to work processes means that individual clients at presentation server level are assigned to a work process for a particular period. In turn, the work process uses another work process in the database. After the work process has processed the user input in a dialog step, the user, along with the program context, is “removed from” the work process. This work process can then be used by another user.

The three-tier architecture is far more scalable than a “fat” client architecture, where the presentation and application levels run on one server. With a three-tier architecture, the number of database users is considerably lower than the number of users active in the system. This has a positive effect on the behavior of the database.

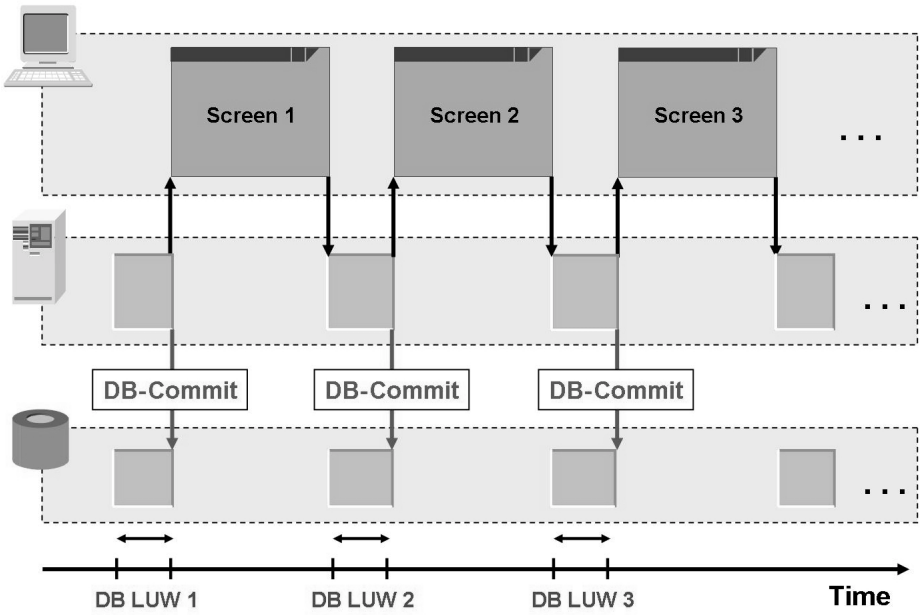


Figure 19: Implicit DB Commits

When each single screen is displayed, the current work process is released on the the application server. This, in turn, triggers the release of the respective database work process and automatically initiates an implicit database commit.

Releasing the work process before each user dialog ensures that the user actions, which usually last longer, do not block any work processes on the application server and, particularly, on the database server. This leads to a lesser load on database recources. Only when the user has completed processing the screen, which will then require internal program processing, is he or she “rolled back”, together with the program context. This means that the user is returned to a work process for processing.

Implicit DB commits are always initiated whenever the program has to wait, as in the following cases:

- When the system sends an SAP screen
- When the system sends a dialog message
- Whenever there are synchronous and asynchronous RFC calls (Remote Function Call)
- Statements `CALL TRANSACTION` <*t_code*> or `SUBMIT` <*program*>.

Hint: Due to the above-mentioned, implicit DB commits, changes that belong to an SAP LUW may not be placed in different dialog steps (dialog step = program processing after a screen). The reason is because these steps would thus not be within a DB LUW.

Exercise 3: LUW Concepts

Exercise Objectives

After completing this exercise, you will be able to:

- Assess function modules and subroutines for LUW processing suitability.

Business Example

You want to bundle corresponding database changes.

Task: Executing an LUW

Program: SAPMZ##_BOOKINGS1

Transaction code: Z##_BOOKINGS1

Copy from: SAPBC414**T**_BOOKINGS_01

Model Solution: SAPBC414**S**_BOOKINGS_01

1. Copy the program template **SAPBC414T_BOOKINGS_01** with **all** subobjects to **SAPMZ##_BOOKINGS1** (## is your group number) and assign transaction code **Z##_BOOKINGS1** to the program. Familiarize yourself with the program and its functions.
2. The copied program allows you to cancel bookings for a flight. To do this, you select the respective bookings for cancellation on screen 200. You then select the *Save* icon (SAVE function code) to change the bookings you selected in the **SBOOK** table. In addition, the respective flight in table **SFLIGHT** must be modified **within the same database LUW** (booking total and number of bookings for the flight will change because of the cancellation) since the database table change for both database tables must follow the "all or nothing" principle.

   You use existing function modules to execute the database changes and encapsulate them in the sub program **SAVE_MODIFIED_BOOKING** which is called from the USER_COMMAND_0200 PAI module (screen 200). A choice of two function modules is available for each table: **UPDATE_SBOOK , UPDATE_SBOOK_A , UPDATE_SFLIGHT and UPDATE_SFLIGHT_A .**

Continued on next page

By combining the correct function modules in the correct call sequence, you ensure that, if there is an error, the data in the database tables concerned remains consistent.

Which function modules must be called and in what order?

For this purpose, check the source code in the function modules for ABAP statements that terminate the database LUW prematurely and can, therefore, result in inconsistent data being written to the tables permanently.

3. Call up the function modules in the appropriate order from the subroutine SAVE_MODIFIED_BOOKING.
4. Deal with the exceptions of the function modules. Possible user messages:

   Flight / bookings updated → message 034

   Error when updating flight / booking → message 044

   Change unsuccessful → message 048

   Flight fully booked or over booked → message 045

   Flight does not exist → message 046

Hint: The data records to be changed in the database table SBOOK are buffered in the internal table **ITAB_SBOOK_MODIFY** .

The key fields of the corresponding flight can be accessed using the **WA_SFLIGHT** structure variable.

Solution 3: LUW Concepts

Task: Executing an LUW

Program: SAPMZ##_BOOKINGS1

Transaction code: Z##_BOOKINGS1

Copy from: SAPBC414**T**_BOOKINGS_01

Model Solution: SAPBC414**S**_BOOKINGS_01

1. Copy the program template **SAPBC414T_BOOKINGS_01** with **all** subobjects to **SAPMZ##_BOOKINGS1** (**##** is your group number) and assign transaction code **Z##_BOOKINGS1** to the program. Familiarize yourself with the program and its functions.

   a) -

2. The copied program allows you to cancel bookings for a flight. To do this, you select the respective bookings for cancellation on screen 200. You then select the *Save* icon (SAVE function code) to change the bookings you selected in the **SBOOK** table. In addition, the respective flight in table **SFLIGHT** must be modified **within the same database LUW** (booking total and number of bookings for the flight will change because of the cancellation) since the database table change for both database tables must follow the "all or nothing" principle.

   You use existing function modules to execute the database changes and encapsulate them in the sub program **SAVE_MODIFIED_BOOKING** which is called from the USER_COMMAND_0200 PAI module (screen 200). A choice of two function modules is available for each table: **UPDATE_SBOOK , UPDATE_SBOOK_A , UPDATE_SFLIGHT and UPDATE_SFLIGHT_A .**

   By combining the correct function modules in the correct call sequence, you ensure that, if there is an error, the data in the database tables concerned remains consistent.

   Which function modules must be called and in what order?

Continued on next page

For this purpose, check the source code in the function modules for ABAP statements that terminate the database LUW prematurely and can, therefore, result in inconsistent data being written to the tables permanently.

a) UPDATE_SBOOK and UPDATE_SFLIGHT are the correct function modules since they do not send a type I success message that could destroy data consistency.

UPDATE_SBOOK must be called first since UPDATE_SFLIGHT evaluates its result.

3. Call up the function modules in the appropriate order from the subroutine SAVE_MODIFIED_BOOKING.

a) See model solution

4. Deal with the exceptions of the function modules. Possible user messages:

Flight / bookings updated → message 034

Error when updating flight / booking → message 044

Change unsuccessful → message 048

Flight fully booked or over booked → message 045

Flight does not exist → message 046

Hint: The data records to be changed in the database table SBOOK are buffered in the internal table **ITAB_SBOOK_MODIFY** .

The key fields of the corresponding flight can be accessed using the **WA_SFLIGHT** structure variable.

a) See model solution

Result

Model Solution SAPBC414S_BOOKINGS_01

Module Pool

```
*&---------------------------------------------------------------------*
*& Modulpool         SAPBC414S_BOOKINGS_01                           *
*&---------------------------------------------------------------------*
INCLUDE BC414S_BOOKINGS_01TOP.
INCLUDE BC414S_BOOKINGS_01O01.
INCLUDE BC414S_BOOKINGS_01I01.
```

Continued on next page

Internal Use SAP Partner Only

```
INCLUDE BC414S_BOOKINGS_01F01.
INCLUDE BC414S_BOOKINGS_01F02.
INCLUDE BC414S_BOOKINGS_01F03.
INCLUDE BC414S_BOOKINGS_01F04.
INCLUDE BC414S_BOOKINGS_01F05.
INCLUDE BC414S_BOOKINGS_01F06.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.
*
PROCESS AFTER INPUT.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  CHAIN.
* cancel booking: check if flight exists or flight can be created
    FIELD: SDYN_CONN-CARRID, SDYN_CONN-CONNID, SDYN_CONN-FLDATE.
    MODULE USER_COMMAND_0100.
  ENDCHAIN.
```

SCREEN 200

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0200.
  MODULE TRANS_DETAILS.
  CALL SUBSCREEN SUB1 INCLUDING SY-CPROG '0201'.
  LOOP AT ITAB_BOOK INTO WA_BOOK WITH CONTROL TC_SBOOK.
    MODULE TRANS_TO_TC.
* allow only modification of bookings, that are not allready cancelled
    MODULE MODIFY_SCREEN.
  ENDLOOP.
*
PROCESS AFTER INPUT.
  LOOP AT ITAB_BOOK.
* mark changed bookings in internal table itab_book
    FIELD SDYN_BOOK-CANCELLED MODULE MODIFY_ITAB ON REQUEST.
  ENDLOOP.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  MODULE USER_COMMAND_0200.
```

Continued on next page

SCREEN 201

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

SCREEN 300

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0300.
  MODULE TABSTRIP_INIT.
  MODULE TRANS_DETAILS.
  CALL SUBSCREEN TAB_SUB INCLUDING SY-CPROG SCREEN_NO.
*
PROCESS AFTER INPUT.
  CALL SUBSCREEN TAB_SUB.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  MODULE TRANS_FROM_0300.
  MODULE USER_COMMAND_0300.
```

SCREEN 301

```
PROCESS BEFORE OUTPUT.
* MODULE HIDE_BOOKID.
PROCESS AFTER INPUT.
```

SCREEN 302

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

SCREEN 303

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

TOP Include

```
*&---------------------------------------------------------------------*
*& Include BC414S_BOOKINGS_01TOP                                       *
*&---------------------------------------------------------------------*
```

Continued on next page

```
PROGRAM  sapbc414s_bookings_01 MESSAGE-ID bc414.

* line type of internal table itab_book, used to display bookings in
* table control
TYPES: BEGIN OF wa_book_type.
INCLUDE: STRUCTURE sbook.
TYPES:   name TYPE scustom-name,
         mark,
       END OF wa_book_type.

* work area and internal table used to display bookings in table
* control
DATA: wa_book TYPE wa_book_type,
      itab_book TYPE TABLE OF wa_book_type.

* bookings to be modified on database table
DATA: itab_sbook_modify TYPE TABLE OF sbook.

* change documents: bookings before changes are performed
DATA: itab_cd TYPE TABLE OF sbook WITH NON-UNIQUE KEY
      carrid connid fldate bookid customid.

* work areas for database tables spfli, sflight, sbook.
DATA: wa_sbook TYPE sbook, wa_sflight TYPE sflight, wa_spfli TYPE
      spfli.

* complex transactions: number of the customer created in the called
* transaction
data: scust_id(20).

* transport function codes from screens
DATA: ok_code TYPE sy-ucomm, save_ok LIKE ok_code.
* define subscreen screen number on tabstrip, screen 300
DATA: screen_no TYPE sy-dynnr.
* used to handle sy-subrc, which is determined in form
DATA  sysubrc LIKE sy-subrc.

* transporting fields to/from screen
TABLES: sdyn_conn, sdyn_book.
* table control declaration (display bookings),
* tabstrip declaration (create booking)
CONTROLS: tc_sbook TYPE TABLEVIEW USING SCREEN '0200',
          tab TYPE TABSTRIP.
```

Continued on next page

PBO Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_01O01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'DYN_100'.
  SET TITLEBAR 'DYN_100'.
ENDMODULE.                             " STATUS_0100  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  STATUS_0200  OUTPUT
*&---------------------------------------------------------------------*
MODULE status_0200 OUTPUT.
  SET PF-STATUS 'DYN_200'.
  SET TITLEBAR 'DYN_200' WITH sdyn_conn-carrid sdyn_conn-connid
                              sdyn_conn-fldate.
ENDMODULE.                             " STATUS_0200  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  STATUS_0300  OUTPUT
*&---------------------------------------------------------------------*
MODULE status_0300 OUTPUT.
  SET PF-STATUS 'DYN_300'.
  SET TITLEBAR 'DYN_300' WITH sdyn_conn-carrid sdyn_conn-connid
                              sdyn_conn-fldate.
ENDMODULE.                             " STATUS_0300  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  TRANS_DETAILS  OUTPUT
*&---------------------------------------------------------------------*
MODULE trans_details OUTPUT.
  MOVE-CORRESPONDING: wa_spfli   TO sdyn_conn,
                      wa_sflight TO sdyn_conn,
                      wa_sbook   TO sdyn_book.
ENDMODULE.                             " TRANS_DETAILS  OUTPUT


*&---------------------------------------------------------------------*
```

Continued on next page

```
*&      Module  TRANS_TO_TC  OUTPUT
*&---------------------------------------------------------------------*
MODULE trans_to_tc OUTPUT.
  MOVE-CORRESPONDING wa_book TO sdyn_book.
ENDMODULE.                             " TRANS_TO_TC  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  MODIFY_SCREEN  OUTPUT
*&---------------------------------------------------------------------*
MODULE modify_screen OUTPUT.
  LOOP AT SCREEN.
    CHECK screen-name = 'SDYN_BOOK-CANCELLED'.
    CHECK ( NOT sdyn_book-cancelled IS INITIAL ) AND
          ( sdyn_book-mark IS INITIAL ).
    screen-input = 0.
    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.                             " MODIFY_SCREEN  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  TABSTRIP_INIT  OUTPUT
*&---------------------------------------------------------------------*
MODULE tabstrip_init OUTPUT.
  CHECK tab-activetab IS INITIAL.
  tab-activetab = 'BOOK'.
  screen_no = '0301'.
ENDMODULE.                             " TABSTRIP_INIT  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  HIDE_BOOKID  OUTPUT
*&---------------------------------------------------------------------*
MODULE hide_bookid OUTPUT.
* hide field displaying customer number when working with number range
* object BS_SCUSTOM
  LOOP AT SCREEN.
    CHECK screen-name = 'SDYN_BOOK-BOOKID'.
    screen-active = 0.
    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.                             " HIDE_BOOKID  OUTPUT
```

Continued on next page

PAI Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_01I01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&---------------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      CASE sy-dynnr.
        WHEN '0100'.
          LEAVE PROGRAM.
        WHEN '0200'.
          LEAVE TO SCREEN '0100'.
        WHEN '0300'.
          LEAVE TO SCREEN '0100'.
        WHEN OTHERS.
      ENDCASE.
    WHEN 'EXIT'.
      LEAVE PROGRAM.
    WHEN OTHERS.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT


*&---------------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&---------------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
ENDMODULE.                             " SAVE_OK_CODE  INPUT


*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
*****************************CANCEL BOOKING*************************
    WHEN 'BOOKC'.
      PERFORM read_sflight USING wa_sflight sysubrc.
* process returncode - if flight does not exist: e-message
```

Continued on next page


```
      PERFORM process_sysubrc_bookc.
      PERFORM read_spfli USING wa_spfli.
      PERFORM read_sbook USING itab_book itab_cd.
      REFRESH CONTROL 'TC_SBOOK' FROM SCREEN '0200'.
**************************CREATE BOOKING*************************
    WHEN 'BOOKN'.
      PERFORM read_sflight USING wa_sflight sysubrc.
* process returncode - if flight does not exist: e-message
      PERFORM process_sysubrc_bookn.
      PERFORM read_spfli USING wa_spfli.
      PERFORM initialize_sbook USING wa_sbook.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN '0100'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0200  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0200 INPUT.
  CASE save_ok.
    WHEN 'SAVE'.
* collect marked (changed) data sets in seperate internal table
      PERFORM collect_modified_data USING itab_sbook_modify.
* perform database changes
      PERFORM save_modified_booking.
      SET SCREEN '0100'.
    WHEN 'BACK'.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0200'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0200  INPUT


*&---------------------------------------------------------------------*
*&      Module  MODIFY_ITAB  INPUT
*&---------------------------------------------------------------------*
MODULE modify_itab INPUT.
  wa_book-cancelled = sdyn_book-cancelled.
  wa_book-mark = 'X'.
  MODIFY itab_book FROM wa_book INDEX tc_sbook-current_line.
```

Continued on next page


```
ENDMODULE.                              " MODIFY_ITAB  INPUT


*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0300  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0300 INPUT.
  PERFORM tabstrip_set.
  CASE save_ok.
    WHEN 'NEW_CUSTOM'.
      PERFORM create_new_customer.
      SET SCREEN '0300'.
    WHEN 'SAVE'.
      PERFORM save_new_booking.
      SET SCREEN '0100'.
    WHEN 'BACK'.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0300'.
  ENDCASE.
ENDMODULE.                              " USER_COMMAND_0300  INPUT


*&---------------------------------------------------------------------*
*&      Module  TRANS_FROM_0300  INPUT
*&---------------------------------------------------------------------*
MODULE trans_from_0300 INPUT.
  MOVE-CORRESPONDING sdyn_book TO wa_sbook.
ENDMODULE.                              " TRANS_FROM_0300  INPUT
```

FORM Routines

F01

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_01F01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  COLLECT_MODIFIED_DATA
*&---------------------------------------------------------------------*
*      -->P_ITAB_SBOOK_MODIFY  text
```

Continued on next page

Internal Use SAP Partner Only

```
*---------------------------------------------------------------------*
FORM collect_modified_data USING p_itab_sbook_modify
                                 LIKE itab_sbook_modify.
  DATA: wa_book LIKE LINE OF itab_book,
        wa_sbook_modify LIKE LINE OF p_itab_sbook_modify.
  CLEAR: p_itab_sbook_modify.
* Only bookings are collected, that
* 1) have been changed (mark = 'X')
* 2) shall be cancelled (cancelled = 'X')
  LOOP AT itab_book INTO wa_book
       WHERE    mark = 'X'
       AND cancelled = 'X'.
    MOVE-CORRESPONDING wa_book TO wa_sbook_modify.
    APPEND wa_sbook_modify TO p_itab_sbook_modify.
  ENDLOOP.
ENDFORM.                               " COLLECT_MODIFIED_DATA


*&---------------------------------------------------------------------*
*&      Form  INITIALIZE_SBOOK
*&---------------------------------------------------------------------*
*      -->P_WA_SBOOK  text
*---------------------------------------------------------------------*
FORM initialize_sbook USING p_wa_sbook TYPE sbook.
  CLEAR p_wa_sbook.
  MOVE-CORRESPONDING wa_sflight TO p_wa_sbook.
  MOVE: wa_sflight-price    TO p_wa_sbook-forcurram,
        wa_sflight-currency TO p_wa_sbook-forcurkey,
        sy-datum            TO p_wa_sbook-order_date.
ENDFORM.                               " INITIALIZE_SBOOK


*&---------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKC
*&---------------------------------------------------------------------*
FORM process_sysubrc_bookc.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0200'.
    WHEN OTHERS.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                               " PROCESS_SYSUBRC_BOOKC
```

Continued on next page


```
*&---------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKN
*&---------------------------------------------------------------------*
FORM process_sysubrc_bookn.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0300'.
    WHEN OTHERS.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                               " PROCESS_SYSUBRC_BOOKN


*&---------------------------------------------------------------------*
*&      Form  TABSTRIP_SET
*&---------------------------------------------------------------------*
FORM tabstrip_set.
  IF save_ok = 'BOOK' OR save_ok = 'DETCON' OR save_ok = 'DETFLT'.
    tab-activetab = save_ok.
  ENDIF.
  CASE save_ok.
    WHEN 'BOOK'.
      screen_no = '0301'.
    WHEN 'DETCON'.
      screen_no = '0302'.
    WHEN 'DETFLT'.
      screen_no = '0303'.
  ENDCASE.
ENDFORM.                               " TABSTRIP_SET


*&---------------------------------------------------------------------*
*&      Form  CREATE_NEW_CUSTOMER
*&---------------------------------------------------------------------*
FORM create_new_customer.
********************** TO BE IMPLEMENTED LATER *********************
ENDFORM.                               " CREATE_NEW_CUSTOMER

*&---------------------------------------------------------------------*
*&      Form  NUMBER_GET_NEXT
*&---------------------------------------------------------------------*
FORM number_get_next USING p_wa_sbook LIKE sbook.
********************** TO BE IMPLEMENTED LATER *********************
```

Continued on next page


```
ENDFORM.                                " NUMBER_GET_NEXT
```

F02

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_01F02
*---------------------------------------------------------------------*

*---------------------------------------------------------------------*
*   FORM ENQ_SFLIGHT
*---------------------------------------------------------------------*
FORM enq_sflight.
********************* TO BE IMPLEMENTED LATER **********************
ENDFORM.                                "ENQ_SFLIGHT


*---------------------------------------------------------------------*
*   FORM ENQ_SBOOK
*---------------------------------------------------------------------*
FORM enq_sbook.
********************* TO BE IMPLEMENTED LATER **********************
ENDFORM.                                "ENQ_SBOOK


*---------------------------------------------------------------------*
*   FORM ENQ_SFLIGHT_SBOOK
*---------------------------------------------------------------------*
FORM enq_sflight_sbook.
********************* TO BE IMPLEMENTED LATER **********************
ENDFORM.                                "ENQ_SFLIGHT_SBOOK


*---------------------------------------------------------------------*
*   FORM DEQ_ALL
*---------------------------------------------------------------------*
FORM deq_all.
********************* TO BE IMPLEMENTED LATER **********************
ENDFORM.                                "DEQ_ALL
```

Continued on next page

F03

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_01F03
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  READ_SFLIGHT
*&---------------------------------------------------------------------*
*      -->P_WA_SFLIGHT  text
*      -->P_SYSUBRC     text
*---------------------------------------------------------------------*
FORM read_sflight USING p_wa_sflight TYPE sflight
                        p_sysubrc LIKE sy-subrc.
  SELECT SINGLE * FROM sflight INTO p_wa_sflight
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid
         AND   fldate = sdyn_conn-fldate.
  p_sysubrc = sy-subrc.
ENDFORM.                                " READ_SFLIGHT
*&---------------------------------------------------------------------*
*&      Form  READ_SBOOK
*&---------------------------------------------------------------------*
*      -->P_ITAB_BOOK  text
*      -->P_ITAB_CD    text
*---------------------------------------------------------------------*
FORM read_sbook USING p_itab_book LIKE itab_book
                      p_itab_cd   LIKE itab_cd.
  TYPES: BEGIN OF wa_custom_type,
           id TYPE scustom-id,
           name TYPE scustom-name,
         END OF wa_custom_type.
  DATA: wa_custom TYPE wa_custom_type,
        itab_custom TYPE STANDARD TABLE OF wa_custom_type
        WITH NON-UNIQUE KEY id,
        wa_book LIKE LINE OF p_itab_book,
        wa_cd   LIKE LINE OF p_itab_cd.
  CLEAR: p_itab_book, p_itab_cd.
* Select customer names in buffer table (array fetch)
  SELECT id name FROM scustom INTO CORRESPONDING FIELDS
         OF TABLE itab_custom.
* Select all bookings on selected flight (array fetch)
  SELECT * FROM sbook INTO CORRESPONDING FIELDS OF TABLE p_itab_book
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid
         AND   fldate = sdyn_conn-fldate.
```

Continued on next page

```
* read customer names corresponding to customer number from buffer
* table
  LOOP AT p_itab_book INTO wa_book.
    READ TABLE itab_custom INTO wa_custom WITH TABLE KEY
                                          id = wa_book-customid.
    wa_book-name = wa_custom-name.
    MODIFY p_itab_book FROM wa_book.
    MOVE-CORRESPONDING wa_book TO wa_cd.
    APPEND wa_cd TO p_itab_cd.
  ENDLOOP.
  SORT p_itab_book BY bookid customid.
ENDFORM.                                " READ_SBOOK


*&---------------------------------------------------------------------*
*&      Form  READ_SPFLI
*&---------------------------------------------------------------------*
*      -->P_WA_SPFLI  text
*---------------------------------------------------------------------*
FORM read_spfli USING p_wa_spfli TYPE spfli.
  SELECT SINGLE * FROM spfli INTO p_wa_spfli
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid.
  IF sy-subrc <> 0.
    MESSAGE e022 WITH sdyn_conn-carrid sdyn_conn-connid.
  ENDIF.
ENDFORM.                                " READ_SPFLI
```

F04

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_01F04
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  SAVE_MODIFIED_BOOKING
*&---------------------------------------------------------------------*
FORM save_modified_booking.
* Modify data on database tables sbook and sflight
  CALL FUNCTION 'UPDATE_SBOOK'
       EXPORTING
            itab_sbook     = itab_sbook_modify
       EXCEPTIONS
            update_failure = 1
```

Continued on next page


```
           OTHERS         = 2.
  CASE sy-subrc.
    WHEN 0.
      PERFORM update_sflight.
    WHEN OTHERS.
      MESSAGE a044 WITH wa_sflight-carrid wa_sflight-connid
                        wa_sflight-fldate.
  ENDCASE.
ENDFORM.                               " SAVE_MODIFIED_BOOKING

*&---------------------------------------------------------------------*
*&      Form  UPDATE_SFLIGHT
*&---------------------------------------------------------------------*
FORM update_sflight.
  CALL FUNCTION 'UPDATE_SFLIGHT'
       EXPORTING
            carrier          = wa_sflight-carrid
            connection       = wa_sflight-connid
            date             = wa_sflight-fldate
       EXCEPTIONS
            update_failure   = 1
            flight_full      = 2
            flight_not_found = 3
            OTHERS           = 4.
  CASE sy-subrc.
    WHEN 0.
      MESSAGE s034 WITH wa_sflight-carrid wa_sflight-connid
                        wa_sflight-fldate.
    WHEN 1.
      MESSAGE a044 WITH wa_sflight-carrid wa_sflight-connid
                        wa_sflight-fldate.
    WHEN 2.
      MESSAGE a045.
    WHEN 3.
      MESSAGE a046.
    WHEN OTHERS.
      MESSAGE a048.
  ENDCASE.
ENDFORM.                               " UPDATE_SFLIGHT

*&---------------------------------------------------------------------*
*&      Form  SAVE_NEW_BOOKING
*&---------------------------------------------------------------------*
FORM save_new_booking.
******************** TO BE IMPLEMENTED LATER **********************
```

Continued on next page

```
ENDFORM.                                " SAVE_NEW_BOOKING
```

F05

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_01F05
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  CONVERT_TO_LOC_CURRENCY
*&---------------------------------------------------------------------*
*       -->P_WA_SBOOK  text
*---------------------------------------------------------------------*
FORM convert_to_loc_currency USING p_wa_sbook TYPE sbook.
  SELECT SINGLE currcode FROM scarr INTO p_wa_sbook-loccurkey
         WHERE carrid = p_wa_sbook-carrid.
  CALL FUNCTION 'CONVERT_TO_LOCAL_CURRENCY_N'
       EXPORTING
            client           = sy-mandt
            date             = sy-datum
            foreign_amount   = p_wa_sbook-forcuram
            foreign_currency = p_wa_sbook-forcurkey
            local_currency   = p_wa_sbook-loccurkey
       IMPORTING
            local_amount     = p_wa_sbook-loccuram
       EXCEPTIONS
            no_rate_found    = 1
            overflow         = 2
            no_factors_found = 3
            no_spread_found  = 4
            derived_2_times  = 5
            OTHERS           = 6.
  IF sy-subrc <> 0.
    MESSAGE e080 WITH sy-subrc.
  ENDIF.
ENDFORM.                                " CONVERT_TO_LOC_CURRENCY
```

F06

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_01F06
*---------------------------------------------------------------------*
```

Continued on next page

```
*&---------------------------------------------------------------------*
*&      Form  CREATE_CHANGE_DOCUMENTS
*&---------------------------------------------------------------------*
FORM create_change_documents.
********************* TO BE IMPLEMENTED LATER **********************
ENDFORM.                               " CREATE_CHANGE_DOCUMENTS
```

Lesson Summary

You should now be able to:

- Explain the meaning of the terms database LUW and SAP LUW
- Explain why you need to bundle changes to database tables in the client/server architecture of the R/3 System

Internal Use SAP Partner Only

Unit Summary

You should now be able to:

- Explain the meaning of the terms database LUW and SAP LUW
- Explain why you need to bundle changes to database tables in the client/server architecture of the R/3 System

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 3

SAP Locking Concept

Unit Overview

- Lock objects
- Lock Modules
- Using locks
- Monitoring

Unit Objectives

After completing this unit, you will be able to:

- Explain the role of lock objects
- why transactions in the SAP environment cannot use database locks as reliable locking mechanisms
- explain the idea behind the SAP locking concept
- Creating lock objects and lock modules.
- Set and release SAP locks by calling the appropriate lock function modules
- Use the various lock modes purposefully

Unit Contents

Lesson: Reasons for Using Locks

Lesson Overview

This lesson explains why locks are required and why transactions in the SAP environment cannot use database locks as reliable locking mechanisms. The idea behind the SAP locking concept will also be illustrated.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the role of lock objects
- why transactions in the SAP environment cannot use database locks as reliable locking mechanisms
- explain the idea behind the SAP locking concept

Business Example

You want to use locks in your change transactions to avoid competing data accesses.

Reasons for Using Locks

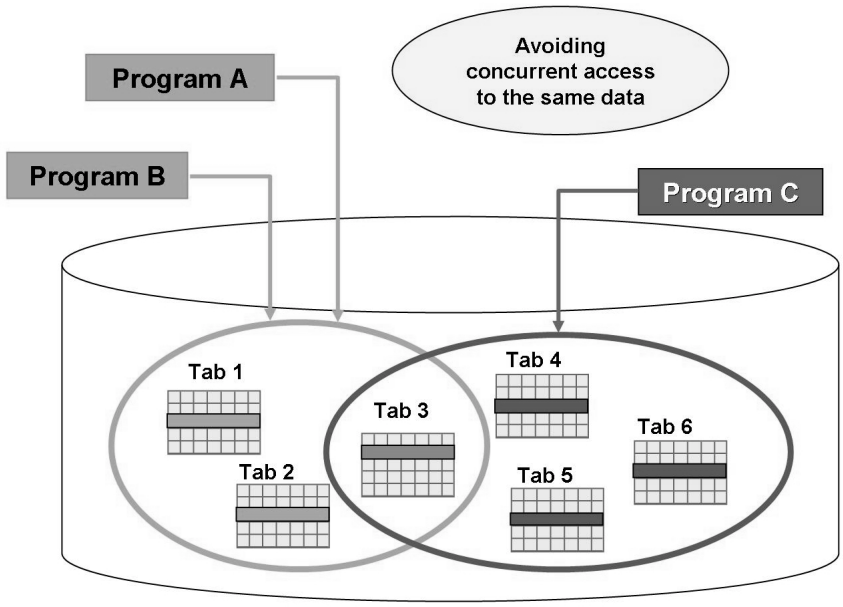


Figure 20: Why Set Locks?

If several users are competing to access the same resource or resources, you need to find a way of synchronizing the access in order to protect the consistency of your data.

Example: In a flight booking system, you would need to check whether seats are still free. You also need a guarantee that critical data (the number of free seats in this case) cannot be changed while you are working with the program.

Locks are a way of coordinating competing accesses to a resource. Each user requests a lock before accessing critical data.

It is important to release the lock as soon as possible so as not to hinder other users unnecessarily.

The SAP Locking Concept

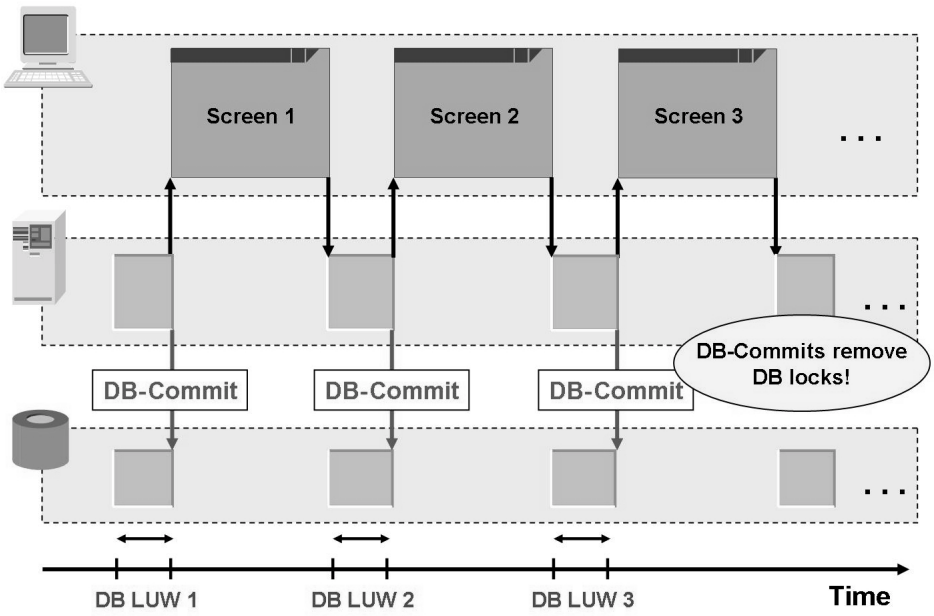


Figure 21: Database Locks Are Not Enough

The Database Management System (DBMS) physically locks the table lines that are **read with the intent of being changed** ("**SELECT SINGLE <f> FROM <dbtab> FOR UPDATE**") (database locks). Other users who want to access the locked record or records must wait until the physical lock has been released.

At the end of a database LUW, that is, with every DB commit, the database system releases all the locks set during the DB LUW.

In the R/3 System, this means that each database lock is released when a new screen is displayed, since a change of screen triggers an implicit database commit. Therefore, database locks are not sufficient if data is collected throughout several screens and the respective data records are to be kept locked during this time frame.

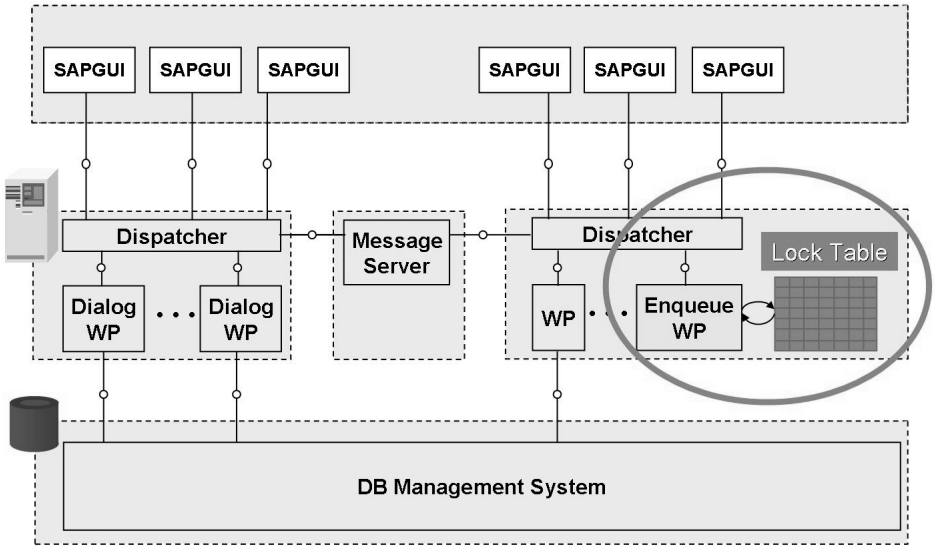


Figure 22: SAP Locking Concept Logical Locks

To keep a lock set through a series of screens (from the dialog program to the update program), the R/3 System has a **global lock table** at application server level, which you can use to set **logical locks** for table entries.

The **lock table** as well as the respective **enqueue work process** that manages the lock table are on a uniquely defined application server of the R/3 System. All the logical lock requirements of the R/3 System - irrespective of the application server where they are triggerred - run on the basis of this system-wide, unique work process.

You can also use logical locks to “lock” table entries that do not yet exist on the database. This is appropriate, for example, when you enter new table lines, and not possible with the help of database locks.

For further information, refer to the ABAP Editor keyword documentation for the term **locking mechanism**.

Lesson Summary

You should now be able to:

- Explain the role of lock objects
- why transactions in the SAP environment cannot use database locks as reliable locking mechanisms
- explain the idea behind the SAP locking concept

Lesson: Lock Objects and Lock Modules

Lesson Overview

This lesson explains the creation of SAP lock objects and lock modules that are used to set and cancel SAP locks.

Lesson Objectives

After completing this lesson, you will be able to:

- Creating lock objects and lock modules.

Business Example

You want to create lock objects and lock modules for setting and canceling SAP locks.

SAP Lock Objects

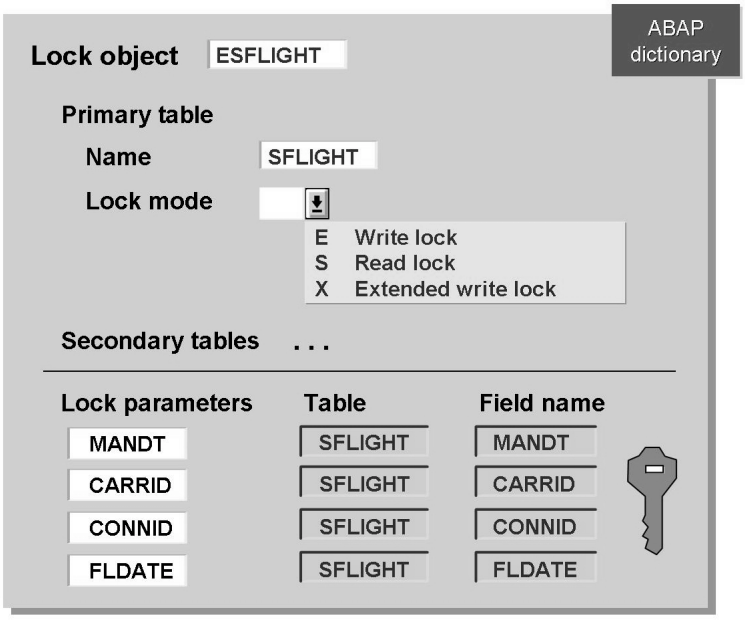


Figure 23: SAP Lock Objects

You set a logical lock in the lock table by calling a **lock module**. **Lock module**) created. This kind of lock module is a special, table-related function module that is created automatically when you activate a table-related **lock object**; when you call the lock module, logical locks are set for entries in the respective table(s).

Lock objects are maintained in the ABAP Dictionary. Customer lock objects must begin with “EY” or “EZ”. When you create a lock object, you only need to specify the table whose entries are to be locked later (primary or basis table). However, you can specify other tables that have a foreign-key relationship to the primary table (secondard tables). By doing so, you can lock an appropriate combination of table entries that belong together through the automatically created lock module.
Example: A lock object that contains the tables SFLIGHT (primary) and SBOOK (secondary) enables you to lock a flight with its bookings.

The lock module that is automatically created by the system contains, as input parameters, the lock parameters that are contained in the lock object (among other things). These lock parameters are used to communicate to the lock module which records are to have a logical lock set in the locking table. The system automatically proposes as names for the lock parameters those names contained in the table key fields (can be overwritten).

Also, when you are defininining a lock object, you can define the lock mode (“E”, “X” or “S”) for each specified table. This lock mode functions as a default import for the lock module and it can be overwritten. Details on the different lock modes are contained in a later section of this unit.

Generating Lock Modules

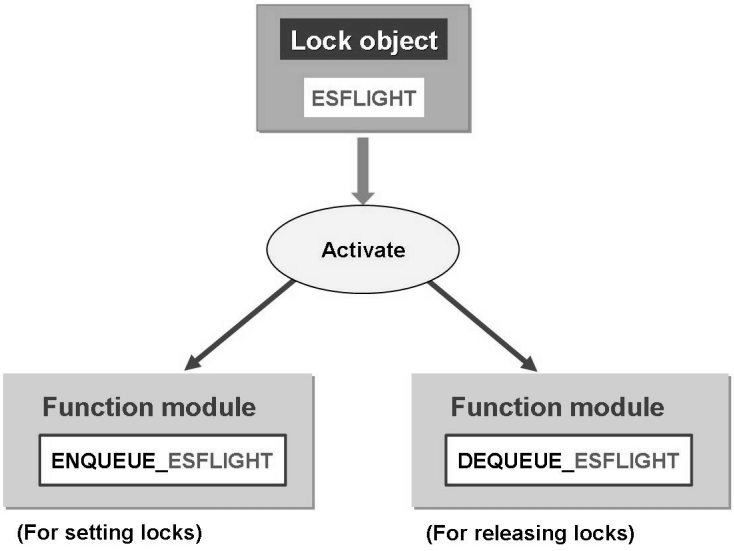


Figure 24: Generating Lock Modules

Whenever a lock object has been successfully activated, the system automatically generates one function module each for setting and releasing locks for entries in the table(s) specified in the lock object.

The function modules have the following name convention:
ENQUEUE_<Lock-Object-Name> or **DEQUEUE_<Lock-Object-Name>**

Lesson Summary

You should now be able to:

- Creating lock objects and lock modules.

Lesson: Setting and Releasing Locks

Lesson Overview

In this lesson, you will learn how to set and release SAP locks using the lock function modules. It also explains the meaning and effect of the various lock modes.

Lesson Objectives

After completing this lesson, you will be able to:

- Set and release SAP locks by calling the appropriate lock function modules
- Use the various lock modes purposefully

Business Example

You want to set and release SAP locks in a purposeful way by calling the appropriate lock function modules

Setting, Removing and Administering SAP Locks

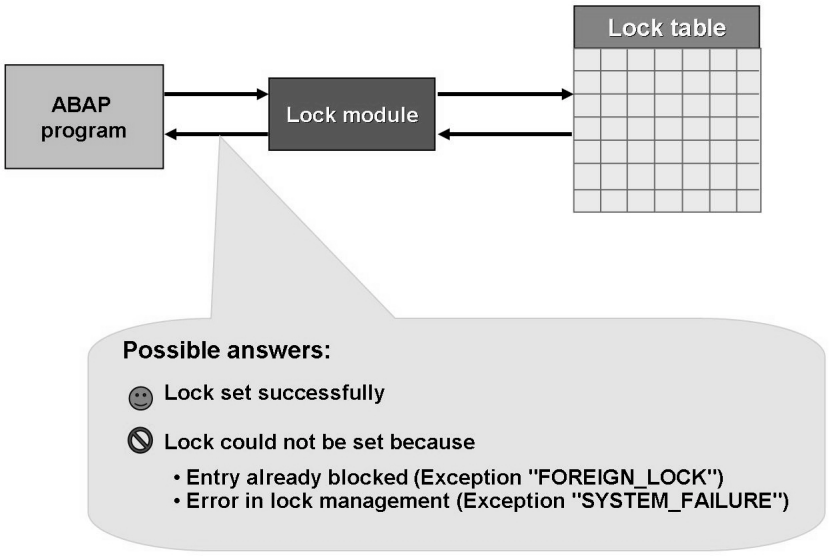


Figure 25: Setting and Releasing Logical Locks

A logical lock is set when you call the lock module; this writes a corresponding entry into the lock table.

You can only set a lock if no lock entries exist in the lock table for the respective table records.

If there is an error, the lock module triggers appropriate exceptions. For this reason, the application program can determine the success or failure of the lock on the basis of the return code delivered by the lock module (see the specifications on the slide), and react accordingly. If there is an error, for example, the current user receives an error message stating that he or she has been rejected by the system.

Depending on the bundle technique used for database updates (see the unit *Organizing Database Changes*), an application program may need to delete again the lock entries it has created (during inline update) or have them deleted automatically (during the update task).

If an application program that has created lock entries is terminated, the locks are released automatically (implicitly). Program termination takes place, for example, if you have messages of the type “A” / “X”, if you have the statements LEAVE PROGRAM and LEAVE TO TRANSACTION , or if the user enters “/n” in the command field..

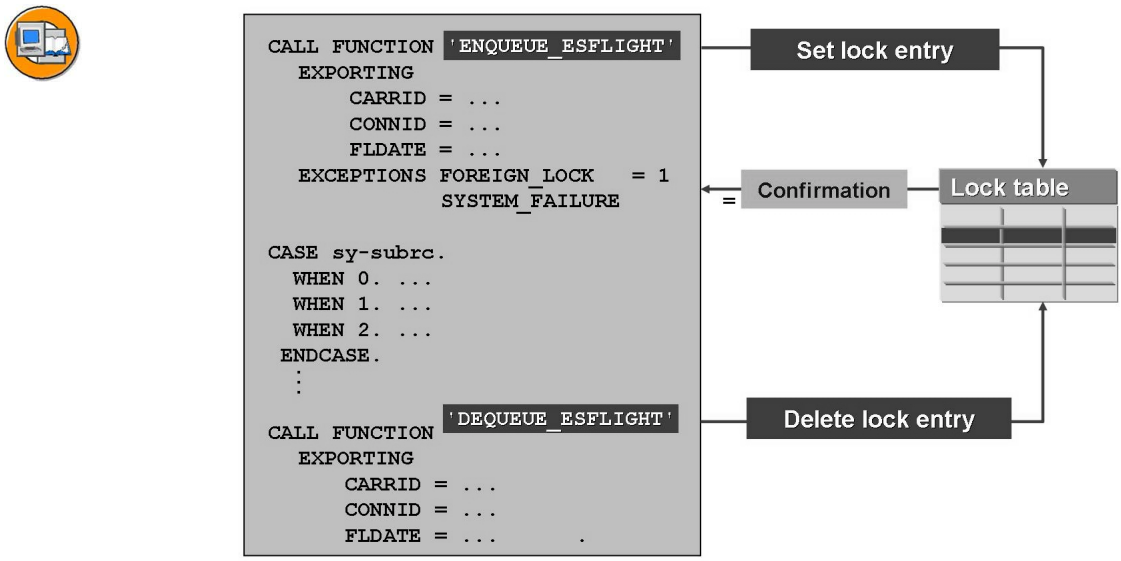


Figure 26: Calling the Lock Modules

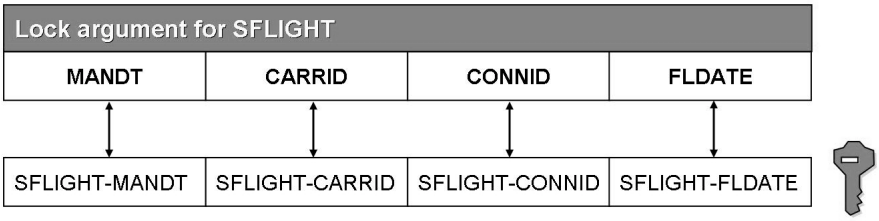
Using the import parameters of an ENQUEUE function module that correspond to the key fields in the respective table, you determine the table lines to be blocked, that is, you find out for which table lines a lock entry is to be written to the lock table. These import parameters are called **lock parameters**.

If the system could not set the lock successfully (sy-subrc not 0), you should issue a corresponding dialog message to the current user.

At the end of the dialog program, you can use the corresponding DEQUEUE function module to delete the entries from the lock table.

DEQUEUE function modules do not trigger any exceptions. If you attempt to release an entry that is not locked, this has no effect.

If you want to release all of the locks that you have set, at the end of your dialog program, for example, you can use the function module DEQUEUE_ALL.



Examples for table SFLIGHT:

Call parameters	Lock argument	Lock effect
`800 LH 0400 20021224`	`800LH 040020021224`	Flight "LH 400 on 12/24/2002" in client 800
`800 LH 0400`	`800LH 0400########`	All flights for "LH 400" in client 800
`800 LH`	`800LH ############`	All flights for LH in client 800

Figure 27: Lock argument

The core part of a lock entry is the lock argument that consists of the **lock parameters** (key fields of the respective table) and the key of the table entry(entries) to be locked.

If, when you call a lock module, a lock parameter is set to its “initial” value or not specified at all, the system will interpret this as a generic value, that is, it will interpret the lock as referring to all table lines to which the other parameter variants apply. The client parameter is an exception. If, on the lock module call

- it is not specified then the lock only applies to the current execution client (SY-MANDT)
- specified with a concrete client then the call applies to this client only
- specified with SPACE then the lock applies to all clients.

SM12

Client	User	Time	Shared	Table	Lock argument
800	Meier	11:00		SFLIGHT	800LH 240720021224
800	Wang	11:00		SFLIGHT	800UA 3504########
800	Mueller	11:01		LFB1	800000047120815
800	Mueller	11:01		LFC1	8000000471208152001
800	Jones	11:02	X	YLFA	800LIEF1
800	Smith	11:03	X	YLFA	800LIEF1
800	Bauer	11:04		YLFB	80012345

Figure 28: Setting Up and Managing the Lock Table

To display the lock table, use transaction SM12.

Parameters	Values	Meaning
mode_< TabName >	'E' 'X' ' S '	Cumulative write lock Non-cumulative write lock Cumulative read lock
<Lock parameters>	<val>	<Lock parameter><Value> Field value to be used for locking
x_<Lock parameters>	SPACE 'X'	Lock acc. to corresponding lock parameter (Default) Lock for table line with initial field value
_scope	' 1 ' ' 2 ' ' 3 '	Lock remains in program Lock passed to V1 update (default) Lock in program + passed to V1 update
_wait	SPACE 'X'	If external lock, no further lock attempt (default) If external lock, second lock attempt
_collect	SPACE 'X'	Setting lock without local lock container (default) Setting lock with local lock container

Figure 29: Parameters in ENQUEUE Module

The *mode_<TabName>* parameter overrides the default lock mode of the lock module that is specified in the lock object.

Using the parameter *x_<lock-parameter>* you can lock the table line that has an initial value in the corresponding lock parameter field. (See comment for figure "Lock argument")

The parameter _*scope* defines the validity area of the lock :

- "1": The lock remains set in the program and is deleted again by the program. (Use for inline updates)
- "2": The lock is passed to the update program (default) (Usage for udpates through update program; see the unit "*Organizing Database Updates*").
- "3": Two locks are set. One stays set in the program and must be deleted from there. The other one is passed to the update program. These functions are required whenever the update task is triggered for a partial update of a record, but the corresponding total record should still remain locked in the program for further partial updates.

The _*wait* parameter defines whether a lock request should be repeated in case the first lock attempt fails. You can configure the retry interval by setting the profile parameter (ENQUE/DELAY_MAX).

You use the parameter _*collect* to store the lock request in the local lock container until it is collected and passed later on to the enqueue server. (Refer also to the figure "Usage of the Lock Container")

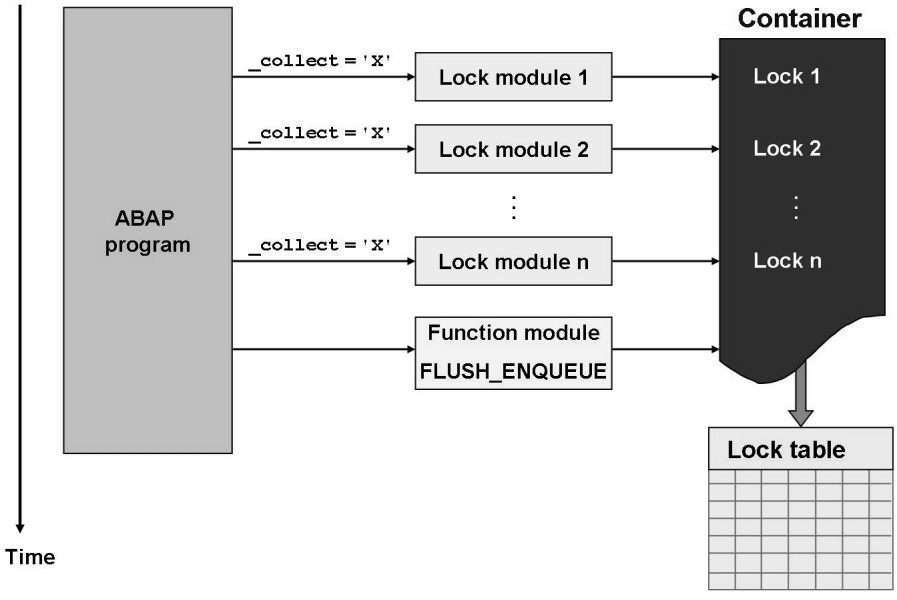


Figure 30: Using the Lock Container

Requesting a lock from a program is a communication step with lock administration. If your program sets locks for several objects, this communication effort occurs more than once.

If you use the **lock container**, you can reduce the technical effort required for this step. To do this, set the parameter _collect = "X" whenever you call the lock module. This has the effect that the respective lock requests are stored in the local container for subsequent collective dispatch.

You can send the contents of the lock container afterwards to the lock management using the function module FLUSH_ENQUEUE.

If you are able to successfully issue all the lock requests, the system will delete the entire content of the lock container. If one of the locks in a container cannot be set, the function module FLUSH_ENQUEUE triggers the exception FOREIGN_LOCK. In this case, none of the locks registered in the container will be set and the container content will remain complete for further dispatch attempts.

You can delete the contents of the lock container using the function module RESET_ENQUEUE.

Use and Effect of the Lock Modes

Lock mode	Meaning
E (Extensible)	Lock for data change (accumulative exclusive lock)
X (Exclusive)	Lock for data change (exclusive write lock)
S (Shared)	Lock for protected data display (shared lock)

Figure 31: Lock mode

If you set a lock when you call the respective lock module, you can specify the **lock mode**, which determines the type and purpose of the lock. If you make no specification, the default in the definition of the respective lock object applies.

There are three different lock modes:

- **Mode “E”**: This sets a lock for changing data. This lock can be accumulated (see following figures). Example: You wish to book a flight. Once you have chosen the flight you want to book, you should make sure that no other customer books the same flight so as to prevent the last free seat from being occupied more than once. To do this, you must lock the respective flight (SFLIGHT entry) with mode “E”.
- **Mode “X”**: This mode is used like mode “E” for changing data. The only difference to mode “E” from a technical viewpoint is that the respective lock does not allow accumulation (for details, see the following figures).
- **Mode “S”**: You implement this mode if you wish to ensure that data you display in your program cannot be changed by other users during the entire display time. Here you do not wish to change the data yourself. Example: Your program has determined a price for a flight and has displayed this price to an interested customer. While the customer is considering whether or not to book the flight, you want to ensure that the price will not change.

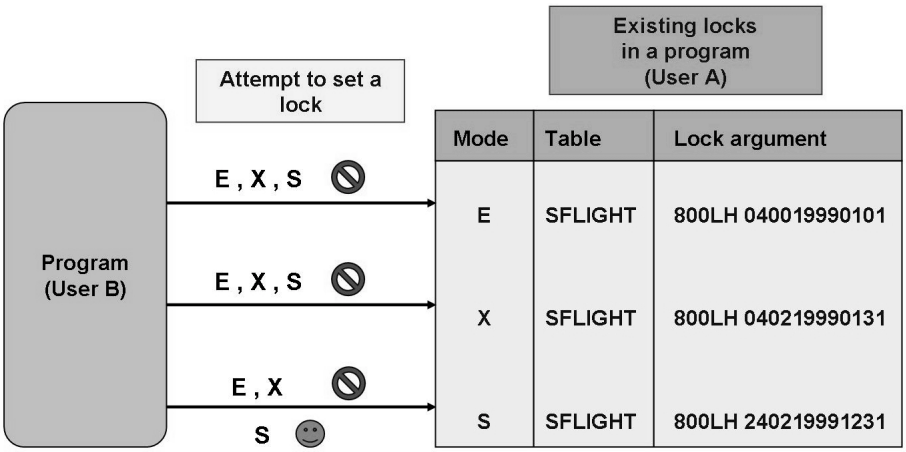


Figure 32: Effect of Lock Modes (View: Other User)

If you have existing locks in the system, attempts by outsiders to set locks (**lock request by the program of another user**) in the same data record are treated as follows:

- Existing exclusive locks (“E” or “X”) categorically reject every lock attempt of another user, irrespective of the mode in which the other user has attempted to set the lock.
- An existing shared lock (“S”) , on the other hand, allows other shared locks for protected display to be set for one and the same data record. Attempts by other users to set shared locks for the same data record will, of course, be rejected.

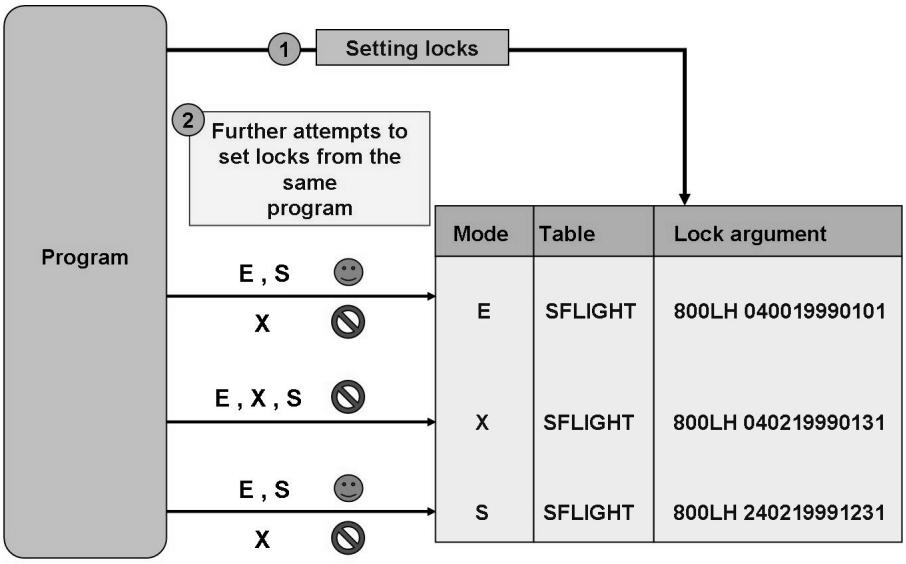


Figure 33: Effect of Lock Modes (View: Same Program)

Lock Requests from Same Program:

If you attempt to lock a data record more than once while a program is running (for example, using a function module that you call and this module itself sets locks), the lock system reacts in the following way:

- If you have an existing “E” lock, further “E” and “S” locks for the same data record will be accepted. Only attempts to set an “X” lock for the same data record will be rejected.
- If you have an existing “X” lock, every further attempt to set a lock will be rejected.
- If you have an existing “S” lock, further “S” locks for the same data record can be set from within the same program. If, in addition, there are no further shared locks set by another user for this data record, you can also set an additional “E” lock. Naturally, no “X” lock can be set if there is an existing “S” lock.

Setting and Releasing Locks (Timecourse)

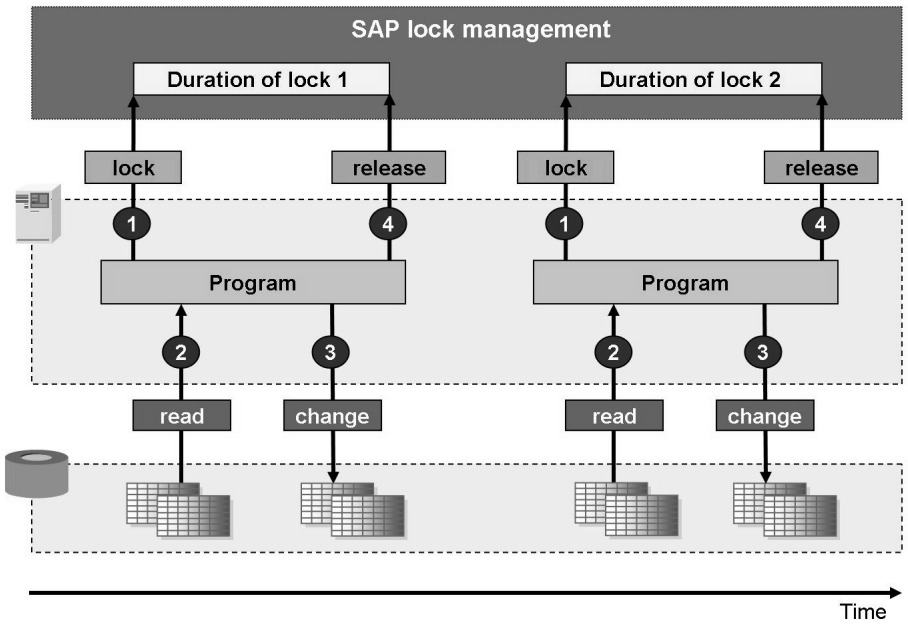


Figure 34: Setting and Releasing Locks (Timecourse)

If you want to ensure that you are **reading up-to-date data** in your program (with the intention of changing the data and saving it to the database), perform the following steps in your program in the order given. These steps pertain to lock requests and database accesses:

- Setting locks for the data to be processed.
- If the lock has been successfully set, read the current data from the database.
- Change the program data (user inputs) and update the changes to the database.
- Release set locks once again.

With this order of steps, you ensure not only that your changes run completely under the protection of locks, but also that you only read data hat has been changed consistently by other programs. This assumes, however, that all the application programs use the SAP lock concept and adhere to the step sequence given.

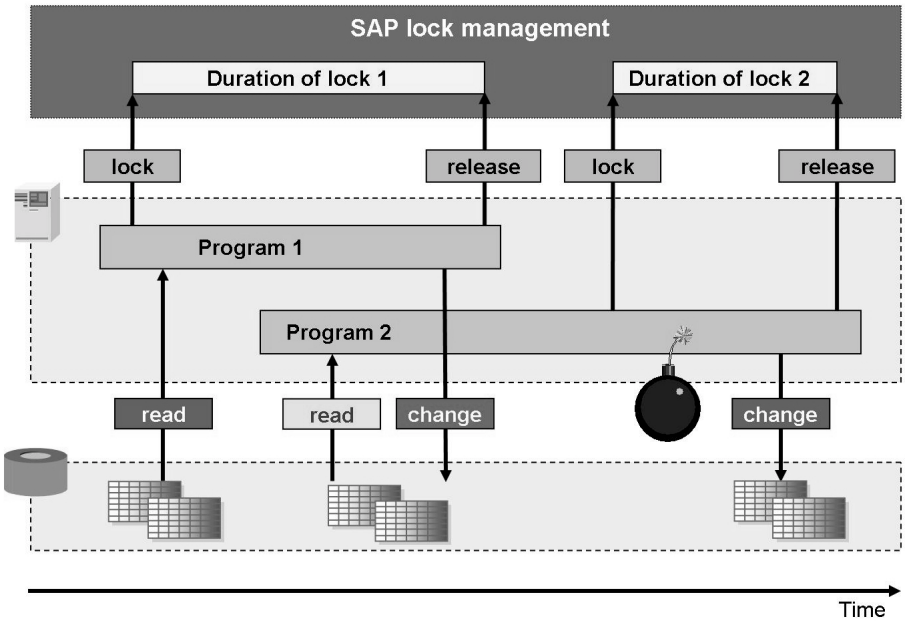


Figure 35: Danger If Locks Used Incorrectly

If you do not adhere to the sequence "*Lock → Read → Change → Unlock*", there is the danger that your program will read data from the database that is currently locked by another program and is also being processed (see figure above). In this case, this could also mean that, even if the lock is successfully set after the read action, the data read by your program and displayed to the user for change is already out of date. This therefore makes it possible for a user of your program to enter changes for data that is no longer up to date. For this reason, you should always follow the procedure described here.

Exercise 4: SAP Locking Concept

Exercise Objectives

After completing this exercise, you will be able to:

- Call and use lock modules.
- Locate the places in programs where locks must be set and released in order to ensure that the data to be changed is protected adequately against competing accesses.

Business Example

The program **SAPMZ##_BOOKINGS1** from the previous unit is to be changed to include locks that will prevent the booking data from being canceled and the flight data from being changed by other programs at the same time.

Task 1: Setting SAP Locks

Program: SAPMZ##_BOOKINGS2

Transaction code: Z##_BOOKINGS2

Copy from: SAPBC414**T**_BOOKINGS_02

Model Solution: SAPBC414**S**_BOOKINGS_02

1. Copy your solution **SAPMZ##_BOOKINGS1** or the program template **SAPBC414T_BOOKINGS_02** with **all** subobjects to **SAPMZ##_BOOKINGS2 (##** is your group number).

   Assign transaction code **Z##_BOOKINGS2** to the program.

2. Call the lock modules **ENQUEUE_ESFLIGHT, ENQUEUE_ESBOOK, ENQUEUE_ESFLIGHT_SBOOK** and **DEQUEUE_ALL** in subroutines. The subroutines in question are already created (blank) and combined in the Include **MZ##_BOOKINGS2F02**. To supply the interface parameters of the lock modules with data, use the fields of the structures SDYN_CONN and SDYN_BOOK because these are placed in screens 100 and 301, respectively, as input fields.

3. Provide solutions for the exceptions of the lock modules.

   Possible user messages:

   Data record is already being edited → message 060

Continued on next page

Processing terminated (booking already locked) → message 061

Flight and/or bookings are already being edited → message 062

Lock request unsuccessful → message 063

4. Protect the database changes related to the **booking cancelations** by calling the corresponding lock modules (or subroutines). If a user action calls up screen 100, the locks must be canceled.

Hint: The lock module ENQUEUE_ESFLIGHT enables locks to be set for entries in table SFLIGHT. The lock module ENQUEUE_ESBOOK enables locks to be set for entries in table SBOOK. The lock module ENQUEUE_ESFLIGHT_SBOOK enables locks to be set in both tables at the same time (SFLIGHT, SBOOK) (reason: to lock a flight with all the bookings belonging to it).

Task 2: (optional)

Extend your program for creating a new customer to include the necessary lock module calls.

1. Copy your solution **SAPMZ##_CUSTOMER1** or the program template **SAPBC414T_CREATE_CUSTOMER_02** with **all** subobjects to **SAPMZ##_CUSTOMER2** (## is your group number).

   Assign transaction code **Z##_CUSTOMER2** to the program.

2. The calls ENQUEUE_ESCUSTOM (lock customer) and DEQUEUE_ALL (remove all locks) are already coded and encapsulated in the subroutines ENQ_SCUSTOM and DEQ_ALL (Include BC414T_CREATE_CUSTOMER_02F01).

   Insert the call for the subroutines **ENQ_SCUSTOM** and **DEQ_ALL** at the appropriate places in your program. When should the customer data record be locked? Locate all the places where the data record lock must be canceled. Familiarize yourself with the program flow, using the debugger if necessary.

Task 3: (optional)

Finding Existing Lock Objects

1. Find out which lock objects / lock modules for logically locking flights, bookings, and flights with all corresponding bookings already exist in the system.

Solution 4: SAP Locking Concept

Task 1: Setting SAP Locks

Program: SAPMZ##_BOOKINGS2

Transaction code: Z##_BOOKINGS2

Copy from: SAPBC414**T**_BOOKINGS_02

Model Solution: SAPBC414**S**_BOOKINGS_02

1. Copy your solution **SAPMZ##_BOOKINGS1** or the program template **SAPBC414T_BOOKINGS_02** with **all** subobjects to **SAPMZ##_BOOKINGS2 (##** is your group number).

   Assign transaction code **Z##_BOOKINGS2** to the program.

   a) -

2. Call the lock modules **ENQUEUE_ESFLIGHT, ENQUEUE_ESBOOK, ENQUEUE_ESFLIGHT_SBOOK** and **DEQUEUE_ALL** in subroutines. The subroutines in question are already created (blank) and combined in the Include **MZ##_BOOKINGS2F02**. To supply the interface parameters of the lock modules with data, use the fields of the structures SDYN_CONN and SDYN_BOOK because these are placed in screens 100 and 301, respectively, as input fields.

   a) See model solution

3. Provide solutions for the exceptions of the lock modules.

   Possible user messages:

   Data record is already being edited → message 060

   Processing terminated (booking already locked) → message 061

   Flight and/or bookings are already being edited → message 062

   Lock request unsuccessful → message 063

   a) See model solution

Continued on next page

4. Protect the database changes related to the **booking cancelations** by calling the corresponding lock modules (or subroutines). If a user action calls up screen 100, the locks must be canceled.

Hint: The lock module ENQUEUE_ESFLIGHT enables locks to be set for entries in table SFLIGHT. The lock module ENQUEUE_ESBOOK enables locks to be set for entries in table SBOOK. The lock module ENQUEUE_ESFLIGHT_SBOOK enables locks to be set in both tables at the same time (SFLIGHT, SBOOK) (reason: to lock a flight with all the bookings belonging to it).

a) See model solution

Task 2: (optional)

Extend your program for creating a new customer to include the necessary lock module calls.

1. Copy your solution **SAPMZ##_CUSTOMER1** or the program template **SAPBC414T_CREATE_CUSTOMER_02** with **all** subobjects to **SAPMZ##_CUSTOMER2** (## is your group number).

   Assign transaction code **Z##_CUSTOMER2** to the program.

   a) -

2. The calls ENQUEUE_ESCUSTOM (lock customer) and DEQUEUE_ALL (remove all locks) are already coded and encapsulated in the subroutines ENQ_SCUSTOM and DEQ_ALL (Include BC414T_CREATE_CUSTOMER_02F01).

   Insert the call for the subroutines **ENQ_SCUSTOM** and **DEQ_ALL** at the appropriate places in your program. When should the customer data record be locked? Locate all the places where the data record lock must be canceled. Familiarize yourself with the program flow, using the debugger if necessary.

   a) See model solution

Continued on next page

Internal Use SAP Partner Only

Task 3: (optional)

Finding Existing Lock Objects

1. Find out which lock objects / lock modules for logically locking flights, bookings, and flights with all corresponding bookings already exist in the system.

   a) Display the transparent tables SFLIGHT (flights) and SBOOK (bookings). Use the where-used list to determine the use of the relevant table in lock objects.

   Alternative: Perform a search in the Repository Infosystem (SE84) for lock objects with either the SFLIGHT or SBOOK basis table.

Result

Model Solution SAPBC414S_BOOKINGS_02

PAI Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_02I01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&---------------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      CASE sy-dynnr.
        WHEN '0100'.
          LEAVE PROGRAM.
        WHEN '0200'.
* remove all database locks
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN '0300'.
          LEAVE TO SCREEN '0100'.
        WHEN OTHERS.
      ENDCASE.
    WHEN 'EXIT'.
      LEAVE PROGRAM.
    WHEN OTHERS.
  ENDCASE.
```

Continued on next page


```
ENDMODULE.                              " EXIT  INPUT


*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
*****************************CANCEL BOOKING*************************
    WHEN 'BOOKC'.
* set database lock for selected flight and depending bookings
      PERFORM enq_sflight_sbook.
      PERFORM read_sflight USING wa_sflight sysubrc.
      PERFORM process_sysubrc_bookc.
      PERFORM read_spfli USING wa_spfli.
      PERFORM read_sbook USING itab_book itab_cd.
      REFRESH CONTROL 'TC_SBOOK' FROM SCREEN '0200'.
*****************************CREATE BOOKING*************************
    WHEN 'BOOKN'.
      PERFORM read_sflight USING wa_sflight sysubrc.
      PERFORM process_sysubrc_bookn.
      PERFORM read_spfli USING wa_spfli.
      PERFORM initialize_sbook USING wa_sbook.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN '0100'.
  ENDCASE.
ENDMODULE.                              " USER_COMMAND_0100  INPUT


*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0200  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0200 INPUT.
  CASE save_ok.
    WHEN 'SAVE'.
      PERFORM collect_modified_data USING itab_sbook_modify.
      PERFORM save_modified_booking.
* remove all database locks
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN 'BACK'.
* remove all database locks
      PERFORM deq_all.
```

Continued on next page

```
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0200'.
  ENDCASE.
ENDMODULE.                              " USER_COMMAND_0200  INPUT
```

FORM Routines

F01

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_02F01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKC
*&---------------------------------------------------------------------*
FORM process_sysubrc_bookc.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0200'.
    WHEN OTHERS.
* remove all database locks
      PERFORM deq_all.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                                " PROCESS_SYSUBRC_BOOKC
```

F02

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_02F02
*---------------------------------------------------------------------*

*---------------------------------------------------------------------*
*   FORM ENQ_SFLIGHT
*---------------------------------------------------------------------*
FORM enq_sflight.
  CALL FUNCTION 'ENQUEUE_ESFLIGHT'
       EXPORTING
            carrid         = sdyn_conn-carrid
```

Continued on next page


```
            connid         = sdyn_conn-connid
            fldate         = sdyn_conn-fldate
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e060.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               "ENQ_SFLIGHT

*---------------------------------------------------------------------*
*   FORM ENQ_SBOOK
*---------------------------------------------------------------------*
FORM enq_sbook.
  CALL FUNCTION 'ENQUEUE_ESBOOK'
       EXPORTING
            carrid         = sdyn_book-carrid
            connid         = sdyn_book-connid
            fldate         = sdyn_book-fldate
            bookid         = sdyn_book-bookid
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e061.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               "ENQ_SBOOK


*---------------------------------------------------------------------*
*   FORM ENQ_SFLIGHT_SBOOK
*---------------------------------------------------------------------*
FORM enq_sflight_sbook.
  CALL FUNCTION 'ENQUEUE_ESFLIGHT_SBOOK'
       EXPORTING
```

Continued on next page

```
            carrid         = sdyn_conn-carrid
            connid         = sdyn_conn-connid
            fldate         = sdyn_conn-fldate
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e062.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                                "ENQ_SFLIGHT_SBOOK


*---------------------------------------------------------------------
*   FORM DEQ_ALL
*---------------------------------------------------------------------*
FORM deq_all.
  CALL FUNCTION 'DEQUEUE_ALL'.
ENDFORM.                                "DEQ_ALL
```

F03

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_02F03
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  READ_SPFLI
*&---------------------------------------------------------------------*
*       -->P_WA_SPFLI  text
*----------------------------------------------------------------------*
FORM read_spfli USING p_wa_spfli TYPE spfli.
  SELECT SINGLE * FROM spfli INTO p_wa_spfli
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid.
  IF sy-subrc <> 0.
* remove all database locks
    PERFORM deq_all.
    MESSAGE e022 WITH sdyn_conn-carrid sdyn_conn-connid.
  ENDIF.
```

Continued on next page


```
ENDFORM.                                " READ_SPFLI
```

Solutions for the optional exercise:

Model Solution SAPBC414S_CREATE_CUSTOMER_02

FORM Routines

F01

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMER_02F01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  SAVE
*&---------------------------------------------------------------------*
FORM save.
PERFORM number_get_next USING scustom.
* lock dataset
  PERFORM enq_scustom.
PERFORM save_scustom.
* unlock dataset
  PERFORM deq_all.
ENDFORM.    " SAVE
```

Lesson Summary

You should now be able to:

- Set and release SAP locks by calling the appropriate lock function modules
- Use the various lock modes purposefully

Internal Use SAP Partner Only

Unit Summary

You should now be able to:

- Explain the role of lock objects
- why transactions in the SAP environment cannot use database locks as reliable locking mechanisms
- explain the idea behind the SAP locking concept
- Creating lock objects and lock modules.
- Set and release SAP locks by calling the appropriate lock function modules
- Use the various lock modes purposefully

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 4

Organizing Database Updates

Unit Overview

- Database changes from within the application program
  - direct
  - using Delayed Subroutines
- Database changes using update techniques
  - Asynchronous, Synchronous, and Local Updates
  - V1 and V2 Updates

Unit Objectives

After completing this unit, you will be able to:

- Executing database updates directly from application programs or by using delayed subroutines.
- Perform database changes using various update techniques
- Use and create update modules
- Implement update types V1 and V2
- Implement the SAP locking concept in accordance with the selected change type

Unit Contents

Lesson: Database changes from within the application program

Lesson Overview

- Database updates from within the application program
  - direct
  - using Delayed Subroutines

Lesson Objectives

After completing this lesson, you will be able to:

- Executing database updates directly from application programs or by using delayed subroutines.

Business Example

You want to execute database updates directly from application programs or by using delayed subroutines.

Direct Updates from Within the Program

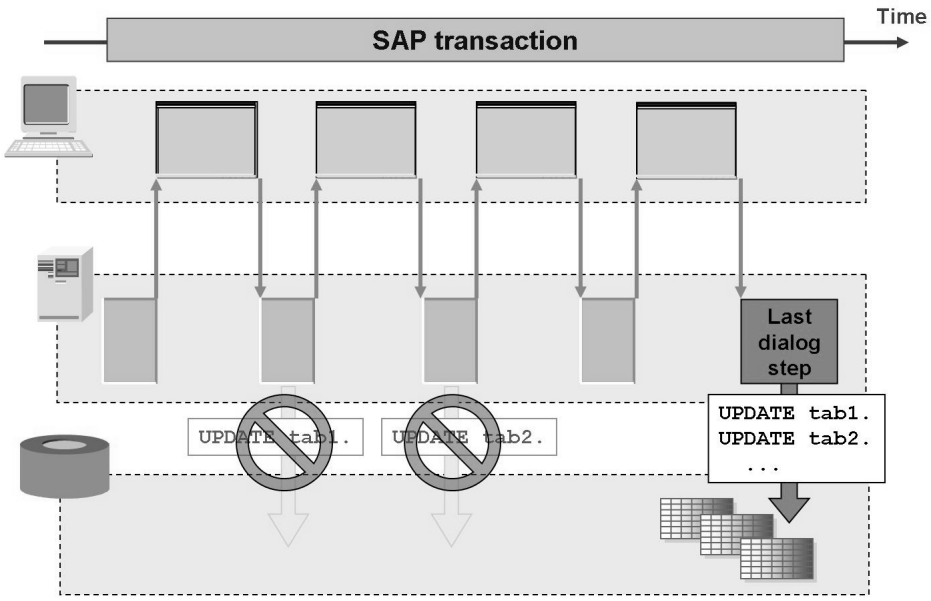


Figure 36: Timescale

If your transaction executes database updates from within the dialog program (**inline updates**), you must bundle all of your database updates into a single dialog step (usually the last). This is the only way to ensure that your database changes are processed using the "all or nothing" principle.

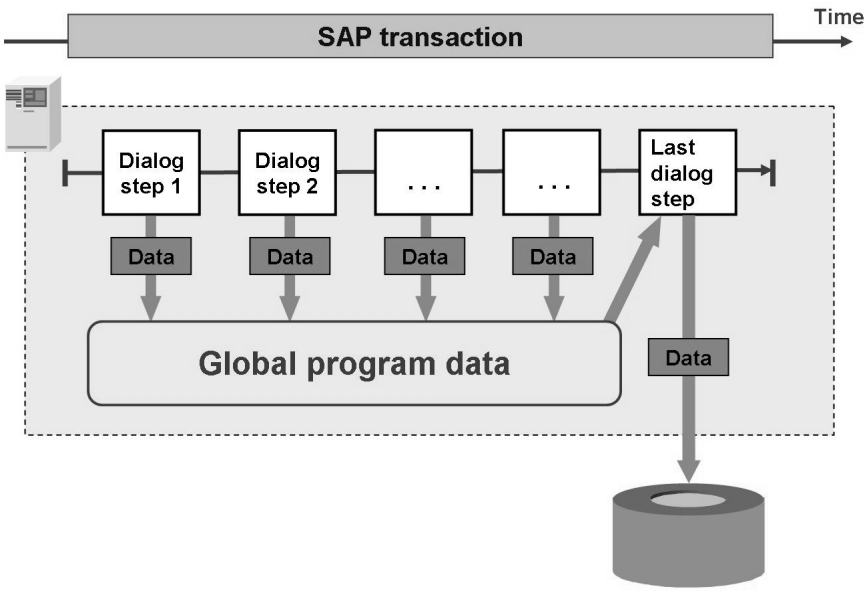


Figure 37: Data flow

If you have updates executed from the dialog program, you must save the data you want to change in the global program data until the database changes are made. This data is written to the database with the status it had for the last dialog step.

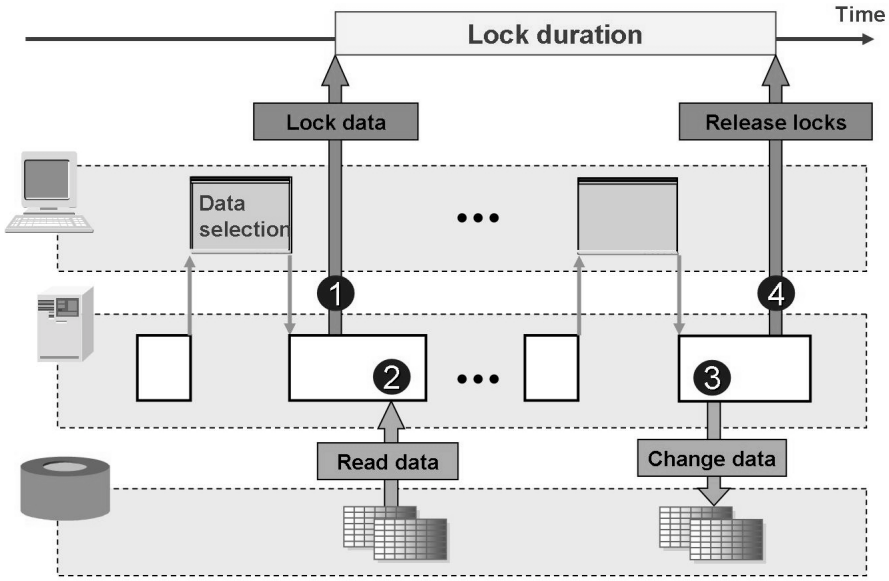


Figure 38: Lock

If you update the database directly from within the program, your program must set and release SAP locks itself. Use the specified sequence of steps (see the unit "SAP Locking Concept"):

1. Lock data
2. Read data (+ display for change)
3. Change data in database
4. Unlock data

Hint: Remember that your program must delete the lock entries itself. For this purpose you can execute either the unlock module DEQUEUE_<*Sperrobject*> , which belongs to the lock object, or the general unlock function module DEQUEUE_ALL.

Using Delayed Subroutines for Database Updates

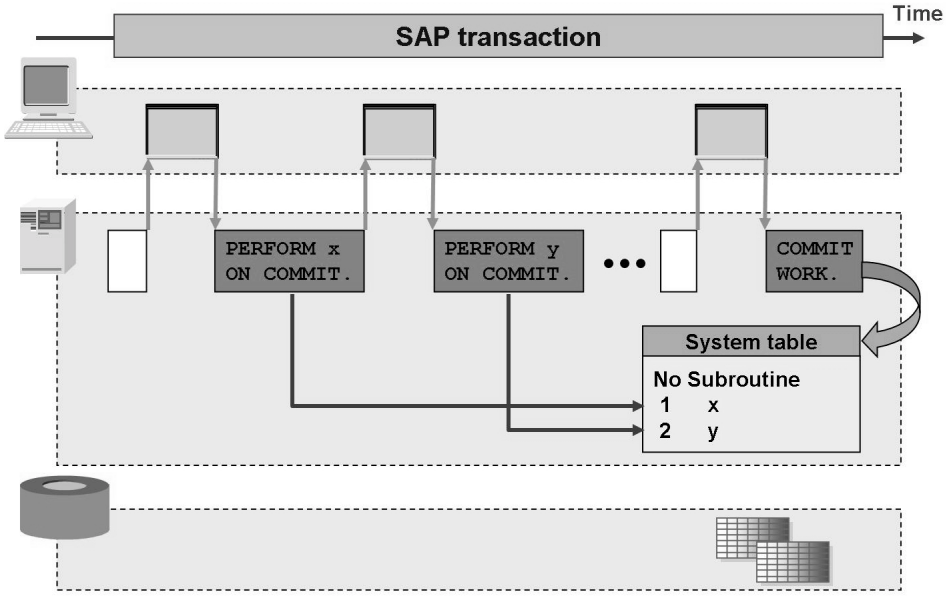


Figure 39: PERFORM ON COMMIT: Timescale (1)

Database updates from dialog mode can be executed in bundled form by using the special subroutine technique `PERFORM <subroutine> ON COMMIT`.

The `PERFORM <subroutine> ON COMMIT` registers the specified subroutine for execution. This will not be executed until the system reaches the next `COMMIT WORK` statement.

If the database updates are “encapsulated” in the subroutines, they can be separated from the program logic and relocated to the end of the LUW processing.

Each subroutine registered with PERFORM ON COMMIT is only executed once per LUW. Calls can be made more than once (no errors); the subroutine, however, is only executed once.

Nested PERFORM ON COMMIT calls after release 4.6 trigger a runtime error.

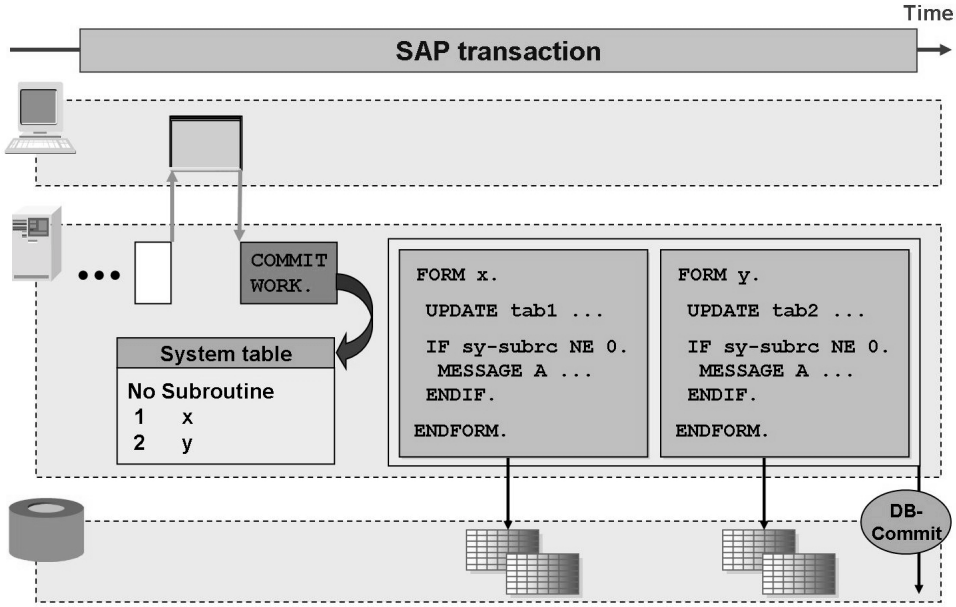


Figure 40: PERFORM ON COMMIT: Timescale (2)

The `COMMIT WORK` statement carries out all subroutines registered to be executed, one after the other, and triggers a database commit.

If there is an error, you can terminate processing from the respective subroutine with a type A dialog message and the previous consistent database status can be restored.

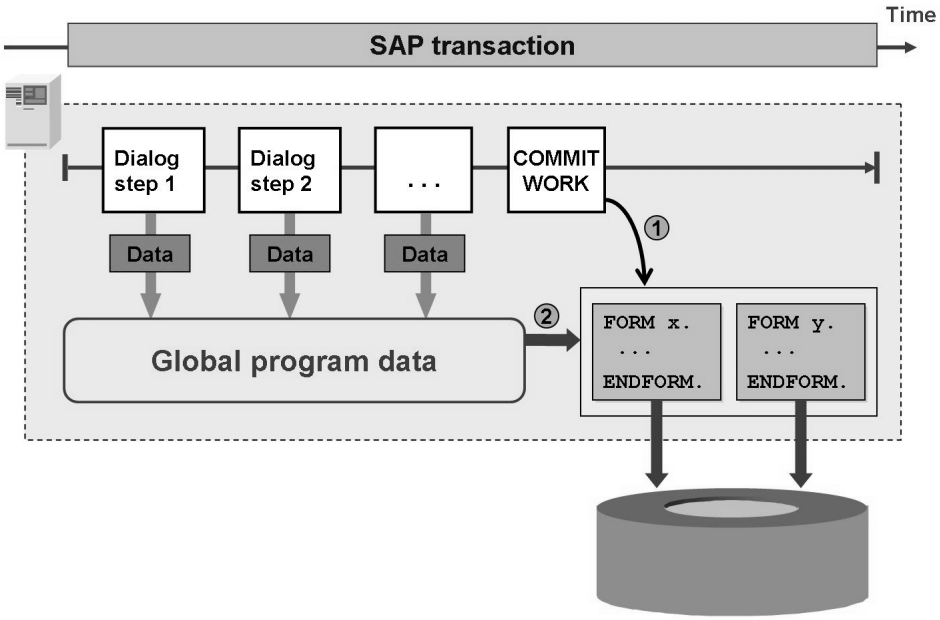


Figure 41: PERFORM ON COMMIT: Data flow

Subroutines called using “PERFORM ON COMMIT” must have no interface. They work instead with **global data**, that is, with the values the data objects contain **at the point when the subroutine is actually run**.

The PERFORM ON COMMIT technique can also be used in the update. This will be discussed later.

Hint: For further information, refer to the ABAP Editor keyword documentation for the term **PERFORM**.

Lesson Summary

You should now be able to:

- Executing database updates directly from application programs or by using delayed subroutines.

Lesson: Database changes using update techniques

Lesson Overview

Update:

- Process
- Technical Implementation
- Update Modes
- V1 and V2 Updates
- Optimization Notes for Database Changes

Lesson Objectives

After completing this lesson, you will be able to:

- Perform database changes using various update techniques
- Use and create update modules
- Implement update types V1 and V2
- Implement the SAP locking concept in accordance with the selected change type

Business Example

You want to execute data base updates (with respect to performance) using update techniques.

Summary: Updates from the Application Program

Summary

Changes from within the application program:

- "Simple" concept, implemented quickly

But:

- User has to wait until the changes have been made
- Dialog work process is not released
- No error logging
- Several DB work processes working parallel => poor DB performance
- No support through system functions if LUW processing and use of locking concept (must all be implemented in the program)

Therefore usage only for:

- "Light" LUWs, that is, LUWs containing only single record updates or updates that are not critical for performance

Figure 42: Summary: Updates from the Application Program

Principle and Process Flow of the Update

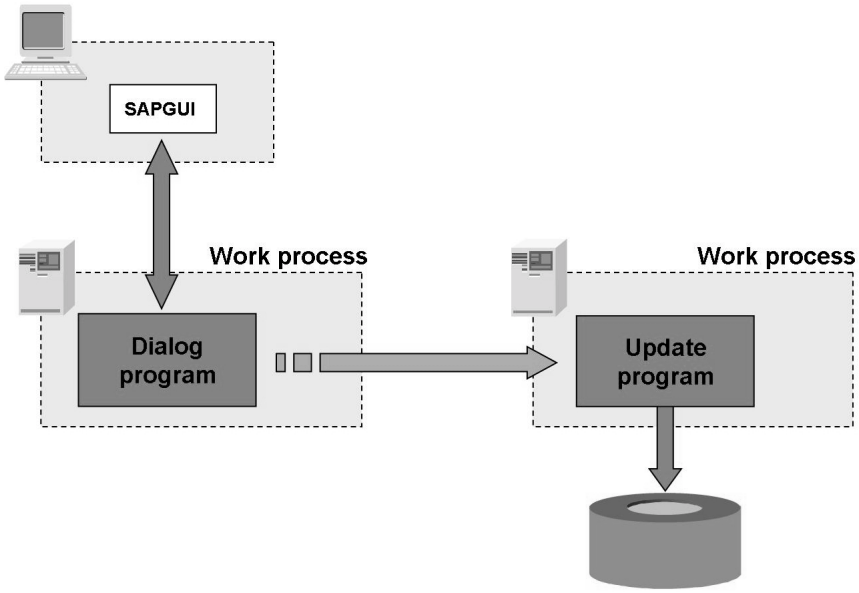


Figure 43: Update Principle

Update techniques allow you to separate user dialogs from the database changes. Both are executed by different programs, which generally run in different work processes.

The program that the user uses to execute dialogs is called a dialog program. It accepts user entries and, at the end, when it passes on the changed data, triggers an update program that updates the corresponding data in the database.

No dialogs run in the update programs.

The following figures describe the various steps in a program that uses an update technique.

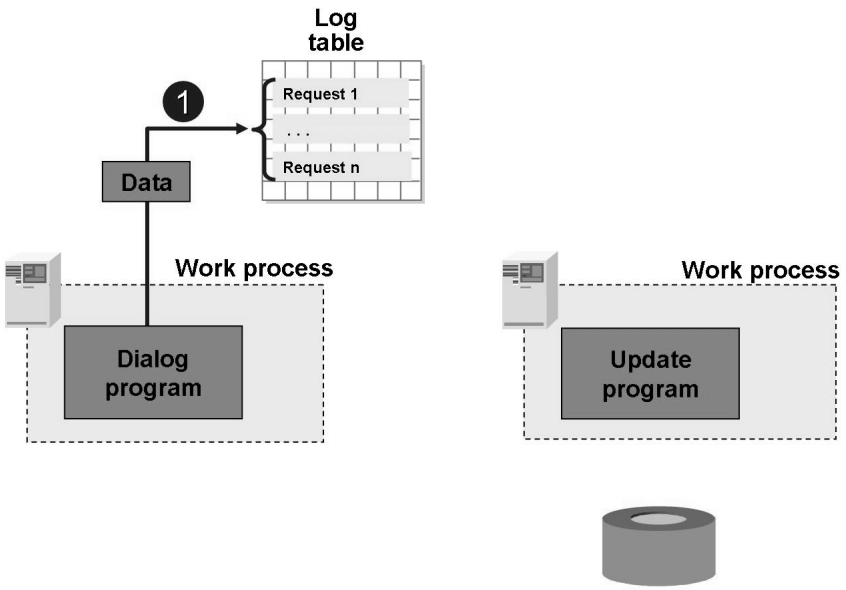


Figure 44: Process: Writing Requests

Step 1: The dialog program receives the data updated by the user and writes it to a special log table. The entries in this table are of a request type. The data contained in them will be written to the database later by program.

A dialog program can write several entries to the log table.

The entries in the log table represent an LUW, in other words, they will either be executed in the database together or not at all ("all or nothing" principle).

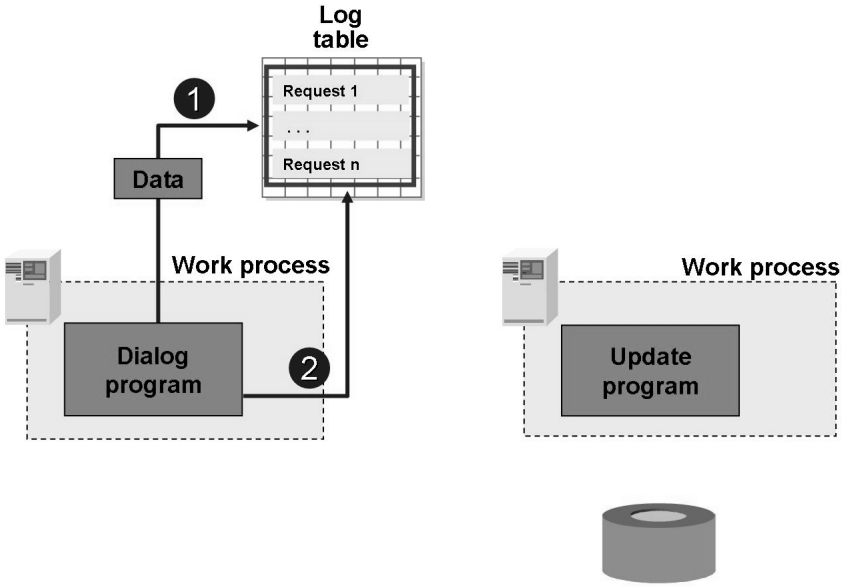


Figure 45: Process: Completing Requests

Step 2: The dialog program closes the logical data packet (LUW) that was written to the log table, and it also informs the basis system that a packet to be updated exists.

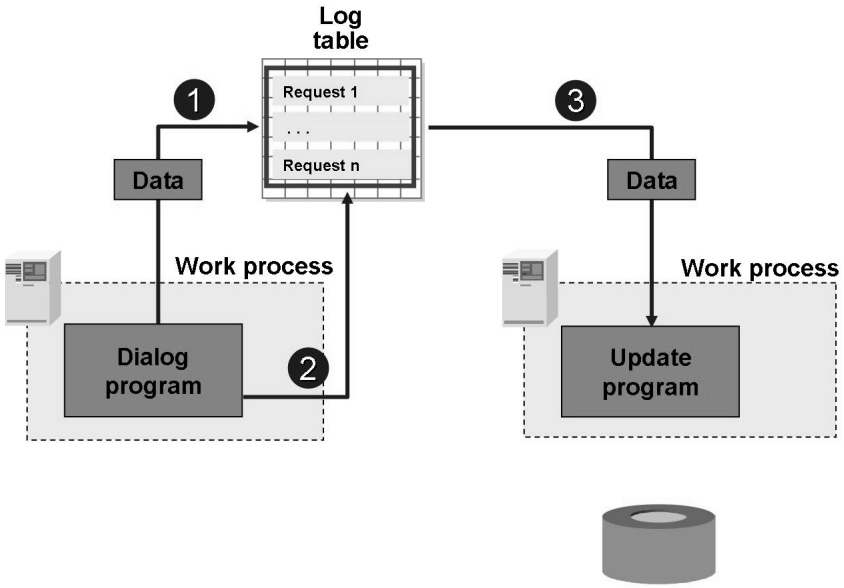


Figure 46: Process: Having Requests Read

Step 3: A basis program reads the data associated with the LUW from the log table and supplies it to the update program.

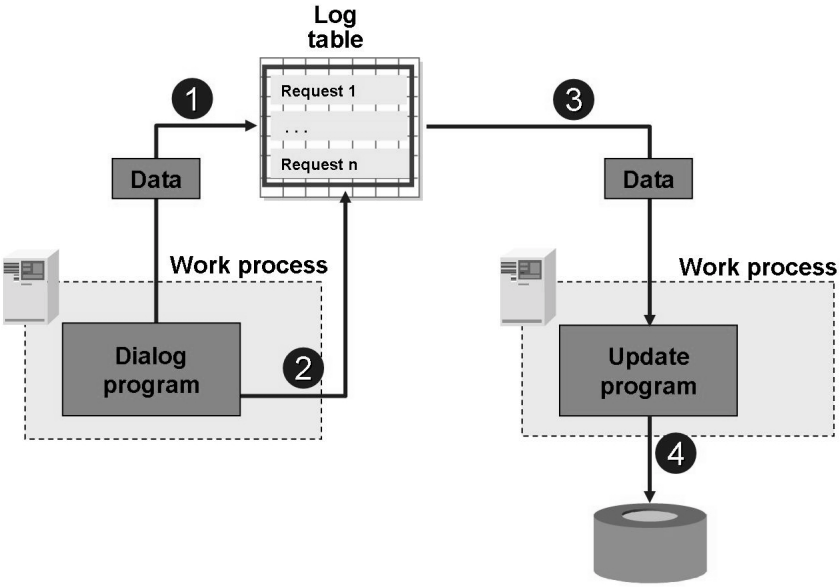


Figure 47: Process: Performing Database Updates

Step 4: The update program accepts the data transferred to it and updates the database entries.

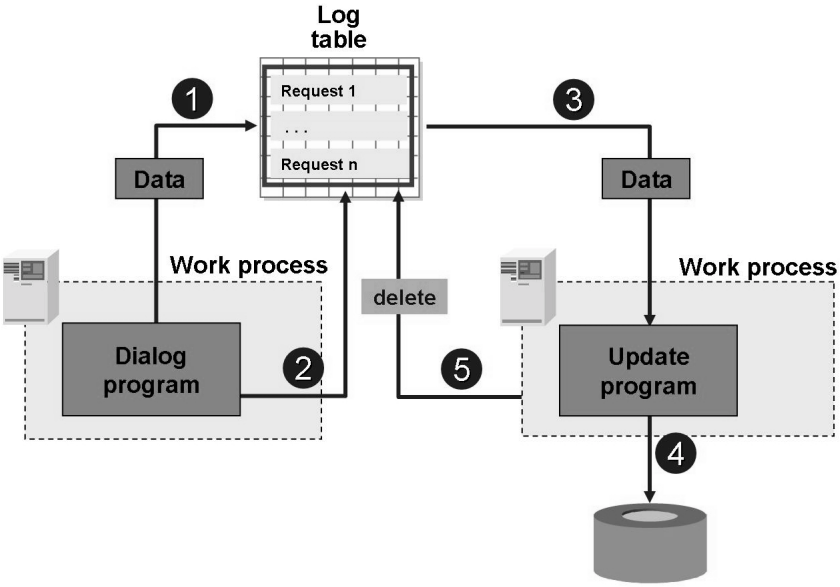


Figure 48: Process: Deleting Requests

Step 5: If the update program runs successfully, a Basis program deletes all entries for the LUW from the log table.

In the event of an error, the entries remain in the log table. They are flagged as incorrect. The user who has triggerred the update is generally informed by express mail about the error.

The option of informing users by mail that an update action has failed can be set using the profile parameters rdisp/vbmail and rdisp/vb_mail_user_list.

- The parameter rdisp/vbmail can be given the value 1 or 0 (mail dispatch if error or not).
- The value given to rdisp/vb_mail_user_list defines who is to be contacted if there is an error. The value "$ACTUSER" means that the user who has created the data record to be updated will be informed.

The monitor transaction for update requests is SM13.

Technical Implementation of the Update

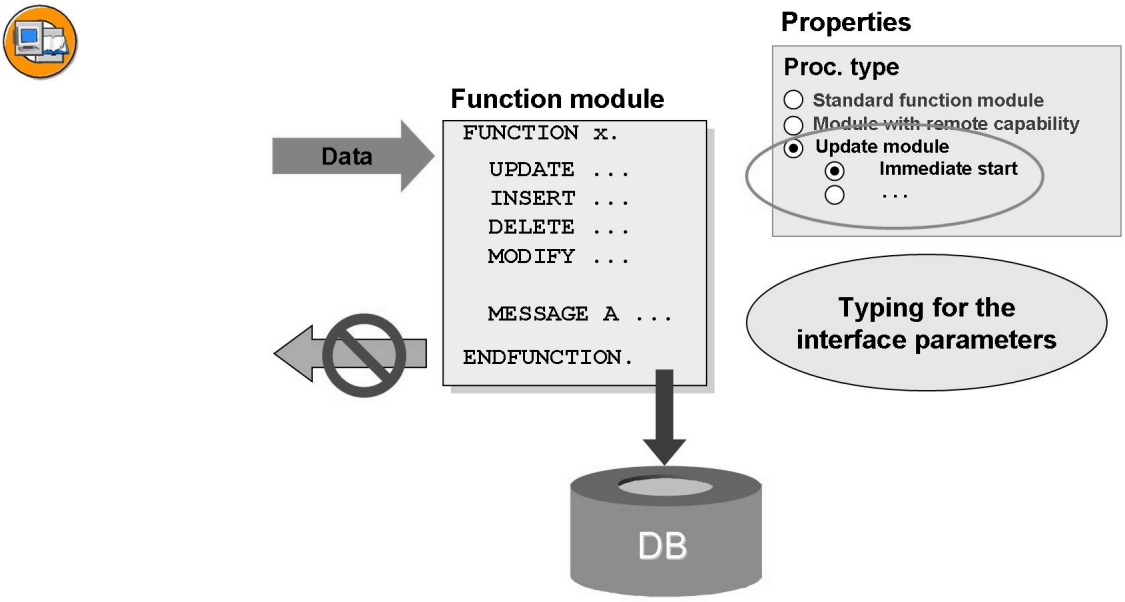


Figure 49: Update Modules

Technical implementation of the update concept requires, in addition to the program that monitors the user dialog, a so-called update program that must be implemented as a special function module **(update module)**.

Update modules, like other function modules, have an interface for transferring data. The interface for update function modules only includes IMPORTING and TABLES parameters. These must be typed using reference fields or structures.

Export parameters and exceptions are ignored in update modules.

The function module contains the actual database update statements.

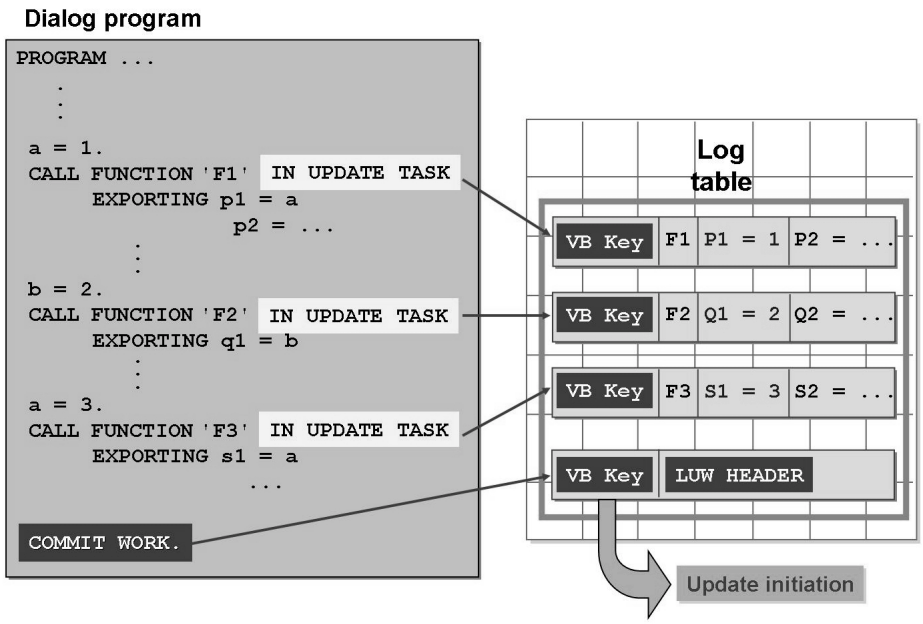


Figure 50: Writing/Closing Requests

You create update requests from the dialog program by calling the respective update function module. Use the IN UPDATE TASK addition. This means that the function module is not executed immediately, but is written to the log table, together with the input data, as an execution request.

All of the update flags in an SAP LUW are stored under the same update key (“VB key”). The update key is a unique worldwide identification code for an SAP LUW.

Only when the system finds a `COMMIT WORK` statement will it create a header entry for the requests that belong together (log header), and then the unit is closed. The log header contains information on the dialog program that wrote the log entries, as well as information on the update modules to be executed.

After the log header has been created, the system informs the dispatcher process that there is an update package for processing.

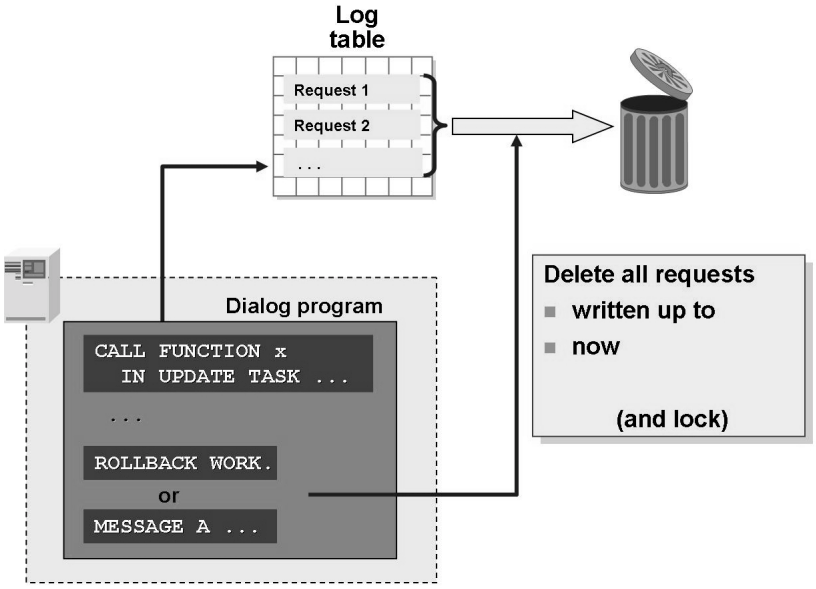


Figure 51: Discarding Requests (Generation Phase)

Sometimes you may need to discard all change requests that were written up to now for the current SAP LUW. This is the case, for example, when you terminate the transaction.

To discard the current SAP LUW during the generation phase, use the ABAP statement `ROLLBACK WORK` or send a type "A" dialog message.

Both procedures

- delete all previous update flags,
- delete all previously set locks,
- discard all of the updates executed in the current DB LUW,
- and discard all of the form routines registered using "PERFORM ON COMMIT".

Caution: The `ROLLBACK WORK` statement does not affect the program context; in other words, all data objects (program-specific objects and objects from function groups that may be used) remain unchanged, and they are NOT reset.

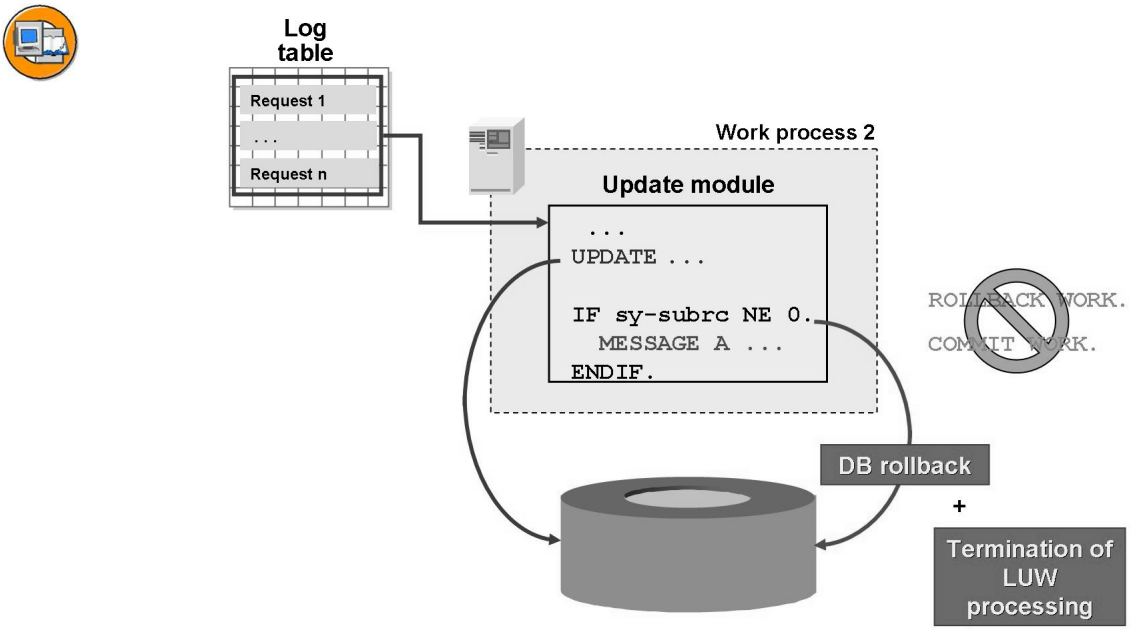


Figure 52: Discarding Requests (Processing Phase)

The task of an update module is to pass the requests for database updates to the database and to evaluate their return codes.

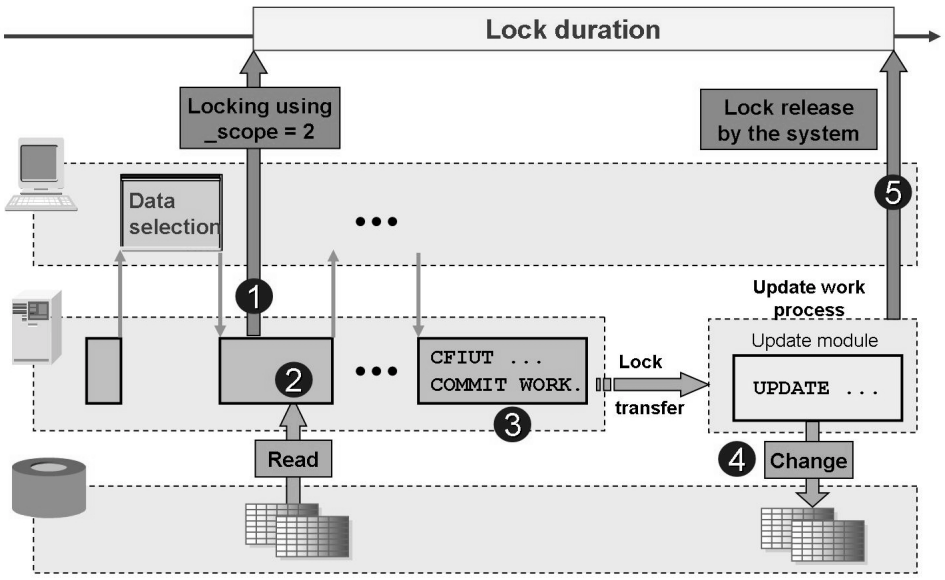
If you want to trigger a database rollback in the update module, issue a type “A” dialog message. In this way, the processing of the current SAP LUW will also be terminated. The log entry belonging to the SAP LUW is flagged as containing an error. The termination message is also entered in the log.

You can examine the log entry using transaction SM13.

The system sends an express mail to the relevant user, telling him or her that the LUW update was terminated. For this purpose, you must set the profile parameters rdisp/vbmail and rdisp/vb_mail_user_list accordingly:

- The parameter rdisp/vbmail can be given the value 1 or 0 (mail dispatch if error or not).
- The value given to rdisp/vb_mail_user_list defines who is to be contacted if there is an error. The value “$ACTUSER” means that the user who has created the data record to be updated will be informed.

You must not use the explicit ABAP statements **`COMMIT WORK`** and **`ROLLBACK WORK`** **in the update module**.




```
CFIUT = CALL FUNCTION IN UPDATE TASK
```

Figure 53: Setting Locks in the Update

If you have locks set in a dialog program that works with the update technique and these locks have been set using **_scope = 2** , you can **pass these on to the update task** at `COMMIT WORK`. After this, they are no longer accessible by the dialog program.

You do not need to release the locks explicitly in the update modules since they are automatically released by a basis program at the end of the update process.

This automatic release of locks by the update task will also take place if there is an error, that is, if one of the update modules terminates and rolls back LUW processing through the output of a termination dialog message.

If the update modules allow failed update requests to be reprocessed (see V1 update), you should note that the data basis in the database tables at the point of reprocessing may be different from that at the point of the failed update attempt. Also, reprocessing generally takes place without locks since these are automatically deleted by the update task if there is an error. (For example, reprocessing is appropriate if a document was not written successfully to the database because of an overflow of table spaces.)

Use and Effect of the Update Modes

This section explains the various update modes and which mode to use for a particular situation.

Asynchronous Update

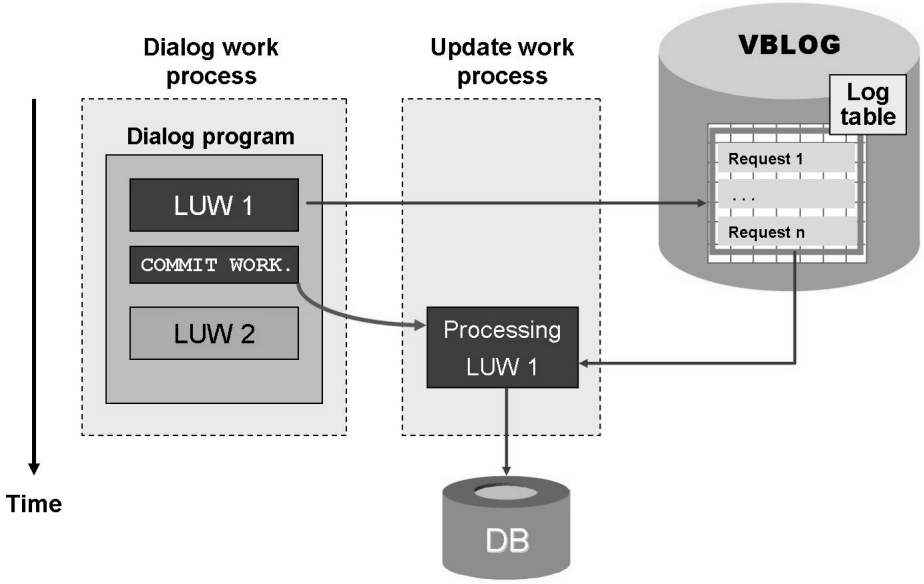


Figure 54: Asynchronous Update

In **asynchronous updates**, the dialog program and update program run separately:

- The dialog program writes the change requests to the log table and closes the LUW with a `COMMIT WORK`.
- The update initiated by the `COMMIT WORK` now processes the change requests. The dialog program is continued; the system does not wait for the update to end.
- The update program runs in a special update work process. This can be on an application server other than the one used for the R/3 System.

Asynchronous updates are useful in transactions where the database updates take a long time and the "perceived performance" by the shorter user dialog response time is important.

Asynchronous updating is the standard technique used in dialog processing.

You can implement the log table VBLOG as a cluster file in your system, or replace it with the transparent tables VBHDR, VBMOD, VBDATA, and VBERROR.

Synchronous Update

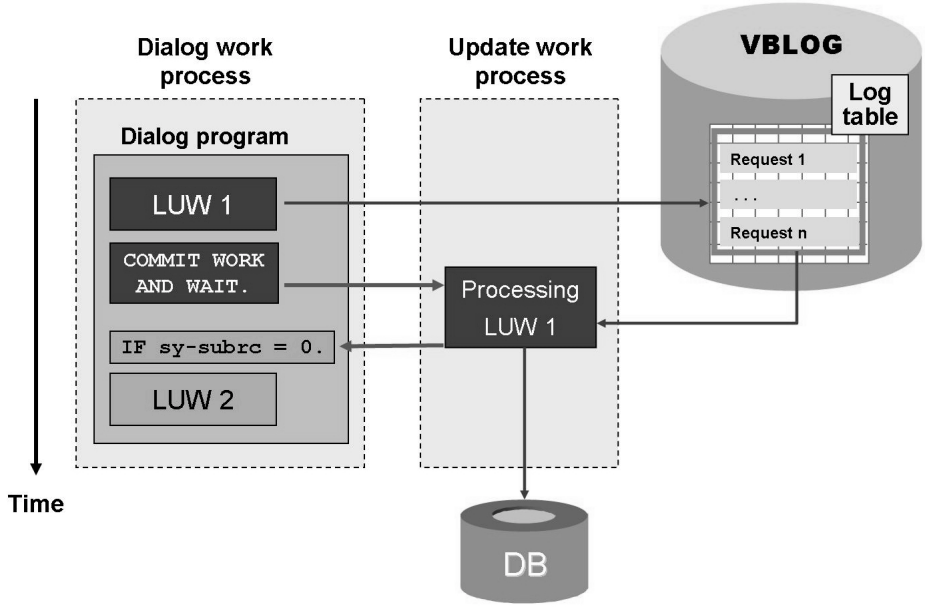


Figure 55: Synchronous Update

If you have a **synchronous update** that is triggered by `COMMIT WORK AND WAIT`, the dialog program waits for the udpate to end before program processing is continued.

You use this update mode if further processing or dialog program terminationdepends on the update result.

You can query the processing success of the synchronous update using the system field sy-subrc.

During the entire waiting phase, the dialog program is in a “rolled out” state. This means that the respective dialog work process is released for further use. When the update is complete, the system again assigns a free dialog work process for further processing.

Local Update

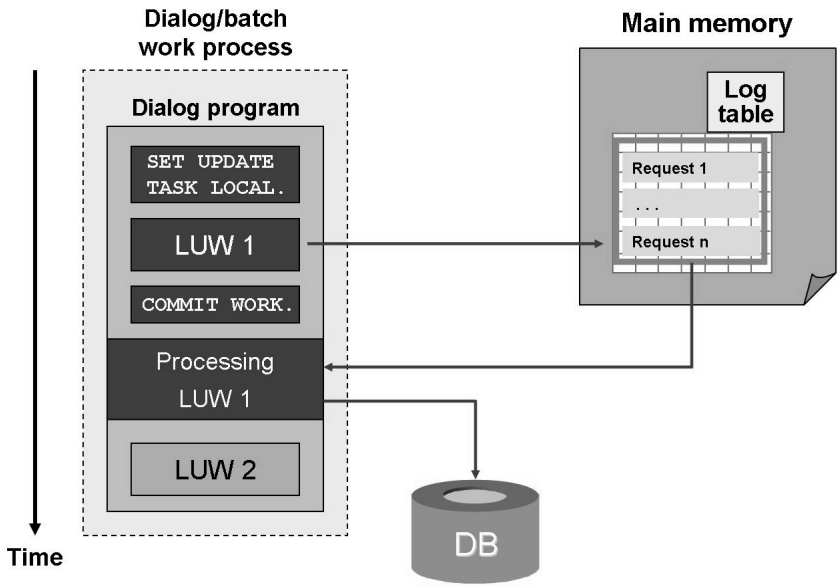


Figure 56: Local Update

In **local updates**, the update functions are run in the same dialog process used by the dialog program containing the dialog program (→ locally). Processing of the dialog program is continued when the update has been complete (synchronous).

To have update modules executed locally, you must use the statement `SET UPDATE TASK LOCAL` before you write the respective requests. Close the written requests with the statement `COMMIT WORK`, and these will be processed in the same dialog work process.

After the local update has been successfully processed, a DB commit is initiated explicitly and the dialog program is continued.

If there is an error and a termination message is dispatched by one of the update modules, the system executes an automatic DB rollback to discard the changes in the current LUW and the dialog program is terminated by the display of a termination message.

In the local update mode, change requests are not written to the database table VBLOG, but kept in main memory. Due to the missing IO accesses, this is quicker than for synchronous or asynchronous updates. The disadvantage, however, lies in the exclusive use of a work process. Therefore, local updates are only apppropriate in batch mode.

`SET UPDATE TASK LOCAL` is only possible iif no requests have been created for the current LUW. This statement only works until the next `COMMIT WORK`.

V1 and V2 Updates

This section explains update types V1 and V2 and which of these to use for a particular database update.

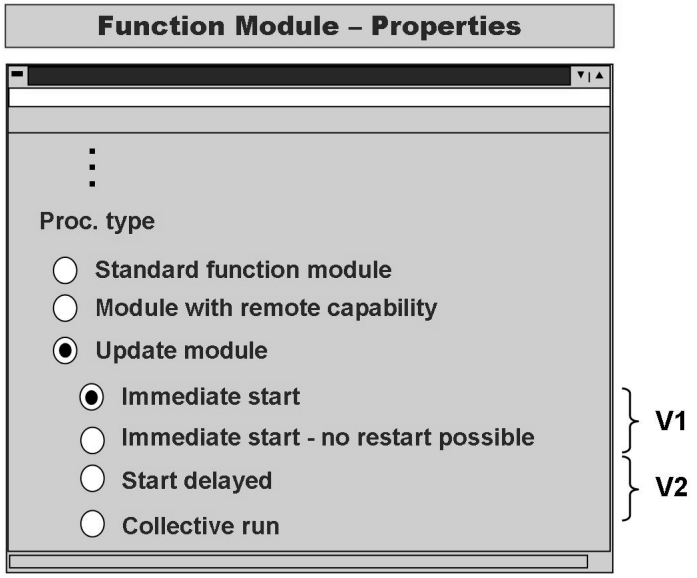


Figure 57: Types of Update Modules (V1/V2)

There are two types of update modules: **V1** and **V2**. The type of update module determines its processing mode. To begin with, all V1 requests in the dialog program are executed as independent DB LUW. Only if they are executed successfully are V2 requests processed, also as independent LUWs. (→ V1- / V2 update phase)

V2 update modules are used for database changes that are linked to the V1 changes (main changes) but do not necessarily have to be executed in the same DB LUW (for example, updating of statistics).

V1 modules can be *restartable* or *non-restartable*. If there has been an update error, you can manually restart requests that were created by restartable update modules using transaction SM13. You do this after you have cleaned up the application error in question. V2 update modules can always be restarted for processing if there has been an error.

The classification “collective run” for V2 modules is a special type of V2 update. Corresponding requests are not executed directly after the V1 update, but only after the collector program RSM13005 (generally planned ahead) has been called.

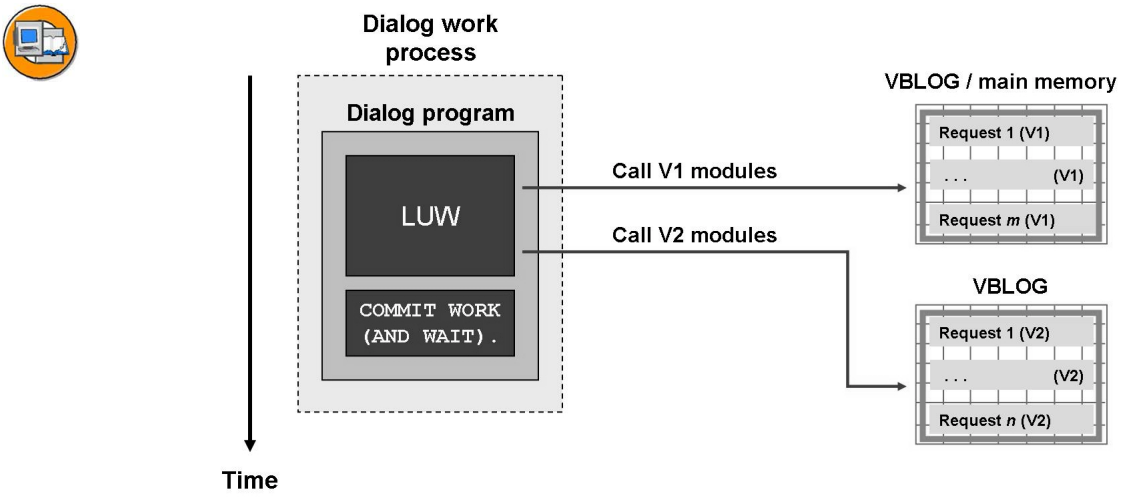


Figure 58: Generating V1 and V2 Requests

The flow diagrams displayed in the previous sections always show V1 requests that were created by V1 update modules.

V1 update modules create requests in the VBLOG table for synchronous and asynchronous updates. For local updates, the V1 requests are retained in the main memory.

V2 requests are always stored in the VBLOG table.

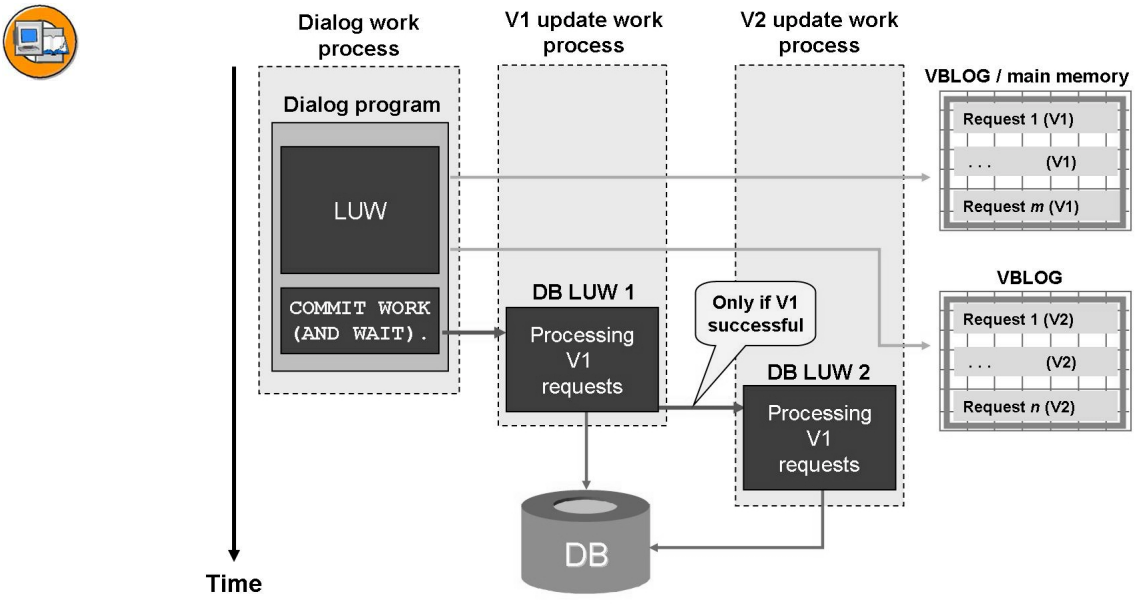


Figure 59: Update Execution

V1 requests are processed in a V1 update work process as an independent DB LUW. If the V1 update has been successful, the system deletes the V1 requests and all the locks passed on to the update task, sets a DB Commit, and triggers the V2 update.

V2 requests are executed in a V2 update work process. These, too, form an independent DB LUW. If no V2 update work process is set up in the R/3 System, V2 requests are executed in a V1 update work process. If all the V2 requests have been processed successfully, they are removed from the VBLOG table and the system also sets a DB Commit. V2 requests generally run without locks since these were already deleted upon completion of the V1 update.

If the V1 update is terminated because a termination message was issued by a V1 update module, all the locks passed on to the update task are deleted, a DB rollback takes place, a mail is sent to the user who created the LUW, and the V1 requests are flagged as incorrect in the VBLOG with the corresponding termination message. The V2 update is not triggered.

However, if the V2 update is terminated by a termination message of a V2 update module, the system triggers a database rollback. All of the V2 changes in the SAP LUW are undone and the V2 requests in VBLOG are flagged as incorrect with the corresponding termination message.

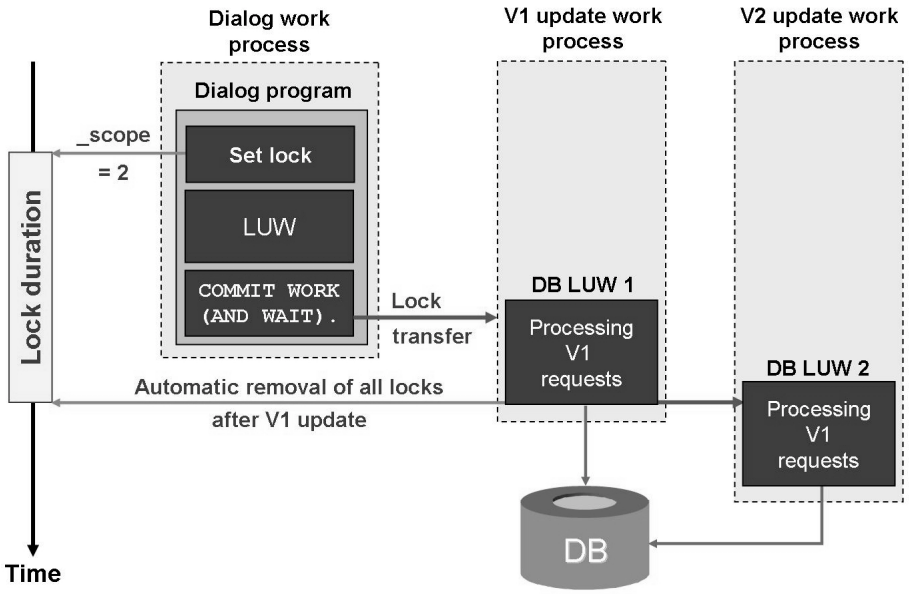


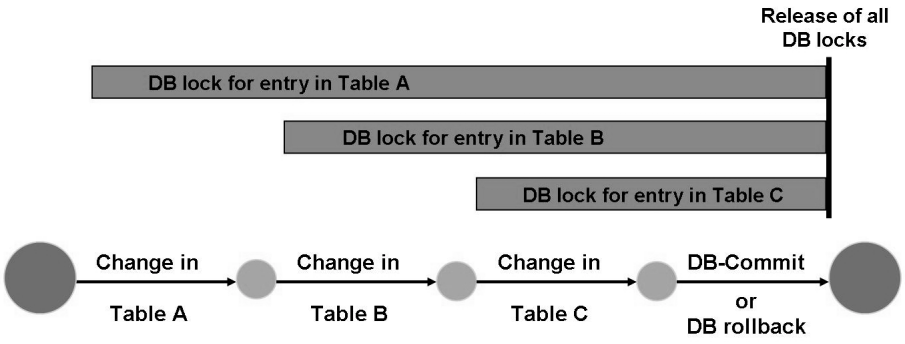
Figure 60: Setting Locks in the Update

The locks created from within the dialog program using **_scope = 2** (default) are transferred to the V1 update task at `COMMIT WORK (AND WAIT)`. At the end of the V1 update, they are automatically deleted, irrespective of whether the V1 update was successful or whether the update was terminated by an error with issue of a termination message.

Therefore, lock entries must not be explicitly removed either in the dialog program (too early) or in the update module (unnecessary).

The V2 update always runs without SAP locks.

Optimization Notes for Database Changes



Performance rules:

- Create new entries first (INSERT)
- Then perform changes to non-performance-critical tables (UPDATE)
- Then perform changes to performance-critical tables (UPDATE)

Figure 61: Shortest Possible Database Locks

Each time a change is updated to the database, the record to be changed is physically locked by the database up to the end of the current database LUW (DB commit or DB rollback). The same applies if you are reading with `SELECT ... FOR UPDATE`

However, since read accesses to the respective record are not allowed for the duration of the physical lock (these must wait until the record is released by the database) and there are many programs that execute read access without locks, it is a good idea - for performance reasons - to keep such automatically-set database locks as short as possible.

For this you should adhere to the following rules when programming inline changes and update modules:

- Create new table entries first. Their database locks pose the least “interference” for other users.
- You should then perform table updates that are not critical to performance. As a rule, these are the tables that are accessed simultaneously by relatively few users.
- Tables that represent central resources in the system (tables accessed by several users simultaneously) should always be changed late, if possible, within an LUW so that the respective database locks only hinder others for a short time.

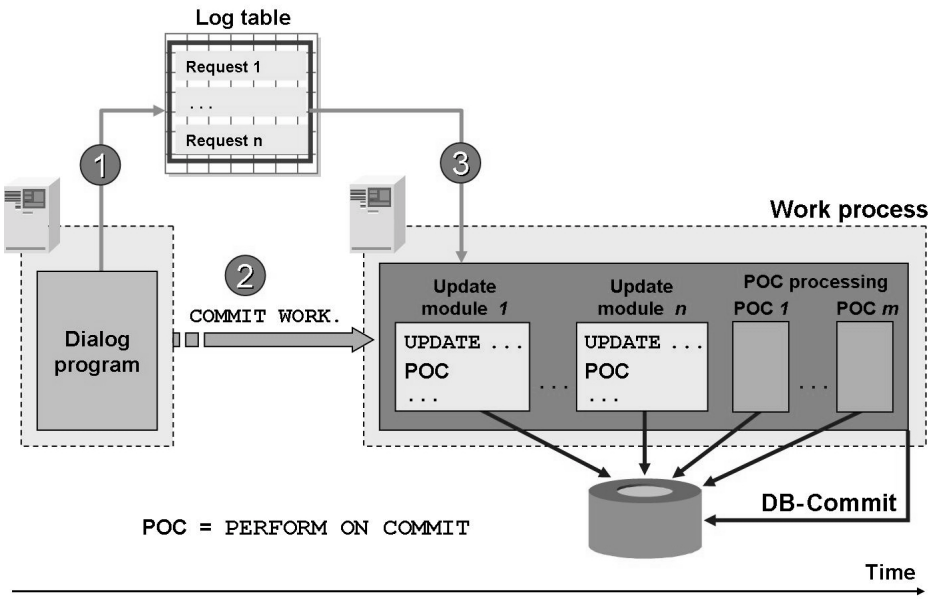


Figure 62: PERFORM ON COMMIT in the Update

During the update, the goal is to execute changes to central tables (performance-critical, often accessed simultaneously by several users) as late as possible in the LUW. To achieve this, you can use the PERFORM ON COMMIT technique in the update.

Encapsulate the changes to central tables as form routines within the appropriate function group that belongs to the corresponding update module. Then call the routines from within the update module using PERFORM ON COMMIT. The form routines are then not executed until the last update module has been processed.

Remember that these form routines must work with the global data of the respective function group.

Exercise 5: Organizing Database Updates

Exercise Objectives

After completing this exercise, you will be able to:

- Perform database updates using the asynchronous update technique

Business Example

You want to execute data base updates (with respect to performance) using asynchronous updates.

Task: Database Updates Using Asynchronous Updates

The program **SAPMZ##_BOOKINGS2** from the previous unit is to be changed or enhanced so that database updates can be performed using the asynchronous update technique.

Program: SAPMZ##_BOOKINGS3

Transaction Code: Z##_BOOKINGS3

Copy from: SAPBC414**T**_BOOKINGS_03

Model Solution: SAPBC414**S**_BOOKINGS_03

1. Copy your solution **SAPMZ##_BOOKINGS2** or the program template **SAPBC414T_BOOKINGS_03** with **all** subobjects to **SAPMZ##_BOOKINGS3** (**##** is your group number). Assign transaction code **Z##_BOOKINGS3** to the program.

2. **Canceling existing bookings:**

   Function modules **UPDATE_SFLIGHT** and **UPDATE_SBOOK** are used to update the table entries in the DB tables SLFIGHT and SBOOK. Can these function modules also be used to perform the updates using the update technique?

   Modify your program so that the updates to the DB tables **SFLIGHT** and **SBOOK** are performed using the update technique:

   Call the corresponding function modules capable of performing updates in the **SAVE_MODIFIED_BOOKING** subroutine.

Continued on next page

Insert the statement **COMMIT WORK** in the PAI module **USER_COMMAND_0200**.

Also, be aware of the fact that the locks set using "_SCOPE = 2" are passed to the update program and therefore must not be explicitly released in the dialog program.

3. **Generating a new booking:**

The data for a new booking is entered on screen 300. Clicking the *Save* icon (function code SAVE) inserts the new booking in the **SBOOK** table and modify the flight in question in the **SFLIGHT** table. The updates are to be performed within a DB LUW using the asynchronous update technique.

To generate a new entry in the DB table SBOOK, use the function module **INSERT_SBOOK**, which is capable of performing updates. This function module is to be called up in the subroutine **SAVE_NEW_BOOKING**. The subroutine is called up from the PAI module **USER_COMMAND_0300** (screen 300) and is already created (blank).

If you have copied your previous solution program SAPMZ##_BOOKINGS2 as a template for this exercise, you should – when the subroutine **SAVE_NEW_BOOKING** starts – copy the subroutine **CONVERT_TO_LOC_CURRENCY**, and pass on **WA_SBOOK**. This is necessary so that you can fill the corresponding currency fields in **WA_SBOOK**:

```
PERFORM convert_to_loc_currency USING wa_sbook.
```

Call up the function modules **INSERT_SBOOK** and **UPDATE_SFLIGHT**, which are capable of performing updates, to update the DB tables SBOOK and SFLIGHT using the update technique.

Insert the statement **COMMIT WORK** in the PAI module **USER_COMMAND_0300**.

Lock the flight and the booking by calling up the corresponding lock modules. Call up subroutine **ENQ_SFLIGHT** and **ENQ_SBOOK** in the appropriate places.

If a user action calls up screen 100, release the locks.

Hint: The booking data is held in structure **WA_SBOOK**.

Solution 5: Organizing Database Updates

Task: Database Updates Using Asynchronous Updates

The program **SAPMZ##_BOOKINGS2** from the previous unit is to be changed or enhanced so that database updates can be performed using the asynchronous update technique.

Program: SAPMZ##_BOOKINGS3

Transaction Code: Z##_BOOKINGS3

Copy from: SAPBC414**T**_BOOKINGS_03

Model Solution: SAPBC414**S**_BOOKINGS_03

1. Copy your solution **SAPMZ##_BOOKINGS2** or the program template **SAPBC414T_BOOKINGS_03** with **all** subobjects to **SAPMZ##_BOOKINGS3** (**##** is your group number). Assign transaction code **Z##_BOOKINGS3** to the program.

   a) -

2. **Canceling existing bookings:**

   Function modules **UPDATE_SFLIGHT** and **UPDATE_SBOOK** are used to update the table entries in the DB tables SLFIGHT and SBOOK. Can these function modules also be used to perform the updates using the update technique?

   Modify your program so that the updates to the DB tables **SFLIGHT** and **SBOOK** are performed using the update technique:

   Call the corresponding function modules capable of performing updates in the **SAVE_MODIFIED_BOOKING** subroutine.

   Insert the statement **COMMIT WORK** in the PAI module **USER_COMMAND_0200**.

   Also, be aware of the fact that the locks set using "_SCOPE = 2" are passed to the update program and therefore must not be explicitly released in the dialog program.

   a) Yes, the attributes ("update function module") of function modules UPDATE_SFLIGHT and UPDATE_SBOOK mean that they can also be used in the update.

      See model solution for source code implementation

Continued on next page

3. **Generating a new booking:**

The data for a new booking is entered on screen 300. Clicking the *Save* icon (function code SAVE) inserts the new booking in the **SBOOK** table and modify the flight in question in the **SFLIGHT** table. The updates are to be performed within a DB LUW using the asynchronous update technique.

To generate a new entry in the DB table SBOOK, use the function module **INSERT_SBOOK**, which is capable of performing updates. This function module is to be called up in the subroutine **SAVE_NEW_BOOKING**. The subroutine is called up from the PAI module **USER_COMMAND_0300** (screen 300) and is already created (blank).

If you have copied your previous solution program SAPMZ##_BOOKINGS2 as a template for this exercise, you should – when the subroutine **SAVE_NEW_BOOKING** starts – copy the subroutine **CONVERT_TO_LOC_CURRENCY**, and pass on **WA_SBOOK**. This is necessary so that you can fill the corresponding currency fields in **WA_SBOOK**:

```
PERFORM convert_to_loc_currency USING wa_sbook.
```

Call up the function modules **INSERT_SBOOK** and **UPDATE_SFLIGHT**, which are capable of performing updates, to update the DB tables SBOOK and SFLIGHT using the update technique.

Insert the statement **COMMIT WORK** in the PAI module **USER_COMMAND_0300**.

Lock the flight and the booking by calling up the corresponding lock modules. Call up subroutine **ENQ_SFLIGHT** and **ENQ_SBOOK** in the appropriate places.

If a user action calls up screen 100, release the locks.

Hint: The booking data is held in structure **WA_SBOOK**.

a) See model solution

Result

Model Solution SAPBC414S_BOOKINGS_03

PAI Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_03I01 .
```

Continued on next page

Internal Use SAP Partner Only

```
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&---------------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      CASE sy-dynnr.
        WHEN '0100'.
          LEAVE PROGRAM.
        WHEN '0200'.
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN '0300'.
* remove all database locks
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN OTHERS.
      ENDCASE.
    WHEN 'EXIT'.
      LEAVE PROGRAM.
    WHEN OTHERS.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT

*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
*************************CANCEL BOOKING************************
    WHEN 'BOOKC'.
      PERFORM enq_sflight_sbook.
      PERFORM read_sflight USING wa_sflight sysubrc.
      PERFORM process_sysubrc_bookc.
      PERFORM read_spfli USING wa_spfli.
      PERFORM read_sbook USING itab_book itab_cd.
      REFRESH CONTROL 'TC_SBOOK' FROM SCREEN '0200'.
*************************CREATE BOOKING************************
    WHEN 'BOOKN'.
* lock flight in Table SFLIGHT, which will be modified when new
* booking is saved
      PERFORM enq_sflight.
      PERFORM read_sflight USING wa_sflight sysubrc.
```

Continued on next page


```
      PERFORM process_sysubrc_bookn.
      PERFORM read_spfli USING wa_spfli.
      PERFORM initialize_sbook USING wa_sbook.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN '0100'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT

*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0200  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0200 INPUT.
  CASE save_ok.
    WHEN 'SAVE'.
      PERFORM collect_modified_data USING itab_sbook_modify.
      PERFORM save_modified_booking.
* start asynchronous update and new SAP-LUW
      COMMIT WORK.
* database locks should not be removed here since they will be
* removed by the update program later.
      SET SCREEN '0100'.
    WHEN 'BACK'.
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0200'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0200  INPUT


*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0300  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0300 INPUT.
  PERFORM tabstrip_set.
  CASE save_ok.
    WHEN 'NEW_CUSTOM'.
      PERFORM create_new_customer.
      SET SCREEN '0300'.
    WHEN 'SAVE'.
      PERFORM save_new_booking.
* start asynchronous update and new SAP-LUW
      COMMIT WORK.
```

Continued on next page


```
* database locks will be removed by update program
      SET SCREEN '0100'.
    WHEN 'BACK'.
* remove all database locks
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0300'.
  ENDCASE.
ENDMODULE.                              " USER_COMMAND_0300  INPUT
```

FORM Routines

F01

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_03F01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKN
*&---------------------------------------------------------------------*
FORM process_sysubrc_bookn.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0300'.
    WHEN OTHERS.
* remove all database locks
      PERFORM deq_all.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                                " PROCESS_SYSUBRC_BOOKN
```

F04

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_03F04
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
```

Continued on next page


```
*&      Form  SAVE_MODIFIED_BOOKING
*&---------------------------------------------------------------------*
FORM save_modified_booking.
  CALL FUNCTION 'UPDATE_SBOOK' IN UPDATE TASK
       EXPORTING
            itab_sbook = itab_sbook_modify.
* no exception handling when using asynchronous update technique
  PERFORM update_sflight.
ENDFORM.                               " SAVE_MODIFIED_BOOKING

*&---------------------------------------------------------------------*
*&      Form  UPDATE_SFLIGHT
*&---------------------------------------------------------------------*
FORM update_sflight.
  CALL FUNCTION 'UPDATE_SFLIGHT' IN UPDATE TASK
       EXPORTING
            carrier    = wa_sflight-carrid
            connection = wa_sflight-connid
            date       = wa_sflight-fldate.
* no exception handling when using asynchronous update technique
ENDFORM.                               " UPDATE_SFLIGHT

*&---------------------------------------------------------------------*
*&      Form  SAVE_NEW_BOOKING
*&---------------------------------------------------------------------*
FORM save_new_booking.
  PERFORM convert_to_loc_currency USING wa_sbook.
* lock booking on DB table sbook to be created
  PERFORM enq_sbook.
  CALL FUNCTION 'INSERT_SBOOK' IN UPDATE TASK
       EXPORTING
            wa_sbook = wa_sbook.
* no exception handling when using asynchronous update technique
  PERFORM update_sflight.
ENDFORM.                               " SAVE_NEW_BOOKING
```

Lesson Summary

You should now be able to:

- Perform database changes using various update techniques
- Use and create update modules
- Implement update types V1 and V2
- Implement the SAP locking concept in accordance with the selected change type

Unit Summary

You should now be able to:

- Executing database updates directly from application programs or by using delayed subroutines.
- Perform database changes using various update techniques
- Use and create update modules
- Implement update types V1 and V2
- Implement the SAP locking concept in accordance with the selected change type

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 5

Complex LUW Processing

Unit Overview

- Runtime Architecture and Storage Access
- Data Transfer
- LUW Logic

Unit Objectives

After completing this unit, you will be able to:

- Call existing programs from your program using different techniques
- Explain the runtime architecture and the storage access options of these programs (ABAP memory and SAP memory).
- Appropriately implement the different methods for data transfer between your program and the programs called from within your program.
- Explain the LUW logic for program-controlled program calls
- Perform complex LUW processing
- Use the SAP lock mechanism for complex LUW processing.

Unit Contents

Lesson: Runtime Architecture and Storage Access for Programs Called Within Programs

Lesson Overview

This lesson explains runtime architecture and storage access for programs called within programs.

Lesson Objectives

After completing this lesson, you will be able to:

- Call existing programs from your program using different techniques
- Explain the runtime architecture and the storage access options of these programs (ABAP memory and SAP memory).

Business Example

You want to call further programs (function modules, reports, transactions) from your program and use the various storage access options.

Programs Called Within Programs

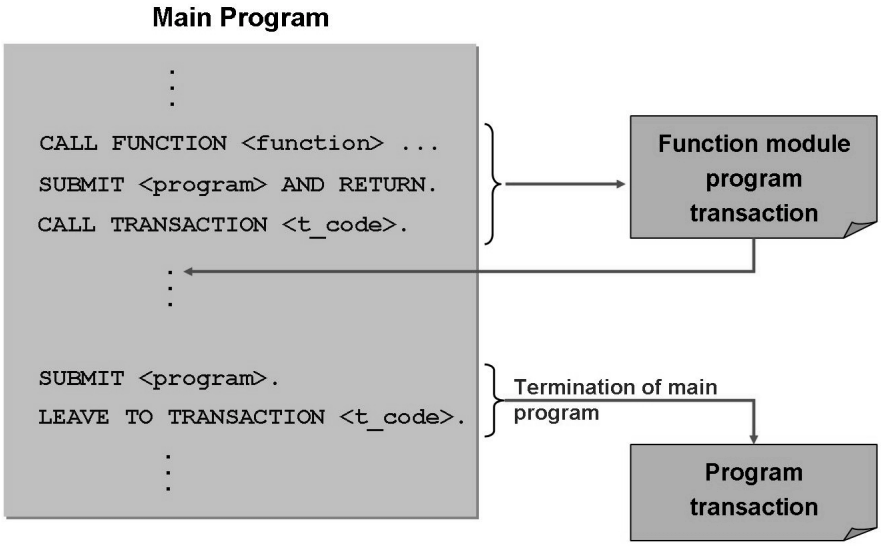


Figure 63: Synchronous Calls

There are two ways of calling other ABAP programs on a synchronous basis from within a program.

- `CALL FUNCTION, CALL TRANSACTION, SUBMIT <program> AND RETURN`:

  The processing of the called program is **inserted**, that is, the processing of the calling program is **interrupted** and then **continued** again after the called program has been completed.

- `SUBMIT <program>, LEAVE TO TRANSACTION <t_code>` :

  The calling program is **terminated** and the called program is started.

`SUBMIT <program>` and `SUBMIT <program> AND RETURN` are used to execute programs that can be executed (program type “1” or “executable program”).

`CALL TRANSACTION` and `LEAVE TO TRANSACTION` call transactions.

For more details on **SUBMIT** , **CALL TRANSACTION**, and **LEAVE TO TRANSACTION**,refer to the keyword documentation integrated in the ABAP Editor.

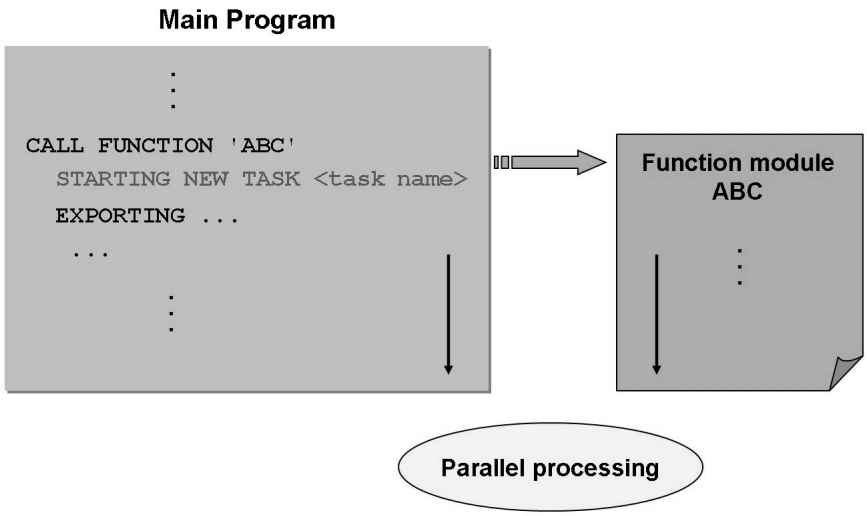


Figure 64: Asynchronous Call of a Function Module

Function modules can also be called asynchronously for executing processes in parallel. For this, the call must be supplied with the addition “STARTING NEW TASK <task name>” stands for a user-individual name for the new independent task in which the function module will be processed.

Asynchronously called function modules are processed **in parallel to and independent of the calling program**.

You can receive the output of the function module (RECEIVE RESULTS FROM FUNCTION) in a later processing phase of the calling program.

Function modules that are to be called using the addition STARTING NEW TASK must be marked as capable of being “called remotely” in their properties (process type: remote-capable module).

For further information, see the keyword documentation in the ABAP Editor for **CALL FUNCTION**.

Accessing ABAP and SAP Memory

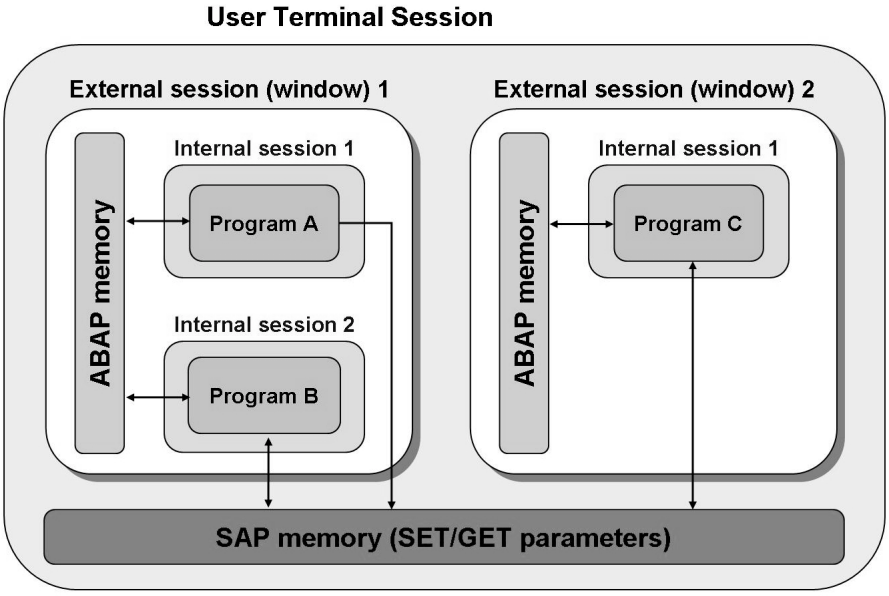


Figure 65: Logical Memory Level Model

Several **external sessions** or modes can be active within a **user session**. An external session is usually linked to an R/3 window.

Several **internal sessions** (up to 20) can run within one external session. One program is always processed within an **internal session**.

Program data is only visible within the program.

You can use SAP memory and ABAP memory to pass data between programs.

Each **SAP session** has one **SAP memory** that can be accessed from all other modes of this session. The SAP memory serves as a storage area for field values and is retained for the duration of the session. You can use the contents of the SAP memory as default values for the respective screen input fields. Since all sessions can access the SAP memory, it is only conditionally suitable for transferring data between internal sessions of an external session. Instead, the ABAP memory should be used.

Each **external session** has its own **ABAP memory**. Access to the ABAP memory is only possible from the respective internal sessions. It is provided as a storage area for internal program variables (fields, structures, internal tables, complex objects) so that they can be passed between internal sessions of an external session. When you end an external session, the corresponding ABAP memory is released automatically.

Accessing the ABAP or SAP memory will be discussed in a later section of this unit.

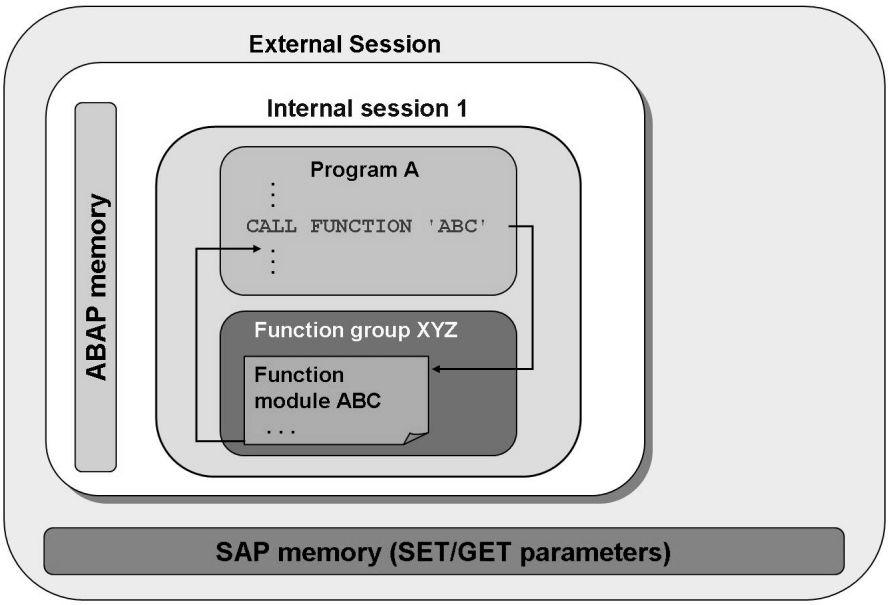


Figure 66: Storage access for a synchronous function module call

When a function module is called, the corresponding function group is loaded into the **current internal session** and the called function module is processed. The procesing of the calling program is interrupted and continued after the function module has been executed.

Note that the loaded function group, together with the global data objects there, are kept in the internal session up to the end of the calling program. This means in particular that when a further function module of this group is called by the current main program, the new function group does not have to be reloaded and the global data objects of the function group have the same contents as they had after the first function module was called.

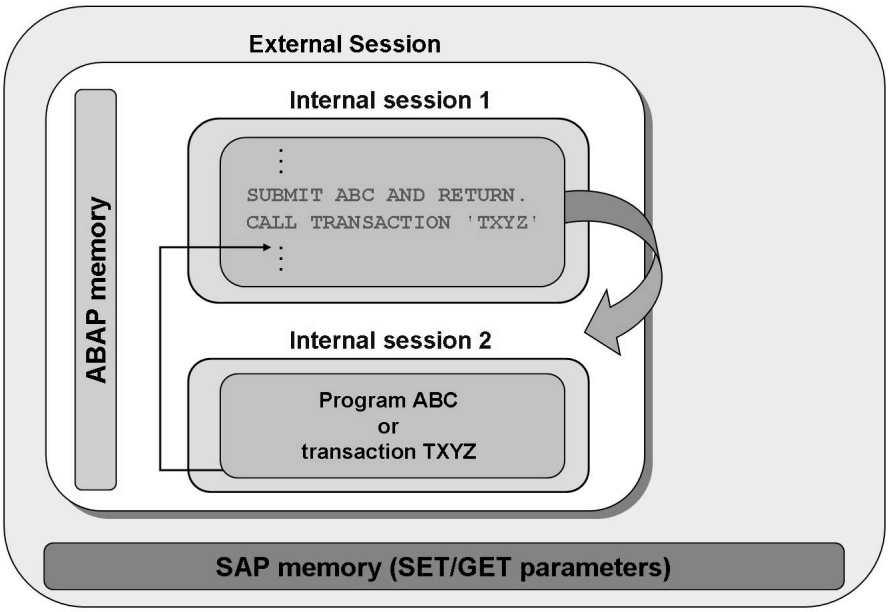


Figure 67: Storage access for "SUBMIT AND RETURN" and "CALL TRANSACTION"

The program called using CALL TRANSACTION or SUBMIT AND RETURN runs in a **separately opened internal session** that contains a new program context.

After the called program has been completed, the new internal session is deleted and processing is continued for the calling program.

The called program can end itself prematurely with the statement LEAVE PROGRAM.

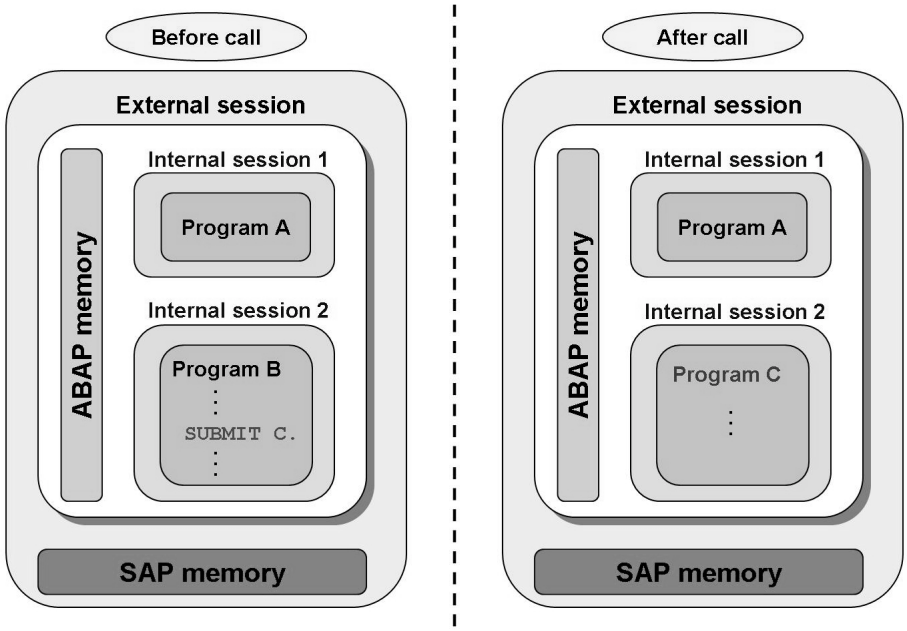


Figure 68: Storage access for "SUBMIT"

If a program is called through the SUBMIT statement, the context of the calling program is removed from the current internal session and the called program is loaded for processing.

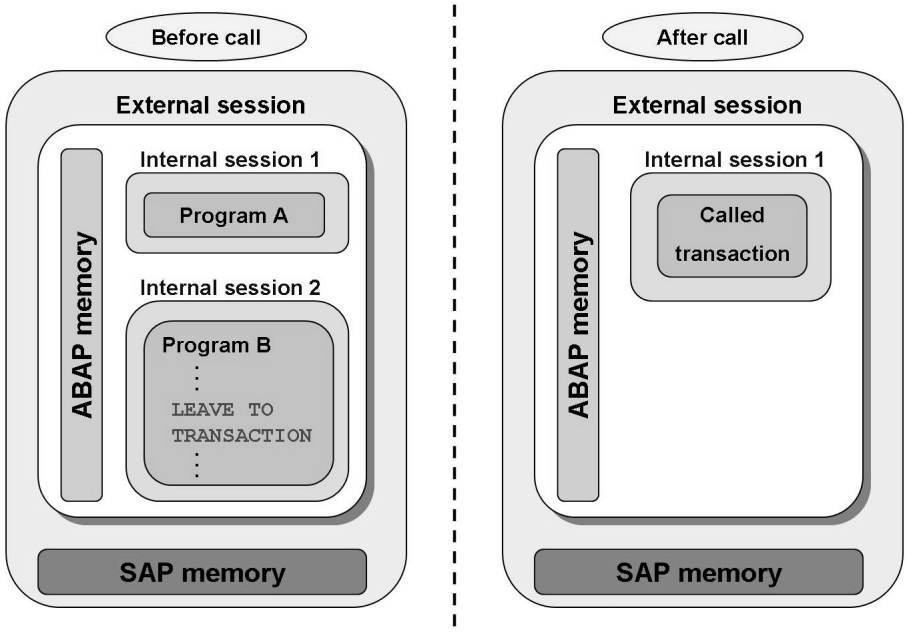


Figure 69: Storage access for "LEAVE TO TRANSACTION"

The statement LEAVE TO TRANSACTION removes all the internal sessions of the current external session and opens a new internal session for processing the called transaction.

When this happens, the ABAP memory is **initialized**! This means, in particular, that you cannot pass any data to the called transaction through the ABAP memory.

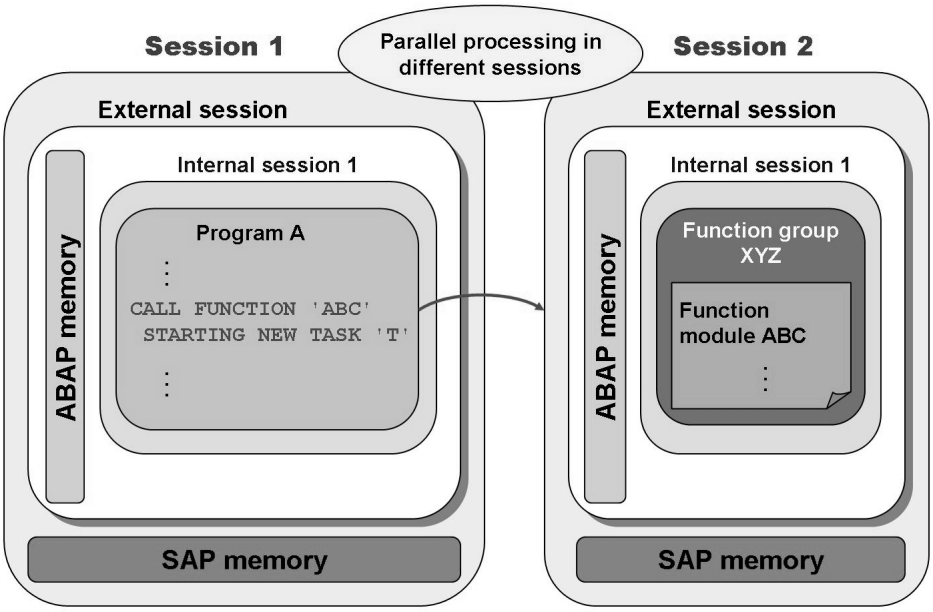


Figure 70: Storage access for an asynchronous function module call

Function modules that have been called asynchronously are processed on the same application server within a **newly opened session**. The processing takes place in parallel and independently of the calling program.

You can receive the output of the function module (RECEIVE RESULTS FROM FUNCTION) in a later processing phase of the calling program.

For further information, see the keyword documentation in the ABAP Editor for CALL FUNCTION and RECEIVE RESULTS FROM FUNCTION.

Lesson Summary

You should now be able to:

- Call existing programs from your program using different techniques
- Explain the runtime architecture and the storage access options of these programs (ABAP memory and SAP memory).

Lesson: Passing Data Between Programs

Lesson Overview

This lesson explains methods for passing data between calling programs and called programs.

Lesson Objectives

After completing this lesson, you will be able to:

- Appropriately implement the different methods for data transfer between your program and the programs called from within your program.

Business Example

You want to appropriately implement the different methods for data transfer between your program and the programs called from within your program.

Passing Data Between Programs

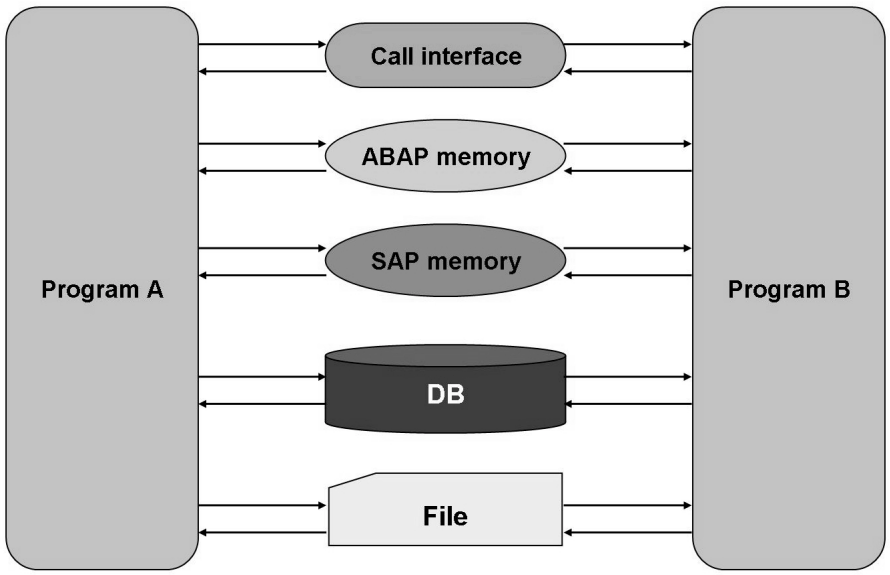


Figure 71: Overview: Passing Data Between Programs

There are different ways of passing on data to a program when it is called:

1. Through the interface of the called program (interface of a subroutine, function module, or dialog modules, standard selection screen of a report)
2. through the ABAP memory
3. through the SAP memory
4. through database tables
5. through files on your presentation server or application server

For further information about transferring data using database tables and the shared buffer, refer to the keyword documentation in the ABAP Editor for the terms **EXPORT** and **IMPORT**.

For further information about transferring data between an ABAP program and your presentation server, refer to the documentation of function modules **GUI_UPLOAD** and **GUI_DOWNLOAD**.

For further information about transferring data between an ABAP program and your application server, refer to the keyword documentation in the ABAP Editor for the terms **TRANSFER** and **READ DATASET**.

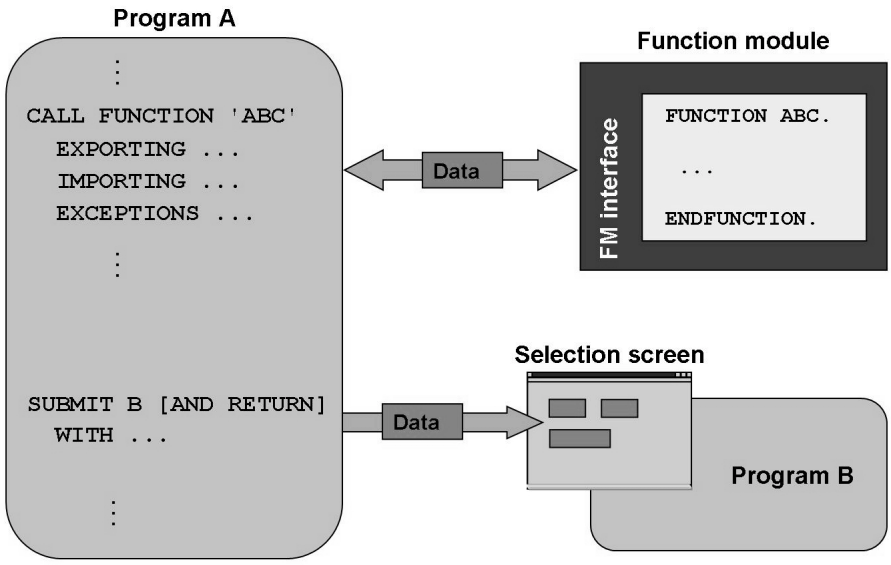


Figure 72: Data Transfer Through the Call Interface

Function modules have an interface, which you can use to pass data between the calling program and the function module itself (there is also a comparable mechanism for ABAP subroutines).

If you are calling an ABAP program that has a standard selection screen, you can pass values to the input fields on the selection screen.. There are two ways to do this:

- You enter a variant for the call (SUBMIT addition `USING SELECTION-SET`)
- By entering actual values for the input fields on the selection screen during the call (see next figure).

```
SUBMIT <program> [AND RETURN] [VIA SELECTION-SCREEN]
 WITH <parameter> EQ <value>.

 WITH <sel_opt> <operator> <value> SIGN <s>.

 WITH <sel_opt> BETWEEN <value1> AND <value2> SIGN <s>.

 WITH <sel_opt> NOT BETWEEN <value1> AND <value2> SIGN <s>.
 WITH <sel_opt> IN <sel_tab>.
```

```
RANGES sel_tab FOR sflight-fldate.

SUBMIT abc AND RETURN
  WITH p_carrid EQ mycarrid
  WITH p_connid EQ myconnid
  WITH s_fldate IN sel_tab.
```

Figure 73: SUBMIT . . . WITH

The SUBMIT addition `VIA SELECTION-SCREEN` is used to start the called program through the display of its selection screen. If this addition is not specified, the system executes the program without processing its selection screen. In this case, you can still pass values for the input fields of its selection screen to the program (WITH addition; for syntax, see above).

Hint: If you use the “pattern” key in the ABAP Editor to insert a program call using SUBMIT, the system will list - in the inserted command - all the selection screen elements of the program to be called by means of the WITH addition.

For further information on the WITH addition, see the ABAP Editor keyword documentation for the term “SUBMIT WITH”.

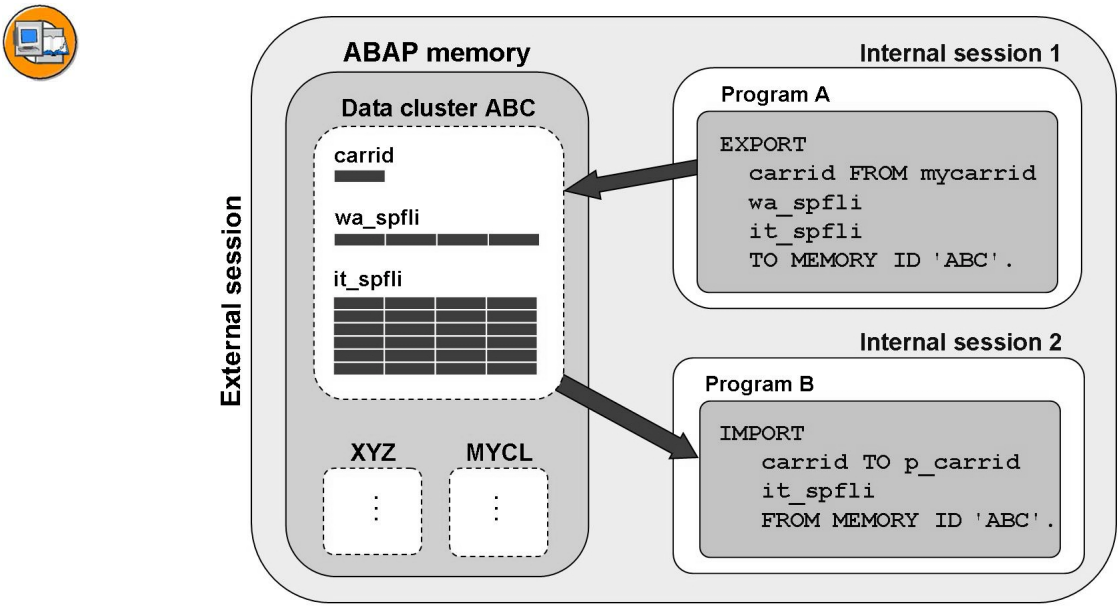


Figure 74: Passing Data Using the ABAP Memory

Using the statement `EXPORT TO MEMORY ID <id>`, you can copy variables of your pgoram with their current values as data clusters into the ABAP memory. The ID you specify here uniquely identifies the created data cluster (maximum 32 characters).

An export to the same memory ID will overwrite the corresponding data cluster.

The `IMPORT FROM MEMORY ID <id>` statement allows you to copy data from the specified data cluster into the fields of your ABAP program.

The source and target variables must have the same format in the write and read programs.

The statement `FREE MEMORY ID <id>` deletes the corresponding data cluster. `FREE MEMORY` **without** the **ID addition** deletes the **entire** ABAP memory of the current external session.

It is possible to read only part of the variables in the data cluster using IMPORT.

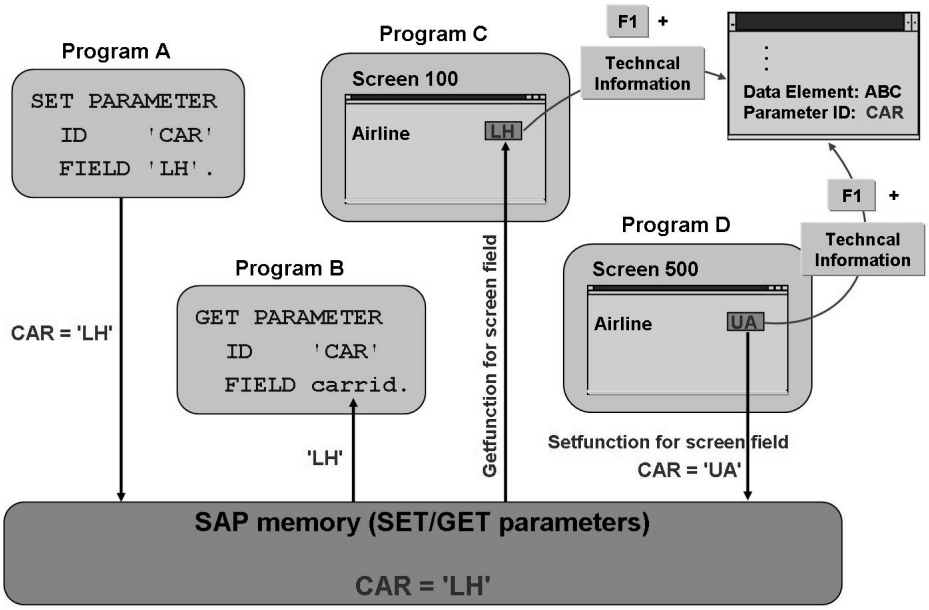


Figure 75: Data Transfer Through the SAP Memory

Using the *Object Navigator*, you can define **parameter IDs** for the R/3 System (→ entries in table TPARA). The ID parameter must not be longer than 20 characters maximum.

Using the `SET PARAMETER ID` statement, ABAP programs can set a value in the SAP memory of the current session for the specified parameter. This value can be read by **all programs of the same session** by means of the `GET PARAMETER ID` statement.

A parameter in the SAP memory can also be set by a user entry in a screen field. For this purpose, the screen field must be defined through a data element that is linked with the respective parameter ID. In addition, the SET functions must be activated in the screen field properties.

Conversely, a screen input field can display the corresponding parameter value in the SAP memory to the user as input proposal. Prerequisites for this are

- the screen field must be defined through a data element that is linked with the respective parameter ID,
- the GET function of the screen filed must be active and
- the screen field must only be occupied by the initial value within the program.

In this way, programs and screens can exchange data through the SAP memory during the same session.

For information on the parameter ID linked to a screen input field, choose the *F1 help* on the input screen →*Technical Info*.

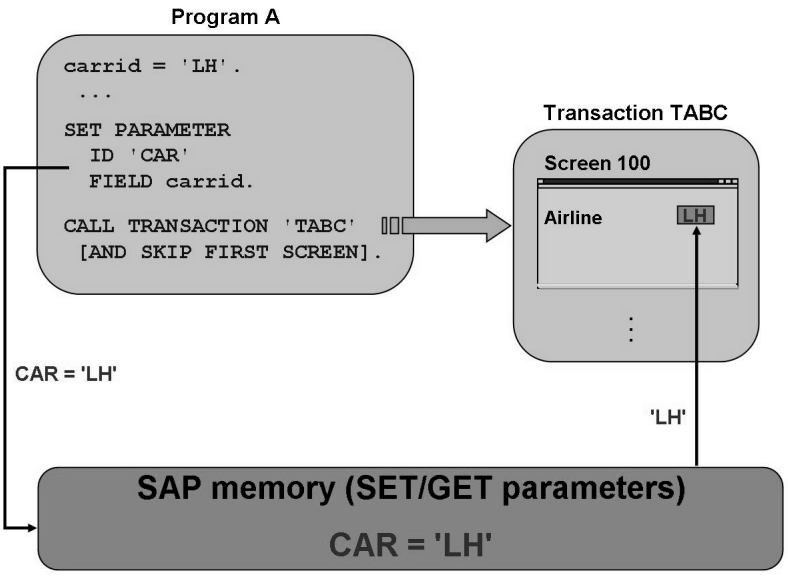


Figure 76: Screen Default Value Through the SAP Memory

The above example shows how you can set default values for input fields of a transaction called by a program. In this way, you can excute the transaction, if required, without displaying the first screen (addition: `AND SKIP FIRST SCREEN`).

Here you should take note of the prerequisites that must be fulfilled *for displaying a default value from the SAP memory by a screen field* (see previous figure).

Lesson Summary

You should now be able to:

- Appropriately implement the different methods for data transfer between your program and the programs called from within your program.

Lesson: LUW Logic in Program-Controlled Calls

Lesson Overview

This lesson explains LUW logic for program-controlled calls.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the LUW logic for program-controlled program calls
- Perform complex LUW processing
- Use the SAP lock mechanism for complex LUW processing.

Business Example

You want to implement complex LUW processing with LUW logic for program-controlled calls.

LUW Logic in Program-Controlled Calls

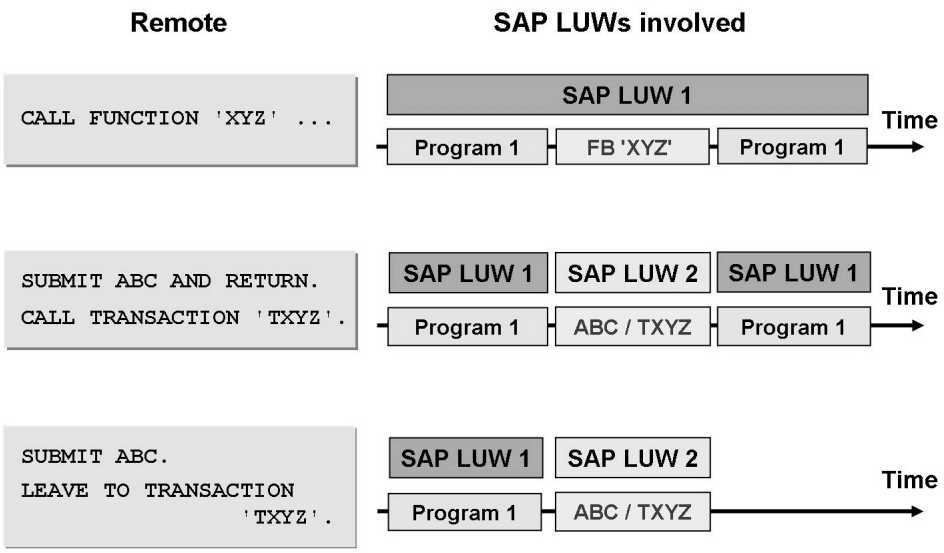


Figure 77: SAP LUWs in Synchronous Program Calls

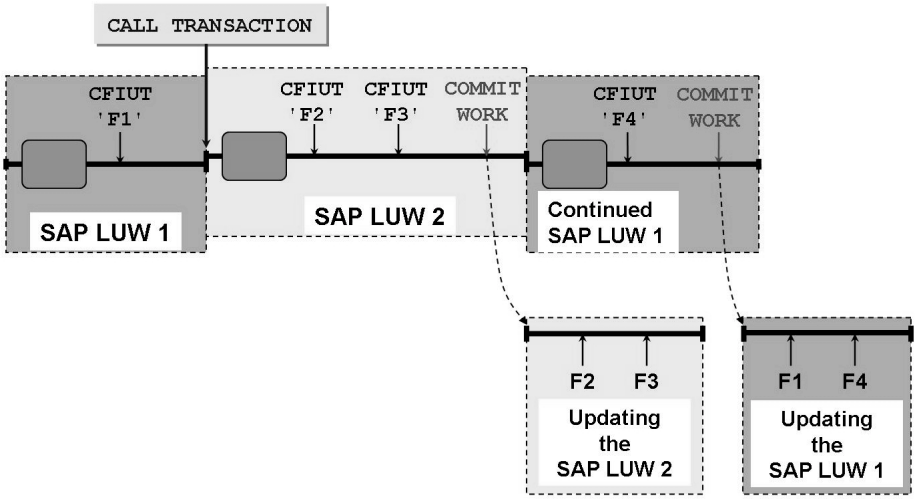
Function modules run in the same SAP LUW as the program that calls them.

Programs called by `SUBMIT AND RETURN`, `CALL TRANSACTION`, `SUBMIT` or `LEAVE TO TRANSACTION` each run in a separate SAP LUW, that is, their update requests have separate update keys.

If you have SUBMIT AND RETURN and CALL TRANSACTION, the SAP-LUW of the calling program is continued as soon as the calling program is completed. The LUWs of calling and called programs are run indepedently of one another. Therefore:

- Direct inline changes are updated at each screen changed to the database.
- Update flags and calls using PERFORM ON COMMIT each require an independent COMMIT WORK in the corresponding SAP LUW (see next figure).

If SUBMIT and LEAVE TO TRANSACTION are used, the SAP LUW of the calling program ends. Direct inline changes were updated at each screen changed to the database. However, if you have not concluded your update flags using COMMIT WORK before the program call, these are not closed in the log table with the corresponding update key, and will therefore not be executed by the update work process. The same applies to change routines flagged with PERFORM ON COMMIT.



CFIUT = CALL FUNCTION IN UPDATE TASK

Figure 78: SAP LUWs for CALL TRANSACTION

If you call up transactions with nested calls, each transaction needs its own COMMIT WORK, since each transaction maps its own SAP LUW.

The same is true for executable programs that are called with SUBMIT <program> AND RETURN.

Hint: Note that an implicit or explicit DB commit in the called transaction also updates all the completed inline changes of the calling program.

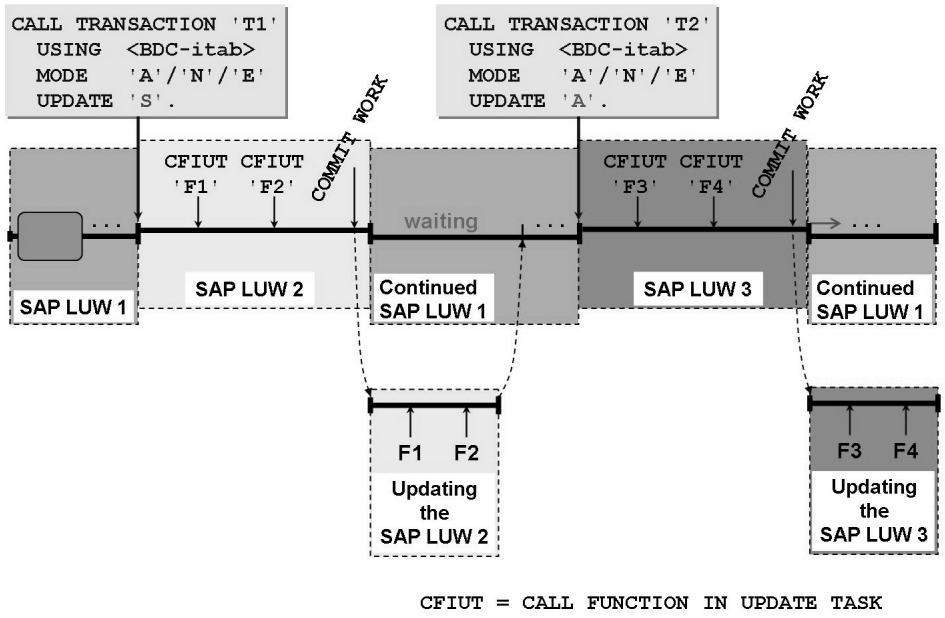


Figure 79: Call Mode in CALL TRANSACTION

If a transaction is called from within a program using `CALL TRANSACTION`, you can also have the transaction executed **without user dialog** (that is, in the background). For this purpose, you need to supply an internal table in batch-input format (see the online documentation for the CALL TRANSACTION) using the USING addition. This table contains the "filled" transaction screens. Also, you should specify the MODE parameter for the call with "N" ("do not display").

Further values for the MODE parameter are "A" ("display" = default) and "E" ("only display if error").

Using the call parameter UPDATE, you can overwrite the default update mode for the transaction to be called, which is usually asynchronous. Possible values for the UPDATE parameter are "A" ("asynchronous", = default), "S" ("synchronous") and "L" ("local").

If you have `UPDATE = 'S'`, the processing of the calling program will only be continued when the updating triggered by the called transaction is completed. The update success is then returned by the system field sy-subrc. You use `UPDATE = 'S'` if further processing after the transaction that is to be called depends on the success of the transaction.

For further information, refer to the keyword documentation in the ABAP Editor for the term CALL TRANSACTION.

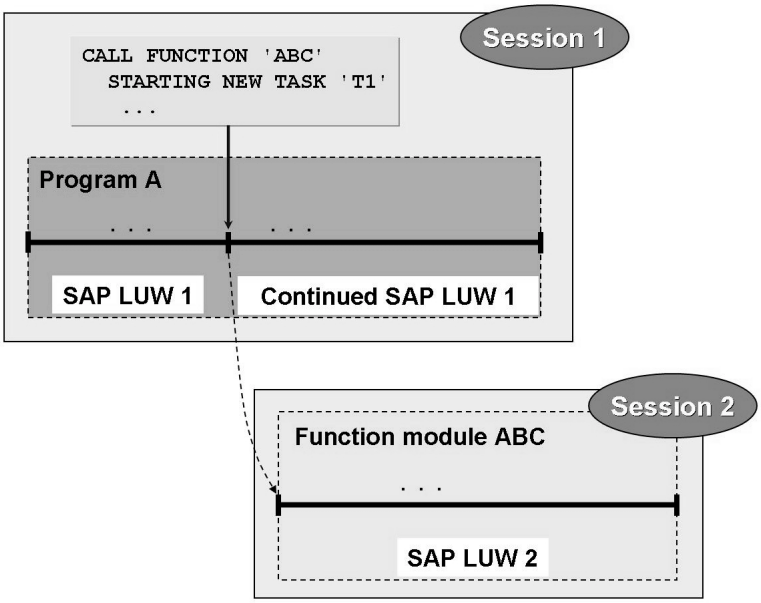


Figure 80: SAP LUWs in Asynchronous Function Calls

A function module that was called asynchronously runs in a separate session and therefore creates its own SAP LUW.

The procesing of the calling program is interrupted briefly, and, after the function module has been triggered, processing is continued. This means that registered update flags and “PERFORM ON COMMIT” subroutines are retained.

Hint: Note, however, that the asynchronous function module call triggers an implicit database commit in the calling program. This means that inline changes processed up to that point will be updated to the database.

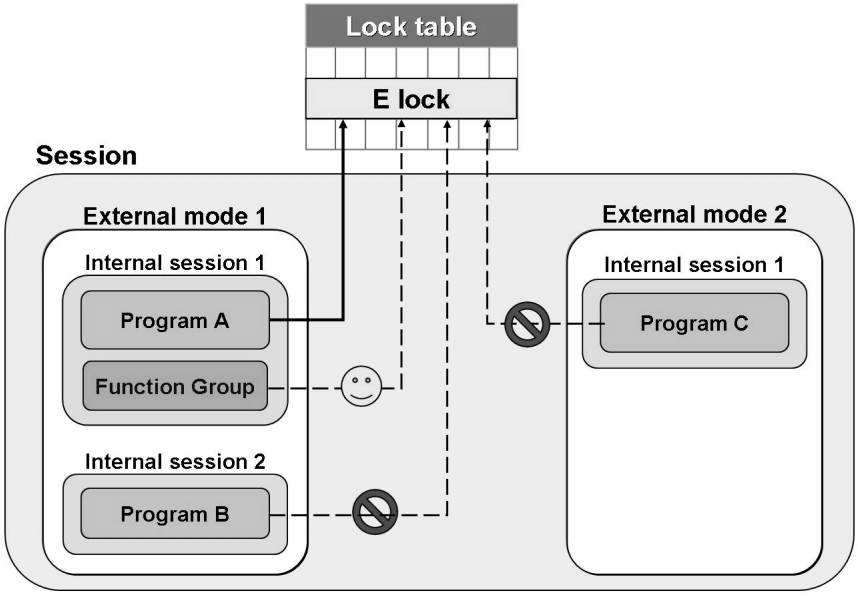


Figure 81: Ability to Accumulate Locks for Program Call

If a lock of type “E” was set in the program for a data record, you can have further locks of type “E” set in the **same program** or in a **function module that was called synchronously**. The cumulation counter of the original lock is increased only by 1.

However, you cannot set further locks to existing ones from within a program called using `SUBMIT AND RETURN` or `CALL TRANSACTION`. Any lock attempts from such programs will be rejected with the exception FOREIGN_LOCK .

If you have a program called using `SUBMIT` or `LEAVE TO TRANSACTION`, the calling program will be terminated immediately. All the locks set up to that point will be automatically deleted. Therefore, no lock conflicts between the calling and the called program can arise.

Lock requests of the same user from different R/3 windows (external sessions) or sessions will be treated as lock requests from different users.

Remote	Data	SAP LUW	Suitable for
CALL FUNCTION	custom	same	- Data display - Data change (within the same LUW)
SUBMIT AND RETURN CALL TRANSACTION	custom	custom	- Data display - Data change (in different LUWs)
CALL FUNCTION ... STARTING NEW TASK	custom	custom	- Parallel tasks

Figure 82: Implementation of Different Program Calls

Programs called by `SUBMIT AND RETURN` or `CALL TRANSACTION` have their own independent SAP LUW. You can use these to perform nested (complex) LUW processing.

Asynchronously called function modules are suitable for tasks that can be processed in parallel. Note that an asynchronously called function module runs in a new session and therefore has its own independent SAP memory. You should therefore use the function module interface for data transfer.

Sample application of the asynchronous function call:

In your program, you want to call up a display transaction that is displayed in a separate window (amodal). Call a function module asynchronously that itself calls the transaction through `CALL TRANSACTION`. Using the function module interface, you pass on the values that are written to the SAP memory (as data transfer to the transaction) before the transaction is called.

Exercise 6: Complex LUW Processing

Exercise Objectives

After completing this exercise, you will be able to:

- Using the "CALL TRANSACTION" technique for modularization.
- Use the SAP memory to transfer data.

Business Example

You want to use the SAP memory to transfer data when calling transactions.

Task 1: CALL TRANSACTION

Program: SAPMZ##_BOOKINGS4

Transaction code: Z##_BOOKINGS4

Copy from: SAPBC414**T**_BOOKINGS_04 / SAPBC414**T**_CREATE_CUSTOMER_03

Model Solution: SAPBC414**S**_BOOKINGS_04 / SAPBC414**S**_CREATE_CUSTOMER

1. Copy your solution **SAPMZ##_BOOKINGS3** or the program template **SAPBC414T_BOOKINGS_04** with **all** subobjects to **SAPMZ##_BOOKINGS4** (## is your group number). Assign transaction code **Z##_BOOKINGS4** to the program.
2. Copy yor solution **SAPMZ##_CUSTOMER2** or the template **SAPBC414T_CREATE_CUSTOMER_03** with **all** subobjects to **SAPMZ##_CUSTOMER3** (## is your group number) and assign the transaction code **Z##_CUSTOMER3** to the program.
3. You can enter new bookings in the copied flight booking program **SAPMZ##_BOOKINGS4** (screen 300). However, if the customer making the booking is not yet maintained in the system, it should be possible to create this customer through the icon "Create New Customer" (function code NEW_CUSTOM) on screen 300 (before you enter his or her booking).

Continued on next page

You need to call the corresponding transaction **Z##_CUSTOMER3** via `CALL TRANSACTION` for this. The transaction call must be encapsulated in the subroutine **CREATE_NEW_CUSTOMER** which is called from the PAI module **USER_COMMAND_0300** (screen 300) and has already been created in an empty state.

Implement the transaction call.

Task 2: Data Transfer Using SAP Memory

The new customer number is defined through internal number assignment within the called program **SAPMZ##_CUSTOMER3**, that is, it is assigned automatically by the program. The SAP memory should be used (see note 1 below) for transferring the customer number to the calling program (for display on the flight booking screen 300/301).

1. Change the **SAPMZ##_CUSTOMER3** program so that the customer number is written to the SAP memory after a customer has been created successfully (see note 2 below).
2. Change the calling program **SAPMZ##_BOOKINGS4** so that the customer number appears in the appropriate field of subscreen 301 after a customer has been created successfully (see note 3 below).

Hint:

1. You can display the name of the SET/GET parameter that is assigned to this field via the **F1 Help** for a screen field. In this case: **Customer Number** field of the **subscreen 301** in the flight booking program .
2. In the **SAPMZ##_CUSTOMER3** program, a customer number is determined for the new customer to be created and stored in the field **SCUSTOM-ID** by calling the subroutine NUMBER_GET_NEXT.
3. There are two implementation options:

   a) By setting the internal program field that supplies the screen field with data. (The customer number field on subcreen 301 is supplied with data by the internal program field **WA_SBOOK-CUSTOMID**.)

   b) Using the GET function of the screen field (field characteristics)

Solution 6: Complex LUW Processing

Task 1: CALL TRANSACTION

Program: SAPMZ##_BOOKINGS4

Transaction code: Z##_BOOKINGS4

Copy from: SAPBC414**T**_BOOKINGS_04 / SAPBC414**T**_CREATE_CUSTOMER_03

Model Solution: SAPBC414**S**_BOOKINGS_04 / SAPBC414**S**_CREATE_CUSTOMER

1. Copy your solution **SAPMZ##_BOOKINGS3** or the program template **SAPBC414T_BOOKINGS_04** with **all** subobjects to **SAPMZ##_BOOKINGS4** (## is your group number). Assign transaction code **Z##_BOOKINGS4** to the program.

   a) -

2. Copy yor solution **SAPMZ##_CUSTOMER2** or the template **SAPBC414T_CREATE_CUSTOMER_03** with **all** subobjects to **SAPMZ##_CUSTOMER3** (## is your group number) and assign the transaction code **Z##_CUSTOMER3** to the program.

   a) -

3. You can enter new bookings in the copied flight booking program **SAPMZ##_BOOKINGS4** (screen 300). However, if the customer making the booking is not yet maintained in the system, it should be possible to create this customer through the icon “Create New Customer” (function code NEW_CUSTOM) on screen 300 (before you enter his or her booking).

   You need to call the corresponding transaction **Z##_CUSTOMER3** via `CALL TRANSACTION` for this. The transaction call must be encapsulated in the subroutine **CREATE_NEW_CUSTOMER** which is called from the PAI module **USER_COMMAND_0300** (screen 300) and has already been created in an empty state.

   Implement the transaction call.

   a) See model solution

Continued on next page

Task 2: Data Transfer Using SAP Memory

The new customer number is defined through internal number assignment within the called program **SAPMZ##_CUSTOMER3**, that is, it is assigned automatically by the program. The SAP memory should be used (see note 1 below) for transferring the customer number to the calling program (for display on the flight booking screen 300/301).

1. Change the **SAPMZ##_CUSTOMER3** program so that the customer number is written to the SAP memory after a customer has been created successfully (see note 2 below).

   a) See model solution

2. Change the calling program **SAPMZ##_BOOKINGS4** so that the customer number appears in the appropriate field of subscreen 301 after a customer has been created successfully (see note 3 below).

Hint:

1. You can display the name of the SET/GET parameter that is assigned to this field via the **F1 Help** for a screen field. In this case: **Customer Number** field of the **subscreen 301** in the flight booking program .
2. In the **SAPMZ##_CUSTOMER3** program, a customer number is determined for the new customer to be created and stored in the field **SCUSTOM-ID** by calling the subroutine NUMBER_GET_NEXT.
3. There are two implementation options:

   a) By setting the internal program field that supplies the screen field with data. (The customer number field on subcreen 301 is supplied with data by the internal program field **WA_SBOOK-CUSTOMID**.)

   b) Using the GET function of the screen field (field characteristics)

a) See model solution

Result

Model Solution SAPBC414S_BOOKINGS_04

Continued on next page

Internal Use SAP Partner Only

FORM Routines F01

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGS_04F01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  CREATE_NEW_CUSTOMER
*&---------------------------------------------------------------------*
FORM create_new_customer.
  CALL TRANSACTION 'BC414S_CREATE_CUST'.
  GET PARAMETER ID 'CSM' FIELD wa_sbook-customid.
* Alternative solution using the GET function of
* the screen field for customer ID
* CLEAR wa_sbook-customid.
ENDFORM.                               " CREATE_NEW_CUSTOMER
```

Model Solution SAPBC414S_CREATE_CUSTOMER

FORM Routines F01

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERF01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  SAVE_SCUSTOM
*&---------------------------------------------------------------------*
FORM save_scustom.
  INSERT INTO scustom VALUES scustom.
  IF sy-subrc <> 0.
    MESSAGE a048.
  ELSE.
    SET PARAMETER ID 'CSM' FIELD scustom-id.
    MESSAGE s015 WITH scustom-id.
  ENDIF.
ENDFORM.                               " SAVE_SCUSTOM
```

Lesson Summary

You should now be able to:

- Explain the LUW logic for program-controlled program calls
- Perform complex LUW processing
- Use the SAP lock mechanism for complex LUW processing.

Unit Summary

You should now be able to:

- Call existing programs from your program using different techniques
- Explain the runtime architecture and the storage access options of these programs (ABAP memory and SAP memory).
- Appropriately implement the different methods for data transfer between your program and the programs called from within your program.
- Explain the LUW logic for program-controlled program calls
- Perform complex LUW processing
- Use the SAP lock mechanism for complex LUW processing.

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 6

Appendix

Unit Overview

- Number assignment with additional exercise and solution
- Document creation with additional exercise and solution
- authorization checks
- Additional information
- Complete solution program “Creating customer data”
- Complete solution program “Creating/canceling a posting”

Unit Objectives

After completing this unit, you will be able to:

- Create number range objects and maintain number range intervals
- Use function modules to determine the next free number of an interval with internal number assignment
- Use function modules to check the validity of numbers assigned externally
- Locate information on function modules for managing number ranges
- Explain the meaning of buffered number assignment.
- Create change-document objects
- Create change documents
- Read change documents
- Find information on the function groups for managing change documents.
- Find information on authorization objects
- Create authorization objects
- Find information on authorizations and profiles
- Perform authorization checks in your program
- Link the execution of transaction codes to authorization objects
- Explain the SAP table buffer.
- Explain Native SQL.
- Explain cluster tables.
- Explain SAP locks.

- Explain BAPI architecture and the use of BAPIs.
- Use the ABAP statements COMMIT WORK and ROLLBACK WORK appropriately.

Unit Contents

Internal Use SAP Partner Only

Lesson: Number Assignment

Lesson Overview

This lesson explains how to assign unique IDs to newly created data records.

Lesson Objectives

After completing this lesson, you will be able to:

- Create number range objects and maintain number range intervals
- Use function modules to determine the next free number of an interval with internal number assignment
- Use function modules to check the validity of numbers assigned externally
- Locate information on function modules for managing number ranges
- Explain the meaning of buffered number assignment.

Business Example

You want to assign a unique ID for data records that are to be created.

Number Range Objects and Intervals

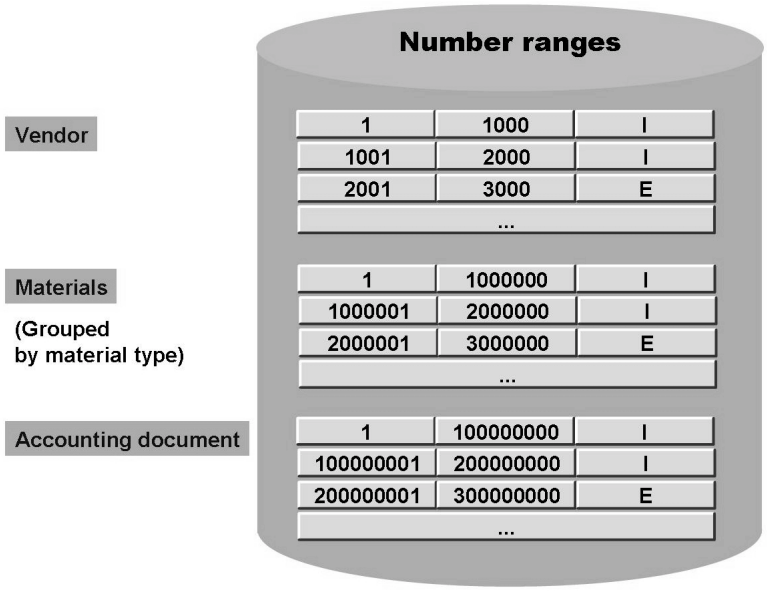


Figure 83: Overview

During number assignment, business objects are assigned to specific number intervals.

Number assignment takes place either externally or internally. With external number assignment, the user inputs a number that is then checked by the system to see if it has been reserved for external assignment. With internal number assignment, the system automatically assigns a number to a business object. Thus, there are different number ranges for external and internal number assignment.

When assigning numbers, you should always pay attention to any regulations and application-specific criteria that apply (for example, mandatory unbroken or chronological number sequences).

In order to assign numbers without any problems, you need a corresponding number range object, a well-maintained number range interval, and access to both of these from within your application program (with the help of special function modules).

Number range documentation can be found in the ABAP Workbench Documentation under "*Expanded Function Builder Applications*".

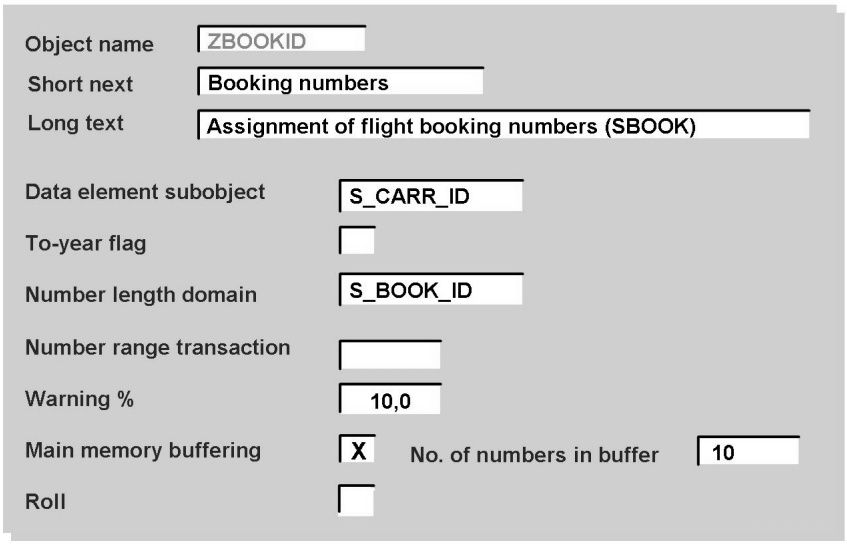


Figure 84: Number Range Objects

Number range object maintenance is located in the development menu under other tools (transaction SNRO).

Number range object names for customer objects should always begin with Y or Z.

Number range objects can be organized according to subobjects (for example, according to company code, user department, or airline carrier).

If you want your number range objects to be organized according to fiscal year, select the field To-year flag.

The length of numbers (maximum number of characters) cannot be entered directly, you must enter a domain instead. This domain can be either type NUMC or type CHAR and can have a field length of up to 20 characters.

If you enter a transaction code in the “number range transaction” field, you must maintain the intervals for exactly this object when the code is called.

You must insert a value between 0.1 and 99.9 in the Warning % field. This determines with which percentage of remaining interval a warning is displayed.

The system automatically suggests a main memory buffering value. You can, however, define this value yourself using the Edit menu. Note that buffering should only be turned off in exceptional cases (for example, when unbroken number sequences are necessary).

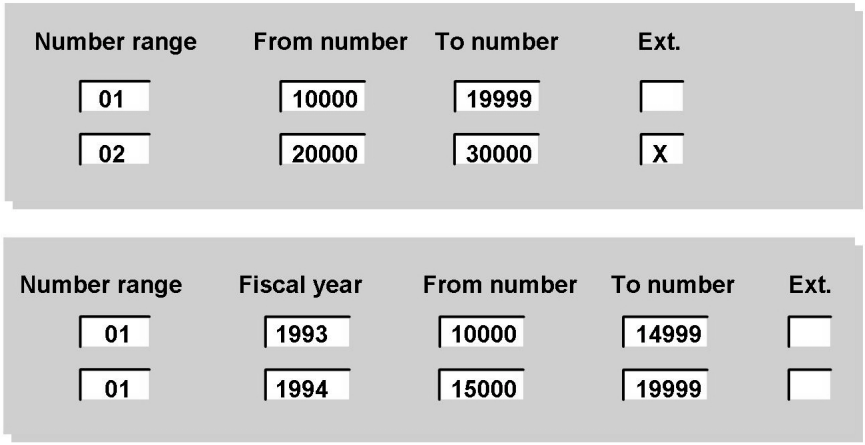


Figure 85: Number Range Intervals

Number range intervals are composed of either numbers or alphanumeric characters (external number ranges only).

The number range number identifies a number range. It can be numeric or alphanumeric.

Multiple intervals can be assigned to a single number range, especially when objects are being organized according to To fiscal year.

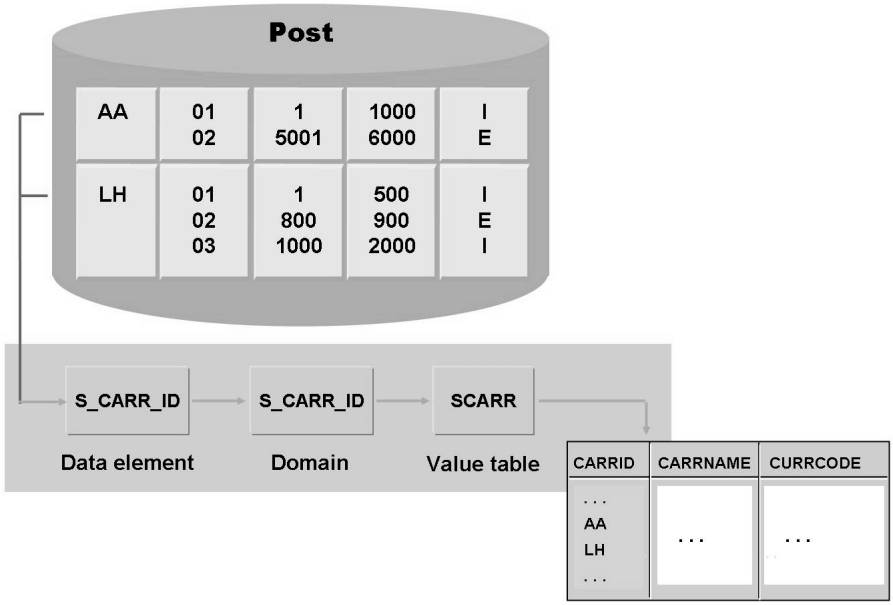


Figure 86: Subobjects

Number range objects can be subdivided into subobjects.

You can, for example, organize documents according to company code, or flight bookings according to airline.

When defining a subobject, you must declare a corresponding data element.

Your sub-object's data element must, by way of its domain, refer to a value table and can have a field length of up to six characters.

Function Modules for Number Management

```
CALL FUNCTION 'NUMBER_GET_NEXT'

  EXPORTING
     nr_range_nr   = <number_range_number>
     object        = <number_range_object_name>
     quantity      = <quantity_of_numbers>
     subobject     = <number_range_sub_object_name>
     toyear        = <to_fiscal_year>
     ignore_buffer = <ignore_buffer_flag>

  IMPORTING
     number        = <next_free_number>
     quantity      = <quantity_of_numbers_to_read>
     returncode    = <return_code>

  EXCEPTIONS
     interval_not_found      = 1
     number_range_not_intern = 2
     object_not_found        = 3
     interval_overflow       = 4.
```

Figure 87: Getting Numbers (From Internal Number Range)

When assigning numbers internally, call the function module NUMBER_GET_NEXT to determine the next number(s) available.

If you have requested more than one number using the import parameter QUANTITY, the export parameter QUANTITY will show the number of assigned numbers, while the export parameter NUMBER will show the last number assigned. The assigned numbers are then in the interval (NUMBER - Export-QUANTITY + 1 to NUMBER.

If the last number of the interval has been assigned, the number assignment begins again at the first number of the interval, provided “Roll” was specified in the definition of the number range object.

The return code allows you to see if the number can be assigned without any problems, or if it lies within a critical area.

- SPACE: Number assignment successful
- 1: Number assignment successful but the number of unassigned numbers is critical
- 2: Number assignment successful; the last available number was just assigned to you
- 3: You requested more numbers than were available. The available numbers were assigned to you.

For more detailed information, refer to the function module documentation.

```
CALL FUNCTION 'NUMBER_CHECK'

  EXPORTING
    nr_range_nr   = <number_range_number>
    number        = <number_to_be_checked>
    object        = <number_range_object_name>
    subobject     = <number_range_sub_object_name>
    toyear        = <to_fiscal_year>

  IMPORTING
    returncode    = <return_code>

  EXCEPTIONS
    interval_not_found          = 1
    number_range_not_extern     = 2
    object_not_found            = 3.
```

Figure 88: Check Number (in the External Number Range)

Use the function module NUMBER_CHECK for checking external numbers to see if they lie within a number range interval that has been designated for external use. The checked number is **not** marked by the function moodule as assigned.

The check results are returned through the export parameter returncode.

- SPACE: Number inside the specified interval
- 'X': Number outside of the specified interval

For more detailed information, refer to the function module documentation.

```
CALL FUNCTION 'NUMBER_GET_INFO'

  EXPORTING
    nr_range_nr   = <number_range_number>
    object        = <number_range_object_name>
    subobject     = <number_range_sub_object_name>
    toyear        = <to_fiscal_year>

  IMPORTING
    interval      = <interval_info>


  EXCEPTIONS
    interval_not_found   = 1
    object_not_found     = 2.
```

Figure 89: Getting Number Range Information

Use the function module NUMBER_GET_INFO to get information about the individual number range intervals of number range objects (interval limits, current status, and so on).

This information is transferred as a structure using the INTERVAL parameter. The parameter has the same structure as the lines of the transparent NRIV table.

For more detailed information, refer to the function module documentation.

Buffering Numbers

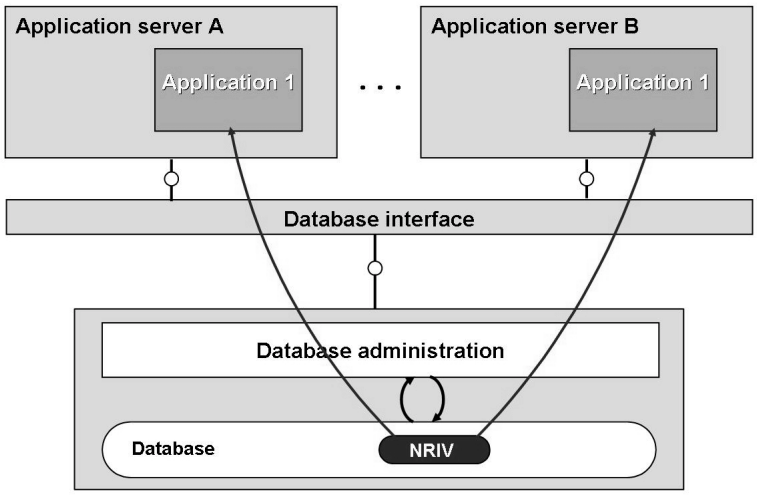


Figure 90: Accessing Table NRIV Without a Buffer (1)

The data for number ranges is stored in table NRIV. In this table, an entry exists for each individual number range.

If a number range object has not been buffered, the system must read directly from the database each time new numbers are assigned. This leads to the corresponding number range being locked until completion of the current LUW.

The advantages of non-buffered access are:

- number assignment without gaps
- chronological sequence of numbers

Disadvantages are:

- Serialization due to database lock
- Possible mulitiplication effects (see next figure)

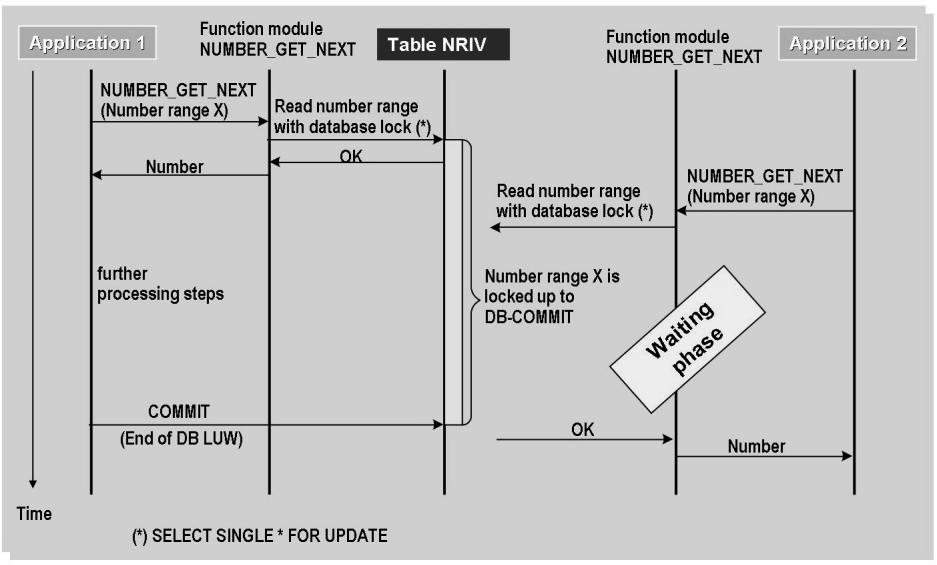


Figure 91: Accessing Table NRIV Without a Buffer (2)

In the function module NUMBER_GET_NEXT, the statement SELECT SINGLE ... FOR UPDATE is used to access table NRIV. This locks the relevant table entry in the database until completion of the current LUW.

This can lead to considerable waiting time when multiple programs request numbers from the same number range.

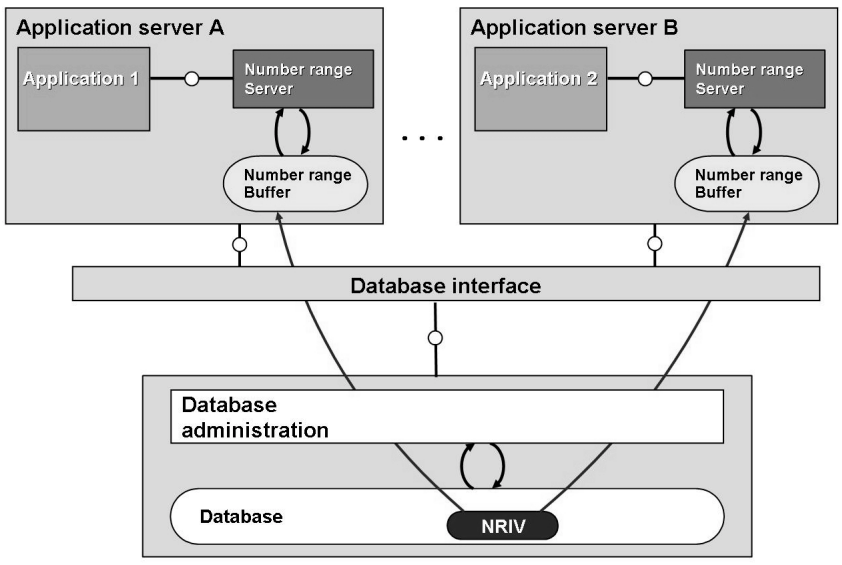


Figure 92: Access Through Number Range Buffer (Standard)

You can enhance performance considerably by buffering number ranges (those that the system suggests by default). Buffering transfers responsibility for most number assignments to your application server.

Your buffer refills itself with a new number 'packet' (= quantity of numbers) as soon as all its numbers have been assigned. You determine how many numbers are stored in the buffer yourself during number range object maintenance.

With this kind of buffering, the application server only has to read directly from the database after it has assigned all of the numbers in its current packet. This leads to better performance and makes clear why number range objects should be buffered as long as application logic and regulations allow.

Function module NUMBER_GET_NEXT offers you the option of ignoring the buffer if you so desire (interface parameter IGNORE_BUFFER).

Number range servers are logical units on the application server and run their own LUWs. Thus gaps can occur in number assignments due to rollback or network errors.

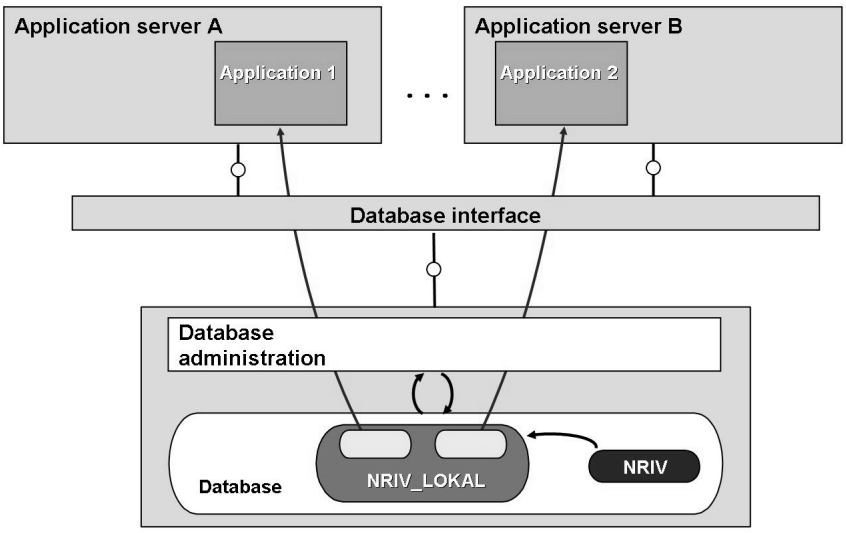


Figure 93: Parallel Access to Number Ranges

Another means of assigning buffered numbers is to buffer the numbers block by block for each application server on the database.

This enables parallel accesses from various servers. Serialization only takes place for each individual server.

All numbers are assigned. There are no gaps. Numbers, however, are usually not assigned in chronological order.

Using Function Groups to Manage Number Ranges

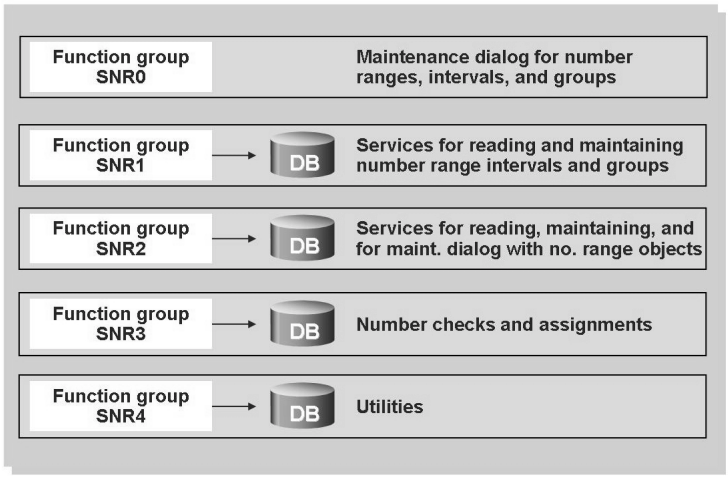


Figure 94: Using Function Groups to Manage Number Ranges

For processing number range objects, number ranges, and groups, there are additional function modules available that allow you to edit dialogs, database accesses, and other tasks.

These function modules are divided into five different function groups.

If you are only interested in using standard number range functions, then you only need the function modules found in function group SNR3.

Exercise 7: Number Assignment

Exercise Objectives

After completing this exercise, you will be able to:

- Implement the technique for internal number assignment in transactions.

Business Example

You want to use the internal number assignment function to assign a unique ID to newly created data records.

Task: Implementation of an internal number assignment

Program: SAPMZ##_BOOKINGS5

Transaction Code: Z##_BOOKINGS5

Copy template: SAPBC414**T**_BOOKINGS_05

Model Solution: SAPBC414**S**_BOOKINGS_05

1. Copy your solution **SAPMZ##_BOOKINGS4** or the program template **SAPBC414T_BOOKINGS_05** with **all** subobjects to **SAPMZ##_BOOKINGS5** (**##** is your group number). Assign transaction code **Z##_BOOKINGS5** to the program.

2. Until now in the copied program the user specifies the booking number when creating a booking. Now the program should be adjusted so that it determines the booking number by the internal number assignment.

   The number assignment should be handled by subroutine **NUMBER_GET_NEXT**. If you have copied the above program template, the subroutine is called from the PAI module `USER_COMMAND_0300` (screen 300) and is already created (empty). If you have copied your previous solution program as a basis for this exercise, you still need to implement the subroutine call in the PAI module `USER_COMMAND_0300`.

   First check the number range object SBOOKID and the assigned number ranges (transactions SNRO/SNUM).

Continued on next page

Insert the call for the function module **NUMBER_GET_NEXT** in the subroutine **NUMBER_GET_NEXT**. The airline abbreviation is to be assigned to the interface parameter SUBOBJECT. Make provisions for the RETURNCODE and the exceptions of the function module. Possible error messages:

- Number set running out → Message 070
- Last number taken → Message 071
- No more numbers available → Message 072
- Internal error with number assignment → Message 073

3. Since the numbers are now assigned by the system, you do not need an input/output field for the booking number on subscreen 301.

   Hide the field for the booking number dynamically. To do so, remove the comment asterisk in front of the module **HIDE_BOOKID** in the flow logic of subscreen 301. The module is already created.

Hint: I the subroutine NUMBER_GET_NEXT the airline name is availble in the data object **P_WA_SBOOK-CARRID**.

The booking number assigned through the number range object must be assigned to the data object **P_WA_SBOOK-BOOKID**.

Solution 7: Number Assignment

Task: Implementation of an internal number assignment

Program: SAPMZ##_BOOKINGS5

Transaction Code: Z##_BOOKINGS5

Copy template: SAPBC414**T**_BOOKINGS_05

Model Solution: SAPBC414**S**_BOOKINGS_05

1. Copy your solution **SAPMZ##_BOOKINGS4** or the program template **SAPBC414T_BOOKINGS_05** with **all** subobjects to **SAPMZ##_BOOKINGS5** (**##** is your group number). Assign transaction code **Z##_BOOKINGS5** to the program.

   a) -

2. Until now in the copied program the user specifies the booking number when creating a booking. Now the program should be adjusted so that it determines the booking number by the internal number assignment.

   The number assignment should be handled by subroutine **NUMBER_GET_NEXT**. If you have copied the above program template, the subroutine is called from the PAI module `USER_COMMAND_0300` (screen 300) and is already created (empty). If you have copied your previous solution program as a basis for this exercise, you still need to implement the subroutine call in the PAI module `USER_COMMAND_0300`.

   First check the number range object SBOOKID and the assigned number ranges (transactions SNRO/SNUM).

   Insert the call for the function module **NUMBER_GET_NEXT** in the subroutine **NUMBER_GET_NEXT**. The airline abbreviation is to be assigned to the interface parameter SUBOBJECT. Make provisions for the RETURNCODE and the exceptions of the function module. Possible error messages:

   - Number set running out → Message 070

   - Last number taken → Message 071

   - No more numbers available → Message 072

   - Internal error with number assignment → Message 073

   a) See model solution

Continued on next page

3. Since the numbers are now assigned by the system, you do not need an input/output field for the booking number on subscreen 301.

   Hide the field for the booking number dynamically. To do so, remove the comment asterisk in front of the module **HIDE_BOOKID** in the flow logic of subscreen 301. The module is already created.

Hint: I the subroutine NUMBER_GET_NEXT the airline name is availble in the data object **P_WA_SBOOK-CARRID**.

The booking number assigned through the number range object must be assigned to the data object **P_WA_SBOOK-BOOKID**.

a) See model solution

Result

Model Solution SAPBC414S_BOOKINGS_05

FORM Routines F01

```
*---------------------------------------------------------------------*
***INCLUDE BC414T_BOOKINGS_05F01
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  NUMBER_GET_NEXT
*&---------------------------------------------------------------------*
FORM number_get_next USING p_wa_sbook LIKE sbook.
  DATA: return TYPE inri-returncode.
* get next free number in the number range '01' of number range
* object 'SBOOKID'
  CALL FUNCTION 'NUMBER_GET_NEXT'
       EXPORTING
            nr_range_nr = '01'
            object      = 'SBOOKID'
            subobject   = p_wa_sbook-carrid
       IMPORTING
            number      = p_wa_sbook-bookid
            returncode  = return
       EXCEPTIONS
            OTHERS      = 1.
  CASE sy-subrc.
```

Continued on next page

Internal Use SAP Partner Only

```
    WHEN 0.
      CASE return.
        WHEN 1.
* number of remaining numbers critical
          MESSAGE s070.
        WHEN 2.
* last number
          MESSAGE s071.
        WHEN 3.
* no free number left over
          MESSAGE a072.
      ENDCASE.
    WHEN 1.
* internal error
      MESSAGE a073 WITH sy-subrc.
  ENDCASE.
ENDFORM.                                " NUMBER_GET_NEXT
```

SCREEN 301

```
PROCESS BEFORE OUTPUT.
  MODULE HIDE_BOOKID.
*
PROCESS AFTER INPUT.
```


Lesson Summary

You should now be able to:

- Create number range objects and maintain number range intervals
- Use function modules to determine the next free number of an interval with internal number assignment
- Use function modules to check the validity of numbers assigned externally
- Locate information on function modules for managing number ranges
- Explain the meaning of buffered number assignment.

Lesson: Creating Change Documents

Lesson Overview

This lesson explains how to create change documents for database updates.

Lesson Objectives

After completing this lesson, you will be able to:

- Create change-document objects
- Create change documents
- Read change documents
- Find information on the function groups for managing change documents.

Business Example

You want to create change documents for each of your database updates.

Principle of Creating Change Documents

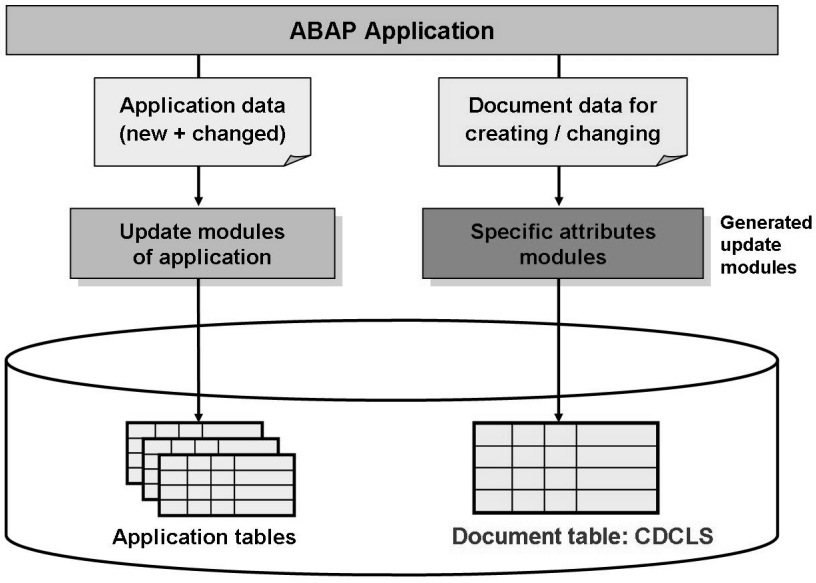


Figure 95: Principle of Creating Change Documents (1)

If you want to log updates to application tables using change documents, you must make sure in your application program that the updates are performed in both the application tables and in the change document table (within the same LUW).

The change documents are written using special function modules (update modules) that are generated using a maintenance transaction. You must use the update module for writing the change document in your program within the LUW that contains the data for the application tables you want to update.

You must transfer the following data to the update module that writes the change documents:

- Application data before it is changed by the program
- Application data updated in the program
- Administration data for the document header (user ID, change date, change time, name of the transaction making the change, and so on).

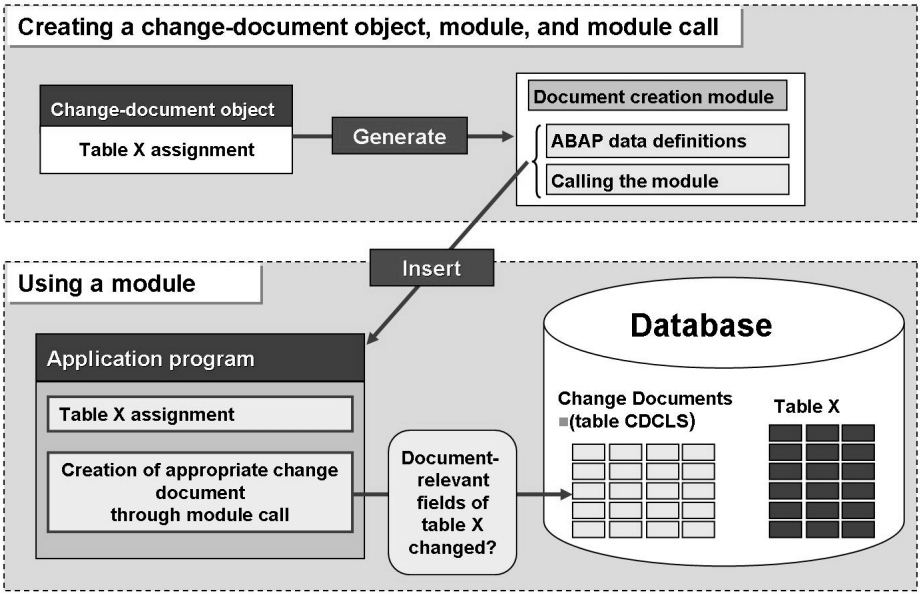


Figure 96: Principle of Creating Change Documents (2)

In order to be able to log changes to a business object in a change document, you must define a change document object in the system.

When defining a change document object, you must also include the corresponding tables.

The system then creates a corresponding document creation function module as well as two Include programs with data object definitions. This takes place during generation or when the document-creation function module is called.

These are then included in the application program and cause change documents to be written when certain requirements are fulfilled.

If you call a document-creation function module, it will only create a document if at least one document-relevant field is changed in the application record. The figure Document-Relevant Fields of a Data Record provides information on application record fields that are document-relevant.

Change documents are stored in cluster table CDCLS (see next figure).

You will find documentation on change documents in the online help (SAP Library) by calling the path *Basis → ABAP Workbench → Extended Function Library Applicationse → Change documents*.

Creating Change Documents

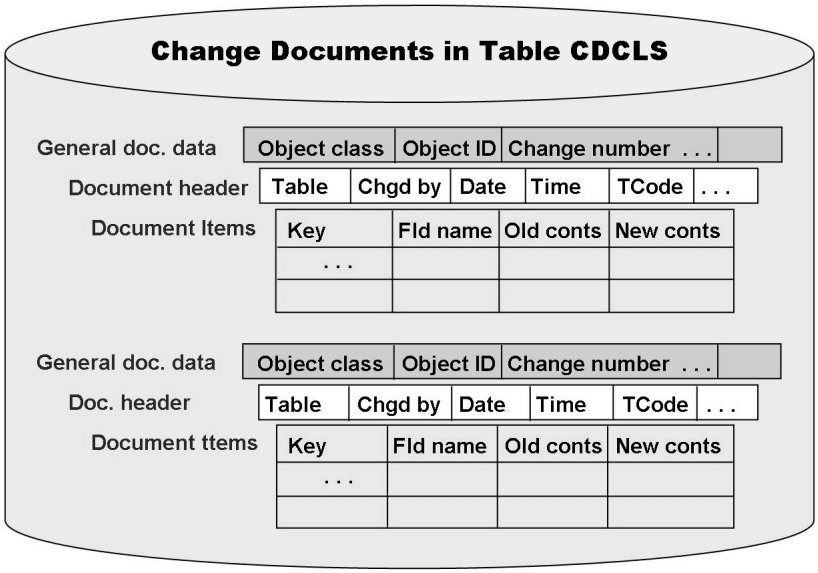


Figure 97: Structure of Change Documents

Cluster table CDCLS contains the structures CDHDR (document header), CDPOS (document item), PCDHDR (planned change document: header), and PCDPOS (planned change document: items).

Change document numbers are automatically generated when a change document is created.

A change document consists of general document data and a document header with the respective document items.

The general document data includes:

- Object Class: Name of the change document object used
- User ID: At document creation, you can decide the name of the document (for later document search, generally tontains the key of the changed records)
- Change Number The number automatically assigned to the document

Generally, multiple document headers and their corresponding document items can belong to one document.

A document header contains the name of the changed table, the person who changed the document, the change date and time, and so on. A document header contains several document items.

A document items constis of the key of the changed record, the name of the changed field, and the old and new field values. Each document item describes the field change in the changed record.

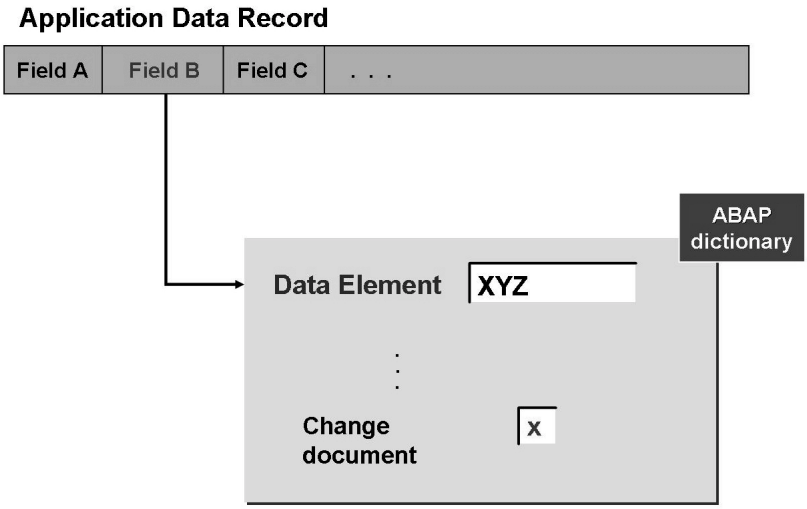


Figure 98: Document-Relevant Fields of a Data Record

When you call the corresponding document-creation function module, it will only create a document if at least one document-relevant field is changed in the application record. Only changes to these fields are logged as document items.

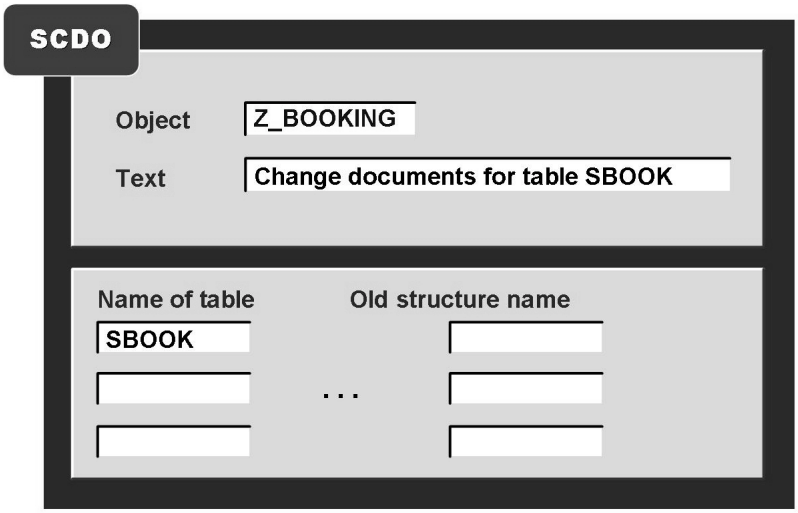


Figure 99: Creating Change-Document Objects

You define a change document object when creating a change document for a business object as follows: *SAP Menu → Tools → ABAP Workbench → Development → Other Tools → Change Documents (Transaction SCDO)*

The name of a customer's change-document object must start with Y_ or Z_.

Change document objects include all tables relevant to the changes made to underlying business objects.

There must be new values in each of the table work areas listed.

For each table work area, you must specify an internal structure variable that contains the old record when it is called up later. By default, the respective *work areas appear (for example, *SFLIGHT).

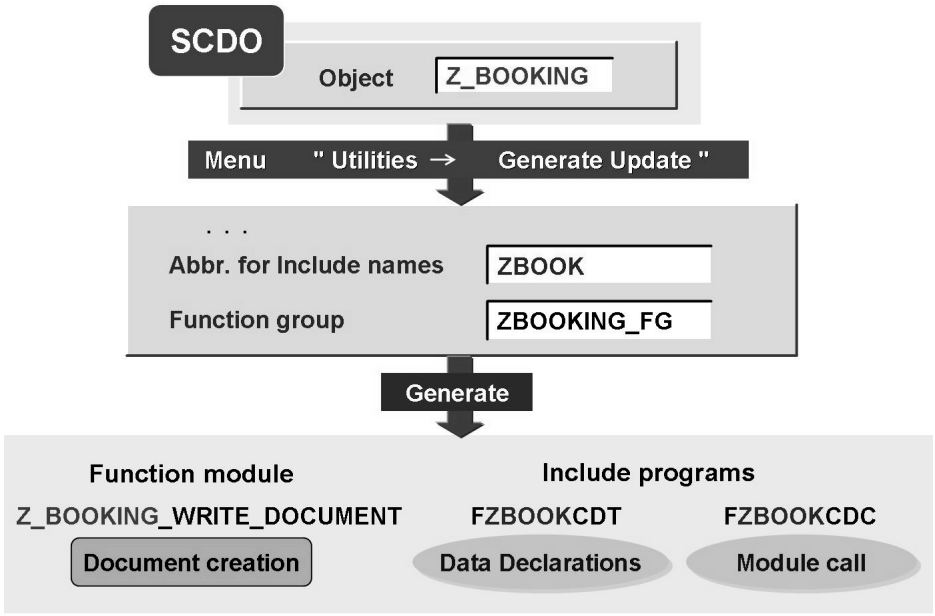


Figure 100: Generating Document-Creation Module and Call

For each defined change-document object, the system can generate the corresponding document-generation function module and also call the module together with the required data declarations (ABAP syntax).

For this purpose, you must specify the name of a function group (to be generated automatically by the system) for storing the function module.

Also, the system requires a randomly selectable abbreviation for assigning the names of the generating program parts (Includes). (see above figure)

After generation, you receive an Include with the required global data definitions for your application program. You also receive an Include program when you call the document-generation function module, which must also be included in your program. You must then include the function module call at an appropriate position in your program.

Include Program FZBOOKCDT

```
INCLUDE FZBOOKCDF.
INCLUDE FZBOOKCDV.
```

Data declarations that are the same for all change objects

Data declarations that are change object specific

Include Program FZBOOKCDC

```
FORM cd_call_z_booking.

  CALL FUNCTION 'Z_BOOKING_WRITE_DOCUMENT'
    IN UPDATE TASK
    EXPORTING
      objectid = ...
      tcode    = ...
       ...
ENDFORM.
```

Full syntax for creating a document when using global data objects

Figure 101: Structure of the generated Includes

The Include **FZBOOKCDT** contains two sub Includes: one with global data declarations that are the same for all change objects and one with data declarations that are specific to the corresponding change object.

The Include **FZBOOKCDC** contains the call for the document-creation function module (embedded in a form routine).

In your application program, you must include both includes and call the form routine for creation of a document at an appropriate position.

Hint: Remember that, before the form routine is called, the global data objects of your program that are used in the function module call must be set accordingly (see next figure).

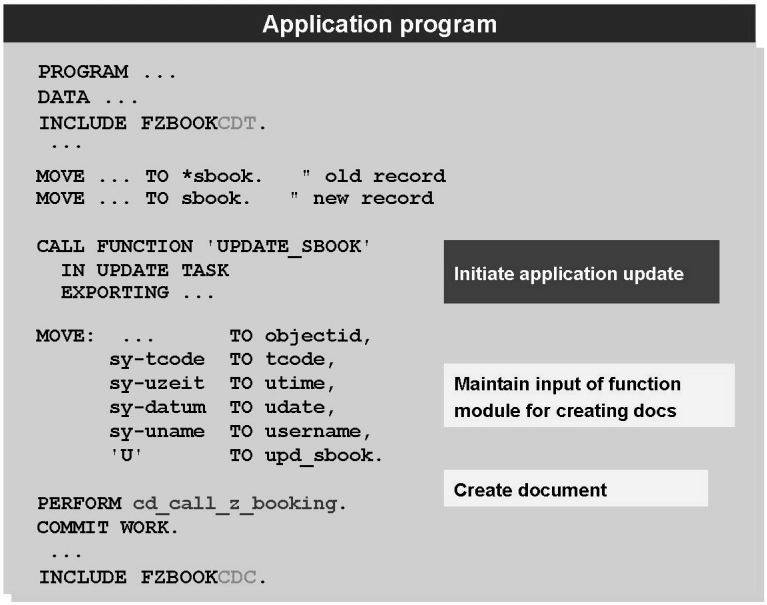


Figure 102: Use of Includes in the Application Program

Include the generated Includes in your dialog program.

Place the old and new data records in their corresponding work areas.

Before you call the included form routine for document creation, you must set the global data objects of your program that are used in the function module call.

Enter a randomly selectable object ID in the field objectid for the changed document to be generated. The object ID is part of the document key. Therefore, you should consider well which ID names you want as this will simplify document search later on. One option would be to specify the key of the changed data record as object ID.

In the global fields *tcode, utime, udate* and *username*, enter the current transaction code, change time and date, as well as the name of the person who made the change.

In the global field *upd_<tab>* you must specify the change ID (U/I/D for Update, Insert, Delete).

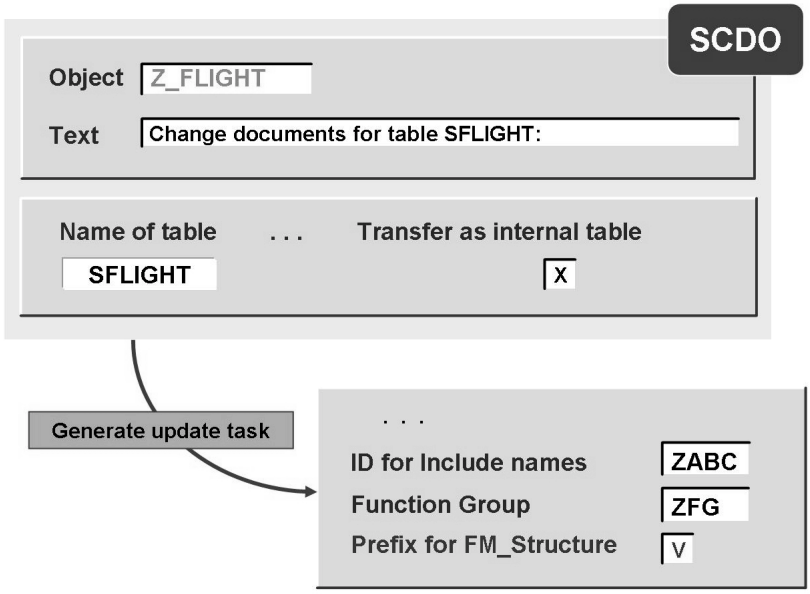


Figure 103: Document for Changing Records That Belong Together

If several objects belong to a business object (for example, if you have an order header and several order items belonging to an order) and these have been changed, you can transfer these to an internal table to be passed to the document function module for creating the corresponding document.

For this purpose, you must maintain the change-document object accordingly (see figure above).

Due to your entries, two internal tables (in this case: XSFLIGHT and YSFLIGHT) were defined in the generated data declaration Include. You must fill these with the new or old data records and return them to the document function module (X table for new records and Y table for old records).

The structure descriptions for both internal tables are generated by the system and stored in the ABAP Dictionary. Their name is made up of the prefix you specified in the change document object and the table name (in this case: VSFLIGHT).

Change Document Management

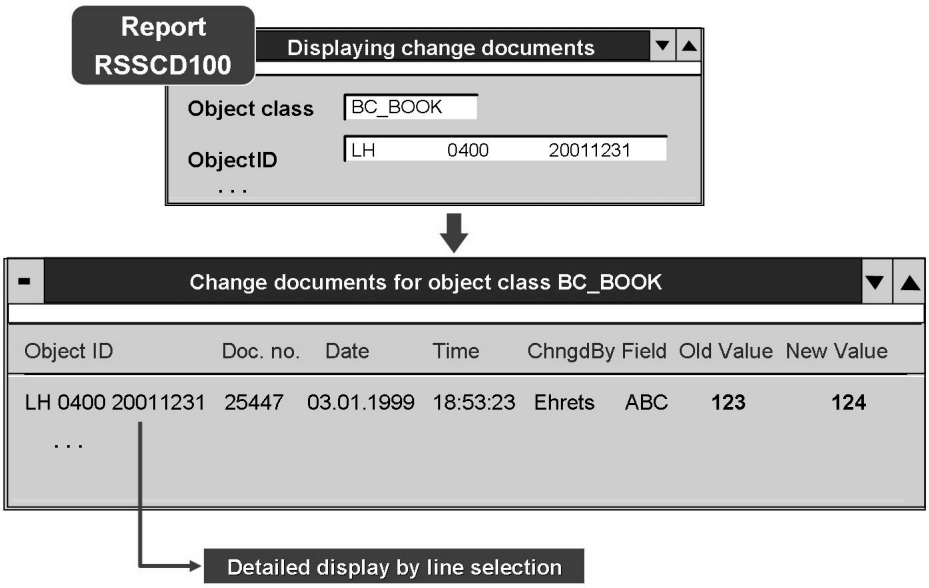


Figure 104: Displaying Change Documents

SAP program **RSSCD100** demonstrates how to evaluate change documents.

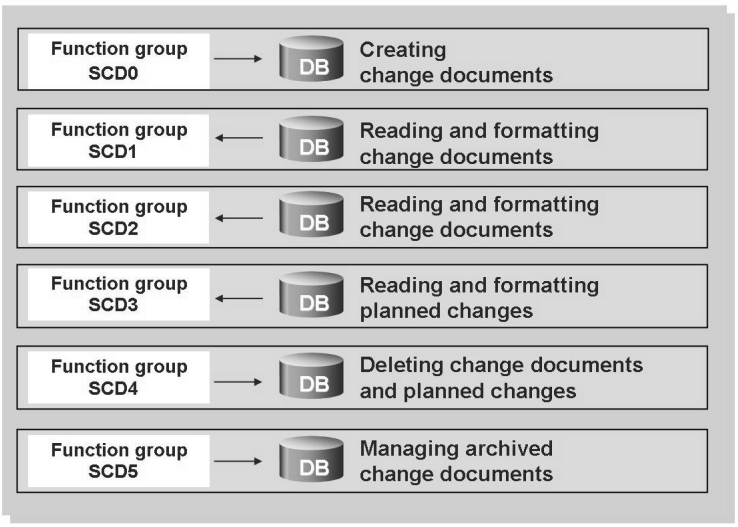


Figure 105: Function Groups for Processing Change Documents

There are SAP function modules available for all the tasks required in processing change documents.

These function modules are grouped together in six function groups.

Function modules of the SCD0 group are called by the document function module that was generated specifically for the object concerned.

Internal Use SAP Partner Only

Exercise 8: Creating Change Documents

Exercise Objectives

After completing this exercise, you will be able to:

- Document database updates by creating change documents from your transaction.

Business Example

You want to create change documents for each of your database updates.

Task: Creating Change Documents

Program: SAPMZ##_BOOKINGS6

Transaction Code: Z##_BOOKINGS6

Copy template: SAPBC414**T**_BOOKINGS_06

Model Solution: SAPBC414**S**_BOOKINGS_06

1. Copy your solution **SAPMZ##_BOOKINGS5** or the program template **SAPBC414T_BOOKINGS_06** with **all** subobjects to **SAPMZ##_BOOKINGS6** (**##** is your group number). Assign transaction code **Z##_BOOKINGS6** to the program.
2. Check the **change-document object BC_BOOK** (transaction **SCDO**).

   Which includes were generated when the update-capable function module **BC_BOOK_WRITE_DOCUMENT** was generated for creating change documents?

   Navigate in the includes and take a look at the source text encapsulated in them, in particular the function module interface. Which structures are used to transfer the original booking data, or the changed booking data to the function module? Where are these structures declared?

   Which of the generated includes must be included in a program to enable change documents to be created? Insert the appropriate statements in your transaction.

Continued on next page

3. The program **SAPMZ##_BOOKINGS6** will now be extended to change documents. These will then be created whenever an existing booking is cancelled. The change document should record modifications to the CANCELLED field of a data record in the DB table SBOOK.

   The creation of change documents is to be encapsulated in the subroutine **CREATE_CHANGE_DOCUMENT**. If you have copied the above program template, the subroutine is called from the PAI module **USER_COMMAND_0200** (screen 200) and is already created (empty). If you have copied your previous solution program as a basis for this exercise, you still need to implement the subroutine call in the PAI module **USER_COMMAND_0200**.

4. A change document is to be generated for every cancelled booking. This can be set by calling up the appropriate function module within a **LOOP** statement using the changed data records.

   Implement the loop. The changed data is maintained in the internal table **ITAB_SBOOK_MODIFY**. Set the data in the **SBOOK** structure.

   Read the corresponding data record (before changing the data) in the internal table **ITAB_CD** and place it in the structure ***SBOOK**. To do this, use the ABAP statement

   ```
   READ TABLE ITAB_CD FROM SBOOK INTO *SBOOK.
   ```

5. Assign the values of the system variables SY-TCODE, SY-UZEIT, SY-DATUM and SY-UNAME to the corresponding variables **TCODE**, **UTIME**, **UDATE** and **USERNAME**.

   Also assign the value "**U**" to **UPD_SBOOK** and the object key defined by you to **OBJECTID**. For example you can use the concatenation of the key fields from the booking record as the object key (ABAP keyword **CONCATENATE**).

   (The set global variables are used later on when the document-creation function module is called.)

6. At the end of each loop, call the subroutine **CD_CALL_BC_BOOK** to create a document.

   (In this subroutine, the call for the document-creation function module **BC_BOOK_WRITE_DOCUMENT** is encapsulated.)

7. Start your program and cancel some bookings. Use the program **RSSCD100** to check whether change documents have been created.

Hint: The internal table **ITAB_CD** is then refilled with data whenever screen 200 is called up from start screen 100 to cancel existing bookings.

Solution 8: Creating Change Documents

Task: Creating Change Documents

Program: SAPMZ##_BOOKINGS6

Transaction Code: Z##_BOOKINGS6

Copy template: SAPBC414**T**_BOOKINGS_06

Model Solution: SAPBC414**S**_BOOKINGS_06

1. Copy your solution **SAPMZ##_BOOKINGS5** or the program template **SAPBC414T_BOOKINGS_06** with **all** subobjects to **SAPMZ##_BOOKINGS6** (## is your group number). Assign transaction code **Z##_BOOKINGS6** to the program.

   a) -

2. Check the **change-document object BC_BOOK** (transaction **SCDO**).

   Which includes were generated when the update-capable function module **BC_BOOK_WRITE_DOCUMENT** was generated for creating change documents?

   Navigate in the includes and take a look at the source text encapsulated in them, in particular the function module interface. Which structures are used to transfer the original booking data, or the changed booking data to the function module? Where are these structures declared?

Continued on next page

Which of the generated includes must be included in a program to enable change documents to be created? Insert the appropriate statements in your transaction.

a) Generated includes:

- **FBC414_CDOCSCDT** (data definitions)
- **FBC414_CDOCSCDF** (data definitions)
- **FBC414_CDOCSCDV** (data definitions)
- **FBC414_CDOCSCDC** (contains a subroutine that encapsulates the call for the document-creation function module.)

Relevant interface parameters of function module **BC_BOOK_WRITE_DOCUMENT**:

- **O_SBOOK** (data before change)
- **N_SBOOK** (data after change)

The structures (table work areas) SBOOK and *SBOOK are declared in the generated Include FBC414_CDOCSCDV

Includes to be included:

- **FBC414_CDOCSCDT** (contains FBC414_CDOCSCDF and FBC414_CDOCSCDV)
- **FBC414_CDOCSCDC**

3. The program **SAPMZ##_BOOKINGS6** will now be extended to change documents. These will then be created whenever an existing booking is cancelled. The change document should record modifications to the CANCELLED field of a data record in the DB table SBOOK.

   The creation of change documents is to be encapsulated in the subroutine **CREATE_CHANGE_DOCUMENT**. If you have copied the above program template, the subroutine is called from the PAI module **USER_COMMAND_0200** (screen 200) and is already created (empty). If you have copied your previous solution program as a basis for this exercise, you still need to implement the subroutine call in the PAI module **USER_COMMAND_0200**.

   a) See model solution

4. A change document is to be generated for every cancelled booking. This can be set by calling up the appropriate function module within a **LOOP** statement using the changed data records.

   Implement the loop. The changed data is maintained in the internal table **ITAB_SBOOK_MODIFY**. Set the data in the **SBOOK** structure.

Continued on next page

Internal Use SAP Partner Only

Read the corresponding data record (before changing the data) in the internal table **ITAB_CD** and place it in the structure ***SBOOK**. To do this, use the ABAP statement

```
READ TABLE ITAB_CD FROM SBOOK INTO *SBOOK.
```

a) See model solution

5. Assign the values of the system variables SY-TCODE, SY-UZEIT, SY-DATUM and SY-UNAME to the corresponding variables **TCODE**, **UTIME**, **UDATE** and **USERNAME**.

   Also assign the value "**U**" to **UPD_SBOOK** and the object key defined by you to **OBJECTID**. For example you can use the concatenation of the key fields from the booking record as the object key (ABAP keyword **CONCATENATE**).

   (The set global variables are used later on when the document-creation function module is called.)

   a) See model solution

6. At the end of each loop, call the subroutine **CD_CALL_BC_BOOK** to create a document.

   (In this subroutine, the call for the document-creation function module **BC_BOOK_WRITE_DOCUMENT** is encapsulated.)

   a) See model solution

7. Start your program and cancel some bookings. Use the program **RSSCD100** to check whether change documents have been created.

Hint: The internal table **ITAB_CD** is then refilled with data whenever screen 200 is called up from start screen 100 to cancel existing bookings.

a) -

Result

Model Solution SAPBC414S_BOOKINGS_06

Module Pool

```
*&---------------------------------------------------------------------*
*& Modulpool         SAPBC414S_BOOKINGS_06                           *
*&---------------------------------------------------------------------*
```

Continued on next page

Internal Use SAP Partner Only

```
INCLUDE bc414s_bookings_06top.
INCLUDE bc414s_bookings_06o01.
INCLUDE bc414s_bookings_06i01.
INCLUDE bc414s_bookings_06f01.
INCLUDE bc414s_bookings_06f02.
INCLUDE bc414s_bookings_06f03.
INCLUDE bc414s_bookings_06f04.
INCLUDE bc414s_bookings_06f05.
INCLUDE bc414s_bookings_06f06.
* The following include contains the subroutine, which is called to
* create a change document for each booking
INCLUDE fbc414_cdocscdc.
```

TOP Include

```
*&---------------------------------------------------------------------*
*& Include BC414S_BOOKINGS_06TOP                                       *
*&---------------------------------------------------------------------*
PROGRAM  sapbc414s_bookings_06 MESSAGE-ID bc414.

* Data definitions for creating change documents
INCLUDE fbc414_cdocscdt.

* Line type of internal table itab_book, used to display bookings in
* table control
TYPES: BEGIN OF wa_book_type.
         INCLUDE STRUCTURE sbook.
TYPES:   name TYPE scustom-name,
         mark,
       END OF wa_book_type.

* Work area and internal table used to display bookings in table
* control
DATA: wa_book TYPE wa_book_type,
      itab_book TYPE TABLE OF wa_book_type.
* Bookings to be modified on database table
DATA: itab_sbook_modify TYPE TABLE OF sbook.

* For change documents: Internal table for bookings before change
DATA: itab_cd TYPE TABLE OF sbook WITH NON-UNIQUE KEY
      carrid connid fldate bookid customid.

* Work areas for database tables spfli, sflight, sbook.
```

Continued on next page


```
DATA: wa_sbook TYPE sbook,
      wa_sflight TYPE sflight,
      wa_spfli TYPE spfli.

* Complex transaction: Customer ID created in the called transaction
data: scust_id(20).

* Transport function codes from screens
DATA: ok_code TYPE sy-ucomm,
      save_ok LIKE ok_code.
* Define subscreen number on tabstrip, screen 300
DATA: screen_no TYPE sy-dynnr.
* used to handle sy-subrc, which is determined in subroutine
DATA  sysubrc LIKE sy-subrc.

* For field transport to/from screen
TABLES: sdyn_conn, sdyn_book.
* Table control declaration (display bookings),
* tabstrip declaration (create booking)
CONTROLS: tc_sbook TYPE TABLEVIEW USING SCREEN '0200',
          tab TYPE TABSTRIP.
```

FORM Routines F06

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGS_06F06
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  CREATE_CHANGE_DOCUMENTS
*&---------------------------------------------------------------------*
FORM create_change_documents.
  LOOP AT itab_sbook_modify INTO sbook.
* read unchanged data from buffer table into *-work area
    READ TABLE itab_cd FROM sbook INTO *sbook.
* define objectid from key fields of sbook
    CONCATENATE sbook-mandt sbook-carrid sbook-connid
                sbook-fldate sbook-bookid sbook-customid
                INTO objectid SEPARATED BY space.
* fill interface parameters for function call (which itself is
* encapsulated in form CD_CALL_BC_BOOK
    MOVE: sy-tcode        TO tcode,
          sy-uzeit        TO utime,
```

Continued on next page


```
          sy-datum        TO udate,
          sy-uname        TO username,
          'U'             TO upd_sbook.
* perform calls the neccessary function to create change document
* 'in update task'
    PERFORM cd_call_bc_book.
  ENDLOOP.
ENDFORM.                               " CREATE_CHANGE_DOCUMENTS
```

Lesson Summary

You should now be able to:

- Create change-document objects
- Create change documents
- Read change documents
- Find information on the function groups for managing change documents.

Lesson: Authorization Checks

Lesson Overview

Contents:

- Authorization Objects
- Authorizations
- Authorization checks

Lesson Objectives

After completing this lesson, you will be able to:

- Find information on authorization objects
- Create authorization objects
- Find information on authorizations and profiles
- Perform authorization checks in your program
- Link the execution of transaction codes to authorization objects

Business Example

You want to link the execution of transaction codes to authorization objects

Program-Controlled Authorization Checks

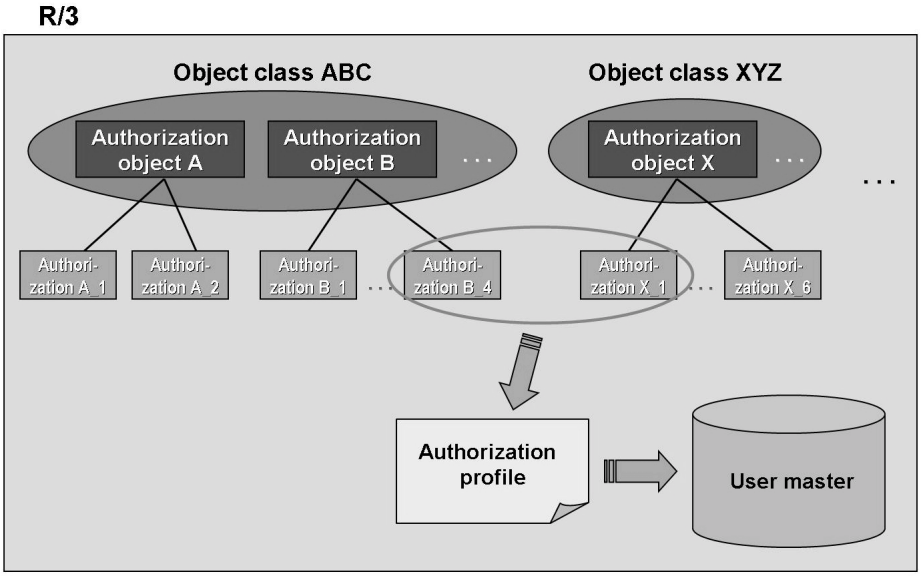


Figure 106: The R/3 Authorization Concept

So that data in the SAP System is protected against unauthorized access and so that each user can only access the data for which he or she has explicit authorization, the application developer must implement the R/3 authorization object in his or her application program.

For this purpose, you must first develop a correspondingly logical, application-related authorization model (which authorization should be checked during which user action?). Then, in the ABAP Workbench, you must create the authoriztation object, together with authorization that suits the model. This object is then assigned to the user by means of an authorization profile. In the application program itself, the application developer must check- from within the user master of the caller - whether the corresponding authorization exists in order to control further processing of the program in accordance with the check results. He or should do this before the action required by the user is performed.

SAP has inlcuded the authorization concept in the implementation of its software. You, the customer, must also do this when you enhance the SAP software or whenever you implement new applications in order to ensure data access authorization. The R/3 System contains tools that help you to manage authorizations and assign them to user master records.

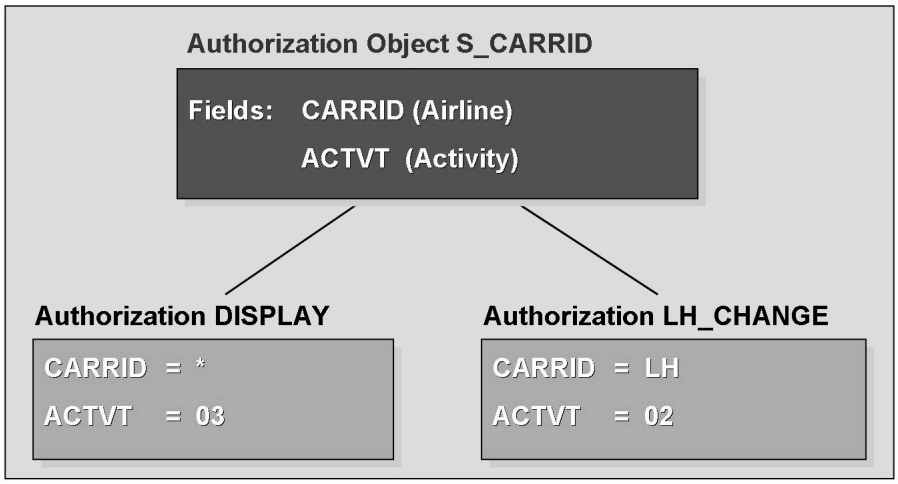


Figure 107: Authorization Object / Authorization (Example)

An authorization object has (maximum 10) fields, which - however - are not valuated.

An authorization is an authorization object with valuated fields. You can create several different authorizations for an authorization object.

Existing authorizations can be grouped into one authorization profile. An authorization profile should contain all authorizations that are required for executing certain tasks, that is, all the authorizations that are checked in the current user master as to whether they exist when the respective programs/transactions are called. Authorizaztion profiles can be assigned to a user. This contains the authorizations that are contained in the profile.

You can create authorization objects with required fields and authorizations. The maintenance transaction for authorization fields, authorization objects, and authorizations can be found in the SAP Menu under the following path:

Tools → ABAP Workbench → Development → Other Tools → Authorization Objects.

For further information, see the ABAP Editor keyword documentation for the term *Authorization concept*.

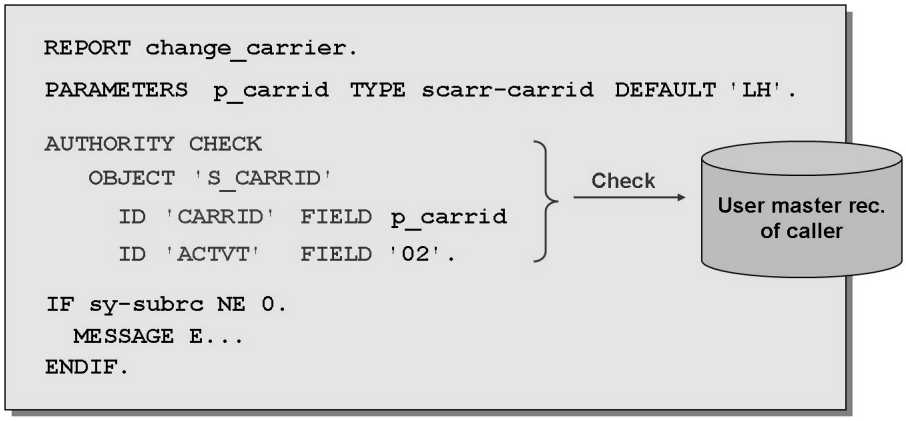


Figure 108: Performing Authorization Checks

In the application program, you must perform a check before execution of the required task by the current caller. You must check whether the authorization you have defined as necessary exists so that you can control further processing of the program in accordance with the check result. You use the AUTHORITY-CHECK statement to implement this. You specify an authorization object together with the field valuation. This means an authorization that is checked for existence by the statement in the user master of the caller.

All the authorization fields, together with the evaluations, must be specified in AUTHORITY-CHECK. If you wish to check whether a particular authorization exists, and the content of the authorization fields does not matter here, write "DUMMY" instead of "FIELD <value>" after this authorization field. The check is performed without consideration of the corresponding field. Here, for example, it is a good idea to perform plausibility checks.

The AUTHORITY-CHECK supplies only one return code in the system field sy-subrc after the check.

- 0 if the caller has the specified authorization;
- 4 if not.

You must use this return code to decide how you want to proceed in the program (read/change data, issue an error message or a different action).

For a full list of all return codes, see the keyword documentation in the ABAP Editor for AUTHORITY-CHECK.

In the ABAP Editor, use the template for the AUTHORITY-CHECK statement in order to include it in your program. In this way, you avoid typing errors that could cause syntax errors or even give the wrong check results.

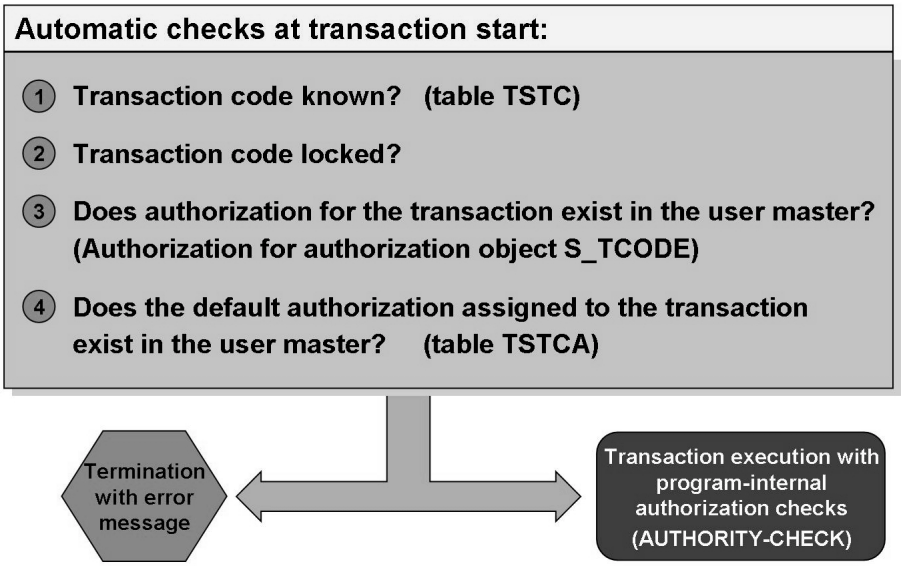


Figure 109: Authorization Checks for Transactions

At transaction start, the system automatically checks whether the specified transaction code is known, that is, whether it is marked in the table TSTC and whether this is locked.

The system then checks whether the caller has the authorization to call the transaction, that is, if he or she has a corresponding authorization for the authorization object S_TCODE.

Then the system runs through the default authorization checks assigned to the transaction. This means the system checks whether the caller has the default authorization assigned to the transaction. The assignment “Transaction / Default Authotization”is specified in the definition of the transaction and stored in the table TSTCA.

Only when all these checks have run successfully will the transaction be allowed to start by the system. Otherwise, processing is terminated with an error message.

The authorization checks contained in the application program (`AUTHORITY-CHECK`) are executed only at transaction runtime.

Lesson Summary

You should now be able to:

- Find information on authorization objects
- Create authorization objects
- Find information on authorizations and profiles
- Perform authorization checks in your program
- Link the execution of transaction codes to authorization objects

Lesson: SAP Buffers

Lesson Overview

This lesson explains the SAP table buffer.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the SAP table buffer.

Business Example

You want to learn about the details and architecture of the SAP table buffer and use them to access tables efficiently.

SAP Buffers

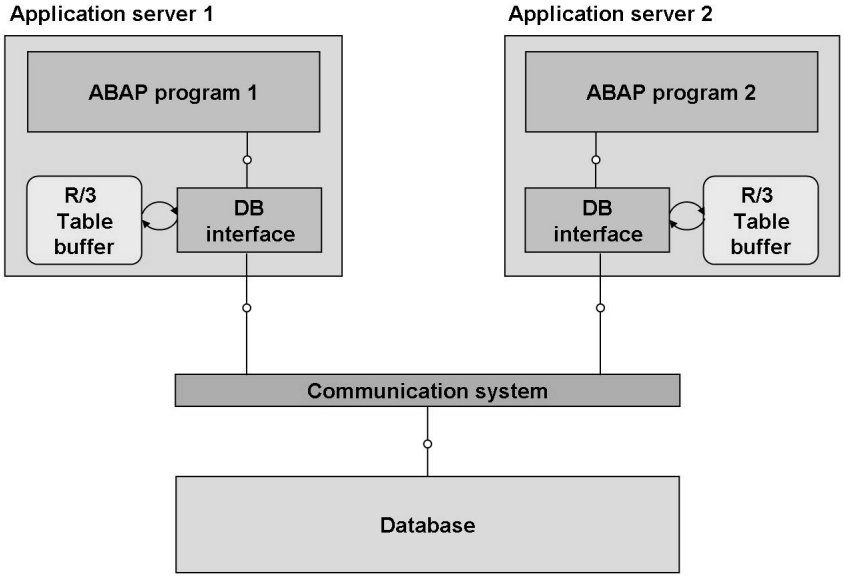


Figure 110: SAP buffers

SAP database tables can be buffered at the application server level. The aims of buffering are to:

- Reduce the time needed to access data with read accesses. Data on the application server can be accessed more quickly than data on the database.
- Reduce the load on the database. Reading the data from application server buffers reduces the number of database accesses.

The buffered tables are accessed exclusively via database interface mechanisms.

When the system executes a Native SQL command, it bypasses the R/3 database interface. Consequently, it does not use the table buffer in the R/3 System for read or change accesses. You should, therefore, not use Native SQL statements for tables that are buffered, since inconsistencies could occur between the data in the database and that in the buffer.

Not every Open SQL read access to buffered tables reads buffered data. For information on which operations for read accesses are performed directly on the database, refer to the keyword documentation for SELECT.

Updating SAP Buffers

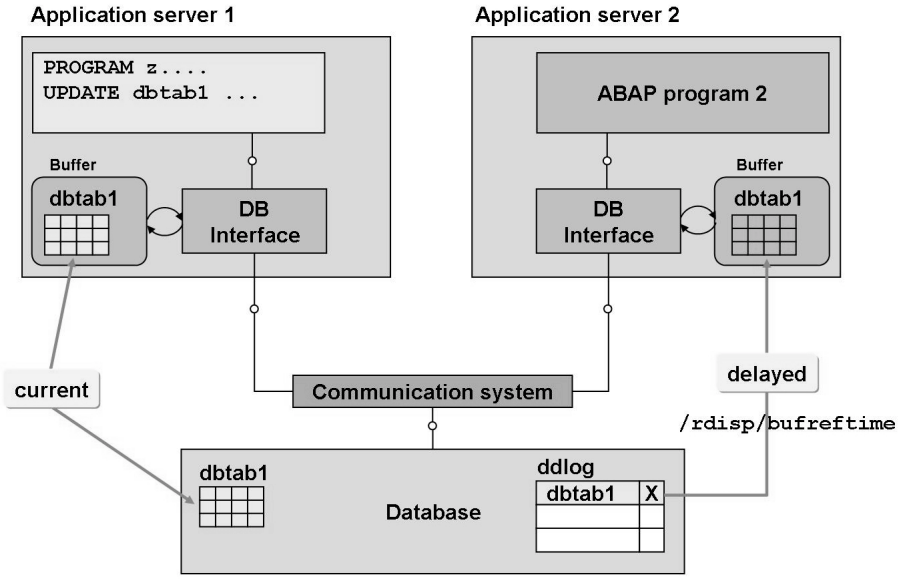


Figure 111: Updating SAP Buffers

Open SQL commands that change data update the data on the database as well as the buffers of the application server on which the program updating the database is running.

If a table is buffered on several application servers, synchronization of the other buffers is delayed and triggered as follows:

- When data on the database is changed by one of the application servers (more precisely, a program that executes an appropriate OPEN SQL command on one of the application servers) the changed data is registered as such in table DDLOG on the database. The application servers read this table at periodic intervals. If an application server finds relevant entries, the buffer contents are marked in the buffer as being no longer up-to-date.
- The next read access to the data in the buffered table is performed by the database interface on the database. The buffer is updated at the same time.

The time between the read accesses to the DDLOG table (invalidation period of the buffers) can be set using the profile parameter //rdisp/bufreftime

Buffering Types

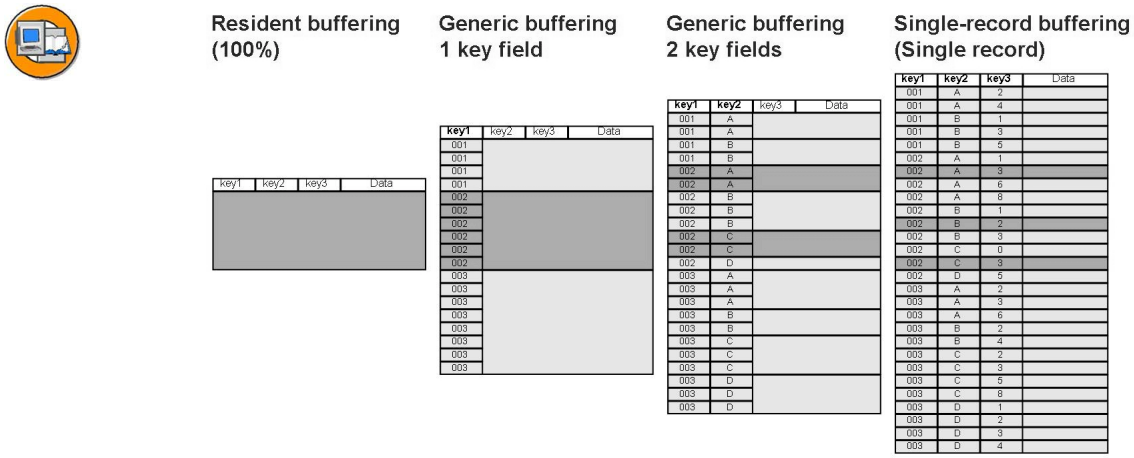


Figure 112: Buffering Types

There are three different types of buffering:

- Resident buffering (100%): The whole table is loaded to the table buffer when the table is accessed for the first time.
- Generic buffering: A generic key (first n key fields) must be specified in the technical settings for the table in the ABAP Dictionary. This is used to divide the contents of the table into generic areas. If data is accessed using one of the generic keys, the entire generic area is loaded to the table buffer. Client-dependent tables are often buffered generically for each client.
- Single-record buffering: Only single records are read by the database and stored in the table buffer.

Lesson Summary

You should now be able to:

- Explain the SAP table buffer.

Internal Use SAP Partner Only

Lesson: Native SQL

Lesson Overview

This lesson explains Native SQL.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain Native SQL.

Business Example

You want to use Native SQL instead of Open SQL in special situations and take advantage of the special SQL functions that are not available in Open SQL.

Native SQL

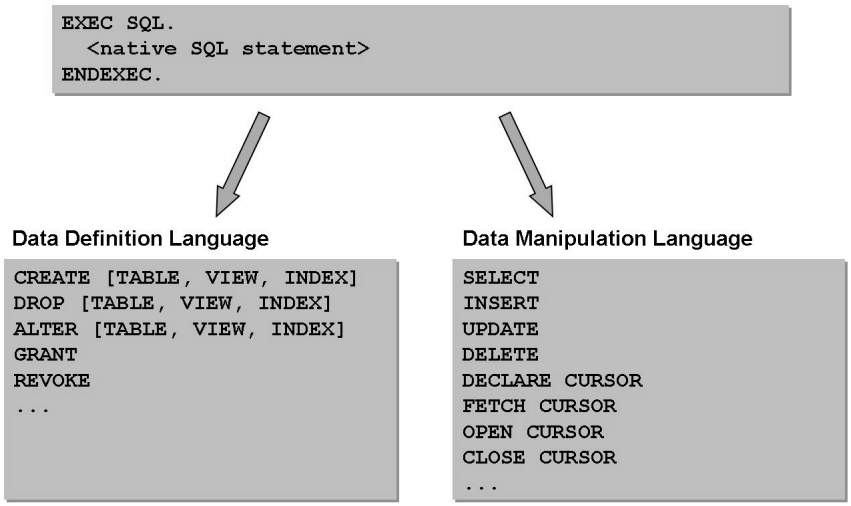


Figure 113: Native SQL

Native SQL allows you to perform operations on databases that exceed the standard Open SQL command set. Unlike Open SQL, Native SQL supports both operations on the local database - active in the SAP system - and on decentralized databases.

Native SQL contains all static statements of the data definition language (DDL) and the data manipulation language (DML) of the relational database system being addressed. Statements for error handling and declaring host variables are not allowed.

The EXEC SQL and ENDEXEC statements must encapsulate a Native SQL command in an ABAP program. The database table does not have to be declared in the ABAP Dictionary to perform the Native SQL command.

Since the SQL database language is only partly standardized, you must always use the correct syntax for the Native SQL command from the documentation of the corresponding database manufacturer.

For further details on Native SQL, see the keyword documentation in the ABAP Editor under the search term SQL.

Lesson Summary

You should now be able to:

- Explain Native SQL.

Lesson: Cluster Tables

Lesson Overview

This lesson explains cluster tables.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain cluster tables.

Business Example

You want to use cluster tables to store data for your applications.

Data Cluster

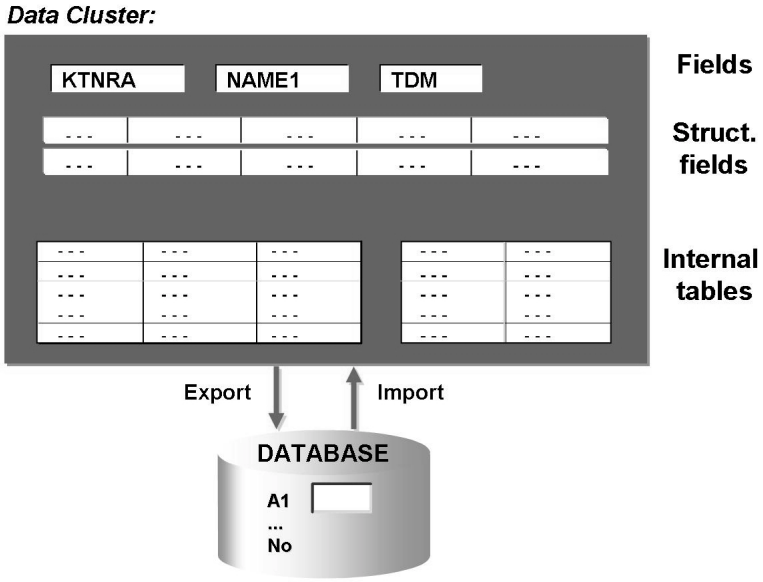


Figure 114: Data Cluster

A data cluster is a combination of data objects. Data objects are fields, structured fields, internal tables, and complex structures derived from these. You process data clusters using the ABAP commands EXPORT, IMPORT, and DELETE. Data clusters can be stored in cluster databases.

You subdivide a cluster database in the ABAP Dictionary into application areas according to your own criteria and using logically related data clusters. The application area name comprises two characters and is freely selectable. You identify the cluster within the application area using an ID (cluster ID).

Syntax: EXPORT to Cluster Database

```
EXPORT <name> FROM <obj>
       [<obj>]
                      TO DATABASE <dbtab>(<ar>) ID <id>.
```

```
REPORT xxx
TABLES indx.
EXPORT                  | optional
       <field 1>        | FROM <field a>
       <field 2>        | FROM <field b>
       .                | .
       .                | .
       <structure 1>    | FROM <structure a>
       .                | .
       .                | .
       <itab 1>         | FROM <itab a>
       .                | .
       .                | .
TO DATABASE indx(<ar>)
ID <id>.
```

Figure 115: Syntax: EXPORT to Cluster Database

For exporting, you require a cluster database. The table INDX is a cluster database for general purposes. Cluster databases should be created as transparent tables in the ABAP Dictionary and must have a standardized structure. For more information, see the online documentation for the EXPORT command.

For an EXPORT, specify the data objects of your cluster in a list.

To perform the export, specify the cluster database and the application area within the cluster database. You identify the cluster itself using the cluster ID. If you want to define a data object name in the cluster that is different from the one in the program, use the optional addition FROM. The data objects can be listed in any order. With an export, there is no write protection facility. Existing clusters, therefore, are overwritten if an EXPORT is performed again.

The data is stored in compressed form in the cluster database.

In your program, you declare an application area for your cluster database at the start using the TABLES statement.

Caution: Header lines in internal tables cannot be exported. Usually, only the table contents are exported.

If you are working within language constructions from the object-oriented ABAP extension (ABAP Objects), you must use the name substitutions marked as optional on this slide.

Syntax: IMPORT and DELETE

```
IMPORT <name> TO <obj>
       [<obj>]
                        FROM DATABASE <dbtab>(<ar>) ID <id>.
```

```
REPORT xxx
TABLES indx.
IMPORT                  | optional
       <field 1>        | TO <field a>
       <field 2>        | TO <field b>
       .                | .
       .                | .
       <structure 1>    | TO <structure a>
       .                | .
       .                | .
       <itab 1>         | TO <itab a>
       .                | .
FROM DATABASE indx(<ar>)
ID <id>.
```

```
DELETE FROM DATABASE INDX(<ar>)
ID <id>.
```

Figure 116: Syntax: IMPORT and DELETE

For an IMPORT, you only need to list a subset of the data objects of your cluster in any order. If you want to define a data object name in the program that is different from the one in the cluster, use the optional addition TO.

After the IMPORT, the system outputs a return code (sy-subrc). This return code refers to the cluster, rather than to an individual object in the cluster. If the cluster does not exist, the return code is not equal to zero.

The structure of the fields, structures, and internal tables to be imported must correspond to the structure of the objects exported to the dataset. If this is not the case, a runtime error occurs. In addition, the objects must be imported using the same name with which they were exported - otherwise, they are not imported. If the cluster exists, the return code is 0, irrespective of whether objects have been imported.

Caution: here too, only the actual table contents of internal tables are imported and therefore the header lines remain unchanged.

DELETE always deletes the entire cluster. You cannot delete an individual data object within the cluster. After DELETE, the system outputs a return code.

If you are working within language constructions from the object-oriented ABAP extension (ABAP Objects), you must use the name substitutions marked as optional on this slide.

ABAP Cluster Databases

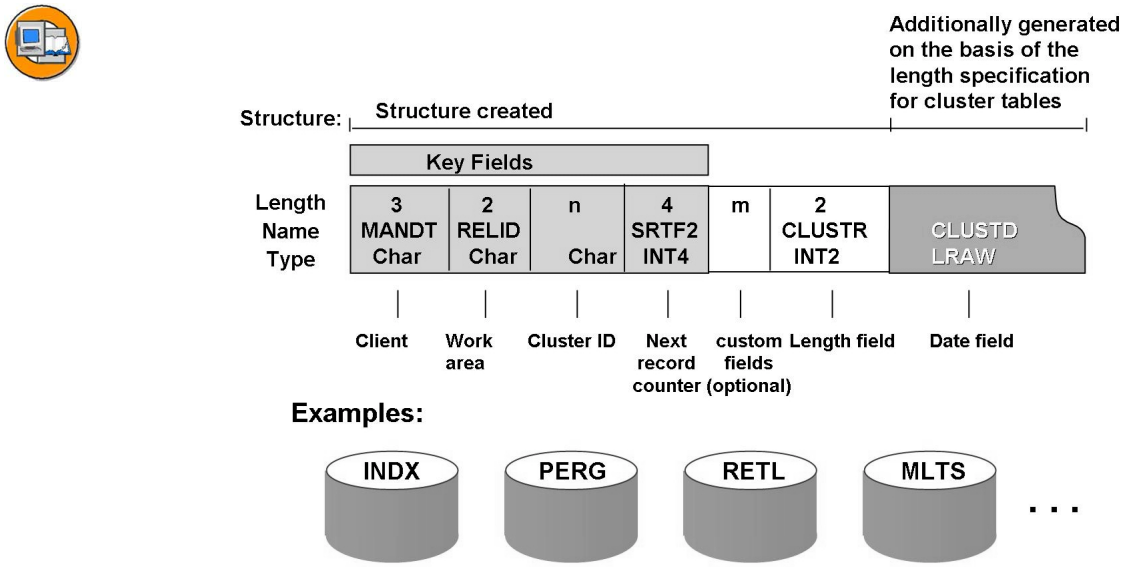


Figure 117: ABAP Cluster Databases

You can create your own ABAP cluster databases To do this, proceed as follows:

- Define a database table as a transparent table in the ABAP Dictionary. This table represents its cluster database.
- Build the table structure as shown above.
- The MANDT field can be omitted (it will be filled automatically if it exists).
- The fields RELID, SRTF2, CLUSTR, CLUSTD and the cluster ID are filled automatically within an EXPORT action.
- Any user-defined fields must be filled before the EXPORT. They can then be evaluated after an IMPORT.
- You can choose the field names for the cluster ID and your own fields. The remaining field names are specified by the system.
- The length of the part used for the data cluster is calculated from the total length of the structure minus the length of the first six fields.

Key of Database Table INDX

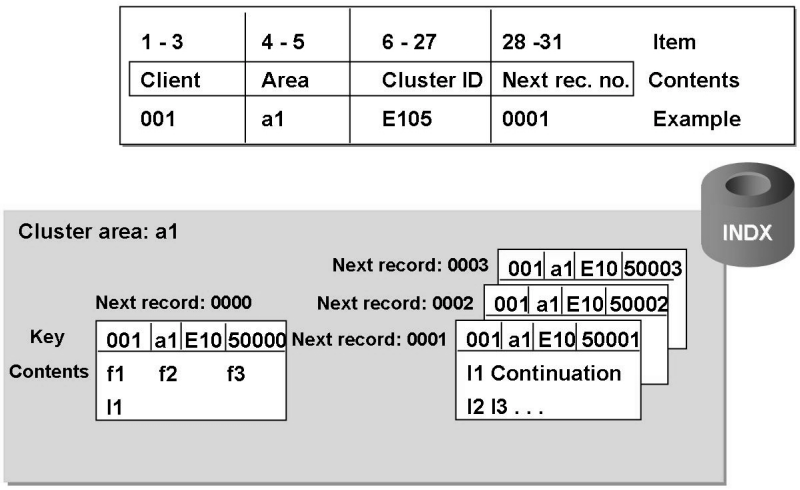


Figure 118: Key of Database Table INDX

The INDX database is an example of a database table in which you can store data clusters. It is installed in your system by default and has a key length of 31 bytes.

You can display the table structure using the keyword help for the table structure INDX. The key consists of a client, area, cluster ID, and subsequent record number. The cluster ID has a default length of 22 bytes, but can have any length in a cluster database.

With larger data clusters, the runtime system appends subsequent records with the same length automatically.

Example: Catalog for INDX

```
REPORT sapbc411d_clustercatalogue.
TABLES indx.
SELECT-OPTIONS:
  area FOR indx-relid,
  clstr_id FOR indx-srtfd.

START-OF-SELECTION.
  SELECT DISTINCT relid srtfd aedat usera pgmid
    INTO (indx-relid, indx-srtfd, indx-aedat,
          indx-usera, indx-pgmid)  FROM indx
    WHERE relid IN area
     AND  srtfd IN clstr_id.
*    AND  srtf2 = 0.

   WRITE: / indx-relid,
     indx-srtfd,
     indx-aedat DD/MM/YYYY,
     indx-usera,
     indx-pgmid.

  ENDSELECT.
```

Figure 119: Example: Catalog for INDX

Apart from the key fields and the data cluster, the structure of the database INDX also includes optional fields for administrative information (for example, change, validity date, created by, see structure INDX). You use the SELECT statement to access the key and administration fields - for example, to create a catalog. The system only fills the administration data fields if you fill them before the EXPORT using the MOVE statement (for example, MOVE SY-DATUM TO INDX-AEDAT).

Cluster Tables and Transparent Tables

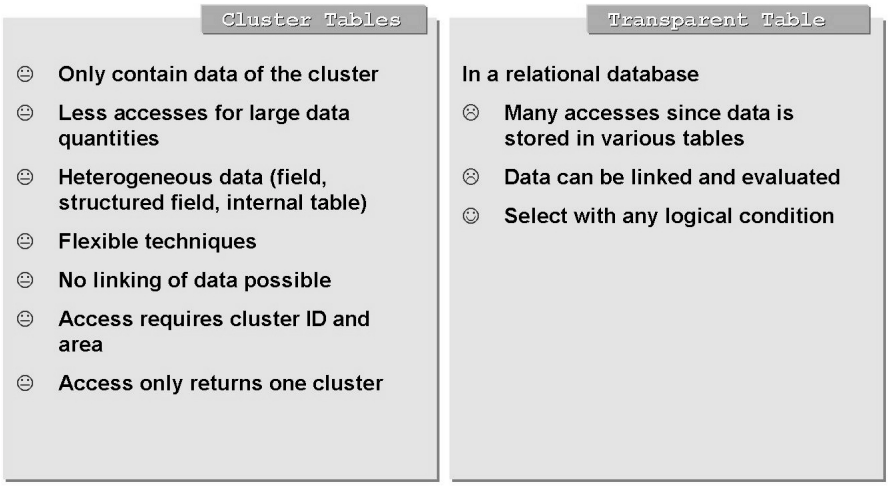


Figure 120: Cluster Tables and Transparent Tables

Cluster databases contain the cluster data in the desired form. Large quantities of data can be read via the internal cluster table administration using a limited number of accesses (I/Os).

Standardization requires that data from several tables be selected in transparent tables, which entails several I/Os. This can, however, be achieved effectively using joins to ABAP Open SQL introduced in release 3.1. Refer also to the unit on ABAP Open SQL. The advantage of this is that individual objects can be read from various tables and linked to each other. In cluster tables, however, only the data from one cluster can be read. Links to data in other clusters are not possible.

Access to cluster data requires knowledge of the cluster ID and the application area. The access also returns, at most, the data of one cluster ID, in other words, it will not return several clusters, as is the case with a SELECT loop. In contrast to this, data can be determined in transparent tables on the basis of any logical expression.

Lesson Summary

You should now be able to:

- Explain cluster tables.

Lesson: SAP Locks

Lesson Overview

This lesson explains SAP locks.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain SAP locks.

Business Example

You want to learn about SAP locks.

Update and Lock Durations for _scope = 1

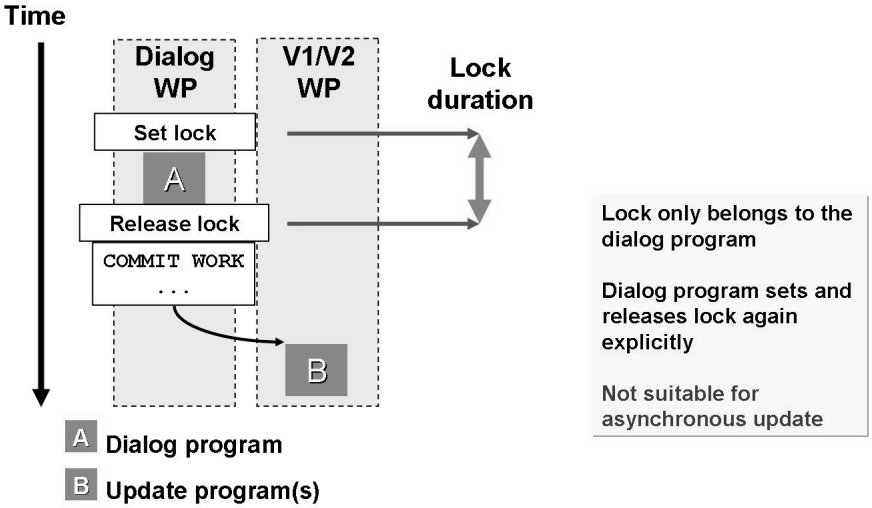


Figure 121: Update and Lock Durations for _scope = 1

For _SCOPE = 1, the dialog program contains the locks that it generates. The locks remain set until they are released, either using the function module DEQUEUE_<object>, or implicitly at the end of the program. This includes the ABAP statements LEAVE PROGRAM, LEAVE TO TRANSACTION <ta> and SUBMIT <program>, and termination messages (message type “A” or “X”).

If the transaction uses asynchronous update, the update program has no guarantee that the data to be changed is not already locked by another user. For this reason, you should not use _SCOPE = 1 for asynchronous updates.

Update and Lock Durations for _scope = 3

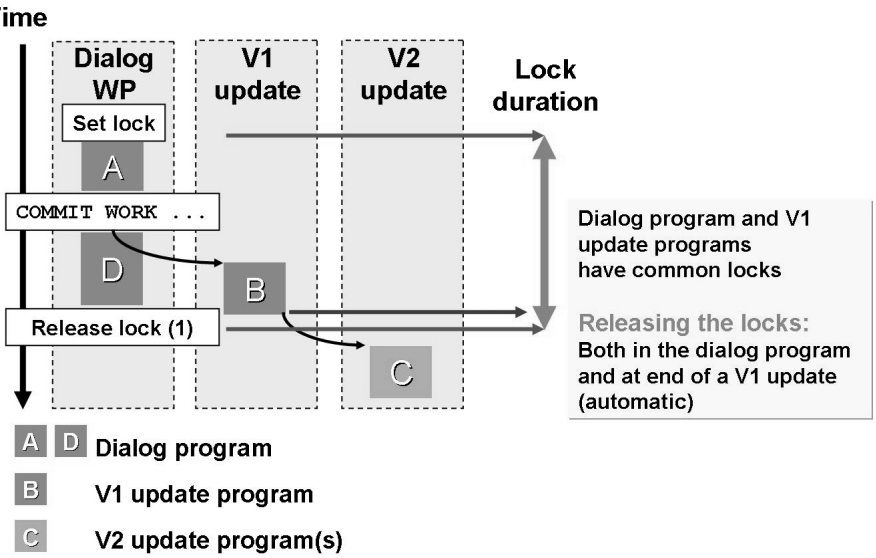


Figure 122: Update and Lock Durations for _scope = 3

If you are using asynchronous update and want to be sure that the locks generated in the dialog program remain set for longer than the V1 update function modules are active, you can use the addition _SCOPE = 3. In this case, the lock is shared between the dialog program and the update program.

Lock entries that you generate with _SCOPE = 3 must be released **both** by the dialog program **and** by the update program.

Lock entries with _SCOPE = 3 are only used in a few special cases.

Lesson Summary

You should now be able to:

- Explain SAP locks.

Lesson: BAPI Transaction Model

Lesson Overview

This lesson explains BAPI architecture and the use of BAPIs.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain BAPI architecture and the use of BAPIs.

Business Example

You want to process data by calling BAPIs.

BAPI – Definition

- **Business Application Programming Interface**

- **A BAPI is a well defined interface for the processes and data of a business application system, implemented as the methods of an object in the Business Object Repository (BOR).**

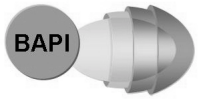


Figure 123: BAPI – Definition

A BAPI is a “point of entry” to the R/3 System - that is, a point at which the R/3 System provides access to business data and processes.

Each object in the BOR can have several methods, one or more of which can be implemented as BAPIs.

BAPIs can perform various functions such as:

- Creating an object
- Retrieving the attributes of an object
- Changing the attributes of an object
- ...

Where Are BAPIs Used?

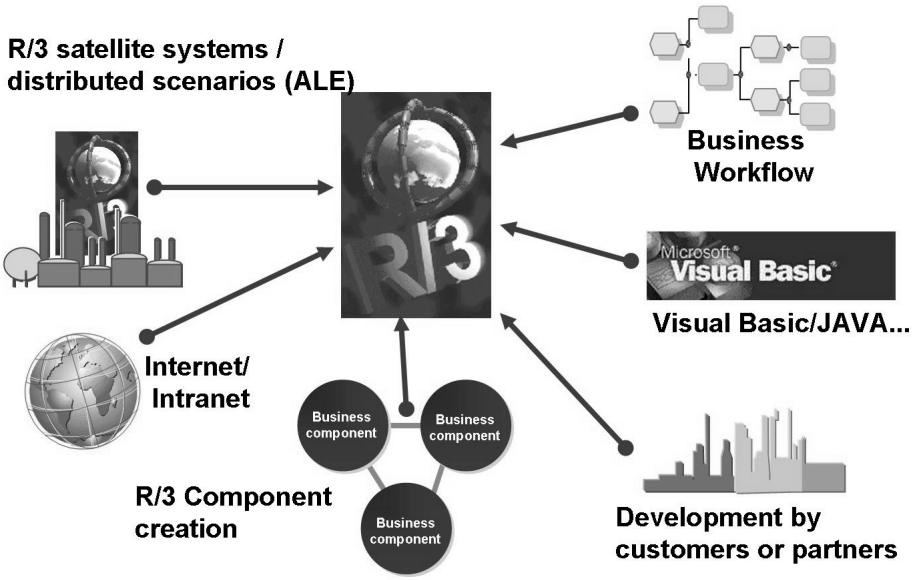


Figure 124: Where Are BAPIs Used?

A BAPI is an interface that can be used for various applications, for example:

- Internet Application Components - Mapping individual R/3 functions on the Internet/Intranet for users who have no R/3 experience
- R/3 component formation - Communication between the business objects of different R/3 components (applications)
- VisualBasic/JAVA/C++ - External clients (for example, alternative GUIs) access business data and processes directly

BAPI – Properties

- **Object-oriented**
  - Access to methods from BOR objects
- **Stable interface**
  - A BAPI interface is "frozen"
    - Release function module for customer
    - Link data elements to the interface parameters of the FM
- **Can be used internally and externally**
  - BAPIs can be used within R/3 and externally
- **But: BAPIs do not contain a presentation layer**
  - Results are visualized externally by the individual caller

Figure 125: BAPI – Properties

For BAPIs with UPDATE semantics:

- BAPIs are always called up synchronously. Exception: Sent asynchronously via ALE in an IDoc, then called up again synchronously in the target system (**new as of release 4.0**).
- BAPIs implement your database changes through the update task.
- New as of release 4.6: BAPIs themselves do not program a COMMIT WORK. Instead of this, methods
  - TransactionCommit (FBS Name: BAPI_TRANSACTION_COMMIT) and
  - TransactionRollback (FBS Name: BAPI_TRANSACTION_ROLL-BACK) of the BAPISERVICE service object (technical name: SAP0001) must be used.

BAPI Transaction Model for Release 3.1 (Example External Client)

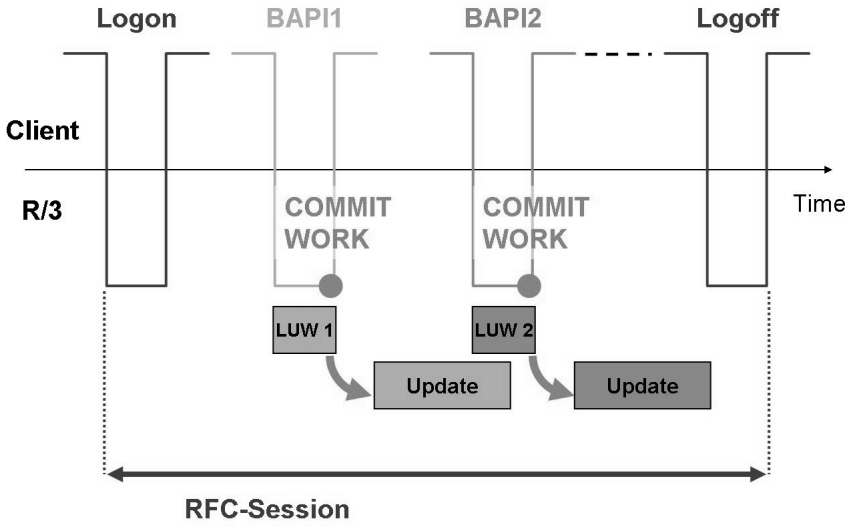


Figure 126: BAPI Transaction Model for Release 3.1 (Example External Client)

In release 3.1, the BAPIs themselves carry out the COMMIT WORK command; in other words, a BAPI is synonymous with an LUW or transaction.

If a BAPI itself carries out a COMMIT WORK command, it must be listed explicitly in the documentation for the BAPI (as of Release 4.0). This is the only way in which the user can find out that a COMMIT WORK takes place in the BAPI.

These BAPIs must also be documented in the Online Service System (OSS) in note no. 0131838 “Collective note for BAPIs with “Commit Work” commands”.

BAPI Transaction Model for Release 4.0 (Example External Client)

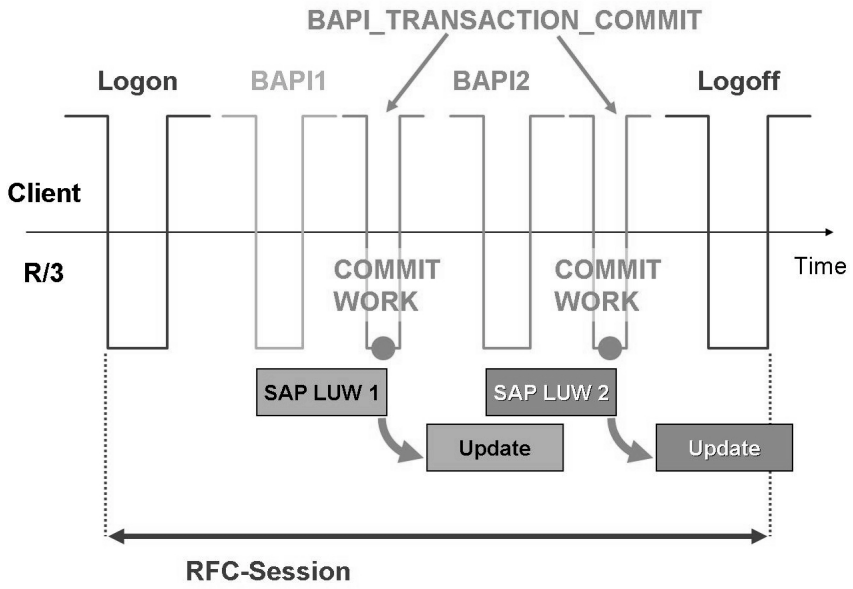


Figure 127: BAPI Transaction Model for Release 4.0 (Example External Client)

In Release 4.0A, the Commit control must be removed from the writing BAPIs, in other words, from the BAPIs that cause the database changes. Here, the existing transaction model used in Release 3.1 should not be changed. For this purpose, the RFC-capable function module BAPI_TRANSACTION_COMMIT, which performs the COMMIT WORK command, must be called.

BAPI Transaction Model as of Release 4.6 (Example External Client)

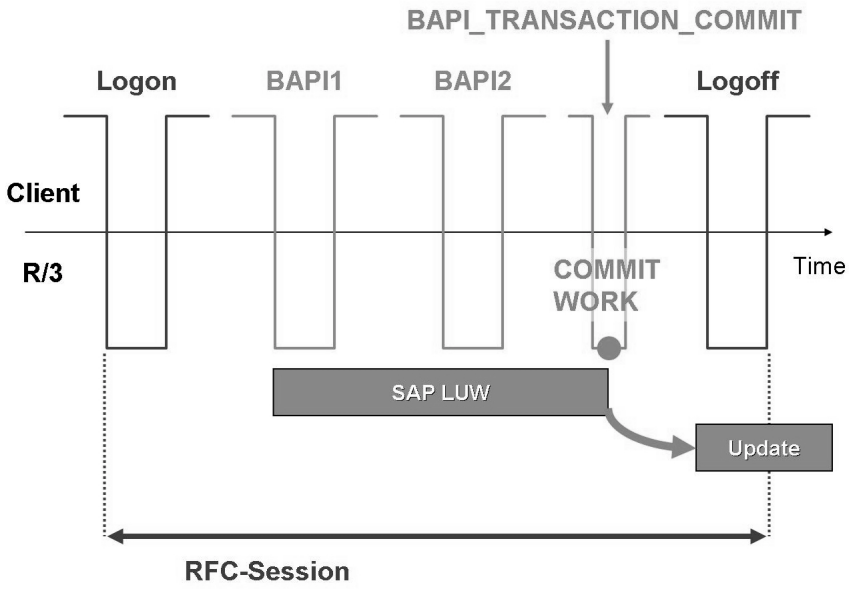


Figure 128: BAPI Transaction Model as of Release 4.6 (Example External Client)

In the transaction model used to develop BAPIs, a transaction represents a processing unit or Logical Unit of Work (LUW).

Operations that change the database **must** be carried out through the update task.

As of Release 4.0, BAPIs no longer carry out COMMIT WORK commands. This enables several BAPIs to be combined with each other in an LUW.

To close LUW processing initiated by the BAPI calls, use the function module BAPI_TRANSACTION_COMMIT. This module implements the TransactionCommit method of the BAPISERVICE business object.

For further details on using BAPIs, see the BAPI User Manual and the BAPI Programming Guide. These are available in the online documentation as well as on the SAP Homepage (http://www.sap-ag.de) under Technology → Open BAPI Network/ BAPI Section.

Lesson Summary

You should now be able to:

- Explain BAPI architecture and the use of BAPIs.

Internal Use SAP Partner Only

Lesson: COMMIT WORK / ROLLBACK WORK (Details and Summary) + Complete Answers for the Exercises

Lesson Overview

This lesson explains the ABAP statements COMMIT WORK and ROLLBACK WORK.

Lesson Objectives

After completing this lesson, you will be able to:

- Use the ABAP statements COMMIT WORK and ROLLBACK WORK appropriately.

Business Example

You want to use the ABAP statements COMMIT WORK and ROLLBACK WORK appropriately.

Actions for COMMIT WORK

- Actions on the database
  - All updates that were made during the current dialog step are committed
  - All database locks are released
  - All open database cursors are closed (even those that were opened using WITH HOLD)
- All subroutines registered using PERFORM ON COMMIT are executed
- Update is triggered (CALL FUNCTION IN UPDATE TASK)
  - COMMIT WORK – does not wait until the end of the update
  - COMMIT WORK AND WAIT – waits until the end of the update
  - For a local update: immediate execution of the update modules
- Locks (_SCOPE = 2) are:
  - NOT released if no update requests have come up
  - Transferred to the update and released when it has ended (as a rule)

Actions for ROLLBACK WORK

- Current DB LUW (DB Rollback) is ended
  - All changes made in the current dialog step are undone
  - All database locks are released
  - All open cursors are closed
- Data in the corresponding SAP LUW is deleted:
  - Registration of update modules registered with CALL FUNCTION IN UPDATE TASK is rejected
  - The same applies to the subroutines registered with PERFORM ON COMMIT
  - Function modules that were registered for transactional or queued RFC (CALL FUNCTION IN BACKGROUND TASK) are not executed
- SAP locks are released (_scope = 2)

Actions That a ROLLBACK WORK Does Not Perform

- Undo updates that were committed previously
- Undo updates to internal tables and other data objects (program context)
- Reset values in calculated “contexts”
- Undo changes made to operating system files
- Undo actions that were executed during synchronous RFCs

SAP LUW: Overall View

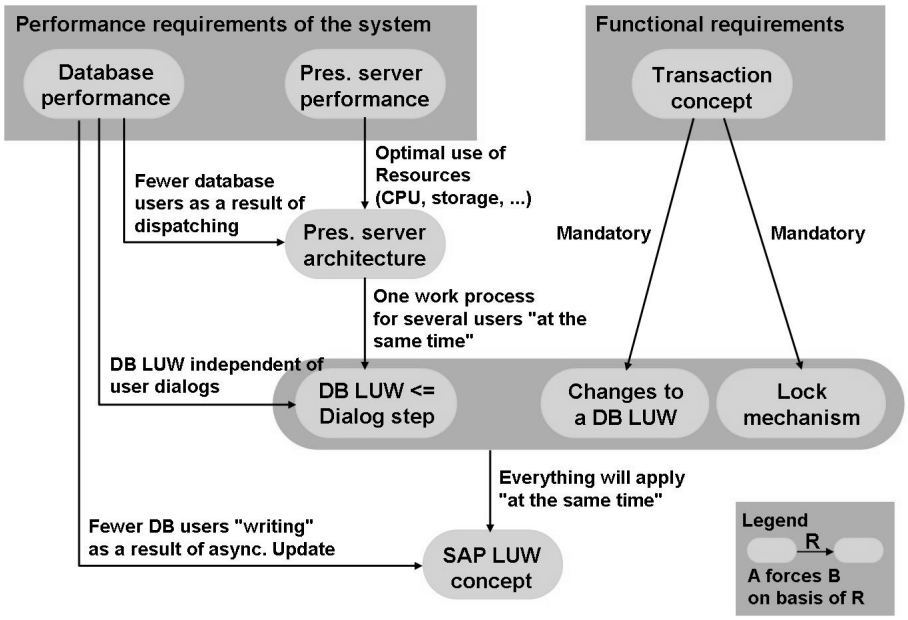


Figure 129: SAP LUW: Overall View

Exercise 9: Complete Solution for “Generating Customer Data Records” (Complete Transaction)

Exercise Objectives

After completing this exercise, you will be able to:

- -

Business Example

-

Task: Complete Solution for “Generating Customer Data Records”

Complete Transaction

1. -

Solution 9: Complete Solution for “Generating Customer Data Records” (Complete Transaction)

Task: Complete Solution for “Generating Customer Data Records”

Complete Transaction

1. -

a) **Model Solution SAPBC414S_CREATE_CUSTOMER**

Module Pool

```
*&---------------------------------------------------------------------*
*& Modulpool         SAPBC414S_CREATE_CUSTOMER                       *
*&---------------------------------------------------------------------*
INCLUDE BC414S_CREATE_CUSTOMERTOP.
INCLUDE BC414S_CREATE_CUSTOMERO01.
INCLUDE BC414S_CREATE_CUSTOMERI01.
INCLUDE BC414S_CREATE_CUSTOMERF01.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE status_0100.

PROCESS AFTER INPUT.
  MODULE exit AT EXIT-COMMAND.
  MODULE save_ok_code.
  FIELD: scustom-name MODULE mark_changed ON REQUEST.
  MODULE user_command_0100.
```

TOP Include

```
*&---------------------------------------------------------------------*
*& Include BC414S_CREATE_CUSTOMERTOP                                   *
*&---------------------------------------------------------------------*
PROGRAM  sapbc414s_create_customer MESSAGE-ID bc414.
```

Continued on next page

Internal Use SAP Partner Only

```
DATA: answer, flag.
DATA: ok_code LIKE sy-ucomm, save_ok LIKE ok_code.
TABLES: scustom.
```

PBO Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERO01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
MODULE STATUS_0100 OUTPUT.
  SET PF-STATUS 'DYN_0100'.
  SET TITLEBAR 'DYN_0100'.
ENDMODULE.                 " STATUS_0100  OUTPUT
```

PAI Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERI01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&---------------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'EXIT'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes on screen 100
        LEAVE PROGRAM.
      ELSE.
        PERFORM ask_save USING answer.
        CASE answer.
          WHEN 'J'.
            ok_code = 'SAVE&EXIT'.
          WHEN 'N'.
            LEAVE PROGRAM.
```

Continued on next page

Internal Use SAP Partner Only


```
          WHEN 'A'.
            CLEAR ok_code.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
    WHEN 'CANCEL'.
      IF sy-datar IS INITIAL AND flag IS INITIAL.
* no changes on screen 100
        LEAVE TO SCREEN 0.
      ELSE.
        PERFORM ask_loss USING answer.
        CASE answer.
          WHEN 'J'.
            LEAVE TO SCREEN 0.
          WHEN 'N'.
            CLEAR ok_code.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT

*&---------------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&---------------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
ENDMODULE.                             " SAVE_OK_CODE  INPUT


*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
    WHEN 'SAVE&EXIT'.
      PERFORM save.
      LEAVE PROGRAM.
    WHEN 'SAVE'.
      IF flag IS INITIAL.
        SET SCREEN 100.
      ELSE.
        PERFORM save.
        SET SCREEN 0.
```

Continued on next page

```
      ENDIF.
    WHEN 'BACK'.
      IF flag IS INITIAL.
        SET SCREEN 0.
      ELSE.
        PERFORM ask_save USING answer.
        CASE answer.
          WHEN 'J'.
            PERFORM save.
            SET SCREEN 0.
          WHEN 'N'.
            SET SCREEN 0.
          WHEN 'A'.
            SET SCREEN 100.
        ENDCASE.
      ENDIF.
  ENDCASE.
ENDMODULE.                              " USER_COMMAND_0100  INPUT


*&---------------------------------------------------------------------*
*&      Module  MARK_CHANGED  INPUT
*&---------------------------------------------------------------------*
MODULE mark_changed INPUT.
* set flag to mark changes were made on screen 100
  flag = 'X'.
ENDMODULE.                              " MARK_CHANGED  INPUT
```

FORM Routines

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_CREATE_CUSTOMERF01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  NUMBER_GET_NEXT
*&---------------------------------------------------------------------*
*      -->P_WA_SCUSTOM  text
*---------------------------------------------------------------------*
FORM number_get_next USING p_scustom LIKE scustom.
  DATA: return TYPE inri-returncode.
* get next free number in the number range '01'
* of number range object 'SBUSPID'
```

Continued on next page


```
  CALL FUNCTION 'NUMBER_GET_NEXT'
       EXPORTING
            nr_range_nr = '01'
            object      = 'SBUSPID'
       IMPORTING
            number      = p_scustom-id
            returncode  = return
       EXCEPTIONS
            OTHERS      = 1.
  CASE sy-subrc.
    WHEN 0.
      CASE return.
        WHEN 1.
* number of remaining numbers critical
          MESSAGE s070.
        WHEN 2.
* last number
          MESSAGE s071.
        WHEN 3.
* no free number left over
          MESSAGE a072.
      ENDCASE.
    WHEN 1.
* internal error
      MESSAGE a073 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " NUMBER_GET_NEXT


*&---------------------------------------------------------------------*
*&      Form  ASK_SAVE
*&---------------------------------------------------------------------*
*      -->P_ANSWER  text
*---------------------------------------------------------------------*
FORM ask_save USING p_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
       EXPORTING
            textline1 = 'Data has been changed.'(001)
            textline2 = 'Save before leaving transaction?'(002)
            titel     = 'Create Customer'(003)
       IMPORTING
            answer    = p_answer.
ENDFORM.                               " ASK_SAVE
```

Continued on next page


```
*&---------------------------------------------------------------------*
*&      Form  ASK_LOSS
*&---------------------------------------------------------------------*
*       -->P_ANSWER  text
*---------------------------------------------------------------------*
FORM ask_loss USING p_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_LOSS_OF_DATA'
       EXPORTING
            textline1 = 'Continue?'(004)
            titel     = 'Create Customer'(003)
       IMPORTING
            answer    = p_answer.
ENDFORM.                               " ASK_LOSS


*&---------------------------------------------------------------------*
*&      Form  ENQ_SCUSTOM
*&---------------------------------------------------------------------*
FORM enq_scustom.
  CALL FUNCTION 'ENQUEUE_ESCUSTOM'
       EXPORTING
            id             = scustom-id
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 1.
* dataset allready locked
      MESSAGE e060.
    WHEN 2 OR 3.
* locking of dataset not possible for other reasons
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " ENQ_SCUSTOM


*&---------------------------------------------------------------------*
*&      Form  DEQ_ALL
*&---------------------------------------------------------------------*
FORM deq_all.
  CALL FUNCTION 'DEQUEUE_ALL'.
ENDFORM.                               " DEQ_ALL
```

Continued on next page


```
*&---------------------------------------------------------------------*
*&      Form  SAVE_SCUSTOM
*&---------------------------------------------------------------------*
FORM save_scustom.
  INSERT INTO scustom VALUES scustom.
  IF sy-subrc <> 0.
    MESSAGE a048.
  ELSE.
    SET PARAMETER ID 'CSM' FIELD scustom-id.
    MESSAGE s015 WITH scustom-id.
  ENDIF.
ENDFORM.                               " SAVE_SCUSTOM


*&---------------------------------------------------------------------*
*&      Form  SAVE
*&---------------------------------------------------------------------*
FORM save.
* get SCUSTOM-ID from number range object SBUSPID
  PERFORM number_get_next USING scustom.
* lock dataset
  PERFORM enq_scustom.
* save new customer
  PERFORM save_scustom.
* unlock dataset
  PERFORM deq_all.
ENDFORM.                               " SAVE
```


Internal Use SAP Partner Only

Exercise 10: Complete Solution for “Creating/Canceling a Booking” (Complete Transaction)

Exercise Objectives

After completing this exercise, you will be able to:

- -

Business Example

-

Task: Complete Solution for “Creating/Canceling a Booking”

Complete Transaction

1. -

Internal Use SAP Partner Only

Solution 10: Complete Solution for “Creating/Canceling a Booking” (Complete Transaction)

Task: Complete Solution for “Creating/Canceling a Booking”

Complete Transaction

1. -

a) **Model Solution SAPBC414S_BOOKINGS**

Module Pool

```
*&---------------------------------------------------------------------*
*& Modulpool         SAPBC414S_BOOKINGS                              *
*&---------------------------------------------------------------------*
INCLUDE bc414s_bookingstop.
INCLUDE bc414s_bookingso01.
INCLUDE bc414s_bookingsi01.
INCLUDE bc414s_bookingsf01.
INCLUDE bc414s_bookingsf02.
INCLUDE bc414s_bookingsf03.
INCLUDE bc414s_bookingsf04.
INCLUDE bc414s_bookingsf05.
INCLUDE bc414s_bookingsf06.
INCLUDE fbc414_cdocscdc.
```

SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0100.
*
PROCESS AFTER INPUT.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  CHAIN.
* cancel booking: check if flight exists or flight can be created
    FIELD: SDYN_CONN-CARRID, SDYN_CONN-CONNID, SDYN_CONN-FLDATE.
    MODULE USER_COMMAND_0100.
  ENDCHAIN.
```

Continued on next page

SCREEN 200

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0200.
  MODULE TRANS_DETAILS.
  CALL SUBSCREEN SUB1 INCLUDING SY-CPROG '0201'.
  LOOP AT ITAB_BOOK INTO WA_BOOK WITH CONTROL TC_SBOOK.
    MODULE TRANS_TO_TC.
* allow only modification of bookings, that are not allready
* cancelled
    MODULE MODIFY_SCREEN.
  ENDLOOP.
*
PROCESS AFTER INPUT.
  LOOP AT ITAB_BOOK.
* mark changed bookings in internal table itab_book
    FIELD SDYN_BOOK-CANCELLED MODULE MODIFY_ITAB ON REQUEST.
  ENDLOOP.
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  MODULE USER_COMMAND_0200.
```

SCREEN 201

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

SCREEN 300

```
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0300.
  MODULE TABSTRIP_INIT.
  MODULE TRANS_DETAILS.
  CALL SUBSCREEN TAB_SUB INCLUDING SY-CPROG SCREEN_NO.
*
PROCESS AFTER INPUT.
  CALL SUBSCREEN TAB_SUB.
```

Continued on next page

Internal Use SAP Partner Only


```
  MODULE EXIT AT EXIT-COMMAND.
  MODULE SAVE_OK_CODE.
  MODULE TRANS_FROM_0300.
  MODULE USER_COMMAND_0300.
```

SCREEN 301

```
PROCESS BEFORE OUTPUT.
  MODULE HIDE_BOOKID.
PROCESS AFTER INPUT.
```

SCREEN 302

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

SCREEN 303

```
PROCESS BEFORE OUTPUT.
PROCESS AFTER INPUT.
```

TOP Include

```
*&---------------------------------------------------------------------*
*& Include BC414S_BOOKINGSTOP                                         *
*&---------------------------------------------------------------------*
PROGRAM  sapbc414s_bookings MESSAGE-ID bc414.

* change documents: data definitions for use of function modules
INCLUDE fbc414_cdocscdt.

* line type of internal table itab_book, used to display bookings in
* table control
TYPES: BEGIN OF wa_book_type.
         INCLUDE STRUCTURE sbook.
TYPES:   name TYPE scustom-name,
         mark,
```

Continued on next page


```
       END OF wa_book_type.

* work area and internal table used to display bookings in table
* control
DATA: wa_book TYPE wa_book_type,
      itab_book TYPE TABLE OF wa_book_type.

* bookings to be modified on database table
DATA: itab_sbook_modify TYPE TABLE OF sbook.

* change documents: bookings before changes are performed
DATA: itab_cd TYPE TABLE OF sbook WITH NON-UNIQUE KEY
      carrid connid fldate bookid customid.

* work areas for database tables spfli, sflight, sbook.
DATA: wa_sbook TYPE sbook,
      wa_sflight TYPE sflight,
      wa_spfli TYPE spfli.

* complex transaction: customer ID created in the called transaction
data: scust_id(20).

* transport function codes from screens
DATA: ok_code TYPE sy-ucomm, save_ok LIKE ok_code.
* define subscreen screen number on tabstrip, screen 300
DATA: screen_no TYPE sy-dynnr.
* used to handle sy-subrc, which is determined in subroutine
DATA  sysubrc LIKE sy-subrc.

* For field transport to/from screen
TABLES: sdyn_conn, sdyn_book.
* table control declaration (display bookings),
* tabstrip declaration (create booking)
CONTROLS: tc_sbook TYPE TABLEVIEW USING SCREEN '0200',
          tab TYPE TABSTRIP.
```

PBO Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGSO01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
```

Continued on next page


```
*&      Module  STATUS_0100  OUTPUT
*&---------------------------------------------------------------------*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'DYN_100'.
  SET TITLEBAR 'DYN_100'.
ENDMODULE.                             " STATUS_0100  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  STATUS_0200  OUTPUT
*&---------------------------------------------------------------------*
MODULE status_0200 OUTPUT.
  SET PF-STATUS 'DYN_200'.
  SET TITLEBAR 'DYN_200' WITH sdyn_conn-carrid
                              sdyn_conn-connid
                              sdyn_conn-fldate.
ENDMODULE.                             " STATUS_0200  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  STATUS_0300  OUTPUT
*&---------------------------------------------------------------------*
MODULE status_0300 OUTPUT.
  SET PF-STATUS 'DYN_300'.
  SET TITLEBAR 'DYN_300' WITH sdyn_conn-carrid
                              sdyn_conn-connid
                              sdyn_conn-fldate.
ENDMODULE.                             " STATUS_0300  OUTPUT



*&---------------------------------------------------------------------*
*&      Module  TRANS_DETAILS  OUTPUT
*&---------------------------------------------------------------------*
MODULE trans_details OUTPUT.
  MOVE-CORRESPONDING: wa_spfli   TO sdyn_conn,
                      wa_sflight TO sdyn_conn,
                      wa_sbook   TO sdyn_book.
ENDMODULE.                             " TRANS_DETAILS  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  TRANS_TO_TC  OUTPUT
*&---------------------------------------------------------------------*
MODULE trans_to_tc OUTPUT.
```

Continued on next page


```
  MOVE-CORRESPONDING wa_book TO sdyn_book.
ENDMODULE.                              " TRANS_TO_TC  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  MODIFY_SCREEN  OUTPUT
*&---------------------------------------------------------------------*
MODULE modify_screen OUTPUT.
  LOOP AT SCREEN.
    CHECK screen-name = 'SDYN_BOOK-CANCELLED'.
    CHECK ( NOT sdyn_book-cancelled IS INITIAL ) AND
          ( sdyn_book-mark IS INITIAL ).
    screen-input = 0.
    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.                              " MODIFY_SCREEN  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  TABSTRIP_INIT  OUTPUT
*&---------------------------------------------------------------------*
MODULE tabstrip_init OUTPUT.
  CHECK tab-activetab IS INITIAL.
  tab-activetab = 'BOOK'.
  screen_no = '0301'.
ENDMODULE.                              " TABSTRIP_INIT  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  HIDE_BOOKID  OUTPUT
*&---------------------------------------------------------------------*
MODULE hide_bookid OUTPUT.
* hide field displaying customer number when working with number range
* object BS_SCUSTOM
  LOOP AT SCREEN.
    CHECK screen-name = 'SDYN_BOOK-BOOKID'.
    screen-active = 0.
    MODIFY SCREEN.
  ENDLOOP.
ENDMODULE.                              " HIDE_BOOKID  OUTPUT
```

Continued on next page

PAI Modules

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGSI01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&---------------------------------------------------------------------*
MODULE exit INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.
      CASE sy-dynnr.
        WHEN '0100'.
          LEAVE PROGRAM.
        WHEN '0200'.
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN '0300'.
          PERFORM deq_all.
          LEAVE TO SCREEN '0100'.
        WHEN OTHERS.
      ENDCASE.
    WHEN 'EXIT'.
      LEAVE PROGRAM.
    WHEN OTHERS.
  ENDCASE.
ENDMODULE.                             " EXIT  INPUT


*&---------------------------------------------------------------------*
*&      Module  SAVE_OK_CODE  INPUT
*&---------------------------------------------------------------------*
MODULE save_ok_code INPUT.
  save_ok = ok_code.
  CLEAR ok_code.
ENDMODULE.                             " SAVE_OK_CODE  INPUT


*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE save_ok.
**************************CANCEL BOOKING************************
```

Continued on next page

```
    WHEN 'BOOKC'.
      PERFORM enq_sflight_sbook.
      PERFORM read_sflight USING wa_sflight sysubrc.
* process returncode - if flight does not exist: e-message
      PERFORM process_sysubrc_bookc.
      PERFORM read_spfli USING wa_spfli.
      PERFORM read_sbook USING itab_book itab_cd.
      REFRESH CONTROL 'TC_SBOOK' FROM SCREEN '0200'.
****************************CREATE BOOKING*************************
    WHEN 'BOOKN'.
      PERFORM enq_sflight.
      PERFORM read_sflight USING wa_sflight sysubrc.
* process returncode - if flight does not exist: e-message
      PERFORM process_sysubrc_bookn.
      PERFORM read_spfli USING wa_spfli.
      PERFORM initialize_sbook USING wa_sbook.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN '0100'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT


*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0200  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0200 INPUT.
  CASE save_ok.
    WHEN 'SAVE'.
* collect marked (changed) data sets in seperate internal table
      PERFORM collect_modified_data USING itab_sbook_modify.
* perform database changes
      PERFORM save_modified_booking.
* create change documents
      PERFORM create_change_documents.
      COMMIT WORK.
* Unlocking data sets is executed by the update program !!
      SET SCREEN '0100'.
    WHEN 'BACK'.
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0200'.
  ENDCASE.
```

Continued on next page


```
ENDMODULE.                             " USER_COMMAND_0200  INPUT


*&---------------------------------------------------------------------*
*&      Module  MODIFY_ITAB  INPUT
*&---------------------------------------------------------------------*
MODULE modify_itab INPUT.
  wa_book-cancelled = sdyn_book-cancelled.
  wa_book-mark = 'X'.
  MODIFY itab_book FROM wa_book INDEX tc_sbook-current_line.
ENDMODULE.                             " MODIFY_ITAB  INPUT



*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0300  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0300 INPUT.
  PERFORM tabstrip_set.
  CASE save_ok.
    WHEN 'NEW_CUSTOM'.
      PERFORM create_new_customer.
      SET SCREEN '0300'.
    WHEN 'SAVE'.
      PERFORM save_new_booking.
      COMMIT WORK.
* Unlocking data sets is executed by the update program !!
      SET SCREEN '0100'.
    WHEN 'BACK'.
      PERFORM deq_all.
      SET SCREEN '0100'.
    WHEN OTHERS.
      SET SCREEN '0300'.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0300  INPUT


*&---------------------------------------------------------------------*
*&      Module  TRANS_FROM_0300  INPUT
*&---------------------------------------------------------------------*
MODULE trans_from_0300 INPUT.
  MOVE-CORRESPONDING sdyn_book TO wa_sbook.
ENDMODULE.                             " TRANS_FROM_0300  INPUT
```

Continued on next page

FORM Routines F01

```
*---------------------------------------------------------------------*
***INCLUDE BC414S_BOOKINGSF01 .
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  COLLECT_MODIFIED_DATA
*&---------------------------------------------------------------------*
*      -->P_ITAB_SBOOK_MODIFY  text
*---------------------------------------------------------------------*
FORM collect_modified_data USING p_itab_sbook_modify
                                 LIKE itab_sbook_modify.
  DATA: wa_book LIKE LINE OF itab_book,
        wa_sbook_modify LIKE LINE OF p_itab_sbook_modify.
  CLEAR: p_itab_sbook_modify.
* Only bookings are collected, that
* 1) have been changed (mark = 'X')
* 2) shall be cancelled (cancelled = 'X')
  LOOP AT itab_book INTO wa_book
       WHERE    mark = 'X'
       AND cancelled = 'X'.
    MOVE-CORRESPONDING wa_book TO wa_sbook_modify.
    APPEND wa_sbook_modify TO p_itab_sbook_modify.
  ENDLOOP.
ENDFORM.                                " COLLECT_MODIFIED_DATA


*&---------------------------------------------------------------------*
*&      Form  INITIALIZE_SBOOK
*&---------------------------------------------------------------------*
*      -->P_WA_SBOOK  text
*---------------------------------------------------------------------*
FORM initialize_sbook USING p_wa_sbook TYPE sbook.
  CLEAR p_wa_sbook.
  MOVE-CORRESPONDING wa_sflight TO p_wa_sbook.
  MOVE: wa_sflight-price    TO p_wa_sbook-forcurram,
        wa_sflight-currency TO p_wa_sbook-forcurkey,
        sy-datum            TO p_wa_sbook-order_date.
ENDFORM.                                " INITIALIZE_SBOOK


*&---------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKC
*&---------------------------------------------------------------------*
```

Continued on next page

Internal Use SAP Partner Only


```
FORM process_sysubrc_bookc.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0200'.
    WHEN OTHERS.
      PERFORM deq_all.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                                " PROCESS_SYSUBRC_BOOKC


*&---------------------------------------------------------------------*
*&      Form  PROCESS_SYSUBRC_BOOKN
*&---------------------------------------------------------------------*
FORM process_sysubrc_bookn.
  CASE sysubrc.
    WHEN 0.
      SET SCREEN '0300'.
    WHEN OTHERS.
      PERFORM deq_all.
      MESSAGE e023 WITH sdyn_conn-carrid sdyn_conn-connid
                        sdyn_conn-fldate.
  ENDCASE.
ENDFORM.                                " PROCESS_SYSUBRC_BOOKN


*&---------------------------------------------------------------------*
*&      Form  TABSTRIP_SET
*&---------------------------------------------------------------------*
FORM tabstrip_set.
  IF save_ok = 'BOOK' OR save_ok = 'DETCON' OR save_ok = 'DETFLT'.
    tab-activetab = save_ok.
  ENDIF.
  CASE save_ok.
    WHEN 'BOOK'.
      screen_no = '0301'.
    WHEN 'DETCON'.
      screen_no = '0302'.
    WHEN 'DETFLT'.
      screen_no = '0303'.
  ENDCASE.
ENDFORM.                                " TABSTRIP_SET
```

Continued on next page


```
*&---------------------------------------------------------------------*
*&      Form  NUMBER_GET_NEXT
*&---------------------------------------------------------------------*
*       -->P_WA_SBOOK  text
*---------------------------------------------------------------------*
FORM number_get_next USING p_wa_sbook LIKE sbook.
  DATA: return TYPE inri-returncode.
* get next free number in the number range '01' of number range
* object 'SBOOKID'
  CALL FUNCTION 'NUMBER_GET_NEXT'
       EXPORTING
            nr_range_nr = '01'
            object      = 'SBOOKID'
            subobject   = p_wa_sbook-carrid
       IMPORTING
            number      = p_wa_sbook-bookid
            returncode  = return
       EXCEPTIONS
            OTHERS      = 1.
  CASE sy-subrc.
    WHEN 0.
      CASE return.
        WHEN 1.
* number of remaining numbers critical
          MESSAGE s070.
        WHEN 2.
* last number
          MESSAGE s071.
        WHEN 3.
* no free number left over
          MESSAGE a072.
      ENDCASE.
    WHEN 1.
* internal error
      MESSAGE a073 WITH sy-subrc.
  ENDCASE.
ENDFORM.                                " NUMBER_GET_NEXT


*&---------------------------------------------------------------------*
*&      Form  CREATE_NEW_CUSTOMER
*&---------------------------------------------------------------------*
FORM create_new_customer.
  CALL TRANSACTION 'BC414S_CREATE_CUST'.
  GET PARAMETER ID 'CSM' FIELD wa_sbook-customid.
```

Continued on next page


```
* Alternative solution using the GET function of the screen field
* for customer ID
*   CLEAR wa_sbook-customid.
ENDFORM.                                " CREATE_NEW_CUSTOMER
```

F02

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF02
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  ENQ_SFLIGHT
*&---------------------------------------------------------------------*
FORM enq_sflight.
  CALL FUNCTION 'ENQUEUE_ESFLIGHT'
       EXPORTING
            carrid         = sdyn_conn-carrid
            connid         = sdyn_conn-connid
            fldate         = sdyn_conn-fldate
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e060.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                                " ENQ_SFLIGHT


*&---------------------------------------------------------------------*
*&      Form  ENQ_SFLIGHT_SBOOK
*&---------------------------------------------------------------------*
FORM enq_sflight_sbook.
  CALL FUNCTION 'ENQUEUE_ESFLIGHT_SBOOK'
       EXPORTING
            carrid         = sdyn_conn-carrid
            connid         = sdyn_conn-connid
```

Continued on next page


```
            fldate         = sdyn_conn-fldate
        EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e062.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " ENQ_SFLIGHT_SBOOK


*&---------------------------------------------------------------------*
*&      Form  ENQ_SBOOK
*&---------------------------------------------------------------------*
FORM enq_sbook.
  CALL FUNCTION 'ENQUEUE_ESBOOK'
       EXPORTING
            carrid         = sdyn_book-carrid
            connid         = sdyn_book-connid
            fldate         = sdyn_book-fldate
            bookid         = sdyn_book-bookid
       EXCEPTIONS
            foreign_lock   = 1
            system_failure = 2
            OTHERS         = 3.
  CASE sy-subrc.
    WHEN 0.
    WHEN 1.
      MESSAGE e061.
    WHEN OTHERS.
      MESSAGE e063 WITH sy-subrc.
  ENDCASE.
ENDFORM.                               " ENQ_SBOOK


*&---------------------------------------------------------------------*
*&      Form  DEQ_ALL
*&---------------------------------------------------------------------*
FORM deq_all.
  CALL FUNCTION 'DEQUEUE_ALL'.
ENDFORM.                               " DEQ_ALL
```

Continued on next page

F03

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF03
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  READ_SFLIGHT
*&---------------------------------------------------------------------*
*       -->P_WA_SFLIGHT  text
*       -->P_SYSUBRC     text
*---------------------------------------------------------------------*
FORM read_sflight USING p_wa_sflight TYPE sflight
                        p_sysubrc LIKE sy-subrc.
  SELECT SINGLE * FROM sflight INTO p_wa_sflight
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid
         AND   fldate = sdyn_conn-fldate.
  p_sysubrc = sy-subrc.
ENDFORM.                                " READ_SFLIGHT


*&---------------------------------------------------------------------*
*&      Form  READ_SBOOK
*&---------------------------------------------------------------------*
*       -->P_ITAB_BOOK  text
*       -->P_ITAB_CD    text
*---------------------------------------------------------------------*
FORM read_sbook USING p_itab_book LIKE itab_book
                      p_itab_cd   LIKE itab_cd.
  TYPES: BEGIN OF wa_custom_type,
           id TYPE scustom-id,
           name TYPE scustom-name,
         END OF wa_custom_type.
  DATA: wa_custom TYPE wa_custom_type,
        itab_custom TYPE STANDARD TABLE OF wa_custom_type
        WITH NON-UNIQUE KEY id,
        wa_book LIKE LINE OF p_itab_book,
        wa_cd   LIKE LINE OF p_itab_cd.
  CLEAR: p_itab_book, p_itab_cd.
* Select customer names in buffer table (array fetch)
  SELECT id name FROM scustom INTO CORRESPONDING FIELDS
```

Continued on next page

```
         OF TABLE itab_custom.
* Select all bookings on selected flight (array fetch)
  SELECT * FROM sbook INTO CORRESPONDING FIELDS OF TABLE p_itab_book
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid
         AND   fldate = sdyn_conn-fldate.
* read customer names corresponding to customer number from buffer
* table
  LOOP AT p_itab_book INTO wa_book.
    READ TABLE itab_custom INTO wa_custom WITH TABLE KEY
                                            id = wa_book-customid.
    wa_book-name = wa_custom-name.
    MODIFY p_itab_book FROM wa_book.
    MOVE-CORRESPONDING wa_book TO wa_cd.
    APPEND wa_cd TO p_itab_cd.
  ENDLOOP.
  SORT p_itab_book BY bookid customid.
ENDFORM.                                " READ_SBOOK


*&---------------------------------------------------------------------*
*&      Form  READ_SPFLI
*&---------------------------------------------------------------------*
*      -->P_WA_SPFLI  text
*----------------------------------------------------------------------*
FORM read_spfli USING p_wa_spfli TYPE spfli.
  SELECT SINGLE * FROM spfli INTO p_wa_spfli
         WHERE carrid = sdyn_conn-carrid
         AND   connid = sdyn_conn-connid.
  IF sy-subrc <> 0.
    PERFORM deq_all.
    MESSAGE e022 WITH sdyn_conn-carrid sdyn_conn-connid.
  ENDIF.
ENDFORM.                                " READ_SPFLI
```

F04

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF04
*---------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  SAVE_MODIFIED_BOOKING
```

Continued on next page


```
*&---------------------------------------------------------------------*
FORM save_modified_booking.
* Modify data on database tables sbook and sflight
  CALL FUNCTION 'UPDATE_SBOOK' IN UPDATE TASK
       EXPORTING
            itab_sbook = itab_sbook_modify.
  PERFORM update_sflight.
ENDFORM.                               " SAVE_MODIFIED_BOOKING


*&---------------------------------------------------------------------*
*&      Form  UPDATE_SFLIGHT
*&---------------------------------------------------------------------*
FORM update_sflight.
  CALL FUNCTION 'UPDATE_SFLIGHT' IN UPDATE TASK
       EXPORTING
            carrier    = wa_sflight-carrid
            connection = wa_sflight-connid
            date       = wa_sflight-fldate.
ENDFORM.                               " UPDATE_SFLIGHT


*&---------------------------------------------------------------------*
*&      Form  SAVE_NEW_BOOKING
*&---------------------------------------------------------------------*
FORM save_new_booking.
* transform amount from foreign currency to local currency (of carrier)
  PERFORM convert_to_loc_currency USING wa_sbook.
* number ranges: Get next number (internal)
  PERFORM number_get_next USING wa_sbook.
* lock booking to be created
  PERFORM enq_sbook.
  CALL FUNCTION 'INSERT_SBOOK' IN UPDATE TASK
       EXPORTING
            wa_sbook = wa_sbook.
  PERFORM update_sflight.
ENDFORM.                               " SAVE_NEW_BOOKING
```

F05

```
*---------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF05
*---------------------------------------------------------------------*
```

Continued on next page

Internal Use SAP Partner Only

```
*&---------------------------------------------------------------------*
*&      Form  CONVERT_TO_LOC_CURRENCY
*&---------------------------------------------------------------------*
*       -->P_WA_SBOOK  text
*----------------------------------------------------------------------*
FORM convert_to_loc_currency USING p_wa_sbook TYPE sbook.
  SELECT SINGLE currcode FROM scarr INTO p_wa_sbook-loccurkey
         WHERE carrid = p_wa_sbook-carrid.
  CALL FUNCTION 'CONVERT_TO_LOCAL_CURRENCY_N'
       EXPORTING
            client           = sy-mandt
            date             = sy-datum
            foreign_amount   = p_wa_sbook-forcuram
            foreign_currency = p_wa_sbook-forcurkey
            local_currency   = p_wa_sbook-loccurkey
       IMPORTING
            local_amount     = p_wa_sbook-loccuram
       EXCEPTIONS
            no_rate_found    = 1
            overflow         = 2
            no_factors_found = 3
            no_spread_found  = 4
            derived_2_times  = 5
            OTHERS           = 6.
  IF sy-subrc <> 0.
    MESSAGE e080 WITH sy-subrc.
  ENDIF.
ENDFORM.                                " CONVERT_TO_LOC_CURRENCY
```

F06

```
*----------------------------------------------------------------------*
*   INCLUDE BC414S_BOOKINGSF06
*----------------------------------------------------------------------*

*&---------------------------------------------------------------------*
*&      Form  CREATE_CHANGE_DOCUMENTS
*&---------------------------------------------------------------------*
FORM create_change_documents.
  LOOP AT itab_sbook_modify INTO sbook.
* read unchanged data from buffer table into *-work area
```

Continued on next page


```
    READ TABLE itab_cd FROM sbook INTO *sbook.
* define objectid from key fields of sbook
    CONCATENATE sbook-mandt sbook-carrid sbook-connid
                sbook-fldate sbook-bookid sbook-customid
                INTO objectid SEPARATED BY space.
* fill interface parameters of function, which itself is encapsulated
* in form CD_CALL_BC_BOOK
    MOVE: sy-tcode        TO tcode,
          sy-uzeit        TO utime,
          sy-datum        TO udate,
          sy-uname        TO username,
          'U'             TO upd_sbook.
* perform calls the neccessary function to create change document
* 'in update task'
    PERFORM cd_call_bc_book.
  ENDLOOP.
ENDFORM.                              " CREATE_CHANGE_DOCUMENTS
```

Lesson Summary

You should now be able to:

- Use the ABAP statements COMMIT WORK and ROLLBACK WORK appropriately.

Unit Summary

You should now be able to:

- Create number range objects and maintain number range intervals
- Use function modules to determine the next free number of an interval with internal number assignment
- Use function modules to check the validity of numbers assigned externally
- Locate information on function modules for managing number ranges
- Explain the meaning of buffered number assignment.
- Create change-document objects
- Create change documents
- Read change documents
- Find information on the function groups for managing change documents.
- Find information on authorization objects
- Create authorization objects
- Find information on authorizations and profiles
- Perform authorization checks in your program
- Link the execution of transaction codes to authorization objects
- Explain the SAP table buffer.
- Explain Native SQL.
- Explain cluster tables.
- Explain SAP locks.
- Explain BAPI architecture and the use of BAPIs.
- Use the ABAP statements COMMIT WORK and ROLLBACK WORK appropriately.

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Course Summary

You should now be able to:

- use the Open SQL statements for database updates
- implement the SAP locking concept for database updates
- implement different update techniques for database changes

Index

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Feedback

SAP AG has made every effort in the preparation of this course to ensure the accuracy and completeness of the materials. If you have any corrections or suggestions for improvement, please record them in the appropriate place in the course evaluation.

Internal Use SAP Partner Only