

Chapter 36: Programming Database Updates

Overview

To program database updates effectively, programmers are mainly concerned with:

- maintaining database correctness
- optimizing response times for users

The SAP System offers a wealth of methods for updating databases. Each takes advantage of different features of the SAP architecture. This chapter describes these methods and provides guidelines for choosing the method best tailored to your application.

The following topics provide information:

The SAP Database Environment

Update Programming Techniques

Locking in the SAP System

Contents

The SAP Database Environment.....	36-2
R/3 Architecture: Overview.....	36-2
Transactions in the SAP System.....	36-4
Introduction to Update Bundling	36-5
Introduction to SAP Locking	36-10
Update Programming Techniques	36-10
Maintaining Database Integrity	36-11
Optimizing Transaction Performance	36-12
Unbundled Updates.....	36-13
Bundled Updating in the Dialog Task.....	36-14
Bundled Updating in the Update Task	36-16
Bundled Updating in the Background Task.....	36-22
COMMIT WORK Processing	36-23
ROLLBACK WORK Processing.....	36-24
Background Processing Considerations.....	36-25
Error Handling for Bundled Updates.....	36-26
Locking in the SAP System	36-27
Defining Lock Objects	36-28
Calling ENQUEUE/DEQUEUE Function Modules.....	36-28

The SAP Database Environment

The SAP database environment offers some special features you should know about if you are updating a database in your transaction. For a quick introduction before programming updates, see:

Transactions in the SAP System

Introduction to Update Bundling

Introduction to SAP Locking

R/3 Architecture: Overview

In order to understand how the R/3 system executes database updates, you should be familiar with the R/3 architecture.

Client/Server Configuration

The three-level Client/Server configuration of SAP consists of the presentation (frontend for the user), the application logic (processing of the dialogs with the user), and the data storage (executing the database requests). The presentation server uses the SAPGUI program to provide the R/3 user interface. If a user starts a transaction, the SAPGUI transfers the entries to the dispatcher. The dispatcher distributes pending processing tasks to a number of work processes. The exact number depends on the configuration. The work processes can directly access the database, which may be accommodated on a different computer, using shared services.

Multi-User System

The dialog work processes alternately take control of current user sessions. If after user input a dynpro needs to be processed by the system, the dispatcher places it in a queue. As soon as a dialog work process is free, the job at the top of the queue is assigned to it for processing. The dialog work process then executes exactly one dialog step. After each change of screen, a database commit is triggered. This frees the dialog work process between screens, allowing it to process a new dialog step assigned to it by the dispatcher.

Switching Work Processes and Updating Databases

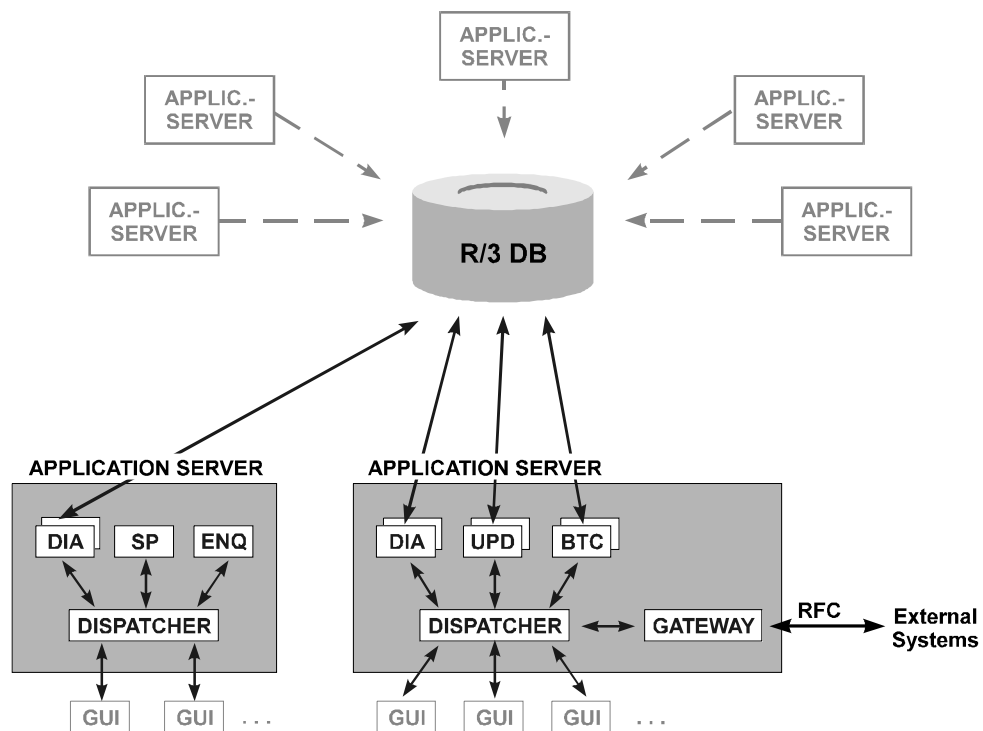
Due to assigning tasks to free work processes, the system resources are optimally used and loads are optimally divided. However, when designing the transaction flow, transaction developers must bear in mind that during the processing of the transaction the work process may be switched several times. This fact is important, if a transaction changes the database, since the system releases database locks and closes the database cursors as soon as a work process is freed. To keep changes correct, you must know when this point is reached and which actions to take. This chapter tells you which techniques to use to program correct and effective database changes. The techniques ABAP/4 offers for optimizing database updates are independent of the underlying database and correspond to the special requirements of dialog programming.

Update and Enqueue Service

In addition to the dialog work processes, each R/3 system contains one or more update services and one enqueue service. Use these services to update databases in R/3.

Dialog transactions can perform changes to the database either directly or indirectly. For direct changes, the update program is executed by the dialog work process. The dialog user is obliged to wait until the update operation has been finished before any new entries can be made. In asynchronous updating, the dialog part of transactions is separated from actual updating of the database (for example, for performance reasons). A special update work process executes the database update.

The enqueue service administers internal locking in the R/3 system. As a rule, locking mechanisms of relational database systems are not adequate for the requirements of R/3. Transactions whose dialog steps are handled by different work processes, must retain any assigned locks even when switching processes. Each lock must apply not only for the application server that executes the locking transaction, but also for any other server of a client/server configuration. For this reason, each R/3 system contains only one enqueue service. (For more information on logical locks, see *Introduction to SAP Locking*.)



Transactions in the SAP System

In the most general sense, a *transaction* is an operation that lets the user make changes to the database. This operation must be carried out in an "all or nothing" fashion. If the transaction runs successfully, all changes should be carried out. If the transaction encounters an error, no changes should be carried out, not even partially.

When an error occurs mid-way through a transaction, any database changes made up to that point should be undone. This leaves the database in the state it had before the transaction started.

In the SAP System, there is an important difference between a transaction at the database level and the transaction you design as a programmer. This difference (clarified in this section) is the motivation for the bundling techniques you use to implement database updates.

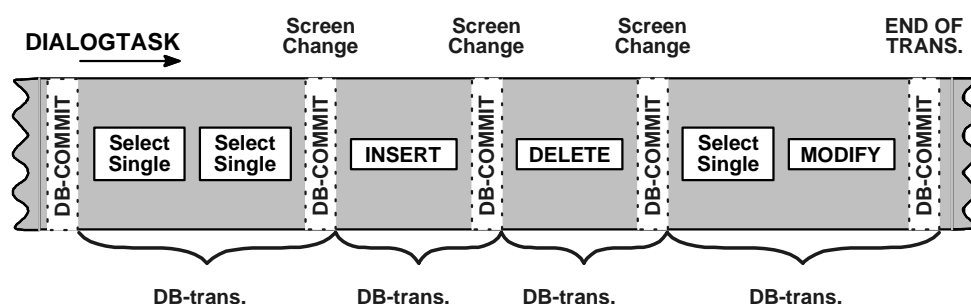
Transactions and LUWs

In the SAP System, the word transaction can have several meanings:

- Database transaction ("LUW" or "database LUW")

In the database world, an "all-or-nothing" transaction is called an LUW (Logical Unit of Work). An LUW is the span of time for which all updates requested must be performed as a unit. At the end of the LUW, the system either commits the updates to the database or throws them away. (Throwing away changes is called "rolling them back"). Every commit or rollback marks the end of one LUW and the beginning of the next.

The SAP System triggers database commit operations automatically at every screen change. This means that a database LUW lasts (at longest) from one screen change to the next.



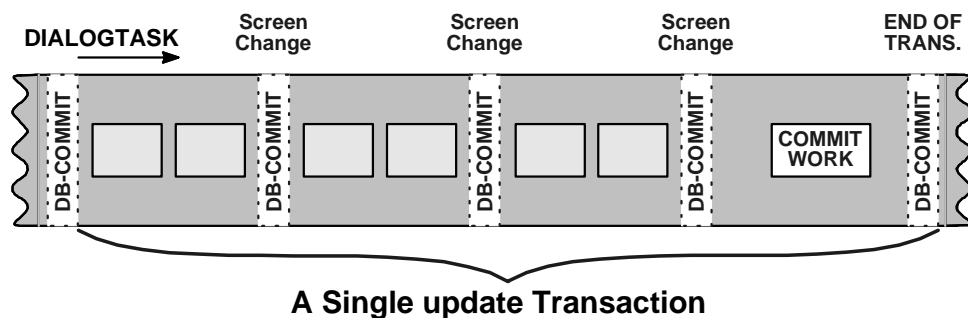
Database Transaction (LUW's)

- Update transaction ("SAP LUW")

A transaction is a business-related task that forms a logical unit. This task can take several SAP screens to perform, but constitutes a single function (for example, displaying a database, making changes to the database, or sending a message). Not

all transactions involve updating a database. However in this chapter, we are concerned only with those that perform updates (“update transaction”).

As a logical unit, update transactions should be executed entirely, or not at all. As a result, they are sometimes called SAP LUWs, to distinguish them from database LUWs. In general, an update transaction usually spans several database LUWs and is closed at the ABAP/4 level with a COMMIT WORK command. (This command performs several tasks, among them triggering a database commit.)



- ABAP/4 transaction (“SAP transaction”)

An ABAP/4 transaction is a set of business-related tasks combined under a single transaction code. ABAP/4 transactions may consist of several “update transactions” (as described in the previous point). As programs, ABAP/4 transactions are complex objects consisting of a module pool, screens, a menu interface, a transaction code, and other parts.

LUWs and Update Transactions

An “LUW” (logical unit of work) is the span of time during which any database updates must be performed in an “all or nothing” manner. Either they are all performed (committed), or they are all thrown away (rolled back). In the ABAP/4 world, LUWs and transactions can have several meanings:

- **LUW** (or “**database LUW**” or “**database transaction**”)

This is the set of updates terminated by a database commit. An LUW lasts, at most, from one screen change to the next (because the SAP System triggers database commits automatically at every screen change).

- **update transaction** (or “**SAP LUW**”)

This is a set of updates terminated by an ABAP/4 commit. An SAP LUW may last much longer than a database LUW, since most update processing extends over multiple transaction screens. The programmer terminates an update transaction by issuing a COMMIT WORK statement.

Introduction to Update Bundling

The ABAP/4 techniques for bundling updates let you program all-or-nothing transactions (logical LUWs) that extend over several screens. You commit changes for these transactions only at the end. This transaction has two important features:

- You can avoid your updates being committed at each screen change.
- You can lock the objects to be updated across multiple screens.

Introductory topics are:

Why Update Bundling?

Summary of Bundling Techniques

Updating in Different Work Processes

Synchronous and Asynchronous Updates

For complete details on bundling updates, see:

Update Programming Techniques

Why Update Bundling?

The SAP System automatically triggers “database commits” at every screen change and also at other times (every CALL SCREEN, CALL DIALOG, or CALL TRANSACTION statement, every MESSAGE statement, and every Remote Function Call).

These database commits allow the system to release the dialog task between screens. The commits occur internally in the system and cannot be influenced by the programmer.

Database rollbacks are similarly triggered whenever a runtime error occurs or your program issues an abend message (MESSAGE type “A”). In general, you can think of a database LUW as spanning all processing for a single screen.

The update transactions you program however, usually span several screens. As a result, you can’t rely on the database commits to finalize updates. If you did, the system would commit your updates sooner than desired (at every screen change), and they would not be rollback-able. If an error later occurred, or the user wanted to cancel the operation, it would be too late to roll back updates made in previous screens.

A related problem concerns database locks. Database locks are automatically created and released by the system. They are created for every update statement and released at every database commit. At latest then, database locks are released at every screen change, which means they do not provide exclusive access to objects you want locked protect for longer than a single screen.

To address these issues, the SAP System provides:

- Update bundling techniques in ABAP/4
ABAP/4 provides commands for bundling updates in special update routines. Execution of these routines is delayed until your program issues an explicit “SAP commit”. The SAP commit is an ABAP/4 statement (COMMIT WORK), that triggers a database commit, but performs other functions as well. With these techniques, you can execute updates at the end of the update transaction, rather than at every screen change.
- SAP locking
The system lets you define SAP locks on database objects. SAP locks are logical locks in the SAP System, not the database locks applied automatically. With SAP locks, you can lock objects to be updated across multiple screens.

Summary of Bundling Techniques

With update bundling, you package your updates in special routines that run only when your program issues a ABAP/4 commit. (Or, if the program issues an ABAP/4 rollback, the execution is cancelled.) To do this, you use:

```
PERFORM ON COMMIT  
CALL FUNCTION IN UPDATE TASK  
CALL FUNCTION IN BACKGROUND TASK
```

These statements specify that a given FORM routine or function module be executed not immediately, but rather at the next ABAP/4 commit. An ABAP/4 commit is an ABAP/4 statement that triggers a database commit, but also performs other functions. The ABAP/4 statements for performing these commits and rollbacks are:

```
COMMIT WORK  
ROLLBACK WORK
```

The COMMIT WORK and ROLLBACK WORK statements each perform many functions relevant to synchronized execution of tasks. One of these, for closing update transactions, is to trigger a database commit or database rollback. Like the database commits (or rollbacks) in an LUW, COMMIT WORK and ROLLBACK WORK define the “operation boundaries” (logical beginning and end) for a set of all-or-nothing updates. For this reason, you use these statements in the last LUW of the *update transaction*.

Using the ABAP/4 bundling techniques, you can tailor your updates procedures as needed. For introductory information, see:

Updating in Different Work Processes

Synchronous and Asynchronous Updates

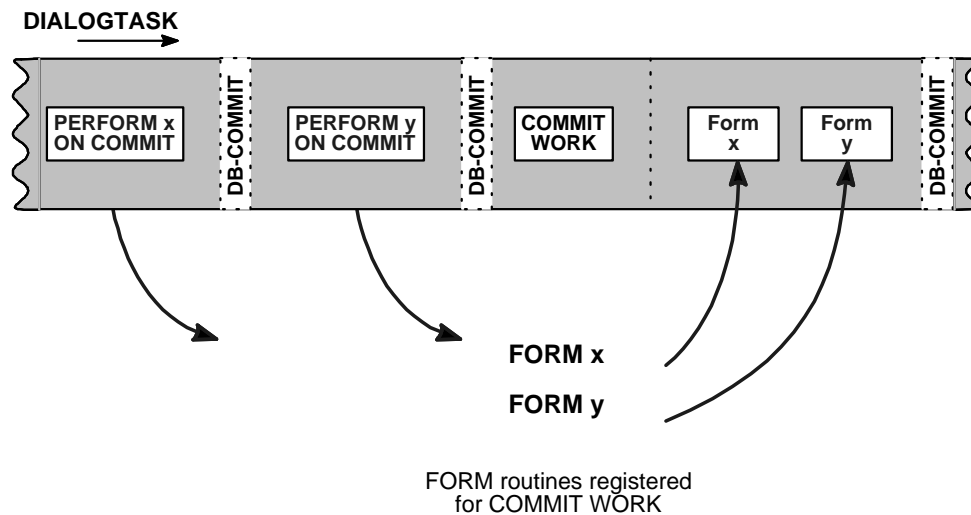
Updating in Different Work Processes

The ABAP/4 bundling techniques let you distribute your updates to different work processes. With each possible technique, actual execution is triggered with the COMMIT WORK statement:

- Updating in the dialog task

The PERFORM ON COMMIT statement calls a form routine in the dialog task, but delays its execution until the system encounters the next COMMIT WORK statement. Since the COMMIT WORK statement occurs at the logical end of the update transaction, any update statements in a form routine called with PERFORM ON COMMIT thus run in the last LUW for the *update transaction*.

DIALOG TASK UPDATING

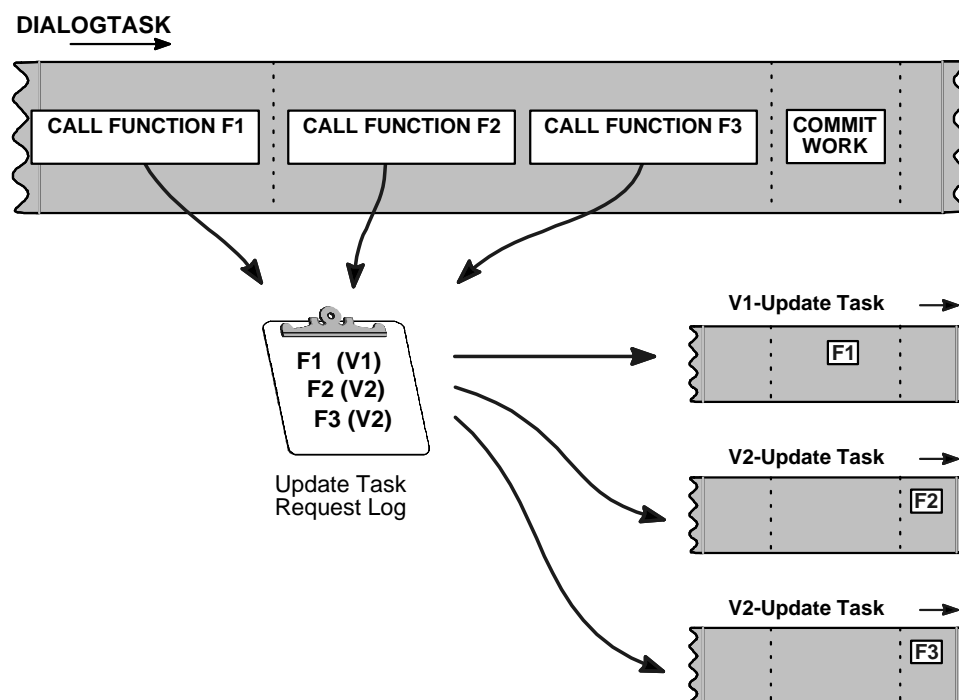


For more information, see *Bundled Updating in the Dialog Task*.

- Updating in the update task

The CALL FUNCTION IN UPDATE TASK statement logs a function module for execution in the update task. The subsequent COMMIT WORK statement triggers actual execution.

UPDATE-TASK UPDATING

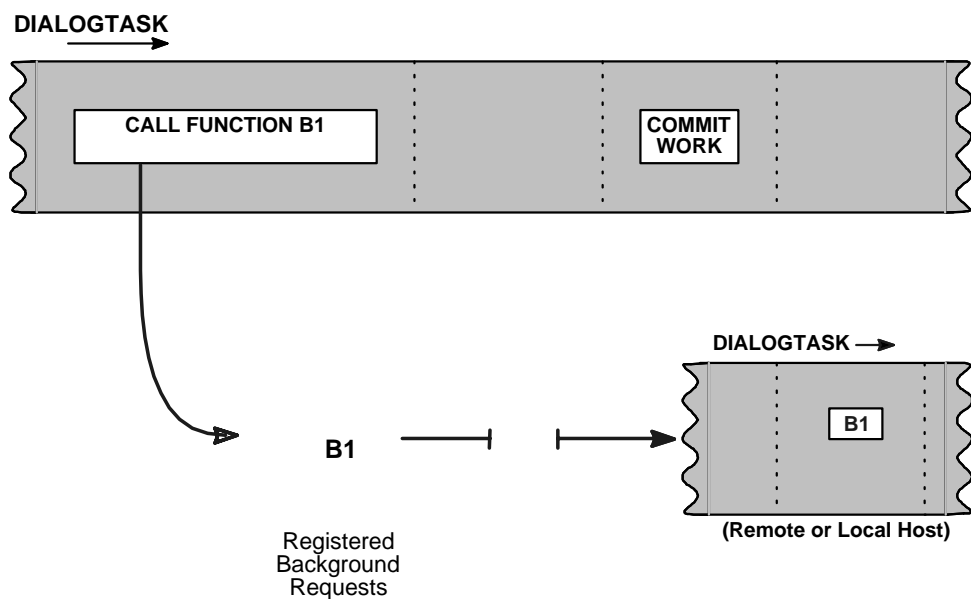


Function modules that run in the update-task have an attribute (the *Process type* field) that specifies how functions run and when. Some update functions (called “V1” functions) run in a single **update transaction** with all other functions of the same process type. These execute immediately, in the order requested. Other update functions (called “V2” functions) always run in their own individual update transactions. These have a delayed-start *Process type* and are for less critical requests. For more information, see *Bundled Updating in the Update Task*.

- Updating in a background task (remote host)

The CALL FUNCTION IN BACKGROUND TASK statement logs a function module to run in a background task. Normally, this statement is used to execute functions on remote hosts (by specifying an additional DESTINATION parameter). The subsequent COMMIT WORK statement triggers the actual remote function call. Background-task functions are processed as low-priority requests, but all requests for the same destination run in a common update transaction.

BACKGROUND PROCESS UPDATING



For more information, see *Bundled Updating in the Background Task*.

For information about RFC programming, see *Remote Communications*.

Synchronous and Asynchronous Updates

The concepts *synchronous* and *asynchronous* hinge on the notion of waiting. A program asks the system to perform a certain task, and then either waits or doesn't wait for the task to finish. In synchronous processing, the program waits: control returns to the program only when the task has been completed. In asynchronous processing, the

program does not wait: the system returns control after merely logging the request for execution.

In the SAP database world, waiting happens not at the requesting statement (the CALL FUNCTION or PERFORM), but rather at the COMMIT WORK that triggers execution. A synchronous update is guaranteed complete by the time the COMMIT WORK statement has finished. An asynchronous update is not guaranteed complete.

The SAP bundling techniques allow both kinds of processing:

- Synchronous updates:
 - a FORM routine called with PERFORM ON COMMIT
 - an update-task function module triggered with COMMIT WORK AND WAIT
- Asynchronous updates:
 - an update-task function module triggered with COMMIT WORK
 - an background-task function module triggered with COMMIT WORK

For more information, see *Optimizing Transaction Performance*.

Introduction to SAP Locking

To support the update bundling scheme, the SAP System offers a locking mechanism completely separate from database locks. SAP locks have the advantage that you can hold them across multiple screens, as needed for your *update transaction*.

Database locks are physical locks in the database system. The system automatically creates database locks whenever you use update statements (SELECT SINGLE FOR UPDATE, INSERT, UPDATE, MODIFY, DELETE) in a program. Database locks are released automatically with every database commit (that is, for every screen change). Thus, database locks are never available for longer than a single screen, and you as programmer have little control over them.

SAP locks are logical locks defined in the SAP System. To use them, you first define *lock objects* that specify the database objects you want to lock. When you activate lock objects, the system generates lock and unlock routines (function modules called **ENQUEUE-<object>** and **DEQUEUE-<object>**) for each lock object defined. You call these functions to set and release the locks explicitly from an ABAP/4 program.

For detailed information on using SAP locks, see *Locking in the SAP System*.

Update Programming Techniques

To implement database updates in a transaction, see the following topics:

Maintaining Database Integrity

Optimizing Transaction Performance

Unbundled Updates

Bundled Updating in the Dialog Task

Bundled Updating in the Update Task

Bundled Updating in the Background Task

COMMIT WORK Processing ROLLBACK WORK Processing

Background Processing Considerations

Error Handling for Bundled Updates

Maintaining Database Integrity

This section tells you what forms of data integrity are guaranteed by the R/3 system, and which you must guarantee yourself in your programs.

Semantic Integrity

Semantic integrity describes the correctness and completeness of the data with respect to itself. This is particularly true of update operations that consist of several steps. All the related steps must occur to guarantee semantic integrity. Some database systems use constraint or trigger techniques to guarantee semantic integrity, but SAP databases do not. It is up to you to:

- program transactions such that all necessary checks on operations occur
- issue a COMMIT WORK statement to finalize changes

Relational Integrity

Relational integrity means upholding the rules of the relational model used to design the database. All SAP databases are relational databases. In general, three kinds of relational integrity are at issue:

- **Primary key integrity**

Every object in a database must be uniquely identified by its primary key. In the ABAP/4 Dictionary it is not possible to create a table without a primary key. As a result, primary key integrity is guaranteed throughout the R/3 System. You never need to check database data for uniqueness.

- **Value set integrity and foreign key integrity**

In a relational database, no field may contain a value that is not one of the permissible values specified for the field. Also, every foreign key in a table must refer to a primary key in the associated check table. Many database systems automatically check that these conditions are maintained. SAP databases, however, do not. In programming transactions, you must keep two levels of protection in mind:

- Setting database values from screen data
You can specify allowable-value sets or foreign keys for fields in the ABAP/4 Dictionary. The Dictionary uses this information to require valid screen input from online users. If your program updates a database using screen field input, the data is guaranteed to be valid.
- Setting database values from non-screen data
Any ABAP/4 program can update a database using data from non-screen sources. This data is not guaranteed to be valid. In this case, it is your responsibility to program all the necessary checks to ensure the correctness of database values.

Operational Integrity

Operational integrity refers to the prevention of simultaneous updates to the same database object by multiple users. The standard solution to this problem is for each user to lock the objects he is currently using, in order to guarantee exclusive access.

The SAP System provides SAP locking, a mechanism for defining and applying logical locks to database objects. This mechanism is more comprehensive than the physical locks provided by the database system.

To guarantee operational integrity, you must use the SAP locking mechanism in your transactions. For more information, see *Locking in the SAP System*.

Optimizing Transaction Performance

Database accesses can be costly in time. The more time your transaction spends accessing a database, the longer the response time for the user. To optimize performance, you can distribute some of the transaction work to update-task (or possibly background-task) processing.

Choosing Asynchronous Processing

Asynchronous processing means updates can be distributed to other work processes or application servers. This usually lowers response times for users. Can and should you run your updates asynchronously? The choice is essentially a trade-off:

- Task results
With synchronous processing, when control returns to the requesting program, the processing requested is guaranteed to be finished. With asynchronous processing, the processing is not guaranteed to be finished. Does your program need the updates completed in order to continue processing? If yes, use synchronous processing.
- Response time
Response time is longer with synchronous processing: the user has to wait for completed updates. Use asynchronous updates if immediate updates to the database are not strictly required.

However, asynchronous updates do involve some overhead costs. When logging an update-task request, the system makes two extra database accesses: once to enter the updates into the log file, and once more to read them out. These accesses don't even include the actual updates that the update task performs. In general, asynchronous updates pay off only if one or more of the following conditions apply:

- User response time in the dialog task is very important.
- The response time is somewhat important and the required updates are so complex that they outweigh the cost of logging the requests.

For example, when a transaction runs in a batch work process, asynchronous processing offers no advantage. For more information, see *Background Processing Considerations*.

LUW processing: Which tasks are mutually dependent?

Some update requests are mutually dependent: they must be executed (or rolled back) together. Other requests can be run completely independently. You must decide whether

your updates need to run together as a logical unit. If so, they should either run together in the dialog task, or as common-LUW requests in the update task.

For update-task updates, you specify how requests are processed when you set the *Process type* attribute for an update-task function module. The *Process type* attribute determines whether update requests run in a common LUW, or in individual ones. See *How Update-Task Processing Works* for more information.

Update priority: how are off-loaded tasks carried out?

Asynchronous update requests can run either immediately or delayed. The immediate-start updates are common-LUW requests that execute in the order they were submitted. Delayed-start requests run later, in independent LUWs, and in any order. They may in fact be distributed to different work processes.

The *Process type* attribute (for update-task function modules) determines whether an update task request runs immediately or delayed. How critical is their timing? Must they run in the order you request them? See *How Update-Task Processing Works* for more information.

Unbundled Updates

You can make database updates in ABAP/4 without bundling them. When you do, however, you must be sure to consider the special aspects of the SAP database environment. The two sections that follow outline your options. In general, however, you should use the bundling update methods described in:

Bundled Updating in the Dialog Task

Bundled Updating in the Update Task

Bundled Updating in the Background Task

Inline Updates

You can place update statements (INSERT, UPDATE, MODIFY, DELETE) directly in your code. These are “inline” (that is, non-bundled) updates, executed without using any of the ABAP/4 bundling techniques. Even if you code no COMMIT WORK statement, the database commit at the next screen change will commit the updates to the database.

This inline method is only suitable for single-screen transactions. With multiple-screen transactions, if errors occur in later screens, you cannot roll back data committed in earlier screens.

If you use this method, do not rely on the automatic commits performed at screen change. SAP recommends you use COMMIT WORK explicitly before the end of screen processing.

Buffered Inline Updates

It is possible to buffer your changes so that you can make the updates at the end of your update transaction. These are still essentially inline updates, since you are not bundling them to execute during commit processing. However, the buffering allows you to collect updates over multiple screens and write them to the database when you are finished.

For example, many transactions take the user through a series of steps, each in a separate screen, and each involving a separate table update. The entire operation is only complete when the entire sequence has run successfully.

To perform simple inline updates, you could UPDATE each table while in the given screen. The result would of course be that, if an error occurred in a later screen, updates from previous screens could not be rolled back.

To perform buffered inline updates, you would save the changes for each screen in an internal table (or other storage) until the end of the operation. If all runs successfully, you can then make the updates for each table using the buffered information. This technique resembles ABAP/4 bundling in that you save the updates for later execution. However, the updates execute in the code as soon as the UPDATE statement (or similar statement) is reached. They do not need to be triggered by a COMMIT WORK statement.

Advantages and Disadvantages

When should you use buffered inline updates, and when should you bundle them with PERFORM ON COMMIT? Depending on your application, the following points may be relevant:

- Inline updating may feel simpler and more natural.
- PERFORM ON COMMIT may make more efficient use of database resources, since all accesses are performed at one time.
- Using PERFORM ON COMMIT everywhere guarantees consistency of implementation. Unintentional commits, or committing too early, are less likely.

Another important consideration is the interdependence of your code with other parts of the application. If your code generates database changes, and will be called by other programs or modules running in the same update transaction, you need a clear design for who performs commits, and when.

If called code generates changes and issues a COMMIT WORK, it may finalize changes generated in the caller that are not ready for committing. On the other hand, if the called code generates changes and does not issue a COMMIT WORK, the calling program may also neglect to commit. In this case, changes generated in the called code may be lost.

A good general rule is to decide where in the application commits should be made. Main or calling code can afford to use buffered inline updates, if desired. Called code on the other hand, if it generates updates, but will have relinquished control at the time of a commit, should schedule bundled updates with PERFORM ON COMMIT.

Bundled Updating in the Dialog Task

Updates performed in the dialog task are always synchronous. Dialog task updates can be bundled or unbundled. Unbundled updates are described in *Unbundled Updates*.

Using PERFORM ON COMMIT

You can bundle updates for dialog-task processing and have them execute in the last LUW for the transaction. You do this using the PERFORM ON COMMIT command. To do this:

1. Place all update statements (INSERT, UPDATE, MODIFY, DELETE, ...) in a FORM routine.
2. Call the routine with `PERFORM <form> ON COMMIT`.

The effect of using `PERFORM ON COMMIT`, instead of a simple `PERFORM`, is that the system delays execution of the form routine until it encounters a `COMMIT WORK` statement. You should position the `COMMIT WORK` at the end of the transaction, or wherever the user can select a *Save* function.



Caution

Do NOT use a `COMMIT WORK` or `ROLLBACK WORK` in the form routine itself.

Assigning Run Priorities to Form Routines

You can assign a run priority to each FORM routine by adding the `LEVEL` parameter to the `PERFORM ON COMMIT` statement. For example:

```
PERFORM update_table1 ON COMMIT LEVEL 2.
...
PERFORM update_table2 ON COMMIT LEVEL 3.
...
PERFORM update_table3 ON COMMIT LEVEL 1.
```

The `LEVEL` priorities specify when each should run, with respect to other routines also running during the same commit. The lower the level number, the higher the priority (that is, the earlier the execution) for the routine. In the example, when the `COMMIT WORK` statement is reached, the FORM routines will run in the order:

```
First:      update_table3
Second: update_table1
Third:      update_table2
```

Combining Synchronous and Asynchronous Execution

A FORM routine called with `PERFORM ON COMMIT` may contain any of the following statements:

```
INSERT, UPDATE, MODIFY, DELETE...
CALL FUNCTION
CALL FUNCTION IN UPDATE TASK
CALL FUNCTION IN BACKGROUND TASK DESTINATION
```

All four statements would execute in the dialog task, in the last LUW for the transaction. Any updates involved in the first two statements:

```
INSERT, UPDATE, MODIFY, DELETE...
CALL FUNCTION    "Updates inside the function module
run strictly synchronously. These updates are committed by the database commit
triggered for this COMMIT WORK. However the last two statements:
```

```
CALL FUNCTION IN UPDATE TASK
CALL FUNCTION IN BACKGROUND TASK DESTINATION
```

themselves schedule function modules to run in other work processes. When these functions contain update statements, they are performing asynchronous updating. The work processes requested (update- or background-task) are triggered at the end of the

commit processing already in progress. (That is, you would not need to code another COMMIT WORK statement to trigger these.)

The main reason to make calls like the latter two is because you want the functions to run with parameter values current at the time of the COMMIT WORK statement. For more information, see *Adding Update-Task Calls to a Subroutine*.

Bundled Updating in the Update Task

The SAP System provides an update task dedicated solely to performing database updates. You can place all your updates in an function module and execute it in the update task using the statement CALL FUNCTION IN UPDATE TASK.

For this statement, the system writes your request (function module and parameter information) to a log table. At the next SAP commit, the system passes the request to the update task for execution. You can issue an SAP commit with either a COMMIT WORK (asynchronous) or COMMIT WORK AND WAIT (synchronous).



Caution

Remember that an update-task function is triggered by an SAP commit in the calling code. Do NOT use a COMMIT WORK in the update-task function itself.

The following topics provide a guide to writing and using update-task function modules:

How Update-Task Processing Works

Creating Update-Task Function Modules

Calling Update-Task Function Modules

Special LUW Considerations

Accessing the Update Task Log

How Update-Task Processing Works

To program asynchronous updates effectively, you must understand how the update task works in the R/3 System. A typical SAP System configuration includes both dialog tasks and update tasks. Dialog tasks handle all interactive sessions with online users. The update tasks are dedicated solely to performing database updates.

You can perform updates in the dialog task as well as in the update-task. Dialog-task updates are synchronous updates. Update-task updates are asynchronous. (An exception is when you trigger an update-task function with COMMIT WORK AND WAIT: this is synchronous.)

Inside your transaction, you can call one or more update-task function modules for each COMMIT WORK statement.

Requesting Update-Task Processing

To request update-task processing, you program all updates in a function module, and call it with the statement CALL FUNCTION...IN UPDATE TASK. Function modules called IN UPDATE TASK must be update-task function modules. That is, you must set update-task attributes for the functions. These attributes specify a priority for the function, and tell whether it runs in a common *update transaction* (SAP LUW), or in its own.

(For more information, see *Creating Update-Task Function Modules*.)

Timing in Asynchronous Processing

The priority you choose for an update-task function module determines how it runs relative to other functions triggered with the same COMMIT WORK. High priority functions ("V1") all run in a common update transaction. Lower priority functions ("V2") always run alone in individual update transactions.

The V1 and V2 distinction separates time-critical from non-time-critical database changes. Time-critical changes (in V1 functions) must be performed as soon as possible and are the updates reflecting a moment-to-moment situation (for example, for flight reservations or warehouse movements). Less critical changes (in V2 functions) need not be performed immediately (for example, periodic statistical updates showing profitability analysis).

The system handles all V1 functions for a single COMMIT WORK before processing any V2 functions. If the V1 functions run without errors, the system releases all the relevant ENQUEUE locks, and deletes the functions from the log table. Finally, the system starts processing for the V2 functions in the log table.

V1 function modules requested for a single COMMIT WORK are always processed in the order they were called. V2 function modules can be processed in any order, even in parallel in different update tasks. However, no V2 function module should rely on ENQUEUE locks created for other V2 modules.

V1 function modules requested for different COMMIT WORK statements can also be processed in parallel in different update processes.

Asynchronous Error-Handling

In contrast to errors that occur in the dialog task, errors in the update task cannot be corrected by the user. Instead, the system stops processing the current update function and rolls back any database changes already performed for this module. Other changes made in previous function modules are also rolled back, depending on whether they are considered to be part of the same **update transaction** (SAP LUW) or not.

- All V1 functions triggered by the same COMMIT WORK are considered part of a single update transaction.

Thus, an error in one V1 function module causes a rollback of all updates made for other V1 functions logged for this COMMIT WORK. All the function modules remain in the log table and the incorrect module receives the marking ERR.

- Each V2 function module for a given COMMIT WORK always runs in its own (separate) update transaction.

An error in one V2 function module causes no rollbacks for other V2 modules. Only changes for the error function are rolled back, and the function itself remains in the log table marked as an error. All other V2 functions are executed.

For details on runtime error handling, see *Error Handling for Bundled Updates*.

The system marks rolled-back function modules as error functions in the update-task log. The error can then be corrected and the function restarted later. To access the update-task log, select the *Tools* → *Administration* → *Monitoring* → *Update*.

For details on this transaction, see *Accessing the Update Task Log*.

Creating Update-Task Function Modules

Writing update-task function modules is essentially like writing other function modules. However, a few special considerations are presented here.

To create a function module, you must first get into the function library. (Select *Tools* → *ABAP/4 Workbench* and then press the *Function Library* button). For general information on creating and writing function modules, see the *BC ABAP/4 Workbench Tools*.

Registering Update-Task Function Modules

Function modules that run in the update task must be registered as such in the function library. When you create the function module, set the *Process Type* attribute to one of the following values:

- *Update with immediate start*

Set this option for high priority (“V1”) functions that run in a shared **update transaction** (SAP LUW). These functions can be restarted by the update task in case of errors.

- *Update w. imm. start, no restart*

Set this option for high priority (“V1”) functions that run in a shared update transaction. These functions may not be restarted by the update task.

- *Update with delayed start*

Set this option for low priority (“V2”) functions that run in their own update transactions. These functions can be restarted by the update task in case of errors.

To reach the attribute values screen inside the function library, select *Goto* → *Administration*

Defining the Interface

Function modules that run in the update task have a limited interface:

- Result parameters or exceptions are not allowed since update-task function modules cannot report on their results.
- You must specify input parameters and tables with reference fields or reference structures defined in the ABAP/4 Dictionary.

Calling Update-Task Function Modules

Synchronous or Asynchronous Processing?

Function modules that run in the update task can run synchronously or asynchronously. You determine this by the form of the commit statement you use:

- **COMMIT WORK**

This is the standard form, which specifies asynchronous processing. Your program does not wait for the requested functions to finish processing.

- **COMMIT WORK AND WAIT**

This form specifies synchronous processing. The commit statement waits for the requested functions to finish processing. Control returns to your program after all high priority (V1) function modules have run successfully.

The AND WAIT form is convenient for switching old programs to synchronous processing without having to re-write the code. Functionally, using AND WAIT for update-task updates is just the same as dialog-task updates with **PERFORM ON COMMIT**.

Parameter Values at Execution

In ABAP/4, you can call update-task function modules in two different ways. The way you choose determines what parameter values are used when the function module is actually executed. Parameter values can be set either at the time of the **CALL FUNCTION** statement, or at the time of the **COMMIT WORK**. The following sections explain.

**Note**

The examples in these sections show asynchronous commits with **COMMIT WORK**.

Calling Update Functions Directly

To call a function module directly, use **CALL FUNCTION IN UPDATE TASK** directly in your code.

```
CALL FUNCTION 'FUNCTMOD' IN UPDATE TASK EXPORTING...
```

The system then logs your request and executes the function module when the next COMMIT WORK statement is reached. The parameter values used to execute the function module are those current at the time of the call. For example:

```
a = 1.  
CALL FUNCTION 'UPD_FM' IN UPDATE TASK EXPORTING PAR = A...  
a = 2.  
CALL FUNCTION 'UPD_FM' IN UPDATE TASK EXPORTING PAR = A...  
a = 3.  
COMMIT WORK.
```

Here, the function module UPD_FM is performed twice in the update task: the first time, with value 1 in PAR, the second time with value 2 in PAR.

Adding Update-Task Calls to a Subroutine

You can also put the CALL FUNCTION IN UPDATE TASK into a subroutine and call the subroutine with:

```
PERFORM SUBROUT ON COMMIT.
```

If you choose this method, the subroutine is executed at the commit. Thus the request to run the function in the update task is also logged during commit processing. As a result, the parameter values logged with the request are those current at the time of the commit. For example:

```
a = 1.  
PERFORM F ON COMMIT.  
a = 2.  
PERFORM F ON COMMIT.  
a = 3.  
COMMIT WORK.  
  
FORM f.  
  CALL FUNCTION 'UPD_FM' IN UPDATE TASK EXPORTING PAR = A.  
ENDFORM.
```

In this example, the function module UPD_FM is carried out with the value 3 in PAR. The update task executes the function module only once, despite the two PERFORM ON COMMIT statements. This is because a given function module, logged with the same parameter values, can never be executed more than once in the update task. The subroutine itself, containing the function module call, may not have parameters.



Note

The method described here is not suitable for use inside dialog module code. However, if you absolutely need to use this method in a dialog module, see *Dialog Modules that Call Update-Task Functions*.

Special LUW Considerations

In the update-task queue, the system identifies all function modules belonging to the same **update transaction** (SAP LUW) by assigning them a common update key. At the next COMMIT WORK, the update task reads the queue and processes all requests with the predefined update key.

If your program calls an update-task function module, the request to execute the module (or the subroutine calling it) is provided with the update key of the current LUW and placed in the queue.

What happens with LUW's when update-task functions calls are embedded in modules (transactions or dialog modules) called by other programs? The following sections explain.

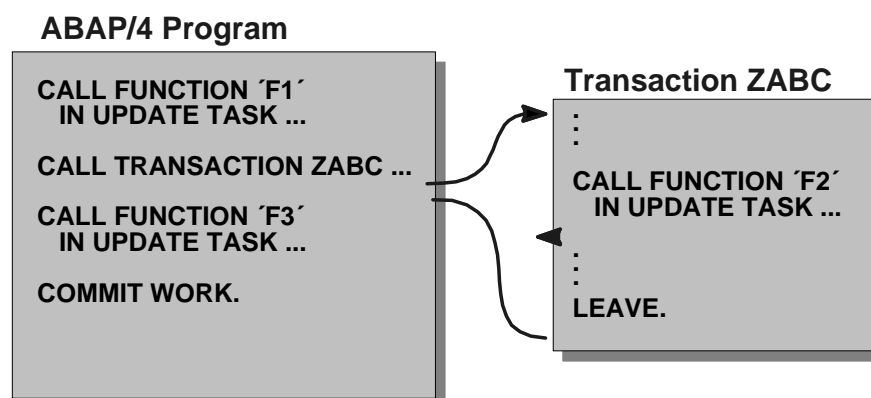
Transactions that Call Update-Task Functions

If your program calls a transaction (or a report) that itself calls an update-task function module, you should be aware of the following:

Every called program or report begins a new *update transaction* (SAP LUW), and with it, a new update key. This key is used to identify all update-task operations requested during the called transaction (or report).

When returning from the transaction (or report), the LUW of the calling program is restored together with the old update key.

If the called transaction (or report) does not contain its own COMMIT WORK, neither the requested database changes nor the calls to update-task function modules are performed. For example, in the following code, F1, F2, and F3 are update-task function modules:



Here, F1 and F3 are executed in the update task, because the COMMIT WORK for the main program triggers their execution. However, since transaction ZABC contains no COMMIT WORK statement, the function F2 is never executed by the update task.

Dialog Modules that Call Update-Task Functions

Unlike transactions and reports, dialog modules do not start a new *update transaction*. Calls to update-task function modules from a dialog module use the same update key as the ones in the calling program. The result is that calls to update-task function modules from a dialog module are executed only if a COMMIT WORK statement occurs in the calling program.

**Note**

If you place a COMMIT WORK in a dialog module, it does commit changes to the database (for example, with UPDATE). However, it does not start the update task. If the dialog module calls update-task function modules, the function modules are not triggered until a COMMIT WORK in the calling program is reached.

**Note**

If you are using dialog modules, try to avoid including calls to update-task function modules in subroutines called with PERFORM ON COMMIT. In general, any occurrence of PERFORM ON COMMIT (with or without update-task function calls) in a dialog module can be problematic.

The reason for this is that dialog modules run in their own roll area, and this roll area disappears when the module finishes. This means that all local data (including data used as parameter values when calling an update-task function module) disappears as soon as the commit in the main program is reached.

If you must use this method in a dialog module (i.e. include the call to an update-task function in a subroutine), you must ensure that the values of the actual parameters still exist when the update-task function actually runs. To do this, you can store the required values with EXPORT TO MEMORY and then import them back into the main program (IMPORT FROM MEMORY) before the COMMIT WORK statement.

Accessing the Update Task Log

To access the update log, select the *Tools* → *Administration* → *Monitoring* → *Update*. In the main screen,

1. Enter the user name in the *User* field.
2. Click on the correct *Status* option for the update records you want to see.
3. Press *Enter*.

The update records in the log either have not yet been written to the database (*Status* = *To be executed*) or terminated with errors (*Status* = *Terminated*).

In case of errors, you can re-run the update in debugging mode by selecting it in the log and then selecting *Administration* → *Debugging*.

Bundled Updating in the Background Task

You can execute updates on a remote host. This might be desirable, for example, if you are maintaining replicated data in two databases. While making the primary updates in the update task of the local system, you would also execute the same updates on a remote database. The system uses RFC (the Remote Function Call, protocol) to send requests to the remote system.

To execute updates on a remote host, use the statement:

```
CALL FUNCTION <fctmod> IN BACKGROUND TASK DESTINATION <dest>...
```

This statement requests execution of an update function in an external system. The `DESTINATION` parameter specifies the remote system. On encountering this statement, the system writes your request to a log table and then executes it (that is, forwards it to the remote system) at the next `COMMIT WORK`.

The statement `CALL FUNCTION IN BACKGROUND TASK` can also be used without the `DESTINATION` parameter. In this case, the function runs in a separate work process, but on the local host. However, there is no advantage to using the statement this way, since the result would be just like running the function in the update task.

All remote-host background-task functions triggered with the same `COMMIT WORK` run together in a common LUW. When you specify a `DESTINATION` parameter, all functions requested for the same destination system (and triggered by the same `COMMIT WORK`) run in a common LUW.

For more information on RFC programming, see *Remote Communication*.

**Note**

For updates on remote systems, background-task processing entails only a single-phase commit. No two-phase commit is performed.

COMMIT WORK Processing

The `COMMIT WORK` statement performs many functions relevant to synchronized execution of tasks. These tasks most often involve database updates, but need not do so. Many operations needed in a distributed processing environment rely on synchronized function execution.

For performing updates, `COMMIT WORK` ends your *update transaction* and commits the database changes. Since `COMMIT WORK` closes the transaction, it is often used to implement the *Save* function.

In general, the `COMMIT WORK` statement performs the following processing:

- Executes all `FORM` routines requested with `PERFORM ON COMMIT`.
The routines are executed in order of ascending priority, as specified by the `LEVEL` parameter for the `PERFORM` statement. For information on this parameter, see *Bundled Updating in the Dialog Task*.
- Triggers all update-task function modules, if requested.
- Triggers all background-task function modules, if requested.
- Triggers a database commit (which in turn releases the database lock)
- Empties the rollback log.
The rollback log contains a snapshot of the table before any changes are applied. This snapshot is used to reset the table to its original values in case a rollback is performed.
- Closes all open database cursors.
- Writes all `TEMSE` objects to permanent files or a database.

TEMSE files are TEMPorary SEquential files buffered during the transaction for performance reasons. Examples of TEMSE files are spool objects or job logs.

- Resets the time slice counter (for access to the work process) back to 0.

A time-slice counter in the system limits the amount of time your program can run in a work process. You can use COMMIT WORK to gain more time for your program, if the program often exceeds the time slice limit.

To do this, however, the processing must be easily divisible into smaller units (all-or-nothing operations). You can then insert a COMMIT WORK statement at the end of every unit. Processing units should of course be logically independent since, in the case of an error, you cannot cancel updates committed in preceding units.



Caution

Do not use COMMIT WORK in any FORM routines called with PERFORM ON COMMIT or in any function module that runs in the update-task. Doing either of these leads to a runtime error. By contrast, you may use COMMIT WORK in a background-task function module. However, you should ensure that your commits (and rollbacks) works correctly.

ROLLBACK WORK Processing

The ROLLBACK WORK statement "cancels" all requests for synchronized execution of tasks. In particular, when the tasks involve database updates, ROLLBACK WORK "discards" all updates for the current transaction:

- discards all FORM routines previously logged with PERFORM ON COMMIT
- marks all previously requested update-task functions as errors in the update task queue
- discards all previously requested background-task functions
- deletes all TEMSE objects (TEMPorary SEquential files like spool objects or job logs) from buffered storage
- triggers a database rollback operation (which in turn releases all database locks)
- closes all open database cursors

Note that all of these operations occur automatically anyway, whenever either of the following occurs:

- a runtime error
- a program issues an "A" MESSAGE (abend).

In these cases, the program terminates. By contrast, the ROLLBACK WORK statement lets your program continue processing after the error. For example, you may want to use ROLLBACK WORK whenever the user wants to cancel an operation or leave a transaction early.



Caution

Do not use ROLLBACK WORK in any FORM routines called with PERFORM ON COMMIT or in any function module that runs in the update-task. Doing either of

these leads to a runtime error. By contrast, you may use ROLLBACK WORK in a background-task function module. However, you should ensure that your rollback logic works correctly.

Background Processing Considerations

The SAP System provides work processes for running jobs in the background. Programs that run in a background process run continuously to the end. They are never interrupted to give other programs access to the work process. As a result, when screen processing is needed in a background process, the system does not trigger the automatic database commits that would occur for online execution.

In background processing, there is no user to call up transactions directly. However, any report can start a transaction with the CALL TRANSACTION USING keywords. As a result, many transactions may run both in a dialog task or in background.

This is important because the asynchronous updates requested for an online transaction may be a performance disadvantage when the transaction runs in background. To execute updates in the update task, the system must log the request and its data (in the database), and then fetch it again when the update task runs. When the transaction is executed in background, these logging activities are wasted effort. The updates could just as well be executed right there in the background process (that is, synchronously).

Requesting Local Updates

In ABAP/4, you can tell the system to perform updates locally, rather than in the update task. To do this, use the statement SET UPDATE TASK LOCAL whenever your program runs in background. Your code can check whether it is currently running in background by querying the SY-BATCH system variable.

The SET UPDATE TASK LOCAL statement sets a “local-update” switch for your program. When this switch is set, the system interprets CALL FUNCTION IN UPDATE TASK as a local-update request. Each function and its data are logged in SAP memory (rather than on the database) and then executed locally (in the same task with the caller) as part of COMMIT WORK processing. Your transaction waits for update completion before continuing.

As in the normal update task, if the program issues any messages (except type S), update processing terminates and causes a rollback. Unlike the normal update task however, if a rollback is required, both the dialog- and update-task parts of the transaction are rolled back. (This is because local updates share the LUW with the dialog part of the transaction.) You cannot use V2 functions (delayed start) with local updates.

(For information on V1 and V2 functions, see *How Update-Task Processing Works*.)

Where to use SET UPDATE TASK LOCAL

The local-update switch is off by default, and reset to off after every COMMIT WORK or ROLLBACK WORK. So you need a SET UPDATE TASK LOCAL statement at the beginning of every update transaction. (Update transactions and LUWs are defined in *Transactions in the SAP System*.)

```
IF SY-BATCH NE SPACE.  
  SET UPDATE TASK LOCAL.  
ENDIF.  
...  
CALL FUNCTION UPDATE_TABLE1 IN UPDATE TASK.  
...  
CALL FUNCTION UPDATE_TABLE2 IN UPDATE TASK.  
...  
COMMIT WORK.
```

If you call any external transactions or submit reports, these programs should have SET UPDATE TASK LOCAL statements in them too (because they get their own update transactions). Dialog modules however, do not need a SET statement.

The SET statement must execute before any CALL FUNCTION IN UPDATE TASK statements in the transaction. If a CALL FUNCTION runs before the SET statement is reached, the SET fails (returns SY-SUBRC = 1). In this case, *all* the function calls for the transaction (both before and after the SET) are treated as normal update-task requests.

Checking on Update Results

With normal asynchronous processing, you can check on results by looking in the update task queue. When errors occur, the request remains in the queue and is marked as an error.

If you use local updating, the SAP memory that held your request goes away at the end of your transaction. As a result, you cannot look there to find out how the updates ran. However, you can check in the batch log for a record of background processing steps. To access the batch log, select *System* → *Services* → *Jobs* → *Job overview* in any screen.

Note that if you use local updating when not running in background, the system cannot provide you with a report on results. You will get no feedback at all about whether your updates succeeded.

Error Handling for Bundled Updates

Runtime errors can occur during execution of bundled updates. How are they handled? In general, COMMIT WORK processing occurs in the following order:

1. All dialog-task FORM routines logged with PERFORM ON COMMIT are executed.
2. All high-priority (V1) update-task function modules are executed.

The end of V1-update processing marks the end of the *update transaction*. If you used COMMIT WORK AND WAIT to trigger commit processing, control returns to the dialog-task program.

3. All low-priority (V2) update-task function modules are triggered.
All background-task function modules are triggered.

Runtime errors can occur either in the system itself, or because your program issues an abend message (MESSAGE type 'A'). Also, the ROLLBACK WORK statement automatically signals a runtime error. The system handles errors according to where they occur:

- **in a FORM routine** (called with PERFORM ON COMMIT)
 - Updates already executed for the current update transaction are rolled back.

- No other FORM routines will be started.
- No further update-task or background-task functions will be started.
- An error message appears on the screen.
- **in a V1 update-task function module** (requested IN UPDATE TASK)
 - Updates already executed for V1 functions are rolled back.
 - All further update-task requests (V1 or V2) are thrown away.
 - All background-task requests are thrown away.
 - Updates already executed for FORM routines called with PERFORM ON COMMIT are **not** rolled back.
 - An error message appears on the screen, if your system is set up to send them.
- **in a V2 update-task function module** (requested IN UPDATE TASK)
 - Updates already executed for the current V2 function are rolled back.
 - All update-task requests (V2) still to be executed are carried out.
 - All background-task requests still to be executed are carried out.
 - No updates for previously executed V1 or V2 function are rolled back.
 - No updates previously executed for FORM routines (called with ON COMMIT) are rolled back.
 - An error message appears on the screen, if your system is set up to send them
- **in a background-task function module** (requested IN BACKGROUND TASK DESTINATION)
 - Background-task updates already executed for the current DESTINATION are not rolled back.
 - All further background-task requests for the same DESTINATION are thrown away.
 - No other previously-executed updates are not rolled back.
 - No error message appears on the screen.

If your program detects that an error in remote processing has occurred, it can decide whether to resubmit the requests at a later time.

For information about RFC programming, see *Remote Communications*.

Locking in the SAP System

The SAP locking mechanism allows programmers to set and release locks throughout an entire *update transaction*. This mechanism consists of two basic steps:

1. Defining a lock object in the ABAP/4 Dictionary

Lock objects are logical locks referring to one or more tables. Activating these lock objects causes the system to generate special function modules for locking and unlocking the relevant table rows. These function modules are called ENQUEUE-<object> and DEQUEUE-<object>.

See *Defining Lock Objects*
2. Calling ENQUEUE-<lock-object> and DEQUEUE-<lock-object> in your program

To use SAP locks, your program calls ENQUEUE-<lock-object> and DEQUEUE-<lock-object> before and after accessing database objects.

See *Calling ENQUEUE/DEQUEUE Function Modules*

Defining Lock Objects

A lock object indicates the database objects you want to lock. The database object can be a specific table or collection of tables.

In defining the lock object, you specify the tables involved, and the lock argument. The lock argument consists of all the primary-key fields for the specified tables. For example,

When you lock the object at runtime (with `ENQUEUE-<lock-object>`), you use the lock argument fields as input parameters. By specifying values for the argument fields, you can lock table rows individually or generically. WHERE-conditions specifying lock-argument values can only contain AND conditions (no OR conditions).

Setting the Lock Mode

Lock objects also specify the kind of locking possible for the table:

- **Shared lock**
This mode allows multiple users to access the specified table rows, but with read-access only. No write-accesses are allowed at any time.
- **Exclusive lock**
This mode gives a single user read and write access to the specified table rows. No other users may access the rows.
- **Extended exclusive lock**
This mode prevents a single user with read-write access from attaining further locks to the same set of table rows. This is useful when you are using recursive routines to make updates.

You define this lock object in the Data Dictionary and then activate it. For information on these tasks, see *BC ABAP/4 Dictionary*.

Calling ENQUEUE/DEQUEUE Function Modules

Activating a lock object causes the system to generate special function modules for locking and unlocking the objects. These function modules are called:

```
ENQUEUE_<lock-object-name>  'For locking objects'
DEQUEUE_<lock-object-name>  'For unlocking objects'
```

At runtime, you can then lock the database object before attempting to read or write it. To lock the object, call the function module `ENQUEUE_<lock-object-name>` in the PAI event for the first screen. To unlock it again, call `DEQUEUE_<lock-object-name>`.

ENQUEUE/DEQUEUE Parameters

The ENQUEUE/DEQUEUE function modules have the following set of parameters:

- `arg` and `x_arg` (ENQUEUE and DEQUEUE)

These two EXPORTING parameters exist for every field *arg* in the lock argument. Set *arg* to the key-field value you want. If you don't want a specific value for the field, omit the *arg* parameter, or set it to the field's initial value. If you want the field's initial value as the actual selection value, indicate this by setting *x_arg* to 'X'.

- **_SCOPE** (ENQUEUE)

If your transaction calls no update task functions, you are updating only in the dialog task. You should explicitly release your locks with the appropriate DEQUEUE function.

If you call any V1 update-task functions, set parameter **_SCOPE** to tell the system how SAP locks should be released. Possible values are:

- 1 This value is for creating locks that will not be needed in the update task. Locks with **_SCOPE=1** are held throughout dialog-task processing, but are not available for any update-task requests. To be sure you don't hold the lock longer than necessary, you should release it explicitly (with the appropriate DEQUEUE function) when your transaction ends.

Release at ROLLBACK WORK: The system does not release locks with **_SCOPE=1**. Use the DEQUEUE function when you program a rollback.

- 2 This value is for creating locks that are
 - used in the dialog task until the update task is triggered
 - used only in the update task, once the update task has been triggered.

This means that after the COMMIT WORK has triggered the update task, the lock is no longer available to the dialog-task transaction, if it continues running. The system releases the lock at the end of V1 update-task processing, (or at the next ROLLBACK WORK.)

This is the default value, if you don't specify **_SCOPE**. You do not need to use DEQUEUE for locks created with this **_SCOPE** value. However, if **_SCOPE = 2**, and you call no update-task functions, the update task is not triggered and the system will not release your locks.

Release at ROLLBACK WORK: The system releases locks with **_SCOPE=2** if the rollback occurs before the commit. After the commit, the lock is held til the end of update task processing.

- 3 Use this value for creating locks that are
 - used in the dialog task until the update task is triggered
 - used by both dialog task and update task, after the update task is triggered.
 That is, your dialog-task program continues using the lock even while the update task function is running. In this case, the lock is "owned" by both the dialog transaction and the update-task function.

The system releases the lock sometime after the end of V1 update-task processing. However, you should release the locks explicitly yourself (with the appropriate DEQUEUE function) to be sure they are freed as soon as possible.

Release at ROLLBACK WORK: Once the update task has been triggered, locks with **_SCOPE=3** have two separate owners. To delete the lock, it must be deleted on both the dialog- and update-task sides. If a rollback occurs before the commit, the lock is freed on the update-task side, but is still held on the dialog-task side. If the rollback occurs after the commit, the system does not release the

lock on either side. In this case, you must use the DEQUEUE function explicitly to release the lock explicitly on the dialog side. The update-task side will release the lock on that side automatically.

If the user exits from the transaction before it is finished (for example with “/n”) or the program terminates abnormally, the system releases all locks. Also, V2 functions cannot inherit any locks created in the dialog task.

- `_WAIT` (ENQUEUE only)

This EXPORTING parameter tells whether ENQUEUE should wait, if the objects to be locked are already locked by another user. Set `_WAIT` to ‘X’ if your program is willing to wait. In this case, the system tries to lock the objects at repeated intervals for up to a specified length of time. If these attempts also fail, the ENQUEUE raises the `FOREIGN_LOCK` exception.

Set `_WAIT` to any other value if your program doesn’t want to wait. In this case, ENQUEUE raises the `FOREIGN_LOCK` exception and sets the system field `SY-MSGV1` to the name of the user who already owns the lock.

ENQUEUE Exceptions

After calling the ENQUEUE function module, you should check that the object has actually been locked in the program. The following exceptions are defined in the function module:

- `FOREIGN_LOCK`
Object has already been locked by another user. The system field `SY-MSGV1` contains the name of this user.
- `SYSTEM_FAILURE`
General system error.

Example Transaction: SAP Locking

For a demonstration of how to use lock objects to lock a table, see transaction TZ90. This transaction lets the user request a given flight (in screen 100) and display or update it (in screen 200). A request to update the flight locks the table; a request to display does not.

Screen Flow Logic

The screen flow logic (PAI only) for screen 200 in transaction TZ90 looks as follows.

```
*-----*
*   Screen 200: Flow Logic                               *
*-&-----*
PROCESS BEFORE OUTPUT.
  MODULE STATUS_0200.
  MODULE MODIFY_SCREEN.
*
PROCESS AFTER INPUT.
  MODULE EXIT_0200 AT EXIT-COMMAND
  '<...Relevant field checks...>'
  MODULE USER_COMMAND_0200.
```

ABAP/4 Code

The main PAI processing for screen 100 takes user input and sets things up to perform the requested operation (*Change* or *Display*). If *Change* was requested, the program locks the relevant database object by calling the related ENQUEUE function.

```

*-----*
*      Module  USER_COMMAND_0100  INPUT      *
*-----*
MODULE USER_COMMAND_0100 INPUT.
  CASE OK_CODE.
    WHEN 'SHOW'....
    WHEN 'CHNG'.
  *    <...Authority-check and other code...>
    CALL FUNCTION 'ENQUEUE_ESFLIGHT'
      EXPORTING
        MANDT      = SY-MANDT
        CARRID     = SPFLI-CARRID
        CONNID     = SPFLI-CONNID
      EXCEPTIONS
        FOREIGN_LOCK = 1
        SYSTEM_FAILURE = 2
        OTHERS       = 3.
    IF SY-SUBRC NE 0.
      MESSAGE ID SY-MSGID
              TYPE 'E'
              NUMBER SY-MSGNO
              WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
    ENDIF.
    MODE = CON_CHANGE.
    CLEAR OK_CODE.
    SET SCREEN 200.
    OLD_SPFLI = SPFLI.
  ENDCASE.
ENDMODULE.

```

The PAI processing for screen 200 unlocks the object again whenever the user tries to exit from the *Change* operation. The program uses module EXIT_0200 to trap exit function-codes (EXIT, BACK, CANC). For *Change*, only one exit function is relevant (CANC), and it calls a FORM routine for releasing the lock on the flight.

```

*&-----*
*&      Module  EXIT  INPUT                  *
*&-----*
MODULE EXIT_0200 INPUT.
  IF MODE = CON_CHANGE.
    CASE OK_CODE.
      WHEN 'CANC'.
        CLEAR OK_CODE.
        PERFORM UNLOCK_FLIGHT.
        LEAVE TO SCREEN 100.
    ENDCASE.
  ELSE. "mode = con_show
    CASE OK_CODE.
      WHEN 'CANC'...
      WHEN 'EXIT'...
      WHEN 'BACK'...
    ENDCASE.
    CLEAR OK_CODE.
  ENDIF.
ENDMODULE.

```

The FORM routine UNLOCK_FLIGHT calls the DEQUEUE function for the given lock object.

```
*&-----*
*&      Form  UNLOCK_FLIGHT
*&-----*
FORM UNLOCK_FLIGHT.
  CALL FUNCTION 'DEQUEUE_ESFLIGHT'
    EXPORTING
      MANDT      = SY-MANDT
      CARRID     = SPFLI-CARRID
      CONNID     = SPFLI-CONNID
    EXCEPTIONS
      OTHERS     = 1.
  SET SCREEN 100.
ENDFORM.
```

The other exit functions for screen 200 (BACK and EXIT) are handled in the main PAI module for screen 200. This allows the program to check whether the user is exiting without saving data. If so, the PROMPT_AND_SAVE routine reminds him, and performs a save if desired. Subsequently, the flight can be unlocked by calling UNLOCK_FLIGHT.


```
*-----*
*      Module  USER_COMMAND_0200  INPUT
*-----*
MODULE USER_COMMAND_0200 INPUT.
  CASE OK_CODE.
    WHEN 'SAVE'....
    WHEN 'EXIT'.
      CLEAR OK_CODE.
      IF OLD_SPFLI NE SPFLI.
        PERFORM PROMPT_AND_SAVE.
      ENDIF.
      PERFORM UNLOCK_FLIGHT.
      LEAVE TO SCREEN 0.
    WHEN 'BACK'.
      CLEAR OK_CODE.
      IF OLD_SPFLI NE SPFLI.
        PERFORM PROMPT_AND_SAVE.
      ENDIF.
      PERFORM UNLOCK_FLIGHT.
      LEAVE TO SCREEN 100.
  ENDCASE.
ENDMODULE.
```