

---

# Enable remote database access and parallel transaction processing using secondary database connections in your ABAP programs

by Jürgen G. Kissner



**Jürgen G. Kissner, Ph.D.**  
Developer, SAP AG

*Jürgen G. Kissner received his doctorate in theoretical physics at the University of Manchester (UK). In 1996, he joined the SAP Server Technology development team as a member of the Database Interface group where he worked on high availability and the integration of parallel database systems with SAP systems. Jürgen has been responsible for the connection and transaction handling aspects of the SAP NetWeaver Application Server, including multiple database connections in a heterogeneous database landscape. Currently he is focused on the Java persistence area. You may reach him at [juergen.kissner@sap.com](mailto:juergen.kissner@sap.com).*

All SAP solutions to date store the vast majority of their information in databases. More specifically, application data, customized and configuration data, and system data (for example, ABAP programs) reside in a single database.<sup>1</sup> Transferring data to and from a database requires a database connection. Applications that run on the SAP NetWeaver Application Server (SAP NetWeaver AS) 2004 and higher do not need to worry about this database connection,<sup>2</sup> because it is already provided by the server infrastructure. During the startup of the SAP system, each work process automatically opens a database connection, and by default, the system routes all SQL accesses (for example, Open SQL for ABAP, EXEC SQL and ADBC<sup>3</sup> calls) through these connections to the database. The connections are therefore called *default connections* (see **Figure 1** on the next page).

Many ABAP programmers are unaware, however, that they can create additional database connections (called *secondary connections*) for use in their ABAP programs to supplement these default connections.<sup>4</sup> This secondary database connection feature is an extension of the Open SQL, EXEC SQL, and ADBC functionalities. Why would you want to use it? There are two key scenarios for which having a secondary connection requires less programming and eliminates runtime overhead, making accessing data easier and more efficient. These scenarios are:

- **Remote database access:** You want to access a remote database other than your SAP system database, for example, an arbitrary

<sup>1</sup> ABAP and Java use separate schemas of the same database.

<sup>2</sup> The same is true for all applications back to SAP R/3, but for this article, the focus is on SAP NetWeaver 2004 and higher.

<sup>3</sup> ADBC stands for ABAP Database Connectivity.

<sup>4</sup> This functionality was initially released with SAP R/3 4.0, but it has been significantly enhanced with SAP NetWeaver AS 2004 and 7.0 (formerly 2004s). See Appendix A for a detailed release history.

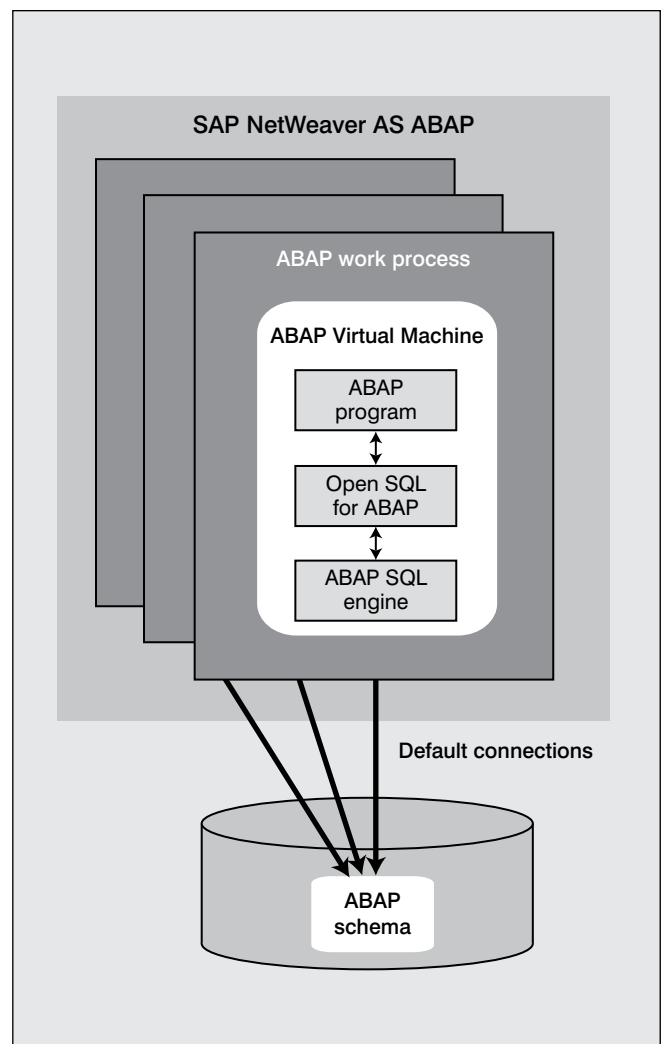
database<sup>5</sup> not associated with any SAP system. Imagine that you want to transfer data from that external database to your SAP system. Prior to SAP R/3 4.0, you could not accomplish this task by simply writing an ABAP program within your SAP system. Instead, you had to use or write your own migration tool in Java, C/C++, Perl, or another programming language that would extract the data in its own proprietary way and send it to the SAP system through some of its external interfaces, such as the Java Connector (JCo), remote function calls (RFCs), or Web services. Using a secondary connection is more convenient because you do not need to leave your SAP environment. As a result of these advantages, it is, for example, used for high-volume data transfer to populate the SAP NetWeaver Business Intelligence (BI) InfoCubes.

### Note!

At first glance, you might think that remote function calls (RFCs) can be used to trigger parallel or nested transactions. Unfortunately, this is not true. At the very moment a program submits a synchronous RFC, its present transaction is terminated (that is, committed), so your business logic cannot be rolled back anymore. Also, for transactional or asynchronous RFCs (tRFCs or aRFCs, respectively), you do not have any influence on the commit or rollback. If your parent transaction succeeds, the system submits (and executes) the RFCs, and if it fails, the system cannot submit or execute the RFCs. For more information on RFCs, see the *SAP Professional Journal* article, “Master the five remote function call (RFC) types in ABAP: Part 1 — A comprehensive guide for SAP programmers and administrators,” by Masoud Aghadavoodi Jolfaei and Eduard Neuwirt (September/October 2006).

### Note!

Each work process creates and retains its own database connection. If a work process cannot connect to the database at startup, it stops automatically and remains in the “stopped” status. So at startup of the application server, the total number of database connections equals the total number of running work processes.



**Figure 1** SAP NetWeaver AS database access architecture

<sup>5</sup> The database must be an SAP-supported database, such as MaxDB, Oracle, Microsoft (MS) SQL Server, and DB2 for z/OS, i5/OS, and Unix/NT.

- **Trigger a second database transaction:**

You need to trigger a second, parallel database transaction to commit or rollback some database operations independently from other database modifications. For example, you need to commit log data separately from your regular application database operations, or you need to increment sequence counters. You can also use secondary connections in situations in which the SAP runtime system does not allow commits of the default connection, such as an “update task.”

In short, knowing how to program with secondary database connections is a valuable tool to have in your ABAP toolbox.

This article will show you how to use SAP NetWeaver 2004’s secondary connection functionality. It is aimed primarily at experienced ABAP developers with at least some basic ABAP object-oriented (OO) knowledge. After a brief description of the system prerequisites and configuration steps you need to perform, we’ll step through the three APIs — Open SQL, ADBC, and EXEC SQL — available so

that you can access more than just the default database connection. Then, I will provide some tips and tricks to remember when working with secondary connections to help you use these programming interfaces efficiently. Because you always need to keep an eye on the resources used by your system, I will introduce you to the monitoring of secondary connections.

Before we begin, I want to use an example that illustrates how simple and straightforward secondary connections can be.

## A quick example

Let’s review a simple example, shown in **Figure 2**, to demonstrate the ease of using secondary connections.<sup>6</sup> This example inserts a new record into table SCARR

<sup>6</sup> You may have come across the table SCARR several times in the past. It is one of the tables provided by the SAP IDES airlines sample application.

```
DATA scarr_wa TYPE scarr.
DATA con_name TYPE dbcon-con_name.

* Defines the name of a secondary connection.
con_name = 'R/3*SPJ'.

* Sets the attributes of the work area to their desired values
scarr_wa-carrid   = 'MF'.
scarr_wa-carrname = 'Multi Flyers'.
scarr_wa-currcode = 'EUR'.
scarr_wa-url      = 'http://www.multifly.com'.

* Inserts the work area scar_wa using the secondary
* connection 'R/3*SPJ'.
INSERT INTO scarr CONNECTION (con_name) VALUES scarr_wa.

* Commits the changes on connection 'R/3*SPJ'.
* Note: the default connection is not affected by this commit.
COMMIT CONNECTION (con_name).
```

**Figure 2** Specifying a secondary connection to the default database

using a secondary connection. Because of the special choice for the connection name R/3\*SPJ, you can run this example without any prerequisite system configuration. As explained later, this secondary connection accesses the SAP database (schema).

As you can see, the INSERT statement and COMMIT statement contain the additional CONNECTION (con\_name) directive that tells the system which connection (R/3\*SPJ) to use — a secondary connection vs. the default connection. Therefore, as you can probably guess, the INSERT and COMMIT statements are executed on that secondary connection. The first database access, which is the INSERT statement in the example, triggers the opening of the physical connection.

If you were to step through the execution of this program in debug mode, you would see your secondary connection in action when you run the report DBCONINFO in parallel from a different screen. You may have noticed that you can also achieve the insertion of the new record through the default connection. That is true — the benefit of the example would only be visible if it contained both the default connection and the secondary connection.

You can commit the changes on one connection and rollback the changes on the other.

Naturally at this point, you probably have a lot of questions, but I am sure that you will gain a greater understanding of this example over the course of the article. Before we continue, however, take a moment to read the sidebar on this page that describes the cases in which secondary connections cannot be used.

## Configuring the prerequisites for using secondary connections

Before you can use the secondary connection functionality on your SAP system, you have to create the software environment needed to enable it and to provide your system with the necessary information about the target database, such as the host, port, user name, and password. To address these prerequisites, you need to complete two steps:

1. Verify that the necessary database libraries are installed on your SAP system.

---

### Limitations of secondary connections

Although secondary connections are a very powerful technique to use, there are a few limitations you need to know. Calls on secondary connections:

- Cannot access pool and cluster tables
- Bypass the table buffer, which means when buffered tables are modified, their buffered contents on the application servers are neither updated nor invalidated
- Do not always recognize uncommitted changes on the default connection, depending on the transaction isolation level of the database system

Furthermore, you can only run a limited number of parallel transactions. Every running task for each work process cannot have more than 16 open (uncommitted) transactions at any point of time. Because we are talking about a single SAP logical unit of work (LUW), rest assured that you'll never run into these limitations unless you're accessing more than 16 databases at the same time without committing any of these connections.

---

2. Define the new secondary database connection(s) in the SAP table DBCON.

Let's look at each of these more closely.

## Step 1: Verify that the necessary database libraries are installed on your SAP system

You can only access database systems supported by SAP, because the SAP NetWeaver AS ABAP has to send SQL statements to the database using a vendor-specific call-level interface (CLI) and a vendor-specific SQL dialect. Opening and closing connections are also part of this interface. As a result, the ABAP engine of the SAP NetWeaver AS uses a different database access layer for each supported relational database management system (RDBMS) — in the form of the Database Shared Library (DBSL). If the remote RDBMS that you want to access happens to be the same RDBMS that exists on your SAP system, the required software is already installed, so your administrator does not need to install any additional software.

The secondary database management system, however, may differ from the database management system of your SAP system (for example, the secondary DBMS is Oracle vs. the MaxDB DBMS of your SAP system). In this case, you will need to install the additional SAP shared library (DBSL), as well as the client software provided by the database manufacturer.<sup>7</sup> If either the required DBSL or the RDBMS client software is missing, you will not even be able to connect to the desired target database.

Next, you need to define a secondary connection in the table DBCON. This table provides all the information necessary to connect to a remote database, enabling a remote connection without your ABAP programs needing to supply a complete set of information. These programs can use a simple logical name that represents the connection information.

<sup>7</sup> See Appendix A for information on installing the additional software as described in the porting-specific SAP Notes for every supported database system.

## Step 2: Define the new secondary database connection(s) in table DBCON

With the correct libraries installed, you now need to create an entry in the table DBCON that describes the databases to which you want to connect. Table DBCON must contain the data that is necessary to establish a secondary connection. Each new entry you create has to contain an arbitrary connection name that serves as the key, for example, "ABC," and the information required for connecting to the external database, such as the user name, password, database host, and database name. You enter this information using transaction DBCO.

### Note!

Transaction DBACOCKPIT<sup>8</sup> will be enhanced to replace transaction DBCO with the next major SAP release (the timing and name of the release have yet to be announced).

Each record within DBCON represents a connection configuration that can be used by any ABAP program using the connection name (that is, the key attribute CON\_NAME). So if you have two different remote database systems that you want to access, table DBCON will have two records, one for each database. **Figure 3** (on the next page) lists and explains the attributes of table DBCON.

### Caution!

Do not insert or change entries in table DBCON directly; instead, use transaction DBCO (or DBACOCKPIT).

<sup>8</sup> Recently, this enhancement has been ported to SAP NetWeaver 7.0 SP 12, where the new functionality can now be used.

## The special connection name R/3\*

In case you want to open additional connections to the SAP system database, you do not need to worry about table DBCON if you use the prefix “R/3\*” with a connection name, for example “R/3\*SPJ.” The system

will automatically apply the same logon information used for the default connection. This way you don’t have to maintain the connection configuration. In other words, all changes to the connection parameters of the default connection are automatically taken into account when you open an R/3\* connection.

Attribute*	Description	Explanation
CON_NAME	Logical name for a database connection	This is the name to which you refer when using secondary connections.
DBMS	Database system	This is the database type to which the connection is set up. Valid database types are shown through the F4 help when you create a new entry.
USER_NAME	Database user	This is the database user for whom the connection is set up.
PASSWORD	Password of the database user	The password will be kept in SAP secure storage in future major releases, after which this table attribute (column) will no longer be used.
CON_ENV	Database-specific information for a database connection	This field contains the technical information that the porting wrapper of the database client software uses to open the database connection. The exact information required, which you can find in the porting-specific OSS Notes, depends on the type of database used. In most cases, you need the database name, the physical host of the database, and a port number.
DB_RECO (optional)	Approach for dealing with a broken connection	Do not change the default value, a blank space. If the flag is set to “X,” a connection interruption is treated like an interruption of the default connection; that is, the server does not accept further requests until the connection is reestablished.
MAX_CONNECTIONS (optional)	Maximum number of connections	You can define the maximum number of simultaneous open transactions (i.e., active database connections) for the logical connection name (per work process). The system rejects connection requests (CONNECT) that exceed that number. You will find this optional parameter useful for limiting the number of connections during development. (You usually know exactly how many connections to this name are necessary for the purpose of your program.) In the vast majority of cases, a program requires only a single secondary connection, so additional connections are not needed. You will then want to set MAX_CONNECTIONS to 1 to prevent unnecessary connections.
OPT_CONNECTIONS** (optional)	Optimum number of connections	You can define the optimal number of database connections. If connections exceed this number in an SAP transaction, the work process automatically ensures that they are closed again after completion of the transactions. This corresponds to an automatic DISCONNECT.
<p>* For all attributes of table DBCON, please refer to the ABAP data dictionary view of table DBCON (transaction SE11). Each column is represented by its own data element with associated documentation.</p> <p>** For releases higher than 6.40, this parameter is obsolete, so you should not use it.</p>		

**Figure 3** Attributes of table DBCON



## Programming secondary connections

Once you have configured a secondary connection, you can start using your secondary connection through any one of three programming interfaces: Open SQL, ADBC, and EXEC SQL.

### *Note!*

Often you do not really have the choice between the interfaces because you are extending existing coding. If everything has been coded using EXEC SQL, you probably should not extend that coding with ADBC calls. However, if you are free to choose the API, it is simply a matter of taste. In case you need an independent transaction to the SAP database, Open SQL is perfect because all tables are already defined in the ABAP dictionary. If you need to access a remote database, it might be too tedious to create the necessary ABAP dictionary tables for Open SQL access. In that case, consider using ADBC or EXEC SQL. For those of you who do not like object-oriented programming, feel free to stay away from ADBC.

We're going to take a two-fold look at secondary connections: First, we'll take into account that a secondary connection is a physical resource that can be created, released (handed back to a pool), and also destroyed. That means we'll look at the lifecycle of a connection — with OPEN and CLOSE being key parts of that lifecycle. We'll consider secondary connections as valuable resources to be used as efficiently as possible. Another aspect of secondary connections is the new world of separate transactions. You'll learn how to use these (parallel) transactions and how to switch between them. Let's start with the simplest one: Open SQL.

### *Note!*

You access a secondary connection using Open SQL, EXEC SQL, or ADBC by specifying its connection name. For a connection to be valid, there has to be a corresponding record in table DBCON (the connection name is the value of the key attribute CON\_NAME).

## Programming secondary connections using Open SQL

In Open SQL, you use the CONNECTION addition to execute Open SQL commands, such as SELECT, INSERT, DELETE, UPDATE, and MODIFY, using the following syntax:

```
... CONNECTION {con|( con_syntax )}
```

With that simple addition to your statement, you execute your Open SQL commands on a specified secondary database connection rather than on the default database connection. You specify the database connection with the connection name either statically with con or dynamically as the content of con\_syntax, where the field con\_syntax has type C or STRING.

You specify the CONNECTION addition immediately after the name of the database table,<sup>9</sup> as shown in **Figure 4** (on the next page). Using an Open SQL command on a secondary connection is only possible if the table definition in the database accessed by the connection matches the ABAP data dictionary definition.

If you do not specify the CONNECTION addition (R/3\*SPJ in the example), or your connection addition is CONNECTION DEFAULT, your statement is sent through the default connection.

<sup>9</sup> If you switch off automatic client handling, which is achieved through the CLIENT SPECIFIED addition, then the CONNECTION addition still follows the CLIENT SPECIFIED addition.

**Caution!**

For every database table accessed by Open SQL, there must also be a database table with the same name and identical type in the ABAP data dictionary on the local SAP system. Open SQL assumes that the type information of these remote database tables corresponds exactly to that of the local database table definition.

This prerequisite is essential for the correct interpretation of the database contents, for instance, their conversion with regards to the ABAP type of the target fields.

Neglecting these prerequisites can lead to erroneous data or runtime errors when your SAP system is trying to read and write data to/from the database. Because the ABAP runtime system cannot ensure the consistency of the type descriptions between the local dictionary description and remote databases, whoever maintains the application needs to be responsible for guaranteeing consistency.

For example, assume that you want to read the table REMOTE\_TABLE, which contains a single character column REMOTE\_KEY with a length of 10. Before you can access this table, you need to create the table REMOTE\_TABLE in your local data dictionary (transaction SE11) with the correct structural information. That is, the new table must contain a single character column REMOTE\_KEY with a length of 10.

**Opening a connection with Open SQL**

In Open SQL, you do not open a secondary connection explicitly, so you do not have a CONNECT command. The ABAP runtime opens a corresponding connection when the first Open SQL command requires a specific secondary database connection. All commands with the same CONNECTION addition use the same database connection and form a common database transaction.<sup>10</sup>

**The lifecycle of a database connection in Open SQL**

In Open SQL, database connections are automatically managed by the runtime system; you can neither open nor close them (see the sidebar on page 51 for more information on closing connections). If a transaction is saved on a database connection (COMMIT) or rolled back (ROLLBACK), then the runtime system can reuse the connection. The system automatically closes connections that are idle for a long time.

**Transactions**

A transaction on a secondary connection is committed by:

```
COMMIT CONNECTION con.
```

<sup>10</sup> This is only true as long as the commands are executed in the context of the same Internal Mode (a SUBMIT command, for example, triggers a new Internal Mode). You will find an explanation for this behavior in Appendix B.

```
DATA: con_name TYPE dbcon-con_name.
DATA: scarr_tab TYPE TABLE OF scarr.
```

```
con_name = 'R/3*SPJ'.
```

```
SELECT * FROM scarr CONNECTION (con_name) INTO TABLE scarr_tab.
SELECT * FROM scarr CONNECTION R/3*SPJ INTO TABLE scarr_tab.
```

**Figure 4** Specifying the secondary connection for the table SCARR



**Note!**

Open a connection only when needed and release it with COMMIT or ROLLBACK as soon as possible (to return it to the connection pool so that it can be reused again). Maintaining a connection takes a considerable effort of resources, for the database client (the application server) and especially for the database server.

The Open SQL COMMIT command is not executed on the default connection<sup>11</sup> but on the database connection specified by con. Here con is the name of the database connection that you defined in the table DBCON in the column CON\_NAME. You can also specify the database connection con dynamically in the form (source\_text) — where the source\_text field contains the name of the database connection. The source\_text field must be of type C or STRING.

On the specified secondary database connection, a database commit performs two actions. It:

- Closes all open database cursors (created by OPEN CURSOR) and releases all allocated database resources
- Releases all database locks

Most ABAP programmers are familiar with the following Open SQL COMMIT command, which should not be confused with the one above:

```
COMMIT WORK.
```

In the old days when there was only the default connection, COMMIT WORK was used to commit the default connection and release all resources. With multiple connections, the COMMIT WORK

still releases all resources (including cursors) and locks, but in contrast to the COMMIT CONNECTION con statement, *all* transactions are committed on *all* connections.

Remember that you cannot use Open SQL COMMIT WORK in all programming contexts. The most important exceptions are:

- COMMIT WORK is not possible in a FORM routine that is called using PERFORM ... ON COMMIT.
- COMMIT WORK is not allowed in an update task.

**Note!**

The COMMIT CONNECTION DEFAULT statement, unlike COMMIT WORK, executes a database commit on the default connection only. Nevertheless, you cannot execute a COMMIT CONNECTION DEFAULT in the special circumstances described in this section, such as in an update task and for FORM routines that have been called by the PERFORM ... ON COMMIT statement.

A transaction on a selected connection is rolled back by:

```
ROLLBACK CONNECTION con.
```

If you want to roll back all transactions on all connections, use:

```
ROLLBACK WORK.
```

If you understand the Open SQL COMMIT WORK, you won't be surprised by its opposite, the ROLLBACK WORK. A more detailed explanation goes beyond the scope of this article; however, you will find a detailed explanation in the ABAP

<sup>11</sup> The special syntax COMMIT CONNECTION DEFAULT allows you to commit the default connection.

Development Workbench documentation, including restrictions on its usage.

## Programming secondary connections using ADBC

Another way in which you can access a secondary connection is using ABAP Database Connectivity (ADBC).<sup>12</sup> With ADBC, an instance of the class

cl\_sql\_connection represents a database connection. This class has a class method get\_connection( ), which takes a logical connection name<sup>13</sup> as its argument and tries to open a connection to the database identified by this connection name. If the connection has been established successfully, the get\_connection method implicitly creates a connection object (instance of class cl\_sql\_connection) and returns a reference to it as its result. **Figure 5** illustrates an example of how to best use ADBC.

<sup>12</sup> For more information on ADBC, see the article, “Maximize the SQL functionality of your database with ABAP Database Connectivity (ADBC),” by Thomas Raupp and Tobias Wenner (*SAP Professional Journal*, September/October 2007).

<sup>13</sup> This logical name has to match a CON\_NAME attribute of table DBCON.

DATA:

```
* Variable that will point to the default connection
def_con_ref TYPE REF TO cl_sql_connection,

* Variable that will point to a secondary connection
abc_con_ref TYPE REF TO cl_sql_connection,

* Variable for the name of a secondary connection
con_name    TYPE dbcon-con_name.

* Creates a default connection object
def_con_ref = cl_sql_connection=>get_connection( ).
* Now def_con_ref can be used to send SQL statements to the SAP database
...

* Opens a database connection 'ABC' that has to be
* defined in table DBCON
con_name = 'ABC'.
abc_con_ref = cl_sql_connection=>get_connection( con_name ).
* Now abc_con_ref can be used to send SQL statements through the
* database connection defined by the name 'ABC'.
...
```

**Figure 5** Example of ADBC for how to open a secondary connection to ABC

You can use the instance methods `create_statement()` and `prepare_statement()` of class `cl_sql_connection` to create statement objects for the connection object. The system executes all SQL commands with such a statement object on the database connection represented by the connection object that created the statement object. An open database connection is closed by calling method `close()`:

```
* Closes the connection 'ABC'
abc_con_ref->close( ).
```

It is not possible to execute SQL commands on a closed connection, so all statement objects created in the context of this connection become invalid (see the sidebar on page 51 for more information on closing connections).

Calling `close()` for a default connection object has no effect because the default connection must remain open until the work process terminates. (See the Raupp-Wenner article for more examples and details.)

### Transactions

For a single connection object, there is only one transaction active at a time. A transaction is started implicitly after the database connection has been opened. A transaction is finished by calling one of the instance methods `commit()` or `rollback()` of the class `cl_sql_connection` for a connection object:

```
abc_con_ref->commit( ).
abc_con_ref->rollback( ).
```

To commit or rollback the current transaction on the default connection, a default connection object is required and has to be obtained explicitly. For example, **Figure 6** shows the lines of code needed to commit the default connection.

## Programming secondary connections using EXEC SQL

Using EXEC SQL with secondary connections has turned out to be rather difficult in practice. There are advantages of EXEC SQL in general, but in combination with secondary connections, EXEC SQL has the problem of the global switch due to the mechanism used to define the “active” connection via an EXEC SQL statement. The global switch changes the programming context in such a way that the system executes all subsequent EXEC SQL calls on the connection chosen by the switch. Consequently, it is called the “active” connection. If your program calls another program that makes use of EXEC SQL, then this call is also run through the chosen connection, which may or may not be intended.

Unless you want to make use of secondary connections in Basis 4.0B or 4.6D, where ADBC is not available, you should consider avoiding EXEC SQL in favor of ADBC.<sup>14</sup>

<sup>14</sup> When you run static native SQL statements solely on the default connection, there is nothing wrong with using EXEC SQL.

```
DATA: def_con_ref TYPE REF TO cl_sql_connection.

* Creates a default connection object
def_con_ref = cl_sql_connection=>get_connection( ).

* Commits the default connection
def_con_ref->commit( ).
```

**Figure 6** Committing the default connection

## Opening a database connection

As with ADBC, EXEC SQL requires you to open the secondary connection explicitly, as shown in **Figure 7**.

The EXEC SQL CONNECT opens the database connection to <con\_name>, where <con\_name> refers to the logical connection name that has been defined in table DBCON to configure a database connection. <con\_name> can either be a string literal or a variable, which is then indicated by :<var>.

You can use the connection identifier <alias\_name> to open more than one database connection with the same configuration <con\_name>. An alias name is required if you want to run several independent transactions with the same <con\_name> in parallel. By default, if not explicitly specified, the alias name is set to the connection name, that is, <alias\_name> = <con\_name>.

If you have not configured the connection name <con\_name> in table DBCON, an ABAP runtime error terminates the running ABAP code. Even if you have configured the connection, however, opening it may still cause an error. For example, an error may result if the database that you want to access has already been shut down. This type of CONNECT error (database error) does not cause an ABAP runtime error (with short dump) but is reported via SY-SUBRC = 4.

After a successful call of CONNECT TO <con\_name>, which is indicated by SY-SUBRC = 0, the newly opened connection is the “active” connection. This means that the system routes all subsequent EXEC SQL calls to this connection until you switch to another connection, as you will learn next. Note that this and all other EXEC SQL calls can access connections independently from the choice of the connection through OPEN SQL and/or ADBC. Also, do not expect a new physical connection to be opened for each CONNECT call: The connection pool<sup>15</sup> tries to reuse idle connections as efficiently as possible; if it can find a suitable connection (an inactive connection with a matching connection name) it will return a handle to it.

## Switching the active database connections

After having opened a secondary connection, you can switch between different database connections using the syntax shown in **Figure 8**.

You can switch to any open connection. The system then routes all subsequent EXEC SQL calls to this connection until you switch to another connection. The SET CONNECTION command does not change the transaction status of the connections,

<sup>15</sup> For information on connection pooling, see the sidebar on the following page.

```
EXEC SQL.
  CONNECT TO <con_name> [ AS <alias_name> ]
ENDEXEC.
```

**Figure 7** Opening a secondary connection using EXEC SQL

```
EXEC SQL.
  SET CONNECTION { <alias_name> | DEFAULT }
ENDEXEC.
```

**Figure 8** Switching between different database connections

## Connection pooling: When do I need to close my secondary connections?

The connection pool manages secondary connections. The APIs Open SQL, EXEC SQL, and ADBC access the same connection pool to obtain their connections. In fact, the connection pool doesn't have any information about the caller (the calling API or the Internal Mode) requesting a (secondary) connection.

If you use Open SQL, you delegate the lifecycle management completely to the runtime system — you do not even open or close a connection. ADBC and EXEC SQL allow you to open a connection. However, this does not necessarily mean that the system opens a new physical connection. The connection pool can hand out an unused connection of the requested type. A work process can have a maximum of 16 database connections. That does not imply that an application cannot use more than 16 connections at a time. If necessary, the connection pool will close committed (inactive) connections and open the requested ones. That means as long as an application does not exceed the limit of 16 simultaneously open transactions, you will not notice the limit. Note, however, on certain databases, you may not be able to reach this number because of a lack of resources: Imagine the situation of 20 application servers with 100 work processes each having 16 secondary connections to the same database. This would require 32,000 open database connections.

Typically, you won't need to care about closing connections. If a transaction is saved on a database connection (COMMIT) or rolled back (ROLLBACK), the connection pool ensures the best possible reuse of that connection. The system automatically closes connections that are idle for approximately 15 minutes.

---

### ***Note!***

Whenever a connection is released by a COMMIT or ROLLBACK, it can be reused by Open SQL, EXEC SQL, or ADBC. Even if the secondary connection has been used for queries only, it has to be committed to be (re)usable.

---

Though not necessary, closing a connection with ADBC or EXEC SQL is possible. In this case, the connection pool closes the physical connection immediately to release the acquired resources.

---

### ***Note!***

There is, however, a situation when explicit closing does not have the effect you would expect intuitively. Imagine a long-running SAP logical unit of work (LUW) that is executed on several work processes (roll-out/roll-in). If this LUW uses a single secondary connection, each work process involved in this LUW will open a physical connection (implicitly) when the secondary connection is accessed within the program logic. So every work process will have to set up a physical connection. When the close command eventually reaches the connection pool (of the present work process), only the physical connection of that work process is closed. All other work processes are left unchanged.

---

so SET CONNECTION does not automatically terminate the transaction: The transaction can be resumed when the connection is active again. If you mix Open SQL and EXEC SQL, please keep in mind that Open SQL selects its connection explicitly (with the CONNECTION addition); the EXEC SQL connection switching therefore does not affect Open SQL. If you try to switch to an <alias\_name> that does not correspond to an open connection, the ABAP runtime sets the SY-SUBRC = 4.

### *Note!*

Switching connections with SET CONNECTION does not affect Open SQL calls. Open SQL calls without a CONNECTION addition are still executed on the default connection.

### *Closing a connection*

In contrast to Open SQL, with EXEC SQL you can close connections manually. You might want to close a connection because you know that you no longer need the connection in the foreseeable future. The correct syntax is:

```
EXEC SQL.
  DISCONNECT <alias_name>
ENDEXEC.
```

This closes the connection, and a ROLLBACK discards all changes that have not been committed.<sup>16</sup> Every DISCONNECT is implicitly followed by a SET CONNECTION DEFAULT, so the default connection becomes the active connection. If <alias\_name> has not been opened in the present user context (mode), then the DISCONNECT leads to SY-SUBRC = 4.

When you close a connection, the physical

connection to the database is terminated, releasing all resources. After you close a connection, you cannot access it until a CONNECT call reopens it. Depending on the underlying database, such a CONNECT might require a considerable amount of time, and I recommend using DISCONNECT only in situations where the time span to the next database call is large. The ABAP runtime ignores any attempt to close the default connection (that is, EXEC SQL DISCONNECT DEFAULT) because the SAP system requires the default connection.

### *Getting the active database connection*

Switching back and forth between open connections is easy, but how do you know which of your connections is active? This question might be especially interesting if your code runs in a subroutine that is called by someone else. To answer this question, you call the following statement:

```
EXEC SQL.
  GET CONNECTION :<con>
ENDEXEC.
```

This statement returns the alias name of the connection which is currently active for EXEC SQL. If the default connection is active, the statement returns the value DEFAULT.

### *Accessing a remote database*

With what we've covered so far, we have the necessary ingredients to put together a little sample program that shows you what the usage of secondary connections typically looks like in EXEC SQL.

**Figure 9** shows a simple code snippet for accessing a secondary connection. It assumes a DBCON configuration for ABC.

You may have noticed that from the isolated COMMIT statement, you cannot tell on which connection the system executes the COMMIT. (This holds true for all EXEC SQL commands.) To correctly identify the connection, you need to know the last SET CONNECTION command that preceded the commit. In the example, it's easy to see, but, in

<sup>16</sup> See Appendix A for explanations of the different behaviors that occur for releases up to and including 4.5.



```

* Opens the connection to 'ABC', and switches
* between the Default Connection and 'ABC'.
DATA con_name(30) VALUE 'ABC'.

* Opens the connection to 'ABC'
EXEC SQL.
    CONNECT TO :con_name
ENDEXEC.

* Accesses connection 'ABC'
EXEC SQL.
    UPDATE ...
ENDEXEC.

* Switches to the Default Connection
EXEC SQL.
    SET CONNECTION DEFAULT
ENDEXEC.

* Accesses the Default Connection
EXEC SQL.
    UPDATE ...
ENDEXEC.

* Commits the Default Connection (but not 'ABC')
EXEC SQL.
    COMMIT WORK
ENDEXEC.

* Switches to connection 'ABC'
EXEC SQL.
    SET CONNECTION :con_name
ENDEXEC.

* Commits 'ABC'
EXEC SQL.
    COMMIT WORK
ENDEXEC.

* Closes connection 'ABC'
* An application should always COMMIT or ROLLBACK explicitly
* before disconnecting (otherwise the changes are rolled back)
EXEC SQL.
    DISCONNECT :con_name
ENDEXEC.

```

**Figure 9** Accessing a secondary connection

general, it is quite tedious to make sure that every statement is executed on the intended connection.

The situation gets even more difficult if you want to provide some generic service involving EXEC SQL on secondary connections. It is difficult because you cannot assume a specific state of the program that is calling your service. Neither can you rely on a specific active connection that the calling program might have set. On the other hand, you should be careful when setting the active connection yourself because this setting will, in turn, affect the program that called your service. If you don't code the calling program to expect any change in the active connection, returning from the routine might lead to surprising results by accessing an unexpected connection. Clearly special care has to be taken to avoid such situations. The solution is to store the active connection at the beginning of the service routine and to reset the original active connection before returning back to the caller. You can use the code shown in **Figure 10** as a program template for using secondary connections without

negatively affecting the caller's setting of the active connection.

So far you've learned how the three programming interfaces, Open SQL, EXEC SQL, and ADBC, make use of secondary connections. Until now, we have mainly talked about the properties of a single connection, but we have neglected collective effects that arise when more than a single connection (and transaction) is present. We are now going to fill that gap.

## Revisiting transactions on secondary connections

So far we have looked at COMMIT and ROLLBACK statements that act on a single connection. In fact, before secondary connections, this was the only scenario. At that time, the world was rather simple from the transactional point of view: With only the default connection, there could only be one open

```
* This function executes statements on a secondary connection.
* On return, the original connection setting is restored.
FUNCTION F_BLACK_BOX.
  DATA: previous_con(30).

  * Remember the active connection
  EXEC SQL.
    GET CONNECTION :previous_con
  ENDEXEC.

  * Tests, if connection 'ABC' has already been opened
  EXEC SQL.
    SET CONNECTION 'ABC'
  ENDEXEC.

  * Opens the connection, if it is not open yet
  IF SY-SUBRC <> 0.
    EXEC SQL.
      CONNECT TO 'ABC'
    ENDEXEC.
```

*Continues on next page*

**Figure 10** Protecting the setting of the active connection of its calling program

```

* Handles connect failures (no secondary connection)
  IF SY-SUBRC <> 0.
*   Error handling
    ENDIF.
  ENDIF.

* Execute the statements on connection 'ABC'
  EXEC SQL.
    ...
  ENDEXEC.

* Resets the active connection to the original one
  EXEC SQL.
    SET CONNECTION :previous_con
  ENDEXEC.

ENDFUNCTION.

```

**Figure 10** (continued)

transaction at a time, and a commit or rollback of that connection terminated the transaction. With secondary connections, we have an additional degree of freedom, because we can terminate a specific transaction on a selected connection or commit or rollback all transactions at (nearly) the same time. Obviously, this adds complexity. Before dealing with that, let us recall the various ways of terminating the transaction on a single connection. (You learned about them when we looked at the varieties of APIs.) A transaction on a single connection is ended by:

- Open SQL call COMMIT CONNECTION or ROLLBACK CONNECTION on the specified connection
- Native SQL COMMIT WORK or ROLLBACK WORK acting on the active database connection
- ADBC methods commit( )/rollback( ) for a connection object

In contrast to these options, there are situations and statements that affect all open connections in an SAP

logical unit of work (LUW). All transactions on *all* connections are ended quasi-simultaneously by an:

- Open SQL COMMIT WORK or ROLLBACK WORK
- Implicit commit by a screen change, specifically the statements CALL SCREEN, CALL DIALOG, CALL TRANSACTION, MESSAGE
- RFC, specifically the following statement:

```

CALL FUNCTION ... DESTINATION,
CALL FUNCTION ... STARTING NEW TASK,
CALL FUNCTION ... IN BACKGROUND TASK,
                and WAIT

```

I can summarize the second and third approaches (i.e., the implicit COMMIT and the RFC) in the following way:

- All open database transactions are completed at the latest when the application program reaches a state in which a change of work process could

occur (e.g., a rollout of the user context). When all transactions are to be terminated “simultaneously” as just described, this is actually done sequentially, with one open connection closing after the other. In this sequence, the default connection is the last connection that is committed or rolled back. However, you cannot rely on any specific order for the commit or rollback of secondary connections.

- In case one of the COMMIT calls in the sequence fails, the processing stops and returns an error (ABAP short dump). The subsequent task handler ROLLBACK does not have any effect on the already committed connections. An application has to take this scenario into account. It cannot rely on a distributed (or coupled) transaction among several database connections. Ideally, an application should use secondary connections only in cases where the application can commit or rollback transactions on these connections independently.

### **Note!**

SAP NetWeaver Application Server (AS) does not support distributed transactions and two-phase commits. This means that changes performed on different database connections cannot be subsumed under a global transaction that guarantees atomicity. Transactions on different database connections run completely independent of each other and must be committed separately.

Up to this point, we’ve only seen examples where Open SQL, EXEC SQL, and ADBC are separated cleanly. However, the question naturally arises: Can the different access modes take part in the same transaction (use the same connection simultaneously)? Up to and including SAP NetWeaver 7.0, the answer has been that only EXEC SQL and Open SQL can access the same secondary connection in one transaction.

However, the next major SAP release will add additional functionality to ADBC, making it possible for ADBC to join an open transaction on a secondary connection. A connection is automatically shared between Open SQL and EXEC SQL whenever you use the same connection name. This is not the case for ADBC. Even if you use the same connection name, you will always get an unused connection, meaning an independent transaction.

## **Tips and tricks**

You can use the following guidelines to improve the stability and performance of your programs:

- **Minimize the number of parallel transactions:** If you know that you are going to need only one secondary connection, set DBCON attribute MAX\_CONNECTIONS to 1. This will prevent you from generating unintended multiple connections and transactions. Avoid creating additional Internal Modes that use secondary connections.
- **Encapsulate all access to the database(s) in some persistency service layer:** Hide the complexity of secondary connections from business application developers. Secondary connections require special care, and your encapsulation should ensure you don’t violate the tips listed in this section. Additionally, the encapsulation should contain careful error handling (the target database might not be available or might refuse an additional connection attempt) and optional tracing to verify that the connection is used as planned and expected.
- **Do not close a connection prematurely:** If you do not know whether you are going to need the connection in the next 15 minutes, do not close the connection. The connection pool automatically closes idle connections.
- **Use COMMIT whenever possible:** Keep your transactions short. Terminating a transaction allows the connection pool to manage connections efficiently. This also applies when the connection has been used for queries (read access) only!

- **Never modify a table through several connections:** If you modify a table using the default connection, do not change the same table with your secondary connection and vice versa (there is a danger of an application deadlock).
  - **Do not modify arbitrary tables of your default schema with secondary connections:** If you plan to modify a special table of the default schema, make sure that buffering is not allowed. Add a comment to the table description pointing out that secondary connections (only) change the table. Modifications of buffered tables through secondary connections result in inconsistent buffer contents. Don't do it!
  - **Beware of application deadlocks:** Working with parallel database connections (and thus parallel transactions) can lead to lock situations involving only one work process: A program changes a database row on the first connection and tries to change the same row on a second connection. This will result in the program waiting for the lock of the first transaction, without this first transaction ever being able to continue. You can only resolve this situation by ending the work process. The dialog session timeout does this automatically for dialog processes, but it must be done manually for background jobs. You should therefore *not modify* the same table in one program on multiple database connections. From the perspective of the database, this situation is not a deadlock, so its deadlock detection does not help. For the database, there is one client (the first connection) holding a database lock, and a second client is waiting for the lock. It does not know that the application cannot continue its program execution and is doomed to wait forever.
  - **Do not rely on secondary connections without taking into account that a CONNECT call might fail:** Database connections are valuable and limited resources. The database might not be able to satisfy a CONNECT request because too many other clients (application server work processes) have already requested one or more connections.
  - **Open SQL requires a table definition in the ABAP data dictionary of the SAP NetWeaver AS:** The only purpose of the definition is to ensure the correct structure of the table (e.g., field names and field types). Until now, it was impossible to mark this definition as a “remote table structure.” If you fill the table unintentionally through the default connection, no error will be raised. The data ends up in the default schema; whereas the application expects the data on the remote database.
  - **Remember that the EXEC SQL call SET CONNECTION affects all subsequent EXEC SQL calls:** For example, in the case of a missing switch back to the default connection, some subsequent EXEC SQL calls access the wrong (the secondary) connection. Incorrect error handling leads to undesired database targets.
  - **An unintentional opening of additional secondary connections occurs if SQL statements to the secondary database are executed in a newly created Internal Mode:** ABAP does not allow passing connections from one Internal Mode to another one (see Appendix B).
  - **Adapt DBCON entries after transporting them:** After you transport DBCON entries, make sure you adapted the entries accordingly. You do not want your productive system to use the “toy” database configured in the test system.
  - **Never modify DBCON entries in a running system:** Modification does not affect open connections. (Use the report DBCONINFO to check.) You might need to restart your application server if you want to ensure that all connections are accessing a new configured database host.
- With these tips in mind, you should be able to work with and manage connections in a way that allows a stable and smooth operation of your SAP system. However, no software is perfect, so you might want to investigate what can result in a waste of resources (e.g., too many database connections). This is where monitoring becomes handy.

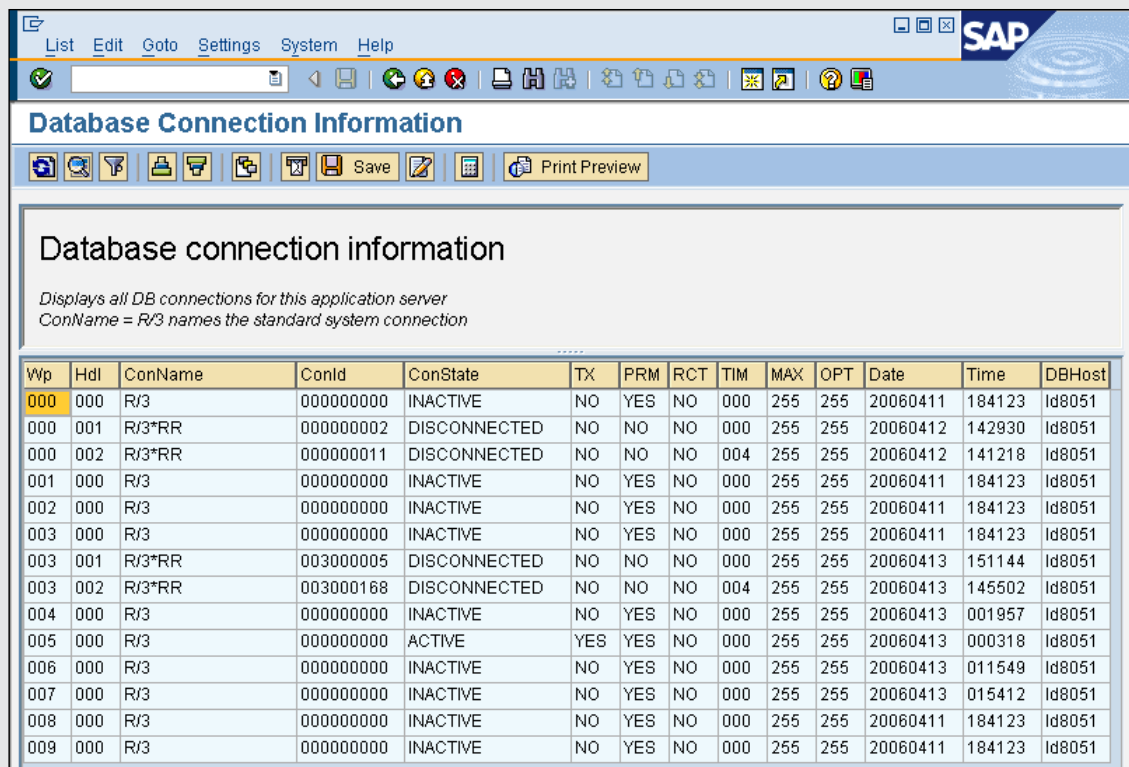
## Monitoring your secondary connections

The server infrastructure administers resources and transactional contexts for connections. There is no need for application developers or system administrators to carry out any kind of maintenance work or regular monitoring. Nevertheless, SAP provides a tool for monitoring all connections of an application server. Please keep in mind that the purpose of this tool is mainly for SAP internal analysis; therefore, some of the information will not be relevant to you as a user of the connection infrastructure. Also, the details of the status information differ between releases and might be changed without notice. That means you should not use or parse this information with mission-critical tools.

When you run report DBCONINFO in an SAP NetWeaver 7.0 system, the system provides the information shown in **Figure 11**. This information is a snapshot of your application server's connection state for all work processes. The system stores the state in the shared memory of the application server, which you may find very useful. In case a work process has stopped, or is busy or seemingly "hanging," you can still see the state of its connections. **Figure 12** lists the attributes displayed by DBCONINFO.

Using two of the work processes shown in the output of DBCONINFO, let's examine the purposes of the attributes more closely.

The work process 5 (Wp = 005) has a single database connection, the default connection



Wp	HdI	ConName	ConId	ConState	TX	PRM	RCT	TIM	MAX	OPT	Date	Time	DBHost
000	000	R/3	000000000	INACTIVE	NO	YES	NO	000	255	255	20060411	184123	Id8051
000	001	R/3*RR	000000002	DISCONNECTED	NO	NO	NO	000	255	255	20060412	142930	Id8051
000	002	R/3*RR	000000011	DISCONNECTED	NO	NO	NO	004	255	255	20060412	141218	Id8051
001	000	R/3	000000000	INACTIVE	NO	YES	NO	000	255	255	20060411	184123	Id8051
002	000	R/3	000000000	INACTIVE	NO	YES	NO	000	255	255	20060411	184123	Id8051
003	000	R/3	000000000	INACTIVE	NO	YES	NO	000	255	255	20060411	184123	Id8051
003	001	R/3*RR	003000005	DISCONNECTED	NO	NO	NO	000	255	255	20060413	151144	Id8051
003	002	R/3*RR	003000168	DISCONNECTED	NO	NO	NO	004	255	255	20060413	145502	Id8051
004	000	R/3	000000000	INACTIVE	NO	YES	NO	000	255	255	20060413	001957	Id8051
005	000	R/3	000000000	ACTIVE	YES	YES	NO	000	255	255	20060413	000318	Id8051
006	000	R/3	000000000	INACTIVE	NO	YES	NO	000	255	255	20060413	011549	Id8051
007	000	R/3	000000000	INACTIVE	NO	YES	NO	000	255	255	20060413	015412	Id8051
008	000	R/3	000000000	INACTIVE	NO	YES	NO	000	255	255	20060411	184123	Id8051
009	000	R/3	000000000	INACTIVE	NO	YES	NO	000	255	255	20060411	184123	Id8051

**Figure 11** Output of report DBCONINFO



Attribute	Explanation
Wp	Work process number (which corresponds to the number shown in the “process overview” that you get when you run transaction SM50). Each work process has its own private connections. Connections cannot be shared among work processes.
Hdl	A connection handle for internal purposes.
ConName	Logical name for a database connection. This is the name you refer to when using secondary connections. The default connection is denoted by R/3.
ConId	Connection ID for internal purposes. A unique ID identifying a connection handle. The system creates the ID when a CONNECT request enters the connection pool. It is used for the association with a physical connection in the connection pool. When a user context is rolled into a new work process, its embedded connection description holds the connection handles of its secondary connections. If one of these handles is used in the new work process, the database interface automatically reopens a physical connection. As a consequence, the same ConId appears in the new work process too. The default connection always has ConId = 0.
ConState	The state of a connection, which can be one of the following: <ul style="list-style-type: none"> <li>• STATE_CONNECTING: The work process is opening a new connection. It has submitted its CONNECT request to the database, but the connect function call has not yet returned.</li> <li>• STATE_INACTIVE: The connection is open but not used at this moment in time. This state is entered after a COMMIT or ROLLBACK. No transaction is running on this connection, and the connection can be used or reused for any new request.</li> <li>• STATE_ACTIVE: This state is entered with the first SQL statement other than COMMIT, ROLLBACK, or DISCONNECT. The connection is associated with the connection handle having the ConId displayed for this entry. The physical connection cannot be accessed by any other connection handle.</li> <li>• STATE_DETACHING: The work process is closing this connection. It has submitted a DISCONNECT request to the database, but the disconnect function call has not yet returned.</li> <li>• STATE_DISCONNECTED: The connection has been closed.</li> </ul>
TX	The transactional state of the connection: <ul style="list-style-type: none"> <li>• YES: A modification, a SELECT FOR UPDATE, or an EXEC SQL statement other than the COMMIT/ROLLBACK has been executed on the connection.</li> <li>• NO: The last SQL statement on this connection was a COMMIT, a ROLLBACK, or an Open SQL query.</li> </ul>
PRM	The value of the DBCON attribute DB_RECO.*
RCT	The “Reconnect” flag: <ul style="list-style-type: none"> <li>• YES: This connection has either been detected as unusable or triggered by the communication between work processes; it has been marked as potentially broken and needs to be checked.</li> <li>• NO: The database connection has not experienced any failure so far, or it has been reestablished.</li> </ul>
TIM	A countdown counter for closing an idle, unused connection. The counter starts with 4 and is decremented. When it reaches 0, the connection will be closed. The decrementing time interval is the rdisp/autoabap time, which has a default of 5 minutes. So the total idle time-out interval is 15-20 minutes.
MAX	The value of the DBCON attribute MAX_CONNECTIONS.*
OPT	The value of the DBCON attribute OPT_CONNECTIONS.*

**Figure 12** Attributes of the DBCONINFO report*Continues on next page*

Attribute	Explanation
Date	Date when the physical database connection has last been opened. When the current ConState is DISCONNECTED, it is the date when the connection has been closed.
Time	Time when the physical database connection has last been opened. When the current ConState is DISCONNECTED, it is the time when the connection has been closed.
DBHost	The physical database host. This information might not been present, in which case the field is left empty.
* For more information on DB_RECO, MAX_CONNECTIONS, and OPT_CONNECTIONS, see <b>Figure 3</b> on page 44.	

**Figure 12** (continued)

(ConName = R/3 and ConId = 0). ConState for the default connection is ACTIVE and TX is YES, which means that there is an uncommitted modification or EXEC SQL statement on that connection. The default connection was established shortly after midnight (00h 03m 18s) on 13.04.2006 (April 13, 2006). The database runs on the physical host ld8051.

The work process 3 (Wp = 003) has three database connections. The default connection (ConName = R/3 and ConId = 0) is not used (INACTIVE). Two secondary connections (both named R/3\*RR) with ConId 000000002 and 000000011 have been closed already. The connection with ConId = 000000011 is closed explicitly, which we can infer from the fact that TIM = 4. A bit later (17m 12s later) the connection with ConId = 00000002 is closed because of the idle timeout, which TIM = 0 indicates.

## Conclusion

Using secondary connections is *not* the standard programming model of a typical business application. There are, however, specific scenarios where a single connection (to the SAP system database only) is an undesired limitation. An example is when you need to access an external remote database or an additional independent database transaction. Under these circumstances, secondary connections are an invaluable enrichment of the programming interfaces Open SQL, EXEC SQL, and ADBC. Used with care and encapsulated in some persistency framework, secondary connections open a new dimension of robust and efficient interaction with the outside (database) world. For the first time, the technique of secondary connections makes it possible to commit or rollback transactions without the danger of unwanted side effects.

---

# Appendix A — Secondary connections functionality

Secondary connections were introduced with Basis release 4.0B. **Figure 1** shows the major improvements in secondary connections by release.

Release	Improvement
Basis release 4.0B (Kernel release 4.0B)	<ul style="list-style-type: none"><li>• Enabled secondary connections for EXEC SQL</li><li>• Provided an implicit “task handler” COMMIT/ROLLBACK, or an explicit Open SQL COMMIT WORK, to commit the default connection before any secondary connection</li><li>• Included a DISCONNECT, which is ignored when the transaction hasn’t been terminated by a COMMIT or ROLLBACK (an unsuccessful DISCONNECT can be traced in the developer trace files dev_w*)</li></ul>
Basis release 4.6D (Kernel release 4.6D)	<ul style="list-style-type: none"><li>• Made the monitoring report DBCONINFO available</li><li>• Committed the default connection after the secondary connections with a task handler COMMIT/ROLLBACK or an Open SQL COMMIT WORK</li></ul>
SAP NetWeaver 2004 (Kernel release 6.40)	<ul style="list-style-type: none"><li>• Supplied Open SQL for secondary connections</li><li>• Offered ABAP Database Connectivity (ADBC) for secondary connections</li><li>• Presented an “R/3*” connection* for accessing the default database through a secondary connection</li><li>• Caused automatic ROLLBACK of the transaction with enhanced DISCONNECT</li><li>• Included new lifecycle management of secondary connections:<ul style="list-style-type: none"><li>- The system closes unused connections after an idle timeout or when the connection pool reaches its limit of 16 connections (per work process).</li><li>- If report DBCONINFO shows the TIM column, the system supports the timeout.</li><li>- Before release 6.40, up to 32 connections could be opened in a work process; the static limit is independent of the connection status (OPENED, CLOSED).</li></ul></li></ul>
SAP NetWeaver 7.0 (Kernel release 7.00)	<ul style="list-style-type: none"><li>• Identified the active connection with EXEC SQL GET CONNECTION, making it possible to determine the active connection</li></ul>
* For more information, refer to the section, “The special connection name R/3*” in the article.	

**Figure 1** Secondary connection improvements by release

## The future of secondary connections

Future releases will include the following secondary connection functionalities:

- The easy-to-use transaction DBACOCKPIT will be available. You will use this transaction to maintain connection data. DBACOCKPIT will replace transaction DBCO.<sup>1</sup>
- ABAP Database Connectivity (ADBC) will be able to use the flag 'X' as an optional parameter of the method create\_statement() of the class cl\_sql\_connection. Before the introduction of the shareable flag, ADBC could not take part in a common transaction with Open SQL or EXEC SQL.
- The password of the database user will no longer be stored in the PASSWORD field of the DBCON table. Instead, the password will be stored in the secure storage.
- Unicode migration of connection data will occur automatically. Reentering passwords will no longer be required.

## RDBMS porting-specific SAP Notes

The SAP Notes listed in **Figure 2** describe porting-specific steps and configurations, especially for:

- The installation of the database client libraries
- The information to be entered in transaction DBCO for the maintenance of table DBCON

Contact the porting teams in case of questions with respect to these notes.

<sup>1</sup> You can find information on the new features of DBACOCKPIT in the SDN at [www.sdn.sap.com](http://www.sdn.sap.com). Logon credentials are required to access information in SDN.

RDBMS	SAP Note	Component	Title
MaxDB	955670	BC-DB-DBI	DB multiconnect with MaxDB as secondary database
DB2 for AS/400 (i5/OS)	445872	BC-DB-DB4	iSeries: DB multiconnect from Windows / LinuxPPC to iSeries
DB2 for AS/400	146624	BC-DB-DB4	AS/400: Database Multiconnect with EXEC SQL and ADBC
DB2 for OS/390 (z/OS)	160484	BC-DB-DB2	DB2/390: Database multiconnect with EXEC SQL
DB2 Universal Database for Unix/NT	200164	BC-DB-DB6	DB6: Database multiconnect with EXEC SQL
Microsoft SQL Server	178949	BC-DB-MSS	MSSQL: Database MultiConnect with EXEC SQL
Oracle	339092	BC-DB-ORA	DB MultiConnect with Oracle as secondary database

**Figure 2** Database vendor-specific notes for secondary connections

---

# Appendix B — Architecture of secondary connections

This appendix provides the background knowledge that is necessary for a deeper understanding of the mechanisms that govern the handling of secondary connections. It is intended for those who want to gain a more in-depth understanding of the system's inner workings.

To help you become more familiar with secondary connections, I have divided the content of this appendix into two topics: the components of secondary connections and the handling of secondary connections.

## Components of secondary connections

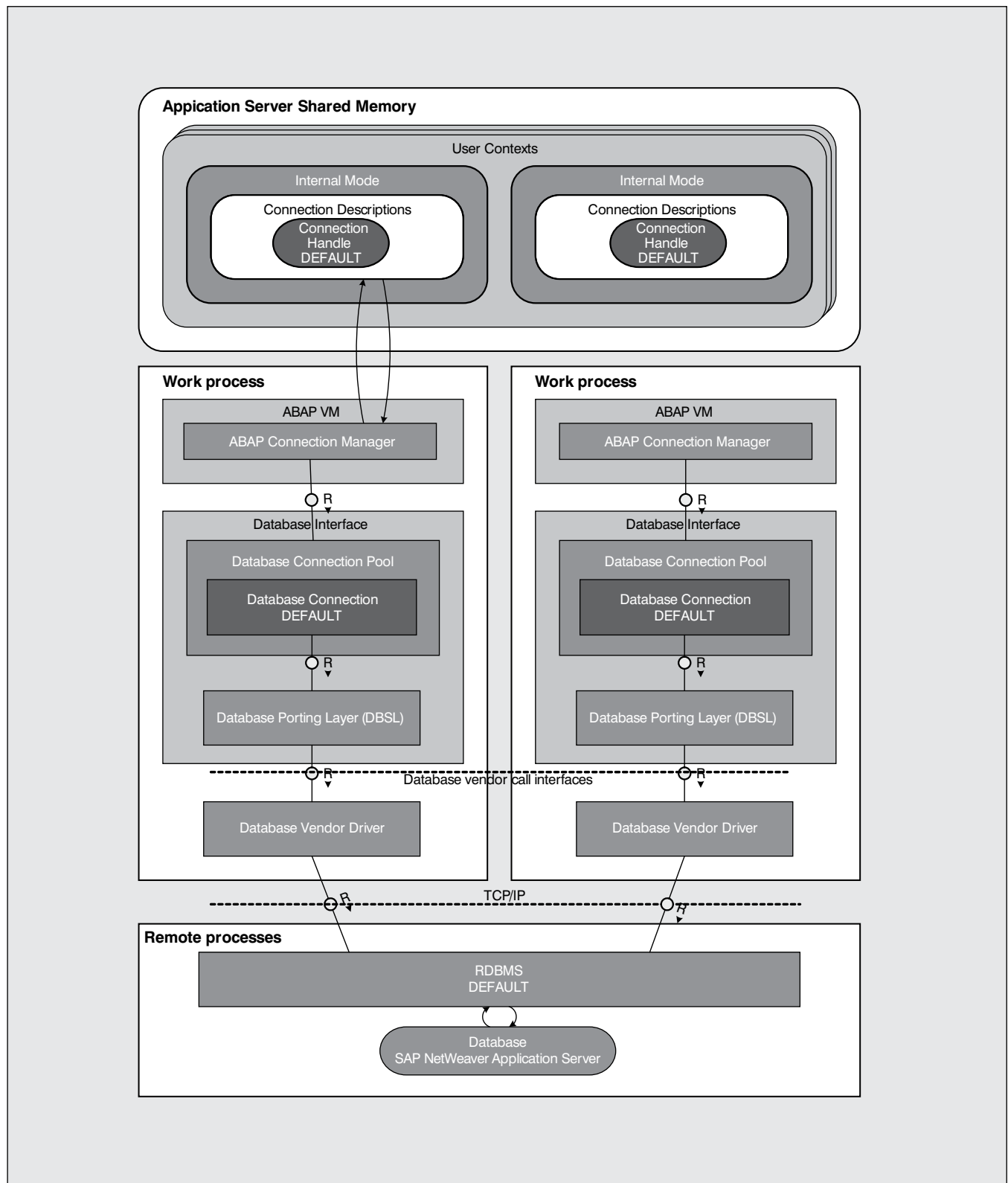
The default scenario of an SAP NetWeaver Application Server (AS) consists of work processes that each have a single connection to the database — the default connection. **Figure 1** (on the next page) shows the component architecture of the database access in this scenario.

Let's look at this figure from the top down.

Every user that logs into the SAP NetWeaver AS has an individual user context that stores all kinds of authorizations, settings, and other metadata. The user context is part of the roll area, which resides in the application server's shared memory and is not tied to any specific work process.

Whenever a work process executes a task requested by the user, his or her user context is attached to and mapped into the memory of this work process. Within an SAP logical unit of work (LUW), the system can execute the different steps of a task on different work processes. When the system changes the executing work process, the user context is detached from the previous work process and attached to the one that continues executing the task.

An important part of the user context is the “Internal Mode” (see sidebar on page 67 for more information). The Internal Mode is a program context, which means it holds all sorts of information about a running program, such as global variables, stack frames (containing local variables and classes), and program meta information. The meta information includes information about all database connections in a set of connection



**Figure 1** Default connection database access



descriptions. A connection description contains the name of the connection, an optional alias name, its status (opened or closed), and the connection handle. If there is only one connection, the default connection, then there is only one connection description.

So far, I have described a collection of information in the shared memory. Let's now turn to the work process that uses and manipulates that information. Not surprisingly, it is the ABAP Virtual Machine (VM) that interacts with the user context, particularly with the Internal Mode. The ABAP VM is responsible for executing the compiled ABAP programs. Whenever such a program submits a SQL statement, the ABAP VM routes that statement to the database. Not only is the SQL statement passed to the database interface, but the ABAP Connection Manager uses the program context, especially the connection descriptions of the active Internal Mode, to determine which connection the statement should use. It then adds the corresponding connection handle to the statement.

When the SQL statement and the connection handle reach the database interface, the database connection pool associates the handle with a physical database connection. The database connection pool knows to which supported database vendor the connection belongs. It passes the SQL statement to a vendor-specific porting wrapper (called the DBSL) provided by SAP, which forms the adapter, to the proprietary, vendor-specific call interfaces delivered by each database vendor in the form of a driver library. Finally, the vendor drivers send the statement to the database.

If there are several Internal Modes, there is an independent connection description for each Internal Mode. **Figure 2** (on the next page) depicts two Internal Modes. Both connection descriptions contain a single entry with the identical connection handle, namely the one for the default connection. When the second Internal Mode becomes active, the ABAP VM, which previously accessed the first Internal Mode, switches to the second Internal Mode and the new connection handle. The database interface maps this (default connection) handle to the same physical connection as before.

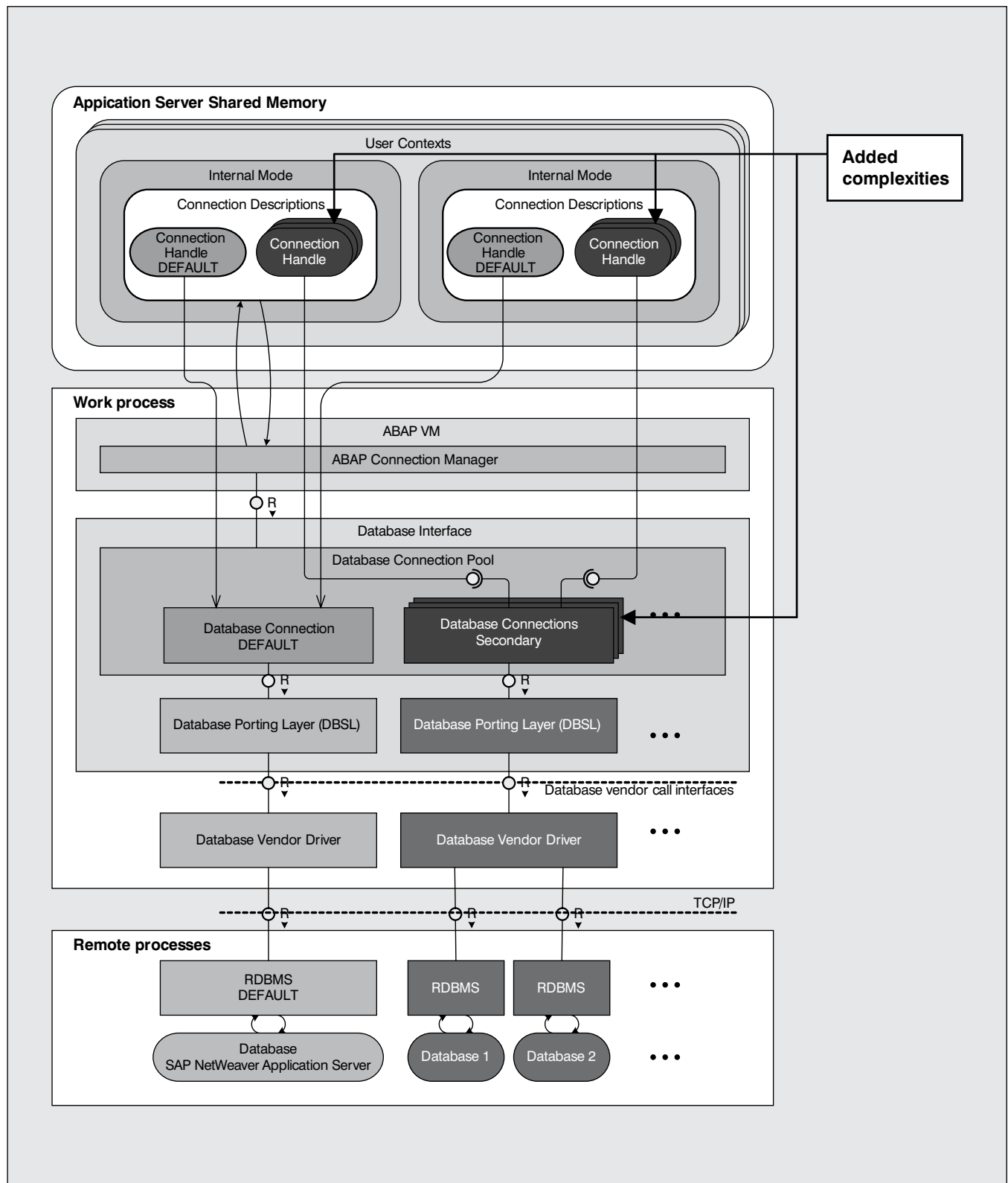
So far, we have only considered the work process on the left side of **Figure 1**. Now let's discuss the work process on the right side.

When a user changes screens, the user context might get rolled out. In other words, the user context detaches from the work process on the left and attaches itself to the second work process shown on the right. In this situation, the default connection handle is processed by the ABAP VM for the second work process and is eventually mapped by the database connection pool to the physical connection of the second work process. Work processes cannot share the same physical connection, but the mapping ensures that the default connection handles of both Internal Modes are mapped to a common physical connection, enabling the application programs that run in the context of these two Internal Modes to take part in the same transaction.

If you are wondering about the roll-out/roll-in and the effect of changing the physical connection on the transactional integrity, remember that every context roll-out is preceded by a "task-handler commit," which implicitly commits all open transactions on all connections. With just the default connection, there is only a single open transaction.

## Handling of secondary connections

With secondary connections, the architecture of the connection handling becomes more intricate, as shown in **Figure 2** (on the next page).



**Figure 2** SAP NetWeaver AS database access architecture

First of all, a secondary connection has to be opened explicitly by the ABAP VM.<sup>1</sup> When the ABAP VM requests a new secondary connection from the database interface, it passes the connection name (not the alias) as input. The database connection pool uses the connection name to identify a connection description in table DBCON. It then loads the required libraries and tries to establish the database connection. A successful connection is registered with the connection pool, and a connection handle is returned to the ABAP VM. This connection handle guarantees exclusive access to the physical connection, meaning that another connection handle cannot access this physical connection. The ABAP VM stores the handle together with the connection name and connection alias in the connection description of the active Internal Mode. Other Internal Modes do not know about this connection handle: The inactive Internal Modes neither see this physical secondary connection, nor can they access it. This applies even if the ABAP VM opens a connection in a second Internal Mode using the same connection name and alias. In that case, the ABAP VM creates a new entry in the Internal Mode's connection description and requests a new connection handle from the database interface. This situation is fundamentally different from the way the default connection handle is treated, because the default connection handle is the same for all Internal Modes. **Figure 2** shows a situation with an active Internal Mode on the left side and an inactive Internal Mode on the right.

But what happens to the association between connection handles of an inactive Internal Mode and their corresponding physical connections? The secondary connection handles in the inactive Internal Mode (shown on the right side of **Figure 2**) are still associated with their physical connections in the database connection pool! The dissociation between the ABAP connection handle and the physical connection of the database connection pool happens when the connection is committed or rolled back. As soon as the transaction is finished (COMMIT/ROLLBACK), the connection pool marks the connection as INACTIVE, which means that it can be used or reused by any request that wants to access a connection with the same name.

<sup>1</sup> This happens when the ABAP VM processes one of the three statements: an ADBC `get_connection()` call, an EXEC SQL `CONNECT TO` call, or an Open SQL statement that accesses a secondary connection the first time in the context of its Internal Mode.

## The ABAP Internal Mode

The Internal Mode is a part of the User Context that resides in the Application Server's shared memory. It is a program context, so it holds substantial information about a running program like global variables, the stack frames (containing local variables and classes), and program meta information. The part of the meta information concerning database connections is the set of Connection Descriptions. Each Internal Mode has its own private set of Connection Descriptions to ensure transaction encapsulation for secondary connections.

The life cycle of an Internal Mode is as follows: The first application program that is started opens an Internal Mode. If the active application program calls a different application program with one of the ABAP commands `CALL TRANSACTION`, `CALL DIALOG` or `SUBMIT`, the system opens another Internal Mode. In other words, another Internal Mode is pushed onto the Internal Mode stack. The called program then runs in the new Internal Mode. When the called program terminates, the system deletes the corresponding Internal Mode and continues processing in the realm of the previous Internal Mode.

If an Internal Mode finishes its lifecycle without any COMMIT/ROLLBACK on its secondary connection(s) — which is possible and permissible — then the transactions of the acquired connections are kept open. The termination of the transaction is delegated to the task handler, which is the “transaction monitor” of the SAP NetWeaver AS. The final “task handler” COMMIT/ROLLBACK terminates all open transactions on all connections. Alternatively, the transaction termination can be achieved via an ABAP COMMIT/ROLLBACK (either by Open SQL, EXEC SQL, or ADBC).