# Take full advantage of the SQL functionality of your database with ABAP Database Connectivity (ADBC)

by Thomas Raupp and Tobias Wenner

**Thomas Raupp**
Development Manager,
SAP AG

**Tobias Wenner**
Senior Developer,
SAP AG

*(full bios appear on page 114)*

ADBC? Sounds like JDBC — Java Database Connectivity — the well-known SQL programming model in the Java world. And indeed, the association between these two acronyms is intended. ADBC stands for ABAP Database Connectivity and is the name of an object-based ABAP API for programming relational database accesses, which in its class and method structure follows the JDBC diction, although — as you will see in this article — it is in no way an ABAP implementation of the JDBC interface. There are simply too many differences in the nature of these languages.

So, have ABAP developers been unhappy with the available database programming models and requested some fancier alternative — something similar to JDBC, for example? The answer is a very clear no! ABAP Open SQL is, and will remain, the SQL programming model of choice for the vast majority of application developers. The seamless and tight integration of database access via Open SQL is a major strength of the ABAP language and allows for optimal performance, platform independence, and compile-time syntax checking. So, why might ABAP developers need another SQL API?

ADBC was developed as an amendment to the ABAP Native SQL functionality provided with the EXEC SQL command. With the EXEC SQL command, developers can execute database-specific SQL statements that are not covered by the Open SQL functionality, and access database tables that are not managed by the ABAP Dictionary, which allows developers to address data created independently of SAP NetWeaver Application Server (AS). However, some deficiencies and conceptual weaknesses in the EXEC SQL command approach limit its applicability in some important cases:

- **The EXEC SQL command supports only static SQL statements.** This means that it cannot be used for a certain class of tasks requiring dynamic SQL, such as generic queries.[1] Of course, it's always possible to dynamically create and execute an ABAP program in such cases, but this is cumbersome to program and also inefficient at execution.

- **Error handling is rather inflexible.** If the execution of a SQL statement fails for some reason, the ABAP runtime system aborts the current application transaction and raises a runtime error. However, in some cases, an application program may still want control after a SQL error has occurred. For example, the program might want to evaluate the SQL code returned and react differently depending on its value.

- **Dealing with multiple database connections requires the utmost programming discipline and is fairly error-prone.** The reason is that SQL statements are executed in the context of a global connection state, rather than on behalf of dedicated connection objects that can be passed as parameters between program units. This situation invites the same kinds of pitfalls and unpleasant surprises encountered with the use of global program variables.

Because there was no clear-cut way to overcome these deficiencies with the existing EXEC SQL concept, ADBC was introduced as a new offering for ABAP programmers who need more flexibility in executing Native SQL statements. ADBC is an object-based call-level API that addresses the limitations of EXEC SQL. This API provides access to the entire SQL functionality of the underlying database system. All SQL statements can be created dynamically and executed through respective API methods. Furthermore, the ADBC API provides a clean concept for dealing with multiple database connections. Each opened database connection is associated with a connection object that you can then use to create and execute SQL statements through this connection. And finally, in case of error situations, all methods of the ADBC API throw well-defined exceptions that can be caught and handled by the caller. The programmer then has full control and can handle any error situation that arises.

This article shows ABAP programmers how to take full advantage of SQL functionality by using ADBC to reach beyond the capabilities of Open SQL. We provide details on the most relevant topics of SQL programming with ADBC by walking through the following examples:

- Establishing a connection to the database
- Inserting data into a table
- Retrieving data from a database
- Creating and changing database objects
- Executing database procedures
- Committing and rolling back transactions
- Accessing secondary connections
- Handling errors

These examples are intended to help you understand the basic concepts of the ADBC classes and their most important methods. For illustration purposes, we have intentionally kept the examples simple. However, keep in mind that the code presented in these examples also works for more complex SQL statements that cannot be expressed by Open SQL. Since the ADBC API is based on ABAP Objects, you should have a basic knowledge of programming with ABAP Objects in addition to some SQL skills.[2]

Before you can send SQL commands to the database and receive results, you need a connection between your SAP system and the database you want to access. So, let's start our ADBC tour by taking a look at how to establish a connection to the database.

[1] For more on dynamic Open SQL, see the article "Enhanced ABAP Programming with Dynamic Open SQL" (*SAP Professional Journal*, September/October 2001).

[2] For a detailed introduction to ABAP Objects, see the article "Not Yet Using ABAP Objects? Eight Reasons Why Every ABAP Developer Should Give It a Second Look" (*SAP Professional Journal*, September/October 2004).

```
DATA con_obj TYPE REF TO cl_sql_connection.

con_obj = cl_sql_connection=>get_connection( ).
```

**Figure 1** Getting the connection object for the default connection

*Note!*

The ADBC API was introduced with SAP Web Application Server (AS) 6.10, and since then its functionality has been slightly enhanced. All the examples given in this article, however, cover functionality that is available in release 6.10 as well as higher releases.

## Establishing a connection to the database

The first thing you have to do is obtain a connection object that points to the database that you want to access. The code snippet in **Figure 1** shows how you can get a connection object for the "default" connection — that is, the database connection that each work process of an application server automatically establishes to the database of the SAP system. (We discuss establishing connections to external databases later in this article.)

A connection object is an instance of class CL_SQL_CONNECTION, which can be created by calling the factory method GET_CONNECTION of this class. In the example, GET_CONNECTION is called without any arguments, which means the object returned refers to the default connection. Throughout the examples in this article we assume that the CON_OBJ variable has been initialized this way and use it to operate on the default connection. The connection object is the crucial point for working with a database, and the anchor for the creation of additional objects required for sending commands to the database and retrieving results. All commands in the context of a certain connection object will be executed on the database connection represented by this object.

Let's start with a simple example that inserts new rows into a table.

## Inserting data into a table

To add new rows to a database table, we need to execute an INSERT statement. Two classes, namely CL_SQL_STATEMENT and its subclass CL_SQL_PREPARED_STATEMENT, provide methods for executing SQL statements. You should use objects of class CL_SQL_STATEMENT if a SQL statement will be executed only once. However, when a SQL statement will be executed several times, but with different input values (e.g., called in a program loop), it's more efficient to use prepared[3] statements that can be created as instances of class CL_SQL_PREPARED_STATEMENT.

We'll take a look at both of these classes individually. We will also explain how to reuse a prepared statement and how to efficiently bind a structured data object to a statement object. Let's start with the CL_SQL_STATEMENT class.

[3] Before a SQL statement can be executed, it first needs to be "prepared," which means that the database compiles the statement and computes a plan for its execution. In the case of statement objects of class CL_SQL_STATEMENT, the "prepare" step is performed implicitly each time a statement is executed. Depending on the complexity of the SQL statement, statement preparation can be a rather expensive operation. Hence, for multiple executions of the same statement using prepared statements is preferable. In this case you prepare the statement only once and then execute it several times.

```
DATA: stmt_obj       TYPE REF TO cl_sql_statement,
      rows_processed TYPE I.

stmt_obj = con_obj->create_statement( ).
rows_processed = stmt_obj->execute_update ( `insert into SFLIGHT `
         && `(CARRID, CONNID, FLDATE, PRICE, CURRENCY) `
         && `values ('LH', '400', '20070719', '666.00', 'EUR')` ).
```

**Figure 2** Inserting data into a table using the CL_SQL_STATEMENT class

## CL_SQL_STATEMENT

The example in **Figure 2** demonstrates how to enter a new flight connection into the database table SFLIGHT. Use the connection object we just defined (CON_OBJ) to create a statement object by calling method CREATE_STATEMENT. This object offers the method EXECUTE_UPDATE, taking the SQL statement text (in this case an INSERT statement) as its argument of type STRING.

You can use the EXECUTE_UPDATE method in the same way to send UPDATE or DELETE statements to the database. After the successful completion of any of these statements, the method returns the number of rows processed (inserted, updated, or deleted) as the result to the caller. In the case of our INSERT example, the return value is 1.

Very often, the values to be written to the database are already given by ABAP program variables (for example, variable PRICE in **Figure 3**). Using the CONCATENATE command to construct the SQL statement text is cumbersome in such cases. Therefore, ADBC offers another approach by using placeholders in the SQL statement text and binding them to ABAP variables. Let's reuse the example in **Figure 2**, but in this case, let's assume that the price information is now given by an ABAP variable.

The code in **Figure 3** shows that the statement text now contains a placeholder — indicated by a question mark (?) — instead of a real column value. Before the statement is executed, the program variable PRICE is bound to the placeholder. This binding is achieved by calling method SET_PARAM and passing a reference to the PRICE variable as its input parameter. At execution time the reference is evaluated and the current value of the PRICE variable is taken as an input value of the column that is marked by the placeholder.

Sometimes you need to execute a SQL statement several times but each time with different values for its input variables. In this case, you should use prepared statements. Let's take a look at populating the database tables using the class CL_SQL_PREPARED_STATEMENT.

***Tip!***

Because ADBC passes the SQL statement to be executed more or less unmodified to the database, it is important that the spelling of identifiers in the SQL text follow the rules of the underlying DBMS. On some database platforms, the database catalog is case-sensitive, and therefore the identifiers in the SQL statement must be identical to those stored in the catalog tables. Furthermore, table or column names must be set in quotation marks if they contain extra characters (e.g., "/BIC/…").

## CL_SQL_PREPARED_STATEMENT

Once a statement has been prepared (compiled) by the DBMS, it can be executed or reexecuted several times with different sets of input values without recompiling.

```
DATA: price          TYPE sflight-price VALUE '666.00',
      stmt_obj       TYPE REF TO cl_sql_statement,
      rows_processed TYPE I,
      ref            TYPE REF TO DATA. "an arbitrary reference to an
                                       "ABAP data object

stmt_obj = con_obj->create_statement( ).
GET REFERENCE OF price INTO ref.
stmt_obj->set_param( ref ).
rows_processed = stmt_obj->execute_update( `insert into SFLIGHT `
          && `(CARRID, CONNID, FLDATE, PRICE, CURRENCY) `
          && `values ('LH', '400', '20070719', ?, 'EUR')` ).
```

**Figure 3** Using placeholders to construct a SQL statement

You use objects of the class CL_SQL_PREPARED_STATEMENT, which is derived from the CL_SQL_STATEMENT class, to represent prepared statements. You can create a CL_SQL_PREPARED_STATEMENT object by calling the PREPARE_STATEMENT method on a connection object. This method takes the text of the SQL statement to be prepared as its argument.

The code snippet shown in **Figure 4** illustrates how you can insert the same flight connection described in the previous examples using a CL_SQL_PREPARED_STATEMENT object.

In this example we used placeholders for all input values of the INSERT statement. For each of these placeholders, method SET_PARAM was called to

```
DATA: prep_stmt_obj TYPE REF TO cl_sql_prepared_statement,
      carrid        TYPE sflight-carrid VALUE 'LH',
      connid        TYPE sflight-connid VALUE '400',
      flight_date   TYPE sflight-fldate VALUE '20070719',
      price         TYPE sflight-price VALUE '666.00',
      currency      TYPE sflight-currency VALUE 'EUR',
      ref           TYPE REF TO DATA.

* Prepare the INSERT statement
prep_stmt_obj = con_obj->prepare_statement( `insert into SFLIGHT `
                  && `(CARRID, CONNID, FLDATE, PRICE, CURRENCY) `
                  && `values (?, ?, ?, ?, ?)` ).

* Bind input variables to each of the placeholders in the SQL statement
GET REFERENCE OF carrid INTO ref.
prep_stmt_obj->set_param( ref ).
```

*Continues on next page*

**Figure 4** Using a prepared statement of class CL_SQL_PREPARED_STATEMENT

**Figure 4** (continued)

```
GET REFERENCE OF connid INTO ref.
prep_stmt_obj->set_param( ref ).
GET REFERENCE OF flight_date INTO ref.
prep_stmt_obj->set_param( ref ).
GET REFERENCE OF price INTO ref.
prep_stmt_obj->set_param( ref ).
GET REFERENCE OF currency INTO ref.
prep_stmt_obj->set_param( ref ).

prep_stmt_obj->execute_update( ).
```

pass a reference to the appropriate program variable to the statement. The call sequence of the SET_PARAM method defines the binding of the input variables to the placeholders (i.e., the first call of SET_PARAM binds the specified variable to the first placeholder, the second call binds to the second placeholder, and so on). When the statement is executed, the placeholders are replaced with the current values of the bound program variables.

***Tip!***

To avoid a runtime error when the statement is executed, ensure that the:

- Number of program variables bound to the statement object corresponds exactly to the number of placeholders in the SQL text
- Referenced program variables are still valid in the scope of the program unit where the statement is executed

The code in **Figure 4** looks pretty much like that in **Figure 3** where we used a CL_SQL_STATEMENT object to execute the INSERT statement. So what's the point of using prepared statements? Unlike a CL_SQL_STATEMENT object, you can reuse a CL_SQL_PREPARED_STATEMENT object to repeat the same kind of operation with another set of parameter values. The example shown in **Figure 5** demonstrates how to enter several flight connections by reusing the prepared statement created in **Figure 4**.

If there is no longer a need to execute a prepared statement, the method CLOSE should be called on the prepared statement object. The CLOSE method frees all resources that were allocated for the execution of this statement on the database. Once you close a prepared statement, you can no longer execute it.

***Tip!***

Use statement objects for one-time SQL commands and prepared statement objects for those being reused with different variable settings.

Very often, you have defined a data structure in your program with components that correspond to the columns of the database table you want to access. In these cases, though possible, it would be very tedious to bind each component of the structure to the statement individually. Alternatively, you can use the method SET_PARAM_STRUCT to bind a structured data object to a statement object. Assuming we have the same PREP_STMT_OBJ that we used in **Figure 4** and a type definition called FLIGHT_T consisting of the

```
DO n TIMES.
* here we can set new values to the program variables bound to PREP_STMT_OBJ
  carrid      = ...
  connid      = ...
  flight_date = ...
  price       = ...
  currency    = ...
  prep_stmt_obj->execute_update( ).
ENDDO.
prep_stmt_obj->close( ).
```

**Figure 5** Reusing a prepared statement

```
DATA: flight TYPE flight_t,
      ref    TYPE REF TO DATA.

GET REFERENCE OF flight INTO ref.
prep_stmt_obj->set_param_struct( ref ).
flight-carrid      = 'AA'.
flight-connid      = '400'.
flight-flight_date = '20070719'.
flight-price       = '500.00'.
flight-currency    = 'USD'.

prep_stmt_obj->execute_update( ).
prep_stmt_obj->close( ).
```

**Figure 6** Binding a structured data object to a prepared statement

fields CARRID, CONNID, FLIGHT_DATE, PRICE, and CURRENCY, we can bind the structured data object FLIGHT by one call of method SET_PARAM_STRUCT to the statement, as shown in **Figure 6**.

In this example, you can see that the field values of the FLIGHT structure were set after a reference to the structure was bound to the statement by method SET_PARAM_STRUCT. Because only a reference is bound, it makes no difference if the field values are set before or after the call of SET_PARAM_STRUCT. At the execution time of the statement, of course, they must be set to the correct values.

***Tip!***

It is crucial that the sequence of components in the bound data structure corresponds exactly to the number and position of the placeholders in the SQL statement text.

Now that you know how to insert data into tables, let's look at how to get that data from the database.

```
DATA: stmt_obj TYPE REF TO cl_sql_statement,
      rs_obj   TYPE REF TO cl_sql_result_set.

stmt_obj = con_obj->create_statement( ).
rs_obj   = stmt_obj->execute_query( `select PRICE, CURRENCY from `
            && `SFLIGHT where CARRID = 'AA' and CONNID = '400'` ).
```

**Figure 7** Executing a query

## Retrieving data from a database

You use queries to get the data you need from a database. For the execution of queries, the class CL_SQL_STATEMENT provides the method EXECUTE_QUERY, which takes the query's SQL text as its argument. As shown in **Figure 7**, the successful execution of method EXECUTE_QUERY implicitly creates a result set object that is an instance of class CL_SQL_RESULT_SET and returns a reference to this object to the caller. This result set object represents the set of rows matching the search condition of the executed query. The code shows the execution of a query that retrieves all flights for connection AA400.

Instead of using literals "AA" and "400," we could also have used placeholders and program variables to specify the flight connections dynamically — just as we demonstrated earlier (see **Figure 3** and **Figure 4**). After a query has been executed successfully, the created result set object provides access to the set of qualifying rows. The following methods are available for retrieving rows of the result set into ABAP data objects:

- Fetching into program variables
- Fetching into a structured data object
- Fetching into an internal table

Next, we'll discuss each of these methods individually.

### Fetching into program variables

To retrieve the matching rows of the query, the method NEXT must be invoked repeatedly on the result set object. This method has no arguments, and returns "1" if the next row has been fetched from the result set and "0" if no rows are remaining (because all rows of the result set have already been delivered).

Before invoking method NEXT, you must define the program variables to where the selected column values must be written. You can use the method SET_PARAM of the CL_SQL_RESULT_SET class to pass a reference to a program variable to the result set object. SET_PARAM must be called once for each column in the query's select list. The call sequence of the SET_PARAM method corresponds to the sequence of columns in the select list (i.e., the first call of SET_PARAM defines the output variable for the first column in the select list, the second call for the second column, and so on). Any subsequent call of the NEXT method fetches the next result row from the database and puts the column values into the respective program variables defined by the previous SET_PARAM calls. The code fragment in **Figure 8** continues the example used in **Figure 7** and shows the processing of the query's result set, which is producing the price list of all flights matching the WHERE condition.

The method NEXT is called in a loop until a return value of "0" indicates that all rows of the result set have been fetched. After each call of the NEXT method that returns with a value greater than "0," the variables PRICE and CURRENCY contain the

```
DATA: price    TYPE sflight-price,
      currency TYPE sflight-currency,
      ref      TYPE REF TO DATA.

...
GET REFERENCE OF price INTO ref.
rs_obj->set_param( ref ).
GET REFERENCE OF currency INTO ref.
rs_obj->set_param( ref ).

WHILE rs_obj->next( ) > 0.
* Now variables price and currency contain the corresponding column values
* of the current row
  WRITE: / price, currency.
ENDWHILE.

rs_obj->close( ).
```

**Figure 8** Fetching data into program variables

corresponding column values of the current row. The method CLOSE finishes the result set processing. You should always call the method CLOSE if the result set processing needs to be stopped prematurely or if the result set has been processed completely, such as in the example in **Figure 8**.

## Fetching into a structured data object

In some cases, it is rather cumbersome to retrieve a table row into a set of program variables. Assume you want to select all data columns of our table SFLIGHT. In this case, it is more convenient to retrieve the result set directly into a predefined ABAP structure where the components correspond to the sequence of columns in the query's select list. Therefore, the class CL_SQL_RESULT_SET provides a special method SET_PARAM_STRUCT that allows you to set a reference to an ABAP structure as its output area. Thereafter, you can use the already mentioned method NEXT to read the rows of the result set into the defined output structure.

***Note!***

The number and sequence of the components in the output structure must completely match the number and sequence of columns in the query's column list (i.e., the first column value will be retrieved into the structure's first component, the second column into the second component, and so on).

**Figure 9** on the next page shows an example that demonstrates this fetch method. This example produces the same price list that is shown in **Figure 8** plus the flight dates, which demonstrates that in contrast to the last example every column value is now accessible.

## Fetching into an internal table

In addition to the method NEXT, which retrieves one row at a time, the class CL_SQL_RESULT_SET

```
DATA: stmt_obj   TYPE REF TO cl_sql_statement,
      rs_obj     TYPE REF TO cl_sql_result_set,
      sflight_wa TYPE sflight,
      struc_ref  TYPE REF TO DATA.

...
rs_obj = stmt_obj->execute_query( `select * from SFLIGHT `
         && `where CARRID = 'AA' and CONNID = '400'`).
GET REFERENCE OF sflight_wa INTO struc_ref.
rs_obj->set_param_struct( struc_ ref ).

WHILE rs_obj->next( ) > 0.
  WRITE: / sflight_wa-price, sflight_wa-currency, sflight_wa-fldate.
ENDWHILE.

rs_obj->close( ).
```

**Figure 9** Fetching into a structured data object

provides an alternative retrieval method that allows you to fetch the result set either completely or packaged into an ABAP internal table.

Similar to fetching into an ABAP structure, this method also requires the component structure of the internal table to match exactly the number and sequence of the columns in the query's select list. Method SET_PARAM_TABLE must be called to define the internal table into which the result set rows should be retrieved. This method expects a reference to the internal table as its argument rather than the internal table itself. After binding the output table by a call of SET_PARAM_TABLE, use the method NEXT_PACKAGE to fetch the result set rows into this internal table. As its result, the method NEXT_PACKAGE returns the number of rows actually retrieved. **Figure 10** shows a code example demonstrating the use of this method, which populates the internal table, FLIGHT_TAB, with all database rows matching the specified WHERE condition.

The method NEXT_PACKAGE has an additional optional input parameter UPTO that you can specify to limit the number of rows that should be retrieved by this call. If the UPTO value is greater than zero, the number of rows to be fetched will be no more than the value specified in UPTO, even if the result set is greater. You can split the complete result set into several batches, reducing the amount of memory required at once.

Regarding performance, fetching into an internal table has the advantage that the call stack of the database interface is traversed less often compared to the row-by-row processing using the NEXT method. However, there is no difference between both methods concerning the number of network roundtrips needed to fetch the data from the database server into the client. Even if the caller uses the NEXT method and processes row-by-row, a pre-fetch mechanism[4] within SAP NetWeaver AS ensures that the number of roundtrips to the database server is minimized.

[4] Usually a certain memory area of the application server is used for the result sets of queries. Instead of transferring the result set row-by-row from the database to the application server, as many rows as fit into the memory area are transferred in one step.

```
DATA: stmt_obj     TYPE REF TO cl_sql_statement,
      rs_obj       TYPE REF TO cl_sql_result_set,
      itab_ref     TYPE REF TO DATA,
      flight_tab   TYPE TABLE OF sflight,
      nbr_of_rows  TYPE I.

...
rs_obj = stmt_obj->execute_query( `select * from SFLIGHT `
           && `where CARRID = 'AA' and CONNID = '400'` ).
GET REFERENCE OF flight_tab INTO itab_ref.
rs_obj->set_param_table( itab_ref ).

* Fetch the complete result set into the internal table
nbr_of_rows = rs_obj->next_package( ).
rs_obj->close( ).
```

**Figure 10** Fetching into an internal table

***Tip!***

Don't forget to close your prepared statement and result set objects to save resources.

Before we turn our attention to some more specific features of the ADBC API, let's summarize the sequence of steps that you must perform to execute a SQL statement with ADBC methods:

1. Establish a database connection to the target data source by calling factory method GET_CONNECTION of class CL_SQL_CONNECTION. Upon a successful connection, this method returns an object of this class, which can further be used to create and execute SQL statements on this connection.
2. On the connection object, call method CREATE_STATEMENT to get a statement object of class CL_SQL_STATEMENT.
3. Build up a data object of type STRING that contains the SQL statement text. You can use placeholders represented by question marks (?) in the SQL text to specify input values to the statement that are evaluated at runtime.
4. For each of the placeholders occurring in the SQL statement text, call method SET_PARAM on the statement object to bind a program variable to that placeholder.
5. Execute the SQL statement by calling either method EXECUTE_QUERY or method EXECUTE_UPDATE on the statement object, thereby passing the SQL string as a method argument. Use EXECUTE_QUERY for SELECT statements, and use EXECUTE_UPDATE for modification operations, such as INSERT, UPDATE, or DELETE statements. Upon a successful execution, EXECUTE_QUERY will return an object of class CL_SQL_RESULT_SET, which can then be used to retrieve the rows of the result set (see steps 6 – 8). EXECUTE_UPDATE returns the number of rows being modified by the statement.
6. In the case of a query, bind each of the output columns of the statement's select list to an appropriate program variable by calling method

SET_PARAM on the CL_SQL_RESULT_SET object returned by EXECUTE_QUERY.

7. To process the result set row-by-row, call method NEXT on the CL_SQL_RESULT_SET object in a loop until the result set is exhausted. Each NEXT call retrieves the column values of the current row into the bound program variables.

8. Finally, close the result set with method CLOSE.

So far we have shown how to retrieve and to change the contents of database tables. Let's now explore how to create new database objects such as tables and procedures using ADBC. In SQL terminology this kind of operation is referred to as Data Definition Language (DDL) commands. Remember that the Open SQL language set does not support DDL commands because the syntax and semantics of these commands are highly database dependent.

## Creating and changing database objects

DDL commands, such as CREATE, DROP, ALTER, etc., should be executed by invoking the instance method EXECUTE_DDL for a given CL_SQL_ STATEMENT object. This method takes only one argument of type STRING where the caller has to pass the SQL text of the DDL command to be executed on the database. **Figure 11** shows an example that creates an Oracle database procedure called GET_BEST_FLIGHT.

This database procedure has two input parameters, CITY_FROM and CITY_TO, one INOUT parameter, PRICE (assuming the caller enters the highest affordable price and receives the price that is closest to it), and one output parameter, FLIGHT_DATE. You can use this procedure to determine the cheapest flight between two cities.

In the next section we demonstrate how to execute such a database procedure with ADBC.

## Executing database procedures

For the execution of database procedures, the class CL_SQL_STATEMENT provides the method EXECUTE_PROCEDURE. As you would for the binding of input variables of a QUERY or UPDATE statement, you need to specify the actual parameters of the procedure call with appropriate calls of method SET_PARAM before the procedure is executed.

In contrast to queries and Data Manipulation Language (DML) operations, a database procedure can have not only IN parameters, but also OUT or INOUT parameters. To distinguish between the different kinds of procedure parameters, the method SET_PARAM has an optional parameter INOUT that you use to specify whether a procedure parameter is an IN, OUT, or INOUT parameter. For setting this parameter accordingly, you can use the public constants C_PARAM_IN, C_PARAM_OUT, and C_PARAM_INOUT, which are defined in the class CL_SQL_ STATEMENT. The default for the INOUT parameter is C_PARAM_IN. The sequence of SET_PARAM calls defines the sequence in which the actual parameters are bound to the formal procedure parameters. The variable referred to in the first SET_PARAM call is passed as the first actual parameter, the second variable as the second parameter, and so on.

Because all the actual procedure parameters must have been specified by appropriate calls of the SET_PARAM method before the invocation of EXECUTE_PROCEDURE, this method itself requires only the procedure name as its argument. **Figure 12** shows an example of the execution of the database procedure GET_BEST_FLIGHT as defined in the previous section. This procedure lists the cheapest flight from New York to Washington.

Now that you understand the main operational features of ADBC, let's take a look at the transaction support it provides.

```
DATA: create_proc_stmt TYPE STRING,
      proc_spec        TYPE STRING,
      proc_impl        TYPE STRING,
      stmt_obj         TYPE REF TO cl_sql_statement.

proc_spec = `create procedure GET_BEST_FLIGHT `
            && `(CITY_FROM in CHAR(30), CITY_TO in CHAR(30), `
            && `PRICE inout NUMBER, FLIGHT_DATE out CHAR(8)) IS`.
proc_impl = ... "Some appropriate implementation
create_proc_stmt = proc_spec && proc_impl.
stmt_obj = con_obj->create_statement( ).
stmt_obj->execute_ddl( create_proc_stmt ).
```

**Figure 11** Creating an Oracle database procedure

```
DATA: price       TYPE sflight-price VALUE '0.00',
      city_from   TYPE spfli-cityfrom VALUE 'New York',
      city_to     TYPE spfli-cityto VALUE 'Washington',
      flight_date TYPE sflight-fldate,
      stmt_obj    TYPE REF TO cl_sql_statement,
      ref         TYPE REF TO DATA.

stmt_obj = con_obj->create_statement( ).
* Bind the program variables to the statement object according to the
* sequence of formal parameters of the GET_BEST_FLIGHT procedure
GET REFERENCE OF city_from INTO ref.
stmt_obj->set_param( ref ).        "IN parameter by default
GET REFERENCE OF city_to INTO ref.
stmt_obj->set_param( ref ).        "IN parameter by default
GET REFERENCE OF price INTO ref.
stmt_obj->set_param( data_ref = ref
                     inout = cl_sql_statement=>c_param_inout ).
GET REFERENCE OF flight_date INTO ref.
stmt_obj->set_param( data_ref = ref
                     inout = cl_sql_statement=>c_param_out ).
stmt_obj->execute_procedure( proc_name ='GET_BEST_FLIGHT' ).
WRITE: / price, flight_date.
```

**Figure 12** Executing a database procedure

```
DATA: con_obj  TYPE REF TO cl_sql_connection,
      stmt_obj TYPE REF TO cl_sql_statement.

con_obj  = cl_sql_connection=>get_connection( ).
stmt_obj = con_obj->create_statement( ).
stmt_obj->execute_update( `delete from SFLIGHT` ).
con_obj->commit( )."This statement persists the deletion
```

**Figure 13** Committing a transaction

```
DATA: con_obj  TYPE REF TO cl_sql_connection,
      stmt_obj TYPE REF TO cl_sql_statement.

con_obj  = cl_sql_connection=>get_connection( ).
stmt_obj = con_obj->create_statement( ).
stmt_obj->execute_update( `delete from SFLIGHT` ).
con_obj->rollback( ). "This statement discards the deletion
```

**Figure 14** Rolling back a transaction

## Committing and rolling back transactions

Transactions are one of the most important concepts when working with a persistency layer like ADBC. Their correct use guarantees the consistency of the underlying data. On every open database connection only one transaction is active at a time. The first 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 on a connection object. As usual the COMMIT method persists all the data changes triggered within the transaction, whereas the ROLLBACK method discards them.

The code snippets in **Figure 13** and **Figure 14** demonstrate how to commit and rollback a transaction running in the default connection. The call of the COMMIT method persists the effect of the DELETE statement on the database (i.e., the SFLIGHT table is empty afterwards), whereas the ROLLBACK method discards the effect of the deletion (i.e., the contents of the SFLIGHT table remain unchanged on the database).

In the examples shown so far, all SQL statements were executed on the default connection. Now we'll explore how you can establish additional connections to the SAP database or even to external databases. We refer to all of these types of connections as "secondary connections" to distinguish them from the default connection.

## Accessing secondary connections

When starting, each work process of an application

```
DATA: con_obj  TYPE REF TO cl_sql_connection,
      con_name TYPE dbcon_name.

con_name = 'ABC'.
con_obj  = cl_sql_connection=>get_connection( con_name ).
```

**Figure 15** Opening a secondary connection

server establishes a connection to the database (the database schema, to be more precise) installed for the SAP system. This connection is called the default connection, as explained earlier. The ADBC API also offers the possibility to open other database connections in addition to the default connection. The two most common scenarios where you might want to open one or more secondary connections are as follows:

- An application needs to access an external data source (i.e., a database or schema different from the SAP system database). The SAP liveCache and the DB Connect architecture of SAP NetWeaver Business Intelligence (BI) Connector are examples of SAP application components used in this scenario. This possibility is also useful for customers who need to connect an SAP system with a customer-specific database.

- An application needs to access the SAP system database, but in the context of an independent transaction that can be committed or rolled back independently of the transaction currently running on the default connection. A typical case for this scenario is protocol data that can be written and committed without being affected by the outcome of the default transaction. That is, protocol data committed on a secondary connection remains persisted even if the default transaction has been rolled back for some reason.

To open a secondary connection, we again use the factory method GET_CONNECTION of class CL_SQL_CONNECTION, but this time with an argument CON_NAME that specifies the logical name of the data source we want to access, as shown in **Figure 15**.

The association between the logical connection name, ABC in the example, and the database-specific parameters needed to establish the connection to the data source (ABC) must be defined externally (outside the program) in the table DBCON. For each external data source to be accessed, this table must contain an entry with all connection data required to identify the target database and for authentication against this database. We refrain from going into more detail on configuring a secondary connection in table DBCON because that would go beyond the scope of this article.[5]

Regarding the second use case mentioned in this section — opening a secondary connection to the SAP system database — it would be unnecessary to create an entry in DBCON because the required connection data is already known to the system. Therefore, the SAP system follows a naming convention where all logical connection names starting with the prefix "R/3*" are assumed to refer to the system database.

After the successful execution of the GET_CONNECTION method, the caller gets a CL_SQL_CONNECTION object that identifies an open connection to the specified database. You can then use this object to create and execute SQL statements

[5] To learn more about secondary connections and the concept of multiple database connections in general, look for an article by Juergen Kissner about multiple database connections, which will appear in an upcoming issue of *SAP Professional Journal*.

```
DATA: con_obj  TYPE REF TO cl_sql_connection,
      con_name TYPE dbcon_name.

con_name ='R/3*-SFLIGHT'.
con_obj  = cl_sql_connection=>get_connection( con_name ).
stmt_obj = con_obj->create_statement( ).
stmt_obj->execute_update( '...' ).
...
con_obj->commit( ).
con_obj->close( ).
```

**Figure 16** Implementing an independent transaction against the system database

***Tip!***

A programming best practice is to use different names for different tasks when using "R/3*" connections. This practice is helpful in error situations to identify the application that opened the connection or when monitoring database connections.

on the referenced database connection in the same manner shown in all previous examples in this article, until it is explicitly closed by a call of the CLOSE method. Thereafter, all statement objects created in the context of this connection object become invalid.

The code in **Figure 16** demonstrates the implementation of an independent transaction running against the SAP system database. Calling the GET_CONNECTION method with a connection name prefixed by "R/3*" opens a new connection to the database and returns a connection object that refers to this connection. Subsequent executions of SQL statements that are created through this connection object belong to the same database transaction that can be committed independently of the transaction running on the default connection. Finally, calling method CLOSE on the connection object closes the database connection. A good programming practice is to close a connection as soon as it is no longer used because database connections allocate valuable resources on the client side (e.g., the application server) but even more on the database server.

***Tip!***

Calling method CLOSE on a CL_SQL_CONNECTION object that refers to the default connection has no effect because the lifecycle of the default connection is coupled to the lifecycle of the work process and therefore cannot be closed programmatically.

Note that the ADBC interface does not support distributed transactions. 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 independently of each other and must be committed separately.

We have now walked through the main features of the ADBC API, but there is one remaining topic that

```
DATA: stmt_obj       TYPE REF TO cl_sql_statement,
      sql_ex         TYPE REF TO cx_sql_exception,
      rows_processed TYPE I.

stmt_obj = con_obj->create_statement( ).
TRY.
    rows_processed = stmt_obj->execute_update( `insert into SFLIGHT `
          && `(CARRID, CONNID, FLDATE, PRICE, CURRENCY) `
          && `values ('LH', '400', '20070719', '666.00', 'EUR') ` ).
CATCH cx_sql_exception into sql_ex.
  IF sql_ex->duplicate_key = 'X'.
    WRITE: / 'Duplicate key error occurred'.
  ELSEIF sql_ex->db_error = 'X'.
    WRITE: / 'SQL error', sql_ex->sql_code, 'occurred:',
           / sql_ex->sql_message.
  ELSE.
    WRITE: / 'Unexpected ADBC error:', sql_ex->internal_error.
  ENDIF.
ENDTRY.
```

**Figure 17** Catching an exception and retrieving error information

***Warning!***

Working with different database connections on the same set of database tables in parallel can lead to deadlock situations where only one work process is involved that can be neither detected nor resolved by the DBMS.

Imagine a program that changes a certain database row on connection one and then tries to modify the same row on connection two without having completed the transaction on connection one. This situation results in the program waiting forever for the lock of the first transaction, because this first transaction will never be able to continue. Such a deadlock situation is difficult to analyze and can be resolved only by terminating the work process. Therefore, be very careful when working with different connections on the same set of database tables!

is too often disregarded by programmers — error handling. Especially when you are dealing with complex software systems like databases, you should always take error situations into account.

## Handling errors

Some of the ADBC methods presented in the previous sections can run into errors that need to be handled by the caller. If such an error situation occurs (e.g., because the execution of a SQL command gets a SQL error from the underlying DBMS) the ADBC method throws an exception object of class CX_SQL_EXCEPTION.

Within a “TRY … ENDTRY” block, the caller can catch the exception and get some error information from the exception object thrown. This technique is illustrated in the code shown in **Figure 17**, which is identical to that in **Figure 3**, except now with proper error handling.

A CX_SQL_EXCEPTION object provides several attributes that you can evaluate within the CATCH block and use to handle error situations. SQL errors raised by the underlying DBMS are indicated by the flag attribute DB_ERROR, which is set to X. The programmer may also get detailed information about the root cause of the SQL error through attributes SQL_CODE and SQL_MESSAGE, which contain the original SQL code and SQL message as returned by the DBMS. This information not only helps to write proper error messages, as shown in **Figure 17**, but you can also use it as part of the program logic. For example, your program might react differently if the SQL code indicates that a certain database table does not exist or if there is a database corruption.

***Tip!***

The attributes SQL_CODE and SQL_ MESSAGE should only be evaluated if db_error = 'X'; otherwise, their values are undefined.

If errors occur that cannot be classified by one of the flag attributes, the INTERNAL_ERROR attribute is set to an internal error code that may help SAP support to identify the problem. The log and trace files usually contain more information for error diagnostics in such cases.

Now that you have a general understanding of the main features and capabilities of the ADBC API, let's compare the different SQL programming models available in ABAP and provide some recommendations on use cases for the different models.

## Comparing the ABAP SQL programming models

The table in **Figure 18** lists the most important characteristics of the three SQL programming models offered by SAP NetWeaver AS ABAP, including the advantages and disadvantages of each. However, the intention of this comparison is not to derive a ranking between these models — all have their use cases. The purpose rather is to determine in which case each model fits best.

In view of this comparison, we recommend the following guidelines to ABAP programmers regarding the use of the different programming models:

- Open SQL is the undisputable choice for ABAP developers who want to write portable applications against database tables residing in the standard database schema of SAP NetWeaver AS. In addition to portability, they can benefit from many performance optimizations and other useful functions of the ABAP database interface for free. A prerequisite, of course, is that all accessed database tables and views are defined by means of the ABAP Dictionary.

- Typical use cases for EXEC SQL commands are static SQL statements against database objects that are not defined in the ABAP Dictionary or SQL statements exploiting database-specific functionality that cannot be expressed by Open SQL. The implementation of the ABAP Dictionary, for example, makes heavy use of EXEC SQL commands to query the database catalog and execute DDL statements that cannot be done in Open SQL.

- ADBC is preferable in scenarios where applications need access to external databases or require SQL functionality not provided by Open SQL. For example, ADBC is the only option for executing dynamic Native SQL statements. Furthermore, it is preferable over EXEC SQL when programmers have to deal with multiple database connections or need complete program control in case of exceptions.

## Conclusion

ADBC amends ABAP Native SQL functionality provided by the EXEC SQL command. This powerful

| Open SQL | EXEC SQL | ADBC |
|---|---|---|
| SQL commands are fully integrated into the ABAP language | SQL statements are embedded in EXEC SQL … ENDEXEC brackets | Object-based call-level interface |
| Comprises a subset of standard SQL with some very useful enhancements in conjunction with other ABAP language constructs | Provides access to the entire SQL functionality of the respective database system, including the execution of stored procedures | Provides access to the entire SQL functionality of the respective database system, including the execution of stored procedures |
| Guarantees portability across all database platforms supported by SAP NetWeaver AS | Does not guarantee portability of SQL statements | Does not guarantee portability of SQL statements |
| Accessed database tables/views must be defined in the ABAP Dictionary | Accessed database tables/views need not be defined in the ABAP Dictionary | Accessed database tables/views need not be defined in the ABAP Dictionary |
| Automatic client handling | No automatic client handling | No automatic client handling |
| Compile-time checking of static SQL statements | No compile-time checking of SQL statements | No compile-time checking of SQL statements |
| As of ABAP 6.10 nearly all parts of a SQL statement can be dynamic | Allows only the execution of static SQL statements | All SQL statements are dynamic by nature of the ADBC API |
| Built-in performance optimizations by the ABAP database interface through client-side table buffering, pre-fetching of result sets, batched DML operations, etc. | No specific performance optimizations by the ABAP database interface | No specific performance optimizations by the ABAP database interface |
| Access to tables residing in data sources other than the standard schema of SAP NetWeaver AS is possible as of ABAP release 6.10, but this feature requires the ABAP Dictionary to contain a corresponding table definition | Access to external data is supported, but the handling of several database connections is rather error-prone | Access to external data sources is supported; the connection handling is straightforward because SQL statements can only be executed in the context of a connection object that can be controlled by the programmer |
| Limited program control in case of exceptions | Limited program control in case of exceptions | Full program control in case of exceptions |

**Figure 18** Comparison of SQL programming models in ABAP

API is suitable for using most of the SQL functionality provided by relational database systems. Going beyond the functionality of EXEC SQL commands, ADBC enables the execution of dynamic SQL statements that are completely created at runtime. Furthermore, it gives the programmer the ability to react to SQL errors in an appropriate manner. Each API method that potentially can run into an error situation throws an exception that can be caught and handled by caller. Last but not least, and because of the object-oriented nature of the API, ADBC provides a clean concept to deal with multiple database connections and external databases that help to avoid the traps and pitfalls of the EXEC SQL approach.

*Thomas Raupp studied computer sciences at the University of Karslruhe, Germany. After receiving his degree in 1985, he worked for the Research Centre of Computer Sciences (FZI) in Karlsruhe focusing on object-oriented databases. He joined the R/3 technology department at SAP in 1990, where he became Development Manager of the Database Interface Group in 1996 and was responsible for the SAP R/3 database interface and the Oracle port of SAP R/3. Recently, Thomas moved from SAP NetWeaver development to the Global Service and Support organization and is now the Development Manager for the IMS (Installed Base Maintenance & Support) ABAP Group. You may reach him at thomas.raupp@sap.com.*

*Tobias Wenner has a degree in Computer Science from the University of Karlsruhe, Germany. He joined SAP in 1994, lending his skills to data dictionary and database interface development, both in the ABAP and Java stack of SAP NetWeaver. Tobias is responsible for the design and implementation of Open SQL as well as Native SQL in the ABAP compiler and interpreter. You may reach him at tobias.wenner@sap.com.*