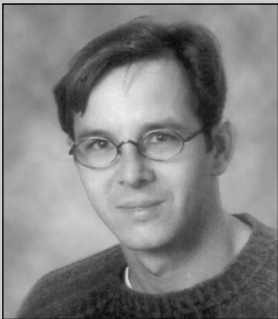
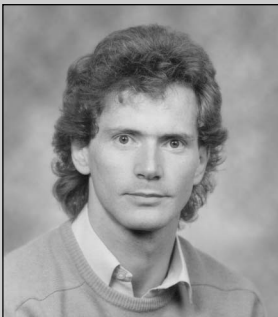


Enhanced ABAP Programming with Dynamic Open SQL

Adrian Görler and Ulrich Koch



*Adrian Görler
Business Programming
Languages Group
SAP AG*



*Ulrich Koch
Business Programming
Languages Group
SAP AG*

(complete bios appear on page 46)

A major strength of ABAP is that database access via Open SQL is embedded in the language. This tight integration allows for optimal performance, platform independence, and compile time syntax-checking. This article deals with a class of problems, such as generic queries, which cannot be addressed with static Open SQL. The reason is that not all database tables, table fields, or logical conditions involved are known at compile time. One possible way to deal with these types of situations is to dynamically generate and execute an ABAP program. But this approach has some severe disadvantages:

- For starters, writing a source code generator is difficult, and so is verifying that it will execute in strict accordance with your expectations.
- In terms of computation time, persistently generating an ABAP program for dynamic purposes is a very expensive operation.
- If the generated source code exists only transiently by the statement `GENERATE SUBROUTINE POOL`, debugging can become quite difficult.

This article focuses on Dynamic Open SQL, which offers a vastly improved way to address development efforts that involve dynamic or semi-dynamic database access when a part of an SQL statement, or even an entire SQL statement, is not statically known. This variant of Open SQL is not hampered by the drawbacks of source code generation. Instead, all possible syntax checks are performed at compile time. The performance overhead added for parsing the dynamic components of the statement is minimal, and because no intermediate source code is generated, the program is easy to debug.

Dynamic Open SQL: General Concepts

Dynamic Open SQL is an extension of Open SQL. It is not required that an entire SQL statement be dynamically specified. The parts of the statement that are statically known are expressed via familiar static ABAP coding. The fragments that contain dynamic elements are denoted by dynamic token specification. The dynamic fragment is not explicitly contained in the ABAP source code. Instead, the source code contains an ABAP variable enclosed in parentheses as a placeholder. The program must contain coding that creates the ABAP source code for the dynamic fragments and stores it in this variable. At runtime, the ABAP source code of the dynamic fragments is parsed and mixed into the static parts of the statement. Dynamic and static parts appear to the database as a single statement. This process is completely transparent to the database.

During the development of ABAP, the feature set of Dynamic Open SQL has been continuously expanded from Release 3.0 to Release 4.6. In ABAP Release 6.10,¹ this offering has been greatly enhanced. In this article, we examine the features offered with ABAP 4.6 and 6.10. We offer some easy-to-follow demonstration programs designed to acquaint you with the world of Dynamic Open SQL.

Unless otherwise stated, this article covers techniques available with ABAP Release 4.6. The ABAP 6.10 examples that you find in these pages will be clearly marked as such. We offer these 6.10 examples to introduce you to some of the new features you'll find in Release 6.10.

¹ That is, the ABAP release that comes with the SAP Web Application Server Release 6.10.

Let's now take a closer look at how Dynamic Open SQL works by walking through some examples of its different features (both 4.6 and 6.10), including using a dynamic table name, retrieving data, using a dynamically created internal table, the dynamic WHERE, SELECT, GROUP BY, HAVING, and FROM clauses, exception handling, and dynamic update operations.

Using a Dynamic Table Name

Our first, most basic example (**Listing 1**) illustrates the use of a dynamically specified table name. The program displays a list with the names of three different database tables. With a double-click, the user can choose one table from the list, and the program will show the number of entries of the selected table.

Listing 1: Table Name Dynamically Specified (ABAP 4.6)

```
1 data: tabname type tabname,  
2     count   type i.  
3  
4 start-of-selection.  
5   write: / 'SPFLI', / 'SFLIGHT', / 'SBOOK'.  
6  
7 at line-selection.  
8   read current line line value into tabname.  
9   select count(*) from (tabname) INTO count.  
10  write: 'The table', tabname(7), 'contains',  
11  count, 'entries.'
```

At the event `line-selection`, the content of the selected line is read into the field `tablename`. In the `SELECT` statement, the name of the database table is not specified statically. Instead, the clause `from (tablename)` denotes that the name of the database table is to be read from the variable `tablename` and substituted into the `SELECT` statement:

```
9  select count(*) from (tablename) INTO
   count.
```

Upon execution of the `SELECT` statement, the field `tablename` is parsed for a valid name of a database table. If `tablename` doesn't contain a valid name, an exception occurs.

✓ Tip

In ABAP Release 4.6 and earlier, a dynamic table name has to be specified in uppercase. This restriction will be lifted with ABAP Release 6.10.

Retrieving Data

Usually, one is interested in retrieving records from the database. The problem is specifying an appropriate work area into which to fetch the data. A feasible approach would be to fetch the data into a container big enough to hold the data from the dynamically

specified table. The fields of the retrieved records could be accessed by casting the container to the appropriate type. Unfortunately, this technique has two disadvantages:

- You have to choose a container that is large enough to hold the data of any possible database table.
- The approach requires a flat container.²

While choosing a very *large* container typically translates into a waste of resources, it frequently can be accepted in practice. The *flat* container requirement is the more severe restriction. It prevents the program from running with tables containing strings, which is something ABAP will be capable of doing with Release 6.10.

A better way to construct an appropriate work area is via the command `CREATE DATA`,³ which is available with Release 4.6. It allows you to create data objects with a dynamically specified type at runtime. Using this technique, you can write a basic query tool (**Listing 2**). In this example, the user can select one of three tables, and the program will display its content.

² A flat container is a container that does not contain strings, internal tables, or references.

³ You will find a detailed discussion of `CREATE DATA` in the article "Build Flexibility and Control into Your ABAP Programs with Dynamic Programming Techniques," which will appear in the upcoming November/December 2001 issue of *SAP Professional Journal*.

Listing 2: Retrieving Data Row by Row (ABAP 4.6)

```
1  start-of-selection.
2  write: / 'SPFLI', / 'SFLIGHT', / 'SBOOK'.
3
4  at line-selection.
5  data: dref    type ref to data,
6         tablename type tablename.
7
8  field-symbols: <row>          type any,
9                 <component> type any.
10
```

(continued on next page)

(continued from previous page)

```

11 read current line line value into tabname.
12
13 * dynamically create appropriate internal table
14 create data dref type (tabname).
15 assign dref->* to <row>.
16
17 * fetch the data
18 select *
19   from (tabname) up to 20 rows
20   into <row>.
21
22 * display the result
23   new-line.
24   do.
25     assign component sy-index of structure <row>
26       to <component>.
27     if sy-subrc <> 0.
28       exit. " no more components
29     endif.
30     write: <component>.
31   enddo.
32 endselect.

```

First, a variable `dref` with type `REF TO DATA` is declared, which can hold a reference to any data object:

```

5 data: dref type ref to data,
[...]
```

Then, a work area with line type `tabname` is created dynamically by the ABAP command `CREATE DATA`:

```

14 create data dref type (tabname).
```

After the execution of this statement, the data reference `dref` is referring to this work area. It cannot be accessed directly. Instead, a field symbol `<row>` is assigned to the work area referred by `dref`:

```

15 assign dref->* to <row>.
```

Now we can safely fetch the data from the table

`tabname` into `<row>` because it is guaranteed that the database table is type-compatible with the work area. We limit the result set to 20 rows:

```

18 select *
19   from (tabname) up to 20 rows
20   into <row>.
[...]
```

Looping over all components of `<row>`, we can display the fetched data as a list:

```

24   do.
25     assign component sy-index of
26       structure <row>
27       to <component>.
28     if sy-subrc <> 0.
29       exit. " no more components
30     endif.
31     write: <component>.
32   enddo.

```

New with ABAP 6.10: Using Dynamically Created Internal Tables

With ABAP Release 6.10, it will be possible to create not only *structures* with a dynamically specified type, but also *internal tables* with a dynamically specified line type. With this new feature, we can rewrite our last example (see **Listing 3**). Here we use an array fetch instead of a `SELECT ... ENDSELECT` loop:

```
13 create data dref type table of
    (tabname).
14 assign dref->* to <itab>.
15
```

```
16 select *
17   from (tabname) up to 20 rows
18   into table <itab>.
```

The array fetch will reduce the execution time significantly because multiple roundtrips to the database interface will be avoided.

Instead of displaying our result set as a list, we show it via an ALV grid control. To do so, we first create an instance of the class `cl_gui_alv_grid` and assign the entire screen as its parent:

```
20 create object alv
21   exporting i_parent =
    cl_gui_container=>screen0.
```

Listing 3: Retrieving Data by Array Fetch (ABAP 6.10)

```
1 start-of-selection.
2   write: / 'SPFLI', / 'SFLIGHT', / 'SBOOK'.
3
4 at line-selection.
5 data: tabname type tabname,
6       dref type ref to data,
7       alv type ref to cl_gui_alv_grid.
8
9 field-symbols: <itab> type any table.
10
11 read current line line value into tabname.
12
13 create data dref type table of (tabname).
14 assign dref->* to <itab>.
15
16 select *
17   from (tabname) up to 20 rows
18   into table <itab>.
19
20 create object alv
21   exporting i_parent = cl_gui_container=>screen0.
22
23 call method alv->set_table_for_first_display
24   exporting i_structure_name = tabname
25   changing it_outtab      = <itab>.
26
27 call screen 100.
```

Then, the `set_table_for_first_display` method is called, submitting the dynamically specified table name `tabname` and the result set `<itab>` to the grid control:

```
23 call method alv->
    set_table_for_first_display
24   exporting i_structure_name =
        tabname
25   changing it_outtab      =
        <itab>.
```

Now we call screen 100 and get a nicely rendered display of the selected data.⁴ Please note that appropriate column headers with tool tips are added automatically.

The Dynamic WHERE Clause

A very useful feature of Dynamic Open SQL is the ability to dynamically specify a WHERE clause.

Listing 4 illustrates how this is done. The program asks the user for the names of a flight's departure and arrival cities. Users can choose whether the two values must match simultaneously (`OP_AND`), or not (`OP_OR`). The program then displays a list of all matching flights.

First, we declare an internal table `where_tab` to hold the source code of the dynamic WHERE clause:

```
6 data: where_tab type table of
          edpline,
7         source_line type edpline,
          [...]
```

Next, we generate the source code for the WHERE clause. It shall restrict the values of the database columns `cityfrom` and `cityto` to the values of the parameters `depart` and `arrive`, respectively. Though the field names involved are known statically, the operator (`AND` or `OR`) that connects the two logical conditions is known only at runtime. To

construct the source code of the WHERE clause, static and dynamic parts are concatenated and appended to `where_tab`:

```
16 concatenate 'cityfrom = '' depart
    ''' into source_line.
17 append source_line to where_tab.
18
19 append operator to where_tab.
20
21 concatenate 'cityto = '' arrive
    ''' into source_line.
22 append source_line to where_tab.
```

Assuming that the user entered the values "FRANKFURT" and "NEW YORK" for the parameters `depart` and `arrive`, respectively, and that he chose the option `OP_AND`, `where_tab` contains the following lines at this point:

```
cityfrom = 'FRANKFURT'
AND
cityto = 'NEW YORK'
```

On execution of the `SELECT` statement, the WHERE clause is read from the variable `where_tab`, dynamically parsed, and then executed:

```
24 select carrid connid cityfrom cityto
25       from spfli
26       into (carrid, connid, depart,
            arrive)
27       where (where_tab).
28       [...]
29 endselect.
```

It is possible to mix static and dynamic parts of a WHERE clause. If, for example, the selection should always be restricted to flights of the carrier Lufthansa (`carrid = 'LH'`), we would alter the `SELECT` statement in Listing 2 (shown above) as follows:

```
select carrid connid cityfrom cityto
       from spfli
       into (carrid, connid, depart,
            arrive)
       where carrid = 'LH' and (where_tab).
       [...]
endselect.
```

⁴ In order to run, the program requires screen 100 to exist. You can create screen 100 by double-clicking on the word "100," and then saving the pop-up screen unchanged.

Listing 4: WHERE Clause Dynamically Specified (ABAP 4.6)

```

1 parameters: depart      type spfli-cityfrom,
2               arrive    type spfli-cityto,
3               op_and    radiobutton group 1 default 'X',
4               op_or     radiobutton group 1.
5
6 data: where_tab  type table of edpline,
7       source_line type edpline,
8       operator(3) value 'AND',
9       carrid     type spfli-carrid,
10      connid     type spfli-connid.
11
12 if op_or = 'X'.
13   operator = 'OR'.
14 endif.
15
16 concatenate 'cityfrom = '' depart '''' into source_line.
17 append source_line to where_tab.
18
19 append operator to where_tab.
20
21 concatenate 'cityto = '' arrive '''' into source_line.
22 append source_line to where_tab.
23
24 select carrid connid cityfrom cityto
25       from spfli
26       into (carrid, connid, depart, arrive)
27       where (where_tab).
28   write: / carrid, connid, depart, arrive.
29 endselect.

```

The static and dynamic parts of the WHERE clause are combined at runtime and sent to the database as a single clause. Any mixture of static and dynamic parts is allowed, as long as every single dynamic fragment represents a logical condition.

✓ Tip

A WHERE clause may consist of a mixture of multiple static and dynamic fragments.

New with ABAP 6.10: More Convenient Ways to Supply Source Code Dynamically

As a general concept of Dynamic Open SQL, the source code of a dynamically specified clause or logical condition is supplied in an ABAP variable. With ABAP Release 6.10, some restrictions that currently apply to the ABAP source code of the dynamic parts will be lifted. To demonstrate the new possibilities, here we rewrite

Listing 5: More Convenient Ways to Supply Source Code Dynamically (ABAP 6.10)

```

1 parameters: depart      type spfli-cityfrom,
2               arrive    type spfli-cityto,
3               op_and    radiobutton group 1 default 'X',
4               op_or     radiobutton group 1.
5
6 data: where_clause type string,
7       operator(3) value 'AND',
8       carrid      type spfli-carrid,
9       connid      type spfli-connid.
10
11 constants: tabname(10) value 'spfli'.
12
13 if op_or = 'X'.
14   operator = 'OR'.
15 endif.
16
17 concatenate ` cityfrom = depart `
18             operator
19             ` cityto = ` arrive ``
20   into where_clause.
21
22 select carrid connid cityfrom cityto
23   from (tabname)
24   into (carrid, connid, depart, arrive)
25   where (where_clause).
26 write: / carrid, connid, depart, arrive.
27 endselect.

```

our last example using some of the upcoming features (see **Listing 5**).

Up to ABAP Release 4.6D, the ABAP programmer is bound to two types of variables to hold the source code information of the dynamic parts of a Dynamic Open SQL statement: dynamically specified table names have to be supplied in variables of type C. The source code of all other clauses has to be submitted in an internal table with line type C and a line width of 72 characters.⁵ With ABAP Release 6.10, a wider variety of data types will be valid so that the following four data types will be available:

- C, any length
- STRING

⁵ A suitable line type for the internal table is the type edpline defined in the Data Dictionary.

- Internal table with line type C, any line width
- Internal table with line type STRING

In our modified example, we use the string `where_clause` to hold the source code of the dynamic WHERE clause:

```
6 data: where_clause type string,
```

Now the static and dynamic parts can be concatenated directly into the string `where_clause` without appending them to an internal table in a second step:

```
17 concatenate ` cityfrom = depart `
[... ]
20   into where_clause.
```

As a second advantage, when using a string to hold the code of the clause instead of an internal table, the programmer doesn't have to worry about exceeding the line width of the internal table. A little disadvantage of using a string is, however, that the source code of the generated clause might not be as easy to read as an internal table, if there is need for debugging.

Up to ABAP Release 4.6D, the right-hand side of a comparison operator contained in a dynamic WHERE clause was restricted to being a literal or column identifier. With ABAP Release 6.10, this restriction will be lifted, so that it will additionally be possible to use identifiers of ABAP variables. As an example, we directly include the ABAP variable `depart` in the source code of the dynamic WHERE clause:

```
17 concatenate ` cityfrom = depart `
   [...] ]
```

At the execution of the program, the WHERE clause is parsed. The runtime engine searches for an ABAP variable with the name `depart` and evaluates its value. This lookup of a variable's value by its name has a minimal computational overhead cost if the ABAP variable is declared in the scope enclosing the SELECT statement. It is therefore recommended that dynamic clauses contain ABAP variables only if they are generated once and executed multiple times with the variables' values. In this case, the computation time saved by generating the source code only once will compensate for the cost of the dynamic lookup of the variable name.

Like our example in Listing 5, the source code of a dynamic clause frequently includes quoted text. When constructing the clause, you have to ensure that:

- All single quotes (') enclosing text literals⁶ are properly escaped
- The coding doesn't rely on trailing spaces of text literals

⁶ A *text literal* is a character sequence enclosed by single quotes ('). Text literals have ABAP type C. Any trailing spaces are ignored.

This often makes the source code generator difficult to read and maintain. In ABAP Release 6.10, one will have the ability to greatly enhance the readability of the coding by using string literals⁷ instead. A single quote contained in a string literal doesn't have to be escaped. Additionally, trailing spaces are recognized:

```
17 concatenate ` cityfrom = depart `
18             operator
19             ` cityto = ` arrive ` `
20             into where_clause.
```

Please compare this code fragment with lines 16-22 of Listing 4.

As our example is to run under Release 6.10, we can specify the dynamic table in uppercase or lowercase:

```
11 constants: tabname(10) value
   'spfli'.
   [...] ]
22 select carrid connid cityfrom cityto
23        from (tabname) ...
```

✓ Tip

Try to use string literals when constructing the source code of a dynamic clause under ABAP Release 6.10.

The Dynamic SELECT, GROUP BY, and HAVING Clauses

The dynamic SELECT, GROUP BY, and HAVING clauses allow you to specify projections and aggregations of selected rows at runtime. Let's take a closer look at how these three clauses work.

⁷ A *string literal* is a character sequence enclosed by single back quotes (`). String literals have ABAP type STRING. Any trailing spaces are recognized.

The Dynamic SELECT and GROUP BY Clauses

In all our examples so far, we retrieved entire rows of data from the database. If one is only interested in the values of certain columns of the database, this represents a waste of database resources. Dynamic Open SQL approaches this problem with a dynamic SELECT clause. Just like a dynamic WHERE clause, the programmer can supply the source code of a SELECT list in an ABAP variable. Like its static counterpart, this dynamic SELECT clause may contain aggregate functions and aliases.

If the SELECT list contains aggregate functions, the SQL standard requires all columns in the SELECT list that are not aggregated to be listed in the GROUP BY clause.

To satisfy this rule, if a dynamic SELECT clause is given, the user may specify a dynamic GROUP BY clause in Dynamic Open SQL. As usual, the pro-

grammer supplies a list of column identifiers in a placeholder variable.

Let's have a first look at our next example (**Listing 6**). It illustrates the usage of the dynamic SELECT and GROUP BY clauses. The user may choose any columns of the table SPFLI. The program determines the projection of SPFLI on the selected columns. All distinct rows of this projection and the frequency of their occurrence in the projection constitute the result set. They are displayed as a list. For example, one could display the number of flights in SPFLI for each connection between two cities.

In our program, we declare two internal tables — `sel_list` and `group_list` — to hold the source code of the dynamic SELECT and GROUP BY clauses:

```
16 data: sel_list    type table of edpline,
17       group_list type table of edpline,
```

Listing 6: Dynamic SELECT, GROUP BY, and HAVING Clauses (ABAP 6.10)

```
1 type-pools abap.
2
3 parameters: lt      radiobutton group 1 default 'X',
4             gt      radiobutton group 1,
5             value   type i.
6
7 data: begin of wa,
8       count   type i.
9       include type spfli.
10 data: end of wa.
11
12 data: checked,
13       name     type fieldname,
14       lines    type i,
15       descr_ref type ref to cl_abap_structdescr,
16       sel_list type table of edpline,
17       group_list type table of edpline,
18       having   type string.
19
20 field-symbols: <fs>      type any,
```

```
21             <comp_wa> type abap_compdescr.
22
23 start-of-selection.
24
25     set pf-status 'MAIN'.
26
27 * get all components of table 'SPFLI'
28     descr_ref ?=
29         cl_abap_typedescr=>describe_by_name( 'SPFLI' ).
30
31     loop at descr_ref->components assigning <comp_wa>.
32         name = <comp_wa>-name.
33         write: / checked as checkbox, name.
34     endloop.
35     lines = lines( descr_ref->components ).
36
37 at user-command.
38
39 * determine selected columns
40     clear: sel_list, group_list.
41     append 'count(*) as count' to sel_list.
42     do lines times.
43         read line sy-index field value checked.
44         if checked = 'X'.
45             read line sy-index field value name.
46             append name to: sel_list, group_list.
47         endif.
48     enddo.
49
50 * determine operator
51     if gt = 'X'.
52         having = 'count(*) > value'.
53     else.
54         having = 'count(*) < value'.
55     endif.
56
57     select (sel_list)
58         from spfli up to 20 rows
59         into corresponding fields of wa
60         group by (group_list)
61         having (having).           "remove for 4.6
62
63 * write all components to list
64     write: / wa-count.
65     loop at group_list into name.
66         assign component name of structure wa to <fs>.
67         write: <fs>.
68     endloop.
69 endselect.
```

To hold the selected data, we declare a work area wa:

```

7 data: begin of wa,
8     count   type i.
9     include type spfli.
10 data: end of wa.

```

At the event `start-of-selection`, a list of all columns of the table `SPFLI` is displayed with checkboxes for the user to choose from. If the user presses the `ENTER` key, the event `user-command` is triggered.⁸

At the event `user-command`, `count(*)` is added to the `SELECT` list as the first field. The alias `count` matches with the component `count` of the work area `wa`. Next, the values of the checkboxes are extracted from the list to determine the chosen columns. To do so, we loop over the list and append the names of the checked columns to the internal table `sel_list`. Because the `SELECT` list contains the aggregate function `count(*)`, we have to include all column identifiers in `sel_list` as well as in `group_list`:

```

41 append 'count(*) as count' to
    sel_list.
42 do lines times.
43     read line sy-index field value
        checked.
44     if checked = 'X'.
45         read line sy-index field value
            name.
46         append name to: sel_list,
            group_list.
47     endif.
48 enddo.

```

The data of the columns specified in `select_list` are fetched from the database table `SPFLI`, and grouped according to `group_list`:

```

57 select (sel_list)
58     from spfli up to 20 rows

```

⁸ To successfully execute, the program must have a GUI status "MAIN" that defines a button with function code "ENTER."

```

59     into corresponding fields of wa
60     group by (group_list)
[...].
69 endselect.

```

All techniques presented so far are available with ABAP Release 4.6.⁹

New with ABAP 6.10: The Dynamic HAVING Clause

The logical condition specified in a `WHERE` clause determines a set of records, which can further be grouped as specified in a `GROUP BY` clause, building an intermediate result set. Because the `WHERE` clause is applied before the data is aggregated and grouped, it cannot contain any aggregate functions. However, applying a logical condition in the `HAVING` clause, the grouped data can be further reduced. With ABAP Release 6.10, it will be possible to specify a `HAVING` clause at runtime. It is used exactly as a dynamic `WHERE` clause, but may additionally contain aggregate functions.

Continuing with our example (Listing 6), we demonstrate the usage of a dynamic `HAVING` clause. As a simple example, the user is asked to enter a value restricting `count(*)`. He can then decide whether value shall be an upper (`lt`) or a lower (`gt`) limit of `count`.

The source code of the dynamic `HAVING` clause is simply chosen from one of two literals, depending on the value of the parameter `gt`. Please note that the `HAVING` clause contains the aggregate function `count(*)` and the ABAP variable value. These are new features that will be available only with ABAP Release 6.10:

```

51 if gt = 'X'.
52     having = 'count(*) > value'.
53 else.
54     having = 'count(*) < value'.
55 endif.

```

⁹ In order to run the example on a 4.6 system, please comment line 61 and close the statement on line 60 with a period (".").

In the `SELECT` statement, the dynamically specified `having_clause` is now used to filter the intermediate data resulting from the `GROUP BY` clause:

```
57 select (sel_list)
58     from spfli up to 20 rows
59     into corresponding fields of wa
60     group by (group_list)
61     having (having).
[...]
```

Finally, the result set is displayed column by column, as it was done before.

✓ Tip

Some empty dynamic clauses are specially treated:

- **WHERE or HAVING clause:** *The expression is TRUE.*
- **SELECT clause:** *All columns are selected.*

New with ABAP 6.10: Exception Handling

In our previous examples, we allowed the user to choose a database table name or column name from a list of predefined values. This technique guarantees that the table and column names are valid. However, if the user were allowed to freely type in a table name, he would likely enter something invalid from time to time. In such cases, a system exception would be thrown, causing the program to terminate with a runtime error. In a productive system, such program behavior is obviously unacceptable. With ABAP Release 6.10, it will be possible to catch the system exceptions¹⁰ raised during execution of an SQL statement, so that such situations can be dealt with correctly.

The concept of class-based exceptions is similar to the exceptions in C++ or Java. We illustrate this technique via a modified version of Listing 2 (see Listing 7). Instead of having the user choose a

¹⁰ For a detailed discussion of exception handling in ABAP Release 6.10, please refer to the online documentation.

Listing 7: Exception Handling (ABAP 6.10)

```
1 parameter tabname type tabname.
2
3 data: dref          type ref to data,
4       xref          type ref to cx_dynamic_check,
5       message_txt  type string.
6
7 field-symbols: <itab>    type standard table,
8                 <row>    type any,
9                 <component> type any.
10
11 start-of-selection.
12
13 try.
14 * dynamically create appropriate internal table
15   create data dref type table of (tabname).
16   assign dref->* to <itab>.
17
18 * fetch the data
```

(continued on next page)

(continued from previous page)

```

19     select *
20     from (tablename) up to 20 rows
21     into table <itab>.
22
23     catch cx_sy_dynamic_osql_error
24           cx_sy_create_data_error into xref.
25     message_txt = xref->get_text( ).
26     message message_txt type 'E'.
27 endtry.
28
29 * display the result
30 loop at <itab> assigning <row>.
31   new-line.
32   do.
33     assign component sy-index of structure <row>
34     to <component>.
35     if sy-subrc <> 0.
36       exit. " no more components
37     endif.
38     write: <component>.
39   enddo.
40 endloop.

```

table name from a list, he may enter a table name as a parameter:

```
1 parameter tablename type tablename.
```

In our program, there are two statements in which an invalid table name can raise a system exception: CREATE DATA and SELECT. To be able to catch a system exception, we enclose the critical section with a TRY ... CATCH ... ENDMETHOD block:

```

13 try.
14 * dynamically create appropriate
   internal table
15   create data dref type table
of (tablename).
16   assign dref->* to <itab>.
17
18 * fetch the data
19   select *
20   from (tablename) up to 20
   rows
21   into table <itab>.
22

```

```

23   catch cx_sy_dynamic_osql_error
24         cx_sy_create_data_error
   into xref.
25   message_txt = xref->
   get_text( ).
26   message message_txt type 'E'.
27 endtry.

```

If the critical section between TRY and CATCH is passed through without errors, the CATCH block is not reached and the control flow continues after ENDMETHOD.

If, however, tablename has an illegal value, either CREATE DATA or SELECT will raise an exception of class cx_sy_create_data_error or cx_sy_dynamic_osql_error, respectively, and create an exception object. The control flow will be redirected to the CATCH block where this exception object can be accessed through the reference xref. The method get_text returns a descriptive error text, which is put out as a message. The

program terminates normally and the user can step back to the selection screen and correct the erroneous table name.

Update Operations

With Dynamic Open SQL, it is possible to perform database changes by statements, which can contain dynamic table names or a dynamic SET or WHERE clause. Let's take a closer look at these capabilities.

Using Dynamic Table Names for Database Changes

Dynamic Open SQL allows you to use dynamic table names to perform database changes. We illustrate the possibilities with a simple demonstration program (**Listing 8**). Let's assume that Lufthansa will no longer carry out flight LH0455. Instead, the partner airline United Airlines will provide the same connection, but as flight UA3505. Our program shall perform the necessary changes in a travel database.

Listing 8: Update Operations — Using a Dynamic Table Name (ABAP 4.6)

```

1 types: begin of key,
2     mandt type sflight-mandt,
3     carrid type sflight-carrid,
4     connid type sflight-connid,
5     end of key.
6
7 data: tabnames type table of tabname,
8     dref type ref to data.
9
10 field-symbols: <tabname> type tabname,
11                <wa>      type any,
12                <key>     type key.
13
14 append 'SFLIGHT' to tabnames.
15 append 'SPFLI'  to tabnames.
16 append 'SBOOK' to tabnames.
17
18 loop at tabnames assigning <tabname>.
19     create data dref type (<tabname>).
20     assign dref->* to <wa>.
21     assign <wa> to <key> casting.
22     select * from (<tabname>) into <wa>
23         where carrid = 'LH' and connid = '0455'.
24         delete (<tabname>) from <wa>.
25         <key>-carrid = 'UA'.
26         <key>-connid = '3505'.
27         insert (<tabname>) from <wa>.
28     endselect.
29 endloop.

```

First we define a data type key, containing the first three key fields that are common to the tables we want to change:

```
1 types: begin of key,
2     mandt type sflight-mandt,
3     carrid type sflight-carrid,
4     connid type sflight-connid,
5     end of key.
```

We loop over the internal table tabnames, which contains the names of the database tables to be updated. Within the loop, we dynamically create a work area matching the line type of the database table being processed and assign the field symbol <wa> and <key> to this work area:

```
19 create data dref type (<tablename>).
20 assign dref->* to <wa>.
21 assign <wa> to <key> casting.
```

In an inner loop, we select all records from the table being processed that correspond to the flight LH0455:

```
22 select * from (<tablename>) into <wa>
23     where carrid = 'LH' and connid =
24         '0455'.
25 [...]
28 endselect.
```

Because we are going to change key values, we cannot change the record using an UPDATE ... FROM statement. Instead, we have to delete the record, change the key values in the work area <wa> via the field symbols <key>, and reinsert the changed record in a second step:

```
24 delete (<tablename>) from <wa>.
25 <key>-carrid = 'UA'.
26 <key>-connid = '3505'.
27 insert (<tablename>) from <wa>.
```

Analogously, dynamic table names can be used with UPDATE and MODIFY.

New with ABAP 6.10: Using a Dynamic Table with a SET Clause in an UPDATE Statement

With ABAP Release 6.10 it will be possible to use a dynamic table in an UPDATE statement in combination with a SET clause. Using this variant, it will be possible to simplify our program (Listing 8) so that the outer loop (lines 18 to 29) would read:

```
loop at tabnames assigning <tablename>.
  update (<tablename>) set   carrid = 'UA'
                           connid = '3505'
                           where carrid = 'LH'
                           and   connid = '0455'.
endloop.
```

This version of the program is not only much slimmer, it also enables superior performance because the update operations are performed entirely on the database.

New with ABAP 6.10: Using the WHERE and SET Clauses in Update Operations Dynamically

In ABAP releases older than 6.10, there are two ways to denote a set of records on which an update operation is to be applied:

- They can be specified by their primary keys contained in a work area wa using the variant:

```
DELETE | UPDATE | INSERT | MODIFY
(dbtab) FROM wa.
```

- Alternatively, records to be deleted or updated can be selected by a WHERE clause with a static logical condition, as in:

```
DELETE | UPDATE (dbtab) WHERE
logical_condition.
```

For our previous example (Listing 8), these techniques were sufficient. If, however, a set of records to be updated or deleted is defined not by their primary keys but by a dynamically determined logical condition, a dynamic WHERE clause is desirable.

Listing 9: Using the Dynamic WHERE and SET Clauses in Update Operations (ABAP 6.10)

```

1 parameters: tabname    type tabname,
2             column     type fieldname,
3             where(80)  type c lower case,
4             value(30)  type c.
5
6 data: set_clause      type string,
7       value_ref       type ref to data,
8       xref            type ref to cx_sy_dynamic_osql_error,
9       message         type string,
10      count           type string.
11
12 concatenate column ` = value` into set_clause.
13
14 try.
15   update (tabname) set (set_clause) where (where).
16   if sy-dbcnt <> 0.
17     count = sy-dbcnt.
18     concatenate 'The field `` column
19               `` has been set to the new value `` value
20               `` in ` count `row[s] of table `` tabname ``.' into message.
21   else.
22     message = 'No columns changed.'.
23   endif.
24   catch cx_sy_dynamic_osql_error into xref.
25     message = xref->get_text( ).
26 endtry.
27 write: / message.

```

So far, an UPDATE statement with a dynamic table name couldn't change a specified field of an individual row. Instead, an entire row of a database table had to be updated at once. If a database table has many columns, and the values of only a few columns are to be changed, this can lead to a lot of excess data being transported to the database.

With ABAP Release 6.10, the programmer will have the ability to specify a dynamic WHERE clause with an UPDATE or DELETE statement. Addition-

ally, he may specify a SET clause at runtime. Our next example, **Listing 9**,¹¹ illustrates these future capabilities via a generic update tool. The program prompts the user for the name of a database table, the name of a column of this table, and a new value for the selected fields of the specified column. The user may restrict the set of rows to be updated by entering a logical condition (the WHERE clause).

¹¹ Please note that this example is for illustration purposes only. A real-world program would require some authority-checking of the user data, of course.

⚠ **Warning!**

Be aware of the following pitfall when specifying a dynamic WHERE clause with an UPDATE or DELETE statement: if the WHERE clause is empty, it will be interpreted as the Boolean value TRUE with the result that all records on the database get updated or deleted!

First, the dynamic SET clause is constructed from the input parameters:

```
12 concatenate column ` = value` into
    set_clause.
```

Please note that the dynamic SET clause contains the identifier of the ABAP variable value. The critical section of a TRY ... CATCH ... ENDDTRY block contains a fully dynamic UPDATE statement:

```
15 update (tablename) set (set_clause)
    where (where).
```

If the user entered valid data, the update statement will pass through normally. If the statement fails due to invalid input data from the user, an exception of class `cx_sy_dynamic_osql_error` will be raised. The execution will continue in the CATCH clause. The program will display a message informing the user of success or failure.

New with ABAP 6.10: The Dynamic FROM Clause

In our examples so far, all data retrieved was contained in a single database table. This table could be accessed by dynamically specifying its name.

If, however, the relevant data is distributed over multiple tables, one might want to efficiently access the data through a join. Up to ABAP Release 4.6, a join can be expressed only by a static FROM clause. ABAP Release 6.10 introduces a dynamic FROM clause, which allows you to dynamically specify a join of multiple tables (**Listing 10**).

Listing 10: The Dynamic FROM Clause (ABAP 6.10)

```
1 parameters: cities    as checkbox,
2             carriers  as checkbox,
3             seats     as checkbox,
4             names     as checkbox.
5
6 data: begin of wa,
7     carrid type sbook-carrid,
8     connid type sbook-connid,
9     fldate type sbook-fldate,
10    customid type sbook-customid,
11    cityfrom type spfli-cityfrom,
12    cityto type spfli-cityto,
13    carrname type scarr-carrname,
14    seatsmax type sflight-seatsmax,
15    seatsocc type sflight-seatsocc,
16    name type scustom-name,
```

```
17     end of wa,
18     from type string value 'sbook as b',
19     list type string value
20         'b~carrid b~connid b~fldate b~customid'.
21
22 if cities = 'X'.
23     concatenate
24         from ' join spfli as p on b~carrid = p~carrid and b~connid = p~connid'
25         into from.
26     concatenate list ' cityfrom cityto' into list.
27 endif.
28
29 if carriers = 'X'.
30     concatenate
31         from ' join scarr as c on b~carrid = c~carrid' into from.
32     concatenate list ' c~carrname' into list.
33 endif.
34
35 if seats = 'X'.
36     concatenate
37         from ' join sflight as f'
38         ' on b~carrid = f~carrid and b~connid = f~connid and b~fldate = f~fldate'
39         into from.
40     concatenate list ' seatsmax, seatsocc' into list.
41 endif.
42
43 if names = 'X'.
44     concatenate
45         from ' join scustom as u on b~customid = u~id' into from.
46     concatenate list ' name' into list.
47 endif.
48
49 select (list) from (from) into corresponding fields of wa.
50     write: / wa-connid, wa-carrid, wa-fldate, wa-customid.
51     if cities = 'X'.
52         write: wa-cityfrom, wa-cityto.
53     endif.
54     if carriers = 'X'.
55         write: wa-carrname.
56     endif.
57     if seats = 'X'.
58         write: wa-seatsmax, wa-seatsocc.
59     endif.
60     if names = 'X'.
61         write: wa-name.
62     endif.
63 endselect.
```

The example shown in Listing 10 demonstrates a possible application of the dynamic FROM clause: the program is a reporting tool that displays a list of flight bookings. It always lists the following:

- Carrier ID
- Flight number
- Date of the flight
- Customer ID

All this information is extracted from the table SBOOK.

Optionally, the user may choose to have the following additional information displayed:

- Names of the arrival and departure cities (from the table SPFLI)
- The name of the carrier (from the table SCARR)
- The number of occupied and available seats (from the table SFLIGHT)
- The name of the customer (from the table SCUSTOM)

Users simply indicate their choices by marking the appropriate checkboxes on a selection screen.

First, four parameters — *cities*, *carriers*, *seats*, and *names* — are declared that indicate the additional fields the user wants to have displayed. The variables *list* and *from* shall hold the source code of a dynamic SELECT and FROM clause, respectively:

```

1 parameters: cities   as checkbox,
2             carriers as checkbox,
3             seats    as checkbox,
4             names    as checkbox.
[... ]
18 data: from type string value `sbook
      as b`,

```

```

19         list type string value
20         `b~carrid b~connid
          b~fldate b~customid`.

```

The variable *from* gets the initial value ``sbook as b``, because data from the table SBOOK is always selected.

Analogously, we initialize *list* with the names of the columns of SBOOK that we always want to be included in the SELECT list. To prevent the identifiers from becoming ambiguous if multiple tables are joined, we prefix the column identifiers with the table identifier “b~”.

Depending on the values of the parameters *cities*, *carriers*, *seats*, and *names*, the tables SPFLI, SCARR, SFLIGHT, and SCUSTOM are joined with the table SBOOK according to their foreign key dependencies.

Figure 1 shows an overview of the tables of the flight data model and how they are joined.

The additional fields that are to be selected from the dependent tables are added to the SELECT list *list*:

```

22 if cities = 'X'.
23   concatenate
24     from ` join spfli as p on
          b~carrid = p~carrid and
          b~connid = p~connid`
25     into from.
26   concatenate list ` cityfrom
          cityto` into list.
27 endif.
[... ]

```

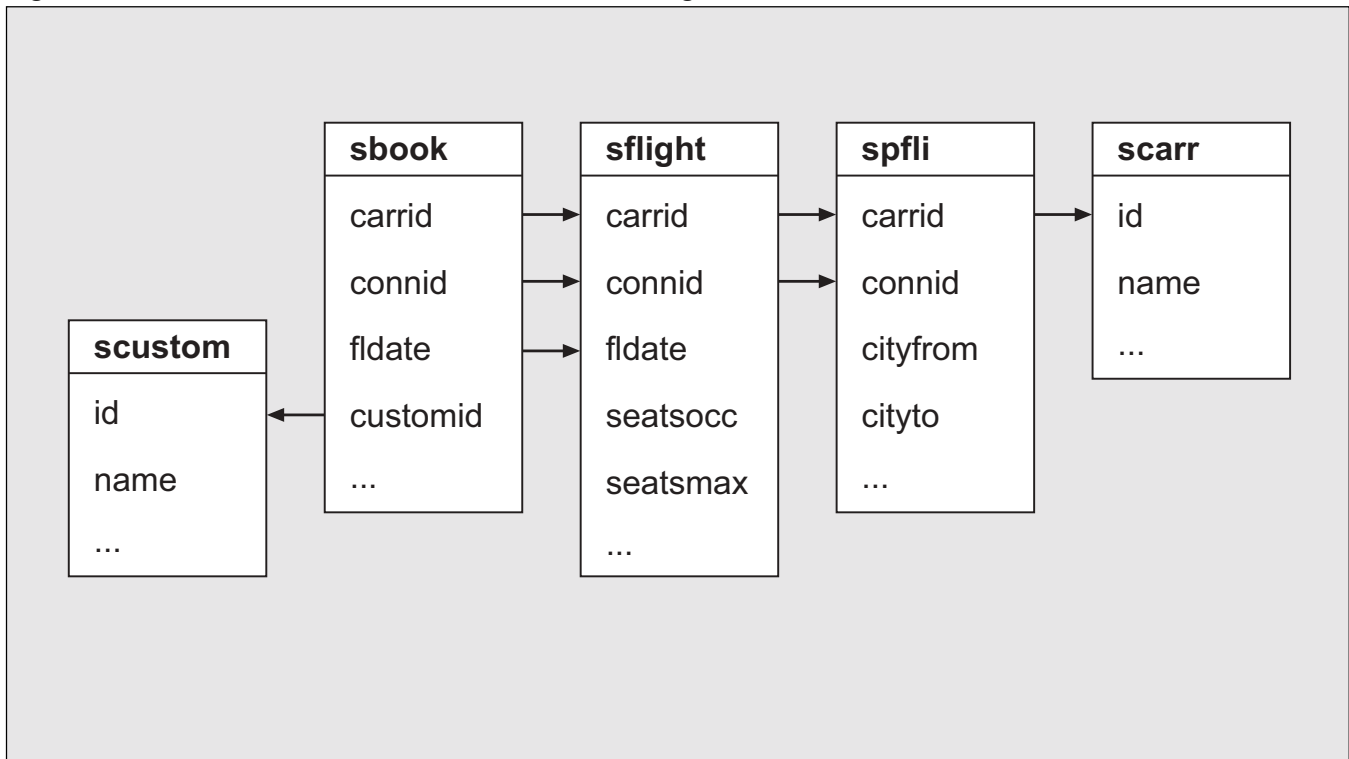
Analogously, the *carriers*, *seats*, and *names* flags are dealt with. If, for example, the user chose “cities” and “names,” the string *from* will have the following value at this point:

```

sbook as b join spfli as p on b~carrid
= p~carrid and b~connid = p~connid join
scustom as u on b~customid = u~id

```

Figure 1 Tables of the Flight Data Model



and the string `list` will have the value:

```
b~carrid b~connid b~fldate b~customid
cityfrom cityto name
```

Next, the fields declared in the string `list` are fetched from the view that is specified in the dynamic clause from:

```
49 select (list) from (from) into
   corresponding fields of wa.
[... ]
63 endselect.
```

The data is stored in the corresponding fields of the work area `wa`. In the `SELECT ... ENDSELECT` loop, the additional fields are written to the list depending on the values of the flags `cities`, `carriers`, `seats`, and `names`.

The use of a dynamic `FROM` clause greatly simplifies the statement, which is sent to the database. When using a static `FROM` clause, all five tables would have to be included in the join, even if no fields are requested from one of the tables.

In our example, all tables and fields possibly required in the `SELECT` statement are known *a priori*. Therefore, the static `INTO` clause

```
INTO CORRESPONDING FIELDS OF wa
```

is suitable to store the data in the corresponding components of the work area `wa` with matching type. Unfortunately, a dynamic `FROM` clause might not be applicable for solving a more general problem where the names of the tables that are to be joined are known only at runtime. The reason is that it is impossible to dynamically create an appropriate work area for a view of multiple tables.

Figure 2 Influence of Dynamic Components on the Execution Time of a Statement

Variant		Linux	NT	AIX
1	<code>select count(*) from sbook where carrid = 'LH'.</code>	2.3 ms	2.4 ms	2.4 ms
2	<code>select count(*) from ('sbook') where carrid = 'LH'.</code>	3.0 ms	2.7 ms	3.2 ms
3	<code>select ('count(*)') from ('sbook') where carrid = 'LH'.</code>	3.7 ms	3.0 ms	3.9 ms
4	<code>select ('count(*)') from ('sbook') where ('carrid = 'LH'').</code>	4.2 ms	3.2 ms	4.5 ms
5	<code>select count(*) from sbook where carrid in rtab.</code>	2.4 ms	2.4 ms	2.4 ms
6	<code>select ('count(*)') from ('sbook') where ('carrid = carr_id').</code>	4.3 ms	3.2 ms	4.5 ms
7	<code>select * from sbook into table itab where carrid = 'LH'.</code>	47.4 ms	45.8 ms	29.1 ms
8	<code>select ('*') from ('sbook') into table itab where ('carrid = 'LH'').</code>	49.8 ms	49.7 ms	31.1 ms
9	<code>generate subroutine pool + select count(*) [...]</code>	31.3 ms	26.3 ms	36.6 ms
10	<code>insert report + generate report + select count(*) [...]</code>	180.3 ms	154.2 ms	140.8 ms

The Linux system was configured with 4 processors and 4 GB RAM.

The NT system was configured with 1 processor and 512 MB RAM.

The AIX system was configured with 1 processor and 4 GB RAM.

Performance Issues

If an SQL statement contains dynamic parts, those parts have to be parsed and evaluated at runtime. Therefore, it is inevitable that the execution time of a dynamic statement is slightly longer than the execution time of a corresponding static variant.

To quantify this impact on performance, we determined the runtime of a simple SELECT statement in eight different variants (**Figure 2**):

```
select count(*) from sbook where carrid
= 'LH'.
```

This statement retrieves only a very small amount of data: the number of records in the table SBOOK fulfilling the condition `carrid = 'LH'`. Therefore,

its total execution time is dominated by the ABAP interpreter and not by the database.

We determined the runtime on three different systems (Linux, NT, and AIX) with the ABAP command `GET RUNTIME`. Each statement has been executed 10 times. Figure 2 shows the minimal execution time of the 10 runs.

The first variant is entirely static. On all platforms, its execution time is about 2.4 ms. Variants 2 to 4 successively add more dynamic elements to the statement: a dynamic table name, a dynamic SELECT clause, and a dynamic WHERE clause. Obviously, adding a dynamic fragment increases the computation time by an almost constant amount. For the three platforms we investigated, this amount is 0.6 ms (Linux), 0.2 ms (NT), and 0.7 ms (AIX).

Because a Dynamic Open SQL statement might change at each execution, it is not cached in the ABAP runtime environment.¹² This fact must contribute to the slightly decreased performance of the dynamic variants (2, 3, and 4) in comparison with the static variant (1). In our example, however, the impact is negligible within the accuracy of our measurement. Variant 5 demonstrates this: the `WHERE` clause contains the condition `IN rtab`, making Variant 5 dynamic as well, so that no cache can be taken advantage of.

As mentioned previously, we expect slightly decreased performance if we include an ABAP variable in a dynamically specified logical condition. We do so in Variant 6, but only on Linux could we observe slightly decreased performance in comparison to Variant 4, where a literal is used as the bound value in the dynamic `WHERE` clause.

In all variants so far, we didn't fetch data from the database, but determined only the number of records in the result set. If we actually retrieve the data (2,220 records), as in Variants 7 and 8, it becomes obvious that the dynamic elements contribute only by a small amount to the overall execution time of the statement. On the platforms we investigated, this overhead is 5 percent (Linux), 9 percent (NT), and 7 percent (AIX).

The situation changes dramatically if we don't use Dynamic Open SQL, but use source code generation instead. In Variant 9, we transiently generated a subroutine pool containing a form that executes the statement corresponding to the static Variant 1. The total execution time for generating the subroutine pool and executing the statement dramatically increases to 31.3 ms (Linux), 26.3 ms (NT), and 36.6 ms (AIX).

The situation gets even worse if we store the generated ABAP report persistently in the database. The total execution time for the sequence `INSERT`

`REPORT`, `GENERATE REPORT`, and `PERFORM ... IN PROGRAM` rises to 180.3 ms (Linux), 154.2 ms (NT), and 140.8 ms (AIX).

Conclusion

Dynamic Open SQL is a very powerful tool — one that is suitable for addressing the majority of programming situations that require generic database access. It is not necessary to generate the ABAP source code of an entire subroutine or program, but only the parts of a statement that contain dynamic elements. This is accomplished by dynamic token specification. All statically specified parts of an SQL statement undergo a static syntax check at compile time.

Most clauses of an Open SQL statement — table name, `SELECT`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` — can be specified dynamically. With ABAP Release 6.10, this list is completed by a dynamic `SET` clause, as well as a `FROM` clause. Though an `INTO` clause cannot be specified dynamically, a data object created at runtime with a dynamically specified type can be used as a container to retrieve data into.

With ABAP Release 6.10, it is possible to catch system exceptions occurring during the execution of a dynamic SQL statement that would cause the program to terminate with a runtime error. This frees programmers from the need to provide code that verifies the validity of input data prior to the execution of the statement. Instead, you redirect the execution to the user to input the correct data.

An Open SQL statement with dynamic components does require slightly more execution time than its static counterpart. Though measurable, this overhead is usually much smaller than the time needed for database access — in fact, it is negligible compared to the huge excess that complete source code generation would require.

¹² It still might be cached by the database, however.

Adrian Görler studied physics at the Ruprecht-Karls-University of Heidelberg, Germany, where he specialized in Computational Biophysics. He received his doctorate at the Max-Planck-Institute for Medical Research, Heidelberg, and did postdoctoral research work in various international laboratories.

In 1999, Adrian joined SAP and became a member of the Business Programming Languages Group, where he is working as a kernel developer. He is responsible for the implementation and maintenance of Open SQL as well as Native SQL in the ABAP compiler and interpreter. One focus of his work during the last year was the realization of the new features in Dynamic SQL.

He can be reached at adrian.goerler@sap.com.

Ulrich Koch received his doctorate in mathematics at the Westfälische-Wilhelms-Universität of Münster, Germany. In 1990, he joined SAP, where he became a member of the Business Programming Languages Group.

As a Development Architect, currently Ulrich is mainly responsible for the external interfaces of the ABAP language. This includes in particular the design of semantics for and implementation of the set of commands that form the Open SQL and Native SQL parts of ABAP. Other special areas he works in include ABAP transaction handling and transparent object persistence.

He can be reached at u.koch@sap.com.