

Creating an SQL Access Layer

The following topics are covered below:

- General Information
 - Different Database Accesses with Adabas C and SQL
 - Creating SQL Tables and DDMs
 - Access to SQL Tables
-

General Information

Encapsulating the Database Accesses

Various database systems can serve as a basis for holding the data of an information object.

This is made easier by encapsulating the database accesses. Then it is only necessary to create a new access layer for each target database.

Access to an "object type" by the application remains as described in the preceding sections of this documentation. Modifications to the application programs are not necessary.

Creating an Access Layer

To create an access layer for an SQL database, for example, Adabas D, suggested codes are available. They are handled similarly to the suggested codes for the Adabas C access layer.

To support the largest possible number of SQL databases, the lowest common denominator must be found, that is, field lengths must be adjusted to the target platform with the most limitations. When SQL statements are used, they must not use database specific syntax features.

Definition of the Access Layer

The following sections describe the definition of the access layer from the viewpoint of redoing an existing Adabas C access layer.

They also contain notes on implementing a new system.

Different Database Accesses with Adabas C and SQL

Accesses to an SQL database differ in some ways from accesses to Adabas C data:

- Most differences are found primarily in the read loops which return multiple records.
- Single accesses can be reshaped easily.

These differences require a new access layer which can be largely derived from the Adabas C access layer.

The following topics are covered below:

- Converting a Sequential Read Access
- Converting a Single Access
- Creating Read Accesses
- Access using a Key with Several Components
- Inserting a New Record

Converting a Sequential Read Access

The conversion of a sequential read access involves use of the SQL ORDER BY statement.

Natural DML (Data Manipulation Language):

```

*
  RLI. READ <view name> BY <key> STARTING FROM <#start key>
*
    <further processing>
*
  END-READ

```

SQL Syntax:

```

*
  RLI. SELECT ALL * INTO VIEW <view name> FROM <table name>
      WHERE <key> >= <#comparison key>
      ORDER BY <key> ASC,
*
    <further processing>
*
  END-SELECT
*

```

Converting a Single Access

The counterpart of a single access to an object can be as follows:

Natural DML:

```

*
  FRE. FIND <view name> WITH <key> = <#start key>
*
    <further processing>
*
  END-FIND
*
  IF *NUMBER(FRE.) GT 0
    MOVE 1 TO PZ_AS_REC_EXIST /* record exists
  ELSE
    MOVE 1 TO PZ_AS_RSP      /* see PZ_AS_REC_EXIST
  END-IF
*

```

SQL Syntax:

```
*
  FRE. SELECT SINGLE * INTO VIEW <view name> FROM <table name>
        WHERE <key> = <#comparison key>
*
  END-SELECT
*
  IF *COUNTER(FRE.) GT 0
*
    <further processing>
    MOVE 1 TO PZ_AS_REC_EXIST /* record exists
  ELSE
    MOVE 1 TO PZ_AS_RSP      /* see PZ_AS_REC_EXIST
  END-IF
*
```

Creating Read Accesses

When creating the read access, you must ensure that, Adabas C-specific elements are converted.

Since periodic groups or multiple fields are not possible in SQL, these elements must be represented by corresponding division into tables and DDMs. The database design of a relational database has a flat structure, which means that the number of tables and relations increases in comparison to that of a design using Adabas C features.

See the following example for an illustration of what is involved.

- Language-dependent fields with several occurrences that are defined as keys are maintained in their own sub-table.
- Access to this sub-table is carried out through a language-dependent key which consists of a main key and an additional language identification.
- The language-independent information is read from the accompanying main table using the corresponding main key.

Inserting a New Record

The insertion of a new record is created using the available DDM and table fields. The individual fields of the DDM are transferred into the table by the appropriate SQL syntax .

```
*  
*   Store record  
*  
   INSERT INTO <table name> (  
       <field-1> ,  
       <field-2> ,  
       <field-3> )  
   VALUES (  
       <view name.field-1> ,  
       <view name.field-2> ,  
       <view name.field-3> )
```

Creating SQL Tables and DDMs

The communication of the access layer with the database uses DDMs even with SQL database systems.

The interface must be modified according to the target database.

When creating the tables and DDMs, the limitations of the target systems must be accommodated. These are:

- no multiple fields or periodic groups;
- no definition of super-/sub-descriptors;
- limitations in the field lengths differ from Adabas C.

The limitations lead to different DDMs. The following sections describe some of the possible modifications.

- Short Fields with Occurrences
- Long Fields with Occurrences
- Multiple Fields that are Descriptors
- Converting Formats
- Example of an Unsupported Field Format Conversion
- Defining Tables

Short Fields with Occurrences

Multiple fields that are not part of a descriptor and are shorter than 251 bytes are converted into individual fields with redefinitions.

DDM for Adabas C

```
V 1 <view name>
M 2 xxx_LNAME_LC A 16 (1:9)
R 2 xxx_LNAME_LC
  3 xxx_LNAME_LC_G (1:9)
  4 xxx_NAME_LC_LONG N 1
  4 xxx_NAME_LC A 15
```

DDM for SQL Databases with Longer Redefinition

```
V 1 <view name>
  2 xxx_LNAME_LC_R A 144
R 2 xxx_LNAME_LC_R
  3 xxx_LNAME_LC A 16 (1:9)
R 3 xxx_LNAME_LC
  4 xxx_LNAME_LC_G (1:9)
  5 xxx_NAME_LC_LONG N 1
  5 xxx_NAME_LC A 15
```

This type of DDM is advantageous in that only a few fields must be defined.

Dialogs and programs access the variables via redefinition. Access to the SQL data is through the unredefined whole fields.

DDM for SQL Databases with Individual Fields Numbered

```

V  1 <view name>
   2 xxx_LNAME_LC_1  A  16
R  2 xxx_LNAME_LC_1
   3 xxx_LNAME_LC_G_1
   4 xxx_NAME_LC_LONG_1  N  1
   4 xxx_NAME_LC_1  A  15
    
```

This form of definition in a DDM has the disadvantage that many fields must be defined.

Access to all variables and SQL data is through the individual fields. The redefinitions here allow access to individual components.

Long Fields with Occurrences

Multiple fields or periodic groups that are not part of a descriptor but, including all occurrences, are longer than 251 bytes, can be converted into large individual fields with the redefinition of the group.

The allocation of the variables is through a redefinition.

The access to the SQL data is through the individual whole fields.

DDM for Adabas C

```

V  1 <view name>
P  2 xxx_field  A  80  (1:9)
    
```

DDM for SQL Databases

```

G  2 xxx_field_G
   3 xxx_field_1 A  240
   3 xxx_field_2 A  240
   3 xxx_field_3 A  24
R  2 xxx_field_G
   3 xxx_field  A  80  (1:9)
    
```

When processing multiple fields, conversion must be carried out in the same way.

To remove blank entries within the array, an appropriate routine must be written.

Multiple Fields that are Descriptors

Searching within a multiple field is possible with Adabas C. The SQL-specific conversion must then convert the multiple field by defining a main table and a sub table.

A second DDM consists of the fields via which a read access is possible.

Redundant holding of data is avoided since the information in the first DDM contains only the non-descriptor components. If a record is read through the multiple field, the data of the first view of the main table must be read through the common main key.

DDM for Adabas C

```

V  1 <view name>
    2 xxx_ID A    4
    2 xxx_FIELD1 A   20
    <further fields>
*
M  2 xxx_LNAME    A   16   (1:9) /* Descriptor
R  2 xxx_LNAME
    3 xxx_LNAME_G      (1:9)
    4 xxx_NAME_LONG   N    1
    4 xxx_NAME        A   15
    
```

DDM for SQL Databases

```

V  1 <view name>
    2 xxx_ID A    4
    2 xxx_FIELD1 A   20
    <additional fields>

V  1 <view name 1>
    2 xxx_ID A    4
    2 xxx_LONG      N    1
    2 xxx_NAME      A   15
    
```

Converting Formats

Some formats must be converted into a corresponding SQL target database format. Here again, a common denominator must be used.

The conversion of unsupported field formats is described in the section below.

Example of an Unsupported Field Format Conversion

In the following example, the conversion of a field not mappable in the length is performed.

In general, fields defined as numeric or integer are less problematic in a client/server environment than fields that are either packed or binary.

DDM for Adabas C

```

1 xxx_ID P    20
    
```

DDM for SQL Databases

```

R  1 xxx_ID_R      A   20
    1 xxx_ID_R
    1 xxx_ID N    20
    
```

Definition in the SQL Table

```
xxx_ID CHAR(20)
```

In all instances in which mapping is possible, a conversion to a simple format is carried out.

DDM for Adabas C

```
1 xxx_ID P 12
```

DDM for an SQL Database

```
1 xxx_ID N 12
```

SQL Table

```
xxx_ID DEC(12)
```

Defining Tables

How is a Table Created with DDMs?

The DDMs must be rewritten in a corresponding SQL database table definition.

The new definitions created are entered as SQL tables in the target database.

The conversion of unsupported formats depends on the target database.

The definition of the individual SQL table matches the DDMs that were converted for an SQL database.

DDM for Adabas C

```
V 1 <view name>
  2 xxx_ID A 4
  2 xxx_FIELD1 A 20
  <further fields>
*
M 2 xxx_LNAME A 16 (1:9) /* Descriptor
R 2 xxx_LNAME
  3 xxx_LNAME_G (1:9)
  4 xxx_NAME_LONG N 1
  4 xxx_NAME A 15
```

DDM for SQL Databases

```

V 1 <view name>
  2 xxx_ID A 4
  2 xxx_FIELD1 A 20
  <further fields>

V 1 <viewl>
  2 xxx_LANGUAGE_ID A 4
  2 xxx_LANG N 1
  2 xxx_NAME A 15

```

SQL Table

```

CREATE TABLE <table name>
(
  xxx_ID CHAR(4),
  xxx_FIELD1 CHAR(20),
  <further fields>
)

CREATE INDEX <table name>.xxx_CODE

CREATE TABLE <table language>
(
  xxx_ID CHAR(4),
  xxx_LANG DEC(1),
  xxx_NAME CHAR(15)
)

CREATE INDEX xxx_LNAME ON <table language> (xxx_LANG, xxx_NAME)

```

Access to SQL Tables

The following topics are covered below:

- Modifications of a Record
- Modifications of Individual Fields
- Optimizing Accesses
- Application of System Variables
- Allocation of Variables

Modifications of a Record

For the modification of an entire record, the field list in the form of the whole view is passed. The field list contains the complete record from the view.

The modification of the record is carried out with a Searched update.

Natural DML

```
FUP. FIND <view name> WITH <xxx_key> = <xxx_key_FROM>
      MOVE <#xxx_fields> TO <view name.xxx_fields>
      MOVE .....
      UPDATE
END-FIND
```

SQL Syntax

```
FUP. SELECT ALL COUNT INTO LZ_COUNT FROM <table name>
      WHERE <xxx_key> = <xxx_key_FROM>
END-SELECT
*
IF LZ_COUNT GT 0
  MOVE 1 TO PZ_AS_REC_EXIST /* record exists
  MOVE <#xxx_fields> TO <view name.xxx_fields>
  MOVE .....
  UPDATE <view name> SET *
  WHERE <xxx_key>      = <xxx_key_FROM>
*
ELSE
  MOVE 1 TO PZ_AS_RSP      /* see PZ_AS_REC_EXIST
END-IF
```

Modifications of Individual Fields

The modifications of individual fields are also carried out with this read access. The field list, however, does not contain the entire record, but the fields that are actually modified.

SQL

```
UPDATE <table name>
      SET <xxx_field1> = <view name>.<xxx_field1>
      WHERE <xxx_key>   = <xxx_key_FROM>
```

Example: Dating

```

UPDATE <table name>
  SET <xxx_EFD_INV>      = RUP.<xxx_EFD_INV>      /* modified value of the view
WHERE <xxx_ID>          = RUP.<xxx_ID>          /* key value found
  AND <xxx_EFD_INV>     = <#EFD_OLD>          /* date of record read

```

Optimizing Accesses

Using SQL syntaxes enables transparent access to the data.

The exact handling of the individual key components and the conversion of data types are easier to apply with embedded SQL statements.

The optimization is thereby more in sync on the target database system. When using SQL syntax, however, the common denominator - i.e. using ANSI/ISO-SQL - must be taken into consideration.

The optimization of the accesses is influenced by the:

- design of the tables;
- definition of the primary key and of indices;
- read accesses using SQL syntax.

For the individual accesses and table creation, database-specific optimizations must be taken into consideration.

In the following example, only one variant of an SQL-specific optimization is presented.

Table Definition

```

CREATE TABLE <table name>
(
  xxx_CLIENT_ID          CHAR( 2) ,
  xxx_ID                 CHAR(12) ,
  xxx_NAME               CHAR(30)

  PRIMARY KEY (xxx_CLIENT_ID, xxx_ID)
)

```

Access to the Table

```

SELECT ALL * INTO VIEW <view name>
  FROM <table name>
  WHERE   <xxx_CLIENT_ID>      >   <#xxx_CLIENT_ID> OR
         <xxx_CLIENT_ID>      =   <#xxx_CLIENT_ID>
        AND <xxx_ID>          >=  <#xxx_ID>
*
  ORDER BY <xxx_CLIENT_ID>  ASC ,
         <xxx_ID>          ASC

```

In this case the access uses the primary key of the table which provides the best performance. With the corresponding number of indices that are created, further sorting sequences are mapped.

```

CREATE INDEX xxx_KEY ON <table name> (xxx_CLIENT_ID, xxx_ID, xxx_NAME)

```

Specifying the sorting sequence in the ORDER clause (in the above example) causes the SQL database system to use the primary key as the preferred access path.

This access can be optimized, when only a minimum of OR concatenations is used in the WHERE clause. As a result of a logical OR concatenation, the SQL database creates two internal lists, which must be validated against each other. This causes poor performance during multi-record accesses.

The optimized SELECT statement runs as follows:

```
SELECT ALL * INTO VIEW <view name>
  FROM <table name>

  WHERE    <xxx_CLIENT_ID>    =    <#xxx_CLIENT_ID>
    AND    <xxx_ID>          >=    <#xxx_ID>
*
  ORDER BY <xxx_CLIENT_ID>    ASC ,
          <xxx_ID>          ASC
```

Application of System Variables

The application of system variables within the access layer is also modified. The available variables are limited by the use of SQL statements.

The variable *COUNTER can be used for checks. *NUMBER is no longer accessed. The variable *ISN is not available.

Natural DML

```
FRE. FIND <view name> WITH <key> = <#start key>
  .....
END-FIND
*
IF *NUMBER(FRE.) GT 0
  .....
END-IF
```

SQL Syntax

```
FRE. SELECT ALL * INTO VIEW <view name>
  FROM <table name>
  WHERE <key> = <#comparison key>
  .....
END-SELECT
*
IF *COUNTER(FRE.) GT 0
  .....
END-IF
```

The number of records is established using another access. For this, a target variable is to be defined, into which the value is put.

```

DEFINE DATA LOCAL
.....
01 LZ_COUNT (I4)
END-DEFINE
*
FUP. SELECT ALL COUNT INTO LZ_COUNT FROM <table name>
      WHERE <xxx_key> = <xxx_key_FROM>
END-SELECT
*
IF LZ_COUNT GT 0
.....
END-IF

```

Allocation of Variables

The allocation of the values from the DDM to the parameter variables of the access module can, in most cases, be implemented by a MOVE BY NAME statement.

If, with the transfer, a format or length is modified, the value must be set individually.

If, in the access layer, a conversion into several individual fields is required, these must be set.

It is also possible to define, in the DDM, a group over the individual fields. A redefinition in the corresponding variables with occurrences is thus possible in the local data area.

The transfer can then be carried out with a MOVE BY NAME. In the parameter data area, the definition then matches the view.

SQL View

V	1	<view name>		
G	2	xxx_field_G		
	3	xxx_field_1	A	240
	3	xxx_field_2	A	240
	3	xxx_field_3	A	240
R	2	xxx_field_G		
	3	xxx_field	A	80 (1:9)

Parameter Data Area Definition

	1	<PDA group name>		
	2	xxx_field	A	80 (1:9)