Adabas SQL Server
Programmer’s Guide

SOFTWARE AG

Manual Order Number: ESQ143-020ALL

This document applies to Adabas SQL Server Version 1.4 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Readers’ comments are welcomed. Comments may be addressed to the Documentation Department at the address on the back cover or to the following e-mail address:

Documentation@softwareag.com

TABLE OF CONTENTS

PREFACE

Using This Manual — Some Basic Information

This manual describes the considerations necessary to successfully program SQL applications, both in general and detailed per host language supported. Its intended audience are application programmers with a basic knowledge of the concepts and facilities of Standard SQL and Adabas C (in the following, the term Adabas refers to Adabas C).

Chapter 1	describes the elements and basic functions of Adabas SQL Server.
Chapter 2	describes how Adabas SQL Server data structures are organized.
Chapter 3	describes the concepts of the Adabas SQL Server security features.
Chapter 4	describes the concepts of SQL programming, such as declaration sections, comments, positioning of statements and transaction logic.
Chapter 5	describes the specifics of static SQL programs.
Chapter 6	describes the specifics of dynamic SQL programs.
Chapter 7	describes the client/server architecture of Adabas SQL Server.
Chapter 8	describes the process of translating SQL statements into procedural form, and generating Adabas commands for retrieval.
Chapter 9	describes the use of the MULTIFETCH feature which is available in order to minimize the data transfer and the interprocess communication between the application, Adabas SQL Server and Adabas.
Chapter 10	describes the rules which apply when embedding SQL statements in the host language C.
Chapter 11	describes the rules which apply when embedding SQL statements in the host language COBOL.
Chapter 12	describes the rules which apply when embedding SQL statements in the host language PL/I.
Chapter 13	describes DB2 transaction mode.

Appendix A	describes the tables (without multiple-value fields and/or periodic groups) used in examples throughout the manual.
Appendix B	contains the sample programs which are used to create the tables (without multiple-value fields and/or periodic groups) used in examples throughout the manual (the same logic is realized in C, COBOL and PL/I).
Appendix C	describes, in detail, the structure of the Adabas SQL Server catalog.
Appendix D	describes special issues occurring when Adabas SQL Server interacts with other Software AG Products.

Other Helpful Manuals

Other manuals you may need are:

– Adabas SQL Server Reference Manual
– Adabas SQL Server Installation and Operations Manual (platform-specific)
– Adabas SQL Server Messages and Codes
– a set of platform-specific Adabas Manuals
– a set of platform-specific Entire Net-Work Manuals
– Entire Broker Reference Manual
– ANSI/ISO Standards SQL (X3.135-1989/X3.168–1989, ISO/IEC 9075).

1

INTRODUCTION TO ADABAS SQL SERVER

Functionality

The product Adabas SQL Server is Software AG's implementation of the ANSI/ISO Standard for the SQL database language. Adabas SQL Server provides an SQL interface to Software AG's database management system Adabas. Software AG is committed to making Adabas SQL Server available in most hardware and operating system environments where Adabas itself is available. Furthermore, the core functionality of Adabas SQL Server will be identical across platforms.

The Adabas SQL Utilities enhance the system by offering a basic as well as a comfortable Interactive SQL Facility and Catalog Retrieval Services.

The SQL statements supported in version 1.4 are in compliance with the ANSI/ISO Standard SQL (X3.135-1989/X3.168–1989, ISO/IEC 9075).

Main Adabas SQL Server Functionality

As already stated, Adabas SQL Server provides an SQL interface for the Adabas database management system. Adabas SQL Server offers three different ways to access and manipulate data:

– Adabas SQL Server supports either static or dynamic SQL statements, embedded in a variety of 3rd generation host languages.
– Statements can be submitted using an interactive interface.
– Statements can be embedded in a Natural application program.

Currently Adabas SQL Server supports SQL statements embedded in either C, COBOL or PL/I. Due to the modular design of Adabas SQL Server, the functionality is identical regardless of the host language chosen.

SQL – Database Query Language

Structured Query Language (SQL) is designed to enable the definition of, the access to, and the control of data in a DBMS. It has been laid down as an industry standard by the ANSI and ISO committees and has been adopted by numerous DBMS suppliers. It must be pointed out, however, that many of the implementations of SQL do in fact diverge from the ANSI/ISO standard.

SQL can be split into three main areas:

DML	Data Manipulation Language is the means by which data is accessed and amended.
DDL	Data Definition Language defines the various data structures which can be accessed.
DCL	Data Control Language defines the privileges in regard to users, particular structures and operations.

SQL statements or commands can be submitted to Adabas SQL Server either interactively or as part of an application program, in which case they are referred to as embedded.

Interactive SQL

Many SQL statements can be entered at the terminal directly and results will be displayed on the screen immediately. Interactive SQL is designed for ad hoc queries, submitted by the end user. Interactive SQL is in fact a special application using the facilities of dynamic SQL.

Embedded SQL

SQL statements can be used from within 3GL Programs and must be incorporated in the program for this purpose. This procedure is commonly called embedded SQL. Embedded SQL statements must always be prefixed with the delimiter EXEC SQL and are generally suffixed by the delimiter END EXEC. A host program with embedded SQL statements must be submitted to Adabas SQL Server precompiler before it can be compiled by the respective host language compiler and be executed.

Static SQL

Static SQL refers to embedded SQL statements which can not be changed or modified after being processed by Adabas SQL Server precompiler. Static SQL statements are coded at the same time as the host program and are fixed thereafter.

Dynamic SQL

Dynamic SQL refers to statements which are dynamically constructed during runtime. The dynamic SQL statements themselves are not directly embedded in the host language program but are submitted to Adabas SQL Server compiler by special embedded SQL statements like PREPARE. Using dynamic SQL offers the opportunity of creating highly flexible user interfaces with minimal coding efforts.

Compiler Options

SQL comes in many different dialects. Software AG’s implementation was developed taking into consideration that in certain areas extra functionality, over and above the ANSI/ISO standard is required. Therefore, useful extensions not provided for by the standard were implemented; particularly to take full advantage of the functionality offered by Adabas.

It is important for the user to have available the required dialect of SQL to ensure that any existing applications can be ported to Adabas SQL Server with as little effort as possible. Compilation only takes place with reference to the current compiler options, which are set either as a default for the SQL environment or by a particular user.

It should be noted that Adabas SQL Server does not permit the mixing of functionality contained in different SQL variations. Indeed the user must actively decide upon which variation is required for a particular application program. The three compatibility modes are:

- **ANSI SQL mode**
  Adabas SQL Server supports standard SQL as described in the ANSI/ISO Standards SQL (X3.135-1989/X3.168–1989, ISO/IEC 9075).
- **DB2 transaction mode**
  Adabas SQL Server supports standard ANSI SQL and automatically issues transaction–control statements (COMMIT. ROLLBACK) for a CICS environment. No DB2-specific SQL statements are supported.
- **Adabas extensions mode**
  Adabas SQL Server also supports extended functionality of its own. In this mode, all extensions are allowed.

The *ADABAS SQL Server Reference Manual* states clearly whether a statement contains extensions or sticks strictly to the ANSI SQL Standard. Divergences will always be brought to the attention of the user as either a warning or as an error, depending on the current compiler option settings.

In addition, the use of DDL/DCL statements can be controlled as well as the ability to mix such statements with DML ones, within a compilation unit.

For further details, refer to the description of the COMPILER MODE parameter in the Parameter Processing Language in the platform-specific *Adabas SQL Server Installation and Operations Manuals.*

How Adabas SQL Server Functions

The development of an application program can be split up into three phases:

– The precompilation phase.
– The host language compilation and link phase.
– The application execution phase.

The Precompilation Phase

The developer of a 3GL application program must create a source file which contains the host language program with embedded SQL statements. Such a program can not be passed directly to the host language compiler because the embedded SQL statements are not part of the host language. Therefore, prior to host language compilation, the Adabas SQL Server precompiler must operate on the source file.

The precompiler is responsible for parsing the source file, extracting all SQL statements and compiling them. It is in fact a three phase precompiler:

- In the first phase the SQL statements are collected and stored in a host-language-independent way. The host language concerned must be known and all necessary host variable information must be collected. The validity of the SQL statements is not checked during this first phase of precompilation.
- During the second phase, the collected SQL statements are compiled. The original host language is irrelevant in this phase. The precompiler checks all statements for validity and if errors occur, logs these and eventually presents them to the user. Assuming there are no errors a host language generation directive is generally produced for each SQL statement.
- Once all SQL statements have been successfully compiled, the third phase takes control. In this phase the original source is copied into another file called generated file and all SQL statements are commented out. In their place the appropriate host language calls are generated, based on the host language generation directive for the statement. The generated file which contains a mix of host language code and SQL statements is thus transformed into a file which contains only host language code.

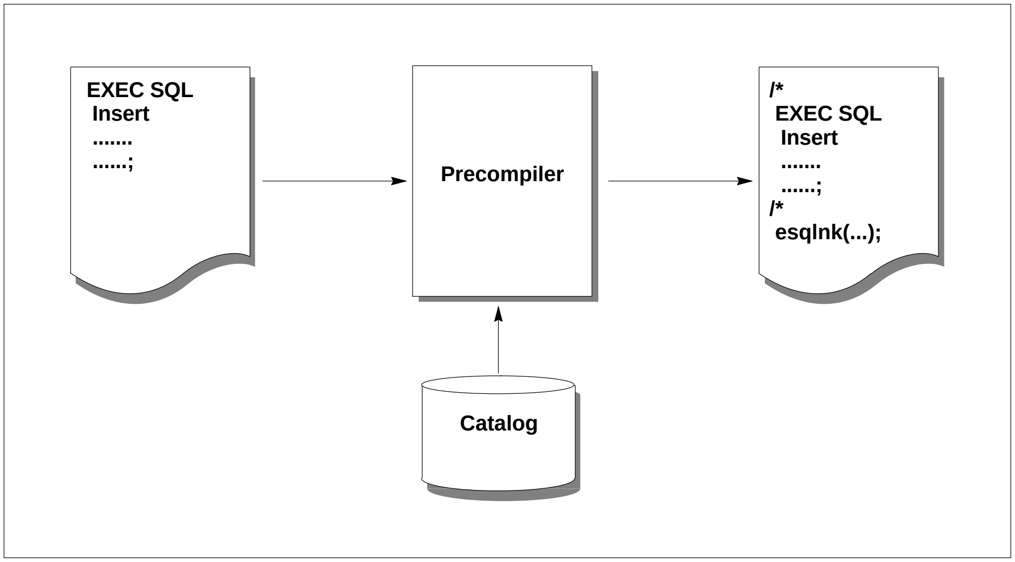


Figure 1-1: The Precompilation Phase

Reference was made earlier to the catalog. This is an Adabas file which is essential for Adabas SQL Server because it contains information on the tables, views and databases accessible by this installation. This information is required during the compilation of SQL statements and so all references to any tables or views are fully checked and resolved at compilation time. No further checking is needed unless the view or table definitions have changed.

Host Language Compilation Phase

Assuming no errors were discovered during the precompilation phase, the generated file must now be submitted to the appropriate host language compiler and, assuming no errors are discovered during its compilation, linked with the necessary Adabas SQL Server client communication modules. These vary from platform to platform but jobs or procedures are provided to facilitate all of these tasks. The application is now ready for execution.

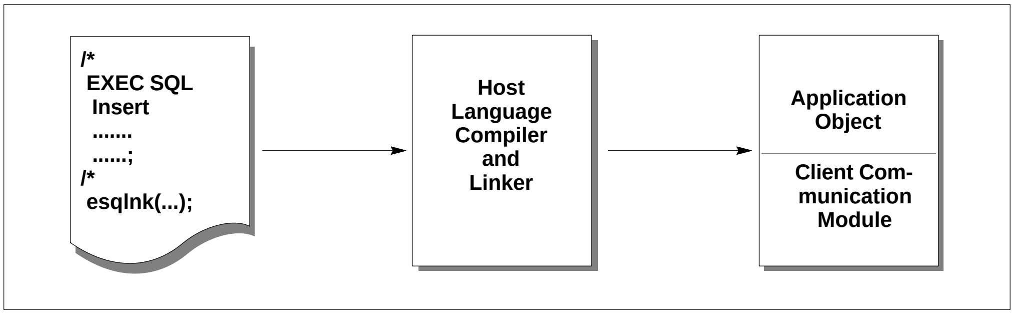


Figure 1-2: The Host Language Compilation Phase

Executing the Application Program

The application program may now be executed. Eventually the program will encounter some generated code and so control passes to the Adabas SQL Server Runtime system (RTS). The RTS receives enough information to be able to retrieve the appropriate meta program from the catalog.

However, if the statement has never been executed before, then the meta program will not be present in the catalog. In such a case, information representing the statement is obtained automatically from the application program and is used to generate a meta program, which is subsequently stored in the catalog. This meta program is then retrieved whenever the statement is to be executed again.

In order to minimize database access, all information kept in the catalog including meta programs is buffered in an incore buffer. Meta programs are also re-entrant, meaning that many users executing the same application program only need one copy of a meta program.

The meta program describes the steps needed to interface with Adabas and to execute the original SQL statement. There may be numerous steps involving many Adabas calls before the meta program will return control to the application program. But, before this is done, the SQL communication area (SQLCA) is loaded as appropriate and any host variables are assigned their values.

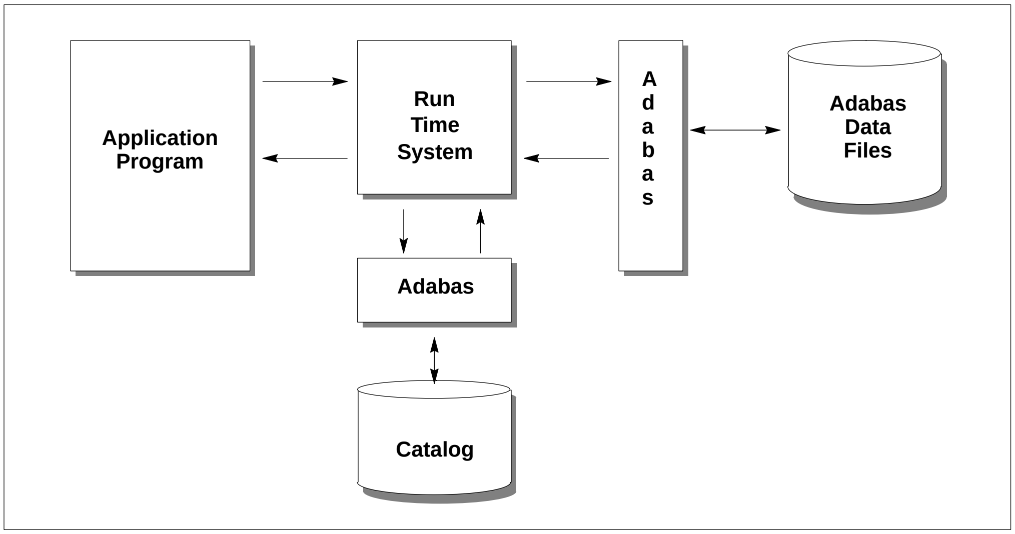


Figure 1-3: The Execution Phase

2

ADABAS SQL SERVER DATA STRUCTURES

Data Organization

Catalog

Adabas SQL Server needs to keep information about all the data structures known to the SQL environment. The catalog which contains all the necessary information about tables, views etc. is maintained by Adabas SQL Server for this purpose. All relevant information about an object must, in general, be present in order to compile statements based on that object. During compilation the validity of a particular statement will be determined based upon the information contained in the catalog.

In a future release of Predict, Software AG's data dictionary, there will be an interface to the catalog enabling the use of dictionary and case tools within the SQL environment.

In addition to various database objects, so called meta programs also reside in the catalog. A meta program is the internal representation of an SQL statement. In the case of statically invoked statements and persistent dynamic statements, meta programs are also stored in the catalog. Any change to the catalog resident objects, is automatically reflected in all dependent meta programs.

The catalog resides in one Adabas database, however, objects described in the catalog may reside in other Adabas databases. There is only one catalog for an Adabas SQL Server.

The catalog must always be accessed using Adabas SQL Server. Attempts to access the catalog using Adabas direct calls could lead to unpredictable results.

Schema

The CREATE SCHEMA statement is used to create a logical container within the catalog. This container is used to group together subsequently created catalog resident objects.

These objects are:

- table clusters (and their descriptions),
- tables (and their descriptions),
- tablespaces,
- indexes,
- privileges,
- constraints and
- views.

All of these objects may only be created within a schema.

Each schema must have a unique identifier within the catalog.

The schema, and the objects contained in the schema, are considered to have an owner. This owner is specified by the resulting authorization identifier established during the execution of the CREATE SCHEMA statement. Only the owner of the schema may subsequently create or destroy objects within the schema. Furthermore, only the owner can initially grant access privileges to objects contained within the schema.

Therefore, a schema is a collection of data structures and objects defined by a set of DDL statements with a set of associated privileges defined by a number of DCL statements.

Only the designated Database Administrator (DBA) can issue CREATE or DROP SCHEMA statements on behalf of the owner.

Tables And Table Clusters

Actual data is contained in tables. A table consists of zero or more rows and of one or more columns. The table is populated with data, i.e. rows are added by the INSERT statement. Tables which actually physically exist are called base tables. An example of a table which does not physically exists is a viewed table which is an extract of one or more tables. Master tables, subtables or tables with rotated fields are also considered to be base tables.

A cluster (of tables) is an Adabas SQL Server extension, which enables a complex Adabas file structure to be mapped to various associated SQL tables.

An Adabas file without multiple value fields (MU) or periodic groups (PE) is generally represented in one base table. Additionally, if an MU or PE field has a fixed number of occurrences, then it can be represented in a base table. The field is said to be 'rotated'. This is where each occurrence is represented by an explicit SQL column.

An Adabas file containing either MUs and/or PEs, where rotation (see below) is not appropriate, is generally represented in a table cluster. A table cluster consists of one master table and generally one or more subtables.

It is permitted to define an Adabas file without MUs and/or PEs as a table cluster. This will result in only one master table.

There are some restrictions which apply when amending subtables. Explicit deletion of rows in a subtable is not permitted. For details of the restrictions regarding the UPDATE and INSERT statements refer to the *Adabas SQL Server Reference Manual*, section **ROW AMENDMENT Expression or** the statement information itself.

The execution of a CREATE TABLE statement or a CREATE CLUSTER statement results in the physical creation of an Adabas file and the insertion of a table description(s) in the catalog. A table description is Adabas SQL Server's internal mapping of the Adabas file in terms of the SQL table. Should the Adabas file already exist, then by executing either a CREATE TABLE DESCRIPTION statement or a CREATE CLUSTER DESCRIPTION statement the file will be introduced to the SQL environment. In such a case, only the description (s) is inserted into the catalog.

Table Levels

Master tables are always considered to be level '0' tables. Directly dependent subtables, representing Adabas MU and/or PE fields are always considered to be level '1' tables. Subtables representing MU fields within a periodic group are always considered to be level '2' tables. The ruling logic is that one or more lower level tables always reference only one higher level table within a cluster.

Master Tables And Subtables

The relationship between a master table and its subtables, in SQL terms, is defined using a primary key/foreign key relationship. As is compatible with the relational model, a foreign key must be equal in value to a unique occurrence of the primary key in the referenced table.

In a cluster, the foreign key and the primary key are actually physically the same Adabas fields. In the SQL environment they are perceived to be distinct column sets. In Adabas terms, this relationship is simply expressed by the existence of :

– an MU or a PE within a particular record (for a level 1 table) or
– an MU within a particular PE (for a level 2 table) which in turn must be within a particular record.

The definition of the primary/foreign key within a cluster for a level 1 table must therefore include sufficient columns so that a particular Adabas record can be uniquely identified, in which the subtable row resides. For a level 2 table, the definition of the key must include sufficient columns of the level 1 table so that not only can the Adabas record be uniquely identified, but also the level 1 PE occurrence within that record can be uniquely identified in which the level 2 row resides.

The significance of this is that a particular row, or set of rows of a subtable, can only exist within an existent master table row and referential integrity is therefore, guaranteed.

Tables with Rotated Fields

If an MU field semantically has a fixed number of occurrences, then this field can be ’rotated’. This means that each occurrence within the field is mapped to an explicit SQL column. For example, if an MU field has 12 occurrences, each representing a month of the year, then 12 separate columns would be defined (January through December) corresponding to each occurrence. Such a table with rotated fields is an alternative to a clustered representation and does not have to be defined within a cluster description.

Columns/Indexes

If one or more columns are defined to be unique, then each row is by definition distinct from any other in the table. Any such column may be considered as a candidate key for the table as a maximum or one row would be identified if a search were initiated over the table where the candidate key column equals a particular value.

A particular column, or indeed group of columns, may be defined as an index. This is intended for better performance upon accessing the table. An index need not necessarily be unique.

SEQNO Columns

In addition to the defined columns, each row of a base table always contains a SEQNO value. Such values can either be accessed using the SEQNO special register, or can be explicitly named as a column. In case of a level 0 table the SEQNO(0) value holds the Adabas ISN otherwise it is the occurrence number.

Tablespaces

The CREATE TABLESPACE/CREATE DEFAULT TABLESPACE statements are nothing more than a method of defining the Adabas File Control Block (FCB), which gives details about the physical limits of an Adabas file. In an Adabas database, the FCB and the File Description Table (FDT) are needed to define an Adabas file.

In earlier Adabas SQL Server versions, it has been necessary to specify a tablespace for each table to be created. From the current version on, the following alternatives will be available:

– Explicit specific tablespace.
– Default tablespace for a specific schema.
– Hard-coded tablespace that is used as a last resort.

To be able to utilize the latter two tablespaces, it is necessary to provide a mechanism to find a free file number. This is performed by doing an Adabas ’LF’ command; looking for the Adabas Response Code 17. As this response code has more than one meaning, Adabas SQL Server has been enhanced by the parameter processing capability to specify the range of file numbers to be searched. Make sure that the file numbers of security-related files and/or otherwise sensitive files are outside of these search ranges. See Appendix: **The Parameter Processing Language** in the *Adabas SQL Server Installation and Operation Manual* for more details).

From the current version onwards, it has also been possible to remove the restrictions on mandatory tablespace elements, because the hard-coded default tablespace settings are merged with those of the CREATE TABLESPACE/CREATE DEFAULT TABLESPACE.

Hard-coded Tablespace

The hard-coded default tablespace has been designed to be used in a test environment where the size of a table (number of rows in a table) is small. The hard-coded default tablespace values are quite low compared with production databases. They are defined below in terms of a CREATE TABLESPACE statement:

```
CREATE TABLESPACE default
(
   database = <catalog database name>,
   maxisn = 300,
   nisize = 10 BLOCK,
   dssize = 10 BLOCK,
   uisize = 10 BLOCK,
   dsreuse = yes
);
```

Note: This type of tablespace is used as a last resort; a tablespace for a specific table has highest priority followed by the default tablespace for the current schema.

Default Tablespace for a Specific Schema (Specified using the CREATE DEFAULT TABLESPACE Statement).

Defines a tablespace that will be used whenever there is no specific tablespace definition for a table to be created. This tablespace is at the schema level; in other words, all tables within the current schema will inherit the values from this tablespace, as long as there is not a specific tablespace (explicit specific tablespace) for a table.

Note: If no other values are explicitly specified, the hard-coded default tablespace values are merged with those specified in the CREATE DEFAULT TABLESPACE statement.

Explicit Specific Tablespace (Specified using the CREATE TABLESPACE Statement).

This is the optimal method of specifying the FCB attributes of an Adabas file.

Note: If no other values are explicitly specified, the hard-coded default tablespace values are merged with those specified in the CREATE TABLESPACE statement.

Conversion of Adabas Field Attributes to Adabas SQL Server Column Attributes

The table below shows how the most important Adabas field attributes are converted into Adabas SQL Server column attributes:

Adabas Field Attributes	**Adabas SQL Server Column Attributes**
FI	FIXED
NU	NOT NULL SUPPRESSION or NULL SUPPRESSION or SUPPRESSION
NC	NULL
NN,NC	NOT NULL
No-attribute field (Adabas field with no attributes, DE and UQ do not count)	NOT NULL DEFAULT ADABAS or NULL DEFAULT ADABAS or DEFAULT ADABAS

From this table, it can be seen that for the ’NU’ and ’no-attribute’ fields, it is possible to specify both NULL and NOT NULL (NULL is also assumed when neither is specified).

– When you specify NULL, the columns of a row that have been compressed will be converted into the SQL NULL value. This means that 0 (zero(s)) and ’ ’ (space(s)) can not be represented.
– When you specify the NOT NULL attribute, the compressed data will be converted to either 0 (zero(s)) or ’ ’ (space(s)).

This introduces a restriction of how Adabas SQL Server can use these columns in a UNIQUE constraint or UNIQUE INDEX specification. This is because of what Adabas stores in the inverted list, when the fields with ’NU’ attribute or fields without the ’NC’ attribute are used.

– The usage of the Adabas attribute ’NU’ must have the SQL column attributes of SUPPRESSION or NULL SUPPRESSION, when specified in a column of a UNIQUE constraint or UNIQUE INDEX specification.
– For fields without the ’NC’ or ’NU’ attributes, the attributes NOT NULL DEFAULT ADABAS must be specified.
– In the case of PRIMARY KEY’s, the field attribute ’NU’ is not allowed.

Note:
DEFAULT ADABAS, like all other defaults, allows the insert statement to have column values implicitly given when they are not specified in the insert statement itself. In the case of DEFAULT ADABAS, the default value is that which is provided by Adabas for the specified data type.

Views

Adabas SQL Server supports the concept of views or viewed tables. These are conceptually virtual tables which are based on the base tables. It should be noted that there are several restrictions on the use of views especially regarding update operations. Views are used either to make data access easier by abstracting data from base tables, (e.g., grouped views or joined views) or as a means of security in which certain portions of a table are ‘blanked out’ by the overlaying view definition. For details refer to the *Adabas SQL Server Reference Manua*l, section **Create View**.

Constraints and Indexes

The following conditions apply when defining and using indexes and/or constraints;

Indexes

– It is not possible to define an index on a column of type ’long alpha’ (a character column of greater than 252 character in length).
– Indexes which are made up of parts of a column are ignored by Adabas SQL Server’s query processor (i.e. if an index is defined across characters 5 through 7 of a character column, then this index is not valid).

UNIQUE Constraints, UNIQUE INDEX and PRIMARY KEY Constraints.

– It is not possible to define an index on a column of type long alpha (a character column of greater than 253 character in length).
– A UNIQUE constraint adds an Adabas UQ attribute to a super-, sub- or column- descriptor.
– The difference between a UNIQUE constraint and a UNIQUE INDEX is, that a UNIQUE INDEX may be used to define sub- or superdescriptors which are only parts of a column. Like indexes, these are also ignored by the SQL query processor.
– A UNIQUE constraint for a subtable (level 1 or 2 table within a CLUSTER), may only be defined on columns of the FOREIGN KEY constraint plus at least a column of the type SEQNO(*n*); where *n* is either 1 or 2 dependent on the table level. It is also possible to ‘over-qualify’ a UNIQUE constraint on a subtable by defining columns that are part of the subtable other than the SEQNO column. This can be used to provide extra columns in a level 2 subtable (column c of the following example):

```
CREATE CLUSTER xxx
CREATE TABLE T0 (
..
UNIQUE (a,b)
)
CREATE TABLE T1 (
a
seqno_level_1
b
c
FOREIGN KEY (a,b) REFERENCES T0(a,b)
UNIQUE (a,b, seqno_level_1,c)
)
CREATE TABLE T2 (
a
b
c
Foreign Key (a,b, seqno_level_1,c)
)
```

– To define an Adabas UQ attribute on column(s) of a subtable that do not obey rule 4, use the UQINDEX.

UQINDEX

A UQINDEX is a method of representing the Adabas UQ attribute which is not considered to be unique in SQL terms. The following rules apply:

– A UQINDEX may only be defined on level 1 or 2 subtables.

– The UQINDEX should be used to define the UNIQUE constraint that built the FOREIGN KEY relationship.

– A UQINDEX should be used to either represent the UQ attribute on columns of a Adabas Periodic group (PE) or Multiple-value field (MU), or represent a sub- or super-descriptor which contains columns of a Periodic group (PE) or Multiple-value field (MU).

Data Definition Language Name Generation Algorithm.

Within the Data Definition Language (DDL), there are two places where names may be generated. These are:

– Adabas filenames when doing a CREATE TABLE or CREATE CLUSTER statements.
– Catalog objects such as constraints, indexes and internal objects.

Generation of Adabas Filenames

When performing a CREATE TABLE or CREATE CLUSTER statement, DDL generates names for the Adabas filename. The rule for generating a Adabas filename is:

`schema name.table name` or `schema name.cluster name`

Whereby the total length of the filename may not exceed 16 characters. This means that the schema name and or the table name may need to be truncated (both schema names and table names are 32 characters long). The following rules exist for this purpose:

– Schema name will be truncated at 7 characters.
– The table or cluster name, will be a minimum of 8 characters long. But in the case of a schema name being less than 7 characters long, the table or cluster name maybe up to 16 (length of schema + 1) long.

Generation of Catalog Object Names

Adabas SQL Server may generate names for the following objects within the catalog:

– Constraint names
– Index names
– Internal catalog objects
  (currently there is only one internal object, this being the object Adabas file)

Note: The generation of names for constraints is a requirement of the SQL-92 standard).

The total length of the generated name as shown below, may not exceed 32 characters. This means that it may be necessary to truncate the object name to facilitate this limitation. All catalog objects use the following name generation rule:

object prefix *object name* [*timestamp*]

Whereby:

Object prefix:	Is a 2 character prefix followed by 2 underline characters (e.g. “TC__”).
Object name:	Is either a cluster, table or column name dependent on where the object was defined.
Timestamp	Iis a timestamp in the format “YYYYMMDDHHMMSS” (as text).

Note: Timestamp is optional and will only be used to generate a unique name within the objects scoping rules. For further details about scoping rules, refer to section **Scoping Rules Of Catalog Objects** *later in this chapter.*

The table below gives a list of all currently defined prefixes, what they mean and the object name type that will be used with them.

Prefix	**Object Name Origin**	**Description**
AF	Table or cluster name	Adabas file. This is an internal object generated by Adabas SQL Server for each Adabas file that is defined in the catalog.
CC	Column name	Column constraint. All column constraints without names will have objects generated with this prefix.
CI	Column name	Column index.
TC	Table name	Table constraint.
TI	Table name	Table index.
PI	Table name	Phonetic indx. This is reserved for future use and will then represent a PHONETCI descriptor within the catalog.
VI	Table name	Virtual index. This is also reserved for future use and will then represent a HYPER descriptor within the catalog.

Note: A column level object may be generated at certain points even though a table level object was originally defined. For example, table constraints and table indexes may be converted if they only contain one column.

Scoping Rules Of Catalog Objects.

Some Objects within the Adabas SQL Server catalog have different scoping rules, at which level their names must be unique. The table below shows these objects and their scoping level:

Object	Catalog	Schema	Table
Database name	✓		
Database number	✓		
Column name			✓
Constraint name		✓	
Cluster name		✓	
File number	✓		
Index name		✓	
Schema name	✓		
Table name		✓	
Tablespace name		✓	
User name	✓		

Databases

The term database within Adabas SQL Server, refers to the Adabas definition of the word, i.e. a collection of zero or more files which are accessible using the unique Adabas file number and the unique Adabas database ID number. Within Adabas SQL Server, a database is merely the physical location of the data.

In SQL statements, the physical database number can not be used. Instead a unique logical database identifier must be used. This is an SQL identifier. The connection between the database number and the database identifier is made by the execution of a CREATE DATABASE statement.

Before any objects can be created in a schema, a suitable database identifier must have been defined.

Whenever a CREATE TABLE statement is executed, the physical table is created in the designated Adabas database as specified in the associated CREATE TABLESPACE statement. At the same time, a description of the table is entered into the catalog. This description contains the value of the database number where the table physically resides. Whenever the table is referenced in a statement, Adabas SQL Server expects to find the table at this specified location.

A particular Adabas SQL Server may access numerous Adabas databases at the same time and under some circumstances an Adabas database may be remote. The catalog must reside in an Adabas database as well, however objects may be described in the catalog which reside in other Adabas databases. Tables from different databases may be freely combined in the same statement. Adabas SQL Server automatically establishes communications with the given Adabas databases as and when required.

Additionally, within the same transaction, tables from distinct databases can be amended. After these changes have been committed, separate Adabas ET commands are issued. If any one of these fails, then the changes that have been committed in the other databases remain valid.

Data in Table Clusters

Describing Adabas Nested Data Structures

In the context of the conceptual SQL schema, Adabas multiple-value fields (MU) and periodic groups (PE) can have various meanings.

One possibility is the interpretation of such a field as a summary of columns (and/or column groups) with the same data type. For details, refer to the section **Tables with Rotated Fields** earlier in this chapter.

Another possibility exists for alpha-type MU fields; they can simply be interpreted as running text.

In general, however, MU/PE fields can be approached as set-valued or as list-valued attributes.

Since the relational data model – and thus SQL as well – recognizes only elementary attribute types, structured attributes must be modeled in individual tables, which are linked to the master table with the help of foreign keys.

The CREATE CLUSTER DESCRIPTION Statement

The SQL structures resulting from any Adabas file structure with MU/PEs are generated with the help of a CREATE CLUSTER DESCRIPTION statement.

The drafting of such statements can be a complex, laborious operation. By using the Generate Table Description Utility (ESQGTD), you can easily draft CREATE TABLE/CLUSTER DESCRIPTION statements. They can then be edited by hand and be submitted to the server.

The following table shows, for example, the definition of Adabas structures which summarize information about states, cities, places and buildings in one file.

The PE group “B0” (“cities”) demonstrates a set-valued attribute of “states.” Along with the simple attributes “city name” and “population” (“BA” and “BB”), “cities” itself also has set-valued attributes. While the MU field “DA” (“places”) is compiled simply as a list of places, the buildings in a city are represented in two MU fields, “CA” (“building name”) and “CB” (“height”). Such parallel MU fields, as they are known, substitute for the ability to nest PE groups within one another, which is not possible.

Field Definition Table (FDT) of Database No. 214, File No. 134 (CITY_GUIDE)

```
   Level  I Name I Length I Format I  Options  I  Comment
------------------------------------------------------------------------------
  1       I  AA  I    2   I    A   I  DE,UQ    I  states.abbreviation
  1       I  AB  I   20   I    A   I  DE,UQ    I  states.state_name
  1       I  AC  I   20   I    A   I  DE,NC    I  states.capital
  1       I  AD  I    4   I    F   I  NC       I  states.population
  1       I  B0  I        I        I  PE       I  cities
   2      I  BA  I   20   I    A   I  DE,NU    I  cities.city_name
   2      I  BB  I    4   I    F   I  NU       I  cities.population
   2      I  CA  I   20   I    A   I  NU,MU    I  buildings.building_name
   2      I  CB  I    2   I    F   I  NU,MU    I  buildings.height
   2      I  DA  I   20   I    A   I  NU,MU    I  places.place_name
------------------------------------------------------------------------------
   Type   I Name I Length I Format I  Options        I  Parent field(s)   Fmt
------------------------------------------------------------------------------
 SUPER    I  X1  I   22   I    A   I  NU,UQ,PE       I  BA (   1 –  20 )   A
          I      I        I        I                 I  AA (   1 –   2 )   A
------------------------------------------------------------------------------
```

It cannot be recognized from the definition of the Adabas structure that “CA” and “CB” are to be seen as parallel, whereas “CA” and “DA” are independent from one another.

The following table shows the Adabas SQL Server cluster representation of the above Adabas file structure. The information available in the FDT is automatically transmitted to the cluster description and must not be entered by hand anymore.

Therefore, the data types of the individual columns were not shown in the example. The link from the column to the Adabas field is established using the SHORTNAME clause. A table description and/or cluster description does not necessarily have to reference all fields of an Adabas file. In any case, it is not possible, and for good reason, to enter several independent descriptions for the same Adabas file in the catalog; i.e., unreferenced fields are invisible for the entire ADBAS SQL Server.

Catalog Representation of the Above Field Definition Table

```
CREATE CLUSTER DESCRIPTION city_guide
DATABASE DB_214  FILE NUMBER 134
(
CREATE TABLE DESCRIPTION states  (
   abbreviation     SHORTNAME ’AA’ PRIMARY KEY DEFAULT ADABAS,
   state_name       SHORTNAME ’AB’ UNIQUE NOT NULL DEFAULT ADABAS,
   capital          SHORTNAME ’AC’ INDEX state_capital,
   population       SHORTNAME ’AD’
   )

CREATE TABLE DESCRIPTION cities  (
   state_abbrev     SHORTNAME ’AA’,
   city_seqno       SEQNO(1),
   city_name        SHORTNAME ’BA’,
   population       SHORTNAME ’BB’,
   PRIMARY KEY (state_abbrev, city_seqno),
   FOREIGN KEY (state_abbrev)  REFERENCES  states,
   INDEX UNIQUENESS city_state (city_name, state_abbrev)
   )

CREATE TABLE DESCRIPTION buildings  (
   state_abbrev     SHORTNAME ’AA’,
   city_seqno       SEQNO(1),
   building_seqno   SEQNO(2),
   building_name    SHORTNAME ’CA’,
   height           SHORTNAME ’CB’,
   PRIMARY KEY (state_abbrev, city_seqno, building_seqno),
   FOREIGN KEY (state_abbrev, city_seqno)  REFERENCES  cities
   )

CREATE TABLE DESCRIPTION places  (
   state_abbrev     SHORTNAME ’AA’ NOT NULL DEFAULT ADABAS,
   city_seqno       SEQNO(1),
   place_name       SHORTNAME ’DA’,
   FOREIGN KEY (state_abbrev, city_seqno)  REFERENCES  cities
   )
);
```

Table Levels and Foreign/Primary Key Relationships

The cluster concept is the only method which allows physical representation of several base tables by means of a single Adabas file. The base tables are connected with one another using referential constraints. In the current version, referential constraints are used solely to describe the physical relationships among the base tables within a cluster. The general rule is that a foreign key must always reference a primary key.

In a cluster, there is exactly one table which does not contain a foreign key. This table is designated as the master table.

All remaining tables are designated as subtables and are either:

- level 1 tables, i.e. tables which contain a foreign key to the master table, or
- level 2 tables, i.e. tables which hold a foreign key to a level-1 table in the cluster.

In the example cluster “city guide,” “states” is thus the master table, “cities” a level-1 table, and “buildings” and “places” are level-2 tables.

The value of a foreign key shows where the row of a subtable belongs. If, for example, R_city is a row of “cities,” then R_city.state_abbrev shows the state to which the city described by R_city belongs. In this way, it is also clearly identified which Adabas record contains R_city. The following rules for CREATE CLUSTER DESCRIPTION must be followed in order to keep this physical relationship clear:

Rules for Cluster Descriptions

1 Foreign keys reference only unique constraints. A subtable contains exactly one foreign key.

2 The same rules apply for the columns, constraints and indexes of the master table as for a CREATE TABLE DESCRIPTION statement.

3 Columns which are not an element of a foreign key and not of a SEQNO type are called data columns. The limitations under rules 4–7 apply to data columns in subtables.

4 The data columns of a level 1 table correspond either to MU fields which do not lie within a PE group, or to fields within a single PE group.

5 The data columns of a level 2 table correspond to MU fields within a specific PE group – the group containing those fields which the data columns in the referenced table correspond to.

6 Not more than one data column may correspond to each field (with rotated fields, each subscript counts as its own field).

7 With parallel MU fields, it is assumed that in all Adabas records, the respective counter values are the same.

8 For x=1 or x=2, a unique constraint of a level x table encompasses the elements of the foreign keys and a column of the type SEQNO(x). Other unique constraints on subtables are not allowed.

9 For indexes to subtables, the same rules apply as for level-0 tables, plus the following additional constraints:

– UNIQUE INDEX is not allowed. In order to model the Adabas UQ option, UQINDEX is used.

Note: A unique constraint is defined as either a UNIQUE or PRIMARY KEY constraint.

As with CREATE TABLE DESCRIPTION, indexes and UNIQUE constraints are also automatically supplemented in the catalog for CREATE CLUSTER DESCRIPTION; in the example, something like an index for the descriptor “BA”. Manual intervention is, however, necessary if the name of the index or constraint is to be determined by the application. Unique constraints on subtables, apart from the strongly regulated primary key constraints, are not allowed because the Adabas UQ option cannot guarantee SQL unique features in structured fields. The keyword UQINDEX serves as a substitute here.

These rules ensure that apart from a few special cases, which are described in the section below, the standard DML operations can be carried out on subtables.

Periodic Groups with only Multiple-value Fields

In the case of a PE data structure containing MU fields only, it is necessary to use an Adabas shortname on the SEQNO(1) of the PE-subtable. In the following example, the table description ’cities’ describes the dummy table which has to be created for later reference purposes.

Field Definition Table (FDT) of Database No. 214, File No. 134 (CITY_GUIDE)

```
   Level  I Name I Length I Format I  Options  I  Comment
-------------------------------------------------------------------------------
  1       I  AA  I    2   I    A   I  DE,UQ    I  states.abbreviation
  1       I  AB  I   20   I    A   I  DE,UQ    I  states.state_name
  1       I  AC  I   20   I    A   I  DE,NC    I  states.capital
  1       I  AD  I    4   I    F   I  NC       I  states.population
  1       I  B0  I        I        I  PE       I  cities
   2      I  CA  I   20   I    A   I  NU,MU    I  buildings.building_name
   2      I  CB  I    2   I    F   I  NU,MU    I  buildings.height
-------------------------------------------------------------------------------
   Type   I Name I Length I Format I  Options        I  Parent field(s)   Fmt
-------------------------------------------------------------------------------
 SUPER    I  X1  I   22   I    A   I  NU,UQ,PE       I  BA (   1 –  20 )   A
          I      I        I        I                 I  AA (   1 –   2 )   A
-------------------------------------------------------------------------------
```

Catalog Representation of the Above Field Definition Table

```
CREATE CLUSTER DESCRIPTION city_guide
DATABASE DB_214  FILE NUMBER 134
(
CREATE TABLE DESCRIPTION states  (
   abbreviation     SHORTNAME ’AA’ PRIMARY KEY DEFAULT ADABAS,
   state_name       SHORTNAME ’AB’ UNIQUE NOT NULL DEFAULT ADABAS,
   capital          SHORTNAME ’AC’ INDEX state_capital,
   population       SHORTNAME ’AD’
   )

CREATE TABLE DESCRIPTION cities  (
   state_abbrev     SHORTNAME ’AA’,
   city_seqno       SEQNO(1)   SHORTNAME ’B0’  NOT NULL,
   PRIMARY KEY (state_abbrev, city_seqno),
   FOREIGN KEY (state_abbrev)  REFERENCES  states,
   )

CREATE TABLE DESCRIPTION buildings  (
   state_abbrev     SHORTNAME ’AA’,
   city_seqno       SEQNO(1)   SHORTNAME ’B0’  NOT NULL,,
   building_seqno   SEQNO(2),
   building_name    SHORTNAME ’CA’,
   height           SHORTNAME ’CB’,
   PRIMARY KEY (state_abbrev, city_seqno, building_seqno),
   FOREIGN KEY (state_abbrev, city_seqno)  REFERENCES  cities
   )
```

Using Adabas Nested Data Structures

There is very little difference between manipulating data in table clusters versus non-clustered tables. However, some limitations apply and are pointed out below. Furthermore, it is advantageous to know how nested data structures are processed in order to really receive the maximum performance benefits, which can be gained from the nesting.

Limitations

When modifying subtables, the following restrictions apply:

1. Individual DELETE is not Supported

   The DELETE statement must not be directly applied to a subtable but always to the relevant master table. An implicit DELETE takes place when deleting a row that is referenced by the table's foreign key constraint, i.e. the DELETE cascades from the master table to the subtable.

2. Non-insertable Columns

   You are not allowed to explicitly insert values into SEQNO(1) and SEQNO(2) columns. Their values are always assigned automatically and can not be set or modified.

3. Non-updatable Columns

   Foreign key columns of subtables and SEQNO columns are not updatable. These contain information concerning the physical location of a row, which can not be changed during an update operation.

4. Anomalies for clustered Tables Containing Non-suppressed Columns

   A subtable containing a column without the SUPPRESSION attribute (i.e. the Adabas NU option), may contain rows where the column values are all defaults, even though these rows have not been explicitly inserted. This is caused by the Adabas policy of expanding the compressed Adabas records from left to right. The effect does not occur when the SUPPRESSION attribute has been used.

5. An INSERT statement on a subtable must specify all elements of the FOREIGN KEY. These elements must uniquely address one row of the referencing table.

Performance Aspects

When data is being retrieved from table clusters, it will often be necessary to formulate joined queries. As a general rule it can be assumed, a join between a subtable and it's master table does not require special join processing, if the join condition compares the foreign key of the subtable against the referenced key of the master table.

THE ADABAS SQL SERVER SECURITY CONCEPT

Introduction to Servers with the Security Features

Access to all data can be controlled using the security functionality provided by Adabas SQL Server. Such functionality is compliant with the ANSI Standard and is, therefore, in addition to any security mechanisms provided by Adabas itself. SQL security functionality is defined by Data Control Language Statements (DCL).

SQL contains the concept of a data owner. A schema always has an owner as specified by the authorization identifier when the the schema is created. The authorization identifier will specify a particular user, which is already known to the system. Any data objects created within the schema, automatically belong to the owner. It is only the owner of the schema who may alter the contents of a schema by executing DDL statements, e.g. CREATE TABLE, DROP VIEW etc.

Adabas SQL Server, therefore, also has the concept of a user, as defined by the CREATE USER statement. The execution of such a statement by the DBA, results in the introduction of a unique user identifier to the Server.

Privileges

Privileges are access rights to a particular data object e.g. base table or view. The owner of the data object always has all privileges for that object. The privileges are :

SELECT	a right to read from the table or view
INSERT	a right to insert into the table or view
DELETE	a right to delete from the table or view
UPDATE	a right to update a table or view or a column

Privileges can be given to another user, other than the owner, using the GRANT statement. The owner initially grants privileges to other users. The owner may not only grant the basic privileges, but may additionally grant the right to grant the privilege to a third user. This is done using the WITH GRANT OPTION. Whole hierarchies of privileges may be established.

A privilege is one of the above access rights for a particular data object for a particular user with the optional WITH GRANT OPTION qualification. The privileges are stored in the catalog and are maintained using the GRANT and REVOKE statements.

It is not only a specific user who may be the recipient of a privilege. The ’pseudo user’ PUBLIC may also be granted access rights. This means that everybody may access the data object.

A particular user may only receive a particular privilege once. For example, if user Tim grants Peter a privilege and Peter in turn grants the same privilege to Kevin, then if Kevin tries to grant the privilege to Tim again this would be rejected. This means, no cyclic privileges are permitted. In addition, in another example, if Tim were to grant to Peter and Kevin the same privilege and then Peter granted Martin the privilege and Kevin tried to grant the privilege to Martin, then Kevin’s attempt would be rejected. This means, no networks of privileges are permitted. Only simple tree structures of privileges are permitted.

Views and Security

In order to create a view, based upon a data object owned by somebody else, the view’s creator must posses at least the select privilege for that data object. Furthermore, although the creator of the view is the owner of the view, he may only grant privileges to the view to other users if he is in possession of the WITH GRANT OPTION for the original data object.

If the view is by its nature read-only (e.g. it is a joined view) then the view will only have associated SELECT privileges. If, however, it is technically possible to update the view, then any privileges in addition to the SELECT privilege possessed for the data object upon which the view is based will be inherited.

Any such view is, therefore, dependent upon the SELECT privilege.

Removing Privileges

Privileges can be removed from a recipient using the REVOKE statement. The REVOKE statement explicitly removes a privilege and deletes it from the catalog. No CASCADE functionality is possible. This means that if there are other privileges dependent upon the one which is to be revoked, then they must be revoked first. In addition, if there is a dependent view, then it must be manually dropped, prior to revoking the privilege.

In contrast to the REVOKE statement, if a data object is dropped, then all associated privileges are indeed automatically revoked.

In an analogous fashion to DDL statements, the GRANT and REVOKE statements are not subject to transaction logic. This means that neither a GRANT nor a REVOKE statement can be rolled back. DCL statements cannot be mixed with DML statements within the same transaction. DCL statements can be mixed with DDL statements within the same transaction.

This means that the action implied by a DCL statement is immediately effective for all users.

Authorization

The whole point of SQL security is to restrict access for certain users to particular data objects and to enable it for others. Therefore, a user must be authorized to perform a particular action. During the execution of a DML statement, the requested action for the particular user against that data object is checked against the privileges contained in the catalog. The action is either rejected or authorized and processing continues.

If it is rejected, then the user will receive the condition code 4286. This states that the object is either not found or the user was not authorized. This ambiguity is intended, as even the knowledge that an object exists, although the user is not authorized to access it, is knowledge that the user is not permitted to obtain. The unauthorized user should not now know if the object actually exists. In addition, any catalog query, will only present catalog entries for which the user has privileges.

The mere act of establishing authorization for a user leads to performance losses.

Authentication

The Adabas SQL Server security concept requires that users are known to the system. Users must, therefore, make themselves known to the system for a particular session using the CONNECT statement. The specified user ID is compared against a list of users contained in the catalog. If in this list a password has been specified, then the user must also provide a password as a parameter to the CONNECT statement. At client site, this password is encrypted prior to being sent to the server. If no password is contained in the catalog, then the password is optional and ignored.

The act of supplying a user identifier and verifying it is called authentication.

User Administration

There are two types of users. The special user, with the user identifier 'DBA' is always present in the system. The other type of user is called a non-DBA user.

The DBA user is established as part of the server generation. The DBA's user identifier is in fact hard coded and cannot be dropped. This user has certain additional rights over normal users.

– A DBA can create and drop schemas.
– A DBA can create/drop and alter other user identifiers and passwords.

Only the DBA can create a schema. This inhibits the uncontrolled ability to execute DDL statements and create data objects. The DBA does not have the right to manipulate the objects of other users however.

The DBA is also the only person who can create users using the CREATE USER statement. This is where a particular unique user identifier is introduced to the system, with an optional password. The DBA can at any time, alter the password of any user. The DBA can, therefore, at any time, connect to the system as any user, in order to perform any necessary administration tasks.

The DBA can remove a user from the system using the DROP USER statement. Should the user still be in possession of any data objects, then the statement will be rejected. There is no cascade functionality for drop user. Any data objects must be manually dropped prior to removing the user altogether.

Normal users may change their password as desired using the ALTER USER statement.

Non-Security Servers

Consequences of Generating a Server Without the Security Features

If the Adabas SQL Server which is generated does not support the security features the following consequences apply:

– at runtime, no security check is performed. It will not be checked if the active user possesses the access rights to the specified table, etc. For this reason, the performance of the server will be more favorable than in a security version.

– the execution of GRANT/REVOKE statements is not possible.

– those tables, established to hold privilege-related data in the catalog will be created but are empty (for example, the table: table_privileges).

The following three points are valid in either version, security or non-security:

– the USER must still be defined using the CREATE USER statement.

– the user DBA is the only one authorized to create a schema. The owner of a schema is the only one authorized to create tables, views, etc. This is true, whether Adabas SQL Server has been generated with or without the security feature.

– full password support is included, the CREATE/DROP/ALTER USER statements can be executed as well as the CONNECT statement with user specification.

Interaction with Adabas Security Functionality

In terms of Adabas, Adabas SQL Server is an application program, which maps client-specific SQL requests to Adabas session contexts. Therefore, the processing chain consists of of three elements:

– the Adabas SQL Server Client, placing SQL requests,
– the Adabas SQL Server, placing Adabas calls on behalf of the Client, and
– the Adabas Nucleus, which finally executes the Adabas calls.

The following picture illustrates this structure:

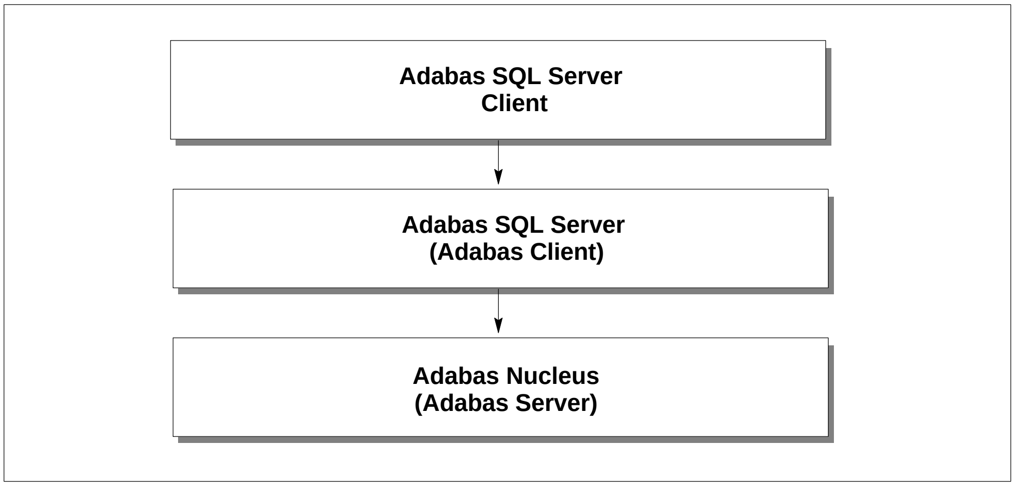


Figure 3-1: The processing chain

Further information on this structure and the implied interaction with Adabas is given in the following documents: *Adabas SQL Server Installation and Operations Manual*, section **USER EXISTS** and in the *Adabas SQL Server Programmer’s Guide*, Appendix **Adabas SQL Server and Other Software AG Products**.

Access to an ADASCR protected Adabas Nucleus

If the Adabas Database Administrator has established Adabas security using the Adabas database security utility (ADASCR), then each Adabas call has to provide the security password in Additions 3 of the Adabas control block.

Adabas SQL Server offers the Server user exit 5 as mechanism to insert the Client- specific security password into the Adabas control block.

The usage and creation of Server user exit 5 is discussed in the *Adabas SQL Server Installation and Operations Manual*, Chapter **USER EXISTS**.

Access to an ADAESI protected Adabas Nucleus

If the Adabas nucleus is protected using the Adabas External Security Interface (ADAESI), then an identification protocol must be adhered to between Adabas SQL Server and ADAESI. This identification protocol is implemetend inside Adabas SQL Server and is executed automatically.

4

GENERAL CONCEPTS OF SQL PROGRAMMING

This chapter describes, in general terms, how to embed SQL statements in an application program. Embedded SQL statements enable the application program to communicate with the underlying DBMS like Adabas in order to inspect and manipulate its data.

SQL Statements

SQL Statements are not part of the host language but are embedded in an application written in the host language. As explained in the previous chapter, the compilation of such a program consists of two phases, namely the precompilation of the SQL statements contained in the application program followed by the compilation of the actual program itself.

The SQL statements must be invisible to the host language compiler during the compilation phase. In fact, the bare embedded SQL statements are commented out by the Adabas SQL Server precompiler and are replaced by statements generated into the application program in a form that corresponds to the requirements of the host language.

The Adabas SQL Server precompiler must be able to identify all embedded SQL statements and therefore all SQL statements are individually delimited by special SQL delimiters. It is not possible to have more than one SQL statement between one set of delimiters.

The SQL Starting Delimiter

The starting delimiter consists of a sequence of two words, namely:

EXEC SQL

In Adabas SQL Server mode they must be separated by one or more white space characters, i.e they may be separated by one or more lines or blanks, and may be in either upper or lower case depending on what the host language permits.

In ANSI mode the two keywords must be in upper case and must be separated by blanks not lines.

The SQL Statement Body

Once the starting delimiter has been specified, the body of the particular statement must be given. It must be separated from the starting delimiter by at least one white space character and may be specified on the same or on a following line to the starting delimiter. The statement may be specified in either upper or lower case as appropriate and may be spread out over numerous lines. Each keyword or token must be separated by at least one white space character and may not be split over two or more lines. In Adabas SQL Server mode, keywords may be written in upper or lower case depending on the host language regulations. In ANSI mode, keywords must be written in upper case letters only.

The SQL Terminating Delimiter

The terminator of every SQL statement is either explicitly determined by a terminating delimiter or is implicitly determined depending on the host language being used. For further details see the host language dependent chapters later in this manual. The terminating delimiter itself is also host language dependent and may either be END EXEC or a semi-colon ‘ ; ’. In Adabas SQL Server mode END EXEC may be specified in upper or lower case letters where the host language permits. In ANSI mode only upper case letters are permitted.

Host Language	Terminating Delimiter
C	;
COBOL	END EXEC or END EXEC.
PLI	;

Comments within an SQL Statement.

Two types of comments are supported:

- Host language comments may be positioned anywhere within an SQL statement where a white space character can appear. Such a comment must obey the rules determined by the host language in question. For examples refer to the host language chapters later in this manual.

  Host language comments are not permitted between the keywords:
  EXEC SQL,
  BEGIN DECLARE SECTION and
  END DECLARE SECTION.

- SQL comments may also be positioned anywhere within the SQL statement body where a white space character can appear. Such a comment is a character string preceded by two minus characters '– –'. All characters following this starting delimiter until the end of the line are interpreted as part of the comment.

  With host languages where nested comments are not permitted, the host language comment delimiters within a statement will be amended in some way so that the actual SQL statement is commented out and thus nested comments are avoided.

Host Variables

During runtime SQL statements must be able to pass data from the application program to the Runtime system and vice versa. This is achieved through the medium of host variables. Host variables are variables which are declared by the application program but are accessed (although not exclusively) from within SQL statements.

The following illustrates the different uses of host variables:

- Host variables can be used to receive values derived from the database according to the specification of a select statement. The derived values are copied into the target host variables as specified in the statement. The application program can then use the returned values contained in the host variables.
- The result of a query can be influenced by the use of host variables within a predicate contained in a where or having clause. A value is loaded into the host variable prior to the execution of the SQL statement. Adabas SQL Server then takes this value and ‘plugs’ it into the designated position in the predicate.
- The contents of a table can be changed by taking the value contained in a host variable and during the execution of an UPDATE or an INSERT statement amending the table accordingly.
- Host variables can be used in other more specialized areas for example as dynamic cursor names, as dynamic SQL statement identifiers and strings, as database identifiers and as an SQL descriptor area.

Declaring Host Variables

The declaration of any host variable must follow the rules of the appropriate host language. In order to be able to access a host variable from within an SQL statement you must declare it in a BEGIN DECLARE SECTION. Any number of declarations may be grouped together in such a block which is delimited by the SQL statements BEGIN DECLARE SECTION, END DECLARE SECTION. Example:

```
EXEC SQL
BEGIN DECLARE SECTION
;
....host language specific declarations ....
EXEC SQL
END DECLARE SECTION
;
```

More than one such delimited block of host variable declarations may be specified within a program, but they can not be nested. An SQL statement which uses host variables must lie within the scope of the host variables according to the rules of the host language.

ANSI Specifics

In ANSI compatibility mode, regardless of any scoping rules, all host variables which are referenced in an SQL statement must have a unique identifier for the whole compilation unit.

Furthermore, host variable structures or their individual elements may not be specified in any SQL statement when compiling in ANSI compatibility mode.

Using Host Variables

Host variables declared in the above-mentioned manner can be accessed in the host program without special consideration.

When used within an SQL statement, the variable must be in scope and should be prefixed with a colon so it can be distinguished from an identifier. The variable identification may be identical to an SQL keyword.

The type of the host variable will be translated to an appropriate SQL internal type depending on the host language. Therefore, the type of the variable must fit the context of its use. Refer to the appropriate host language specific section for more details. In any case, the correct type will be required and will be checked by Adabas SQL Server.

Any host variable referenced in a static DECLARE CURSOR statement must also be in scope for the associated OPEN CURSOR statement.

Host Variable Structures

Up to now the discussion has centered on single host variables. However, under certain circumstances a structure may be specified as an alternative. For example, in the fetch statement it is of course perfectly valid to individually list host variables to receive the derived values. An alternative would be to specify a structure whose individual fields matched the format of the associated projection list. Adabas SQL Server is able to resolve the individual fields of the structure. This method may also be used within an embedded SELECT statement. Structures may also be specified in an USING clause, which is an Adabas SQL Server extension.

In other situations whole structures are not permitted as they are meaningless, e.g as a value specification in a predicate. However, an individual field of a structure may be specified, as long as it is uniquely identified according to the host language rules.

If one or more of the fields in the structure require an associated INDICATOR variable then a completely new structure must be declared which maps all the fields of the original host structure with INDICATOR variables. There is an one-to-one relationship between the actual variable fields and their associated indicator values in the two structures. An indicator structure is specified in the SQL statement by appending its name with a colon to that of the actual host variable structure. in an analogous fashion to individual host variables and their associated indicator values.

Note:
Pointer expressions will be supported in the next release version.

The SQL Communications Area (SQLCA)

Any application program needs to be able to check the success or failure of any particular SQL statement once it has been executed. At least one special host variable structure needs to be declared in the program, so that there is always one in scope for each SQL statements. For this purpose, a host variable structure, called SQL communication area or SQLCA is used. Adabas SQL Server updates certain fields of the structure depending on the nature of the particular SQL statement and the outcome of its execution. The application program can verify the successful execution of an SQL statement by inspecting the contents of the sqlcode element of the SQLCA.

Declaring the SQLCA

As stated above, the SQLCA is a special type of host variable structure. In order to ensure that the structure has the correct format, the application program should use the definition of the SQLCA provided by Adabas SQL Server. To facilitate this, the following SQL statement should be embedded in the application program:

```
INCLUDE SQLCA;
```

Executing this statement has the effect of generating an appropriate SQLCA definition and declaration at the point where it is specified. Thus, the SQLCA obeys the rules of scoping set by the host language relative to the position of the INCLUDE SQLCA statement.

Application programs can explicitly declare an SQLCA without using the INCLUDE statement. It is then the responsibility of the programmer to ensure that the structure is correctly defined and declared. Failure to do so may lead to unpredictable results.

Using the SQLCA

Once the SQL statement execution has completed, the application program should quiz the SQLCODE field of the SQLCA. The program logic should then be in a position to deal with any eventuality. This may laboriously be done for every SQL statement. However, by using the precompiler directive WHENEVER, such coding can be generated automatically. Refer to the *Adabas SQL Server Reference Manual*, section **WHENEVER Statement** for more details.

Currently not all fields in the SQLCA are used.

Note:
Following static statements do not result in any update of the SQLCA:

DECLARE CURSOR,	*BEGIN DECLARE SECTION,*
END DECLARE SECTION,	*WHENEVER*
INCLUDE.	

Field	Description
sqlcaid	An eight-byte character string containing the constant ‘SQLCA’. This field serves mainly as an eye-catcher for easier memory dump interpretation.
sqlcabc	A four-byte integer variable containing the length in bytes of the SQLCA. It normally contains the value 136.
sqlcode	A four-byte integer variable containing the status of the executed SQL command. The standard defines three categories of results. **zero** The command has been successfully executed. (There may have been warning messages c.f. sqlwarn0.) **negative** An error has occurred. The negative number indicates the nature of the error. Adabas SQL Server allows the installation to define its own error values. Thus compatibility with different SQL DBMSs can be achieved. (The ANSI/ISO standard does not specify which negative values should be used with a particular error status). When a negative code is returned, the SQLERROR condition of the WHENEVER statement is activated.

Field	Description
	positive The command executed successfully, but an exceptional condition occurred. **+100** This value is returned to indicate that the command was successfully executed but processed no rows. It is used in conjunction with the following commands: DELETE FETCH INSERT SELECT UPDATE
sqlerrm	A variable containing two fields holding the actual values to replace the variables contained in error messages. **sqlerrml** A two-byte integer field indicating the length of sqlerrmc. The range is from 0 through 70. If the value is zero, there is no data in the sqlerrmc field. **sqlerrmc** A character string of variable length which may not exceed 70 characters. The string contains one or more actual values for the variables of the associated error messages. As many error messages contain no text variables this field is not always filled. Each value in the string is terminated by one byte containing the hex value ‘FF’.
sqlerrp	An eight-byte character variable. This field is not currently used.
sqlerrd1 – 6	A group of six integer fields, each four bytes in length. **sqlerrd1** is currently unused **sqlerrd2** is currently unused **sqlerrd3** contains the number of rows affected after the execution of an INSERT, UPDATE or DELETE statement. **sqlerrd4** is currently unused

Field	Description
	sqlerrd5 is currently unused **sqlerrd6** is currently unused
sqlwarn0 –	A group of eight character variables, each one byte in length. **sqlwarn7** The default contents is blank. A "W" denotes a warning. **sqlwarn0** When this variable is set to "W" it signifies that at least one other sqlwarn variable is also set to "W". When this variable is set, then the SQL WARNING condition of the WHENEVER statement is activated. **sqlwarn1** This variable signifies that truncation has occurred during the assignment of a character string to a host variable. **sqlwarn2** Currently unused. **sqlwarn3** This variable is set when a mismatch on the number of columns occurs between the SELECTs within the DECLARE and the FETCH statements. **sqlwarn4** Currently unused. **sqlwarn5** Currently unused. **sqlwarn6** Currently unused. **sqlwarn7** Currently unused.
sqlext	A character string eight bytes in length. It is unused.

Error Handling

Errors occurring during the precompilation of an SQL application program produce an error message in the form of an error number and an associated brief description text. For a more detailed description of the error and the necessary steps needed to correct this error, refer to the *Adabas SQL Server Messages and Codes Manual.*

During runtime however, the only means of communication between Adabas SQL Server and the application program is using the SQLCA. Although an error number is returned in the SQLCODE field, there is no provision in the SQLCA for the associated error message text. For simple application programs relying on static SQL this usually poses no problem. However, especially for a PREPARE or EXECUTE IMMEDIATE statement, a dynamic application program often needs to be able to present a meaningful error message to the user. Such a requirement can be satisfied by the application program calling the routine ‘esqerr’. This is not an SQL statement but rather a routine delivered as part of Adabas SQL Server. This routine requires the following parameters:

– the address of the SQLCA,
– the address of the user supplied target buffer,
– the address of the length of the user supplied buffer,
– the address of the language, currently only English is supported,
– the minimum length of the error text buffer.

The function identifies the required error message based on the value contained in the sqlcode field of the SQLCA. Any textual parameters are to be found in the field sqlerrmc and these are merged by the function with the raw error message texts. The final text is then copied into the user supplied buffer up to the length given in the length parameter. This length parameter is then updated to show the actual length of the message copied. The function only responds to the contents of the SQLCA and it is therefore the programmer’s responsibility to ensure that the SQLCA really contains the required information. For detailed information refer to the appropriate host language chapters later in this manual.

Note:
The use of this function may result in a significant increase in the size of the application program’s object code depending on the platform.

Program Structure

To develop correct SQL application programs, it is important to understand the difference between the physical order of the SQL statements in a program and the order of their execution.

The Adabas SQL Server precompiler scans the source application program for SQL statements and effectively skips any host language statements or commands. The Adabas SQL Server precompiler has no understanding of the underlying logic of the application or indeed of the context of any particular SQL statement. All it can actually understand is an isolated collection of SQL statements.

Under the following circumstances, the physical order of the statements is relevant and does not have anything to do with the actual order in which the statements will be executed.

- When running in ANSI compatibility mode, any statements which reference a cursor must physically follow the associated DECLARE CURSOR statement. In Adabas SQL Server mode, this restriction does not exist and so the physical order of such statements is irrelevant.
- In Adabas SQL Server mode, although the DECLARE and OPEN CURSOR statements must be in the same source file, other associated statements need not be. Refer to the chapter **Static SQL,** later in this manual, for more details.
- The physical ordering of other statements can now follow freely as long as any host variables accessed within an SQL statement have been declared physically prior to usage and an SQLCA is in scope for each statement.

Positioning the SQL Statement

Almost all SQL statements may be positioned anywhere within an application program where a host language statement would be permitted. This is, because in general, SQL statements are replaced with appropriate generated host language statements by the Adabas SQL Server precompiler. The rules governing the positioning of host language statements also apply to the embedding of SQL statements. Obviously the positioning of the SQL statement must also conform to the context of the logic of the application program. As long as each SQL statement is individually delimited and where the host language permits, more than one SQL statement may be positioned on a single line.

The following statements are exceptions to the above rules:

BEGIN DECLARE SECTION
END DECLARE SECTION
INCLUDE
	these statements can only be positioned where host language declarations are allowed.

WHENEVER DECLARE CURSOR (static)	these statements can be placed anywhere dependent on the desired control flow

Default Table Qualification

When referring to a particular table from within an SQL statement, the table identifier can be qualified by an explicit schema identifier. However, if no such identifier is specified then Adabas SQL Server assumes that the current default schema identifier is intended. Adabas SQL Server will, therefore, append this identifier to the unqualified table identifier in order to produce a table specification. The default schema identifier may be specified as a system parameter.

If a default schema identifier has not been set explicitly, the unqualified table name will implicitly be qualified by the user identifier as has been set by a CONNECT statement. In the precompiler environment this user identifier is derived from the operating system user name.

The same application program precompiled with different default schema identifiers may produce different results. This could result in a particular table not being found, as the table specification is not listed in the catalog or at runtime, the two instances of the application might run against different physical tables.

In practice, such a mechanism can be used during the application program’s development. The program can be developed against certain test tables with test data, whose table identifiers match those of real existing tables but whose schema identifiers differ. When developing and testing, the default schema identifier should be set to that of the test tables and the table identifiers in the application program should not be explicitly qualified. When the development is finished and the program is ready to go into production, all that is needed is a change of default schema identifier, to match that of the production tables and to recompile.

Transaction Logic

Modifying data in the database involves the execution of at least two statements.

1 The normal INSERT, DELETE, or UPDATE statements. Such statements define the changes which are to take place.

2 A COMMIT statement which ‘fixes’ the changes or makes them permanent.

The period of time between the execution of two such COMMIT statements or between the beginning of a program and the first COMMIT statement is called a transaction or a unit of recovery.

It may, however, be determined that the changes should not be ‘fixed’ in the database. By issuing a ROLLBACK statement, all the changes are backed out or recovered. This means that all changes that have been made during the current transaction are reversed and the database is reset to the state it was in at the beginning of the transaction.

Note:
Only changes initiated by the user during the current transaction are affected by the COMMIT or ROLLBACK statements.

Any rows in the database which are modified are set in hold by Adabas SQL Server until the transaction is completed. This blocks any conflicting modification by other users as they must wait until the row is released from hold, i.e the transaction has been completed. Only then may any other waiting users access the row. It is important to realize that other users only have to wait if they attempt to modify a row that is in hold. If there is no conflict then no waiting occurs. Other users may always read rows which are currently in hold but can not put that same row in hold.

Static Cursors

It is not just rows which are directly affected by modification statements which are put in hold. If a cursor is determined to be FOR UPDATE then rows which it reads are also set in hold in case those rows are going to be amended. Even if they are not actually amended they are kept in hold until the transaction is terminated.

A cursor is determined to be FOR UPDATE:

– if it has an associated UPDATE or DELETE statement within the same compilation unit or
– if it is explicitly attributed as FOR UPDATE or
– if the default, when not explicitly labelled as FOR FETCH ONLY, is that it should be determined to be marked as FOR UPDATE.

If such a cursor reads an entire table, then all rows are eventually put into hold, thus blocking amendment of the table by other users.

A row is only put into the hold status when it is actually read for the first time. It is therefore possible that two competing users can mutually lock each other out. In other words a dead lock situation can occur. In such a case, both user will have to wait until the prevailing system time-out period has passed before they can be backed out. For the user this means that eventually an exception condition will be received by the application program.

Numerous changes can be specified within a transaction. There is no theoretical limit to the number of amendment statements which can be executed within a transaction. It is however advisable to keep transaction units small and manageable because:

– large numbers of rows in hold severely effect system performance.
– if a back-out is required a large number of changes will be lost and not just the most recent.
– if the system terminates abnormally, the changes within an uncommitted transaction will be lost.

The termination of a transaction also has the effect that all cursors that were open are closed automatically. This means that they must be re-opened in order to use them again.

Dynamic Cursors

For dynamically created cursors the same rules apply as for normal static cursors with the following exceptions:

- If the associated positioned UPDATE or DELETE statements is generated dynamically or if the cursor name is a host variable, the Adabas SQL Server precompiler can not determine at precompile time if there are any associated UPDATE or DELETE statements in the same compilation unit or not. Therefore, whether a dynamic cursor is FOR UPDATE or not is solely determined by the explicit specification of a FOR UPDATE or FOR FETCH ONLY clause and by the global default. You are recommended to always use the FOR UPDATE or FOR FETCH ONLY clause for dynamic cursors.
- Another difference is that the FOR UPDATE and FOR FETCH ONLY clauses are not part of the dynamic DECLARE CURSOR statement but must be specified in the SELECT statement. Example:

```
EXEC SQL
     PREPARE fetch_id FROM ”SELECT cruise_id FROM cruise
                         WHERE cruise–id = 10000001 FOR UPDATE”;
EXEC SQL
     DECLARE c1 CURSOR FOR fetch_id;
EXEC SQL
     OPEN c1;
EXEC SQL
     FETCH c1 USING :hv;
EXEC SQL
     PREPARE update_id FROM ”DELETE FROM cruise WHERE CURRENT OF c1”;
EXEC SQL
     EXECUTE update_id;
```

DDL/DCL Statements

The processing logic for DDL/DCL transactions differs in some points from the rules outlined above. Even though all transactions must be closed by a COMMIT or ROLLBACK statement, it is important to know, that in case of DDL/DCL statements:

- the ROLLBACK statement does not cancel the changes that were made to the database during that transaction,
- the COMMIT statement does not permanently fix the changes that were made to the database during that transaction, but changes become effective immediately after the execution of each individual statement within the transaction.

Transactions Containing Different Types of Statements

The execution of DDL and DCL statements may be mixed in the same transaction, but may not be mixed with the execution of DML statements.

The mixing of DML and DDL/DCL statements’ execution within one transaction will be detected, the violating statements’ execution will be rejected and an error message will be issued. The current transaction status will not be affected. For example, in a transaction with all DDL/DCL statements a DML statement will be considered a violating statement and vice versa.

Transaction neutral statements (PREPARE, EXECUTE, EXECUTE IMMEDIATE and DESCRIBE) may be mixed with all other statements in one transaction, i.e. they may be contained in a DDL transaction, a DCL transaction, a mixed DDL and DCL transaction and also in an DML transaction.

A COMMIT/ROLLBACK statement with a KEEPING ALL clause will not change the transaction status.

Views

Creating Views

Views are established by the execution of a CREATE VIEW statement. Unlike the CREATE TABLE statement, no physical data structures are established. Only the definition of the view is stored in the catalog.

A view can be thought of as a virtual table, one which does not exist physically. Whenever a view is referenced, conceptually a temporary table is established, based on the query specification in the CREATE VIEW statement and data is derived from the underlying base tables. The view referenced can then conceptually be processed as if it were a normal table.

Any view reference is replaced with appropriate references to the underlying base tables as defined by the query specification in the CREATE VIEW statement during compilation.

Any statement which references views can be expressed by a valid equivalent statement which only references the underlying base tables. The view references are said to be merged. This imposes certain restrictions on the use of views, which bring their use into conflict with the conceptual idea that a virtual table is established when required.

The process of merging occurs during compilation of statements which reference the view. There is no performance disadvantage to be considered with the use of views. The use of views in dynamic statements will also not lead to a significant performance loss.

Updating Views

A view can not be have the status read-only if data is to be amended.

A view is called a **“read-only view”** if the view is either grouped or joined or at least one of the derived columns does not have a label. For details on derived column labels see the *Adabas SQL Server Reference Manual*, chapter **Common Elements**, section **Query Specification**.

A view is called a **“joined view”** if more than one table has been specified or a joined view has been referenced in the FROM clause.

A view is called a **“grouped view”** if the view is derived from a grouped query specification.

When an UPDATE, INSERT or DELETE statement is applied to a view, it is not the view which is updated, rather the underlying base table. If a row is inserted or changed such that its contents take it out of the domain of the view as defined by the WHERE clause in the view definition, then the row still physically exists in the base table, but is no longer accessible to the view user.

When issuing an INSERT statement on a view which does not enclose all columns of the base table, default values will be assigned to those columns not referenced in the statement.

Dropping Views

A DROP VIEW statement also has no effect on the underlying data. Only the definition is deleted from the catalog and any statements referencing this view are marked invalid.

Dropping a view which has other dependent views based on it will result in these dependent views being invalid. A DROP VIEW statement can not be successfully completed if dependent views exist. In this case, the option CASCADE has to be specified which will drop the desired view and its dependent views.

Reasons for Using Views

- By only specifying certain columns in the derived column list of the query specification in the CREATE VIEW statement, access to the other columns of any table is effectively restricted. Such columns are simply not part of the view definition and hence can not be referenced in conjunction with the view. This can act as an aid to security as if certain users only have access to particular views then access to the ’omitted’ information is denied them.
- Similarly by specifying a suitable WHERE clause in the view definition, particular rows can be selected for the view and other rows will never be part of the views domain.
- It can therefore be seen that by using the above techniques, a view can be used to present carefully chosen portions of any underlying base tables as if the view were in fact a real table.
- Views can also be used simply to save typing effort. If a particular complex expression is going to often be used, then it may be easier to include it in a view.
- Views may also be used to present data in a differing format to that of the raw data in the base tables. This can be achieved by the use of numeric expressions in the derived column list of the view definition.

Limitations on Using Views

- A grouped view, i.e one that is based on a grouped query specification can not in turn be referenced in a grouped query. Otherwise, once merged, the resulting statement would be ’doubly’ grouped which is then not a valid equivalent statement.
- The keyword ’DISTINCT‘ must not appear in the derived column list of a view.
- If the view is said to be read-only, it can not be referenced in an INSERT, a DELETE or an UPDATE statement.

STATIC SQL

Introduction

Static SQL refers to a particular type of application where SQL statements are fixed or static. This is as opposed to dynamic SQL where the actual statement to be executed against a database is created at run time. Thus static SQL statements are embedded SQL statements that do not vary during the execution of the application program.

Strictly speaking, embedded statements like the PREPARE statement, for instance, are also static. They are embedded in an application program but they enable the use of dynamic SQL statements. Such statements are described in the chapter **Dynamic SQL** later in this manual.

The statements of this product version comprise:

- DDL Statements
  Data Definition Language. This defines the structures which are to reside in the database, e.g., the CREATE TABLE statement
- DML Statements
  Data Manipulation Language. This enables operations to be performed on the data contained in the structures in the database, e.g., the SELECT statement.
- DCL Statements
  Data Control Language. This enables the controlled access of all data using specially designed security functionality, e.g., the GRANT/REVOKE statements.

Defining the SQL Data Structures

Before any manipulation of any data can take place, the necessary data structures have to be defined. Adabas SQL Server needs to store information on these structures which are established or amended using DDL statements. This information is stored in the catalog associated with Adabas SQL Server. The structures themselves are established in the desired Adabas database. It is essential that the information in the catalog reflects the current state of the structures and this is ensured by the correct use of DDL statements.

Note:
Through the use of Adabas Utilities or Adabas Online Services it is possible to change the nature of the structures in the ADABAS database. However, this must not be attempted in an SQL environment because the catalog will not be updated correspondingly and this will lead to unpredictable results.

DDL statements can be statically embedded like any other SQL statement. New Tables and views can be created or deleted using the CREATE TABLE, CREATE TABLESPACE and CREATE VIEW statements or the corresponding DROP statements. All of these statements change the appropriate entries in the catalog and possibly in any underlying Adabas file which represents a table.

Note:
Dropping a base table using the DROP TABLE statement ***destroys*** *all the data contained in it. Once the statement has been executed, Adabas SQL Server provides no means of recovering the data. The only chance to restore the data and catalog is to load a backup tape.*

Before a CREATE TABLE statement can be executed successfully, the corresponding CREATE TABLESPACE must have been executed without an error. The CREATE TABLESPACE statement sets the specific parameters for the resulting Adabas file. The default settings of this statement are identical to the default values specified when using Adabas Online Services.

If a table already exists in Adabas (i.e., was created using Adabas Online Services or Predict) its description must be introduced to Adabas SQL Server using the CREATE TABLE DESCRIPTION statement. Even though Adabas SQL Server is able to generate the Adabas short names, they should be explicitly specified to ensure that the SQL table description and Adabas file description matches. The corresponding DROP TABLE DESCRIPTION statement removes the table’s description from the SQL environment.

The use of the Generate Table Description Utility can simplify the creation of CREATE TABLE DESCRIPTION/CREATE CLUSTER DESCRIPTION statements and at the same time ensure accuracy.

Note:
Dropping a base table using the DROP TABLE DESCRIPTION statement ***does not destroy*** *the data contained in it but merely deletes the description of it in the catalog.*

The execution, not merely the compilation of a CREATE TABLE statement will result in a new table being listed and defined in the catalog as well as in a new Adabas file in the appropriate Adabas database. At compilation time, Adabas SQL Server does not check to see if a table referenced in a CREATE or DROP statement actually exists. Only during execution will it be recognized that a table is to be created which already exists, or that a table is to be dropped which does not exist. Attempts to access undefined tables will result in a runtime exception condition. A newly created table may be dropped in the same application program.

When creating or dropping views there is no underlying Adabas file, only an entry in the catalog. Therefore, the tables and views referenced in a CREATE VIEW statement within an application program must exist at compilation time, which means the appropriate CREATE statements and the CREATE VIEW statement based on this newly created table can not be within the same application program.

It is possible, if the correct compiler options have been set, to mix DDL and DML in the same application program. However, any table or view referenced in a DML statement must also exist at compilation time. Attempts to reference undefined tables will result in an compiler error as the compiler needs to gather information on the particular tables at this point.

The CREATE DATABASE and CREATE TABLESPACE statements act in a somewhat different manner. No compiler checks are performed. Therefore, a CREATE DATABASE statement can be followed by table creation statements referencing the newly created database identifier.

Manipulating Data

The DML (Data Manipulation Language) component of SQL encompasses the following functionality:

– populating the data structures as described in the previous section with actual data,
– enabling the retrieval of data from the data structures,
– amending the actual data by either changing or removing values.

Two distinct concepts exist in order for the program to be able to manipulate data. They are non-cursor or cursor operations.

Non-cursor-based Statements

Statements which are not based on a cursor are not associated with other statements in any way.

Data is generally retrieved by using the SELECT statement and when embedded it is sometimes called the single row SELECT statement. An INTO clause and an appropriate host variable list must be provided in order to receive the returned data. This mechanism, therefore, does not facilitate the retrieval of more than one row as it is impossible to be able to accept multiple rows in one go. An embedded static SELECT statement may only generate one row, otherwise an exception condition will occur during run time. It is the programmer’s responsibility to ensure that the SELECT statement really does only return a single row. It is not possible for this to be checked in any way during the compilation of the statement.

Note:
In Interactive SQL or Dynamic SQL there is no such restriction on the number of rows which can be retrieved by a SELECT statement.

The host variable list should match the derived columns list in every aspect. There is an one-to-one correspondence between a derived column and a host variable. The basic type of the host variable must match that of the corresponding derived column. The relative number of items should be the same but strictly need not be. If there are insufficient host variables then data will be lost. If there are too many then the contents of the extra host variables will be undefined after the statement has completed. In either case a compiler warning is issued. Should an error occur during the execution of the query, the values in the host variables are undefined.

Inserting Single Rows

Data is put into a table using the INSERT statement. Data is inserted on a row by row basis. The source of the data can either come from literals or host variables (i.e., non-SQL derived data) or from a subquery (i.e., SQL derived data). By specifying non-SQL data, only one row may be inserted for one execution of the statement. If a subquery is used then as many rows as the subquery delivers are inserted for one execution of the statement. The subquery may not access the target table. There is no corresponding cursor-based INSERT statement.

Updating Rows

Data should be changed by using the UPDATE statement or more correctly the searched UPDATE statement. Data is changed on a row by row basis as identified by the search expression. Therefore, more than one row can be updated at a time.

Deleting Rows

Rows of data can be removed from the table by using the searched DELETE statement. Again this works on a row by row basis as identified by the search condition and, therefore, many rows can be deleted at once. If no search condition is specified, then all rows are identified and the table is cleared of all its data.

Level 1 or level 2 tables can not be the target of DELETE statements. Data from such tables can only be removed by deleting the associated level 0 row. In such a case the referencing level 1 and level 2 rows are automatically deleted with the level 0 row. This is analogous to a DELETE CASCADE in pure referential integrity terminology.

Cursor-based Statements

From the above description of non-cursor-based statements it can be seen that the retrieval and subsequent manipulation of more than one row is not possible without some other mechanism. The use of cursors addresses this restriction.

Declaring and Opening a Cursor

A cursor is declared in a DECLARE CURSOR statement along with the underlying query expression. The query expression defines a resultant table and so the cursor can be thought of as a pointer to a particular row of this table. At runtime, a static DECLARE CURSOR statement has no effect, it is purely a declaration for the SQL compiler.

Only once the cursor is opened by an OPEN CURSOR statement does the run time system conceptually establish the resultant table with the cursor pointing to just before the first row.

Other SQL compilers insist on the physical order of the DECLARE CURSOR statement followed by the open statement. This is because the information contained in the DECLARE CURSOR statement has to be ’attached’ to the OPEN statement. Adabas SQL Server does not have this restriction because it effectively has a multiple pass compilation phase. As the static DECLARE CURSOR statement has no effect in itself at run time, the logical order is also irrelevant.

Once the cursor has been opened, other statements can be executed against it.

Retrieving Data Using a Cursor

Data is retrieved from the resultant table using the FETCH statement. This statement specifies the cursor in question and a target buffer list which is similar to that of the single row SELECT statement. The target buffer list must match the projection list of the query expression. Each time the FETCH is executed it moves the cursor on one row and copies the values of the derived columns into the corresponding host variables of the target buffer list. For a newly opened cursor, the first row of the resultant table will be retrieved and the values will be made available to the application program in the host variables. The cursor now points to the first row. Each execution of the FETCH statement results in successive rows of the resultant table being retrieved.

Once all the rows have been fetched, the cursor is said to be exhausted and conceptually points past the last row. After the last row has been fetched, the next and any subsequent FETCH statements will result in a return code of +100 being issued by the runtime system. This needs to be checked for by the application program either explicitly or by specifying a WHENEVER statement with the NOT FOUND option. Once a cursor is exhausted, it should generally be closed.

The current row, as determined by the cursor's position in the resultant table can be amended by using a positioned UPDATE statement. The use of such a statement does not affect the position of the cursor. Only one row, i.e., the current one, can be amended by using this statement. However, by embedding the statement in the same loop as the FETCH statement, each row of the resultant table can be successively amended. For this reason, Adabas SQL Server permits the use of a FETCH statement without having to specify a target buffer.

Similarly, the current row can be removed from the underlying base table by executing a positioned delete statement against the cursor. After execution of the DELETE statement, the row no longer exists, but a FETCH statement must still be executed in order to position the cursor onto the next row.

Both the positioned UPDATE and DELETE statements are only valid if it is determined during compilation that the cursor can be updated as specified in the *Adabas SQL Server Reference Manual*, Chapter: **SQL Statements**.

Closing a Cursor

A cursor can be closed at any time but is generally closed once all rows have been fetched and the cursor is positioned past the last row. Closing the cursor means that the resultant table is discarded along with any internal resources required for the cursor’s processing. A cursor is also implicitly closed if a COMMIT or ROLLBACK statement is executed without the KEEPING ALL option.

Programming Logic for Cursor Usage

In general the FETCH, the positioned UPDATE or DELETE and the CLOSE statements should appear in the same compilation unit as the associated DECLARE CURSOR statement. This is because all the necessary checks to see if the statement is valid can be performed at compile time. Adabas SQL Server, however, does permit such statements to be physically contained in another compilation unit. This is intended to aid the modular design of the application. However, it should be noted that the necessary compile time checks are effectively performed at run time and could lead to a loss of performance.

An application program may contain many DECLARE CURSOR statements but each cursor identifier must be unique. For each DECLARE CURSOR statement there must be at least one OPEN CURSOR statement and it must be in the same compilation unit. As long as not compiled under ANSI compatibility mode the OPEN CURSOR statement need not physically follow the DECLARE CURSOR statement. There may also be many instances of the FETCH, positioned UPDATE or DELETE and the CLOSE CURSOR statements. As long as a cursor is closed, either explicitly or implicitly, it may be opened as many times as required.

Closing the cursor does not commit any changes made to the underlying base table. However, these changes are visible to the user, once the cursor is re-opened during the current transaction.

6

DYNAMIC SQL

Introduction

The principle difference between static and dynamic SQL statements lies in the point in time when the SQL statements are constructed and compiled.

Static SQL Statements

A static SQL statement is embedded in a host language program. The type of statement, the tables, views and columns referenced are known, and the format of the search conditions is fixed at the time the program is coded. Of course, by using host variables it is possible to postpone the search values until runtime. But at runtime it is not possible to change anything in the layout of the statement, e.g. the derived column list, the search conditions. A static SQL statement is compiled by Adabas SQL Server at precompile time.

Dynamic SQL Statements

A dynamic SQL statement is constructed at runtime. The whole statement, including the tables, views and columns referenced, the way the search condition is built up, etc. is only determined at runtime. A dynamic SQL statement is compiled at runtime.

The way to use dynamic SQL statements in a program is through some special SQL statements:

PREPARE, EXECUTE, EXECUTE IMMEDIATE and DESCRIBE.

There are also a number of SQL statements which are normally used as static SQL statements, but have an extended functionality for dynamic SQL:

DECLARE CURSOR, OPEN and FETCH.

Note:
The above statements are embedded in the application program like any other static SQL statement, However, they enable the use of dynamic SQL statements which are not embedded in the application program.

There are various methods of using dynamic SQL statements, mainly depending on the type of SQL statement, i.e.

SELECT or NON-SELECT statements,

and the degree of flexibility required. These methods are all described in this chapter, starting from the least flexible and, therefore, simplest form, and ending with the most flexible and therefore also most complex one.

General Aspects

Dynamic SQL Principles

The processing of a dynamic SQL statement consists of the following steps:

– A string containing the SQL statement is created. The application program has complete control over the contents of the string and therefore, the SQL statement is dynamic in nature.
– After the SQL statement has been constructed it needs to be passed on to Adabas SQL Server for compilation. This is done either using a PREPARE statement or using an EXECUTE IMMEDIATE statement. The compiled form of the dynamic SQL statement is called the prepared statement.
– If the statement was processed using an EXECUTE IMMEDIATE statement then it is not only compiled but also executed at the same time. The prepared form of the statement is not retained.
– If the statement was processed using a PREPARE statement, then the prepared statement can be executed using an EXECUTE statement or using cursor processing, as many times as required.
– Sometimes additional information about the prepared statement may be required before it can be executed. This is for example true for statements with an unknown derived column list. This information can be retrieved from Adabas SQL Server through an SQL descriptor area (SQLDA) using a DESCRIBE statement. An SQLDA can also be used to resolve host variable markers. Such information must be obtained prior to execution.

A special form of the PREPARE Statement is capable of storing the prepared statement in the catalog. The prepared statement is thus called a persistent procedure.

After the dynamic SQL statement has been prepared, it may be executed more than once, by using the same statement identifier. The lifespan of a prepared statement ends with the completion of the current session, unless it has been prepared as a persistent procedure. In that case, it may also be executed later on by different sessions. The existence of a prepared statement can be terminated by a DEALLOCATE PREPARE statement.

Dynamic versus Static SQL - Considerations

The choice between static and dynamic SQL is a choice of flexibility versus complexity and performance. It is easier to code static SQL statements into a program than to dynamically construct the SQL statements. In most cases, it will be possible to use static SQL, but there are some applications where the use of dynamic SQL is unavoidable. One of those applications is ADVANCED (ADVANCED Interactive Facilities) where the user may formulate almost any possible SQL statement. Obviously, such a requirement may only be resolved with dynamic SQL. But also in less obvious cases, it may be recommended to use dynamic SQL. If the number of static SQL statements that would be required for a certain application exceeds a manageable amount, dynamic SQL may be the solution.

In principle, the question to be answered is:

– Is it possible to define all necessary SQL statements in my application and will this be a manageable and feasible amount of coding?

  If the answer is no, then dynamic SQL needs to be considered.

Once dynamic SQL has been identified as a viable possibility, the particular variation or degree of complexity of dynamic SQL must be decided upon. The following questions to be answered are:

– Must the program contain dynamically constructed SELECT statements or not?
– If SELECT statements are dynamically constructed, does the derived column list vary dynamically?
– Are host variable markers going to be used, and if so, does the number and type of host variable markers vary dynamically per prepared statement or not?

Another point of consideration, when deciding whether to use dynamic SQL, is the issue of performance. Obviously, compiling SQL statements at runtime has an influence on the overall performance of the execution of that SQL statement. The compilation of an SQL statement also includes the access of information stored in the catalog, like table and column descriptions. These descriptions are buffered, but the possibility exists that additional database requests need to be issued.

The consequence of the fact that a dynamic SQL statement is compiled at runtime is also that the statement is compiled with more accurate, i.e. more current information concerning the existence of indices and other optimization information.

Using dynamic SQL, also means that syntactical and semantical errors are only detected during runtime. This means that the PREPARE and EXECUTE IMMEDIATE statements may return an SQLCODE indicating a syntactical or semantical error.

Limitations

The following SQL statements can not be used as dynamic SQL statements, i.e. they can not be prepared or executed:

BEGIN DECLARE,	CLOSE,	DEALLOCATE PREPARE	DECLARE,
DESCRIBE,	DISCONNECT,	END DECLARE,	EXECUTE,
EXECUTE IMMEDIATE,	FETCH,	INCLUDE,	OPEN,
PREPARE,	WHENEVER.		

Dynamic SQL may require the use of addresses and pointers within the application program. It may also require dynamically obtained memory. As a consequence, dynamic SQL is not completely supported from within a COBOL application program. Therefore, the use of SQL descriptor areas is not supported directly in COBOL.

NON-SELECT Statements

The simplest form of dynamic SQL programs do not contain SELECT statements. In such a case, there is no resultant table and no data has to be passed back to the application program.

There are two ways to execute a NON-SELECT SQL statement dynamically: one is using the EXECUTE IMMEDIATE statement the other is using the PREPARE and EXECUTE statements.

Using EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement takes only one parameter. This parameter must be a character string which contains the dynamic SQL statement. The string has to be properly constructed by the application program. The dynamic SQL statement is then compiled and immediately executed. The compiled form of the SQL statement (the prepared statement) is discarded after execution.

Example:

```
EXEC SQL
EXECUTE IMMEDIATE :dyn_sql_statement;
```

dyn_sql_statement	is a character string containing the dynamic SQL statement.

All SQL statements which can be PREPAREd can be given to the EXECUTE IMMEDIATE statement except for a SELECT.

Note:
If the string representing the dynamic SQL statement can not be compiled, e.g., due to a syntax error, the SQLCODE will indicate this error after execution of the EXECUTE IMMEDIATE statement.

Note:
It is not possible to use host variable markers in the dynamic SQL statement when using the EXECUTE IMMEDIATE statement.

Using PREPARE and EXECUTE

By using the PREPARE and EXECUTE method, the compilation and the execution of the dynamic SQL statement is split over two statements. The dynamic SQL statement is again contained in a string and is constructed by the application program. The PREPARE statement initiates the compilation of the dynamic SQL statement, and the EXECUTE statement executes it.

The result of a PREPARE statement is a statement ready for execution. This prepared statement is identified by an SQL statement identifier which can either be set by the user as a fixed identifier or is generated by Adabas SQL Server when a host variable has been specified. The prepared statement is kept for later execution. If it is intended that the statement identifier is to be generated by Adabas SQL Server it is necessary to initialize the variable with blanks or an empty string prior to execution. Otherwise, Adabas SQL Server will use the actual (non-blank) value of the variable. The same statement identifier must then be specified with the EXECUTE statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM :dyn_sql_statement;

EXEC SQL
EXECUTE STATEMENT_ID;
```

dyn_sql_statement: is a character string containing the dynamic SQL statement.

All SQL statements except those mentioned earlier in this chapter in section: **Limitations** can be PREPAREd by the PREPARE statement. Only NON-SELECT statements can be executed by the EXECUTE statement.

It is possible for the dynamic SQL statement to contain host variable markers. For a general discussion of this subject, see the section: **Using Host Variable Markers** later in this chapter.

Note:
If the string representing the dynamic SQL statement can not be compiled, e.g. due to a syntax error, the SQLCODE will indicate this error upon return from the PREPARE statement.

Summary

A program which issues dynamic NON-SELECT statements must include the following steps:

$\boxed{1}$ Construct the dynamic SQL statement.
The dynamic SQL statement must be constructed as a character string. The process of creating this string is application-dependent. It may be that the user enters the SQL statement or part thereof direct using a terminal, or that the application program dynamically builds up the statement based on other sources of information.

$\boxed{2}$ PREPARE and EXECUTE the dynamic SQL statement.
One of two methods must be used, either the EXECUTE IMMEDIATE method or the PREPARE and EXECUTE method to execute the dynamic SQL statement.

Variable input values, as specified by a host variable marker '?' may have to be provided by specifying an USING clause and specifying an SQLDA in an EXECUTE statement.

$\boxed{3}$ Check the result.
All the statements involved, the EXECUTE IMMEDIATE, the PREPARE and the EXECUTE statement can result in errors which are reported back by Adabas SQL Server to the application program using the SQLCODE in the SQLCA. This has to be handled just like any other error situation.

SELECT Statements

SELECT statements can only be dynamically executed by using a separate PREPARE statement and the dynamic cursor logic. The statements DECLARE CURSOR, OPEN, FETCH and CLOSE must be used.

There are two ways to execute a SELECT statement dynamically. The method to use depends on the characteristics of the SELECT statements to be processed:

– If the derived column list of the SELECT statement has a constant format, i.e. the number of elements in the resultant table and their data types stay constant, the fixed derived column list method can be used.
– If the derived column list varies, the varying derived column list method must be used. In the latter case, an SQL descriptor area (SQLDA) is required.

Note:
It is not possible to dynamically execute a single-row SELECT.

Fixed Derived Column List Method

Dynamic SELECT statements with a fixed derived column list produce resultant tables which have a fixed layout, i.e. the number of columns is the same and the data type of each column is fixed and known at the time the application program is precompiled.

The fixed derived column list method assumes that the dynamically created SELECT statements have a fixed derived column list, so that a normal FETCH statement can be used to retrieve the rows of the resultant table. This FETCH statement requires that the columns of the resultant table are each assigned to specific hard-coded host variables. As these host variables have to be known at precompilation time, the layout of the derived column list must be determined at the same time. All other clauses of the SELECT statement, the FROM clause, the WHERE clause, etc., can vary dynamically every time the statement is prepared. This means that the fixed derived column list method can be used in those cases where the result and format of the query is known, but the search criteria etc. can vary to such a degree that the rest of the query needs to be constructed dynamically at runtime.

The fixed derived column list method consists of a number of steps:

PREPARE

The entire SELECT statement must be constructed in a host variable which is passed on to Adabas SQL Server as a parameter of a PREPARE statement. The application needs to ensure that the resulting format of the query can not vary dynamically.

Example:

```
EXEC SQL
PREPARE statement_id FROM :dyn_sql_statement;
```

dyn_sql_statement is a character string containing the dynamic SQL statement.

Note, that as an Adabas SQL Server extension a host variable may be used to identify a statement. If so, Adabas SQL Server returns a unique value in this variable which must have been initialized with blanks upon return from the PREPARE statement. This value is then to be used for all subsequent references to the prepared statement.

Example:

```
EXEC SQL
PREPARE :statement_id FROM :dyn_sql_statement;
```

DECLARE

The prepared statement must then be associated with a cursor. This can either be achieved explicitly by means of a dynamic DECLARE CURSOR statement or implicitly by an OPEN statement. The dynamic DECLARE CURSOR statement is similar to the static DECLARE, but instead of specifying the SELECT statement, it specifies the statement identifier as defined in the PREPARE statement,thus associating the prepared SELECT statement with the cursor. Such a DECLARE statement may also be executed prior to the associated PREPARE statement or may be omitted altogether, if the associated OPEN statement specifies the SQL statement identifier instead.

Example:

```
EXEC SQL
DECLARE ABC CURSOR FOR statement_id;
```

Note:
Alternatively, an Adabas SQL Server extension allows a host variable to be used to identify the cursor. This host variable must be initialized with a suitable value by the application program before use.

Example:

```
EXEC SQL
DECLARE :cursor_name CURSOR FOR STATEMENT_ID;
```

Note:
If in the original PREPARE statement, a host variable was used to express the statement identifier, then a host variable containing the same assigned value must be used here in order to identify the statement. If used at all, the DECLARE statement must be executed after the PREPARE statement.

Example:

```
EXEC SQL
DECLARE ABC CURSOR FOR :statement_id;
```

It can be seen that the dynamic DECLARE CURSOR statement differs from its normal static counterpart in that during runtime the statement is of significance, i.e., the prepared statement is associated to the particular cursor. The order of execution is important in a dynamic SQL application. Once the PREPARE and then the DECLARE CURSOR statements have been successfully executed, other cursor associated statements can be executed in the normal way, except that the cursor may need to be expressed as a host variable. The normal OPEN, FETCH, CLOSE logic is still applicable.

OPEN

The cursor associated with the dynamic SELECT statement is opened by means of an OPEN statement. Note that the cursor name may be expressed as a host variable.

Example:

```
EXEC SQL
OPEN ABC;
```

If the SELECT statement contains host variable markers, the parameters can be submitted by the USING clause or the USING DESCRIPTOR clause. For more details see the section: **Using Host Variable Markers**, later in this chapter.

Example:

```
EXEC SQL
OPEN ABC USING :hv1, :hv2;
```

or using an SQL descriptor area:

```
EXEC SQL
OPEN ABC USING DESCRIPTOR :input_sql_da;
```

In addition, an SQL statement identifier can be specified in case the DECLARE CURSOR statement has been omitted.

Example:

```
EXEC SQL
OPEN ABC CURSOR FOR :statement_id;
```

FETCH

As the format of the derived column list of the dynamic SELECT statement is constant, the FETCH statement can be identical to the static case. For each one of the columns in the resultant table, a host variable needs to be specified which is of a compatible data type.

Note:
Although the format of the derived column list does not vary dynamically, it is still not ‘visible’ to the Adabas SQL Server compiler. Therefore, the compiler can not actually check the validity of the FETCH statement and in particular its target buffer list. Naturally, at run time, such checks are performed.

Note:
An attempt to fetch a dervied column of type binary, using a dynamically prepared select statement and a fetch statement which is identical to the static counterpart, will always result in an error condition. This is because, upon pre-compiling the fetch statement, the fact that a character host variable is going to be used for the retrieval of a derived column of type binary is not foreseeable. If a derived column of type binary is to be retrieved using a dynamically prepared select statement, even if has a fixed derived column list, then a fetch statement which uses a descriptor area must be used.

Example:

```
EXEC SQL
FETCH ABC INTO :hv1, :hv3;
```

CLOSE

The closing of the cursor is identical to the static case. By executing the CLOSE statement, all resources reserved by the cursor are released.

Example:

```
EXEC SQL
CLOSE ABC;
```

Likewise, once closed, the cursor may simply be re-opened again.

Summary

A program which issues dynamic fixed derived column list SELECT statements must include the following steps:

$\boxed{1}$ Construct the dynamic SELECT statement. The statement is constructed as a character string in a similar fashion to NON-SELECT dynamic statements. However, the derived column list must remain fixed and its format must have been determined at compile time.

$\boxed{2}$ PREPARE the dynamic SQL statement.

$\boxed{3}$ Optionally, DECLARE a cursor for the prepared statement using a dynamic DECLARE CURSOR statement.

$\boxed{4}$ OPEN the cursor in a similar way to a normal static cursor.

$\boxed{5}$ Variable input values, as specified by a host variable marker '?' may have to be provided by using an USING clause appended to the OPEN statement and specifying an SQLDA.

$\boxed{6}$ FETCH from the cursor as required until all rows have been processed.

$\boxed{7}$ CLOSE the cursor.

Varying Derived Column List Method

Dynamic SELECT statements with a varying derived column list are SELECT statements which produce resultant tables which have differing formats, i.e., the format of the resultant table is dynamically specified and may vary from instance to instance.

This method is more complicated than the one of using a fixed derived column list but is only required if indeed the format of the possible resultant tables can vary. Otherwise the fixed derived column list method may be used. In order to be able to use the varying list method, the application program must be able to acquire dynamic storage and be able to manipulate pointers or addresses. This obviously limits the use of this method to those host languages which provide these facilities, or appropriate specially written sub-routines are needed.

The application program needs to get information about the layout of the resultant table for a varying derived column list statement as target buffers must be dynamically provided. Adabas SQL Server provides special functions to aid the application program in this task. This information is passed to the program using an SQL descriptor area or an SQLDA.

Note:
As the use of an SQLDA involves the use of dynamically acquired memory and addresses, the varying derived column list method can not be directly used from a COBOL program. Specially written sub-routines are needed.

PREPARE

The SELECT statement must be constructed in a host variable which is passed on to Adabas SQL Server as a parameter to a PREPARE statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM :dyn_sql_statement;
```

Note that alternatively an Adabas SQL Server extension allows a host variable to be used to identify the statement. If so, Adabas SQL Server returns a unique value in this variable which must have been initialized with blanks upon return from the PREPARE statement. This value is then to be used for all subsequent references to the prepared statement.

Example:

```
EXEC SQL
PREPARE :statement_id FROM :dyn_sql_statement;
```

DECLARE

The prepared statement must then be associated with a cursor. This can either be achieved explicitly by means of a dynamic DECLARE CURSOR statement or implicitly by an OPEN statement. The dynamic DECLARE CURSOR statement is similar to the static DECLARE, but instead of specifying the SELECT statement, it specifies the statement identifier as defined in the PREPARE statement,thus associating the prepared SELECT statement with the cursor. Such a DECLARE statement may also be executed prior to the associated PREPARE statement or may be omitted altogether, if the associated OPEN statement specifies the SQL statement identifier instead.

Example:

```
EXEC SQL
DECLARE ABC CURSOR FOR statement_id;
```

Note:
Alternatively, an Adabas SQL Server extension allows a host variable to be used to identify the cursor. This host variable must be initialized with a suitable value by the application program before use.

Example:

```
EXEC SQL
DECLARE :cursor_name CURSOR FOR statement_id;
```

Note:
If in the original PREPARE statement, a host variable was used to express the statement identifier, then a host variable containing the same assigned value must be used here in order to identify the statement. If used at all, the DECLARE statement must be executed after the PREPARE statement.

Example:

```
EXEC SQL
DECLARE ABC CURSOR FOR :statement_id;
```

DESCRIBE

A description of the resulting format of the query may now be retrieved from Adabas SQL Server. This is done using an SQLDA and a DESCRIBE statement. Note, that the functionality of the DESCRIBE statement can also be achieved by using an INTO clause in the PREPARE statement.

Example:

```
EXEC SQL
DESCRIBE STATEMENT_ID INTO :output_sqlda;
```

After successful execution of the DESCRIBE statement, the SQLDA contains detailed information concerning the resulting format of the SELECT statement.

The total number of columns and the particular type of each column will be supplied. The application program must act on this information by dynamically supplying an appropriate target buffer for each of the columns. The address of each target buffer must be written into the SQLDA. In addition, an associated indicator value may have to be assigned.

OPEN

The cursor associated with the dynamic SELECT statement is opened by means of an OPEN statement. Note, that the cursor name may be expressed as a host variable.

Example:

```
EXEC SQL
OPEN ABC;
```

If the SELECT statement contained host variable markers, the parameters can be submitted by the USING clause or the USING DESCRIPTOR clause. For more details see the section: **Using Host Variable Markers**, later in this chapter.

Example:

```
EXEC SQL
OPEN ABC USING :hv1, :hv2;
```

or using an SQL input descriptor area:

```
EXEC SQL
OPEN ABC USING DESCRIPTOR :input_sqlda;
```

In addition, an SQL statement identifier can be specified in case the DECLARE CURSOR statement has been omitted.

Example:

```
EXEC SQL
OPEN ABC CURSOR FOR :statement_id
     USING DESCRIPTOR :input_sqlda;
```

FETCH

The FETCH statement must be executed in conjunction with the SQLDA that has been constructed for this particular dynamic SELECT statement. The resulting values are copied into the locations specified in the corresponding column description in the SQLDA. Note that Adabas SQL Server can only assume that such locations are of sufficient size to accommodate the returned data. It is the responsibility of the application program to properly provide such locations. Using a DESCRIBE statement greatly simplifies this task.

Example:

```
EXEC SQL
FETCH ABC USING DESCRIPTOR :output_sqlda;
```

CLOSE

The closing of the cursor is identical to the static case. By executing the CLOSE statement, all resources reserved by the cursor are released.

Example:

```
EXEC SQL
CLOSE ABC;
```

Likewise, once closed, the cursor may simply be re-opened again within the current transaction.

Summary

A program which issues dynamic varying derived column list SELECT statements must include the following steps:

$\boxed{1}$ Construct the dynamic SELECT statement. The statement is constructed as a character string in a similar fashion to NON-SELECT dynamic statements. The nature of the SELECT statement is completely determined by the application program.

$\boxed{2}$ PREPARE the dynamic SQL statement.

$\boxed{3}$ Allocate and build an appropriate SQLDA. This may be done using a DESCRIBE statement. Assign appropriate target buffers.

$\boxed{4}$ Optionally, DECLARE a cursor for the prepared statement using a dynamic DECLARE CURSOR statement.

$\boxed{5}$ OPEN the cursor in a similar way to a normal static cursor.

$\boxed{6}$ Variable input values, as specified by a host variable marker ‘?’ may have to be provided by supplying a USING clause appended to the OPEN statement specifying an SQLDA.

$\boxed{7}$ FETCH from the cursor as required until all rows have been processed. The output SQLDA must be specified in order to receive retrieved data.

$\boxed{8}$ CLOSE the cursor.

Using Host Variable Markers

A dynamic SQL statement can not contain host variables directly. It is, however, possible to provide a dynamic SQL statement after it has been prepared with value parameters at execution time. The dynamic statement must then contain a host variable marker for every host variable. A host variable marker is represented by a question mark (?).

Example:

```
EXEC SQL
PREPARE statement_id FROM “DELETE FROM CRUISE WHERE CRUISE_ID = ?”;

EXEC SQL
EXECUTE STATEMENT_ID USING :cruise_id;
```

The dynamic DELETE statement contains one host variable marker, so the USING clause in the EXECUTE statement contains one host variable. The host variable cruise_id is used to provide a parameter for the PREPAREd DELETE statement. It is, as if the following static SQL statements were executed:

```
EXEC SQL
DELETE FROM CRUISE WHERE CRUISE_ID = :cruise_id;
```

Obviously the host program can repeatedly re-execute the PREPAREd statement by supplying a fresh value in the host variable with each iteration.

Restrictions

In principle, a host variable marker may appear everywhere in a statement where a host variable may appear. Because of the nature of dynamic SQL, however, there are certain restrictions. The following rules apply:

– A host variable marker is not allowed to appear in a derived column list
– Only one operand of a diadic arithmetic operator or comparison operator may be a host variable marker, e.g. ? = ? or ? * ? is not allowed.
– The first two operands of a BETWEEN or IN operator can not be host variable markers, e.g. ? IN (?,...) is not allowed, however, 5 + ? IN (?,...) is allowed.

The reason for these restrictions is that at the time the dynamic SQL statement is compiled, the data type of each one of the host variable markers needs to be determined. In the cases described above this can not be done.

Different Methods

As with SELECT statements, there are different methods to deal with host variable markers. One method can be applied in situations where the number of host variable markers is constant and their type is known and also constant. Another method must be applied if the number of host variable markers varies. Both methods are described in the following sections.

Constant Number of Host Variable Markers

When the number and data types of the host variable markers are constant and known at compilation time in a dynamic SQL statement, a matching set of host variables can be defined to be used to provide values prior to the execution of the prepared dynamic statement. These host variables can be specified in the USING clause of either an EXECUTE or an OPEN statement.

NON-SELECT Statements

For NON-SELECT statements, the host variables used to resolve the host variable markers must be specified in the USING clause on the EXECUTE statement. The host variables in the USING clause must be specified in the same order as the host variable markers were specified in the dynamic SQL statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM “INSERT INTO CRUISE VALUES (?,?,?,?,?)”;

EXEC SQL
EXECUTE statement_id USING :hv1,:hv2,:hv3,:hv4,:hv5;
```

SELECT Statements

For SELECT statements, the host variables used to resolve the host variable markers must be specified in the USING clause appended to the OPEN statement. The host variables in the USING clause must be specified in the same order as the host variable markers were specified in the dynamic SQL statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM “SELECT CRUISE_ID FROM CRUISE WHERE CRUISE_ID = ?”;

EXEC SQL
DECLARE ABC CURSOR FOR statement_id;

EXEC SQL
OPEN ABC USING :hv1;
```

Varying Number of Host Variable Markers

When the number and data types of the host variable markers varies with each dynamically PREPAREd statement and/or their data type can not be pre-determined, it is not possible to define a matching set of host variables to provide values prior to the execution of the PREPAREd statement.

In that case, the application program needs to get information about the host variable markers in a prepared statement dynamically. The application program can either do this itself by analyzing the dynamic SQL statement, or Adabas SQL Server can provide this information in an SQL descriptor area using a PREPARE or DESCRIBE statement.

Upon return from an appropriate PREPARE or DESCRIBE statement, Adabas SQL Server will have filled the SQLDA with information about each one of the host variable markers. This information can then be used by the application program to allocate and assign host variables for each one of the host variable markers. Note that it is possible at this stage to change the data type description of a host variable in the SQLDA. Be aware that this may lead to runtime errors if the data type of a host variable is changed to one that is incompatible with the one established by Adabas SQL Server.

Note:
Such an input SQLDA is a separate instance of an output SQLDA but has the same structure.

NON-SELECT Statements

For NON-SELECT statements, the input SQLDA must be supplied with the EXECUTE statement. The host variables described in the SQLDA must be specified in the same order as the host variable markers were specified in the dynamic SQL statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM “INSERT INTO CRUISE VALUES (?,?,?,?,?)”;

EXEC SQL
DESCRIBE statement_id INTO INPUT :input_sqlda;

EXEC SQL
EXECUTE statement_id USING DESCRIPTOR :input_sqlda;
```

SELECT Statements

For SELECT statements, the input SQLDA must be supplied with the OPEN statement. The host variables in the input SQLDA must be specified in the same order as the host variable markers were specified in the dynamic SQL statement.

Example:

```
EXEC SQL
PREPARE statement_id
    FROM “SELECT CRUISE_ID FROM CRUISE WHERE CRUISE_ID = ?”;

EXEC SQL
DESCRIBE statement_id INTO INPUT :input_sqlda;

EXEC SQL
DECLARE ABC CURSOR FOR statement_id;

EXEC SQL
OPEN ABC USING DESCRIPTOR :input_sqlda;
```

Summary

A program which issues dynamic statements which contain host variable markers must cover the following steps:

$\boxed{1}$ Construct the dynamic SQL statement.
The dynamic SQL statement must be constructed as a character string, which will contain host variable markers (?).

$\boxed{2}$ Prepare the dynamic SQL statement.
The dynamic SELECT statement always has to be prepared using a PREPARE statement.

$\boxed{3}$ Establish information about the host variable markers.
If the host variable markers are constant in number and data type, host variables may be statically supplied. Otherwise an INTO input clause in the PREPARE or DESCRIBE statement must be used in order to obtain the information about the host variable markers. Variables must then be dynamically allocated.

$\boxed{4}$ a: Either assign values to any static host variables.
b: Or load the input SQLDA.
If the host variables are assigned dynamically, the SQLDA has to be supplied with information about them. The host variables themselves must have appropriate values assigned to them

$\boxed{5}$ Execute the dynamic SQL statement.
A USING clause containing either references to the static host variables or the input SQLDA is appended to either the EXECUTE statement or the OPEN statement as required.

Updating a Dynamic Cursor

In dynamic SQL, it is not possible for Adabas SQL Server to determine automatically if a cursor has been established in order to be updated or not. Note that there are important performance consequences based on this information. Two methods of letting Adabas SQL Server know that a cursor is FOR UPDATE or FOR FETCH ONLY are available:

– The first possibility is to set a system wide default using Adabas SQL Server parameters. All dynamic cursors will be assumed to be of this default type.

– The second possibility is to use the FOR UPDATE and FOR FETCH ONLY clauses appended to the dynamic SELECT statement. Such an explicit specification overrides any default.

There is a difference in the positioning of both clauses compared to static SQL, where the clauses can be specified only in conjunction with the DECLARE CURSOR statement. In dynamic SQL, they can only be specified in conjunction with a SELECT statement. In case a DELETE or UPDATE statement is issued against a cursor which is not FOR UPDATE, a negative SQLCODE is returned to the calling program.

Example:

```
EXEC SQL
PREPARE statement_id FROM “SELECT CRUISE_ID
    FROM CRUISE WHERE CRUISE_ID = ? FOR UPDATE”;
```

Note, that the second method overrides the system default in either case.

There are again two methods of using positioned UPDATE or DELETE statements:

– the first method uses the normal static positioned UPDATE and DELETE statements,

– the second method requires that the UPDATE or DELETE statements are dynamically constructed.

Static Method

Using the static method, the UPDATE or DELETE statement is statically embedded in the application program. This implies that the table to be updated must be statically specified in the UPDATE or DELETE statement and can not be dynamically substituted with another table specification.

Example:

```
EXEC SQL
PREPARE statement_id
    FROM “SELECT CRUISE_ID FROM CRUISE WHERE CRUISE_ID = ? FOR UPDATE”;

EXEC SQL
DECLARE ABC CURSOR FOR statement_id;

EXEC SQL
OPEN ABC;

EXEC SQL
FETCH ABC INTO :cruise_id;

EXEC SQL
UPDATE CRUISE SET CRUISE_NAME = :cruise_name WHERE CURRENT OF ABC;
```

Adabas SQL Server will check that the table specified in the SELECT statement is indeed consistent with the desired target table. If not, a runtime error will be produced.

The FOR UPDATE clause of the SELECT statement or the appropriate setting of the system default is required.

Using the static method, it is possible to specify the cursor name in all statements using a host variable.

Dynamic Method

Using the dynamic method, the positioned UPDATE or DELETE statement is dynamically constructed. The normal methods of EXECUTE IMMEDIATE or PREPARE and EXECUTE can be used. Using this method, there are no restrictions on the dynamic SELECT statement regarding the choice of target tables.

Example:

```
EXEC SQL
PREPARE statement_id
    FROM “SELECT CRUISE_ID FROM CRUISE WHERE CRUISE_ID = ? FOR UPDATE”;

EXEC SQL
DECLARE ABC CURSOR FOR statement_id;

EXEC SQL
OPEN ABC;

EXEC SQL
FETCH ABC INTO :cruise_id;

EXEC SQL
PREPARE UPDATE_STATEMENT
    FROM “UPDATE CRUISE SET CRUISE_NAME = :cruise_name
    WHERE CURRENT OF ABC”;

EXEC SQL
EXECUTE UPDATE_STATEMENT;
```

Adabas SQL Server will check if the dynamic cursor allows updates and if the update is on the same table as specified by the cursor, otherwise a runtime error will be produced.

Using the dynamic method it is **not** possible to specify the cursor name as a host variable in UPDATE or DELETE statements, i.e. a ‘WHERE CURRENT OF ?’ is not valid.

Dynamic SQL with Persistent Procedures

When dynamic SQL statements are compiled using a normal PREPARE statement they result in prepared statements that can only be executed within the same session which issued the PREPARE statement. This means that these prepared statements are temporary and can not be shared. For an application that issues the same SQL statements dynamically and is used by a large number of users simultaneously, this can lead to an unnecessarily high amount of superfluous compilation processes and resource usage. By using the possibility of persistent procedures, dynamically prepared statements can be made permanent and can then be shared amongst an unlimited number of sessions.

Creating a Persistent Procedure

A dynamic SQL statement is made into a persistent procedure by specifying the MODULE clause on a PREPARE statement. Normally, with the PREPARE statement a dynamic SQL statement is compiled and turned into a prepared statement which is given an SQL statement identifier. Using the MODULE clause, the prepared statement is not given a temporary SQL statement identifier, but a permanent one instead. Each persistent procedure has to be given an unique identification consisting of a 27 character module name and a 5 character procedure name. Both names are concatenated to form the identification of the persistent procedure as stored in the catalog. In addition a version indicator must be specified, which is used to distinguish different versions of the same persistent procedure from each other. The version indicator specified when using a persistent procedure must conform with the version indicator as it present in the catalog. By re-definition of the module name, a more finer subdivision of persistent procedures can be achieved.

Example:

```
EXEC SQL
     PREPARE MODULE ”esq_test” PROCEDURE ”00001” VERSION :tmstmp
     FROM ”SELECT CRUISE_ID FROM CRUISE
           WHERE CRUISE_ID = ? FOR UPDATE”;
```

tmstmp	is a variable which contains an 8 byte binary value.

Using a Persistent Procedure

Once a persistent procedure has been established it can be used by any user session knowing the name and version of the procedure. In principle, the MODULE clause can be used with any SQL statement instead of an SQL statement identifier. Specifically, this means with the DECLARE CURSOR, DESCRIBE, OPEN and EXECUTE statements. In order to minimize the number of statements required in cursor processing, the DECLARE CURSOR statement is optional. By specifying the MODULE clause with the OPEN statement the DECLARE CURSOR statement is not required.

Ideally, an application using persistent procedures would first attempt to execute a persistent procedure by means of either an OPEN or EXECUTE statement. If the execution fails due to the fact that the persistent procedure is not present in the catalog or has the wrong version, the persistent procedure can be established using a PREPARE statement.

Example:

```
EXEC SQL
     DESCRIBE MODULE ”esq_test” PROCEDURE ”00001” VERSION :tmstmp
     INTO :output_sqlda INPUT :input_sqlda;

EXEC SQL
     OPEN ABC CURSOR FOR MODULE ”esq_test” PROCEDURE ”00001”
          VERSION :tmstmp
     USING DESCRIPTOR :input_sqlda;

EXEC SQL
     FETCH ABC USING :output_sqlda;

EXEC SQL
     CLOSE ABC;
```

Note:
It is also possible to omit the DECLARE CURSOR statement for non persistent dynamic SQL, i.e. when using an SQL statement identifier. The SQL statement identifier may be specified with the OPEN statement directly as well.

Deleting a Persistent Procedure

A particular or a set of persistent procedures can be deleted by means of a DEALLOCATE PREPARE statement. A DEALLOCATE PREPARE statement also specifies the MODULE clause, but without the version indicator. By specifying a procedure name, one specific persistent procedure is deleted, by omitting the procedure name, all persistent procedures with the same module name are deleted.

Example

```
EXEC SQL
    DEALLOCATE PREPARE MODULE ”esq_test” PROCEDURE ”0001

EXEC SQL
    DEALLOCATE PREPARE MODULE ”esq_test”;
```

SQL Descriptor Area (SQLDA)

General Information

An SQL descriptor area is used as a communication area between an application program and Adabas SQL Server for dynamic SQL. It is used for communicating information between Adabas SQL Server and the application program in both directions.

The information on a dynamic SQL statement that can be retrieved from Adabas SQL Server by an application program using an SQLDA originates from either of two sources:

– **OUTPUT SQLDA:**
  The derived column list of a dynamic SELECT statement. The application program can retrieve information about the layout of the resulting format of a SELECT statement. The information comprises a list of elements where each element describes the corresponding derived column. An SQLDA describing this type of information is called an output SQLDA. The information is assigned to the output SQLDA by either an extended PREPARE statement (example 1a) or a separate DESCRIBE statement (example 1b). The keyword OUTPUT is the default and therefore optional.

  Example 1a:

  ```
  EXEC SQL
  PREPARE statement_id INTO :output_sqlda FROM dyn_sql_statement;
  ```

  Example 1b:

  ```
  EXEC SQL
  PREPARE statement_id FROM :dyn_sql_statement;
  EXEC SQL
  DESCRIBE statement_id INTO OUTPUT :output_sqlda;
  ```

- **INPUT SQLDA:**
  The host variable markers in a dynamic SQL statement.
  The application program can retrieve information about all host variable markers used in a dynamic SQL statement. The information comprises a list of elements where each element describes the corresponding host variable marker. An SQLDA describing this type of information is called an input SQLDA. The information is assigned to the input SQLDA by either an extended PREPARE statement (example 2a) or a separate DESCRIBE statement (example 2b). The keyword INPUT is mandatory.

  Example 2a:

  ```
  EXEC SQL
  PREPARE statement_id INTO INPUT :input_sqlda FROM dyn_sql_statement;
  ```

  Example 2b:

  ```
  EXEC SQL
  PREPARE statement_id FROM :dyn_sql_statement;
  EXEC SQL
  DESCRIBE statement_id INTO INPUT :input_sqlda;
  ```

Note, that both input and output SQLDAs can be specified in the same PREPARE and DESCRIBE statements if desired. However, one SQLDA can not be used for both an input and an output SQLDA simultaneously.

Once Adabas SQL Server has filled an SQLDA with this information the application program must provide a host variable reference for each element. This must be done prior to the execution of the PREPAREd statement.

Corresponding to the two types of SQLDAs two types of host variable references must be supplied.

– Target host variables for receiving resultant data.
  The elements of an output SQLDA associated with a PREPAREd SELECT statement each describe the expected format of the data to be received. The application program must assign to each element a suitable host variable which is capable of receiving the expected data. Adabas SQL Server can now determine where to copy the resulting data to by means of the pointer reference in each element. Such an output SQLDA is only used in conjunction with a FETCH statement.

  Example 1:

  ```
  EXEC SQL
  FETCH ABC USING DESCRIPTOR :output_sqlda;
  ```

– Host variables as host variable marker replacements.
  The elements of an input SQLDA each describe the expected format of any additional parameters required by the PREPAREd statement as represented by host variable markers. The application program must assign a suitable host variable to each element of the input SQLDA and each host variable must be loaded with the desired value before execution of the PREPARED statement. Such an input SQLDA is used in conjunction with either an OPEN statement (example 2a) or an EXECUTE statement (example 2b).

  Example 2a:

  ```
  EXEC SQL
  OPEN ABC USING DESCRIPTOR :input_sqlda;
  ```

  Example 2 b:

  ```
  EXEC SQL
  EXECUTE statement_id USING DESCRIPTOR :input_sqlda;
  ```

The SQLDA Structure

Exactly the same structure is used for both, input and output SQLDA. It consists of two distinct parts:

– a header containing general information about the PREPAREd statement,

– a consecutive list of elements corresponding to fields in the derived column list or the host variable markers.

The whole structure consists of the four fields of the SQLDA header immediately followed as many occurrences of the sqlvar structure as stated in the sqln field.

Field	**Description**
sqldaid	An eigth-bytes character string containing the constant SQLDA, serves as an eye catcher for easier memory dump interpretation.
sqldabc	A four-bytes integer field containing the total length of the SQLDA in bytes, i.e. the length of the header plus the length of the variable descriptor elements multiplied by the number of available elements (sqln)
sqln	A two-bytes integer field containing the total number of variable descriptor elements available in the SQLDA.
sqld	A two-bytes integer field containing the total number of variable descriptor elements filled during the execution of a DESCRIBE statement.
sqlvar	An array containing sqln variable descriptor elements. **sqltype** A two-bytes integer field containing the data type of the required/specified host variable and whether there is an INDICATOR variable present or not. **sqllen** A two-bytes integer field containing the length of the required/specified host variable. The interpretation of this field depends on the data type. **sqldata** containing the typespecific pointer to the host variable which is to receive or which should contain the data.

Field	Description
	sqlind A two-bytes integer field containing the typespecific pointer to the host variable acting as an indicator value, if one is required. **sqlname** An array of 32 bytes containing the derived column label of the resulting column. The first two bytes contain the length of the label.

The sqlname field is only relevant for an output SQLDA and only in the particular case of the corresponding derived column having a derived column label.

The sqltype field is set by Adabas SQL Server to reflect the particular type of the required field.

In addition the sqllen field is also set by Adabas SQL Server depending on the value assigned to the sqltype field. This field specifies the required size of the host variable.

The sqltype field also specifies whether a null indicator variable is required or is supplied. This is shown by the type value being increment by 1.

The following table lists the various values and combinations:

Type Name	Value	Data Type	SQLLEN
SQL_TYP_CHAR SQL_TYP_NCHAR	452 453	fixed length character string	length of string in bytes
SQL_TYP_CSTR SQL_TYP_NCSTR	460 461	null terminated C string	length of string in bytes, including the null terminator
SQL_TYP_FLOAT SQL_TYP_NFLOAT	480 481	floating point	set to 4 for single precision set to 8 for double precision
SQL_TYP_DECIMAL SQL_TYP_NDECIMAL	484 485	decimal	first byte specifies the precision second byte specifies the scale

Type Name	Value	Data Type	SQLLEN
SQL_TYP_INTEGER SQL_TYP_NINTEGER	496 497	integer	set to 4, equating to number of bytes
SQL_TYP_SMALL SQL_TYP_NSMALL	500 501	small integer	set to 2, equating to number of bytes
SQL_TYP_BINARY SQL_TYP_NBINARY	4 5	binary	length of binary value in bytes
SQL_TYP_NUMERIC SQL_TYP_NNUMERIC	16 17	numeric	first byte specifies the precision second byte specifies the scale

Declaring an SQLDA

The SQLDA is a special type of host variable structure. To ensure that the structure has the correct format, the application program should use the definition of the SQLDA provided by Adabas SQL Server. To facilitate this, an SQL statement like the following one should be embedded in the application.

```
EXEC SQL
INCLUDE SQLDA AS sqlda_ptr;
```

This statement has the effect of generating a declaration of a variable *sqlda_ptr* at the point where it is specified. This variable can then be used as a pointer to a descriptor area.

Allocating an SQLDA

When using an SQLDA to retrieve descriptive information from Adabas SQL Server either for input or output purposes, the application program normally does not know the number of variable descriptions required. The application program however has to allocate an SQLDA of a certain dimension before the PREPARE or DESCRIBE statements can be issued. In general, there are two techniques which can be used:

– The application program allocates an SQLDA of maximum size to cater for the maximum possible number of derived column list elements or host variable markers. Obviously, this might cause a significant waste of storage if the maximum has to be set very high.

– The application program allocates an SQLDA of minimum size. The dimension of the SQLDA is determined by the sqln element in the SQLDA header. If the number of derived column list elements or host variable markers exceeds this number, Adabas SQL Server will refrain from attempting to provide information on the remaining elements or markers. Adabas SQL Server, however, does return the correct number of elements in the sqld element of the SQLDA. The application program can then use this number to allocate a new SQLDA of sufficient size and re-issue the PREPARE or DESCRIBE statement. The application program must explicitly have an SQLDA declaration such that the resulting structure is in scope for all SQL statements which access it. Such a declaration does not need to be in a BEGIN DECLARE SECTION.

Determining the type of SQL statement

Although the SQLDA does not explicitly return the SQL statement type, enough information is returned in the SQLDA for the application program to determine whether the dynamic statement is a SELECT statement or not. If the field sqln is 0, the statement did not contain a derived column list and must therefore be a NON-SELECT statement.

CLIENT/SERVER TOPICS

Introduction

In the past, the traditional set-up was that an application program had to run on the same hardware platform as the database it accessed. Utilizing the client/server architecture, Adabas SQL Server is divided into a client and a server part, and makes the best use of both hardware and software resources.

It is now possible, for example, for a Windows application to access Adabas SQL Server located on an UNIX platform or on a mainframe platform, as the following figure shows:

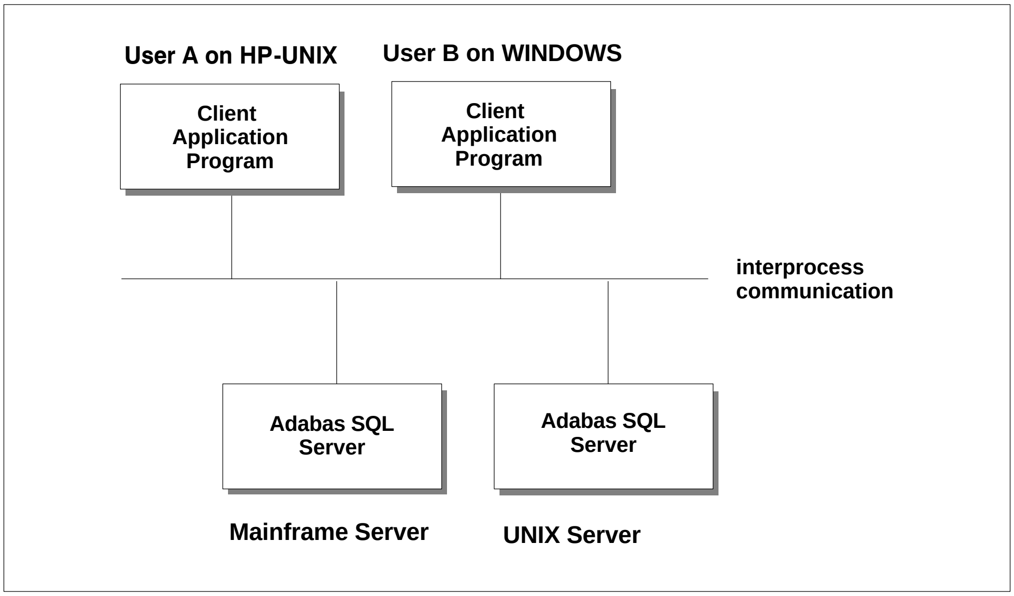


Figure 7-1: Client/Server Architecture

The following diagram shows a detailed flow of data and control in a Adabas SQL Server client/server computing environment:

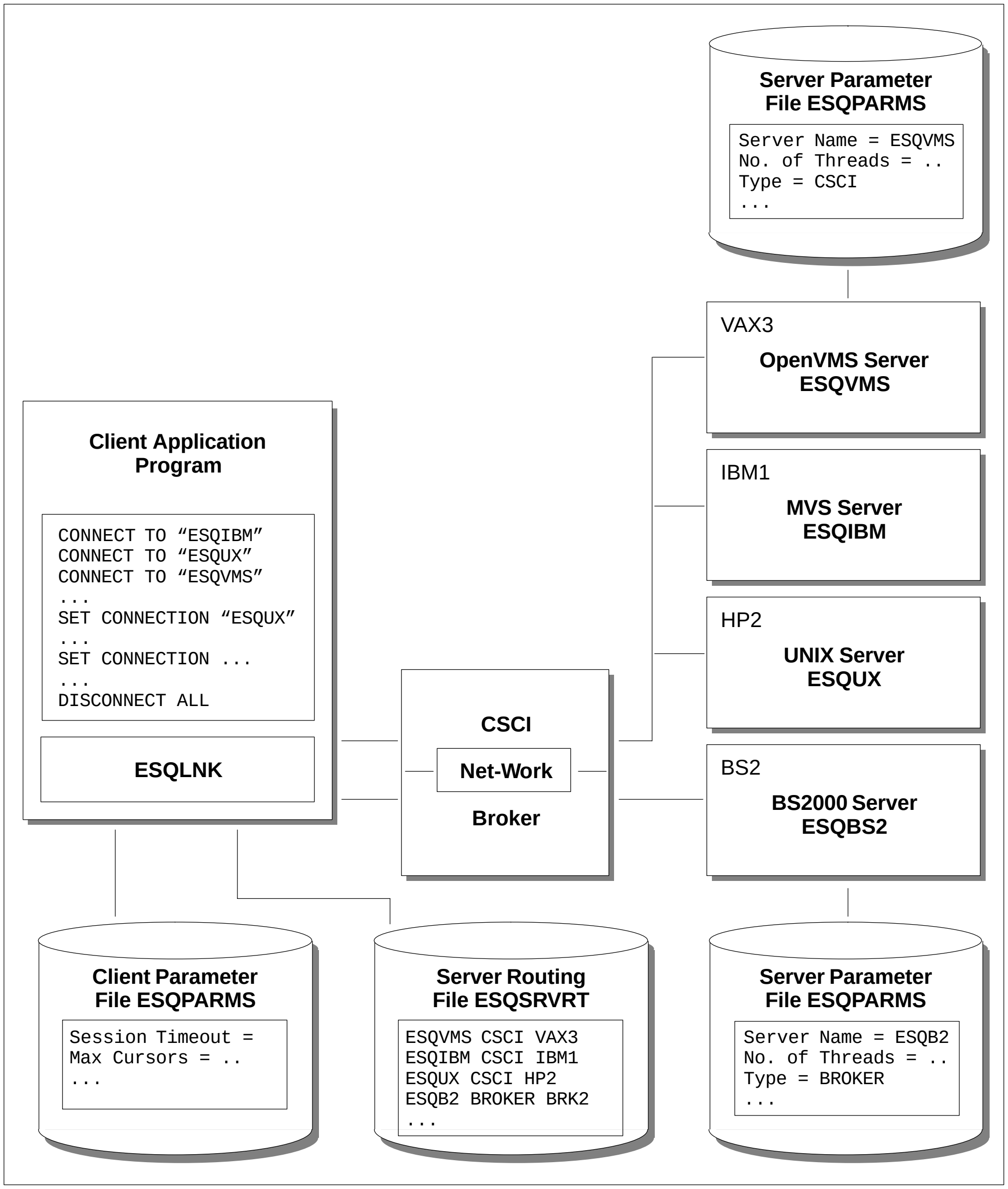


Figure 7-2: Client/Server Data Flow Diagram

After generating a server environment (on UNIX/OpenVMS with the esqgen command) a server parameter file (ESQPARMS) is generated. This file contains the default server and client parameters. This file must be modified if you want to change these default values.

During server start-up, Adabas SQL Server reads the server parameter file to establish and use the current server parameters and to establish the default client (session) parameters.

This file contains the following information: the server name, the communication protocol (CSCI or Broker) , the number of threads of the server. After start-up, the server waits for incoming client requests delivered by CSCI or Broker.

The client application program uses the statements CONNECT, DISCONNECT, and SET CONNECTION to establish, set, and terminate sessions to active Adabas SQL Servers. Each CONNECT statement that is executed on the client side reads the server routing file ESQSRVRT to get the routing information about the desired server. Also, the client parameter file ESQPARMS is read in to get the client parameter to overwrite the session-dependent parameter on server side. ESQLNK packs all SQL requests, sends it using CSCI or Broker to the server and waits for the reply. After receiving the reply, the returned data is unpacked and offered to the client application program.

Client/Server Configuration

LINKED-IN Mode

If the SQL request execution is performed in the process context of the client application, it is called LINKED-IN mode. The SQL request execution is done by ESQTHS.

- If a LINKED-IN application uses the shared memories of a server, then this is called LINKED-IN multi-user mode. The shared memories are used by an application process if Adabas SQL Server that is used is active.
- If the server is not active, then the application is running in LINKED-IN single-user mode. No shared memories are used.

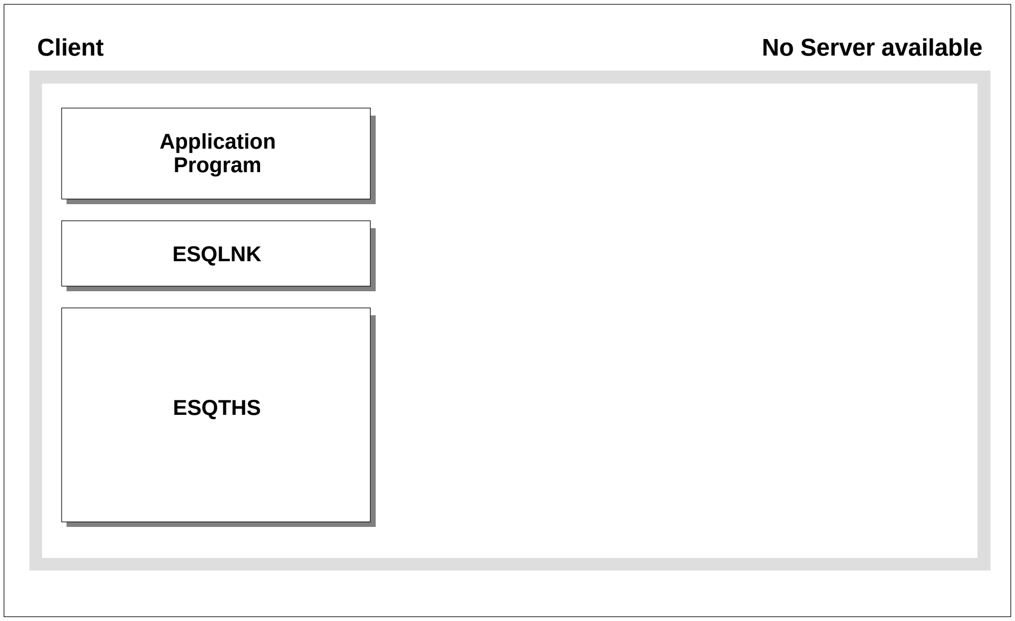


Figure 7-3: LINKED-IN Single-User Mode

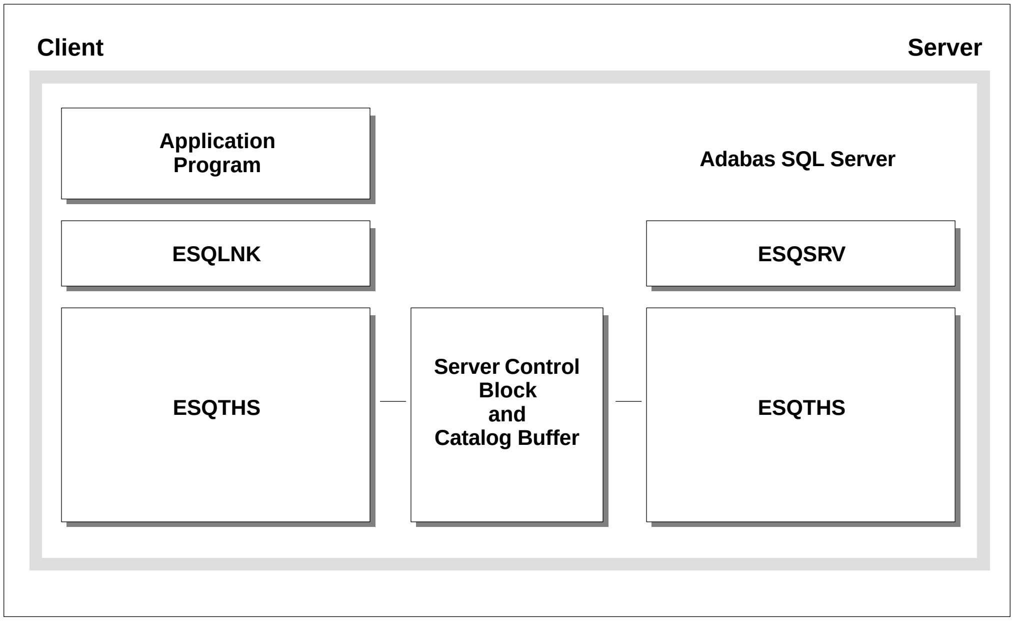


Figure 7-4: LINKED-IN Multi-User Mode

Note:
UNIX: Adabas SQL Server allows a server to start with 0 threads. In such a case, only the shared memories of a server exist and can be used by LINKED-IN applications in the multi-user mode.

Client/Server Mode

If the SQL request execution is performed by a server and not by the application program, then this is called Client/Server mode. The SQL request execution is done by ESQTHS on server side. ESQLNK does all packing, C/S communication, and unpacking of the SQL requests. No shared memories of any server are used by the application program. If the client and the server are on one and the same node, we talk about local client/server mode. If the server is located on a remote node, we talk about remote client/server mode.

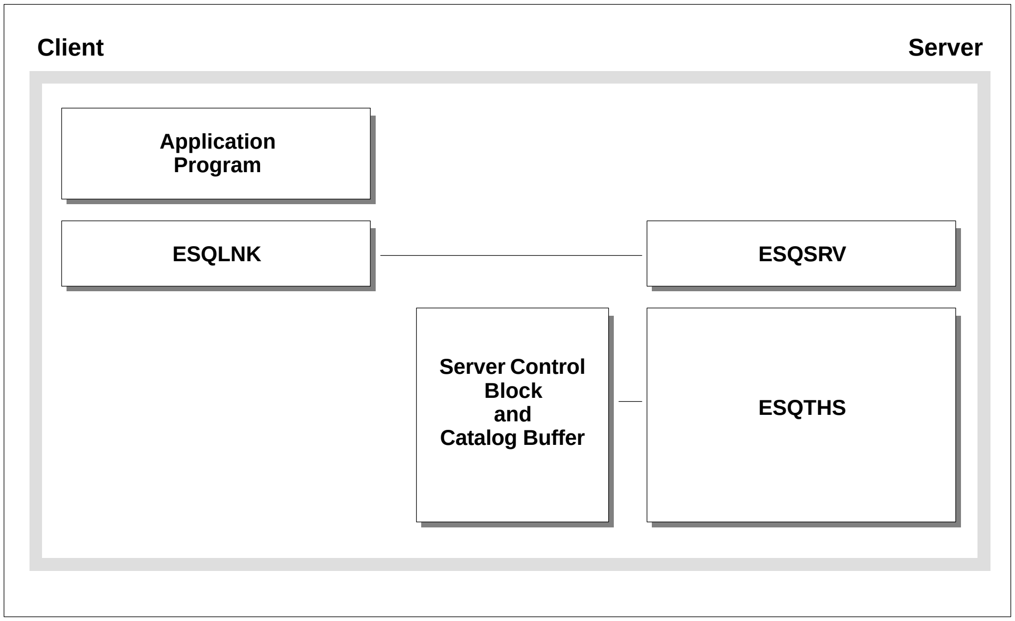


Figure 7-5: Local Client/Server Mode

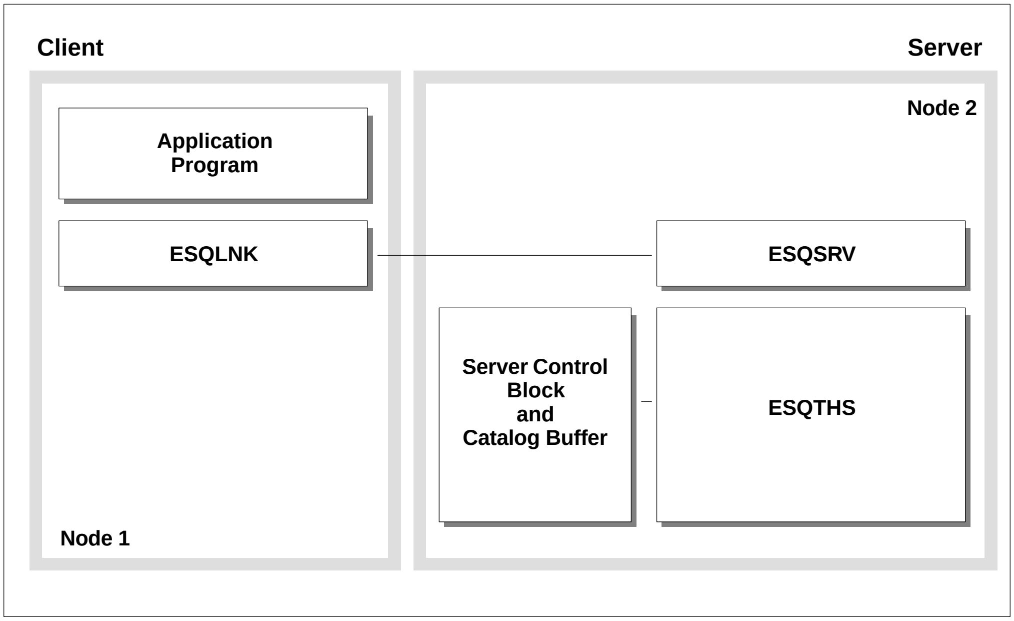


Figure 7-6: Remote Client/Server Mode

For details about the platform-dependent server architecture refer to the *Adabas SQL Server Installation and Operations Manual*, Chapter: **Operating Adabas SQL Server**.

Mode Determination

The following table shows how the various modes of Adabas SQL Server can be achieved:

Server active		**Entry in routing file (CSCI or Broker)**		**Mode**
Y	**N**	**Y**	**N**	
–	X	–	X	SINGLE-USER LINKED-IN (DEFAULT)
X	–	–	X	MULTI-USER LINKED-IN
LCL	–	X	–	LOCAL CLIENT/SERVER
RMT	–	X	–	REMOTE CLIENT/SERVER

Y(LCL) stands for ”server active on local node” and Y(RMT) stands for ”server active on remote node”. LINKED-IN single-user is the default. On UNIX and OpenVMS platforms, the ”esqshow” command gives information about the above topics. With the Mode or Client/Server Communication Logging, you are able to find out in which mode your application is currently running. For more information about logging refer to the chapter **Logging Facilities** in the *Adabas SQL Server Installation and Operations Manual*.

Features And Restrictions

The following is a list of some features and restrictions that apply to the various client modes:

LINKED-IN Mode

An application can only connect to one Adabas SQL Server at a time. No parallel multiple connections are possible in LINKED-IN mode. An application can connect to multiple servers only by connecting one server after the other (CONNECT, ... , DISCONNECT, ..., CONNECT, ...).

Client/Server Mode

Only applications in client/server mode can communicate to multiple servers at the same time. It is possible to have multiple connections to the same or different servers.

Mixed Mode

Mixed usage of the above modes is possible. Simultaneous usage of the modes is possible.

Communication Protocol

The communication protocol used by Adabas SQL Server for the interprocess communication between client and server is the Client Server Communication Interface (CSCI) or the Entire Broker (Broker).

CSCI or Broker are programming interfaces to a transport service in a client/server environment. CSCI and Broker use Net-Work as the basic transport layer.

For details about Entire Net-Work, Entire Broker and CSCI refer to the relevant Software AG documentation.

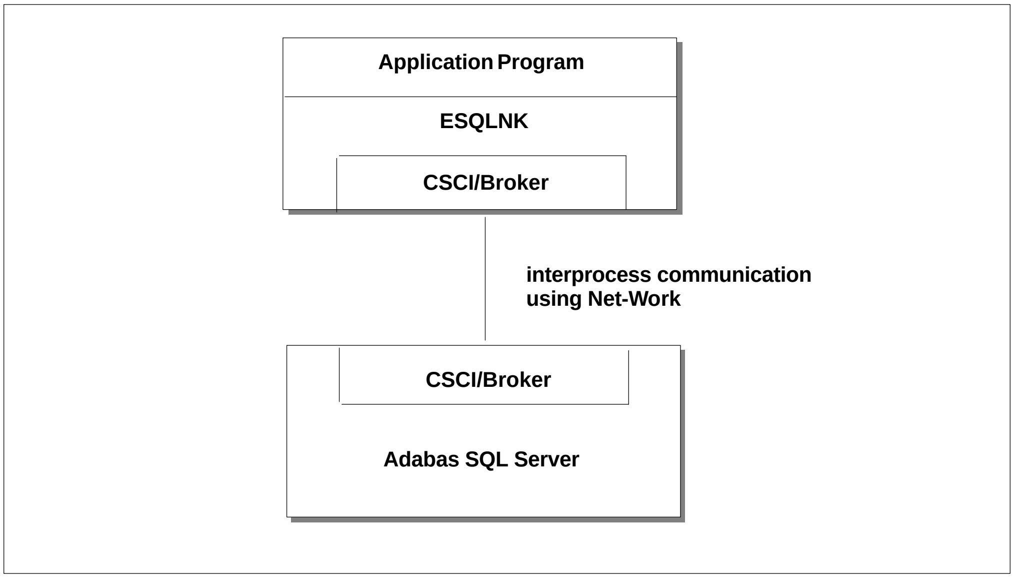


Figure 7-7 Client/Server using CSCI/Broker

Local Client/Server Mode with CSCI

The default communication protocol of Adabas SQL Server is CSCI.

The following diagrams show the different configurations of the CSCI communication protocol and how Adabas SQL Server uses CSCI:

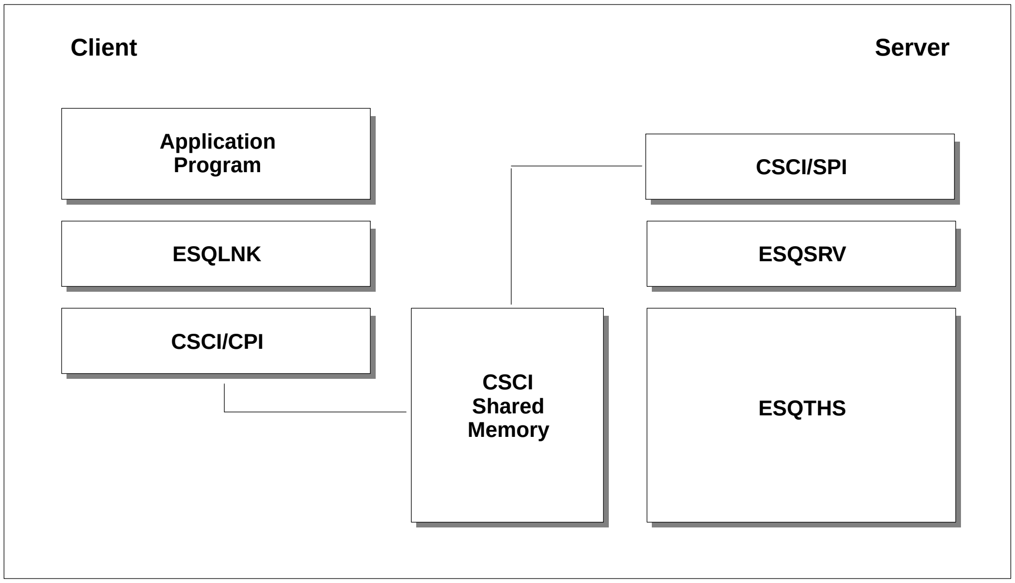


Figure 7-8: Local Client/Server with CSCI

Net-Work must not be active for this mode, only CSCI must be active on the local node.

CPI and SPI are either client or server programming interfaces. For more information about CSCI, refer to the *Adabas SQL Server Programmer’s Guide*, **Appendix D**, section **Adabas SQL Server and Entire CSCI**.

Remote Client/Server Mode with CSCI:

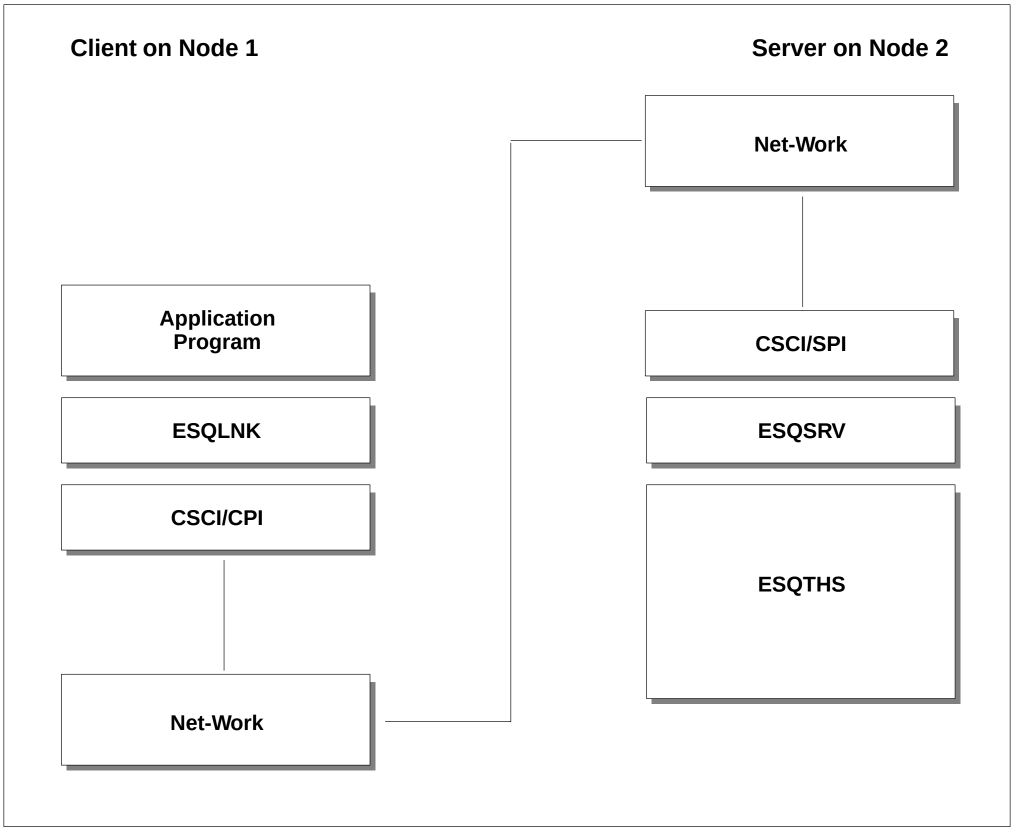


Figure 7-9: Remote Client/Server with CSCI

For this mode you need an active CSCI as well as an active Net-Work.

Client/Server Mode with Broker

As an alternative to CSCI, Adabas SQL Server can also use Broker as client/server communication protocol.

The following diagram shows one possible configuration of the Broker communication protocol and how Adabas SQL Server uses Broker, mostly used on BS2000 platforms:

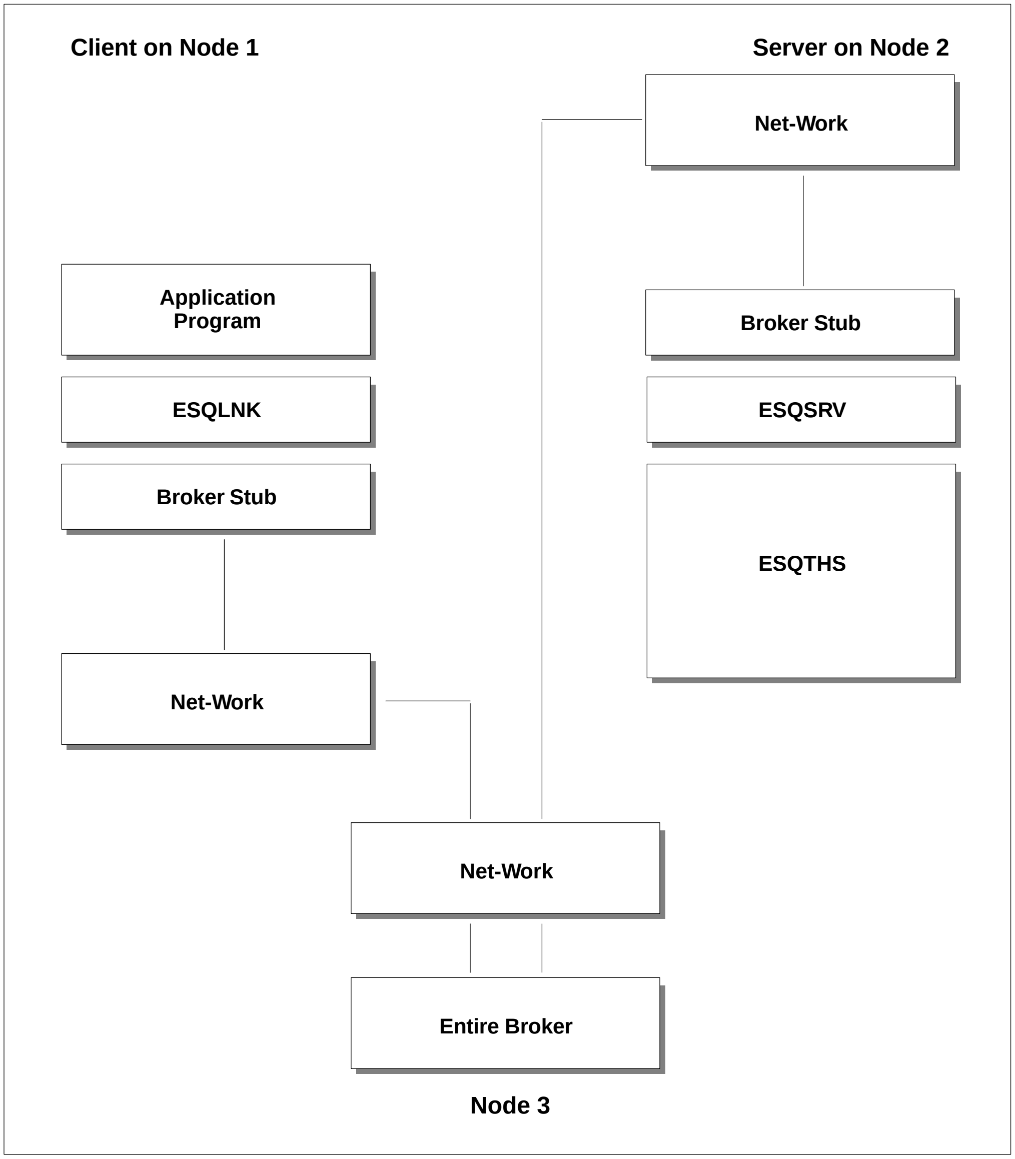


Figure 7-10: Remote Client/Server with Broker

If client, server, and Broker are located on one and the same node then Net-Work does not need to be active. If only one of the three components is not located on the local node then you need and active Net-Work for the participated nodes.

To set up an Adabas SQL Server with Entire Broker communication, the following two entries must be made in the parameter file:

```
SERVER BEGIN
.........
     TYPE = Broker
     Broker ID = <broker_id>
.........
END
```

For more information about Broker, refer to the *Adabas SQL Server Programmer's Guide*, **Appendix D**, section **Adabas SQL Server and Entire Broker.**

Time-Out Checking

ADABAS SQL Server provides two types of client/server time-outs: these apply only in real client/server mode and not for LINKED-IN mode.

Server Session Time-Out

The server session time-out is a client inactivity time out and is checked on the server side. It prevents server threads from being occupied by ”lazy” or even ”dead” clients (e.g. PC crash, clients exit-handler was not executed).

The time-out means that a particular client session is terminated by the server after a specified period of inactivity. The default value for all clients is specified in the server's parameter file. The default is 15 minutes.

The client can overwrite this value by using a client parameter file containing a SERVER SESSION TIMEOUT clause. A value of zero disables any server session time-out checks.

Note:
In the current version of Adabas SQL Server (UNIX and OpenVMS) there is no way to terminate a specific client session on the server side using an operator command.

Server Reply Time-Out

The server reply time-out is checked on the client side. It prevents client applications from hanging up if (for any reason) the server does not reply to the client's request. The modifications of the server reply time-out currently depends on the client/server protocol.

CSCI

If CSCI is used, this time-out is implemented in the CSCI/CPI layer, which defines a default value of 1 minute. This default may be overwritten on the client side by using the CSCI environment variable CSC_TIMEOUT.

Broker

If Broker is used, the time-out value is modifiable using the parameter file with the SERVER REPLY TIMEOUT clause. The default value is 5 minutes.

Access to Adabas SQL Server

The SQL application uses the CONNECT, SET CONNECTION, and DISCONNECT statements to define or change the current SQL session.

An SQL session starts with the CONNECT statement and ends with the DISCONNECT statement. The CONNECT statement establishes two things:

– communication between client and server,
– an SQL session on the server side.

SQL Statements

The statements referenced below are described in detail in the *Adabas SQL Server Reference Manual*, chapter **SQL Statements**.

CONNECT Statement

The CONNECT statement establishes an SQL session between the application and Adabas SQL Server.

Although an application can issue multiple CONNECT statements, it is set to only one Adabas SQL Server at a time. Adabas SQL Server used in the most recently executed CONNECT statement is the active one.

SET CONNECTION Statement

If at least two SQL sessions are active, the SET CONNECTION statement is used to switch from the current session to the specified one. The session context of Adabas SQL Server is restored to its exact state at the time of the suspension.

DISCONNECT Statement

The DISCONNECT statement terminates the SQL session and performs an implicit ROLLBACK.

Default Adabas SQL Server (ESQSRV)

To avoid having to specify a server name in an application, Adabas SQL Server offers the concept of a default Adabas SQL Server. This prevents unnecessary editing, precompilation and linking steps, when wanting to switch to another server.

It is not required to specify any server name in a CONNECT statement. The client environment can be set up in such a way that the default server is determined an environment variable named ESQSRV. The content of this variable is simply a server identifier or a complete server identification.

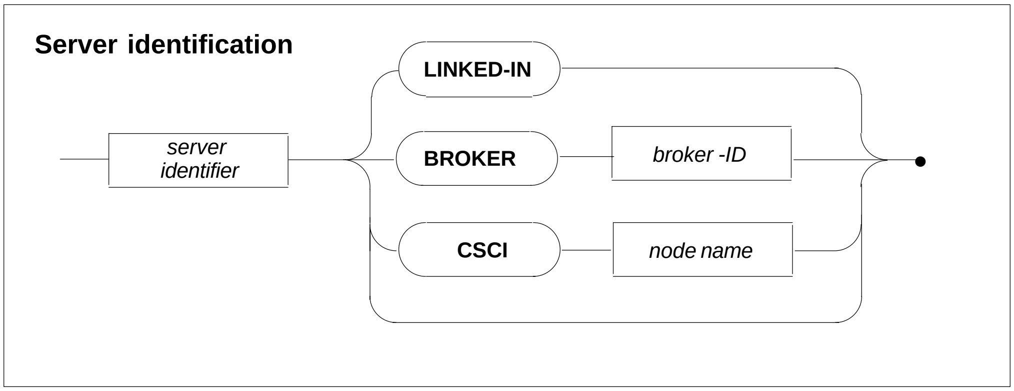
Note:
On the mainframe, the default server is specified in the parameter unit VOPRM. In UNIX and OpenVMS environments ESQSRV is set by ‘esqset’.

ESQSRV is available in any user environment and is set by the Adabas SQL Server system administrator but may be overwritten by any user. If no default server name is specified, then SAGESQSV is assumed to be the server name.

Server Identification

The server identification holds information that gives answers to the following questions:

- Which Adabas SQL Server names are known from the client side?
- Which type of client/server communication protocol has to be used to address the server (LINKED-IN, CSCI, BROKER)?
- What is the server's destination specification? For CSCI = node name, for BROKER = BROKER-ID.



LINKED-IN	no communication protocol is used. The server thread (ESQTHS) is linked together with the SQL application.
BROKER	the client is using BROKER protocol to communicate with the specified Adabas SQL Server.
CSCI	the client is using CSCI protocol to communicate with the specified Adabas SQL server.

If no server identification is specified for a server, then LINKED-IN is assuned. In such a case ESQLNK tries to load ESQTHS.

The server identification can be set using the environment variable ESQSRV or in the routing file. The following is a UNIX example to set ESQSRV:

```
> setenv ESQSRV ”MYSRV CSCI HPE”
```

Server Routing File (ESQSRVRT)

The server routing file is located on the client side (defined using the environment variable ESQSRVRT) and holds server identifications. Each line of ESQSRVRT contains a complete server identification.

If a server and its identification are not found in ESQSRV during the execution of a CONNECT statement, then ESQLNK uses ESQSRVRT to get the server identification. This technique allows the application to issue multiple CONNECT statements to different servers without knowing the communication protocol and server location.

Note:
On mainframes, ESQSRVRT is contained in the VOPRM unit.

In the first column of each line you can specify a comment line with a hash character: “#”. Example of the Server Routing File:

```
########################################################################
#
# File name:   esq_routing.dat
#
# Description: Define ESQ server Communication
#
# Usage:       This file is used by the ESQ client interface
#
# Each line in this file defines an ESQ server, it’s location and the
# type of communication to be used by the ESQ client interface.
#
# Syntax:  <server–name> <communication> <destination>
#
#  Where:  server–name   = ESQ server name (max. 8 chars case sensitive)
#          communication = ”CSCI” or ”BROKER” or ”LINKED–IN”
#          destination   = for CSCI:      <node–name>
#                          for BROKER:    <broker–id>
#                          for LINKED–IN: <>
#          node–name     = Network node name where the server resides on
#                          (max. 8 chars not case sensitive)
#          broker–id     = Id of the ENTIRE BROKER
#                          (max. 32 chars case sensitive)
#
# Example: SAGESQ01 CSCI   SAGVAX01
#          SAGESQ02 BROKER BKR34
#          SAGESQ03 LINKED–IN
#
# Version: 1.3 95/01/31 16:15:47 (co) Software AG
#
########################################################################
#
# <server–name> <communication> <destination>
#
ESQVMS CSCI VAX3
ESQIBM CSCI IBM1
ESQUX CSCI HP2
ESQB2 BROKER BRK2
```

8

UNDERSTANDING SQL QUERY TRANSLATION AND OPTIMIZATION

The following sections discuss a number of topics that refer to the process of translating SQL statements into procedural form, and generating Adabas commands for retrieval.

The mechanism that Adabas SQL Server uses to carry out the execution of an SQL statement is the meta program. The meta program can be thought of as a series of instructions for a virtual machine. This virtual machine is implemented inside the Adabas SQL Server runtime system. Most of the Adabas commands performed during the execution of a translated SQL statement result from a particular instruction of that virtual machine.

Information Available on Translated Queries

There are three ways of visualizing the behavior of a translated SQL statement:

– First, the generated code can be displayed in pseudo code form by turning on the EXPLAIN logging. This provides a view of the procedural code resulting from the compilation: in particular the structure of the loops and the Adabas commands will be shown. This logging is generated by the time a meta program has been compiled, that is, translations of static embedded SQL statements can only be logged during their first execution.

– Second, the generated meta program can be dumped after compilation. In comparison to the pseudo code logging, this shows the exact code that will be executed by the virtual machine at run time. Thus it is more detailed and usually less readable, because it does not present a structured view of the generated program. However, the Adabas access information is displayed more accurately here. The meta program dump is also written at compile time, that is, for static embedded statements during their first execution.

– Third, the Adabas command logging can be used to monitor the execution of an SQL statement. This will show the dynamic behavior of the program as each Adabas command is executed.

More details on how to use any of these methods can be found in the chapter **Logging Facilities** of this manual.

The Generated Adabas Direct Calls and Their Consequences

Adabas Access Paths Used for Retrieval

Adabas SQL Server uses one of four methods to access data in an Adabas file:

– An S1 to get the first record, followed by an L1 with the GET NEXT option to get additional records. The S1 makes use of the HOLD ISN LIST option should the ISN list need to be processed more than once (from within a loop) or if a complex search expression requires a combination of that ISN list with further search conditions.

– L3 commands. These may be used in order to retrieve a range of descriptor values; the VALUE START option will be set in this case. Another occasion where L3 will be used is when there is an ORDER BY clause present in the SQL statement that requires sorting in descriptor order.

– L1 commands with the READ ISN SEQUENCE option. This method is used when the retrieval is based on a range of ISNs. Additionally, it is necessary to use it in case the complete file must be scanned and a subsequent UPDATE may have to be carried out at a later stage (L2 cannot be used because the UPDATE may affect the physical order of records).

– L2 commands to read a full file, with no subsequent UPDATE.

Adabas Hold Queue Usage

In any of the following situations, the above commands will be dynamically replaced by their respective equivalents that enter records into the hold queue (S4, L4, L5, L6):

- The LOCK WHEN READING option has been used in the Adabas SQL Server parameter file. This will have the effect of setting each and every record in hold condition upon reading it.
- The retrieval occurs on behalf of an UPDATE or DELETE command on the table that will be affected by the UPDATE or DELETE operation.
- The commands are issued for the target table of an updatable cursor. In this case, positioned UPDATE or DELETE commands have to be expected. The ‘updatable’ property for a cursor may be specified explicitly for each individual cursor by means of the FOR UPDATE clause or globally by using the UPDATE EXPECTED parameter in the Adabas SQL Server parameter file. In static embedded SQL, cursors are automatically determined to be updatable when there is a positioned UPDATE or DELETE present in the same compilation unit as the cursor declaration.

Usage of the Adabas MULTIFETCH Option

In general, all read commands are issued using the MULTIFETCH option. There are three exceptions to this rule:

- Records read from the target table of an updatable cursor will not be multifetched. This is because COMMIT or ROLLBACK, when used with KEEPING ALL, might remove locks established for multifetched records before these records have been FETCHed by the SQL application and thus before an UPDATE or DELETE occurs which requires the lock to be held.
- When all of the M (MULTIFETCH), N (GET NEXT), and R (RETURN) options are to be used for an L1 command, and Adabas SQL Server is communicating with Adabas UNIX v1.2 not supporting the O option, the M (MULTIFETCH) option will be dropped. The R (RETURN) option may have been set in accordance of the Adabas SQL Server parameter HOLD = OFF.
- When MULTIFETCH has been explicitly turned off by the Adabas SQL Server parameter line COMPILER (EXTENDED FEATURES = (4)) in the parameter file.

The number of records and the memory size used for MULTIFETCH may be influenced by run time parameters.

Adabas Search Buffer Entries Generated for S1 Commands

Adabas SQL Server can generate the following search buffer entries:

– Comparison operators EQ, GE, LE, GT, LT
– Range operator S
– Boolean connectors D and R
– Single field descriptors
– Superdescriptors (see below for conditions of use)
– Nondescriptors, where supported by the Adabas nucleus
– Command IDs for combining ISN lists with extra search conditions

Adabas SQL Server does not use the NE operator. Also, the O connector is not used.

For a LIKE predicate, a range will be constructed containing all values that start with the initial non-wildcard portion of the pattern to be matched.

Preconditions for Superdescriptor Usage

The following preconditions must be met for Adabas SQL Server to choose the search for a superdescriptor:

– The WHERE clause must contain comparisons for a leading portion of the superdescriptor, i.e. a predicate for at least the first element must be present.
– The superdescriptor definition must include its elements with their respective standard lengths as per their definition.
– The comparison predicates must occur in an AND conjunction.

Superdescriptors can be used with both L3 and S1 accesses. If the superdescriptor is chosen for S1 access, the individual comparisons will not show up in the search buffer.

When a superdescriptor contains component fields defined with the Adabas NU or NC (without NN) options, any records containing NULL and Adabas default values in these component fields are not represented by the superdescriptor. This means that these values can not be retrieved by a descriptor search, as shown in the following example:

```
create table t (a char, b char);
create index ii on t (a, b);
commit;
insert into t values (’A’, null);
select * from t where a = ’A’;
```

The superdescriptor that implements index ii can not be used for the specified query, because due to the NULL value of column B, no descriptor value will be stored. By specifying an additional predicate which will exclude NULL values from the results, the descriptor search can be performed as follows:

```
select * from t where a = ’A’ and b is not null;
```

Complex SQL Structures and Their Translation

Procedural Code for Join Processing

When processing joins, the Adabas SQL Server query compiler will generate a system of nested loops, where each loop is associated with a particular table that occurs in the join. Thus the principal structure of a two-table join is as follows:

```
for ( each row of first table satisfying some predicate evaluation )
{
  for ( each row of second table satisfying some predicate evaluation )
  {
    add a record to the result set
  }
}
```

The actual number of loops is identical to the number of tables in the from clause.

The evaluation of the predicates applying to each table may come out in four different ways:

– A predicate may be evaluated beforehand (that is, by an Adabas search command) and the loop must then run only across the set of rows that are known to fulfill the condition.
– A predicate may be evaluated only if an tabled is accessed and filtered for the result once each record has been read. It might be necessary to open inner loops before a predicate can be evaluated.
– There may be combinations of the above, that is, an Adabas search command gives us a superset that we may further reduce with extra filters.
– There may be no predicates at all, in which case a full table scan has to be performed.

In order for the system of loops to execute with acceptable performance, the predicates of the WHERE clause must be mapped on to table restrictions such that they can be:

– Most selective, that is, they should result in a small set because this will save on the number of passes for any inner loop.
– Cheap to apply, that is, Adabas descriptor search must be preferred over applying filters after reading.

Example:

```
select person.person_id, person.surname,
       yacht.yacht_id, yacht.yacht_name
from   person, yacht
where  person.person_id = yacht.id_owner
and    yacht_name = ’CUCA RACHA’;
```

There are two tables in this query, so there will be two loops in the translation. There are two possible choices of arranging these loops: which of the tables should be associated with the outer loop needs to be chosen; the inner loop will then apply to the other table.

Provided that person_id is a descriptor and id_owner and yacht_name are not, the following code will be generated:

```
for ( each row of yacht read by L2 )
{
  if (yacht_name = ’CUCA RACHA’)
  {
    create ISN list by S1 for person, person_id = yacht.id_owner
    for ( each row of the above ISN list )
    {
  add a record to the result set
    }
  }
}
```

Note that the inner loop will, in fact, degrade to a single record access if person_id is a unique descriptor.

Principles of Adabas SQL Server Join Optimization

With the fixed structure of the code shown above, the primary degree of freedom that will be used to achieve the above goals is the ordering of the loops within each other. Whereas logically any arbitrary order of the loops will do, there are, of course, large differences in terms of performance.

The goal of Adabas SQL Server join optimization is to find an optimal arrangement of these loops. The input criteria for this decision are:

– The information on which descriptors and superdescriptors may be applied.
– The structure of the query, that is, the information about which restrictions and join conditions are present in the WHERE clause and their interdependencies.

The Adabas SQL Server join optimizer does not currently consider any information about Adabas file size.

The technique that Adabas SQL Server uses for join optimization is called “query decomposition”. This is a graph-based algorithm that works on a query graph. The query graph is constructed from a query by assigning a node to each table and an edge to each predicate conjunction element. The join conditions are mapped onto edges connecting several nodes. Simple restrictions are mapped into loops that touch on a single node. The edges in the query graph are then labeled with the following information:

– Predicate classification, for example, an equality comparison predicate is considered simpler than one using different operators.
– Descriptor information, that is, which predicates are covered by an Adabas descriptor. For join predicates this information is given for each participating node.

During the decomposition process, the nodes may be labeled with

– Size information, i.e. some nodes are considered as representing smaller tables than others, because they have previously been reduced by applying some restriction.

The query decomposition algorithm will successively remove edges and nodes from the query graph in order to decompose it. The order of the meta program table loops will be derived from the order in which the algorithm removes nodes from the graph. The algorithm repeatedly applies the following basic operations to the graph (ordered by decreasing precedence):

– Replace a node that has a loop labeled ‘simple’ by a node that is labeled ‘small’.
– Remove a node that is labeled ‘small’.
– Remove a node that is connected by an edge which indicates a descriptor only for the other nodes.
– Remove a node that will disconnect parts of the graph. Nodes having simple edges will be preferred.
– Remove a node. Nodes having simple edges will be preferred.

Whenever two nodes have the same properties and cannot be distinguished by the above criteria, preference will be given according to the order specified in the FROM clause. Note that if all fields have descriptors, then, for purposes of join optimization, this is just as if there were no descriptors at all, because the descriptor information then does not contribute anything to the decision process.

Join Elimination for Nested Data-Structure Access

When a subtable is accessed, it is often necessary to combine the subtable data with data from the master table according to the clustering.

These joins can be eliminated by the query compiler, that is, it is not necessary to consider the subtable and the master table as two independent entities. Rather these queries can be implemented by going along the records of the master table and associating the subtable records as they are clustered with each master table record.

In order for joins to be eliminated this way, the join condition must make it clear that a given subtable record must be combined with nothing but the master table record it is clustered with. In terms of referential integrity, the join condition thus has to be formulated as an equijoin (join condition with EQUALS operator) on those columns making up the referential constraint between the subtable and the master table.

Subquery Processing

Adabas SQL Server will generate the code for a subquery to be similar to the code for the primary queries. The subquery code will be positioned inside the system of loops generated for its superior query.

The Adabas SQL Server compiler will attempt to move the code generated for a subquery to the outermost position so that the subquery can be evaluated. This aims at evaluating the subquery as early as possible in order to save processing any inner loops in the case where the subquery predicate evaluates as false.

For positioning the subquery code into the code for the surrounding query, you must consider whether a subquery is

– Correlated or non-correlated, that is, whether or not it depends on its environment by referencing it;
– Single-row or multiple-row, that is, whether the result is a single row (by definition) or can be a set of rows (as in a quantified subquery) that must be processed in order to evaluate the subquery predicate.

The simplest case of course is a non-correlated, single-row subquery. It can be evaluated independently of its environment and it can be replaced by a single value for purposes of evaluating the surrounding query. Its code will be generated outside any other loops.

Example:

```
select person_id, surname
from   person
where  person_id = (select id_owner
                   from   yacht
                   where  yacht_name = ’CUCA RACHA’
                   );
```

The subquery here is a non-correlated, single-row subquery, so its associated code will not be positioned inside of the outer queries code, but it will be evaluated beforehand. The following code will be generated:

```
for ( each row of yacht read by L2 )
{
  if (yacht_name = ’CUCA RACHA’)
  {
    make sure subquery produces a single result row
    save id_owner value
  }
}
if (subquery result does exist)
{
  create ISN list by S1 for person, person_id = subquery result
  for ( each row of the above ISN list )
  {
    add a record to the result set
  }
}
```

Reasons for External Sorting

As already mentioned, Adabas SQL Server can exploit the ordering provided by an Adabas descriptor by using the Adabas L3 command to implement the ORDER BY clause.

In cases where this is not possible, external sorting – driven by the Tuple Manager – will occur. This is a component of Adabas SQL Server that has been designed for storing, sorting, and retrieval of intermediate result sets.

The Tuple Manager implementation uses both main memory and disk space for storing its files. Disk space will not be used until some size threshold has been exceeded. This behavior can be influenced by setting environment variables (see chapter **Operating Adabas SQL Server**, section **Sort Buffer Size Setting** in the *Adabas SQL Server Installation and Operation Manual*).

The SQL language constructs that require sorting are:

– SELECT DISTINCT, it may be necessary to sort a result in order to perform duplicate elimination.
– DISTINCT functions, for example, COUNT (DISTINCT column), sorting required for duplicate elimination.
– UNION (without an ALL specification) requires sorting for duplicate elimination.
– GROUP BY, sorting required to identify groups and perform the aggregation.
– ORDER BY, sorting required to deliver a sorted result.

One SQL statement may require several of these constructs and thus it may become necessary to sort several times for different purposes.

The Adabas SQL Server compiler contains an optimization that will attempt to combine the sorting generated for ORDER BY and that for GROUP BY. The precondition for this is that all elements of the ORDER BY clause show up in the GROUP BY clause.

Adabas SQL Server does not generally detect whether any duplicate elimination is applied to data that cannot contain duplicates by definition. In such cases, it will be necessary to modify the SQL statements to remove the unnecessary sorting.

Setting the BLOCK SIZE for Nested Data Structures

For a nested subtable, the DDL definition allows for the specification of a BLOCK SIZE.

The number of occurrences that can be nested into a row of the referenced table is not limited by the specified size. Rather, it drives the allocation of a memory buffer that is used to hold the retrieved data. By setting this to an appropriate value, it is possible to specify the number of occurrences of a multiple field or periodic group that are read by a single Adabas command.

When scanning a nested subtable, the Adabas read command may have to deal with up to three different levels: the Adabas records, the first-level nested fields and the second-level nested fields.

Two different mechanisms are used to control the buffering of data during these scans:

- Adabas records are accessed using the Adabas MULTIFETCH option, that is, several records are read at a time. The characteristics of this operation can be influenced using parameter processing directives.
- Extra buffering occurs for first-level and second-level nested data. A BLOCK SIZE is associated with each level of nesting.

If the BLOCK SIZE chosen is too small for a particular record, then additional read commands will be issued to refill this buffer. This may happen several times until all of the information of this record has been read.

If, on the other hand, the BLOCK SIZE is too large, space will be wasted in the meta program, especially in the format and record buffers of the Adabas command. This will also lead to a slightly increased processing time.

Thus the BLOCK SIZE should be set appropriately for each application near the expected average of nested records.

9

MULTIFETCH FEATURE

The MULTIFETCH feature is available in order to minimize the data transfer and the inter-process communication between the application, Adabas SQL Server and Adabas. MULTIFETCH means that more than one record is transported per READ or FETCH request. Adabas SQL Server offers two different levels of MULTIFETCH:

– SQL MULTIFETCH
  The SQL MULTIFETCH applies between the application and Adabas SQL Server.
– DBS MULTIFETCH
  The DBS MULTIFETCH applies between Adabas SQL Server and Adabas.

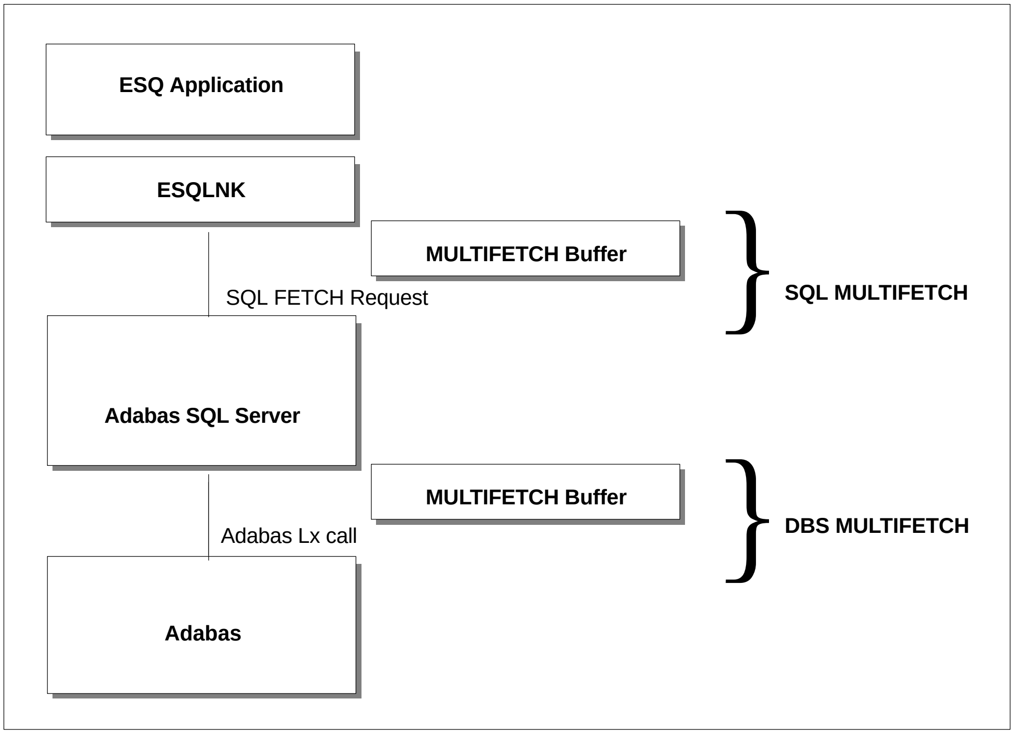


Figure 9-1: Overview of MULTIFETCH Activities

SQL MULTIFETCH

The MULTIFETCH feature offered by Adabas SQL Server is used by ESQLNK on the application client side. The MULTIFETCH feature is used for all SQL FETCH requests. For the first FETCH request performed by the application, ESQLNK fetches a variable number (block factor) of records and uses a local MULTIFETCH buffer to store this data. Each FETCH request performed by the application picks one record out of this buffer until the buffer is empty. When the buffer is empty, a variable number of records are fetched from the server and stored in this buffer.

The SQL MULTIFETCH is transparent, that is, the application does not realize whether a record was fetched out of the local buffer or from the server.

ESQLNK uses this feature per default. It is possible to switch off the SQL MULTIFETCH. The number of records (block factor) transferred from Adabas SQL Server into the local buffer can be set using the parameter file. The default is 16. For details, refer to the *Adabas SQL Server Installation and Operations Manual*, **Appendix A – The Parameter Processing Language**, section: **GLOBAL Settings**.

DBS MULTIFETCH

Adabas offers a MULTIFETCH feature which is being used by Adabas SQL Server. This feature of Adabas is called DBS MULTIFETCH here. The MULTIFETCH feature is used for any of the Adabas commands L1/L4, L2/L5, L3/L6 and L9. For the first Lx call performed by Adabas SQL Server, it reads a variable number (block factor) of records and uses the server MULTIFETCH buffer to store this data. Each Lx call picks up one record out of this buffer until the buffer is empty. When the buffer is empty again, a variable number of records are read from Adabas and stored into this buffer.

Adabas SQL Server uses this feature per default. It is possible to switch off the DBS MULTIFETCH. The number of records (block factor) transferred from Adabas into the buffer of Adabas SQL Server by the Lx can be set using the parameter file. The default is 16. For details refer to the *Adabas SQL Server Installation and Operation Manual*, **Appendix B – The Parameter Processing Language**, section: **GLOBAL Adabas Settings**.

For details about DBS MULTIFETCH, see also the *Adabas Command Reference Manual*, **Using the MULTIFETCH Feature**.

Restrictions

Under a few conditions, the MULTIFETCH is switched off internally by ESQLNK or Adabas SQL Server:

- The SQL MULTIFETCH is switched off if memory is not reliable between SQL requests. This can be valid if ESQLNK is used under CICS with pseudo-conversational mode.
- The SQL MULTIFETCH is switched off if you use a mixed version environment, that is, ESQLNK v13x with Adabas SQL Server v14x.
- Both MULTIFETCHs are switched off if you use an updatable cursor.

In addition to this, it is not possible to use different host variable types between FETCH requests of one and the same cursor. If you try to do this, you will get the response code –8647: “FETCH host variable differs from that of previous FETCH statement”. In such a situation, you have to change your FETCH host variables (make them homogeneous) or you have to switch off the SQL MULTIFETCH.

Please note that the maximum amount of data which can be transferred between application and Adabas SQL Server or between Adabas SQL Server and Adabas is limited to 64KB (on some platforms only 32KB). Example: If your record length (sum of all host variable lengths) is 500 Bytes, then the maximum MULTIFETCH block factor is 128.

How to Use MULTIFETCH

The block factor specifies the number of records to be transferred for one FETCH request or for one Lx call. The default for both block factors is 16. These parameters can be changed using the parameter file.

If you want to change these defaults, please keep in mind that the DBS MULTIFETCH block factor should be a integer factor of the SQL MULTIFETCH block factor. Example:

DBS MULTIFETCH block factor	32
SQL MULTIFETCH block factor	16

In general, this rule will give you the maximum performance.

If you want to monitor the SQL MULTIFETCH, please use the Brief SQL Command Logging. If you want to monitor the performance, please use the Elapsed Time Logging.

EMBEDDING SQL STATEMENTS IN HOST LANGUAGES – C

General Rules

SQL Statement Delimiters

SQL statements are delimited by the prefix EXEC SQL and the terminator ';'. The prefix may be written in upper or lower case letters. In Adabas SQL Server mode, upper or lower case is permitted and the prefix may be split over numerous lines, separated by any white space character. In ANSI and DB2 modes, upper case is required and only white spaces may separate the prefix keywords.

SQL Statement Placement

SQL statements may be specified wherever a C statement may be specified within a C function block, as the Adabas SQL Server compiler replaces them with generated C statements. Included C source code must not contain any SQL statements nor any host variable declaration for the use in SQL statements. Similar restriction apply to C macro bodies.

The INCLUDE SQLCA statement may be positioned anywhere a C variable declaration could be positioned. As this statement results in a declaration of an SQLCA structure, it must be positioned to be in scope for any statement using this SQLCA declaration. The C scoping rules apply.

The SQL WHENEVER statement may be coded anywhere in the C program.

Comments

SQL statements may contain C comments wherever a blank is permitted. Comments are not allowed in strings and may not be nested. The C comment delimiters ‘/*’ and ‘*/’ are replaced by ‘**’.

C Example:

```
EXEC SQL WHENEVER SQLERROR
/*                         CONTINUE */
                           GOTO HANDLE–ERROR;
```

SQL Example:

```
EXEC SQL WHENEVER SQLERROR
                      – – CONTINUE
                           GOTO HANDLE–ERROR;
```

Host Variables

C host variables used in SQL statements must be declared within the SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements. Adabas SQL Server allows the use of single host variables, arrays of host variables and host variable structures.

Host Variable Declaration

C arrays and structures are named sets of C single host variables and must conform to the ANSI Standard (X3.159-1989) for C. The use of C structures and arrays within SQL statements is an Adabas SQL Server extension and not part of the ANSI SQL Standard. In this version of Adabas SQL Server it is not possible to declare host variables as pointers or to use a union of host variables. The use of the ’enum’ type is also not possible.

Binary Data Type

There is no intrinsic binary data type in the C host language. In order to retrieve and supply binary data using host variables, the intrinsic C data type of character must be mapped to the Adabas/Adabas SQL Server data type of binary. A pseudo type has been introduced using an Adabas SQL Server ’macro’. Therefore, host variables which are to be used for binary data type transportation, must be declared as ESQ_BINARY. Adabas SQL Server will then associate such variables with the type binary. Such variables can, therefore, only be used as binary host variables. The type declaration, is indeed a C macro, provided by Adabas SQL Server and defined in esqca.h. An ESQ_BINARY variable is declared as follows :

```
EXEC SQL BEGIN DECLARE SECTION ;
ESQ_BINARY ( hv_name, x ) ;
EXEC SQL END DECLARE SECTION ;
```

where	hv_name is the name of the host variable.
	x is the length in bits of the variable.

The macro is resolved as follows :

```
char hv_name [ y ] ;
```

where	y is the required length in bytes.

Binary host variables are not subject to byte swapping, nor are they subject to any ASCII/ EBCDIC conversion. No string terminator is to be provided for binary host variables. Therefore, the direct binary contents of a host variable will be received by Adabas SQL Server, with each element of the character array representing a full 8 bytes.

Numeric Strings

There is an intrinsic limit to the size of a C host variable integer value. This is because such variables are limited to 31 bits. Adabas SQL Server however, supports the SQL data types DECIMAL and NUMERIC, which are limited to 27 digits. This represents significantly larger numbers than is possible within a C long variable.

The host languages Cobol and PL/1 do support such variables using the variable types numeric and decimal. Adabas SQL Server, within the C language supports the numeric data type by mapping it to a character string representation.

Such variables must however be declared in a special way:

```
ESQ_NUMERIC_STRING ( hv_name, pr, sc, signed, point, terminator ) ;
```

where: hv_name is the intended name of the host variable.
pr is the precision. Maximum 27. Greater than 0.
sc is the scale. Minimum 0 and <= pr.
signed is either ESQ_SIGN or ESQ_NO_SIGN.
point is either ESQ_POINT or ESQ_NO_POINT.
terminator is either ESQ_TERM or ESQ_NO_TERM

It can be seen that the precision and scale of the variable must be fully defined at pre-compile time. Furthermore, the provision of a sign character can be defined, as well as the insertion of a decimal point character and provision of an extra character to accommodate a C string terminator (’\0’).

For example the following declaration would result in a host variable called tax_number of type character array with 21 elements:

```
EXEC SQL BEGIN DECLARE SECTION ;

ESQ_NUMERIC_STRING ( tax_number, 20, 0, ESQ_NO_SIGN,
                     ESQ_NO_POINT, ESQ_TERM ) ;

EXEC SQL END DECLARE SECTION ;
```

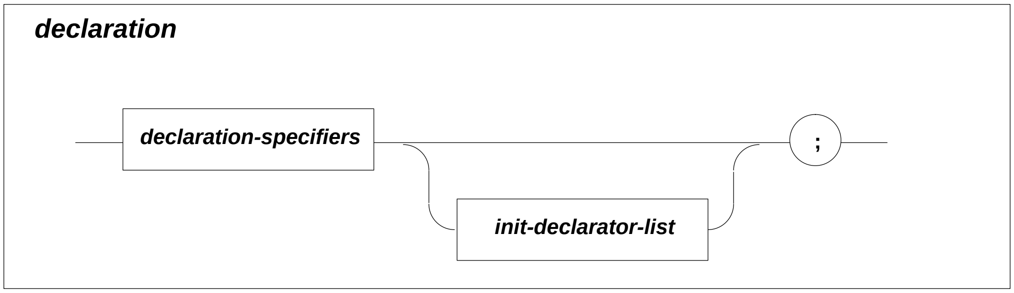
The inclusion of the switches ESQ_SIGN and ESQ_POINT would increase the array size to 23 elements.

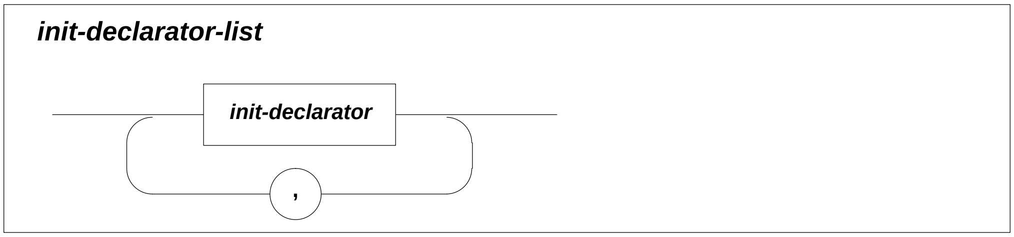
The host program interacts with such variables as it would with any other character array. It must, however, ensure that the contents conform to the original declaration, e.g. the position of any decimal point, the provision of any sign character and that, otherwise, only numeric characters are present.

The use of such host variables within SQL statements, is subject to the same restrictions as for any other numeric host variable.

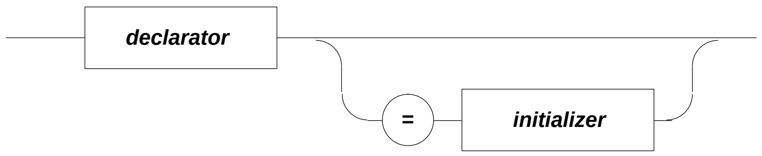
The pre-compiler must run prior to the C macro processor. No corresponding functionality is supported for the C language when using dynamic SQL and the SQLDA.

Syntax

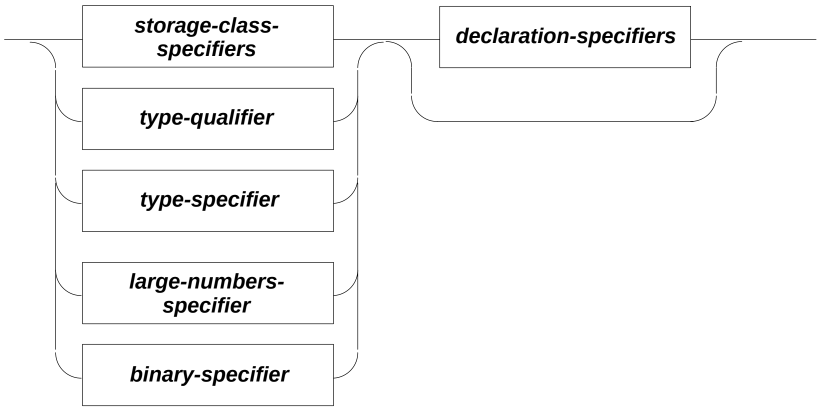




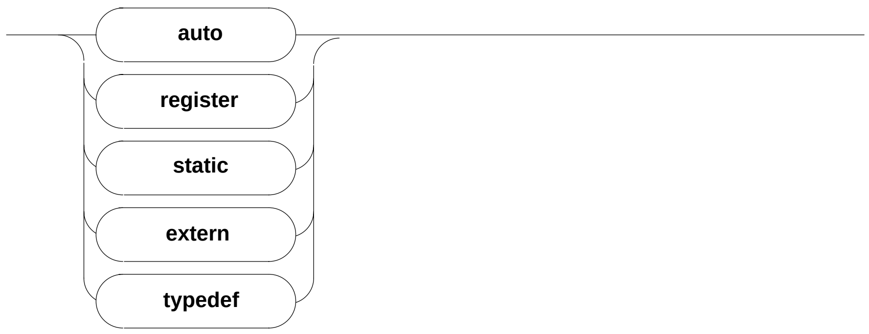
init-declarator



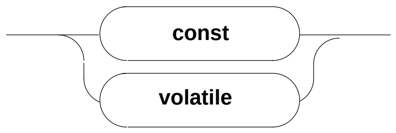
declaration-specifiers



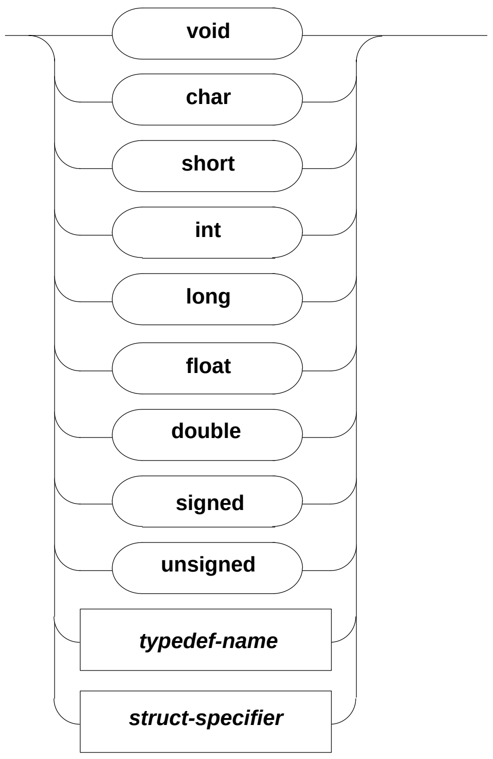
storage-class-specifier



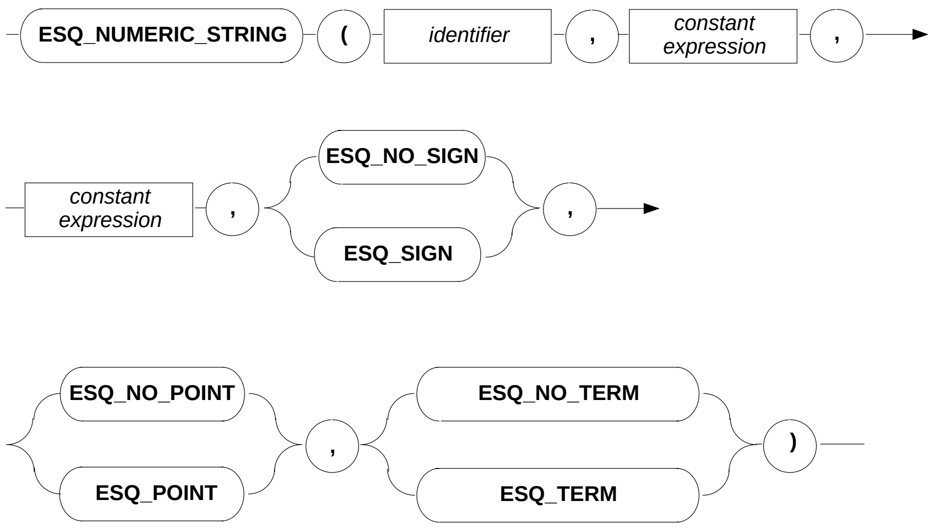
type-qualifier



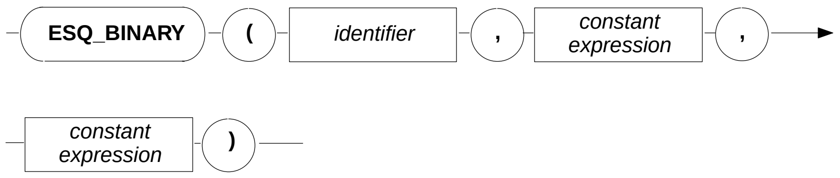
type-specifier



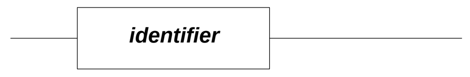
large-numbers-specifier



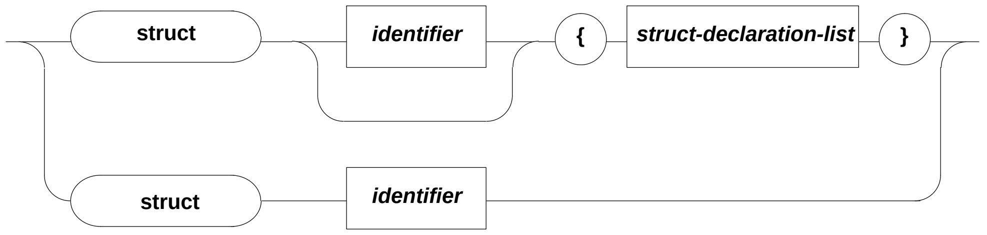
binary-specifier



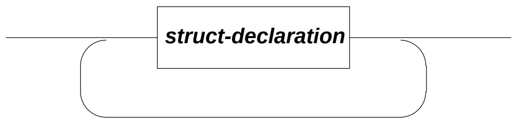
typedef-name



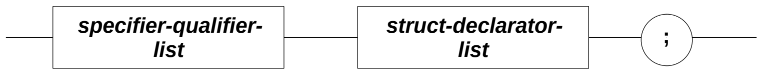
struct-specifier



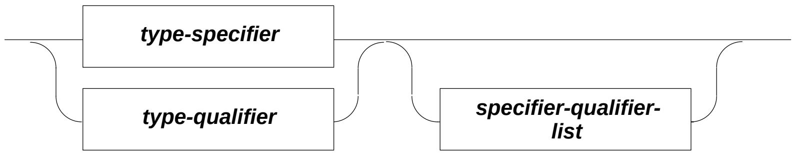
struct-declaration-list



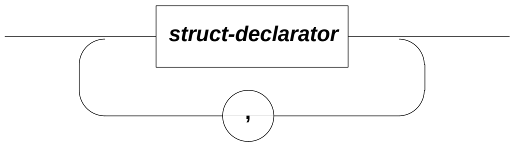
struct-declaration



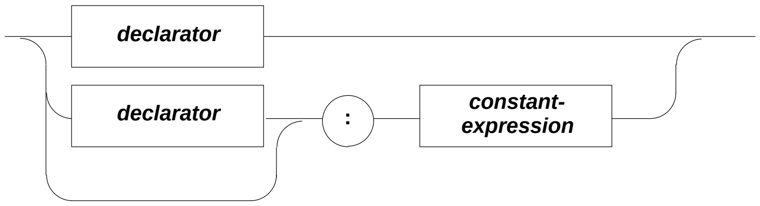
specifier-qualifier-list

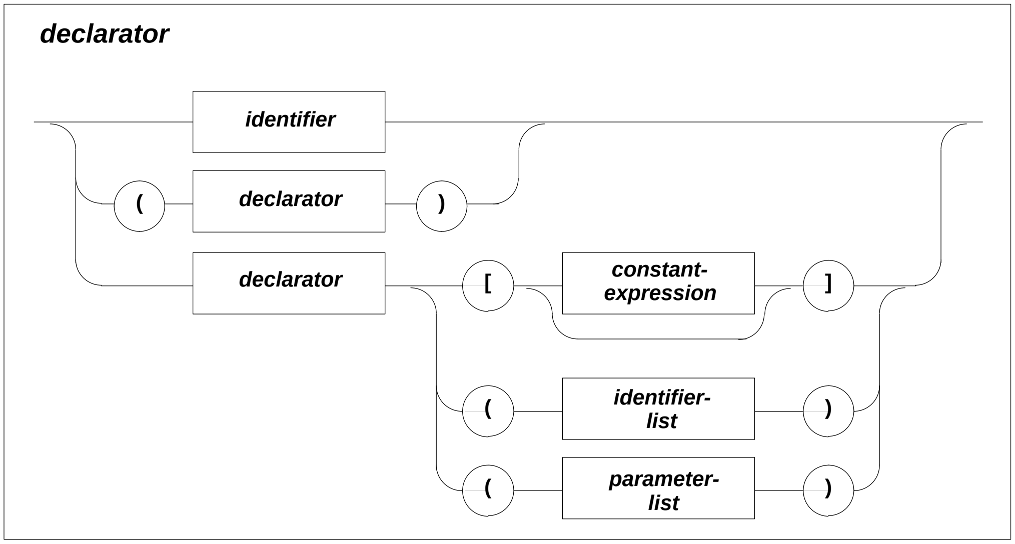


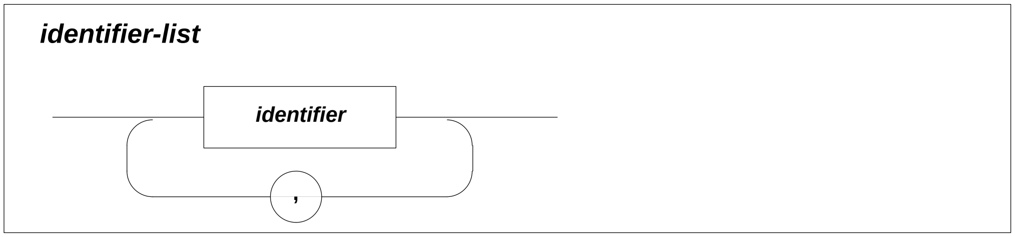
struct-declarator-list

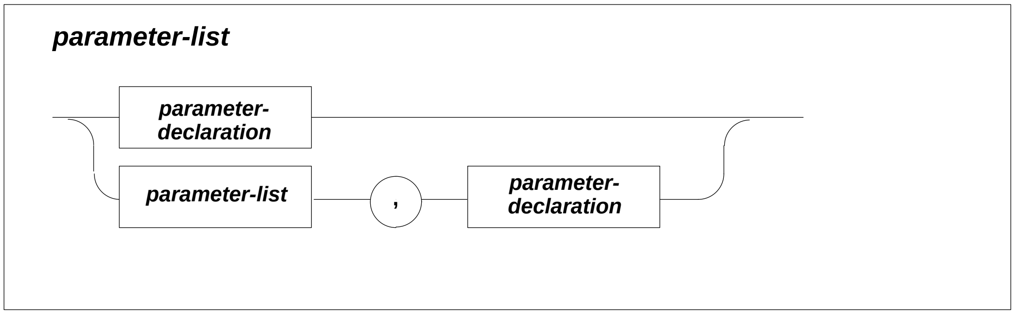


struct-declarator

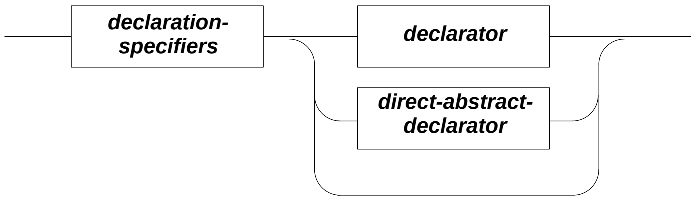



declarator
identifier
(
declarator
)
declarator
[
constant-expression
]
(
identifier-list
)
(
parameter-list
)

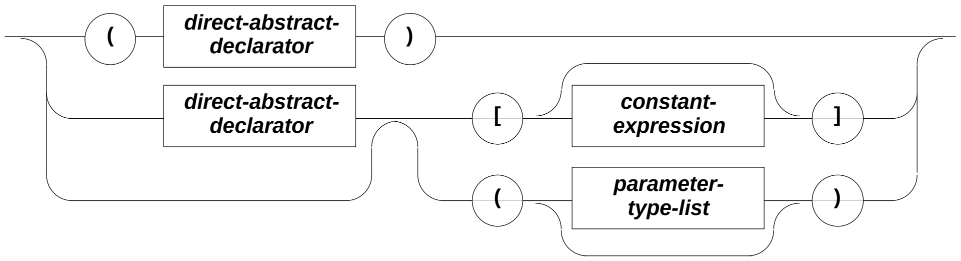

identifier-list
identifier
,


parameter-list
parameter-declaration
parameter-list
,
parameter-declaration

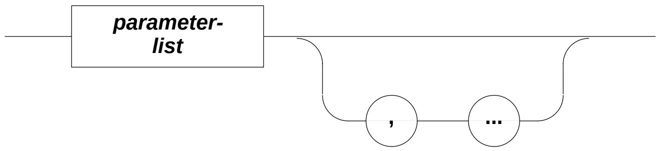
parameter-declaration

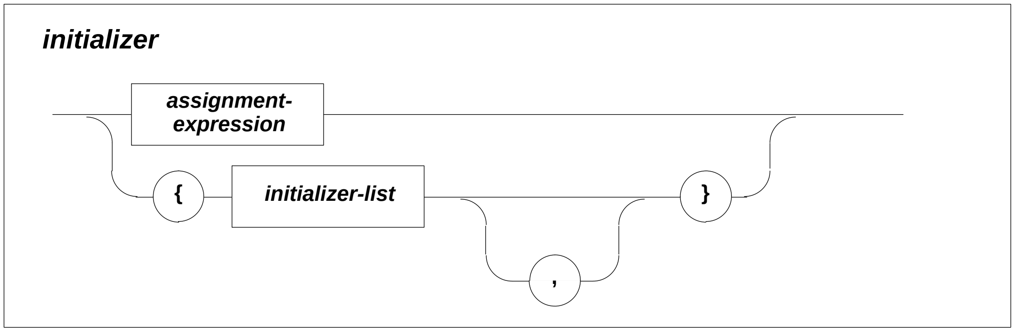


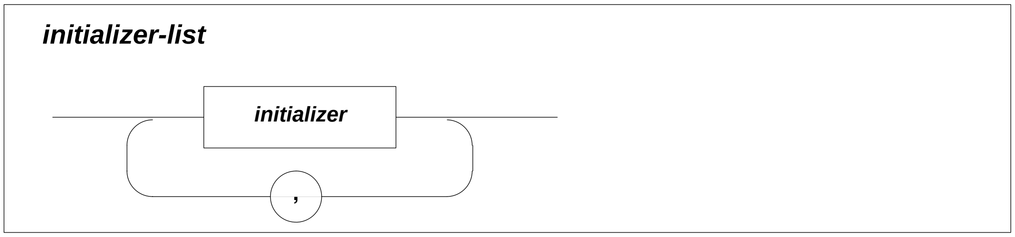
direct-abstract-declarator



parameter-type-list







Within embedded SQL statements the C naming qualification rules for structure and array elements are as defined in the ANSI standard (X3.159-1989) for C. There may be any number of SQL BEGIN DECLARE SECTIONs. Host variables must not be explicitly initialized in the declaration.

Ambiguous References and Multiple Declarations

A declaration that appears more than once with the same identifier is called a muliple declaration. If a host variable refers to such a multiple declaration, and the different declarations are of different types, an error occurs. Otherwise the host variable is accepted.

Data Type Conversion

The following table shows the conversion of C data types to SQL data types:

C Data Types	SQL Data Types
char (array)	CHARACTER
ESQ_BINARY (char array)	BINARY
long int/int long/long	INTEGER
unsigned long int/unsigned int long/unsigned long	INTEGER
signed long int/signed int long/signed long	INTEGER
float	REAL
double/long double/double long	DOUBLE PRECISION
short/short int/int short	SMALLINT
unsigned short/unsigned short int/unsigned int short	SMALLINT
signed short/signed int short/signed short int	SMALLINT
int/unsigned int/signed int	SMALLINT/INTEGER (machine-dependent)

In addition, all SQL numeric data types can be assigned to/from the pseudo data type ESQ_NUMERIC_STRING.

For more details on SQL data types and their usage in SQL statements refer to the *Adabas SQL Server Reference Manual*, chapter **Common Elements**. The C data type ‘int’ is only permitted in Adabas SQL Server mode. The size of an ‘int’ is usually dependent on the underlying hardware. Adabas SQL Server uses the C ‘sizeof’ macro to determine the exact size of an ‘int’ which will be equivalent to either a ’short’ or a ‘long’ and will be integrated as such.

The following tables shows the conversion of SQL data types to C data types:

SQL Data Types	C Data Types
CHAR (more than one character)	char (array)
BINARY	ESQ_BINARY (char array)
INTEGER	long
SMALLINT	short
REAL	float
DOUBLE PRECISION	double
FLOAT	float
DECIMAL	float, double or long
NUMERIC	float, double or long

Any C numeric data type is compatible with any SQL numeric data type. However, conversion between data types may be necessary and accuracy or fractional values may be lost.

Adabas SQL Server C-String Logic

When passing strings to and from Adabas SQL Server, the host variables used have to be declared as arrays of data type char. One extra character space has to be declared to accommodate the ’\0’ string terminator.

The Adabas SQL Server precompiler option TRAILING BLANK SUPPRESSION specifies whether the values returned from Adabas SQL Server will contain any trailing blanks before the terminator or not.

Passing data to Adabas SQL Server

When such a string variable is supplied to Adabas SQL Server only characters up to the ‘\0’ are significant and the ’\0’ is effectively removed from the string. The length of the string for SQL purposes is therefore up to — but not including — the ‘\0’.

Should the variable not contain a ’\0’ then only the first *n-1* characters are significant to Adabas SQL Server, where *n* is the declared length of the host variable character array. If such a variable is used to insert or update a field, then any discrepancies between the value length and the field length are corrected by either truncating the string or appending sufficient blanks at the end of the string.

Receiving data from Adabas SQL Server

When Adabas SQL Server assigns a string value to such a variable e.g., in a FETCH statement, then a ’\0’ is appended to the value. This is the reason for reserving the last position of a character array for the ’\0’. Should the variable not provide enough space only the first n-1 characters are returned with a ’\0’ being added into the *nth* (last) position.

Should the field be smaller than the variable, blanks are appended between the end of the value and the ’\0’ terminator in the final position. If, however, the precompiler option TRAILING BLANK SUPPRESSION is set, all string values returned from Adabas SQL Server come with trailing blanks removed. In this case, the ’\0’ terminator is after the last non-blank character.

Example:

```
EXEC SQL BEGIN DECLARE SECTION;
char h_surname [20];
EXEC SQL END DECLARE SECTION;
```

Error Handling

As already explained in the chapter **General Concepts of SQL Programming** earlier in this manual, textual error messages associated with a particular error number may be retrieved using the function esqerr.

The C program must also declare a character array to receive the error text, and an integer, which initially holds the length of this array. The length of the array must have the length of ESQ_ERROR_MESSAGE_SIZE, which is a macro generated at precompilation time. Both of these data items must be in scope whenever the esqerr() function is called. The appropriate SQLCA must also be in scope.

A programming example with a call to esqerr looks like this:

```
/* -------------------------------------------------- */
/* container for error-text                           */
/* (one character more for C string terminator EOS)   */
/* -------------------------------------------------- */
char my_error_text [ ESQ_ERROR_MESSAGE_SIZE + 1 ];
/* -------------------------------------------------- */
/* length and language container                      */
/* -------------------------------------------------- */
long   my_length;
long   my_language;
/* -------------------------------------------------- */
/* for each esqerr() call you have to update the      */
/* length and language container                      */
/* -------------------------------------------------- */
my_length = ESQ_ERROR_MESSAGE_SIZE;
my_language = ESQ_ENGLISH;
/* -------------------------------------------------- */
/* call esqerr()                                      */
/* -------------------------------------------------- */
esqerr(
  &sqlca,
  my_error_text,
  &my_length,
  &my_language        );
/* -------------------------------------------------- */
/* set C string terminator                            */
/* -------------------------------------------------- */
my_error_text [ my_length ] = (char)0;
/* -------------------------------------------------- */
/* from here on you can use the error-text as known   */
/* C string, for example:                             */
/* -------------------------------------------------- */
printf( my_error_text );
```

Where:		
	`sqlca`	is the SQLCA structure variable
	`my_error_text`	is the target buffer
	`my_length`	is the length of the target buffer
	`my_language`	is a macro generated by the precompiler
	`ESQ_ENGLISH`	is the English language indicator
	`ESQ_ERROR_MESSAGE_SIZE`	is the maximum length of the error-text-buffer returned by esqerr(). Value is 162.

The value contained in *my_length* informs the esqerr function of the initial target buffer size in bytes so that memory locations are not overwritten. This value must be reset with each call to esqerr. Upon return it contains the length of text contained in the buffer. The text itself is not null-terminated.

SQL Communication Area (SQLCA)

The SQLCA provides the programmer with comprehensive information about the success or failure of each SQL command. For further details refer to chapter **General Concepts of SQL Programming** earlier in this manual.

The following is the declaration of the SQLCA structure in C:

```
struct sqlca
 {   unsigned char sqlcaid [ 8 ];   /*   eye catcher 'sqlca'             */
    long sqlcabc;                   /*   size of SQLCA in bytes (136)    */
    long sqlcode;                   /*   SQL return code                 */
    short sqlerrml;                 /*   length of error message         */
    unsigned char sqlerrmc [ 70 ];  /*   error message                   */
    unsigned char sqlerrp [ 8 ];    /*   internal error information      */
    long sqlerrd [ 6 ];             /*   internal error information      */
    unsigned char sqlwarn [ 8 ];    /*   warning flags                   */
    unsigned char sqlext [ 8 ];     /*   reserved                        */
  };
```

SQL Descriptor Area (SQLDA)

The SQLDA provides the programmer with comprehensive information about each resulting column of a dynamic SELECT statement. For further details refer to chapter **General Concepts of SQL Programming** earlier in this manual.

The following is the declaration of the SQLDA structure in C:

```
struct sqlda
  { char sqldaid [ 8 ];           /* eye catcher: ’SQLDA’            */
    ESQ4 sqldabc;                 /* size of sqlda in bytes          */
    short sqln;                   /* #sqlvar elements allocated      */
    short sqld;                   /* #sqlvar elements returned       */
    struct sqlvar
    { short sqltype;              /* data type of variables          */
      short sqllen;               /* length of variable              */
      char *sqldata;              /* pointer to value of variable    */
      short *sqlind;              /* pointer to null indicator       */
      struct sqlname
      { short length;             /* length of element name          */
        char data [ 30 ];         /* name of element                 */
      } sqlname ;
    } sqlvar [ 1 ];
  };
#define SQLDASIZE(n) (sizeof(struct sqlda)+(n–1)*sizeof(struct sqlvar))
/* macros definitions for data types returned or set in sqltype      */

#define SQL_TYP_BINARY      (4)   /* BINARY binary data              */
#define SQL_TYP_NBINARY     (SQL_TYP_BINARY + SQL_TYP_NULLINC)
#define SQL_TYP_NUMERIC     (16)  /* NUMERIC(n,m) fixed point number */
#define SQL_TYP_NNUMERIC    (SQL_TYP_NUMERIC + SQL_TYP_NULLINC)
#define SQL_TYP_CHAR        (452) /* CHAR(n) fixed length string     */
#define SQL_TYP_NCHAR       (SQL_TYP_CHAR + SQL_TYP_NULLINC)
#define SQL_TYP_CSTR        (460) /* variable length C string        */
#define SQL_TYP_NCSTR       (SQL_TYP_CSTR + SQL_TYP_NULLINC)
#define SQL_TYP_FLOAT       (480) /* FLOAT floating point number     */
#define SQL_TYP_NFLOAT      (SQL_TYP_FLOAT + SQL_TYP_NULLINC)
#define SQL_TYP_DECIMAL     (484) /* DECIMAL(n,m) fixed point number */
#define SQL_TYP_NDECIMAL    (SQL_TYP_DECIMAL + SQL_TYP_NULLINC)
#define SQL_TYP_INTEGER     (496) /* INTEGER integer number          */
#define SQL_TYP_NINTEGER    (SQL_TYP_INTEGER + SQL_TYP_NULLINC)
#define SQL_TYP_SMALL       (500) /* INTEGER integer number          */
#define SQL_TYP_NSMALL      (SQL_TYP_SMALL + SQL_TYP_NULLINC)
```

11

EMBEDDING SQL STATEMENTS IN HOST LANGUAGES – COBOL

General Rules

SQL Statement Delimiters

SQL statements are delimited by the prefix EXEC SQL and the terminator END-EXEC. The prefix may be written in upper or lower case letters. In Adabas SQL Server mode upper or lower case is permitted and the prefix may be split over numerous lines, separated by any white space character. In ANSI mode upper case is required and only white spaces may separate the prefix keywords.

The terminator END-EXEC may be followed by an optional period '.'. It has to be coded at the same line to be recognized. However, the generated code is different for the two COBOL standards that are supported.

For the ANSI 74 standard (Precompiler setting COBOL II = off) every SQL statement is treated as if the optional period was coded. That means the generated code will always be terminated with a period. It is not possible to code more than one SQL statement in an IF statement.

For the ANSI 85 standard (Precompiler setting COBOL II = on) the terminating period is only generated if a period was coded. If SQL statements are nested in an IF-set of COBOL statements, the ending period must not be coded.

SQL Statement Placement

All SQL statements with the exception of the BEGIN/END DECLARE and INCLUDE statements may be specified wherever a COBOL statement may be specified within the Procedure Division of the embedded SQL COBOL program. Included COBOL source code must not contain any SQL statements nor any host variable declaration to be used in SQL statements.

The SQL INCLUDE , BEGIN/END DECLARE statements must be specified in the WORKING STORAGE SECTION of the COBOL program.

Comments

SQL statements may contain COBOL comments marked with an asterisk in column 7 or SQL comments preceded by two minus characters.

COBOL Example :

```
000015     EXEC SQL WHENEVER SQLERROR
000016*                              CONTINUE
000017                               GOTO HANDLE–ERROR
000018     END–EXEC.
```

SQL Example :

```
000015     EXEC SQL WHENEVER SQLERROR
000016                            –– CONTINUE
000017                               GOTO HANDLE–ERROR
000018     END–EXEC.
```

Host Variables

COBOL host variables used in SQL statements must be declared within the SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements as well as in the COBOL DATA DIVISION. There may be any number of SQL BEGIN DECLARE SECTIONs. The host variable definition must be a valid COBOL data declaration, as described below. Adabas SQL Server allows the use of single host variables and host variable structures.

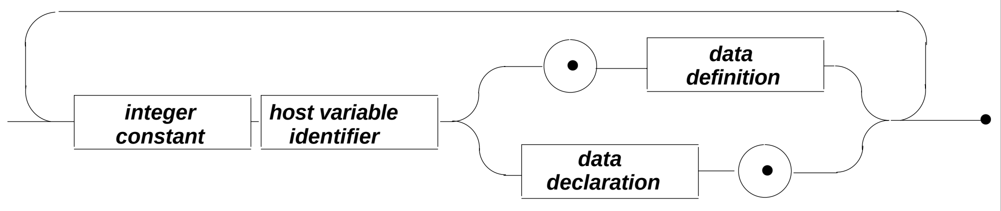
Host Variable Declaration

COBOL host structures are a named set of COBOL single host variables and must conform to the ANSI Standard for COBOL.

Host Variable Structures

The use of COBOL host structures within SQL statements is an Adabas SQL Server extension and not part of the SQL ANSI Standard.

Data Definition



integer constant	level number as described in the ANSI Standard for COBOL.
host variable identifier	specifies the identifier of the COBOL single variable or structure. Any valid COBOL identifier may be used.
data definition	recursive definition for nested structure level specification.
data declaration	see the syntax diagram below.

Example of a structure definition:

```
01          LEVEL1.
    02      LEVEL2.
       05   ELEMENT1 PIC 9.
       05   ELEMENT2 PIC 9.
```

Within embedded SQL statements the COBOL naming qualification rules for structure elements do not apply. Instead they must be specified top down to read *LEVEL1.LEVEL2.ELEMENT1* as shown in the example above.

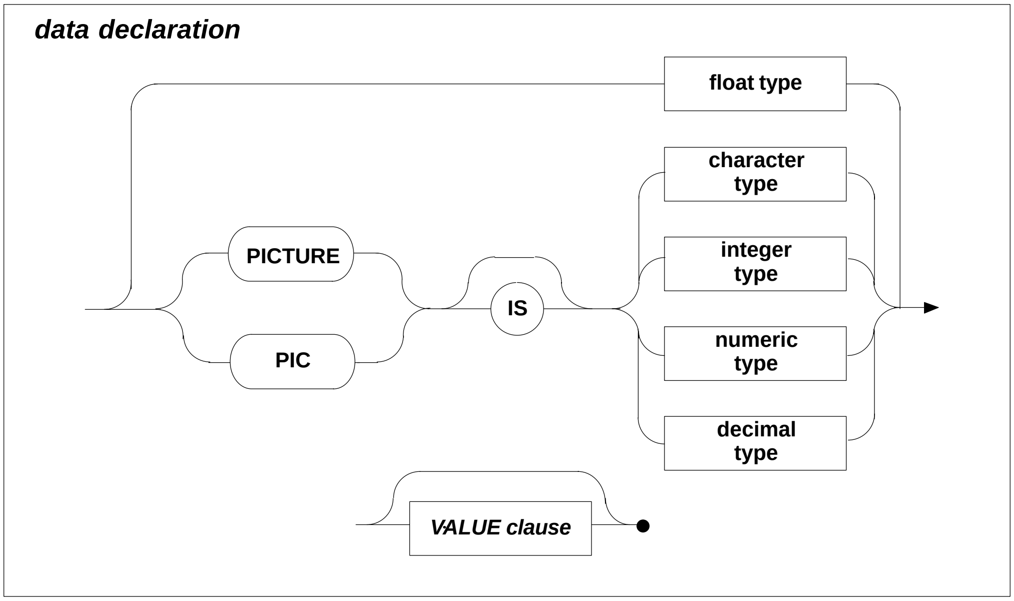
In COBOL statements, however, the structure elements must still be specified (bottom up) according to the ANSI COBOL rules: *e.g. ELEMENT1 IN LEVEL2 IN LEVEL1*.

When referencing a structure element which is not uniquely identified within the compilation unit it must be sufficiently qualified with enough containing structure identifiers to unambiguously identify the variable concerned. If for example the identifier *ELEMENT1* has been used in more than one structure definition then it must be qualified to give either *LEVEL2.ELEMENT1* or even if necessary *LEVEL1.LEVEL2.ELEMENT1*.

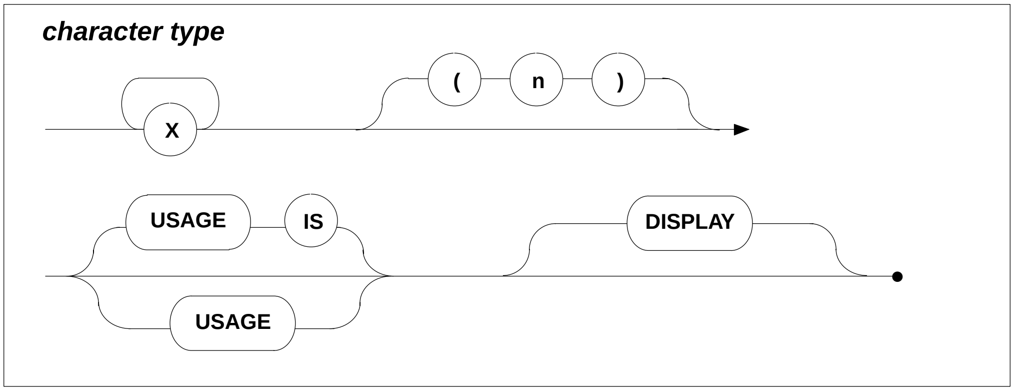
For more information about the general usage of host variables within SQL the chapter **Common Elements** in the *Adabas SQL Server Reference Manual* or chapter **General Concepts of SQL Programming** in the *Adabas SQL Server Programmer’s Guide*.

Single Host Variables

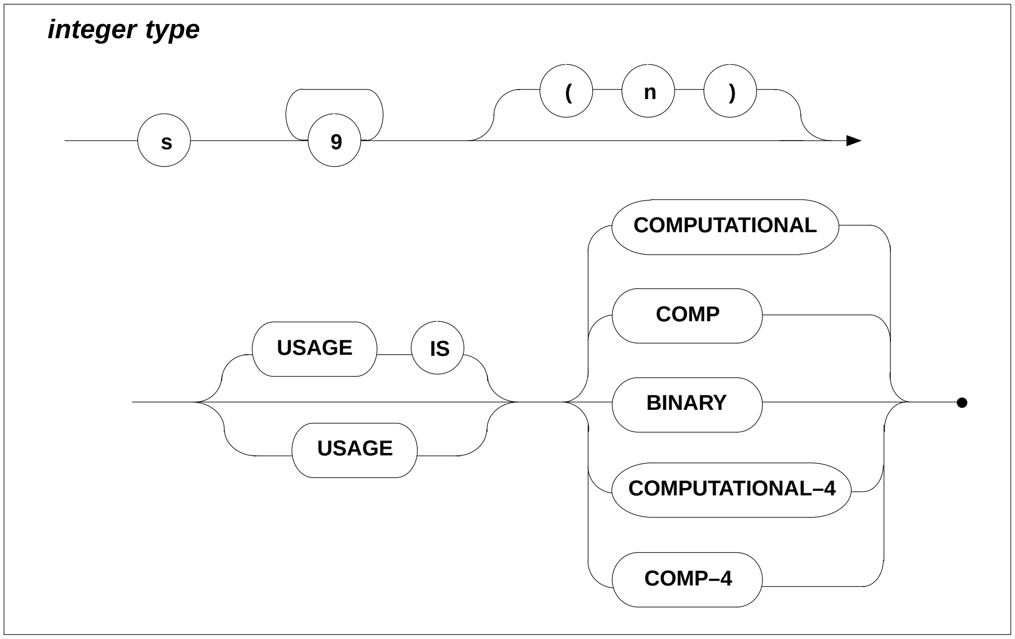
The declaration must conform to the following COBOL syntax.



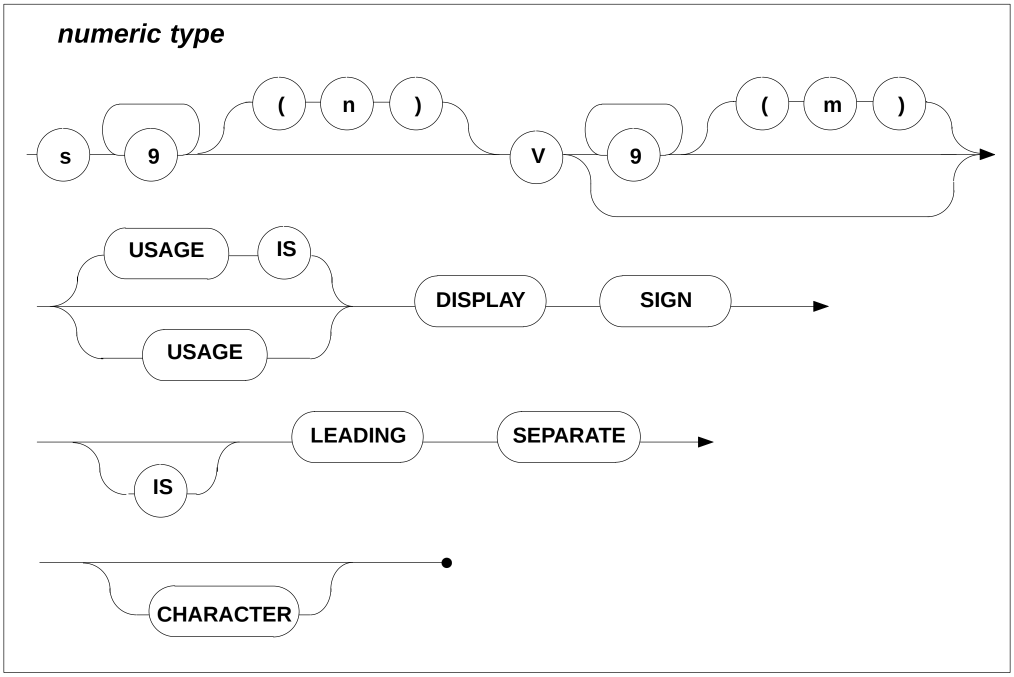
VALUE clause specifies any valid COBOL VALUE clause.



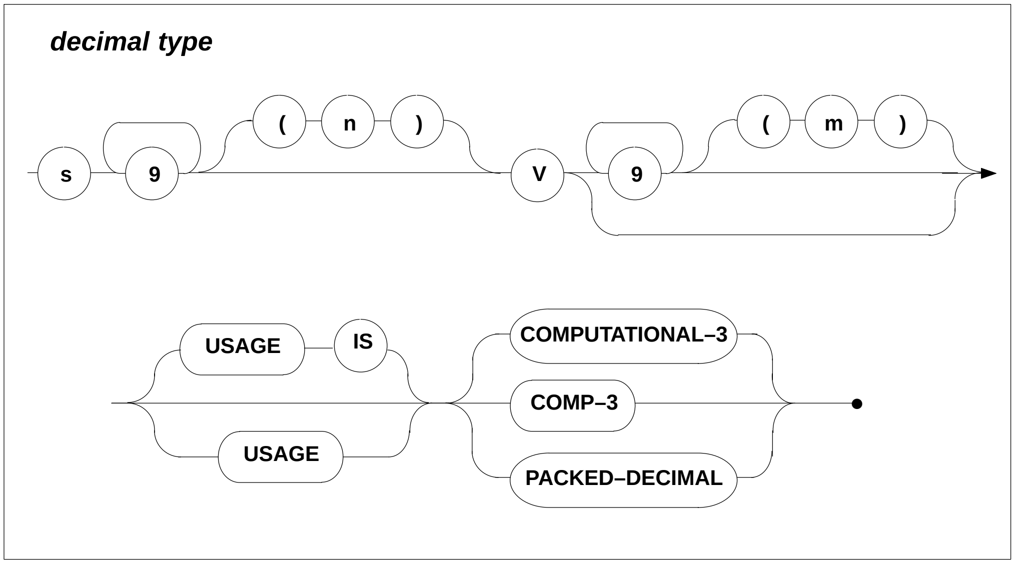
The number of significant characters must not exceed 253.



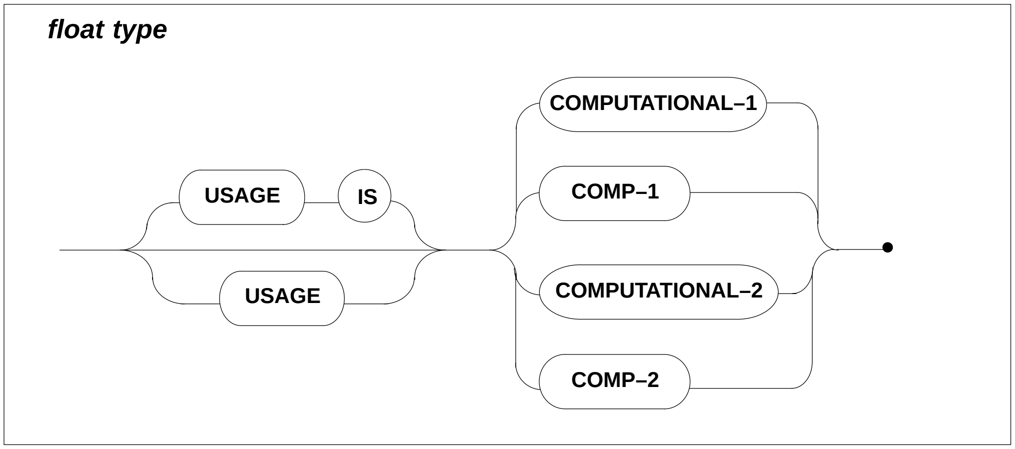
The number of digits must not exceed 9.



The number of digits must not exceed 27.



The number of digits must not exceed 27.



Data Type Conversion

The following table shows the conversion of COBOL data types to SQL data types and vice versa:

COBOL Data Types	SQL Data Types
character	CHARACTER
char (array)	BINARY
integer (5–9 digits)	INTEGER
integer (1–4 digits)	SMALLINT
numeric	NUMERIC
decimal	DECIMAL
float (comp–1)	REAL
float (comp–2)	DOUBLE–PRECISION

For more details on SQL data types and their usage in SQL statements refer to the *Adabas SQL Server Reference Manual*, Chapter **Common Elements**. The COBOL data type 'COMP-3' is not permitted in ANSI compatibility mode. The number of digits for an integer type must not exceed 9.

Error Handling

As already generally explained in the Chapter: **General Concepts of SQL Programming**, earlier in this manual, textual error messages associated with a particular error number may be retrieved using the sub-program “esqerr”.

Using the SQL statement INCLUDE SQLCA the declaration of the SQLCA structure must be made known to the COBOL program.

The COBOL program must also contain the declaration of a character variable to receive the error text and an integer variable holding the length of it. The length should be 300 bytes and must be set to that value prior to every call of “esqerr”. Upon return it contains the length of the retrieved error text.

Furthermore an integer variable must be declared to receive a language code number determining the natural language of the message texts.

An example using the sub-program “esqerr” could look like this:

```
000001 IDENTIFICATION DIVISION.
000002 PROGRAM–ID.  TEST.
000003 ENVIRONMENT DIVISION.
000004 DATA DIVISION.
000005 WORKING–STORAGE SECTION.
000006     EXEC SQL BEGIN DECLARE SECTION END–EXEC.
000007 01  UID              PIC X(18) VALUE ”ESQ”.
000008     EXEC SQL END   DECLARE SECTION END–EXEC.
000009     EXEC SQL INCLUDE SQLCA END–EXEC.
000010 01  ERR–MESSAGE       PIC X(300).
000011 01  MESSAGE–LEN  COMP PIC S9(5) VALUE 300.
000012 01  ESQ–LANGUAGE COMP PIC S9(5) VALUE 0.
000013 PROCEDURE DIVISION.
000014 FIRST SECTION.
000015     EXEC SQL WHENEVER SQLERROR CONTINUE END–EXEC.
000016     EXEC SQL CONNECT :UID END–EXEC.
000017     IF SQLCODE = 0
000018        DISPLAY ”CONNECTED”
000019     ELSE PERFORM ERROR–MESSAGE.
000020     EXEC SQL DISCONNECT END–EXEC.
000021     STOP RUN.
000022 ERROR–MESSAGE.
000023     DISPLAY ”FAILED WITH SQLCODE ” SQLCODE.
000024     MOVE SPACES TO ERR–MESSAGE.
000025     MOVE 0      TO ESQ–LANGUAGE.
000026     CALL ”ESQERR” USING BY REFERENCE SQLCA
000027                         BY REFERENCE ERR–MESSAGE
000028                         BY REFERENCE MESSAGE–LEN
000029                         BY REFERENCE ESQ–LANGUAGE.
000030     DISPLAY ”ERROR : ” ERR–MESSAGE.
```

SQL Communication Area (SQLCA)

The SQLCA provides the programmer with comprehensive information about the success or failure of each SQL command. For further details refer to chapter **General Concepts of SQL Programming** earlier in this manual.

The following is the declaration of the SQLCA structure in COBOL:

```
01 SQLCA.
   05 SQLCAID     PIC X(8)       VALUE ”sqlca   ”.
   05 SQLCABC     PIC S9(9) COMP VALUE +136.
   05 SQLCODE     PIC S9(9) COMP VALUE +0.
   05 SQLERRM.
      49 SQLERRML PIC S9(4) COMP.
      49 SQLERRMC PIC X(70).
   05 SQLERRP     PIC X(8).
   05 SQLERRD     OCCURS 6 TIMES
                  PIC S9(9) COMP.
   05 SQLWARN.
      10 SQLWARN0 PIC X.
      10 SQLWARN1 PIC X.
      10 SQLWARN2 PIC X.
      10 SQLWARN3 PIC X.
      10 SQLWARN4 PIC X.
      10 SQLWARN5 PIC X.
      10 SQLWARN6 PIC X.
      10 SQLWARN7 PIC X.
   05 SQLEXT      PIC X(8).
```

SQL Descriptor Area (SQLDA)

The SQLDA provides the programmer with comprehensive information about each resulting column of a dynamic SELECT statement. For further details refer to chapter **General Concepts of SQL Programming** earlier in this manual.

The following is the declaration of the SQLDA structure in COBOL:

```
01 SQLDA.
   05 SQLDAID     PIC X(8)       VALUE “sqlda   ”.
   05 SQLCABC     PIC S9(9) COMP VALUE +15360.
   05 SQLN        PIC S9(9) COMP VALUE +256.
   05 SQLD        PIC S9(4) COMP VALUE +0.
   05 SQLERRM OCCURS 1 TO 256 TIMES
              DEPENDING ON SQLN.
      10 SQLTYPE  PIC S9(4) COMP.
      10 SQLLEN   PIC S9(4) COMP.
      10 SQLDATA  POINTER.
      10 SQLIND   POINTER.
      10 SQLNAME.
         15 SQLNAMEL  PIC S9(4) COMP.
         15 SQLNAMEC  PIC X(30).
```

This structure may be used in COBOL II only. The COBOL statement SET has to be used to set the pointers SQLDATA and SQLIND to a storage area. Programs for other compilers must provide an assembler or C subroutine to do this.

EMBEDDING SQL STATEMENTS IN HOST LANGUAGES – PL/I

General Rules

SQL Statement Delimiters

SQL statements are delimited by the prefix EXEC SQL and the terminator ’;’ . The prefix may be written in upper or lower case letters. In Adabas SQL Server mode upper or lower case is permitted and the prefix may be split over numerous lines, separated by any white space characters. In ANSI mode upper case is required and only white spaces may separate the prefix keywords.

SQL Statement Placement

An SQL statement with the exception of BEGIN DECLARE, END DECLARE and INCLUDE statements may be specified wherever a PL/I statement may be specified within a procedure block. Included PL/I source code must not contain any SQL statements nor any host variable declaration to be used in SQL statements.

The SQL INCLUDE statement must be specified inside the procedure block. The BEGIN DECLARE and END DECLARE statements may be specified wherever a PL/I declaration is permitted, with the exception that a host variable has to be declared before its usage.

Comments

A comment begins with a /* and ends with */. Any character may appear between /* and */ except the consecutive pair */. A comment is permitted wherever a PL/I comment is permitted. In addition, PL/I comments are also allowed between EXEC SQL and ’;’.

PL/I host variables used in SQL statements must be declared within the SQL BEGIN DECLARE and END DECLARE section statements. Adabas SQL Server allows the use of single host variables, arrays of host variables and host variable structures.

```
EXEC SQL WHENEVER SQLERROR
  /* CONTINUE */
  GOTO HANDLE–ERROR;
```

SQL Example:

```
EXEC SQL WHENEVER SQLERROR
                       ––   CONTINUE
                            GOTO HANDLE–ERROR
END–EXEC.
```

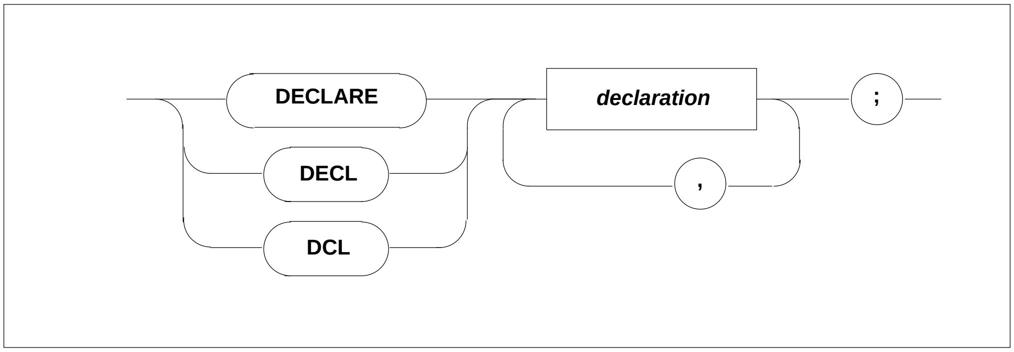
Host Variables

PL/I host variables used in SQL statements must be declared within the SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements. Adabas SQL Server allows the use of single host variables, arrays of host variables and host variable structures.

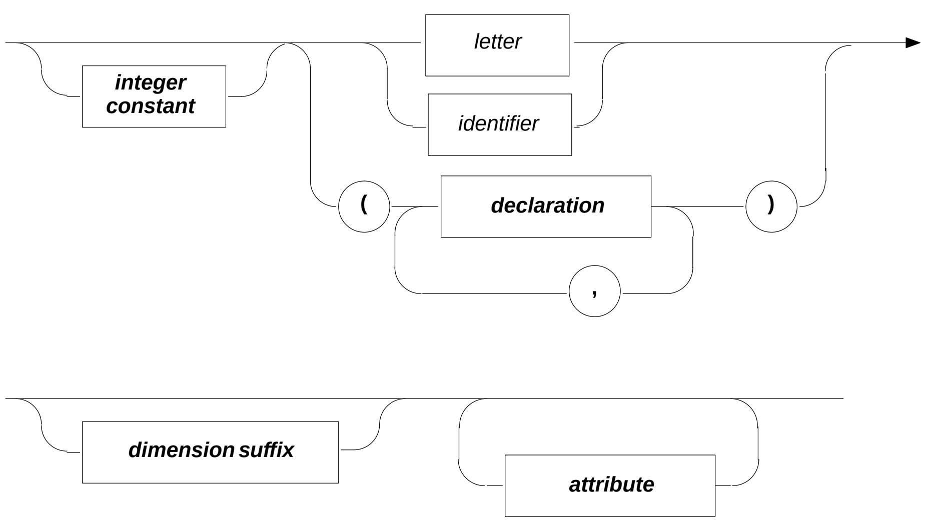
Host Variable Declaration

PL/I arrays and structures are named sets of PL/I single host variables and must conform to the ANSI Standard (X3.53-1576) for PL/I. The syntax diagrams below show the PL/I Declare Statement:

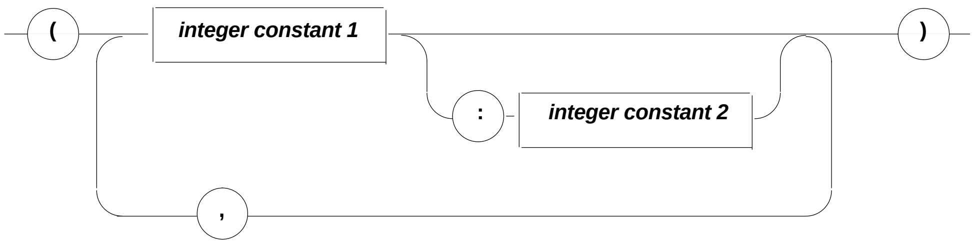
The values to be entered must conform to the ANSI Standard for PL/I.

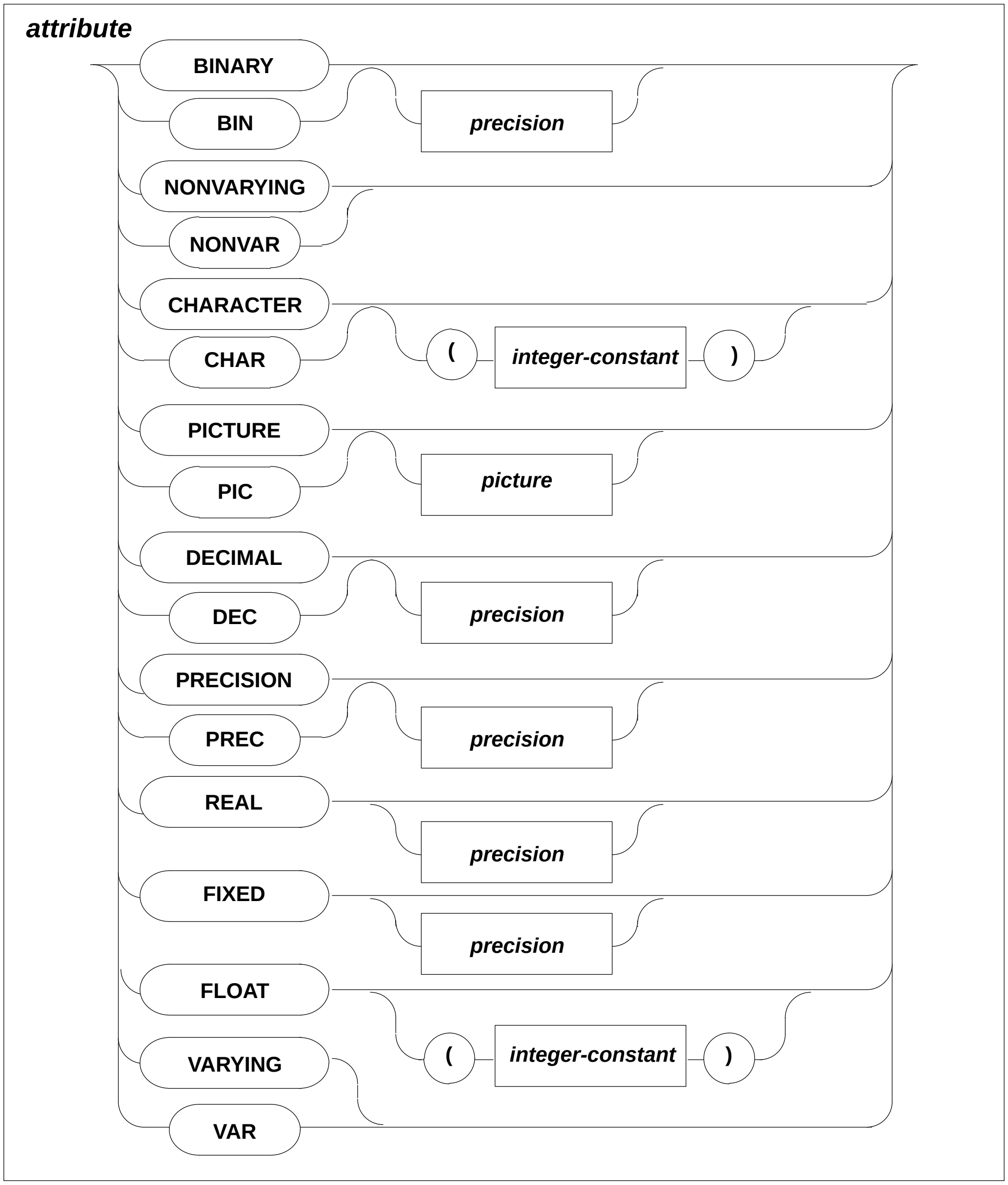


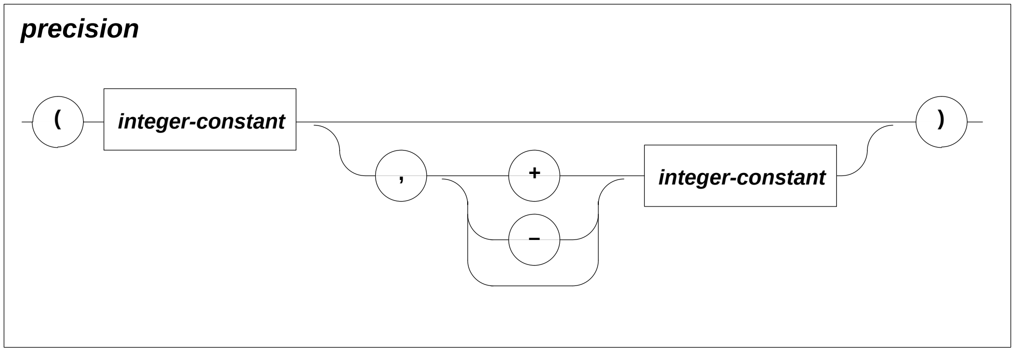
declaration



dimension suffix

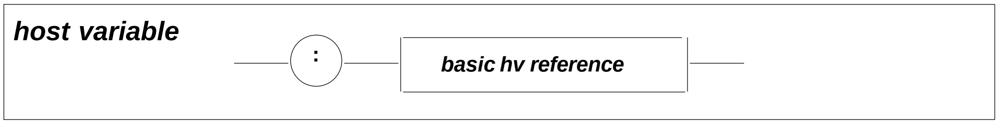


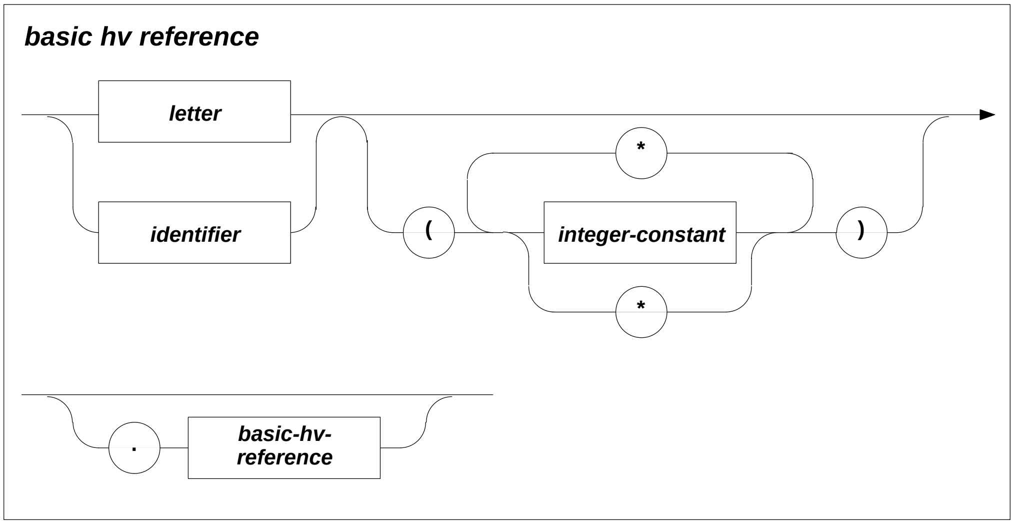

attribute
BINARY
BIN
precision
NONVARYING
NONVAR
CHARACTER
CHAR
(
integer-constant
)
PICTURE
PIC
picture
DECIMAL
DEC
precision
PRECISION
PREC
precision
REAL
precision
FIXED
precision
FLOAT
(
integer-constant
)
VARYING
VAR


precision
(
integer-constant
,
+
–
integer-constant
)

Single Host Variables, Arrays and Structures

The host variable reference in an SQL statement is reflected in the following diagram:





The use of PL/I structures within SQL statements is an Adabas SQL Server extension and not part of the SQL ANSI standard.

Within embedded SQL statements the PL/I naming qualification rules for structure and array elements are as defined in the ANSI standard (X3.53-1576) for PL/I.

If a host variable refers to an elementary data type a single host variable of that data type is generated. If a host variable refers to a structure/substructure or array/sub–array it will be broken down to elementary data types as the following example shows:

Example: single host variables

```
DECLARE HV (5) BIN FIXED;
EXEC SQL
     SELECT column_xxx INTO :HV(2) from table_xxx
END EXEC;
```

In this example, HV(2) refers to a single host variable of type REAL BIN FIXED, therefore, only one host variable is generated.

Example: arrays

```
EXEC SQL
     SELECT column_a, column_b, column_c, column_e, column_f
     ,  INTO :HV from table_xxx
END EXEC;
```

In this example, HV refers to an array of 5 host variables of type REAL BIN FIXED, therefore, five host variables with the same type are generated.

Example: structures

```
DECLARE  1I,  2J,  3 K (2)  BIN FIXED;
                   3 L      BIN FIXED;
              2 M  3 N      BIN FIXED;
```

In this example, host variable references are as follows:

:I.J.K(1)	refers to a single host variable of type BIN FIXED.
:I.J	refers to I.J.K(1), I.J.K(2) and I.J.L.
:I	refers to I.J.K(1), I.J.K(2), I.J.L and I.M.N.

Ambiguous References and Multiple Declarations

A declaration that appears more than once with the same identifier is called a multiple declaration. If a host variable refers to such a multiple declaration, and the different declarations are of different types, an error occurs. Otherwise, the host variable is accepted.

When referencing a structure element which is not uniquely identified within the compilation unit it must be sufficiently qualified with enough containing structure identifiers to unambiguously identify the variable concerned: as shown in the example below:.

```
DCL 1 I,
         3 J,
         3 J,
         3 K,
              4 L,
         3 L;
```

The declaration of J is multiple. I.L refers to the L of level 3 because this is a complete qualification. A reference to L of level 4 would be I.K.L.

The following examples show unambiguous declarations.

```
DCL 1 I,
         2 J,
              3 K,
         2 J,
              3 K;
```

I.K is ambiguous

```
DCL 1 I,
         2 I,
              3 I;
```

I refers to I of level 1, I.I to I of level 2 and I.I.I to I of level 3.

```
DCL J;
DCL 1 I,
         2 J,
              3 K,
              3 L,
         2 I,
              3 K,
              3 L;
```

J refers to the first DCL statement. I.K is ambiguous. I.J.K refers to the first K and I.I.K to the second one.

Data Type Conversion

The following table shows the conversion of PL/I data types to SQL data types and vice versa:

PL/I Data Types	**SQL Data Types**
CHARACTER	CHARACTER
CHARACTER	BINARY
BIN FIXED	INTEGER
DEC FIXED	NUMERIC
DEC FLOAT	REAL
BIN FLOAT (4 Bytes)	REAL
BIN FLOAT (8 Bytes)	DOUBLE PRECISION

For more details on SQL data types and their usage in SQL statements refer to the *Adabas SQL Server Reference Manual*, Chapter **Common Elements.**

Error Handling

As already generally explained in the chapter **General Concepts Of SQL Programming** earlier in this manual, textual error messages associated with a particular error number may be retrieved using the sub-routine ”esqerr”.

The PLI program must also declare a character variable to receive the error text and an integer variable holding the length of it. The array should be 300 bytes in length. Both of these data items must be in scope whenever the esqerr subroutine is called. The appropriate SQLCA must also be in scope.

An example using the subroutine ”esqerr” looks like this:

```
DCL ESQERR ENTRY (ANY,ANY,ANY,ANY) EXTERNAL (’esqerr’);EXAMPLE: PROCEDURE
OPTIONS (MAIN);
EXEC SQL
   BEGIN DECLARE SECTION;
   DCL UID              CHARACTER (18) VARYING;
EXEC SQL
   END DECLARE SECTION;EXEC SQL
   INCLUDE SQLCA;
DCL MESSAGE_LEN    FIXED BIN (16) INIT (300);
DCL ERR_MESSAGE    CHARACTER (300);
DCL FINAL_ERR_MESS CHARACTER (300);
DCL ESQ_ENGLISH    FIXED BIN (16) INIT (0);
EXEC SQL
   WHENEVER SQLERROR CONTINUE;
ON ERROR BEGIN;
   PUT EDIT (’ERROR CODE :’ ,ONCODE()) (A,F(6)) SKIP;
   PUT LIST (’Program aborted!’) SKIP;
   PUT SKIP (1);
   STOP;
END;
UID = ’ESQ’;
PUT EDIT ( ’Connecting to ESQ AS ’,UID,’ ... ’ ) (A,A,A) SKIP;
EXEC SQL
   CONNECT :UID;
IF SQLCODE = 0 THEN
DO;
   PUT EDIT (’CONNECTED’) (A);
END;
ELSE
DO;
   PUT EDIT (’FAILED with SQLCODE ’, SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,(ESQ_ENGLISH));
   FINAL_ERR_MESS = COPY(’ ’,300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( ’ERROR: ’ ) ;
   PUT LIST ( FINAL_ERR_MESS ) SKIP ;
END ;
```

SQL Communication Area (SQLCA)

The SQLCA provides the programmer with comprehensive information about the success or failure of each SQL command. For further details refer to the chapter **General Concepts of SQL Programming** earlier in this manual.

In PL/I, the SQLCA structure is declared as follows:

```
DCL 1 SQLCA,
     2 SQLCAID     CHAR(8) INIT (’SQLCA ’),
     2 SQLCABC     BIN FIXED(31) INIT (136),
     2 SQLCODE     BIN FIXED(31) INIT (0),
     2 SQLERRM     CHAR(70) VAR INIT (’’),
     2 SQLERRP     CHAR(8) INIT (’’),
     2 SQLERRD(6)  BIN FIXED(31),
     2 SQLWARN,
          3 SQLWARN0   CHAR(1) INIT (’ ’),
          3 SQLWARN1   CHAR(1) INIT (’ ’),
          3 SQLWARN2   CHAR(1) INIT (’ ’),
          3 SQLWARN3   CHAR(1) INIT (’ ’),
          3 SQLWARN4   CHAR(1) INIT (’ ’),
          3 SQLWARN5   CHAR(1) INIT (’ ’),
          3 SQLWARN6   CHAR(1) INIT (’ ’),
          3 SQLWARN7   CHAR(1) INIT (’ ’),
     2 SQLEXT          CHAR(8) INIT (’’);
```

SQL Descriptor Area (SQLDA)

The SQLDA provides the programmer with comprehensive information about each resulting column of a dynamic SELECT statement. For further details refer to the chapter **General Concepts of SQL Programming** earlier in this manual.

In PL/I, the SQLDA structure is declared as follows:

```
DCL 1 SQLDA              BASED(SQLDAPTR),
     2 SQLDAID           CHAR(8),
     2 SQLDABC           BIN FIXED(31),
     2 SQLN              BIN FIXED(15),
     2 SQLD              BIN FIXED(15),
     2 SQLVAR            (SQLSIZE REFER (SQLN)),
          3 SQLTYPE      BIN FIXED(15),
          3 SQLLEN       BIN FIXED(15),
          3 SQLDATA      POINTER,
          3 SQLIND       POINTER,
          3 SQLNAME      CHAR(30) VAR;
```

DB2 TRANSACTION MODE

Adabas SQL Server does not support DB2-specific SQL statements, and was not designed to provide standard application packages SQL-based access to Adabas.

Adabas SQL Server provides a compiler mode setting for “DB2 transaction mode”, which must be set during the precompilation of the source program. The DB2 compiler mode setting restricts the precompiler, permitting only the use of ANSI standard statements.

DB2 transaction mode also enables Adabas SQL Server to issue transaction control statements (COMMIT, ROLLBACK) in a CICS environment and under certain circumstances also in batch mode.

DB2 transaction mode enables Adabas SQL Server

- To automatically create and destroy user sessions (CONNECT, DISCONNECT), and
- To issue transaction-control statements (COMMIT, ROLLBACK).

Note:
DB2 transaction mode is only available on IBM mainframe platforms.

Interface Requirements

The interface requirements are described in the installation manual and are environment dependent.

Environment	Module	Description
Batch mode	ESQCTRL	Adabas SQL DB2-mode control program
CICS	ESQRMCI	Enables the Adabas SQL / CICS interface
	ESQOCI	Adabas SQL link interface
	ESQTRUE	Adabas SQL task-related user exit
	ESQLNKCI	Adabas SQL link module

Please note that when you are executing under CICS, special parameter settings are required in the VO-parameter module (VOPRM).

For a detailed description of the installation requirements, please refer to the section **Phase D: Installing Clients** in Chapter 3 **Installing the Adabas SQL Server System** of the *Adabas SQL Server Installation and Operation Manual*.

User Session

Adabas SQL Server requires that all users be defined in the server catalog by using the DDL statement CREATE USER and that a user registers with the server by using a CONNECT statement, prior to issuing any other SQL statement.

The CONNECT and DISCONNECT statements are extensions to the ANSI standard, and thus are not permitted in DB2 transaction mode.

To solve this, an ‘implicit’ CONNECT statement is generated and passed with the first SQL statement. The user identification which is used in the ‘implicit’ CONNECT is dependent upon many factors and is described in the section **CONNECT** in Chapter 2 **SQL Statements** of the *Adabas SQL Reference Manual*.

Under CICS, the user session is automatically terminated via an ‘implicit’ DISCONNECT at the end of a CICS transaction or at the end of a CICS task.

In batch mode, the user session is terminated at the end of program execution.

Transaction Logic

Under CICS, the Adabas SQL Server module ESQTRUE is called by the following CICS services to manage the transaction logic:

- The CICS SYNCPOINT Manager, and
- The CICS Task Manager

When called, ESQTRUE executes the appropriate SQL statement, either a ROLLBACK or a COMMIT.

In batch mode, the control program ESQCTRL issues a COMMIT upon program completion.

Note:
The size of the ADABAS Hold Queue limits the size of a SQL transaction.

Security Considerations

The use of DB2 transaction mode with external security is not recommended.

In DB2 transaction mode, an 'implicit' CONNECT is generated and issued with the first SQL statement. A password is not provided in the 'implicit' CONNECT statement.

The external security interface used by the Adabas SQL server requires BOTH -

- The user identification and
- The user password.

As the password has not been supplied, the authentication check will always terminate unsuccessfully with the SQL response code ESQ6701.

Note:
You are not recommended to use DB2 transaction mode with the external security interface.

APPENDIX A — THE SAMPLE TABLES

Throughout the entire Adabas SQL Server manual set reference is made to and examples are based on the tables SAILOR, YACHT, CONTRACT, CRUISE and PERSON of the database YACHT_DB.

These tables do not contain multiple-value fields or periodic groups (MU/PE). To create MU/PE-related examples the cluster city_guide was added to the catalog.

Data Definition for the Base Table “SAILOR”

```
( sailor_id           integer           index    ind_sailor not null unique,
age                   integer           ,
sailor_name           char(30)          ,
experience            char(1)           ,
l_1                   char(3)           ,
l_2                   char(3)           ,
l_3                   char(3)           ,
id_contract           integer           not null,
id_cruise             integer           not null );
```

Data Definition for the Base Table “YACHT”

```
( yacht_id                integer           index    ind_yacht not null
unique,
yacht_name            char(30)          ,
id_owner              integer           not null,
yacht_type            char(30)          ,
yacht_length          numeric (5,2)     ,
yacht_width           numeric (5,2)     ,
yacht_draft           numeric (5,2)     ,
sail_surface          integer           ,
motor                 integer           ,
head_room             numeric (5,2)     ,
bunks                 integer           not null );
```

Data Definition for the Base Table "CONTRACT"

```
( contract_id          integer          index  ind_contract not null unique,
price                  numeric (13,3)   not null,
date_reservation       integer          ,
date_booking           integer          ,
date_cancellation      integer          ,
date_deposit           integer          ,
amount_deposit         numeric (13,3)       ,
date_payment           integer          ,
amount_payment         numeric (13,3)       ,
id_customer            integer          not null,
id_cruise              integer          not null )
```

Data Definition for the Base Table "CRUISE"

```
( cruise_id            integer          index  ind_cruise not null unique,
start_date             integer          ,
start_time             integer          ,
end_date               integer          ,
end_time               integer          ,
start_harbor           char(20)         ,
destination_harbor     char(20)         ,
cruise_price           numeric (13,3)       not null,
bunk_number            integer          ,
bunks_free             integer          not null,
id_yacht               integer          not null,
id_skipper             integer          not null,
id_predecessor         integer          ,
id_successor           integer           )
```

Data Definition for the Base Table “PERSON”

```
( person_id             integer          index    ind_person not null
unique,birth_date       integer          ,
sex                     char(1)          ,
surname                 char(20)         ,
first_name_1            char(20)         ,
first_name_2            char(20)         ,
title                   char(20)         ,
form_of_address         char(8)          ,
address_addition_1      char(20)         ,
address_addition_2      char(20)         ,
street_number           char(20)         ,
country                 char(3)          ,
zip_code                char(10)         ,
city                    char(20)         ,
area_code_priv          char(6)          ,
phone_number_priv       char(15)         ,
area_code_office        char(6)          ,
phone_number_office     char(15)         );
```

Data Definition for the Cluster “CITY-GUIDE”

The following shows the SQL definition of Adabas structures which summarize information about states, cities, places and buildings in one file and serve as sample data for MU/PE examples.

```
CREATE CLUSTER DESCRIPTION city_guide
 DATABASE DB_214 FILE NUMBER 134
 (
 CREATE TABLE DESCRIPTION states (
    abbreviation     SHORTNAME ’AA’ PRIMARY KEY DEFAULT ADABAS,
    state_name       SHORTNAME ’AB’ UNIQUE NOT NULL DEFAULT ADABAS,
    capital          SHORTNAME ’AC’ INDEX,
    population       SHORTNAME ’AD’
    )

 CREATE TABLE DESCRIPTION cities (
    state_abbrev     SHORTNAME ’AA’,
    city_seqno       SEQNO(1)   NOT NULL DEFAULT ADABAS,
    city_name        SHORTNAME ’BA’ INDEX NULL SUPPRESSION,
    population       SHORTNAME ’BB’ NULL SUPPRESSION,
    PRIMARY KEY (state_abbrev, city_seqno),
    FOREIGN KEY (state_abbrev) REFERENCES states,
    UQINDEX ( city_name, state_abbrev)
    )

 CREATE TABLE DESCRIPTION buildings (
    state_abbrev     SHORTNAME ’AA’,
    city_seqno       SEQNO(1)   NOT NULL DEFAULT ADABAS,
    building_seqno   SEQNO(2)   NOT NULL DEFAULT ADABAS,
    building_name    SHORTNAME ’CA’ NOT NULL SUPPRESSION,
    height           SHORTNAME ’CB’ NOT NULL SUPPRESSION,
    PRIMARY KEY (state_abbrev, city_seqno, building_seqno),
    FOREIGN KEY (state_abbrev, city_seqno) REFERENCES cities
    )

 CREATE TABLE DESCRIPTION places (
    state_abbrev     SHORTNAME ’AA’ NOT NULL,
    city_seqno       SEQNO(1)   NOT NULL DEFAULT ADABAS,
    place_name       SHORTNAME ’DA’ NULL SUPPRESSION,
    FOREIGN KEY (state_abbrev, city_seqno) REFERENCES cities
    )
 );
```

APPENDIX B — SAMPLE PROGRAMS

Sample C–Program for the Creation of the Following SQL Tables

```
/*****************************************************************
*                                                                *
*                Name –– yacht_cr.pc                             *
*                                                                *
*         Description –– Program to create the tables :          *
*                                                                *
*                         contract                               *
*                         cruise                                 *
*                         person                                 *
*                         sailor                                 *
*                         yacht                                  *
*                                                                *
*                                                                *
*            for the SAG Tours demonstration system .            *
*                                                                *
* 95/05/18 07:30:00 (co) Software AG                             *
*                                                                *
*****************************************************************/

#include <stdio.h>

EXEC SQL
  BEGIN DECLARE SECTION;
  char uid[80];

EXEC SQL
  END DECLARE SECTION;

EXEC SQL
  INCLUDE SQLCA;

int  esq_language = 0  ;
int  message_len  = 300;
char err_message        [300];
char final_err_mess     [300];

main()
{
```

```
EXEC SQL
  WHENEVER SQLERROR CONTINUE;

strcpy ( uid , ”ESQ” , strlen (”ESQ”) ) ;

EXEC SQL
  CONNECT :uid ;

if ( SQLCODE == 0 )
   printf (”\nCONNECT performed successfully\n”);
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,’ ’,300);
   strncpy(final_err_mess,err_message,message_len);
   printf(”\nError: %s\n”,final_err_mess);
   }

/*****************************************************************
* CREATE DATABASE YACHT_DB                                       *
*****************************************************************/

EXEC SQL
  CREATE DATABASE yacht_db AS DATABASE NUMBER 240;

printf(”\nCreating database YACHT_DB as 240 ...\n”);

if ( SQLCODE == 0 )
   printf (”\nDatabase YACHT_DB created successfully\n”);
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,’ ’,300);
   strncpy(final_err_mess,err_message,message_len);
   printf(”\nError: %s\n”,final_err_mess);
   }
```

```
/*****************************************************************
* CREATE TABLE CONTRACT                                          *
*****************************************************************/

EXEC SQL
   CREATE TABLESPACE FOR TABLE contract
   (
   DATABASE=yacht_db,
   FILE=4,
   FILENAME="CONTRACT",
   ASSOPFAC=5,
   DATAPFAC=5,
   DSSIZE=10BLOCK,
   NISIZE=10BLOCK,
   UISIZE=10BLOCK,
   MAXISN=300,
   REUSE=(ISN,DS)
   );

if ( SQLCODE == 0 )
   printf ("\nTable space for CONTRACT created successfully\n");
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,' ',300);
   strncpy(final_err_mess,err_message,message_len);
   printf("\nError: %s\n",final_err_mess);
   }

printf("\nCreating table CONTRACT ...\n");

EXEC SQL
  CREATE TABLE contract
     ( contract_id          integer    index    ind_contract
                                       not null
                                       unique,
       price                numeric (13,3) not null,
       date_reservation     integer    ,
       date_booking         integer    ,
       date_cancellation    integer    ,
       date_deposit         integer    ,
       amount_deposit       numeric (13,3) ,
       date_payment         integer    ,
       amount_payment       numeric (13,3) ,
       id_customer          integer    not null,
       id_cruise            integer    not null );
```

```
if ( SQLCODE == 0 )
   printf (”\nTable CONTRACT created successfully\n”);
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,’ ’,300);
   strncpy(final_err_mess,err_message,message_len);
   printf(”\nError: %s\n”,final_err_mess);
   }

/*****************************************************************
* CREATE TABLE CRUISE                                            *
*****************************************************************/

EXEC SQL
   CREATE TABLESPACE FOR TABLE cruise
   (
   DATABASE=yacht_db,
   FILE=5,
   FILENAME=”CRUISE”,
   ASSOPFAC=5,
   DATAPFAC=5,
   DSSIZE=10BLOCK,
   NISIZE=10BLOCK,
   UISIZE=10BLOCK,
   MAXISN=300,
   REUSE=(ISN,DS)
   );

if ( SQLCODE == 0 )
   printf (”\nTable space for CRUISE created successfully\n”);
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,’ ’,300);
   strncpy(final_err_mess,err_message,message_len);
   printf(”\nError: %s\n”,final_err_mess);
   }

printf(”\nCreating table CRUISE ...\n”);
```

```
EXEC SQL
  CREATE TABLE cruise
       ( cruise_id          integer    index    ind_cruise
                                       not null
                                       unique,
         start_date         integer    ,
         start_time         integer    ,
         end_date           integer    ,
         end_time           integer    ,
         start_harbor       char(20)   ,
         destination_harbor char(20)   ,
         cruise_price       numeric (13,3) not null,
         bunk_number        integer    ,
         bunks_free         integer    not null,
         id_yacht           integer    not null,
         id_skipper         integer    not null,
         id_predecessor     integer    ,
         id_successor       integer    );

if ( SQLCODE == 0 )
   printf (”\nTable CRUISE created successfully\n”);
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,’ ’,300);
   strncpy(final_err_mess,err_message,message_len);
   printf(”\nError: %s\n”,final_err_mess);
   }
```

```
/*****************************************************************
* CREATE TABLE PERSON                                            *
*****************************************************************/

EXEC SQL
   CREATE TABLESPACE FOR TABLE person
   (
   DATABASE=yacht_db,
   FILE=6,
   FILENAME=”PERSON”,
   ASSOPFAC=5,
   DATAPFAC=5,
   DSSIZE=10BLOCK,
   NISIZE=10BLOCK,
   UISIZE=10BLOCK,
   MAXISN=300,
   REUSE=(ISN,DS)
   );

if ( SQLCODE == 0 )
   printf (”\nTable space for PERSON created successfully\n”);
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,’ ’,300);
   strncpy(final_err_mess,err_message,message_len);
   printf(”\nError: %s\n”,final_err_mess);
   }

printf(”\nCreating table PERSON ...\n”);
```

```
EXEC SQL
  CREATE TABLE person
     ( person_id            integer    index    ind_person
                                       not null
                                       unique,
       birth_date           integer    ,
       sex                  char(1)    ,
       surname              char(20)   ,
       first_name_1         char(20)   ,
       first_name_2         char(20)   ,
       title                char(20)   ,
       form_of_address      char(8)    ,
       address_addition_1   char(20)   ,
       address_addition_2   char(20)   ,
       street_number        char(20)   ,
       country              char(3)    ,
       zip_code             char(10)   ,
       city                 char(20)   ,
       area_code_priv       char(6)    ,
       phone_number_priv    char(15)   ,
       area_code_office     char(6)    ,
       phone_number_office  char(15)   );

if ( SQLCODE == 0 )
   printf (”\nTable PERSON created successfully\n”);
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,’ ’,300);
   strncpy(final_err_mess,err_message,message_len);
   printf(”\nError: %s\n”,final_err_mess);
   }
```

```
/*****************************************************************
* CREATE TABLE SAILOR                                            *
*****************************************************************/


EXEC SQL
   CREATE TABLESPACE FOR TABLE sailor
   (
   DATABASE=yacht_db,
   FILE=7,
   FILENAME=”SAILOR”,
   ASSOPFAC=5,
   DATAPFAC=5,
   DSSIZE=10BLOCK,
   NISIZE=10BLOCK,
   UISIZE=10BLOCK,
   MAXISN=500,
   REUSE=(ISN,DS)
   );


if ( SQLCODE == 0 )
   printf (”\nTable space for SAILOR created successfully\n”);
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,’ ’,300);
   strncpy(final_err_mess,err_message,message_len);
   printf(”\nError: %s\n”,final_err_mess);
   }


printf(”\nCreating table SAILOR ...\n”);


EXEC SQL
  CREATE TABLE sailor
     ( sailor_id            integer    index    ind_sailor
                                       not null
                                       unique,
       age                  integer    ,
       sailor_name          char(30)   ,
       experience           char(1)    ,
       l_1                  char(3)    ,
       l_2                  char(3)    ,
       l_3                  char(3)    ,
       id_contract          integer    not null,
       id_cruise            integer    not null );
```

```
if ( SQLCODE == 0 )
   printf ("\nTable SAILOR created successfully\n");
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,' ',300);
   strncpy(final_err_mess,err_message,message_len);
   printf("\nError: %s\n",final_err_mess);
   }

/*****************************************************************
* CREATE TABLE YACHT                                             *
*****************************************************************/

EXEC SQL
   CREATE TABLESPACE FOR TABLE yacht
   (
   DATABASE=yacht_db,
   FILE=8,
   FILENAME="YACHT",
   ASSOPFAC=5,
   DATAPFAC=5,
   DSSIZE=10BLOCK,
   NISIZE=10BLOCK,
   UISIZE=10BLOCK,
   MAXISN=100000,
   REUSE=(ISN,DS)
   );

if ( SQLCODE == 0 )
   printf ("\nTable space for YACHT created successfully\n");
else
   {
   message_len = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,' ',300);
   strncpy(final_err_mess,err_message,message_len);
   printf("\nError: %s\n",final_err_mess);
   }

printf("\nCreating table YACHT ...\n");
```

```
EXEC SQL
  CREATE TABLE yacht
     ( yacht_id             integer    index    ind_yacht
                                       not null
                                       unique,
       yacht_name           char(30)   ,
       id_owner             integer    not null,
       yacht_type           char(30)   ,
       yacht_length         numeric (5,2) ,
       yacht_width          numeric (5,2) ,
       yacht_draft          numeric (5,2) ,
       sail_surface         integer    ,
       motor                integer    ,
       head_room            numeric (5,2) ,
       bunks                integer    not null );

if ( SQLCODE == 0 )
   printf (”\nTable YACHT created successfully\n”);
else
   {
   esq_language = 0;
   message_len  = 300;
   esqerr(&sqlca,err_message,&message_len,&esq_language);
   memset(final_err_mess,’ ’,300);
   strncpy(final_err_mess,err_message,message_len);
   printf(”\nError: %s\n”,final_err_mess);
   }

EXEC SQL
  DISCONNECT;

printf (”\n\nEnd of the ESQ example program.\n”);

return;

}
```

Sample COBOL Program for the Creation of the Following SQL Tables

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  YACHTCR.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  xyz.
 OBJECT-COMPUTER.  xyz.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
***************************************************************
*                                                             *
*            Name -- yacht_cr.pco                             *
*                                                             *
*     Description -- Program to create the tables :           *
*                                                             *
*                    contract                                 *
*                    cruise                                   *
*                    person                                   *
*                    sailor                                   *
*                    yacht                                    *
*                                                             *
*           for the SAG Tours demonstration system .          *
*                                                             *
* 95/05/07 07:30:00 (co) Software AG                          *
*                                                             *
***************************************************************

     EXEC SQL BEGIN DECLARE SECTION END-EXEC
 01  UID            PIC X(18).
     EXEC SQL END DECLARE SECTION END-EXEC
     EXEC SQL INCLUDE SQLCA END-EXEC

 01  MESSAGE-LEN     COMP PIC S9(5) VALUE 300.
 01  ERR-MESSAGE     PIC X(300).
 01  FINAL-ERR-MESS  PIC X(300).
 01  ESQ-LANGUAGE    COMP PIC S9(5) VALUE 0.

     EXEC SQL WHENEVER SQLERROR CONTINUE
     END-EXEC

 PROCEDURE DIVISION.
 PR SECTION.
```

```
     MOVE ”ESQ” TO UID.
     DISPLAY ” ”.
     DISPLAY ”Connecting to ESQ as ” UID NO ADVANCING.

     EXEC SQL CONNECT :UID END–EXEC

     IF SQLCODE = 0
        DISPLAY ” CONNECTED”
     ELSE
        PERFORM FAILED–MESSAGE.

****************************************************************
* CREATE DATABASE YACHT_DB                                     *
****************************************************************

     DISPLAY ” ”.
     DISPLAY ”Creating database YACHT_DB as 240 ... ”
        NO ADVANCING.

     EXEC SQL
        CREATE DATABASE yacht_db AS DATABASE NUMBER 240
     END–EXEC

     IF SQLCODE = 0
        DISPLAY ”CREATED”
     ELSE
        PERFORM FAILED–MESSAGE.
```

```
***************************************************************
* CREATE TABLE CONTRACT                                       *
***************************************************************

     DISPLAY ” ”.
     DISPLAY ”Creating table space for CONTRACT ... ”
        NO ADVANCING.

     EXEC SQL
        CREATE TABLESPACE FOR TABLE contract
        (
            DATABASE=yacht_db,
            FILE=4,
            FILENAME=”CONTRACT”,
            ASSOPFAC=5,
            DATAPFAC=5,
            DSSIZE=10BLOCK,
            NISIZE=10BLOCK,
            UISIZE=10BLOCK,
            MAXISN=300,
            REUSE=(ISN,DS)
        )
     END–EXEC

     IF SQLCODE = 0
        DISPLAY ”CREATED”
     ELSE
        PERFORM FAILED–MESSAGE.

     DISPLAY ”Creating table           CONTRACT ... ”
        NO ADVANCING.

     EXEC SQL
        CREATE TABLE contract
           ( contract_id        integer    index    ind_contract
                                           not null
                                           unique,
             price              numeric (13,3) not null,
             date_reservation   integer    ,
             date_booking       integer    ,
             date_cancellation  integer    ,
             date_deposit       integer    ,
             amount_deposit     numeric (13,3) ,
             date_payment       integer    ,
             amount_payment     numeric (13,3) ,
             id_customer        integer    not null,
             id_cruise          integer    not null )
     END–EXEC
```

```
          IF SQLCODE = 0
             DISPLAY ”CREATED”
          ELSE
             PERFORM FAILED–MESSAGE.

     ***************************************************************
     * CREATE TABLE CRUISE                                         *
     ***************************************************************

          DISPLAY ”Creating table space for CRUISE ..... ”
             NO ADVANCING.

          EXEC SQL
             CREATE TABLESPACE FOR TABLE cruise
             (
                 DATABASE=yacht_db,
                 FILE=5,
                 FILENAME=”CRUISE”,
                 ASSOPFAC=5,
                 DATAPFAC=5,
                 DSSIZE=10BLOCK,
                 NISIZE=10BLOCK,
                 UISIZE=10BLOCK,
                 MAXISN=300,
                 REUSE=(ISN,DS)
             )
          END–EXEC

          IF SQLCODE = 0
             DISPLAY ”CREATED”
          ELSE
             PERFORM FAILED–MESSAGE.

          DISPLAY ”Creating table           CRUISE ..... ”
             NO ADVANCING.
     EXEC SQL
        CREATE TABLE cruise
           ( cruise_id          integer    index    ind_cruise
                                           not null
                                           unique,
             start_date         integer    ,
             start_time         integer    ,
             end_date           integer    ,
             end_time           integer    ,
             start_harbor       char(20)   ,
             destination_harbor char(20)   ,
             cruise_price       numeric (13,3) not null,
             bunk_number        integer    ,
             bunks_free         integer    not null,
             id_yacht           integer    not null,
             id_skipper         integer    not null,
             id_predecessor     integer    ,
             id_successor       integer    )
     END–EXEC

     IF SQLCODE = 0
        DISPLAY ”CREATED”
     ELSE
        PERFORM FAILED–MESSAGE.

****************************************************************
* CREATE TABLE PERSON                                          *
****************************************************************

     DISPLAY ”Creating table space for PERSON ..... ”
        NO ADVANCING.

     EXEC SQL
        CREATE TABLESPACE FOR TABLE person
        (
            DATABASE=yacht_db,
            FILE=6,
            FILENAME=”PERSON”,
            ASSOPFAC=5,
            DATAPFAC=5,
            DSSIZE=10BLOCK,
            NISIZE=10BLOCK,
            UISIZE=10BLOCK,
            MAXISN=300,
            REUSE=(ISN,DS)
        )
     END–EXEC
```

```
IF SQLCODE = 0
   DISPLAY ”CREATED”
ELSE
   PERFORM FAILED–MESSAGE.

DISPLAY ”Creating table           PERSON ..... ”
   NO ADVANCING.

EXEC SQL
   CREATE TABLE person
      ( person_id            integer    index    ind_person
                                        not null
                                        unique,
        birth_date           integer    ,
        sex                  char(1)    ,
        surname              char(20)   ,
        first_name_1         char(20)   ,
        first_name_2         char(20)   ,
        title                char(20)   ,
        form_of_address      char(8)    ,
        address_addition_1   char(20)   ,
        address_addition_2   char(20)   ,
        street_number        char(20)   ,
        country              char(3)    ,
        zip_code             char(10)   ,
        city                 char(20)   ,
        area_code_priv       char(6)    ,
        phone_number_priv    char(15)   ,
        area_code_office     char(6)    ,
        phone_number_office  char(15)   )
END–EXEC

IF SQLCODE = 0
   DISPLAY ”CREATED”
ELSE
   PERFORM FAILED–MESSAGE.
```

```
****************************************************************
* CREATE TABLE SAILOR                                          *
****************************************************************


     DISPLAY ”Creating table space for SAILOR ..... ”
        NO ADVANCING.

     EXEC SQL
        CREATE TABLESPACE FOR TABLE sailor
        (
            DATABASE=yacht_db,
            FILE=7,
            FILENAME=”SAILOR”,
            ASSOPFAC=5,
            DATAPFAC=5,
            DSSIZE=10BLOCK,
            NISIZE=10BLOCK,
            UISIZE=10BLOCK,
            MAXISN=500,
            REUSE=(ISN,DS)
        )
     END–EXEC


     IF SQLCODE = 0
        DISPLAY ”CREATED”
     ELSE
        PERFORM FAILED–MESSAGE.


     DISPLAY ”Creating table           SAILOR ..... ”
        NO ADVANCING.


     EXEC SQL
        CREATE TABLE sailor
           ( sailor_id            integer    index    ind_sailor
                                             not null
                                             unique,
             age                  integer    ,
             sailor_name          char(30)   ,
             experience           char(1)    ,
             l_1                  char(3)    ,
             l_2                  char(3)    ,
             l_3                  char(3)    ,
             id_contract          integer    not null,
             id_cruise            integer    not null )
     END–EXEC
     IF SQLCODE = 0
        DISPLAY ”CREATED”
     ELSE
        PERFORM FAILED–MESSAGE.

****************************************************************
* CREATE TABLE YACHT                                           *
****************************************************************

     DISPLAY ”Creating table space for YACHT ...... ”
        NO ADVANCING.

     EXEC SQL
        CREATE TABLESPACE FOR TABLE yacht
        (
            DATABASE=yacht_db,
            FILE=8,
            FILENAME=”YACHT”,
            ASSOPFAC=5,
            DATAPFAC=5,
            DSSIZE=10BLOCK,
            NISIZE=10BLOCK,
            UISIZE=10BLOCK,
            MAXISN=100000,
            REUSE=(ISN,DS)
        )
     END–EXEC

     IF SQLCODE = 0
        DISPLAY ”CREATED”
     ELSE
        PERFORM FAILED–MESSAGE.

     DISPLAY ”Creating table           YACHT ...... ”
        NO ADVANCING.
```

```
    EXEC SQL
       CREATE TABLE yacht
          ( yacht_id             integer    index    ind_yacht
                                            not null
                                            unique,
            yacht_name           char(30)   ,
            id_owner             integer    not null,
            yacht_type           char(30)   ,
            yacht_length         numeric (5,2) ,
            yacht_width          numeric (5,2) ,
            yacht_draft          numeric (5,2) ,
            sail_surface         integer    ,
            motor                integer    ,
            head_room            numeric (5,2) ,
            bunks                integer    not null )
    END–EXEC

    IF SQLCODE = 0
       DISPLAY ”CREATED”
    ELSE
       PERFORM FAILED–MESSAGE.

    DISPLAY ” ”.
    DISPLAY ”Disconnecting from ESQ”.

    EXEC SQL
       DISCONNECT
    END–EXEC

    DISPLAY ”End of the ESQ example program.”.
    STOP RUN.

FAILED–MESSAGE.
       DISPLAY ”FAILED with SQLCODE ” SQLCODE.
       MOVE SPACES TO ERR–MESSAGE.
    MOVE 0 TO ESQ–LANGUAGE.
       CALL ”esqerr” USING BY REFERENCE SQLCA
                           BY REFERENCE ERR–MESSAGE
                           BY REFERENCE MESSAGE–LEN
                           BY REFERENCE ESQ–LANGUAGE.
       DISPLAY ”ERROR : ” ERR–MESSAGE.
FAILED–MESSAGE–END.
```

Sample PL/I Program for the Creation of the Following SQL Tables

```
/*****************************************************************
*                                                                *
*               Name –– yacht_cr.pli                             *
*                                                                *
*        Description –– Program to create the tables :           *
*                                                                *
*                        contract                                *
*                        cruise                                  *
*                        person                                  *
*                        sailor                                  *
*                        yacht                                   *
*                                                                *
*                                                                *
*           for the SAG Tours demonstration system .             *
*                                                                *
* 95/05/07 07:30:00 (co) Software AG                             *
*                                                                *
*****************************************************************/

DCL ESQERR ENTRY (ANY,ANY,ANY,ANY) EXTERNAL (’esqerr’);

YACHT_CR: PROCEDURE OPTIONS (MAIN);

EXEC SQL
   BEGIN DECLARE SECTION;
   DCL UID              CHARACTER (18) VARYING;

EXEC SQL
   END DECLARE SECTION;

EXEC SQL
   INCLUDE SQLCA;

DCL MESSAGE_LEN    FIXED BIN (16) INIT (300);
DCL ERR_MESSAGE    CHARACTER (300);
DCL FINAL_ERR_MESS CHARACTER (300);
DCL ESQ_LANGUAGE   FIXED BIN (16) INIT (0);

EXEC SQL
   WHENEVER SQLERROR CONTINUE;
```

```
ON ERROR BEGIN;
   PUT EDIT ('ERROR CODE :' ,ONCODE()) (A,F(6)) SKIP;
   PUT LIST ('Program aborted!') SKIP;
   PUT SKIP (1);
   STOP;
END;

UID = 'ESQ';

PUT EDIT ( 'Connecting to ESQ AS ',UID,' ... ' ) (A,A,A) SKIP;

EXEC SQL
   CONNECT :UID;

IF SQLCODE = 0 THEN
DO;
   PUT EDIT ('CONNECTED') (A);
END;
ELSE
DO;
   PUT EDIT ('FAILED with SQLCODE ', SQLCODE) (A,F(5));
   CALL ESQERR
(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(' ',300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( 'ERROR: ' ) ;
   PUT LIST ( FINAL_ERR_MESS ) SKIP ;
END ;

/*****************************************************************
* CREATE DATABASE YACHT_DB                                       *
*****************************************************************/

PUT LIST ('Creating database YACHT_DB as 240 ...') SKIP(2);

EXEC SQL
   CREATE DATABASE yht_db AS DATABASE NUMBER 240;
```

```
IF SQLCODE = 0 THEN
DO;
   PUT EDIT (’CREATED’) (A) ;
END;
ELSE
DO;
   PUT EDIT (’FAILED with SQLCODE ’, SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(’ ’,300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( ’ERROR: ’ ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;

PUT SKIP (1);

/*****************************************************************
* CREATE TABLE CONTRACT                                          *
*****************************************************************/

PUT LIST (’Creating table space for CONTRACT ...’) SKIP;

EXEC SQL
   CREATE TABLESPACE FOR TABLE contract
   (
      DATABASE=yht_db,
      FILE=29,
      FILENAME=”CONTRACT”,
      ASSOPFAC=5,
      DATAPFAC=5,
      DSSIZE=10BLOCK,
      NISIZE=10BLOCK,
      UISIZE=10BLOCK,
      MAXISN=300,
      REUSE=(ISN,DS)
   );
```

```
IF SQLCODE = 0 THEN
DO;
   PUT EDIT ('CREATED') (A) ;
END;
ELSE
DO;
   PUT EDIT ('FAILED with SQLCODE ', SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(' ',300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( 'ERROR: ' ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;


PUT LIST ('Creating table           CONTRACT ...') SKIP;


EXEC SQL
   CREATE TABLE contract
      ( contract_id          integer    index    ind_contract
                                        not null
                                        unique,
        price                numeric (13,3) not null,
        date_reservation     integer    ,
        date_booking         integer    ,
        date_cancellation    integer    ,
        date_deposit         integer    ,
        amount_deposit       numeric (13,3) ,
        date_payment         integer    ,
        amount_payment       numeric (13,3) ,
        id_customer          integer    not null,
        id_cruise            integer    not null );


IF SQLCODE = 0 THEN
DO;
   PUT EDIT ('CREATED') (A) ;
END;
ELSE
DO;
   PUT EDIT ('FAILED with SQLCODE ', SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(' ',300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( 'ERROR: ' ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;
```

```
/*****************************************************************
* CREATE TABLE CRUISE                                            *
*****************************************************************/

PUT LIST (’Creating table space for CRUISE .....’) SKIP;

EXEC SQL
   CREATE TABLESPACE FOR TABLE cruise
   (
      DATABASE=yht_db,
      FILE=30,
      FILENAME=”CRUISE”,
      ASSOPFAC=5,
      DATAPFAC=5,
      DSSIZE=10BLOCK,
      NISIZE=10BLOCK,
      UISIZE=10BLOCK,
      MAXISN=300,
      REUSE=(ISN,DS)
   );

IF SQLCODE = 0 THEN
DO;
   PUT EDIT (’CREATED’) (A) ;
END;
ELSE
DO;
   PUT EDIT (’FAILED with SQLCODE ’, SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(’ ’,300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( ’ERROR: ’ ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;

PUT LIST (’Creating table           CRUISE .....’) SKIP;
```

```
EXEC SQL
   CREATE TABLE cruise
      ( cruise_id          integer    index    ind_cruise
                                      not null
                                      unique,
        start_date         integer    ,
        start_time         integer    ,
        end_date           integer    ,
        end_time           integer    ,
        start_harbor       char(20)   ,
        destination_harbor char(20)   ,
        cruise_price       numeric (13,3) not null,
        bunk_number        integer    ,
        bunks_free         integer    not null,
        id_yacht           integer    not null,
        id_skipper         integer    not null,
        id_predecessor     integer    ,
        id_successor       integer    );

IF SQLCODE = 0 THEN
DO;
   PUT EDIT (’CREATED’) (A) ;
END;
ELSE
DO;
   PUT EDIT (’FAILED with SQLCODE ’, SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(’ ’,300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( ’ERROR: ’ ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;
```

```
/*****************************************************************
* CREATE TABLE PERSON                                            *
*****************************************************************/

PUT LIST (’Creating table space for PERSON .....’) SKIP;

EXEC SQL
   CREATE TABLESPACE FOR TABLE person
   (
      DATABASE=yht_db,
      FILE=31,
      FILENAME=”PERSON”,
      ASSOPFAC=5,
      DATAPFAC=5,
      DSSIZE=10BLOCK,
      NISIZE=10BLOCK,
      UISIZE=10BLOCK,
      MAXISN=300,
      REUSE=(ISN,DS)
   );

IF SQLCODE = 0 THEN
DO;
   PUT EDIT (’CREATED’) (A) ;
END;
ELSE
DO;
   PUT EDIT (’FAILED with SQLCODE ’, SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(’ ’,300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( ’ERROR: ’ ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;

PUT LIST (’Creating table           PERSON .....’) SKIP;
```

```
EXEC SQL
   CREATE TABLE person
      ( person_id            integer    index    ind_person
                                        not null
                                        unique,
        birth_date           integer    ,
        sex                  char(1)    ,
        surname              char(20)   ,
        first_name_1         char(20)   ,
        first_name_2         char(20)   ,
        title                char(20)   ,
        form_of_address      char(8)    ,
        address_addition_1   char(20)   ,
        address_addition_2   char(20)   ,
        street_number        char(20)   ,
        country              char(3)    ,
        zip_code             char(10)   ,
        city                 char(20)   ,
        area_code_priv       char(6)    ,
        phone_number_priv    char(15)   ,
        area_code_office     char(6)    ,
        phone_number_office  char(15)   );

IF SQLCODE = 0 THEN
DO;
   PUT EDIT (’CREATED’) (A) ;
END;
ELSE
DO;
   PUT EDIT (’FAILED with SQLCODE ’, SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(’ ’,300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( ’ERROR: ’ ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;
```

```
/*****************************************************************
* CREATE TABLE SAILOR                                            *
*****************************************************************/

PUT LIST (’Creating table space for SAILOR .....’) SKIP;

EXEC SQL
   CREATE TABLESPACE FOR TABLE sailor
   (
      DATABASE=yht_db,
      FILE=32,
      FILENAME=”SAILOR”,
      ASSOPFAC=5,
      DATAPFAC=5,
      DSSIZE=10BLOCK,
      NISIZE=10BLOCK,
      UISIZE=10BLOCK,
      MAXISN=500,
      REUSE=(ISN,DS)
   );

IF SQLCODE = 0 THEN
DO;
   PUT EDIT (’CREATED’) (A) ;
END;
ELSE
DO;
   PUT EDIT (’FAILED with SQLCODE ’, SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(’ ’,300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( ’ERROR: ’ ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;

PUT LIST (’Creating table           SAILOR .....’) SKIP;
```

```
EXEC SQL
   CREATE TABLE sailor
      ( sailor_id            integer    index    ind_sailor
                                        not null
                                        unique,
        age                  integer    ,
        sailor_name          char(30)   ,
        experience           char(1)    ,
        l_1                  char(3)    ,
        l_2                  char(3)    ,
        l_3                  char(3)    ,
        id_contract          integer    not null,
        id_cruise            integer    not null );

IF SQLCODE = 0 THEN
DO;
   PUT EDIT ('CREATED') (A) ;
END;
ELSE
DO;
   PUT EDIT ('FAILED with SQLCODE ', SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(' ',300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( 'ERROR: ' ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;
```

```
/*****************************************************************
* CREATE TABLE YACHT                                              *
*****************************************************************/

PUT LIST (’Creating table space for YACHT ......’) SKIP;

EXEC SQL
   CREATE TABLESPACE FOR TABLE yacht
   (
      DATABASE=yht_db,
      FILE=33,
      FILENAME=”YACHT”,
      ASSOPFAC=5,
      DATAPFAC=5,
      DSSIZE=10BLOCK,
      NISIZE=10BLOCK,
      UISIZE=10BLOCK,
      MAXISN=100000,
      REUSE=(ISN,DS)
   );

IF SQLCODE = 0 THEN
DO;
   PUT EDIT (’CREATED’) (A) ;
END;
ELSE
DO;
   PUT EDIT (’FAILED with SQLCODE ’, SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(’ ’,300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( ’ERROR: ’ ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;

PUT LIST (’Creating table           YACHT ......’) SKIP;
```

```
EXEC SQL
   CREATE TABLE yacht
      ( yacht_id             integer    index    ind_yacht
                                        not null
                                        unique,
        yacht_name           char(30)   ,
        id_owner             integer    not null,
        yacht_type           char(30)   ,
        yacht_length         numeric (5,2) ,
        yacht_width          numeric (5,2) ,
        yacht_draft          numeric (5,2) ,
        sail_surface         integer    ,
        motor                integer    ,
        head_room            numeric (5,2) ,
        bunks                integer    not null );

IF SQLCODE = 0 THEN
DO;
   PUT EDIT ('CREATED') (A) ;
END;
ELSE
DO;
   PUT EDIT ('FAILED with SQLCODE ', SQLCODE) (A,F(5));
   CALL ESQERR(SQLCA,ERR_MESSAGE,MESSAGE_LEN,ESQ_LANGUAGE);
   FINAL_ERR_MESS = COPY(' ',300);
   FINAL_ERR_MESS = ERR_MESSAGE ;
   PUT LIST ( 'ERROR: ' ) SKIP;
   PUT EDIT ( FINAL_ERR_MESS ) (A);
END ;

PUT LIST ('Disconnecting from ESQ' ) SKIP(2);

EXEC SQL
   DISCONNECT;

PUT LIST ('End of the ESQ example program.') SKIP;
PUT SKIP (1);

END YACHT_CR;
```

APPENDIX C — THE ADABAS SQL SERVER CATALOG STRUCTURE

In this appendix, the structure of the Adabas SQL Server catalog (in the following called catalog) is explained and displayed in various tables.

The Schemas

The Adabas SQL Server catalog consists of three schemas: The DEFINITION_SCHEMA, the DBA_SCHEMA, and the INFORMATION_SCHEMA.

The schema DEFINITION_SCHEMA contains the base tables used internally by Adabas SQL Server, whereas the two others offer views on this information which are relevant to administration. Their structures (view and column names) are absolutely identical.

Warning:
Data manipulation performed on the DEFINITION_SCHEMA may result in catalog inconsistencies and is therefore not recommended.

If the security mode is on, the schema DBA_SCHEMA can be accessed only by the user DBA, whereas the schema INFORMATION_SCHEMA is always public. But the INFORMATION_SCHEMA restricts the information to those objects, on which the current user has privileges.

The user DBA is treated like any other user. For example, a SELECT statement against any of the INFORMATION_SCHEMA tables will return results for those objects only which have been GRANTed access to. To see all information, the DBA should use the DBA_SCHEMA tables (which are only accessible to the DBA).

The View Description Tables

The tables on the following pages describe the views of the DBA_SCHEMA and the INFORMATION_SCHEMA. For each view, the columns are listed with their names, data types, indications, if NULL is possible, and short descriptions of the semantics. The data types are shown here as symbolic identifiers to describe a specific role:

yes_no	stands for CHAR(1) with a content of 'Y' or 'N'.
identifier	stands for CHAR(32).
enumeration	stands for CHAR(40) with possible values listed in the column "Description".
long_alpha	stands for CHAR(>253).
timestamp	stands for BINARY(8).

Note, that all identifiers in the catalog are represented in upper case characters.

Note:
***y** in the column "N" means that the column may contain NULL values,*
***n** in the column "N" means that the column must not contain NULL values.*

Note:
***—>** (arrow) in the column "Description" references an explanation in the glossary of the **Adabas SQL Server Reference Manual**. For some values, "**n/a**" (not applicable) indicates that they do not appear in the current product version, but they are listed for reason of standard compatibility and/or for use in future versions.*

SERVER_INFO View:

Gives some information about the server.

Column	Data Type	N	Description
SERVER_NAME	identifier	n	—> Server is identical with the catalog name.
NODE_NAME	identifier	n	—> Server
ESQ_VERSION	identifier	n	—> Server
SECURITY_ON	yes_no	n	’Y’ if server is in security mode. ’N’ if not. —> Security Mode

Role:	
SERVER_NAME	identifies the server.

INFORMATION_SCHEMA_CATALOG_NAME View:

Gives the name of the catalog.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
CATALOG_NAME	identifier	n	—> Catalog is identical with the server name.

Role:	
CATALOG_NAME	identifies the catalog.

SCHEMATA View:

Describes the schemas of the catalog.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
SCHEMA_NAME	identifier	n	—> Schema
SCHEMA_OWNER	identifier	n	—> User —> Privileges

Role:

SCHEMA_NAME	identifies the selected schema.

CLUSTERS View:

Describes the clusters of the catalog.

Column	Data Type	N	Description
CLUSTER_SCHEMA	identifier	n	—> Schema
CLUSTER_NAME	identifier	n	—> Cluster
DATABASE_NAME	identifier	n	—> Database
FILE_NR	integer	n	—> Adabas File

Roles:

CLUSTER_SCHEMA, CLUSTER_NAME	identifies the selected cluster.
DATABASE_NAME, FILE_NR	identifies the realizing Adabas file.

TABLES View:

Lists the tables defined in the catalog.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
TABLE_TYPE	enumeration	n	—> Table ’BASE TABLE’ ’VIEW’

Roles:

TABLE_SCHEMA, TABLE_NAME	identifies the selected table.

BASE_TABLES View:

Lists the base tables defined in the catalog.

Column	Data Type	N	Description
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
DATABASE_NAME	identifier	n	—> Database
FILE_NR	integer	n	—> Adabas File
TABLE_LEVEL	smallint	n	possible values: –1, 0, 1, 2 —> Table Level

Roles:

TABLE_SCHEMA, TABLE_NAME	identifies the selected base table.
DATABASE_NAME, FILE_NR	identifies the realizing Adabas file.

VIEWS View:

Lists the views defined in the catalog.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
VIEW_DEFINITION	long_alpha	n	The SQL source string of the view definition. —> View
CHECK_OPTION	enumeration	n	’NONE’ ’CASCADED’ n/a ’LOCAL’ n/a
IS_UPDATABLE	yes_no	n	’Y’ if view is updatable, ’N’ if not —> View

Roles:

TABLE_SCHEMA, TABLE_NAME	identifies the selected view.

COLUMNS View:

Describes the columns of the tables defined in the catalog.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
COLUMN_NAME	identifier	n	—> Column
ORDINAL_POSITION	integer	n	Ordinal position inside the containing table. —> Column
COLUMN_DEFAULT	long_alpha	y	Default option in character representation. —> Default Clause
IS_NULLABLE	yes_no	n	’Y’ if NULL is possible, ’N’ if certainly not possible. —> Constraint
DATA_TYPE	enumeration	n	’CHAR’ ’DECIMAL’ ’NUMERIC’ ’INT’ ’SMALLINT’ ’FLOAT’ ’REAL’ ’DOUBLE’ ’BINARY’ —> Data Type
CHARACTER_MAXI-MUM_LENGTH	integer	n	Physical length of data in bytes. —> Data Type
NUMERIC_PRECISION	integer	n	Physical length of data in bytes. —> Data Type
NUMERIC_SCALE	integer	y	Binary precision for approximate numeric data types. —> Data Type

Column	Data Type	N	Description
COLUMN_LEVEL	smallint	n	possible values: –1, 0, 1, 2 —> Column Level
COLUMN_TYPE	enumeration	n	’ORDINARY’ ’SEQ_NR’ ’COMPUTED’ n/a ’SEARCH_ONLY’ n/a ’CONSTANT’ ’SIMULATED_LONG’ —> Column

Roles:

TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME	identifies the selected column.
TABLE_SCHEMA, TABLE_NAME, ORDINAL_POS	is unique in this view.

TABLE_PRIVILEGES View:

Describes the privileges on the tables defined in the catalog.

This view is part of the ANSI SQL2 standard Information Schema.

Column	**Data Type**	**N**	**Description**
GRANTOR	identifier	n	—> Privileges —> User
GRANTEE	identifier	n	—> Privileges —> User
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
PRIVILEGE_TYPE	enumeration	n	’SELECT’ ’INSERT’ ’DELETE’ ’UPDATE’ ’REFERENCES’ n/a —> Privileges
IS_GRANTABLE	yes_no	n	’Y’ if privilege is grantable. ’N’ if not. —> Privileges

Roles:

GRANTEE, TABLE_SCHEMA, TABLE_NAME, PRIVILEGE_TYPE	identifies the selected table privilege.
TABLE_SCHEMA, TABLE_NAME	identifies the accessible table.

COLUMN_PRIVILEGES View:

Describes the privileges on the columns defined in the catalog.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
GRANTOR	identifier	n	—> Privileges —> User
GRANTEE	identifier	n	—> Privileges —> User
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
COLUMN_NAME	identifier	n	—> Column
PRIVILEGE_TYPE	enumeration	n	’SELECT’ n/a ’INSERT’ n/a ’UPDATE’ ’REFERENCES’ n/a —> Privileges
IS_GRANTABLE	yes_no	n	’Y’ if privilege is grantable. ’N’ if not. —> Privileges

Roles:

GRANTEE, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, PRIVILEGE_TYPE	identifies the selected column privilege.
TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME	identifies the accessible column.

TABLE_CONSTRAINTS View:

Describes the table constraints.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
CONSTRAINT_SCHE-MA	identifier	n	—> Constraint —> Schema
CONSTRAINT_NAME	identifier	n	—> Constraint
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
CONSTRAINT_TYPE	enumeration	n	'NOT NULL' 'UNIQUE' 'PRIMARY KEY' 'FOREIGN KEY' 'CHECK' n/a —> Constraint
IS_DEFERRABLE	yes_no	n	'Y' if constraint is deferrable n/a 'N' if not. —> Constraint
INITIALLY_DEFERRED	yes_no	n	'Y' if constraint is initially deferred n/a 'N' if not. —> Constraint
IMPLEMENTING_IN-DEX	identifier	y	—> Constraint —> Index

Roles:

CONSTRAINT_SCHEMA, CONSTRAINT_NAME	identifies the table constraint.
TABLE_SCHEMA, TABLE_NAME	identifies the containing table.

TABLE_INDEXES View:

Describes the indices of the tables and gives the type of the index.

Column	Data Type	N	Description
INDEX_SCHEMA	identifier	n	—> Index —> Schema
INDEX_NAME	identifier	n	—> Index
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
INDEX_TYPE	enumeration	n	’ORDINARY’ ’PHONETIC’ n/a ’VIRTUAL’ n/a —> Index
SHORT_NAME	char(2)	n	Internal Adabas identification of the index. —> SHORTNAME
IS_MULTIPLE	yes_no	n	—> Index
IS_UQINDEX	yes_no	n	—> Index
INDEX_DATA_TYPE	enumeration	n	’CHAR’ ’DECIMAL’ ’NUMERIC’ ’INT’ ’SMALLINT’ ’FLOAT’ ’REAL’ ’DOUBLE’ ’BINARY’ —> Data Type —> Index

Roles:

INDEX_SCHEMA, INDEX_NAME	identifies the index.
TABLE_SCHEMA, TABLE_NAME	identifies the containing table.

ALL_TABLE_INDEX_ELEMENTS View:

For every constraint of the type PRIMARY KEY or UNIQUE and every index, which is of type INDEX or UNIQUE, the used columns and their ordinal position within this order are listed.

Strictly speaking:

A PRIMARY KEY constraint implies an index and leads to two rows in this view for each column, one with CONSTRAINT_TYPE ’PRIMARY KEY’ and one with CONSTRAINT_TYPE ’INDEX’ or ’UNIQUE’.

A ’UNIQUE’ constraint implies an index and leads to two rows in this view for each column, one with CONSTRAINT_TYPE ’UNIQUE’ and one with CONSTRAINT_TYPE ’INDEX’ or ’UNIQUE’.

A UNIQUE INDEX implies a constraint and leads to two rows in this view for each column, one with CONSTRAINT_TYPE ’UNIQUE’ and one with CONSTRAINT_TYPE ’INDEX’ or ’UNIQUE’.

An INDEX, which is not UNIQUE, leads to one row in this view for each column with CONSTRAINT_TYPE ’INDEX’.

Column	Data Type	N	Description
CONSTRAINT_SCHE-MA	identifier	n	—> Constraint —> Schema
CONSTRAINT_TABLE	identifier	n	—> Table
CONSTRAINT_NAME	identifier	n	—> Constraint
CONSTRAINT_TYPE	enumeration	n	’UNIQUE’ ’PRIMARY KEY’ ’INDEX’ —> Constraint —> Index —> UNIQUE CONSTRAINT
COLUMN_NAME	identifier	n	—> Column
ORDINAL_POSITION	integer	n	Ordinal position of the column element inside the containing index —> Constraint

Roles:

CONSTRAINT_SCHEMA, CONSTRAINT_NAME	identifies the using constraint.
CONSTRAINT_SCHEMA, CONSTRAINT_TABLE, COLUMN_NAME	identifies a used colum.
CONSTRAINT_SCHEMA, CONSTRAINT_NAME, ORDINAL_POSITION	is unique in this view.

KEY_COLUMN_USAGE View:

For every constraint (except NOT_NULL) and every index, the used columns and their ordinal positions within this order are listed.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
CONSTRAINT_SCHE-MA	identifier	n	—> Constraint —> Schema
CONSTRAINT_NAME	identifier	n	—> Constraint
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
COLUMN_NAME	identifier	n	—> Column
ORDINAL_POSITION	integer	n	Ordinal position of the column element inside the containing key —> Constraint

Roles:

CONSTRAINT_SCHEMA, CONSTRAINT_NAME, COLUMN_NAME	is unique in this view.
CONSTRAINT_SCHEMA, CONSTRAINT_NAME	identifies the using constraint.
TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME	identifies the used column.
CONSTRAINT_SCHEMA, CONSTRAINT_NAME, ORDINAL_POSITION	is unique in this view.

VIEW_TABLE_USAGE View:

Identifies the tables on which the catalog’s views are dependent.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
VIEW_SCHEMA	identifier	n	—> View —> Schema
VIEW_NAME	identifier	n	—> View
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table

Roles:

VIEW_SCHEMA, VIEW_NAME, TABLE_SCHEMA, TABLE_NAME	is unique in this view.
VIEW_SCHEMA, VIEW_NAME	identifies the using view.
TABLE_SCHEMA, TABLE_NAME	identifies the used table.

VIEW_COLUMN_USAGE View:

Identifies the columns on which the catalog’s views are dependent.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
VIEW_SCHEMA	identifier	n	—> View —> Schema
VIEW_NAME	identifier	n	—> View
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
COLUMN_NAME	identifier	n	—> Column

Roles:

VIEW_SCHEMA, VIEW_NAME, TABLE_SCHEMA TABLE_NAME, COLUMN_NAME	is unique in this view.
VIEW_SCHEMA, VIEW_NAME	identifies the using view.
TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME	identifies the used column.

CONSTRAINT_TABLE_USAGE View:

Identifies the tables that are referenced by referential constraints.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
CONSTRAINT_SCHE-MA	identifier	n	—> Constraint —> Schema
CONSTRAINT_NAME	identifier	n	—> Constraint

Roles:

CONSTRAINT_SCHEMA, CONSTRAINT_NAME	is unique in this view.
CONSTRAINT_SCHEMA, CONSTRAINT_NAME	identifies the referential constraint.
TABLE_SCHEMA, TABLE_NAME	identifies the referenced table.

CONSTRAINT_COLUMN_USAGE View:

Identifies the columns that are referenced by referential constraints.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
TABLE_SCHEMA	identifier	n	—> Schema
TABLE_NAME	identifier	n	—> Table
COLUMN_NAME	identifier	n	—> Column
CONSTRAINT_SCHE-MA	identifier	n	—> Constraint —> Schema
CONSTRAINT_NAME	identifier	n	—> Constraint

Roles:

CONSTRAINT_SCHEMA, CONSTRAINT_NAME, COLUMN_NAME	is unique in this view.
CONSTRAINT_SCHEMA, CONSTRAINT_NAME	identifies the referential constraint.
TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME	identifies the referenced column.

REFERENTIAL_CONSTRAINTS View:

Describes the referential constraints.

This view is part of the ANSI SQL2 standard Information Schema.

Column	Data Type	N	Description
CONSTRAINT_SCHE-MA	identifier	n	—> Constraint —> Schema
CONSTRAINT_NAME	identifier	n	—> Constraint
UNIQUE_CONSTRAINT_SCHEMA	identifier	n	—> Schema
UNIQUE_CONSTRAINT_NAME	identifier	n	—> Constraint
MATCH_OPTION	enumeration	n	’NONE’ n/a ’PARTIAL’ n/a ’FULL’ —> Referential Constraint.
UPDATE_RULE	enumeration	n	’NO ACTION’ n/a ’CASCADE’ ’SET NULL’ n/a ’SET DEFAULT n/a —> Referential Constraint.
DELETE_RULE	enumeration	n	’NO ACTION’ n/a ’CASCADE’ ’SET NULL’ n/a ’SET DEFAULT’ n/a —> Referential Constraint.
IS_CLUSTERING	yes_no	n	’Y’ if the referential constraint is impl. by a table cluster. ’N’ else. n/a —> Cluster

Roles:

CONSTRAINT_SCHEMA, CONSTRAINT_NAME	identifies the referential constraint.
UNIQUE_CONSTRAINT_SCHEMA, UNIQUE_CONSTRAINT_NAME	identifies the referred constraint.

DATABASES View:

Identifies the databases.

Column	Data Type	N	Description
DATABASE_NAME	identifier	n	Name of the database. —> Database
DATABASE_NUMBER	integer	n	Number of the containing database. —> Database

Roles:

DATABASE_NAME	identifies the database for ESQ.
DATABASE_NUMBER	identifies the database for Adabas.

TABLESPACES View:

Describes the tablespaces.

Column	Data Type	N	Description
SCHEMA_NAME	identifier	n	—> Schema
TABLESPACE_NAME	identifier	n	—> Tablespace
DATABASE_NAME	identifier	y	—> Database
FILE_NR	integer	y	—> Adabas File
FILE_NAME	identifier	y	—> Adabas File
TS_ACRABN	integer	n	—> Tablespace
TS_ASSOPFAC	integer	n	—> Tablespace
TS_CONTIGUOUS_AC	yes_no	n	—> Tablespace
TS_CONTIGUOUS_DS	yes_no	n	—> Tablespace
TS_CONTIGUOUS_NI	yes_no	n	—> Tablespace
TS_CONTIGUOUS_UI	yes_no	n	—> Tablespace
TS_DATAPFAC	integer	n	—> Tablespace
TS_DSDEV	integer	n	—> Tablespace
TS_DSRABN	integer	n	—> Tablespace
TS_DSREUSE	yes_no	n	—> Tablespace
TS_DSSIZE	integer	n	—> Tablespace
TS_DSSIZE_UNIT	enumeration	n	’BLOCK’ ’MEGABYTE’ —> Tablespace
TS_ISNREUSE	yes_no	n	—> Tablespace
TS_ISNSIZE	integer	n	—> Tablespace
TS_MAXDS	integer	n	—> Tablespace
TS_MAXISN	integer	n	—> Tablespace
TS_MAXNI	integer	n	—> Tablespace
TS_MAXRECL	integer	n	—> Tablespace
TS_MAXUI	integer	n	—> Tablespace

Column	Data Type	N	Description
TS_MIXDSDEV	yes_no	n	—> Tablespace
TS_NIRABN	integer	n	—> Tablespace
TS_NISIZE	integer		—> Tablespace
TS_NISIZE_UNIT	enumeration	n	’BLOCK’ ’MEGABYTE’ —> Tablespace
TS_PGMREFRESH	yes_no	n	—> Tablespace
TS_UIRABN	integer	n	—> Tablespace
TS_UISIZE	integer	n	—> Tablespace
TS_UISIZE_UNIT	enumeration	n	’BLOCK’ ’MEGABYTE’ —> Tablespace

Roles:

SCHEMA_NAME, TABLESPACE_NAME	identifies the tablespaces.
SCHEMA_NAME	identifies the schema of the tablespace.

D

APPENDIX D — ADABAS SQL SERVER AND OTHER SOFTWARE AG PRODUCTS

This appendix discusses points of special interest when Adabas SQL Server interacts with other Software AG Products.

Adabas SQL Server and Adabas

In terms of Adabas, Adabas SQL Server is an application program which uses the standard Adabas call interface.

In order to make best use of the capabilities of the Adabas database system, Adabas SQL Server makes use of some special Adabas features.

The information given below may help the Adabas database administrator to tune the Adabas nucleus for best performance.

Setting of Adabas Nucleus parameters

There are some sensitive Adabas nucleus parameters which should be set to appropriate values. For details refer to the Adabas documentation for your platform.

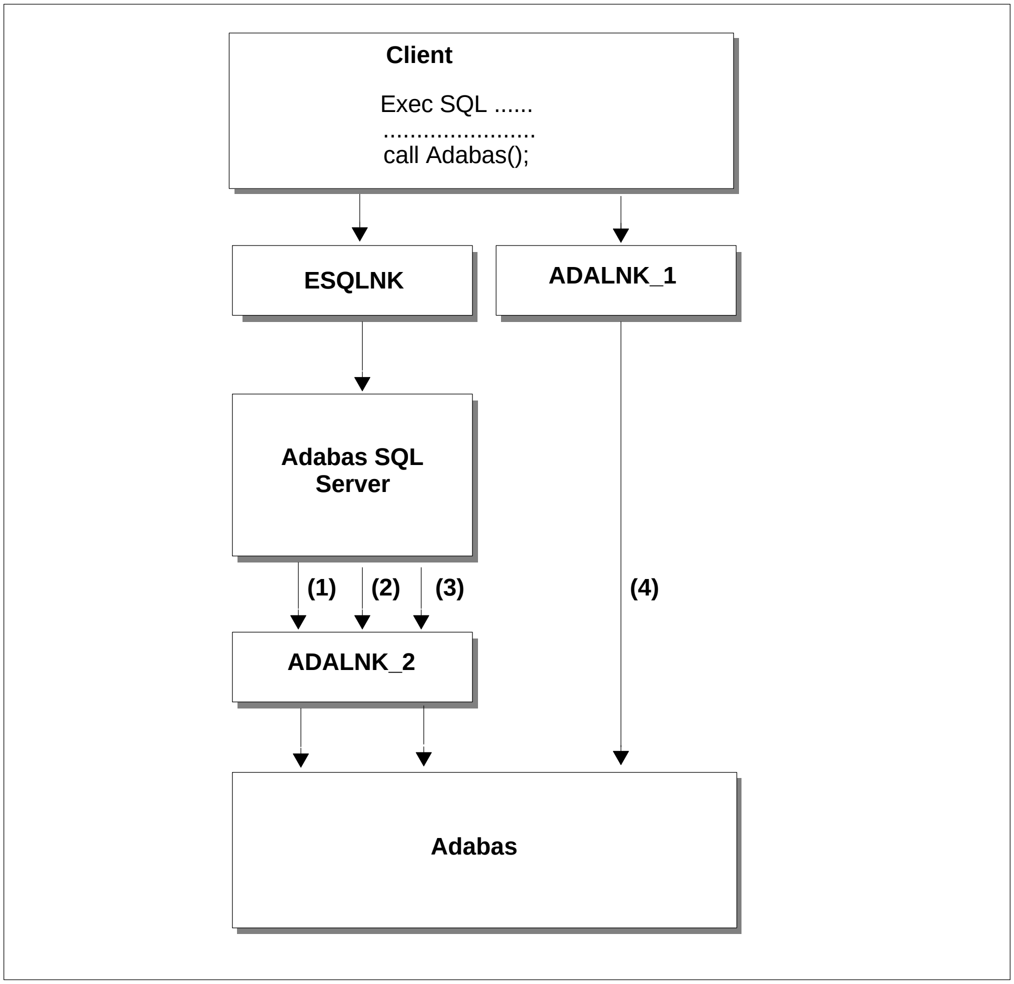
Adabas Session Contexts

Each client using Adabas SQL Server may require up to three independent Adabas session contexts (Adabas user queue elements):

– The “Adabas SQL Server context”:
  This session context is used to execute client-specific Adabas calls, derived from Adabas SQL Server internal operations on the Adabas SQL Server catalog files (for example: read table information).
– The “CLIENT’s DML context”:
  This session context is used to execute client-specific Adabas calls, which are placed by Adabas SQL Server on behalf of the client while executing SQL statements (for example: SELECT * FROM CUSTOMER).
– The “CLIENT’s UTILITY context”:
  This session context is used similarly to the CLIENT’s DML context. To process a client-specific SQL statement, in very special cases Adabas utility calls can be used. As these calls are privileged and strictly Software AG internal, a separate context is used. The lifetime of this contexts is kept as short as possible.

The above mentioned contexts have to be independent because of different transaction scopes.

The identification of the different contexts is built up by Adabas SQL Server automatically, and is different from the default Adabas session context constructed by ADALNK. This is important, because the Adabas session contexts must be distinguishable even if the client places native Adabas calls. This may be the case when Adabas SQL Server is used in LINKED-IN mode.



General overview on ADALNK usage

Adabas SQL Server is using up to three Adabas session contexts ((1), (2) and (3)).

If Adabas SQL Server is used in LINKED-IN mode, then ADALNK_1 and ADALNK_2 are identical. This implies that a client using native Adabas calls (using ADALNK_1) may occupy an additional Adabas session context (4).

Adabas OP Command

Adabas SQL Server issues an explicit Adabas OP command when an Adabas nucleus is addressed in a client-specific manner for the first time. This will be done for each of the above mentioned Adabas session contexts.

Exclusive File Usage

The Adabas database system offers different classes of users. For details refer to the *Adabas Command Reference Manual*, section **Concepts and Facilities**.

Per default, each client resides inside Adabas as an ET-user. For special operations (for example: mass updates using SQL), it might be useful to switch the default status ET-user to the status EXU-user. To become an EXU-user the following steps must be taken:

– enable the EXU file list processing in the client’s parameter file by setting the following parameter:

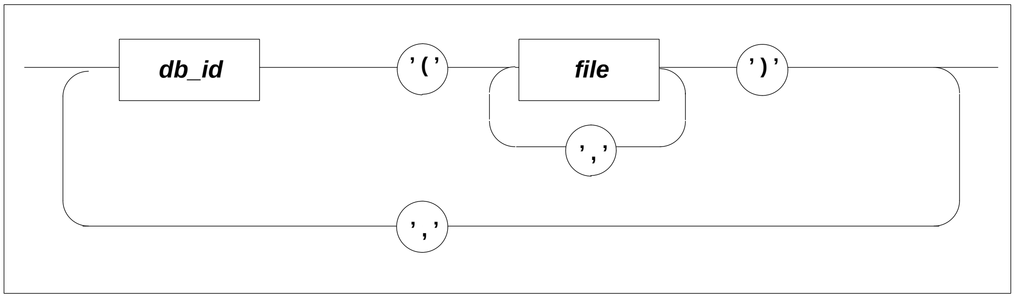
```
EXCLUSIVE UPDATE USER = ON
```

– define the EXU file list by setting the environment variable ESQ_EXULIST. The specified EXU file list wlll be used to build up a restricted file list. This list will be sent along in the Adabas Record Buffer when issuing the corresponding Adabas OP command.

If a client is an EXU-user, the EXU-sensitive Adabas commands like ET/BT are not issued by Adabas SQL Server for this client.

ESQ_EXULIST

The syntax of the environment variable ESQ_EXULIST is:



Example:

```
ESQ_EXULIST 22(4,12)
```

will result in RB = 4,12 for the OP command on database 22.

Set ESQ_EXULIST on Various Platforms

– **Mainframe:**
Enter following syntax into the corresponding line of the VOPRM parameter module:

```
VOENV NAME=ESQ_EXULIST,VALUE=22(4,12)
```

– **VAX/OpenVMS:**
Define a logical name:

```
DEFINE ESQ_EXULIST ”22(4,12)”
```

– **UNIX:**
Use shell-specific commands to set an environment variable:

```
setenv ESQ_EXULIST ”22(4,12)”       (C–Shell)

ESQ_EXULIST=22(4,12)
export ESQ_EXULIST                  (Bourne Shell)
```

– **WINDOWS ODBC:**
Enter the following line to the sagprod.ini file within the Adabas SQL group:

```
ESQ_EXULIST=22(4,12)
```

ADDITIONS1 Field

The ADDITIONS1 field of the Adabas control block may be used to ship a user identification to the Adabas nucleus. For details refer to the *Adabas Command Reference Manual*.

Adabas SQL Server uses the following setting in Additions1:

– For the Adabas SQL Server context: ”*ESQRTS ”.
– For the CLIENT context: ”*<user_name>”.
Where <user_name> is derived from the first seven characters of the CONNECT ID, padded with blanks.

The usage of ’*’ in the first byte of Additions1 implies that no check for uniqueness is performed by the addressed Adabas nucleus.

The above mentioned setting may be used by ADASTAR or can be analyzed using special ADALNK user exits.

Command IDs

Adabas SQL Server makes use of Adabas command IDs. Within Adabas SQL Server, there are two types of command IDs, namely internal command IDs and temporary command IDs.

Internal Command IDs

Adabas SQL Server uses internal command IDs to access the information in the catalog files. Each internal, low-level access to the catalog uses a special command ID.

An internal command ID consists of four bytes, giving a mnemonic abbreviation of the executed low-level function. The naming rule for internal command IDs is:

XYZZ

where:

X	is ‘I’,
Y	∈ {‘A’ – ‘P’} indicating an internal function,
ZZ	∈ {‘00’ – ‘99’} indicating an internal level number.

For metaprogram processing, four further command IDs, IRMP, MRMP, ISTM and IDMP are defined.

Each command ID refers to a specific Adabas Format Buffer.

Temporary Command IDs

Adabas SQL Server dynamically generates temporary command IDs which are used to identify intermediate ISN lists within the Adabas nucleus. The internal generation algorithm attempts to minimize the number of Adabas RC commands.

Temporary command IDs have the following structure:

S*nnn*

with **nnn** running from ”001” till ”999”.

Global Format IDs

Adabas SQL Server uses Global format IDs to make best use of the Adabas format pool.

The current version of Adabas SQL Server uses global format IDs for most Adabas SQL Server internal command IDs.

Global format IDs used by Adabas SQL Server have the following structure:

ES*ccccxy*

where:

ES	is an Adabas SQL Server internal prefix
cccc	is the used internal command ID
x	is the binary value of the catalog file
y	is the version indicator, assigned by Software AG.

Different versions of Adabas SQL Server may run against one Adabas nucleus. Make sure that none of your global format IDs start with the characters ’ES’.

Example:

Assuming an ASCII platform and the catalog being the Adabas file number 15, then the global format ID used to read meta programs would look like this:

”ESIRMP..” Hex: 0x455349524D500F01

On an EBCDIC platform, the same global format ID would look like this:

”ESIRMP..” Hex: 0xC5E2C9D9D4D70F01

Since global format IDs for **all** users are held **only once** in the Adabas format pool, it must be assured that the global format IDs used by Adabas SQL Server do not collide with global format IDs otherwise used at an installation. This would result in an unpredictable condition.

Note:
Ensure that no global format id starting with ’ES’ is used at your site.

Adabas SQL Server and Entire Net-Work

Adabas SQL Server uses the Entire Broker or Entire CSCI for its client/server communication. Both in turn use Entire Net-Work to forward requests from the client to the server. Entire Net-Work must be installed and must be active if you want to use client/server with Adabas SQL Server.

This means that the Entire Net-Work parameters concerning message size and buffer pool size need to be set to appropriate values to accommodate Adabas SQL Server. Especially the parameter TRANSFER_UNIT (old LU parameter) specifies the maximum size of the request and reply buffer. The default is 8000 bytes; the absolute maximum is 64kB.

For more details, please refer to the *Entire Net-Work Reference Manual* for the client or server platform.

Adabas SQL Server and Entire CSCI

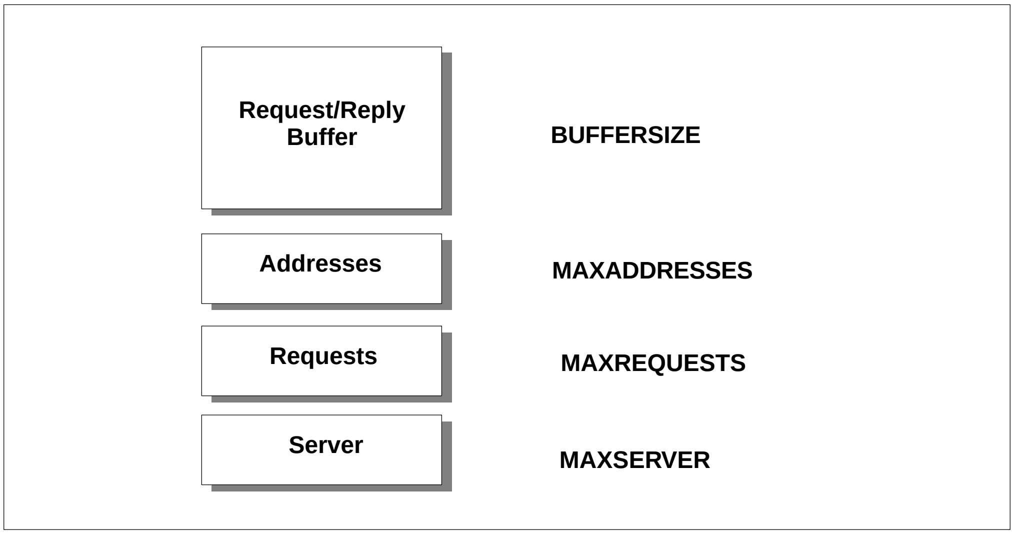
Entire CSCI is used by Adabas SQL Server as default client/server communication protocol. Adabas SQL Server uses the connection oriented communication mode of CSCI.

After generating a server environment (on UNIX/OpenVMS with ”esqgen”), a Server Parameter File was generated with no information about the client/server communication protocol to be used. In such a case, CSCI is assumed. If you want to use Broker, please refer to the next section, **Adabas SQL Server and Entire Broker**.

For UNIX, OpenVMS, and WINDOWS, CSCI is part of the Entire Net-Work installation. For IBM Mainframes, CSCI is part of the Entire Service Manager.

For UNIX and OpenVMS platforms, the operator utility cscopr is provided to administer CSCI. For WINDOWS and IBM Mainframes, this utility does not exist.

CSCI is represented by a shared memory (also called global section) on UNIX and OpenVMS platforms. This shared memory is structured as follows:



The existence of this shared memory indicates that CSCI is active. You can create this shared memory with the cscopr utility:

```
> cscopr
%CSCOPR–I–STARTED, 15–AUG–1995 14:22:30, Version 1.2.1.0 (HP–UX)
%CSCOPR–I–GSNOTAVAL, No global section available
cscopr: create=mem
%CSCOPR–I–CREMEMOK, Memory created
cscopr: quit
%CSCOPR–I–TERMINATED, 15–AUG–1995 14:22:40, Version 1.2.1.0 (HP–UX)
```

The default size of the request/reply buffer is 40kB (10 x 4kB units). If you get the response code ESQ8645 (”no space in communication protocol buffer”) or a similar message, then use cscopr to increase the size of this part of the CSCI shared memory. Before you can do this, you have to shut down all active Adabas SQL Server as well as CSCI. Refer to the following example:

```
> cscopr
%CSCOPR–I–STARTED, 15–AUG–1995 14:22:30, Version 1.2.1.0 (HP–UX)
%CSCOPR–I–GSATTACHED, CSCI global section attached
cscopr: delete=mem
%CSCOPR–I–DELMEMOK, Memory deleted
cscopr: quit
%CSCOPR–I–TERMINATED, 15–AUG–1995 14:22:40, Version 1.2.1.0 (HP–UX)
> cscopr
%CSCOPR–I–STARTED, 15–AUG–1995 14:22:50, Version 1.2.1.0 (HP–UX)
%CSCOPR–I–GSNOTAVAL, No global section available
cscopr: buffersize=20
cscopr: create=mem
%CSCOPR–I–CREMEMOK, Memory created
cscopr: quit
%CSCOPR–I–TERMINATED, 15–AUG–1995 14:23:10, Version 1.2.1.0 (HP–UX)
```

With this setting, the size of the request/reply buffer is 80kB (20 x 4kB units) now. After this, you can restart all Adabas SQL Server.

In addition to the request/reply buffer size, you are able to specify three additional parts of the CSCI shared memory: MAXADDRESSES, MAXREQUESTS, and MAXSERVERS. For Adabas SQL Server you should follow the following rules:

MAXADDRESSES	>= 3 * n_threads
MAXREQUESTS	>= n_threads
MAXSERVER	>= 2 * n_threads + n_servers

n_threads is the sum of all active Adabas SQL Server thread processes for one node. Example: If you have two active servers on one node, one with 5 threads, the other one with 2 threads, then n_threads is 7. n_servers is the sum of all active Adabas SQL Server. For our example, n_servers would be 2. For all three parameters, the default is 50.

To get information about the current parameter setting use cscopr:

```
 > cscopr
%CSCOPR–I–STARTED, 15–AUG–1995 14:22:30, Version 1.2.1.0 (HP–UX)
%CSCOPR–I–GSATTACHED, CSCI global section attached
             cscopr: displ=param

                           CSC Configuration Parameter

                Node Name: WSESQ7       Table Number: 0    Table Size: 49644

                Creation Date: 16–AUG–1995 11:41:33

                Structurelevel: CSCOPR 1    CSCI database 1

                     Table        Num. of entries     Size in byte

            Administration         50                  1876
             Adress broker         50                  1856
             Request Queue         50                  4800
                    Buffer         10                  41000

cscopr: quit
%CSCOPR–I–TERMINATED, 15–AUG–1995 14:23:10, Version 1.2.1.0 (HP–UX)
```

Table size is the complete length of the CSCI shared memory.

To get information about the client/server requests performed by the application program and the server, use the client/server communication logging. For more information, see the *Adabas SQL Server Programmer's Guide*, chapter: L**ogging Facilities**.

For more information about CSCI, see the *Entire Net-Work Reference Manual* of your client or server platform.

Adabas SQL Server and Entire Broker

As an alternative to CSCI, Adabas SQL Server can also use the Entire Broker as client/server communication protocol. Adabas SQL Server uses the conversational communication mode of Broker.

- Before starting your Broker, make a SERVER-specific entry in the Entire Broker Attribute File for Adabas SQL Server, as described in the *Entire Broker Reference Manual, Section* **Attribute File**. The following in an example:

```
*-------------------------------------------------------------------------
*    Adabas SQL Server Service
*-------------------------------------------------------------------------
     DEFAULTS = SERVICE
          CONV–NONACT       =  <conv–nonact>
          SERVER–NONACT     =  <server–nonact>
     CLASS                  =  SAG
     SERVER                 =  ESQ
     SERVICE                =  <service-name>
```

where:
<conv–nonact> >= max. Adabas SQL Server session time–out (default 15 minutes)
<server–nonact> >= max. Adabas SQL Server reply time–out (default 5 minutes)

Note:
The SERVER-Non-Activity-Time (SERVER-NONACT) should be minimally smaller than the value assigned the Entire Net-Work Reply-Time parameter (REPLYTIM).

Note:
Do not define translation services for Adabas SQL Services.

- Furthermore, modify the Entire Broker specific attributes according to the needs of your application. The following is an example:

```
DEFAULTS = BROKER
     NUM-CLIENT        = <max_number_of_clients>
     NUM-SERVER        = <max_number_of_server>
     NUM-SHORT-BUFFER  = <number_of_short_buffers>
     NUM-LONG-BUFFER   = <number_of_long_buffers>
     CLIENT–NONACT     = <client–nonact>
```

where:
<client–nonact> >=max. Adabas SQL Server session time-out (default 15 minutes)

Note:
The relationship of Adabas SQL Clients to Adabas SQL Servers is one to one. Therefore, the number of servers should equal the number of concurrent clients.

Note:
Adabas SQL Server issues requests in conversational mode, which should be taken into regard when calculating the number of long and short buffers.

The average length of a Adabas SQL request or reply is application dependent and controlled by the Adabas SQL Server Parameter (PPL):

– SERVER MAXIMUM REPLY LENGTH
– SERVER MAXIMUM REQUEST LENGTH

These parameters are described in the *Adabas SQL Server Installation and Operations Manual,* **Appendix A – The Parameter Processing Language (PPL)**.

To get information about the client/server requests performed by the application program and the server, use the client/server communication logging. For more information, see chapter **Logging Facilities** in the *Adabas SQL Server Installation and Operations Manual*

For more information about Broker, see the *Entire Broker Reference Manual* of your client, server, or Broker platform.

Adabas SQL Server and Entire Service Manager (CSCI Interface on IBM Mainframes)

The CSCI protocol is used for communication between client and server. This may be conducted using Entire Net-Work if the client and server are on different machines, or using partition communication (ADALNK) if they are on the same machine.

The CSCI interface on the ESG server is supplied as a component of the Entire Service Manager (ESG) and is loaded into the resident page of that server. For communication with Entire Net-Work, the NETCSI module is also supplied as part of the Entire Service Manager (ESG) and needs to be copied into the Entire Net-Work component before installing Adabas SQL Server and the Entire Service Manager.

The CSCI interface on the client side has been modified for mainframe activities and is also supplied as a component of the Entire Service Manager (ESG).

Adabas SQL Server/ADVANCED Interactive Facilities (MVS and VSE only) and Natural

The ADVANCED Interactive Utilities is an application which is based on Software AG's products Natural and the Software AG Editor. It can only be used in environments where these products are available.

Natural User Exits

The ADVANCED Interactive Facilities (AIF) make use of Natural User Exits to communicate with Adabas SQL Server. The User Exit functions process the dynamic SQL statements entered by the user. For details refer to the chapter **Dynamic SQL** earlier in this manual. During the installation process, the modules with the appropriate functions are delivered. To make these functions available, the delivered file must be linked to the existing Natural nucleus. This will be done during the installation of AIF.

The handling of Natural User Exits varies depending on the hardware platform and the Natural version. For details refer to the appropriate Natural documentation.

Natural Database Access

To enable the retrieval functions, the ADVANCED Interactive Facilities (AIF) makes use of Natural data definition modules (DDM). They are created for the physical file in the database which represents the catalog. In order to specify the location of the catalog dynamically, AIF uses logical DDMs. The LFILE parameter in the Natural parameter module contains the information about the file like database ID and actual file number.

Natural Parameter Module

ADVANCED Interactive Facilities (AIF) need a specific Natural environment which is assigned by Natural profile parameters. In order to run the ADVANCED Interactive Utilities, the Natural administrator has to modify the Natural Parameter Module ESQPARM which is delivered with the installation. For details refer to the *Adabas SQL Server Installation and Operations Manual.*

Adabas SQL Server and Adabas ODBC Client

Open Database Connectivity (ODBC) is Microsoft’s open, vendor-neutral, and communication-protocol independent, database connectivity Application Programming Interface (API). ODBC is based on the Call Level Interface (CLI) specification of the SQL Access Group and is sponsored by the X/OPEN group.

Adabas ODBC is Software AG’s standard for data access from Windows-based clients to Adabas databases. SQL-based end-user tools can now access Adabas with the help of Adabas ODBC, thus eliminating the need for users to create their own data access requests.

Adabas SQL Server and Esperant

Esperant is Software AG’s end-user tool enabling the user to generate SQL statements without having a knowledge of either SQL or the underlying database structure. Statements can be entered in the English language and the expert system Esperant will generate valid SQL statements. Esperant supports the Microsoft ODBC Standard and has a direct interface to Entire Access. Esperant provides data access to Adabas using the ODBC Interface and Adabas ODBC.

Adabas SQL Server and Entire Access (Open Systems)

Software AG’s SQL-based middleware product Entire Access enables distributed computing by providing interfaces to multiple relational DBMSs in both client/server and single-platform environments. With an interface to Adabas SQL Server and an application programming interface (API) to Natural, it is possible to use either original Natural DML or common SQL statements embedded in Natural programs against Adabas SQL Server as well as other DBMSs.

Adabas SQL Server and Natural For Adabas SQL (Mainframes)

With the Natural For Adabas SQL interface, Natural users can access Adabas SQL Server using the Natural DML and DDL statements. Depending on the parameter setting the interface uses either standard dynamic SQL or Adabas SQL Server-specific persistent procedures. Using the latter avoids overhead by not having to compile the SQL statements each time they are used.

INDEX

Notes

Notes