
1 Preface

DRIVE/WINDOWS provides you with access to the database system SESAM/SQL-Server V2 by means of SQL statements. This manual contains a brief description of the exact syntax of the DRIVE SQL statements for SESAM/SQL V2 in DRIVE/WINDOWS Version 2.1 for BS2000, MS-Windows and SINIX.

You will find a detailed description of the SQL statements for SESAM/SQL V2 in the "SESAM/SQL-Server (BS2000/OSD), SQL Language Reference Manual, Part 1: SQL Statements" [18],

1.1 Summary of contents

Chapter 3, "DRIVE SQL statements", contains an alphabetical reference section containing all the SQL statements.

Complex statement elements that are used in both DRIVE and SQL statements are described separately in the chapter on "Metavariables" in the "Directory of DRIVE Statements" [3]. DRIVE SQL metavariables that are different from this description are described in this statement directory in a separate chapter entitled "DRIVE SQL metavariables". Some of the examples of the metavariables describe only the metavariable and not the entire syntactical context that makes it possible to output a record.

Access to the database system SESAM/SQL V2 is supported by all three platforms (BS2000, SINIX, MS-Windows) on which DRIVE/WINDOWS executes. Please note that there is no interactive mode or operating mode with the TP monitor (in BS2000 UTM mode) on the MS-Windows platform. Furthermore, this platform does not support the DISPLAY or DISPATCH statement. Corresponding information relates to DRIVE applications that execute under SINIX and BS2000.

Grouping SQL errors in DRIVE error classes (system variable &DML_STATE) is performed using the SQLCODE (system variable &SQL_CODE) that is compatible with SESAM/SQL V1. An SQL error with SQLSTATE XXXXX, SQLCODE -xxx or -1xxx or -2xxx and the message text "SEWXXXX < message_text>" are no longer output as an error message with the format DRI9xxx but as the error message DRI0536 with the format DRI "DRI0536 XXXXX xxxx < message_text>". DRIVE/WINDOWS supports the SQLSTATE of SESAM/SQL V2 by means of the new system variable &SQL_STATE (see chapter 3, "Using variables and constants", in the "DRIVE Programming Language" manual [2]). You will find

the SQLSTATEs including a comparison of all the SQLCODEs in the SESAM “Messages” manual [24]. A brief overview is included in the present manual in section “Mapping the SESAM SQLCODEs to &DML_STATE” on page 253.

In chapter “Syntax overview” you will find an alphabetical reference section containing all the statements and metavariables used in the manual.

The statements are arranged in alphabetical order. There is an entry for each statement: this contains the name of the statement as the heading followed by a brief description.

Statement name - Brief description

A brief description of the function of the statement follows the heading.

This section also describes the prerequisites for successfully executing the statement. In particular, the required access permissions are mentioned.

STATEMENT NAME CLAUSE parameter ...

CLAUSE

Explanation of the clause.

parameter

Explanation of the parameter.

The clauses and parameters are described in the order in which they appear in the syntax diagram.

1.2 Structure of DRIVE SQL statements

DRIVE SQL statements consist of the following elements:

- keywords
- names
- literals
- metavariables
- delimiters
- comments

Example

```

CYCLE cursor_name INTO & variable_name.*;' /*read cursor_name row by row */
                                         /* after variable_name */

CYCLE WHILE &SQL_CLASS <> 02  AND variable <= 1000;
                                         /* Loop until there are no */
                                         /* more rows, but not more */
                                         /* than 1000 times */

```

Keywords: CYCLE, INTO, WHILE, AND, SQL_CLASS (name of a DRIVE system variable)

Names: cursor_name, variable_name

Literals: '02', 1000

Meta variables: variable, .*

Delimiters: Blanks, comparison operators <>, <=, semicolon

Comments: /* read cursor_name row by row */
/* after variable_name */

Keywords

Keywords are words that must be specified as shown in the manual. You will find a list of all DRIVE keywords in the appendix of the “Directory of DRIVE Statements” [3].

Naming conventions

Names identify variable values that the user must replace with current values when entering a statement.

Names can include letters, digits and special characters if no further restrictions are described.

Names that include letters, digits and the underscore character (_) can be entered normally. Names that use other special characters, must be enclosed in double quotes (“”).

In SESAM V2, a catalog name cannot be longer than 18 characters, and an (unqualified) schema name cannot be longer than 31 characters. The name of an authorization key (SQL user name) cannot be longer than 18 characters. The other SESAM rules for catalog names and authorization identifiers must be observed.

If a name is not enclosed in double quotes (regular name), it must start with a letter followed by other letters, digits or underscore characters. It cannot be a DRIVE keyword.

If a name is enclosed in double quotes (special name), it cannot begin with an underscore character and can include any printable character. The special character “ must be entered twice. The first “ character is not included in the length of the name, and the second counts as one character.

When qualifying special schema, table or column names, it is recommended that you place the qualifying dot (the special character .) outside of the special character “, i.e. restrict specialization to unqualified names. DRIVE/WINDOWS treats special names like unqualified names.



There are DRIVE keywords that are not SESAM keywords (e.g. KEY). If you want to use such a keyword as the name of an SQL object (e.g. column) in an SQL statement, you must specify it as a special name (unlike SESAM).

Literals

Literals are constants that are passed to the language processor in the form specified.

Numeric literals can be entered directly and hexadecimal literals (only in DRIVE statements) in the form *X'literal'*.

Alphanumeric literals must be enclosed in single quotes.

In the case of date/time literals, you must specify whether the literal contains a date, time or a time stamp. For interval literals (only in DRIVE statements) you must specify a unit for the interval.

A literal that contains single quotes (') must be enclosed in single quotes.

Example

The literal “That's it“ must be written as follows:

```
'That''s it:'
```

Metavariables

Metavariables are complex statement elements that have been omitted from a statement to facilitate comprehension. They are described in a separate chapter.

Delimiters

Delimiters must be specified between keywords, names, literals and metavariables in order to uniquely identify them. The following can be used as delimiters:

- a semicolon (statement delimiter)
- a blank
- a tabulator character
- a comma (,)
- the concatenation operator ||
- all comparison operators = < > <= >= <>
- all arithmetic operators + - * / % **

A comment or end of line that is not part of a character string that is enclosed in single quotes (') or double quotes (") has the same effect as a delimiter.

Comments

A DRIVE comment starts with the character string /* and ends with the character string */. Any text may be written between these characters, even if it extends over more than one line.


The character strings /* and */ do not indicate comments if they are enclosed in single quotes (') or double quotes (").

SQL comments (see the SESAM manual [18], section 3.2.5) are not permitted in DRIVE/WINDOWS.

1.3 Notational conventions

The following notational conventions are used in this manual:

<hr/>	Syntax definitions
UPPERCASE	SQL keywords
bold	Used for emphasis in running text
<i>italics</i>	Freely selectable names and metavariables in syntax definitions and in running text
Fixed-width font	Predefined names (e.g. commands at operating system level, file names) and error messages in running text, program text in examples and the names of the tables in the examples when used in running text

::=	Definition character The specification to the right of ::= defines the syntax of the element on the left.
[]	Optional specification The brackets are metacharacters and must not be entered in an SQL statement.
{ }	Alternative specifications in syntax definitions. One of the alternatives enclosed in the braces must be specified. The braces are metacharacters and must not be entered in an SQL statement.
{ } ...	Encloses clauses in syntax definitions that can be repeated. The braces are metacharacters and must not be entered in an SQL statement.
...	In syntax definitions, an ellipsis means that you can repeat the preceding specification any number of times. In examples, the ellipsis means that the rest of the statement is of no significance to the example. The ellipsis is a metacharacter and must not be entered in an SQL statement.
	This symbol calls your attention to very important information.

2 Working with SESAM/SQL V2

This chapter describes:

- the organization of a SESAM V2 database and the migration of a SESAM V1 database using an example (page 10)
- rules and recommendations for accessing SESAM V2 SQL objects using DRIVE programs (page 12)
- how to define the database environment (current SQL user, default catalog and default schema) (page 25)
- the syntax and semantic rules for the DRIVE statements `PARAMETER DYNAMIC` and `OPTION` used to define the database environment, as well as alternatives to the `SHOW` statement, which is not currently supported for SESAM V2 access. This statement is used to output information on the metadata of permanent SQL objects and temporary dialog objects (from page 36)
- examples and the sample database (page 42)

2.1 Organization of a SESAM V2 database using an example

This section uses an example to introduce you to the most important objects in a SESAM V2 database, as well as the resources used to create, manage and process these objects. In this example, a table is created by migrating a SESAM V1 database. You will find a complete description of the example and of migration starting on page 42.

2.1.1 Terminology for logical organization

In terms of logic, a SESAM V2 database comprises the following SQL objects:

- catalog

The catalog is a BS2000 file containing the system tables for managing the database. It is generated by the universal user, who is the highest-ranking database administrator, using the `CREATE CATALOG ...` utility statement. A catalog can comprise several

schemas containing a number of base tables. In the example, the catalog "PERSONALVERWALTUNG" consists of the two schemas "STAMMDATEN" and "PROJEKTDATEN".

An application can only access catalogs by means of a DBH (database handler) (see the "DRIVE Programming Language" manual [2], section 9.2, "Special features of SESAM V2"). In the configuration file of the DBH, the logical and physical names of the catalogs that are to be managed by this DBH are declared with ADD-SQL-DATABASE-CATALOG-LIST. All the databases must be located on the computer on which the DBH is running. This means that the physical catalog names can be prefixed with a BS2000 ID. The logical catalog names are unqualified (max. 18 characters) and must be unique.

- schema

A schema is the term used to refer to a set of tables that are logically related. Each schema is assigned to exactly one SQL user, who is the owner of the schema and is thus responsible for the tables. In the example, the schema "STAMMDATEN" consists of the tables "MITARBEITER" and "ABTEILUNG", and the schema "PROJEKTDATEN" consists of the table "PROJEKT".

The schema name must be unique within a catalog.

- tables and columns

A table can be considered a set of columns that represent the user data. A table can be accessed by its owner and any SQL users who the owner has authorized to do so (see the GRANT statement).

The table name must be unique within the schema. The fully qualified name of a table consists of: catalog_name.schema_name.table_name. Outside of the DRIVE application, a name can be qualified in DRIVE/WINDOWS by means of

- compiler options for static programs (see section "OPTION CATALOG/SCHEMA/AUTHORIZATION" on page 28
- dynamic SET statements for dynamic statements used to set the default values for the runtime system (see section "SET CATALOG/SCHEMA/SESSION AUTHORIZATION" on page 26).

- constraint

A constraint (integrity constraint) consists of 1-n columns of one or more tables, which all satisfy a condition. A constraint can be defined by the owner of a table or any SQL user with the appropriate privilege. The following constraints exist:

- UNIQUE (in the example, "ABTEILUNG_NR", "COMP_PERS_NR" and "PROJEKT_NR")
- PRIMARY KEY (see UNIQUE)
- CHECK checks whether the column value lies within a certain range of values or is the NULL value.
- FOREIGN KEY ensures the referential integrity between the source table and the reference table (in the example, "ABT_MIT_NR", "PROJ_MIT", "ABT_LEITER"). A UNIQUE or PRIMARY KEY constraint must be valid for the source table. When the reference table is loaded (via DDL or the Utility Monitor), a check is performed to make sure that the 1-n columns of the FOREIGN KEY contain values that are already in the source table. In order to do this, the SQL user loading the source table must have permission to read the referenced schema (GRANT ... REFERENCES).

The constraint name must be unique within the schema. The fully qualified constraint name consists of: `catalog_name.schema_name.constraint_name`.

- index

In order to improve performance, 1-n columns in a table can be indexed. For SESAM V2, DRIVE/WINDOWS syntax does not currently support indexes. However, base table indexes that were specified in the Utility Monitor, for example, can be used in DRIVE/WINDOWS.

The CREATE INDEX ... statement assumes that the schema has been defined and that table fields exist. The fully qualified index name consists of:
`catalog_name.schema_name.index_name`.

2.1.2 Terminology for physical organization

A database is described physically by a storage group and spaces and is managed using SSL (Storage Structure Language) language elements. A space is a BS2000 file in which the records and indexes are stored in the form of tables. A storage group groups together a number of spaces as a logical unit.

Each SQL user has a default space D0user (11 characters) that is qualified by the catalog name. In addition, each SQL user can set up other spaces that he or she manages as their owner (`catalog_name.space_name`).

Example

A privileged user (e.g. universal user) must first grant an SQL user the right to create a storage group with the GRANT statement. After issuing the CREATE STOGROUP statement, this SQL user can grant other SQL users permission to use this storage group (GRANT USAGE ON STOGROUP). The following table indicates a possible series of transactions in the Utility Monitor for creating the physical database organization:

Transaction	SQL statement
TA1	GRANT CREATE STOGROUP ON CATALOG <i>catalog</i> TO <i>user1</i>
TA2: User <i>user1</i>	CREATE STOGROUP <i>stogroup</i> ...
TA3	GRANT USAGE ON STOGROUP <i>stogroup</i> TO USER <i>user2</i>
TA4: User <i>user2</i>	CREATE SPACE ... USING STOGROUP <i>stogroup</i>

SSL language elements are not supported by DRIVE/WINDOWS.

2.2 Database structure and migration of SESAM V1 tables

In this section, the SESAM V1 tables “MITARBEITER”, “ABTEILUNG” and “PROJEKT” (see section “Examples and sample database” on page 42) are used to illustrate how to create a SESAM V2 database with the Utility Monitor and migrate SESAM V1 tables. It also shows you how to create new SESAM V2 tables with DRIVE programs and grant permissions.

Each access (read, write) performed on an SQL object requires an authorization identifier. This identifier corresponds to the SQL user name. Each SQL user can only access certain SQL objects. Initially, these are all the objects that the SQL user owns. It also includes any objects whose owner has granted read or write permission to the SQL user.

Creating a V2 database and migrating V1 databases (Utility Monitor)

In order to create a database, the universal user (“SYSTEMVERWALTER” in the example), who has all privileges, must perform the following steps in the Utility Monitor (see the SESAM “Utility Monitor” manual [21], chapter 5).

1. Select the function CONFUIRATION (CNF) from the start menu and specify the SQL environment as follows:

```
SEE-AUTHID      : SYSTEMVERWALTER
SEE-CATALOG    : PERSONALVERWALTUNG
SEE-SCHEMA     : STAMMDATEN
```

2. Select the function “Instruction File” (IFP) from the start menu and specify the following instruction files (see page 47) in order to create the catalog, the system entries, the schemas, and in order to define the SQL user:

```
UTI.PERSONALVERWALTUNG.CATALOG
```

```
UTI.PERSONALVERWALTUNG.USER
```

```
UTI.PERSONALVERWALTUNG.SYSTEMUSER
```

```
UTI.PERSONALVERWALTUNG.SCHEMA
```

3. Select the function MIGRATE (MIG) from the start menu and specify the following in order to migrate the SESAM V1 database to a SESAM V2 table:

```
MIGRATE DATABASE      : $Kennung.MITARBEITER.DB-SIB.0006
```

```
WITH INDEX (y/n)     : Y
```

```
TO TABLE             : MITARBEITER
```

The data saved (DB-SIB) in the SESAM V1 database is stored as a SESAM V2 table in the schema “STAMMDATEN”.

In order to migrate the data saved in MITARBEITER.DB-SIB.0006, you must specify the SQL user, the catalog and the schema using the Utility Monitor (see step 1).

The SQL user “SYSTEMVERWALTER“, who has migrate permission, migrates “MITARBEITER“ to the schema PERSONALVERWALTUNG.STAMMDATEN.

Create the SQL table “ABTEILUNG“ (DRIVE program)

The DRIVE program “DRI.TABLE.ABTEILUNG“ creates the SQL table “ABTEILUNG“ with the primary key KEY_ABT_NR in the schema “STAMMDATEN“, whose owner is “PERSONALLEITER“ (see page 50).

Granting access permissions and defining foreign keys (DRIVE program)

As the owner, the SQL user “PERSONALLEITER“ can grant privileges to other SQL users (GRANT statement) so that they can access the data. In the example, the SQL user “PERSONALLEITER“ must grant the SQL user “PROJEKTLEITER“ access permission for the data in the schema “STAMMDATEN“ with the tables “MITARBEITER“ and “ABTEILUNG“. The SQL user “PROJEKTLEITER“ grants access permission for the table “PROJEKT“ in the schema “PROJEKTDATEN“, which he or she owns. In particular, this grants “PERSONALLEITER“ reference permission for “PROJEKT“ (REFERENCES privilege). “PERSONALLEITER“ defines a foreign key reference from “MITARBEITER“ to the primary key of “PROJEKT“ and from “MITARBEITER“ to the primary key of “ABTEILUNG“ (see page 52).

2.2.1 DRIVE requirements for SESAM migration

The following rules must be observed if you want existing DRIVE new-style programs and DRIVE old-style procedures that access SESAM databases of SQL V1 or V14 to execute, without modification if possible, after migration of the database to SESAM V2:

- All database backups (DB SIBs) that are accessed by the DRIVE application are migrated to the same schema.
- Databases that are processed by DRIVE new-style programs using SQL DML statements are migrated as SQL tables. Databases that are processed by DRIVE procedures in old-style or mixed operation using CALL DML statements are migrated as CALL DML tables. If a database has a password catalog (PK-SIB) and is to be migrated to a CALL DML table, the password catalog must be included in migration.
- System entries are created for the DRIVE application (e.g. UTM conversation or TIAM mode).
- If a DRIVE SQL user is not the owner of the schema, this user must be granted all the necessary privileges with the GRANT statement.
- The catalog or CALL DML tables are entered in the configuration file of the SESAM DBH (DBH start instructions ADD-SQL-DATABASE-CATALOG-LIST and ADD-OLD-TABLE-CATALOG-LIST).

If you observe the above mentioned rules for migration, there is no need to modify existing new-style, old-style or mixed-mode DRIVE programs. The SQL tables are made known with the PARAMETER DYNAMIC CATALOG/SCHEMA/AUTHORIZATION statement. In the case of old-style programs or mixed mode with access to CALL DML tables, no settings need be defined for the database environment with the PARAMETER DYNAMIC statement. You can use the PERMIT statement to pass any necessary passwords from new-style to old-style or mixed mode.

2.3 DRIVE program access to SESAM

DRIVE/WINDOWS supports the SQL standard (ISO/IEC 9075:1992). It provides full support for the , a high level of support for the intermediate level, and supports the most important parts of the full level (see the “Database Language SQL” manual [47]). The level of support is determined for the most part by the functional scope of SESAM/SQL Version 2.x.

SESAM databases can be accessed by

- **old-style procedures**

whose SQL language elements are identical with those of DRIVE Version 5.1 (see the “DRIVE V5.1, Part 2: System Directory” [15]). They access the CALL DML tables via the CALL DML interface (see MIGRATE and CREATE TABLE statements).

- **new-style programs**

whose set of SQL language elements is more extensive, i.e.

- almost all the standard statements of SESAM V2 are included (see next section)
- already includes the most important SESAM V2 extensions to the SQL standard (see next section)
- includes convenient DRIVE-specific extensions to the SQL standard (see the “DRIVE Programming Language” manual [2], chapter 9, “Database support”).
- provides a transparent means of formulating dynamic SQL statements with the EXECUTE statement.

You access CALL DML and SQL tables via the static SQL interface (ICSQLE) or the dynamic SQL interface (ESQL/COBOL).

The DRIVE/WINDOWS operating mode determines whether this program style can be used:

- in old-style operation, only old-style procedures can be executed (with the DO statement)
- in new-style operation, only new-style programs can be executed (with the DO, CALL and ENTER statements)
- in mixed operation, both old-style procedures and new-style programs can be executed. A new-style program can call an old-style procedure with DO or CALL (see the “DRIVE Programming Language” manual [2], chapter 15, “Integration of old-style procedures“)

An **SQL statement** in a new-style program is referred to as being


- **static** if it is explicitly coded in the source program and can thus be compiled
- **dynamic** if it is generated, compiled and executed during execution of an EXECUTE statement (see the “DRIVE Programming Language” manual [2], chapter 4, “Programming logic“, and the description of the EXECUTE statement in the “Directory of DRIVE Statements” [3]).

The declarative DRIVE statement `DECLARE VARIABLE ... LIKE TABLE/CURSOR ...` is also considered a static SQL statement. Cursor statements that reference variable cursors are considered dynamic. The executable DRIVE statement `CYCLE cursor INTO...` is considered a static SQL statement if *cursor* is static, and as a dynamic SQL statement if *cursor* is variable.

A **new-style program** is referred to as being

- **static** if it includes at least one executable static SQL statement other than `COMMIT WORK` and `ROLLBACK WORK` and no dynamic SQL statements.
- **dynamic** if it does not include any executable static SQL statement other than `COMMIT WORK` and `ROLLBACK WORK`.
- All other new-style programs are referred to as being **general**.

The following provides you with a description of the rules and recommendations for static and dynamic SQL statements and programs.

 SQL statements entered in **interactive mode** are processed by DRIVE/WINDOWS like dynamic program statements, i.e. in such a way that they observe the appropriate rules and recommendations as far as possible.

If SESAM V1 is being accessed, DRIVE/WINDOWS uses the same interface to access SESAM for both static and dynamic SQL statements. If SESAM V2 is being accessed, DRIVE/WINDOWS uses two different interfaces for performance reasons (see above).

2.3.1 SQL language resources in new style

DRIVE/WINDOWS supports the following interfaces for the DB server SESAM/SQL V2 in new-style operation(see the SESAM manuals [18] and [20] for the classification used):

- Utility statements (e.g. `CREATE CATALOG` and `MIGRATE`) via the SESAM Utility Monitor (see the SESAM manuals [19] and [21])
- UDL (User Definition Language) for managing user entries (e.g. `CREATE USER` and `CREATE SYSTEM_USER`) also via the Utility Monitor
- DDL (Data Definition Language) for defining and managing schemas (e.g. `CREATE SCHEMA` and `CREATE TABLE`). All DDL statements are already included in the DRIVE language resources with only minor restrictions to the functional scope (e.g. no `CALL DML` tables and no cascading deletion yet)
- SSL (Storage Structure Language) for managing the storage structure (e.g. `CREATE INDEX`) via the Utility Monitor
- all statements for transaction management
- all statements for session control

- DML (Data Manipulation Language) for querying and updating data (in particular INSERT, UPDATE, DELETE and DECLARE CURSOR). All DML statements are already included in the DRIVE language resources with no restrictions regarding the functional scope of SESAM/SQL V2.0 and with only minor restrictions regarding the new functions of SESAM/SQL V2.1 (e.g. for SQL expressions and table specifications)
- CALL DML for CALL DML tables via mixed operation (see the DRIVE V5.1 manuals [14] and [15]).

A new-style transaction must either include only DML statements or no DML statements. SESAM only executes an SQL statement if the SQL environment of its transaction permits this. This SQL environment comprises an SQL user whose name is referred to as an authorization identifier, an associated default catalog and an associated default schema. The SQL user is authorized to access the data and metadata of the transaction if this user is the owner or has been granted permission by the owner (see the GRANT statement).

A new-style program can initiate an old-style transaction by calling an old-style procedure (with DO or CALL). This is only permitted outside of new-style transactions. The old-style transaction does not need a (new-style) SQL environment in order to execute its CALL DML statements. Instead it only needs a password, if one is required, which must be entered in new-style operation with the PERMIT statement. DRIVE/WINDOWS then passes this password to the old-style transaction.

2.3.2 Static programs

The SQL statements in a static (new-style) program correspond to the precompiled statements with regard to SESAM terminology. Each DRIVE compilation of a static program is performed as a SESAM precompilation with database contact. This means that SESAM must be available for compilation, i.e. the DBH assigned to DRIVE/WINDOWS must already have been started. In addition, specifications must already have been made for the following:

- AUTHORIZATION (authorization identifier),
- CATALOG (default catalog) and
- SCHEMA (default schema).

These specifications can be made using the following:

- DRIVE compilation options (OPTION statement in the program or OPTION clause of the COMPILE statement). The latter takes precedence over the former.
- DRIVE parameter settings (PARAMETER DYNAMIC statement, see section “Defining the database environment” on page 25).

This results in the following attributes for the transaction profile (abbreviated in the following to TA profile) of a static program without CALL statements:

- If it contains n COMMIT statements, it executes up to n transactions at the time it is executed; if $n = 0$, it can only participate in one transaction of the calling program (see section “Program communication” on page 17)
- All these transactions are assigned to exactly one SQL users, i.e. the one specified via OPTION AUTHORIZATION or the one preset at compilation time with PARAMETER DYNAMIC.
- Access to all data fields and metadata that are not fully qualified (e.g. integrity constraints) is performed using the program qualification, i.e. the default values for the schema and catalog names specified with SCHEMA and CATALOG, respectively.

You can obtain information on the valid values for AUTHORIZATION, CATALOG and SCHEMA from the compiler listing of the program.

2.3.3 Dynamic programs

The SQL statements of a dynamic (new-style) program correspond to prepared statements with regard to SESAM terminology. Preparation is performed at execution time. Therefore, there is no SESAM contact at compilation time. The current SQL environment, i.e. the current authorization key, default catalog name and default schema name are set dynamically during execution by means of SET SESSION AUTHORIZATION/CATALOG/SCHEMA statements. If the program does not include any SET statements, the DRIVE default setting defined by the last corresponding PARAMETER DYNAMIC statement is valid (see section “Defining the database environment” on page 25).

SET SESSION AUTHORIZATION and PARAMETER DYNAMIC AUTHORIZATION are only permitted outside of transactions. It is always possible to change the default catalog or default schema. Therefore, the SQL user can be changed within a transaction; the default catalog and the default schema can even be changed from statement to statement. This means that each transaction has “its own” SQL user and can access the catalogs and SQL objects that “its own” SQL user can access.

This establishes the following attributes for the TA profile of a dynamic program without CALL statements:

- If it contains n COMMIT statements, it executes up to n transactions when it is executed; if $n = 0$, it can only participate in one transaction of the calling program (see section “Program communication” on page 17)
- Each transaction is assigned to exactly one SQL user, i.e. the one last specified with SET SESSION AUTHORIZATION or the last one preset (at compilation time) with PARAMETER DYNAMIC; this authorization identifier can be inherited from the calling program or from interactive mode.

- Access to all data fields and metadata that are not fully qualified is performed using the currently valid default values for SCHEMA and CATALOG, i.e. according to the last SET or PARAMETER DYNAMIC statement. Each default value can be inherited from the calling program or from interactive mode.

The currently valid values for AUTHORIZATION, CATALOG and SCHEMA can be controlled at development time in debugging mode using a trace and can be logged in the debugging listing by means of debugging actions (start program with trace and specify the debugging actions TRACE and DISPLAY LIST in each appropriate SET and PARAMETER DYNAMIC statement and in each transaction statement) (see the “Directory of DRIVE Statements” [3], DEBUG, AT and TRACE statements).

2.3.4 Program communication

DRIVE programs communicate with one another via the CALL statement. A distinction must be made between the following cases:

1. static (new-style) program calls static program
2. dynamic (new-style) program calls dynamic program
3. static program calls dynamic program
4. dynamic program calls static program
5. general (new-style) program calls general program
6. new-style program calls old-style procedure

These cases are explained below using examples.



These distinctions between different cases apply to all platforms (BS2000, SINIX, MS-Windows). On the SINIX platform, the programs that can be executed in BS2000 are referred to as DRIVE alpha programs. These also communicate with each other by means of the CALL statement only. DRIVE windows programs, i.e. programs with a graphical user interface, on the other hand, can also communicate with each other via ADD/NEXT/NEW WINDOW statements and events. In the case of a CALL statement, the body of a DRIVE windows program is executed. In the case of ADD/NEW/NEXT WINDOW, the initialization block is executed and, in the case of an event, the event block of the associated script (see the ON statement in the “Directory of DRIVE Statements” [3]).

There are only DRIVE windows programs on the MS-Windows platform. With regard to the TA profile, however, DRIVE windows programs behave just like DRIVE alpha programs. In the following examples, each CALL statement could also be an ADD/NEW/NEXT WINDOW statement or an ON statement, and each section could be an initialization block or event block rather than just an execution section (body).

A (new-style) transaction is implemented by the three DRIVE programs P1, P2 and P3, where P2 is called by P1 and P3 by P2 using CALL. P2 and P3 must not include any transaction statements. After P2 is called, P1 contains a COMMIT WORK statement. This CALL chain results in the following five program sections, each of which contains SQL statements:

```
PROC P1;
```

```
...
```

```
Section 1 (does not contain a TA statement)
```

```
...
```

```
CALL P2;
```

```
    PROC P2;
```

```
    ...
```

```
    Section 2 (does not contain a TA statement)
```

```
    ...
```

```
    CALL P3;
```

```
        PROC P3;
```

```
        ...
```

```
        Section 3 (does not contain a TA statement)
```

```
        ...
```

```
        END PROC;
```

```
    ...
```

```
    Section 4 (does not contain a TA statement)
```

```
    ...
```

```
    END PROC;
```

```
...
```

```
Section 5 (contains a COMMIT WORK statement)
```

```
...
```

```
END PROC;
```

Case 1 (static calls static)

P1, P2 and P3 are static programs.

CALL P2 and CALL P3 are only executed if P2 and P3 already exist as intermediate code or object code. If P1 itself is part of a CALL chain (i.e. not called with DO or ENTER), this also applies to P1 if a transaction is open when CALL P1 is executed.

The rules mentioned in section 2.3.2, "Static programs", are valid for the programs P1, P2 and P3. In particular, each program has "its own" SQL user, "its own" default catalog and "its own" standard schema in accordance with the compiler listing.

1. If P1 is not called with CALL, the following applies to the execution of P1: P1 starts a new transaction in section 1 and terminates it in section 5 with "its own" SQL environment. P2 resumes this transaction in sections 2 and 4 with "its own" SQL environment. P3 resumes this transaction in section 3 with "its own" SQL environment. The same applies if P1 is called with CALL but no transaction is open when CALL P1 is executed.
2. If, however, P1 is called with CALL and a transaction is open at this time, P1 resumes this transaction with "its own" SQL environment, i.e. P1 assumes the roll of P2 but with the additional attribute that P1 terminates the inherited transaction.

This results in the following attributes for the TA profile when P1 is executed:

- If P1 is not called with CALL, but with DO or ENTER instead, a transaction with up to three SQL users and up to three default catalogs and up to three default schemas exists.
- If P1 itself is part of a CALL chain, the TA profile depends on whether a transaction is open when P1 is called.

If no transaction is open, the aforementioned applies to DO P1. If, on the other hand, a transaction is open, P1 resumes this "foreign" transaction in section 1 and terminates it in section 5 using "its own" SQL environment; this resumed transaction can have more than three SQL environments.

- If P1 returns to its caller (not for successor DO and ENTER), it does not take its SQL environment with it. The SQL environment that was valid before P1 was called resumes to be valid for the caller of P1.

Case 2 (dynamic calls dynamic)

P1, P2 and P3 are dynamic programs.

P1, P2 and P3 do not need to exist as intermediate code or object code. Because dynamic statements cannot be generated or compiled until execution time, the existence of intermediate code or object code has no influence on the performance of the SQL statements (exception: variable cursors).

The rules specified in the section “Dynamic programs” apply to the programs P1, P2 and P3. In particular, it is not possible to change the SQL user within a transaction. It is, however, possible to change the default catalog and default schema from statement to statement (inasmuch as this is useful).

1. If P1 is not called with CALL, the following applies to the execution of P1:

P1 starts a new transaction in section 1 with an SQL user that it set itself with SET SESSION AUTHORIZATION or which has been set according to the current default value specified for PARAMETER DYNAMIC AUTHORIZATION. P1 terminates this transaction in section 5 with the same SQL user. P2 and P3 resume this transaction in sections 2 and 4 and section 3, respectively, each with the same SQL user. The same applies if P1 is called with CALL but no transaction is open when CALL P1 is executed (see case 3).

2. If, on the other hand, P1 is called with CALL and a transaction is open at this time, P1 resumes this transaction with the current authorization identifier, i.e. P1 assumes the role of P2 but with the additional attribute that P1 terminates the inherited transaction.

This results in the following attributes for the TA profile when P1 is executed:

- If P1 is not called with CALL, a transaction with one SQL user and “many” default catalogs and schemas exists.
- If P1 is part of a dynamic CALL chain, the TA profile depends on whether a transaction is open when P1 is called.

If no transaction is open, P1 initiates “its own” transaction in section 1 and terminates it in section 5 using “its own” SQL user.

If a transaction is open, P1 resumes this “foreign” transaction in section 5 using “its own” SQL user. The initiated and resumed transactions can each only have one SQL user.

- If P1 was called with CALL, it takes its current SQL environment with it when it returns to its caller. This means that a different current SQL user is valid for the P1 caller if no transaction was open for CALL P1, and P1 issued a new SET SESSION AUTHORIZATION statement.

A different standard catalog or schema is valid for the P1 caller if P1 or P2 or P3 issued a new SET CATALOG or SET SCHEMA statement.

- If P1 was called with DO, DRIVE/WINDOWS reestablishes the SQL environment according to the last PARAMETER DYNAMIC AUTHORIZATION/CATALOG/SCHEMA statement when the interactive mode is resumed.

Case 3 (static calls dynamic)

P1 and P3 are static, and P2 is dynamic.

1. If P1 is not called with CALL, the following applies to the execution of P1:

P1 starts a new transaction in section 1 and terminates it in section 5 with “its own” SQL environment.

P2 resumes this transaction in sections 2 and 4 with the SQL user of the last SET SESSION AUTHORIZATION or PARAMETER DYNAMIC AUTHORIZATION statement (before P1 was called).

P3 resumes this transaction in section 3 with “its own” SQL environment.

The same applies if P1 is called with CALL but no transaction is open for CALL P1.

2. If, on the other hand, P1 is called with CALL and a transaction is open at this time, P1 resumes this transaction with “its own” SQL environment. The SQL user valid in P2 was set outside of this inherited transaction.

This results in the following attributes for the TA profile when P1 is executed:

- If P1 is not called with CALL and no transaction is open for CALL P1, a transaction with up to three SQL users and “many” default catalogs and schemas exists.
- If a transaction is open for CALL P1, P1 resumes this transaction in section 1 and terminates it in section 5 with “its” SQL user. This transaction can have more than three SQL users and “many” default catalogs and schemas.

Case 4 (dynamic calls static)

P1 and P3 are dynamic, and P2 is static.

1. If P1 is not called with CALL, the following applies to the execution of P1:

P1 starts a new transaction in section 1 with an SQL user that it set itself with SET SESSION AUTHORIZATION or which has been set according to the current default specified for PARAMETER DYNAMIC AUTHORIZATION. P1 terminates this transaction in section 5 with the same SQL user.

P2 resumes this transaction in sections 2 and 4 with “its” SQL user, and P3 resumes it in section 3 with the SQL user of P1. The same applies if P1 is called with CALL but no transaction is open for CALL P1.

2. If, on the other hand, P1 is called with CALL and a transaction is open, P1 resumes this transaction with the current authorization identifier and terminates the inherited transaction with this SQL user.

This results in the following attributes for the TA profile when P1 is executed:

- If P1 is not called with CALL or no transaction is open for CALL P1, a transaction with up to two SQL users and “many” default catalogs and schemas exists.
- If a transaction is open for CALL P1, P1 resumes this transaction in section 1 and terminates it in section 5 with the current authorization identifier. This transaction can have more than two SQL users and “many” default catalogs and schemas.

Fall 5 (general calls general)

P1, P2 and P3 are general programs.

In the cases 1 to 4, each program is assigned exactly one SQL user; in this case, each program can have one “static” and one general “dynamic” SQL user. The TA profile is thus more general and unclear.

Case 6 (new-style calls old-style)

If a new-style program P4 calls and old-style procedure P5 with DO or CALL, no new-style transaction can be open. Furthermore, no old-style transaction can be open when the program returns to new-style operation (calling program or interactive mode) from old-style.

New-style transactions ignore old-style passwords and old-style transactions ignore new-style SQL users. The TA profiles of P4 and P5 are therefore disjunctive.

- You can obtain the TA profile for P4 by replacing the CALL statement in P4 with a COMMIT WORK statement. In this case, the information supplied in cases 1 through 5 applies to P4.
- An old-style procedure can only work with one password. The TA profile of an old-style procedure therefore corresponds to the TA profile of a static new-style program.

2.3.5 Programming recommendations

The following recommendations increase the readability of the DRIVE programs and makes the SQL environment in which they execute easier to understand:

- Try to use only one SQL user per DRIVE session (TIAM session or UTM conversation) and define this user as the default value using PARAMETER DYNAMIC AUTHORIZATION (single-SQL-user session. DRIVE/WINDOWS can be configured accordingly (see the “DRIVE Programming System” manual [1], chapter 11, “Data security”).
- Develop as many static and dynamic programs as possible and use as few general programs as possible.
- Develop dynamic programs without PARAMETER DYNAMIC AUTHORIZATION/CATALOG/SCHEMA statements, and conversely develop programs containing PARAMETER DYNAMIC statements without SQL statements.
- Design dynamic programs as single-SQL-user programs, i.e. no SET SESSION AUTHORIZATION statement (the SQL user is passed on by the caller) or only one such statement and place it before the SQL statement that is considered the first statement with regard to program logic.
- Define the TA profile of general programs by means of an OPTION AUTHORIZATION statement and dynamic SET SESSION AUTHORIZATION statements (multi-SQL-user programs)

2.3.6 Incompatibilities

If you are working with the database system SESAM/SQL V1.1, access to the database system is the same for all DRIVE/WINDOWS versions. If you are accessing the database system SESAM/SQL V2, you will find there are the following three differences between DRIVE/WINDOWS V2.x and earlier DRIVE versions (DRIVE/WINDOWS V1.x or DRIVE V6.x):

- Static SQL objects cannot be referenced in dynamic SQL statements:

- If a static cursor is declared but is referenced dynamically, e.g.

```
DECLARE C1 CURSOR FOR SELECT SCHLUESSEL FROM FIRMA;
EXEC 'OPEN C1';
```

DRIVE/WINDOWS V2.x issues the following message upon execution:

DRI0033 DYNAMIC STATEMENT NOT PERMITTED.

The system variable &ERROR contains 'SYNTAX ERROR'.

- If a static temporary view is declared but is referenced dynamically, e.g.

```
CREATE TEMPORARY VIEW V1 AS SELECT SCHLUESSEL FROM FIRMA;
EXEC 'INSERT INTO V1 VALUES (''HUG001'')';
```

DRIVE/WINDOWS V2.x also issues message DRI0033 upon execution and supplies &ERROR with the value 'SYNTAX ERROR'.

- Dynamic temporary views defined in programs should be deleted explicitly before the program in which they were defined is exited with DROP TEMPORARY VIEW..., or at the end of the DRIVE application with DROP TEMPORARY VIEWS. Otherwise, the following occurs:

- At the end of the application, i.e. when you switch to interactive mode or to the SPU, the following message is output:

DRI0488 EXISTING DYNAMIC TEMPORARY VIEWS RELEASED

- If a successor program is called with DO at the end of the program, the program is aborted with message DRI0488, i.e. the successor program is not executed.



The owner of a dynamic temporary view is the SQL user current at the time that the dynamic CREATE TEMPORARY VIEW statement is executed (last SET SESSION AUTHORIZATION or PARAMETER DYNAMIC statement). If there is a different current SQL user when the DROP TEMPORARY VIEW(S) statement is executed, the dynamic temporary view cannot be deleted since only the owner can delete a temporary view.

- If a program that includes static SQL statements other than COMMIT WORK and ROLLBACK WORK is called with CALL in an open transaction, it must already exist as intermediate code or object code. Otherwise DRIVE/WINDOWS issues the following message and aborts execution:

DRI0490 SESAMSQL COMPILATION NOT POSSIBLE WITH TRANSACTION OPEN

2.4 Defining the database environment

The database environment for SESAM/SQL V2 consists of all the SQL2 databases that can be referenced with a SESAM/SQL2-Server (SESAM DBH). This is defined in the server configuration file with ADD-SQL-DATABASE-CATALOG-LIST.

If you want to work with SESAM/SQL V2, you must specify the following information:

- default catalog name (see section “Changing the database and schema” on page 28)
- default schema name (see page 28)
- current authorization identifier (SQL user name).

These three names are used to define the current SQL environment for the DRIVE session (TIAM session or UTM conversation).

When a DRIVE program accesses the data, the SESAM/SQL V2 server checks the following with regard to the SQL user:

- whether the SQL user has been entered (see the CREATE USER statement)
- whether a system entry exists for the SQL user (see the CREATE SYSTEM_USER statement in the SESAM manual [18])
- whether the SQL user has access permission for the data (see the GRANT statement)

For compilation and for programs containing static SQL statements, these parameters are supplied with a value with OPTION CATALOG/SCHEMA/AUTHORIZATION (page 28). In the case of dynamic SQL statements, these parameters are supplied with values with PARAMETER DYNAMIC or SET CATALOG /SCHEMA/ AUTHORIZATION (see page 36, page 136, page 138, page 140). SET values are only valid for the current DRIVE program, PARAMETER values are considered the last values set and are valid beyond the end of the DRIVE program in interactive mode and are used as the default values in the parameter mask.

If during compilation OPTION statements or clauses are missing, the parameter settings are used. If these do not exist, DRIVE/WINDOWS aborts in the event of AUTHORIZATION with error message DRI0525 and uses as the default value a blank for CATALOG and SCHEMA.

This section describes the interaction between the DRIVE and SQL statements that you use to define the database environment: PARAMETER, OPTION and SET with the operands CATALOG/SCHEMA/AUTHORIZATION.

2.4.1 SET CATALOG/SCHEMA/SESSION AUTHORIZATION

You use the SET statement to define the SQL environment in which you want to work. The SET statement does not take effect until it is executed. For DRIVE/WINDOWS (as for SESAM), SET SCHEMA and SET CATALOG only affect dynamic statements. In DRIVE/WINDOWS, SET SESSION AUTHORIZATION only affects dynamic statements (unlike SESAM).

The following is valid:

- A SET CATALOG statement defines the default catalog name for dynamic SQL statements. In program mode, it is permitted in static and dynamic statements. It is not permitted in interactive mode.
- A SET SCHEMA statement defines the default schema name for dynamic SQL statements. In program mode, it is permitted in static and dynamic statements. It is not permitted in interactive mode.
- A SET SESSION AUTHORIZATION statement defines the current authorization identifier (SQL user name) for subsequent SQL statements. In program mode, it can only be used dynamically and thus only affects dynamic SQL statements. In an open transaction, SESAM rejects this statement with SQLSTATE 25SA4. It is not permitted in interactive mode.

2.4.2 PARAMETER DYNAMIC CATALOG/SCHEMA/AUTHORIZATION

You use the PARAMETER DYNAMIC CATALOG/SCHEMA/AUTHORIZATION statement to define the SQL environment for the DRIVE session. The values you specify are managed by DRIVE/WINDOWS and are passed to the database system SESAM V2. This means that DRIVE/WINDOWS ensures that the DRIVE and SQL environments remain consistent. You can set the parameters CATALOG/SCHEMA/AUTHORIZATION

- in interactive mode (as a single statement or with the help of screen masks in BS2000 and SINIX, or by means of menus in the SPU)
- in program mode
- as startup parameters when you start UTM
- in the UTM leader program via user labels

These default settings are committed in the event of an explicit COMMIT WORK statement and in the following situations:

- when the program is terminated normally or with errors (END PROC at the highest level or BREAK)
- before successor DO statements
- after ROLLBACK WORK in interactive mode
- after termination of the first transaction in program mode by means of ROLLBACK WORK
- at the start of a UTM conversion if startup parameters exist for the UTM application or user labels with the corresponding PARAMETER DYNAMIC statements.

The effect of PARAMETER DYNAMIC CATALOG/SCHEMA/AUTHORIZATION is different for static and dynamic statements.

When programs are compiled, the priority of the values specified for CATALOG/SCHEMA/AUTHORIZATION is as follows:

- if values for CATALOG/SCHEMA/AUTHORIZATION are specified in the program with the OPTION statement, these values are valid if no corresponding OPTION values were specified for COMPILE,
- otherwise the compiler options are valid (OPTION clause of the COMPILE statement).
- If these also do not exist, the values specified as the default parameters are valid.
- If these also do not exist, a blank is the default value for CATALOG and SCHEMA. In the case of AUTHORIZATION, the program is aborted and error message DRI0525 is issued.

The three valid values are written in the compiler listing and are committed in the stored intermediate code (by SESAM) so that they are valid for all static statements when the program is executed regardless of the current values of the runtime system.

The following applies to dynamic statements:

- If values are specified for CATALOG/SCHEMA/AUTHORIZATION in the program with SET statements, these are valid regardless of any other settings.

A COMMIT WORK statement in the program commits the values set for CATALOG/SCHEMA/AUTHORIZATION with SET statements. These values are valid until the end of the program if no further SET statements follow and the parameter defaults are not modified.

A ROLLBACK WORK statement in the program resets the values set for CATALOG/SCHEMA/AUTHORIZATION by means of SET statements if they have not already been committed with COMMIT WORK. This means that the values specified as the parameter defaults or the values from the last committed SET statement are then valid.

- If the program does not contain any SET statements for setting the values for CATALOG/SCHEMA/AUTHORIZATION, the current parameter default values of the runtime system are valid.

2.4.3 OPTION CATALOG/SCHEMA/AUTHORIZATION

You can also use the OPTION statement to control the compilation of a DRIVE program that accesses a SESAM database. This statement is valid for compilation. In addition, the values are committed in the stored intermediate code so that they are valid for all static SQL statements when the program is executed, regardless of the values currently valid for the runtime system.

When a program is compiled, option values are only taken over for static statements. If no options are set, DRIVE/WINDOWS checks whether parameter values have been set. If this is not the case, the program is aborted and error message DRI0525 is issued in the case of AUTHORIZATION. In the case of CATALOG or SCHEMA, DRIVE/WINDOWS uses a blank as the default value.

Because DRIVE/WINDOWS **always compiles with database contact**, SESAM opens a database transaction (“SESAM precompilation”) at the beginning of DRIVE compilation. SESAM does not roll back this transaction until the end of DRIVE compilation. This means in particular that only one authorization identifier can be used per compilation. The use of several authorization identifiers would only be appropriate for SESAM precompilation without database contact. This is not permitted for DRIVE/WINDOWS.

Compilation can only be started if no transaction is open since the first (declarative or executable) static SQL statement opens a transaction in SESAM.

2.4.4 Changing the database and schema

The SCHEMA operand of PARAMETER DYNAMIC and OPTION, and the SET SCHEMA statement allow you to define a default schema name. Any subsequently specified unqualified table names (base table or permanent view) or integrity constraint names refer to this schema.

The CATALOG operand of PARAMETER DYNAMIC and OPTION, and the SET CATALOG statement allow you to define a default database name (SESAM catalog). Any subsequently specified unqualified schema, table or integrity constraint names refer to this database.

You can reference a different schema than the default schema by qualifying the name of table or integrity constraint with a schema name. Likewise, you can reference a different database than the default database by qualifying the name of a schema, table or integrity constraint with a catalog name. This access can, for example, be used if you want to reference two databases in a single transaction. In this case, however, it must be possible to access both databases with the same SQL user since the SQL user can only be changed outside of a transaction. The various transaction profiles of static and dynamic (new-style) programs are described in section “DRIVE program access to SESAM” on page 12.

2.4.5 Current authorization identifier for compilation and execution

The current authorization identifier is determined as follows:

- If `OPTION AUTHORIZATION` is specified, the authorization identifier specified is valid for the entire compilation process and for the execution of every compiled static SQL statement.
- If `OPTION AUTHORIZATION` is omitted, the current authorization identifier from the parameter settings for `DRIVE/WINDOWS` at compilation time are valid for compilation and for the execution of every compiled static SQL statement.

The current authorization identifier cannot be changed by means of `SET SESSION AUTHORIZATION` or `PARAMETER DYNAMIC AUTHORIZATION` statements.

The current authorization identifier is reset by an implicit `ROLLBACK WORK` issued by SESAM once compilation has been completed if it was specified in the program by means of `OPTION`. In this case, the current parameter default is again valid.

At execution time, all `SET SESSION` and `PARAMETER DYNAMIC AUTHORIZATION` statements are executed, and consequently the current authorization identifiers are updated. These updates of the current SQL user only affect dynamic statements. The authorization identifier valid for static statements is the authorization identifier specified for compilation as described above.

2.4.6 Examples of database environments

2.4.6.1 SESAM database and DRIVE programs

The SQL user FLE has access to the catalog TLDBSQL2 and owns the schema FLE. The schema FLE comprises the table FIRMA with the column SCHLUESSEL of the type CHAR(6).

The SQL user HUGO does not have any access to the catalog TLDBSQL2, i.e. either the user does not exist, or the user does not have a system entry for the BS2000 user ID under which DRIVE/WINDOWS is running or for DRIVE/WINDOWS as an UTM application.

The SQL user DOM has access to the catalog TLDBSQL2, but no access to the table FIRMA, i.e. the owner FLE has not executed an appropriate GRANT statement for DOM.

The DRIVE library "\$YDRI6FLE.FLE.LIB" (link name USEROML) contains the following DRIVE programs:

```
/* DRIVE program ENVIRONMENT */
PROC ENVIRONMENT;
PAR DYN AUTHORIZATION=FLE;
PAR DYN CATALOG=TLDBSQL2;
PAR DYN SCHEMA=FLE;
END PROC;

/* DRIVE program BEISPIEL */
PROC BEISPIEL;
DCL VAR &FIRMA LIKE TABLE FIRMA;
EXEC 'SET SESSION AUTHORIZATION ''HUGO''';
INSERT INTO FIRMA (SCHLUESSEL) VALUES ('HUG000');
EXEC 'INSERT INTO FIRMA (SCHLUESSEL) VALUES (''HUG001'')';
ROLLBACK WORK;
END PROC;
```

2.4.6.2 TIAM operation

The following examples illustrate the interaction of the statements PARAMETER, OPTION and SET used to define the SQL environment for TIAM operation.

do environment

compile beispiel option listing=lib authorization=hugo

```
PROGRAMM      : BEISPIEL
BIBLIOTHEK    : FLE.LIB
```

```
ZEILE  QUELLE  NEST  BEISPIEL                                SEITE :      1

1      1      0  PROC BEISPIEL;
2      2      0  DCL VAR &FIRMA LIKE TABLE FIRMA;
***          *
***          % DRI0536 42SQG -550 SYSTEMZUGANG FUER DEN
ZUGREIFBAR  BERECHTIGUNGSSCHLUESSEL HUGO IM CATALOG TLDBSQL2 NICHT
3      3      0  EXEC 'SET SESSION AUTHORIZATION ''HUGO''';
4      4      0  INSERT INTO FIRMA (SCHLUESSEL) VALUES ('HUG000');
***          *
***          % DRI0536 42SQG -550 SYSTEMZUGANG FUER DEN
ZUGREIFBAR  BERECHTIGUNGSSCHLUESSEL HUGO IM CATALOG TLDBSQL2 NICHT
5      5      0  EXEC 'INSERT INTO FIRMA (SCHLUESSEL) VALUES (''HUG001'')';
6      6      0  ROLLBACK WORK;
7      7      0  END PROC;
```

```
UEBERSETZUNGSOPTIONEN  BEISPIEL  SEITE :      2
```

```
OPTION DBSYSTEM      = SESAMSQL  DURCH VOREINSTELLUNG
OPTION LISTING       = LIBRARY   DURCH KOMMANDO
OPTION CODE          = OFF       DURCH VOREINSTELLUNG
OPTION SCHEMA        = FLE       DURCH VOREINSTELLUNG
OPTION CATALOG       = TLDBSQL2  DURCH VOREINSTELLUNG
OPTION AUTHORIZATION = HUGO      DURCH KOMMANDO
```

```
ANZAHL FEHLER :      2
```

compile beispiel option listing=lib authorization=dom

```
PROGRAMM      : BEISPIEL
BIBLIOTHEK   : FLE.LIB
```

```
ZEILE QUELLE NEST BEISPIEL                SEITE :      1
```

```
1          1      0  PROC BEISPIEL;
2          2      0  DCL VAR &FIRMA LIKE TABLE FIRMA;
***
***          *
***          % DRI0536 42SQK -127 TABELLE TLDBSQL2.FLE.FIRMA FUER
BENUTZER DOM NICHT ZUGREIFBAR
3          3      0  EXEC 'SET SESSION AUTHORIZATION ''HUGO''';
4          4      0  INSERT INTO FIRMA (SCHLUESSEL) VALUES ('HUGO00');
***
***          *
***          % DRI0536 42SQK -127 TABELLE TLDBSQL2.FLE.FIRMA FUER
BENUTZER
          DOM NICHT ZUGREIFBAR
5          5      0  EXEC 'INSERT INTO FIRMA (SCHLUESSEL) VALUES (''HUGO01'')';
6          6      0  ROLLBACK WORK;
7          7      0  END PROC;
```

```
UEBERSETZUNGSOPTIONEN  BEISPIEL                SEITE :      2
```

```
OPTION DBSYSTEM      = SESAMSQL  DURCH VOREINSTELLUNG
OPTION LISTING       = LIBRARY   DURCH KOMMANDO
OPTION CODE          = OFF       DURCH VOREINSTELLUNG
OPTION SCHEMA        = FLE       DURCH VOREINSTELLUNG
OPTION CATALOG       = TLDBSQL2  DURCH VOREINSTELLUNG
OPTION AUTHORIZATION = DOM       DURCH KOMMANDO
```

```
ANZAHL FEHLER :      2
```


compile beispiel option listing=lib code=on

```
PROGRAMM      : BEISPIEL
BIBLIOTHEK   : FLE.LIB
```

```
ZEILE  QUELLE NEST BEISPIEL                               SEITE :      1
```

```
1      1      0  PROC BEISPIEL;
2      2      0  DCL VAR &FIRMA LIKE TABLE FIRMA;
3      3      0  EXEC 'SET SESSION AUTHORIZATION 'HUGO'';
4      4      0  INSERT INTO FIRMA (SCHLUESSEL) VALUES ('HUG000');
5      5      0  EXEC 'INSERT INTO FIRMA (SCHLUESSEL) VALUES ('HUG001')';
6      6      0  ROLLBACK WORK;
7      7      0  END PROC;
```

```
UEBERSETZUNGSOPTIONEN  BEISPIEL                               SEITE :      2
```

```
OPTION DBSYSTEM      = SESAMSQL  DURCH VOREINSTELLUNG
OPTION LISTING       = LIBRARY   DURCH KOMMANDO
OPTION CODE          = ON        DURCH KOMMANDO
OPTION SCHEMA        = FLE       DURCH VOREINSTELLUNG
OPTION CATALOG       = TLDBSQL2  DURCH VOREINSTELLUNG
OPTION AUTHORIZATION = FLE       DURCH VOREINSTELLUNG
```

```
ANZAHL FEHLER :      0
```

debug beispiel

```
***** BIBLIOTHEK : FLE.LIB
      ZEILE PROGRAMM : BEISPIEL
      1 PROC BEISPIEL;
% DRI0593 ABLAUFVERFOLGUNG BEENDET: ZEILE      1 IN PROZEDUR
'FLE.LIB(BEISPIEL)
,
% DRI0576 ANFANGS-HALTEPUNKT ERREICHT
*t all
***** BIBLIOTHEK : FLE.LIB
      ZEILE PROGRAMM : BEISPIEL
      3 EXEC 'SET SESSION AUTHORIZATION 'HUGO'';
      4 INSERT INTO FIRMA (SCHLUESSEL) VALUES ('HUG000');
      5 EXEC 'INSERT INTO FIRMA (SCHLUESSEL) VALUES ('HUG001')';
% DRI0536 42SQG -550 SYSTEMZUGANG FUER DEN BERECHTIGUNGSSCHLUESSEL HUGO
IM CA
TALOG TLDBSQL2 NICHT ZUGREIFBAR
% DRI0579 FEHLER BEI AUSFUEHRUNG DER ANWEISUNG 'EXEC'
% DRI0578 AKTUELLER HALTEPUNKT: ZEILE      5 IN PROZEDUR 'FLE.LIB(BEISPIEL)'
*break debug
```

2.4.6.3 UTM operation

The examples for TIAM operation behave exactly the same during UTM operation. Only the definition of system entries for SQL users is different (see the CREATE SYSTEM_USER statement in the SESAM manual [18]).

The SQL environment can also be defined by means of parameters in the UTM start procedure and by means of user labels with pointers to UTM leader procedures.

The example below illustrates how you can define the SQL environment with PARAMETER statements in UTM operation.

The UTM start procedure contains the following DRIVE startup parameters:

```
.DRIVE PAR DYN LIB="$YDRI6FLE.FLE.LIB";
.DRIVE PAR DYN CATALOG=WRZLPRMPFT;
.DRIVE PAR DYN SCHEMA=HOKUS;
.DRIVE PAR DYN AUTHORIZATION=POKUS;
```

The DRIVE library "\$YDRI6FLE.FLE.LIB" (link name USEROML) contains an S-type member with the name "DRISQL@@FELIX@@@" or "DRISQL@@@@@@@@@" and, as the contents, the single line

```
"$YDRI6FLE.FLE.LIB"(ENVIRONMENT)
```

(user label with a pointer to a leader procedure).

The UTM leader procedure ENVIRONMENT was described earlier.

The following steps are carried out when DRIVE is started in UTM mode:

1. When UTM is started, the parameter values WRZLPRMPFT, HOKUS and POKUS are retained without sending them so SESAM.
2. Once the UTM user FELIX has signed on with KDCSIGN and the first transaction code DRISQL has been entered, the SET CATALOG 'WRZLPRMPFT', SET SCHEMA 'HOKUS' and SET SESSION AUTHORIZATION 'POKUS' are sent to SESAM.
3. The user label DRISQL@@FELIX@@@ is subsequently found and the header procedure ENVIRONMENT executed.

In the interest of simplicity and clarity, it is recommended that you program UTM header procedures without SQL statements and place desired SQL access in a successor DO or CALL program that is called immediately before END PROC.

The current parameter defaults are committed when END PROC is executed. The UTM user FELIX will then find himself/herself in DRIVE interactive mode, and the SQL user FLE is the default value.

In case the leader procedure includes a successor DO statement as the last statement before END PROC, the parameter defaults are also committed before the new interactive program is started. If the last statement before END PROC is, however, a CALL statement, they are not committed before the new program is started but when the next COMMIT WORK statement is executed.

2.5 DRIVE statements for SESAM V2

You can use the DRIVE statements `PARAMETER DYNAMIC` and `OPTION` for SESAM to define the SQL environment.

Examples under `SHOW` illustrate how you can obtain information on permanent database objects with `SELECT` queries on SESAM system view.

2.5.1 PARAMETER DYNAMIC - Define dynamic parameters

You use the `PARAMETER DYNAMIC` statement to define the SQL environment for the DRIVE session (TIAM session or UTM conversation). This statement can be used in both program and interactive modes.

You will find a complete description of the operands of the `PARAMETER DYNAMIC` statement in the “Directory of DRIVE Statements” [3].

The following applies to static programs:

If no values are set for the operands `AUTHORIZATION`, `CATALOG` and `SCHEMA` in either an `OPTION` statement in the program or the `OPTION` clause of the `COMPILE` statement, the values set or predefined with the `PARAMETER DYNAMIC` statement are used. If the values for `CATALOG` and `SCHEMA` have been omitted, a blank is used as the value. If `AUTHORIZATION` has not been specified, compilation is aborted with the DRIVE error message `DRI0525`.

The following applies to dynamic programs:

If `CATALOG`, `SCHEMA` and `AUTHORIZATION` are not defined with `SET` statements, the values from the parameter settings are used.

The `PARAMETER DYNAMIC` statement does not open a transaction.

```
PARAMETER DYNAMIC [ AUTHORIZATION=authorization_id |
                    CATALOG=sesdb_name |
                    SCHEMA=schema_name ]
```

AUTHORIZATION

This operand can only be specified in a transactionless state, otherwise DRIVE/WINDOWS issues error message `DRI0084`.

Specifies the current authorization identifier *authorization_id* (max. 18 characters) for new-style access to SESAM databases.

Under MS-Windows and SINIX, you must also specify the dynamic parameter DBSYSTEM=SESAMSQL. In BS2000, the default value is automatically the loaded database version.

This operand is also evaluated during compilation if OPTION AUTHORIZATION is omitted.

Default value: blank in DRIVE/WINDOWS, D0USER in SESAM

CATALOG

Specifies a SESAM database.

Under MS-Windows and SINIX, you must also specify the dynamic parameter DBSYSTEM=SESAMSQL. In BS2000, the default value is automatically the loaded database version.

CATALOG=*sesdb_name* is valid for SQL statements that are entered in interactive mode. In programs, this operand is only valid for dynamic SQL statements.

This operand is also evaluated during compilation if no OPTION CATALOG is specified.

Default value: empty string in DRIVE/WINDOWS, blank in SESAM.

SCHEMA

Name of an SQL schema that is accessed if no name was specified in the SQL statements.

SCHEMA=*schema_name* is valid for SQL statements that are entered in interactive mode. In programs, this operand is only valid for dynamic SQL statements. This operand is also evaluated during compilation if OPTION SCHEMA is omitted.

Default value: empty string in DRIVE/WINDOWS, blank in SESAM.

2.5.2 OPTION - Control program compilation

OPTION controls the compilation run of a DRIVE program. Compiler options are specified for compiling DRIVE programs into intermediate or object code. You will find a complete description of the operands for the OPTION statement in the “Directory of DRIVE Statements” [3].

In BS2000 and SINIX, OPTION can be specified in the source program (before the PROCEDURE and DECLARE TYPE statements) or as an operand of the COMPILE statement. In MS-Windows, OPTION **must** be specified in the source program before the PROCEDURE and DECLARE TYPE statements.

Default parameter settings are overwritten in the source program by OPTION specifications. OPTION specifications in the source program are overwritten by OPTION specifications made in COMPILE or compilation options that you enter using menus.

The OPTION values valid for compilation are stored in the intermediate code so that they are also valid for execution. The operands AUTHORIZATION, CATALOG and SCHEMA are mandatory for compilation. If no values are specified, DRIVE/WINDOWS uses the SQL environment values set with the PARAMETER DYNAMIC statement.

```
OPTION { AUTHORIZATION=authorization_id |  
        CATALOG=sesdb_name |  
        SCHEMA=schema_name }
```

AUTHORIZATION

Specifies the authorization identifier *authorization_id* for a SESAM database. The authorization identifier specified here cannot be modified with a SET SESSION AUTHORIZATION statement.

This means that if the program is compiled with this operand, any modification of the current authorization key with SET or PARAMETER only affects dynamic SQL statements upon execution.

Default value: The current setting from PARAMETER DYNAMIC AUTHORIZATION. If this does not exist, i.e. is an empty string because no PARAMETER DYNAMIC AUTHORIZATION has yet been issued during the DRIVE session, DRIVE/WINDOWS aborts compilation with error message DRI0525.

CATALOG

Specifies the default value for a SESAM database.

If this operand is omitted, the catalog name last specified for the parameter settings is used. If this does not exist, compilation is aborted.

Default value: The current setting specified for PARAMETER DYNAMIC CATALOG or, if this does not exist, a blank. If CATALOG is preset to a blank, the names of schemas, tables and integrity constraints in static SQL statements must be qualified with a catalog name.

SCHEMA

If, in a program, the schema name is not specified in SQL statements, the *schema_name* specified here is used at execution time. If this operand is omitted, the schema name last specified for the parameter settings is used. If this does not exist, compilation is aborted.

Default value: The current setting specified for PARAMETER DYNAMIC SCHEMA or, if this does not exist, a blank. If SCHEMA is preset to a blank, the names of tables and integrity constraints in static SQL statements must be qualified with a schema name.

2.5.3 SESAM V2 settings in foreign environments

The table below illustrates the effect of the parameter and option settings in the SESAM V1, UDS and INFORMIX environments:

	PAR DYN CATALOG	PAR DYN SCHEMA	PAR DYN AUTHORIZATION	OPTION CATALOG	OPTION SCHEMA	OPTION AUTHORIZATION
SESAM V1	rejected with DRI0229	only permitted in interactive mode; <i>schema_name</i> must be the name of a base relation; in interactive mode, only this base relation can be referenced without <i>schema_name</i>	rejected with DRI0229	ignored	<i>schema_name</i> must be the name of a base relation; in the program with OPTION SCHEMA only this base relation can be referenced without <i>schema_name</i>	ignored
UDS	rejected with DRI0229	only permitted in interactive mode	rejected with DRI0229	ignored	<i>schema_name</i> must be the name of a relational view of a CODASYL subschema	ignored
INFORMIX	rejected with DRI0229	rejected with DRI0229	rejected with DRI0229	ignored	ignored	ignored

2.5.4 SHOW - Output information about metadata

This statement is not supported for SESAM/SQL V2. You can obtain information via the SESAM Utility Monitor or by means of SELECT queries on the SESAM system view (see the "SESAM/SQL-Server, SQL Reference Manual: Part 1" [18]). You can store these SELECT queries as, for example, a DRIVE COPY member for outputting metadata on permanent database objects. You cannot output information on temporary objects, i.e. cursors and temporary views.

Example 1

You must perform the following steps if you want to query schema information:

- set the database with OPTION
- declare a cursor on the attributes of the system view that you want to query
- read in the cursor into an auxiliary variable with a loop
- and output this with DISPLAY.

```
OPTION CATALOG=TLDBSQL2 SCHEMA=TLDB AUTHORIZATION=TLAB;
PROC "schema_query";
DCL SCHEMATA_C FOR S CATALOG_NAME, SCHEMA_NAME, SCHEMA_OWNER
    FROM INFORMATION_SCHEMA.SCHEMATA
    WHERE SCHEMA_NAME LIKE 'TL%';
DCL VAR &SCHEMATA_VAR LIKE CURSOR SCHEMATA_C;
CYCLE SCHEMATA_C INTO &SCHEMATA_VAR.*;
    DISPLAY FORM LINE NAMES VALUES &SCHEMATA_VAR;
END CYCLE;
COMMIT WORK;
END PROC;
```

Example 2

Declare the following cursor in order to query the structure of a base table:

```
OPTION CATALOG=TLDBSQL2 SCHEMA=TLDB AUTHORIZATION=TLAB;
PROC "table_query";
DCL TAB_COL_C FOR S TABLE_NAME, COLUMN_NAME, COLUMN_DEFAULT, DATA_TYPE,
    CHARACTER_MAXIMUM_LENGTH, NUMERIC_PRECISION, NUMERIC_SCALE,
    DATETIME_PRECISION
    FROM INFORMATION_SCHEMA.BASE_TABLE_COLUMNS
    WHERE TABLE_NAME LIKE 'TLDB%';
```


Example 3

Declare the following cursor in order to query permanent views of a base table:

```
OPTION CATALOG=TLDBSQL2 SCHEMA=TLDB AUTHORIZATION=TLAB;  
PROC "view_query";  
DCL VIEWS_C CURSOR FOR S * FROM INFORMATION_SCHEMA."TABLES"  
    WHERE TABLE_TYPE = 'VIEW' AND TABLE_NAME LIKE 'TLDB%';
```

2.6 Examples and sample database

This section contains the SESAM V1 databases from the program language for DRIVE/ WINDOWS V1.1 and DRIVE V6 before and after migration to a SESAM V2 database, as well as the DRIVE programs that access them.

2.6.1 Sample tables before and after migration

This section describes the structure and the content of the sample data. This data exists as SESAM/SQL V1 databases (see the manual “DRIVE Programming Language V1.1”). The database backup (DB-SIB) MITARBEITER was migrated to SESAM V2 in section “Database structure and migration of SESAM V1 tables” on page 10.

Three base tables are used for the sample data. These base tables contain the employees, the departments and the projects within a company.

This section includes a list of the data in each base table.

Structure of the base tables

The following three base tables are used in the examples:

- “ABTEILUNG”
- “MITARBEITER”
- “PROJEKT”

ABTEILUNG

The base table “ABTEILUNG“ contains information on the departments within the company. It is generated and loaded with a DRIVE program (see page 50).

Description of the individual columns in the base table “ABTEILUNG”:

Column	Data type	Description
ABTEILUNG_NR	CHARACTER (4)	Primary key with the name KEY_ABT_NR
BEZEICHNUNG	CHARACTER (10)	Name of the department
STANDORT	CHARACTER (20)	Location of the department
LEITER	CHARACTER (8)	Personnel number of the department manager

MITARBEITER

The base table "MITARBEITER" contains information on the employees of the company. They are migrated with the Utility Monitor (see section "Database structure and migration of SESAM V1 tables" on page 10).

Description of the individual rows in the base table "MITARBEITER":

Column	Data type	Description
SESAM V1: COMP_PERS_NR	CHARACTER (8)	SESAM V1: Personnel number of the employee, compound key consisting of ABT_MIT_NR and LFD_NR, an automatic count field SESAM V2: This compound key with a count field cannot be transferred directly to a SESAM V2 database, it is defined there as a primary key constraint involving two columns instead
ABT_MIT_NR	CHARACTER (4)	Department number of the employee
LFD_NR	INTEGER	SESAM V1: Sequence number of the employee (automatic count field) SESAM V2: CONSTRAINT of the type PRIMARY KEY
NACHNAME	CHARACTER (20)	Last name of the employee
VORNAME	CHARACTER (20)	First name of the employee
LAND	CHARACTER (3)	Country
STRASSE	CHARACTER (26)	Street
PLZ	CHARACTER (10)	Zip code
ORT	CHARACTER (20)	City
GEHALT	NUMERIC (7,2)	Salary of the employee
SPRACHEN SESAM V2: SPRACHEN (4)	CHARACTER (3)	Languages spoken by the employee (multiple field with 4 occurrences)
ABT_LEITER	CHARACTER (8)	Personnel number of the department manager
PROJ_MIT	CHARACTER (6)	Contains the primary key of the project on which the employee is working

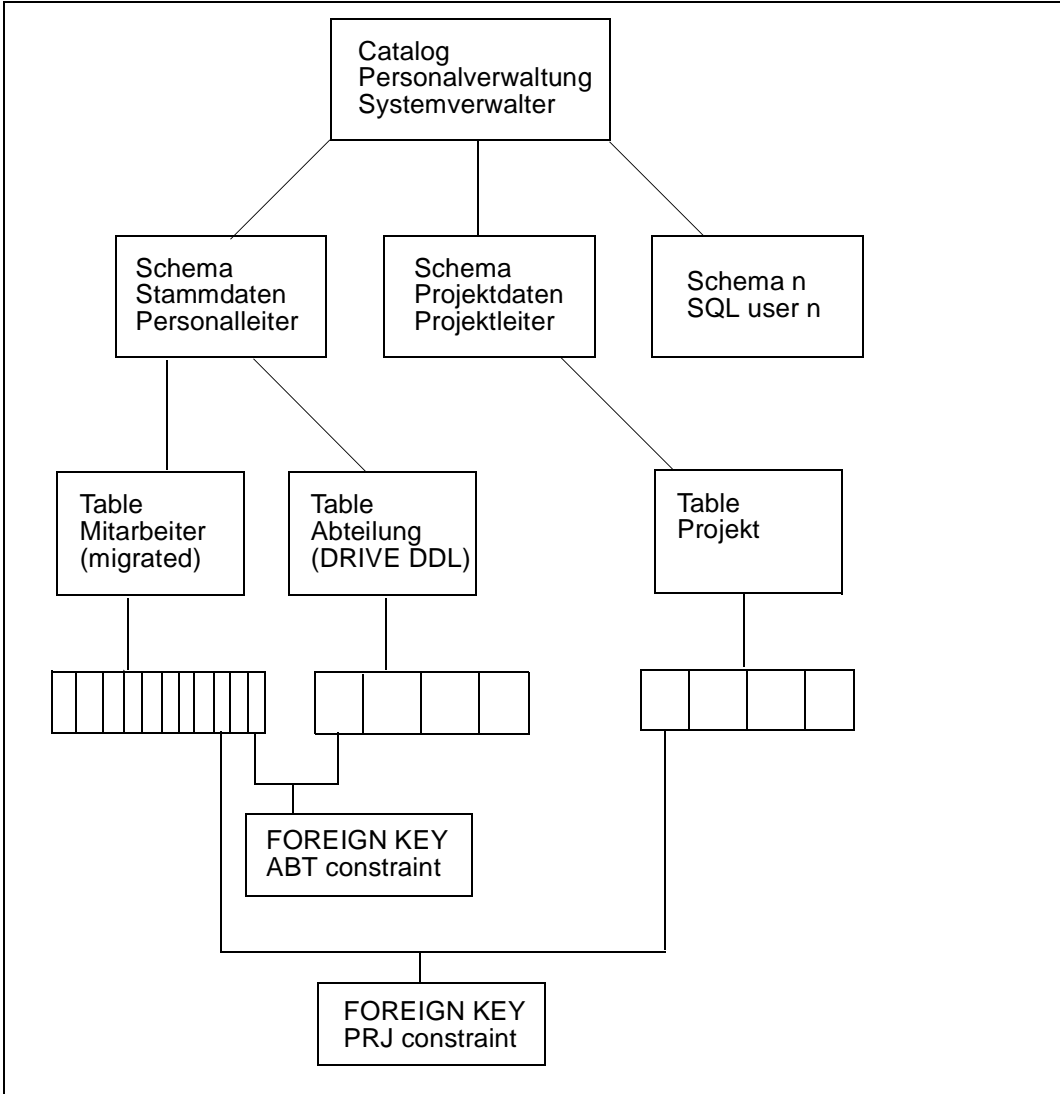
PROJEKT

The base table "PROJEKT" contains information on the current projects within the company, including the personnel number of the project manager and the available budget. Description of the individual columns in the base table "PROJEKT":

Column	Data type	Description
PROJEKT_NR	CHARACTER (6)	Primary key with the name KEY_PROJEKT_NR
BEZEICHNUNG	CHARACTER (10)	Name of the project
BUDGET	NUMERIC (12,2)	Available budget
PROJ_LEITER	CHARACTER (8)	Personnel number of the project manager

Relationships between the base tables

Overview of the SESAM V2 sample database



Data in the base tables

ABTEILUNG

ABTEILUNG_NR	BEZEICHNUNG	STANDORT	LEITER
0106	Zentrale	Muenchen	01060002
0107	Marketing	Kiel	01070004
0108	Transport	Hamburg	01080008
0109	Technik	Hannover	01090004
0110	Personal	Muenchen	01100002
0111	Ausland	Muenchen	01110004
0112	Forschung	Muenchen	01120003

MITARBEITER

ABT MIT NR	LFD NR	NACH- NAME	VORNAME	LAND	STRASSE	PLZ	ORT	GEHALT	SPRACHEN	ABT LEITER	PROJ MIT
0106	1	Sennert	Gustl	FRG	Petuelring 26	8000	Muenchen	0580000	ENG	01060002	000106
0106	2	Bergmann	Susanne	FRG	Alter Platz 3	8000	Muenchen	0690000	ENG	01060002	
0106	3	Berghoff	Ruth	FRG	Isartor 7	8000	Muenchen	0300000		01060002	
0106	4	Bergner	Erich	FRG	Sonnemannstr. 512A	8000	Muenchen	0450000		01060002	000106
0107	1	Olsen	Mattes	FRG	Kiekendamm 13b	2300	Kiel	0400000	ENG	01070004	000108
0107	2	Mattsen	Gunter	FRG	Spোকemansswatt	2300	Kiel	0480000	ENG	01070004	000108
0107	3	Nonnen	Paul	FRG	Jollischweg 2b	2300	Kiel	0450000		01070004	
0107	4	Dormagen	Siegfried	FRG	Gundelstr.91	2322	Hemlstorf	0550000	FRASPA	01070004	000108
0108	1	Winterberg	Hannelore	FRG	Mitterstr. 8	2000	Hamburg	0350000	ENG	01080008	
0108	2	Mitscherlich	Hermann	FRG	Hansstadtallee 88	2000	Hamburg	0270000	ENG	01080008	
0108	3	Grenzer	Wilhelm	FRG	Am Kiefernpfad 4	2081	Heist	0230000	FRA	01080008	
0108	4	Paarungen	Morten	FRG	Ollenvatt 6	2050	Hamburg 80	0320000	ENG	01080008	
0108	5	Andermatt	Sylvia	FRG	Hohe Strasse 5	2000	Hamburg	0510000		01080008	000107
0108	6	Schmidt	Bettina	FRG	Hansaring 388	2000	Hamburg	0620000	ENG	01080008	000107
0108	7	Pollinger	Nora	FRG	Goethestr.32	2050	Hamburg 80	0400000	ENG	01080008	
0108	8	Mauer	Rita	FRG	Wiesenstr. 381	2000	Hamburg	0650000	ENG	01080008	
0109	1	Mitscher	Fred	FRG	Wiesenstr. 96	3000	Hannover	0490000	FRA	01090004	000109
0109	2	Jansen	Jutta	FRG	Giesestr. 17	3510	Hannover-Muenden	0470000		01090004	
0109	3	Zimmermann	Peter Johannes	FRG	Berger Markt 119c	3000	Hannover	0580000	ENGSPAFRA	01090004	
0109	4	Sennelaub	Andrea	FRG	Simonenpfad 57	3000	Hannover	0650000	ENG	01090004	000109
0109	5	Maier	Bernd	FRG	Kreuzgasse 8	3000	Hannover	0350000	ENG	01090004	
0109	6	Maler	Guenther	FRG	Rohrdamm 2	3000	Hannover	0290000	ENGFRA	01090004	
0110	1	Lorenz	Erna	FRG	Kiesgraben 61	8000	Muenchen	0330000	ENG	01100002	000111
0110	2	Ammerl	Sepp	FRG	Luxemburger Str. 427	8000	Muenchen	0570000	SPA	01100002	000111
0110	3	Dollinger	Johanna	FRG	Pferdesaalstr. 54	8000	Muenchen	0280000	FRA	01100002	000111
0111	1	Manson	Dr. Jack	FRG	Marienburg 1	8000	Muenchen	1200000	GER	01110004	000110
0111	2	van Claeren	Sandra	CH	Berner Weg 59	CH-3000	Bern 33	0590000	ENG	01110004	
0111	3	Miller	James	USA	5009 Beach Boulevard	CA 93440	Los Alamos	0960000	FRA	01110004	000110
0111	4	von Haalen	Walter	USA	30 Third Avenue	NY 11217	New York	1000000	GERSPAFRA	01110004	000110
0112	1	Plenzner	Wolfgang	FRG	Sesemieweg 104	8000	Muenchen	0360000	ENG	01120003	
0112	2	Heimlott	Hansi	FRG	Kanonenstr. 15	8000	Muenchen	0350000		01120003	
0112	3	Sindlinger	Max	FRG	Hollermesse 35	8000	Muenchen	0690000	ENGSPA	01120003	

PROJEKT

PROJEKT_NR	BEZEICHNUNG	BUDGET	PROJ_LEITER
000106	Zeus	000035000000	01060001
000107	Hera	000015000000	01080006
000108	Poseidon	000061000000	01070004
000109	Athene	000050000000	01090004
000110	Aphrodite	000008000000	01110004
000111	Prometheus	000002000000	01100002

2.6.2 Command file for migration

1. Create catalog

You create a catalog with the utility statement CREATE CATALOG ("PERSONALVERWALTUNG" in the example).

The result is a BS2000 file PERSONALVERWALTUNG.CATALOG under the SESAM ID. The universal user "SYSTEMVERWALTER" is the owner of the new SQL2 database.

Example

Command file "UTI.PERSONALVERWALTUN.CATALOG" , which creates the catalog "PERSONALVERWALTUNG" (see page 10):

```
*
SQL CREATE CATALOG "PERSONALVERWALTUNG" -
    CODE_TABLE "'EBCDIC_DF_03"' -
    CATALOG_SPACE -
    PRIMARY 200 -
    SECONDARY 36 -
    PCTFREE 40 -
    SHARE -
    NO DESTROY -
    NO LOG -
    USER "SYSTEMVERWALTER"
*
SQL COMMIT WORK
END
```

2. Define SQL users

You define privileged SQL users with the CREATE USER ... statement ("PERSONALLEITER" and "PROJEKTLEITER" in the example) since only one SQL user can access a database.

Example

Command file "UTI.PERSONALVERWALTUNG.USER", which defines the SQL users (see page 10):

```
*
SQL SET CATALOG ' "PERSONALVERWALTUNG' "
SQL SET SESSION AUTHORIZATION ' "SYSTEMVERWALTER' "
*
SQL CREATE USER "PERSONALLEITER" AT CATALOG "PERSONALVERWALTUNG"
SQL CREATE USER "PROJEKTLEITER" AT CATALOG "PERSONALVERWALTUNG"
SQL CREATE USER "OLDSTYLE-ADMIN" AT CATALOG "PERSONALVERWALTUNG"
SQL COMMIT WORK
END
```

3. Generate system entry for the SQL user

An SQL user communicates with SESAM V2 via the SYSTEM_USER. In order to do this, the DRIVE application must be loaded under the same ID under which the system entries were generated (TIAM in the example).

Example

Command file "UTI.PERSONALVERWALTUNG.SYSTEMUSER", which defines the system entries (see page 10):

```
*
SQL SET CATALOG ' "PERSONALVERWALTUNG" '
SQL SET SESSION AUTHORIZATION ' "SYSTEMVERWALTER" '
*
SQL CREATE SYSTEM_USER (*, , 'STMQM235' ) -
    FOR "SYSTEMVERWALTER" -
    AT CATALOG "PERSONALVERWALTUNG"
*
SQL CREATE SYSTEM_USER (*, , 'STMQM235' ) -
    FOR "PERSONALLEITER" -
    AT CATALOG "PERSONALVERWALTUNG"
*
SQL CREATE SYSTEM_USER (*, , 'STMQM235' ) -
    FOR "PROJEKTLEITER" -
    AT CATALOG "PERSONALVERWALTUNG"
```



```

*
SQL CREATE SYSTEM_USER (*,, 'STMQM235') -
    FOR "OLDSTYLE-ADMIN" -
    AT CATALOG "PERSONALVERWALTUNG"
*
SQL COMMIT WORK
END

```

4. Create schemas

You use the `CREATE SCHEMA` statement to create schemas (“STAMMDATEN“ and “PROJEKTDATEN“ in the example), which are assigned to the SQL users (“PERSON-ALLEITER“ and “PROJEKTLLEITER“) as their owners (`CREATE SCHEMA ... AUTHORIZATION ...`); these owners manage the schemas.

Example

Command file “UTI.PERSONALVERWALTUNG.SCHEMA“, which creates the schemas (see page 10):

```

*
SQL SET CATALOG ' "PERSONALVERWALTUNG" '
SQL SET SESSION AUTHORIZATION ' "SYSTEMVERWALTER" '
*
SQL CREATE SCHEMA "STAMMDATEN" -
    AUTHORIZATION "PERSONALLEITER"
*
SQL CREATE SCHEMA "PROJEKTDATEN" -
    AUTHORIZATION "PROJEKTLLEITER"
*
SQL CREATE SCHEMA "OLDSTYLE-DATEN" -
    AUTHORIZATION "OLDSTYLE-ADMIN"
*
SQL COMMIT WORK
END

```

2.6.3 DRIVE DDL programs for the table ABTEILUNG

The following DRIVE program "DRI.TABLE.ABTEILUNG" creates the base table "ABTEILUNG" as a SESAM V2 table in the schema "STAMMDATEN" (catalog "PERSONALVERWALTUNG" (see page 11).

```

OPTION AUTHORIZATION=" PERSONALLEITER"
        CATALOG      =" PERSONALVERWALTUNG"
        SCHEMA       =" STAMMDATEN";

/* */
PROC "DRI.TABLE.ABTEILUNG";
/* */
/* ----- table, column, unique (of the type primary key) -----*/
/* */
CREATE TABLE "ABTEILUNG"
(
  "ABTEILUNG_NR" CHAR(4) ,
  "BEZEICHNUNG" CHAR(10),
  "STANDORT"     CHAR(20),
  "LEITER"       CHAR(8) ,
  CONSTRAINT "KEY_ABT_NR" PRIMARY KEY
    ( "ABTEILUNG_NR")
) ;
COMMIT WORK ;
END PROC;

```

The following DRIVE program loads the table "ABTEILUNG" with records:

```

OPTION AUTHORIZATION=" PERSONALLEITER" CATALOG=" PERSONALVERWALTUNG"
        SCHEMA="STAMMDATEN" ;

/*      */
PROC "DRI.LOAD.ABTEILUNG";
DCL VAR &V1 LIKE TABLE ABTEILUNG;
/*      */
SET &V1 = <
          '0106'           ,
          'ZENTRALE '     ,
          'MUENCHEN      ' ,
          '01060002'
        > ;

/*      */
INSERT INTO ABTEILUNG VALUES (&V1.*) ;
DISPLAY FORM LINE RETURN &V1 ;
/*-----*/

```

```

SET &V1 = <
          '0107'           ,
          'MARKETING '    ,
          'KIEL           ' ,
          '01070004'
        > ;
/*          */
INSERT INTO ABTEILUNG VALUES (&V1.*) ;
DISPLAY FORM LINE RETURN &V1 ;
/*-----*/
SET &V1 = <
          '0108'           ,
          'TRANSPORT '    ,
          'HAMBURG        ' ,
          '01080008'
        > ;
/*          */
INSERT INTO ABTEILUNG VALUES (&V1.*) ;
DISPLAY FORM LINE RETURN &V1 ;
/*-----*/
SET &V1 = <
          '0109'           ,
          'TECHNIK '       ,
          'HANNOVER        ' ,
          '01090004'
        > ;
/*          */
INSERT INTO ABTEILUNG VALUES (&V1.*) ;
DISPLAY FORM LINE RETURN &V1 ;
/*-----*/
SET &V1 = <
          '0110'           ,
          'PERSONAL '      ,
          'MUENCHEN        ' ,
          '01100002'
        > ;
/*          */
INSERT INTO ABTEILUNG VALUES (&V1.*) ;
DISPLAY FORM LINE RETURN &V1 ;
/*-----*/
SET &V1 = <
          '0111'           ,
          'AUSLAND '       ,
          'MUENCHEN        ' ,
          '01110004'
        > ;
/*          */

```

```

INSERT INTO ABTEILUNG VALUES (&V1.*) ;
DISPLAY FORM LINE RETURN &V1 ;
/*-----*/
SET &V1 = <
            '0112'                ,
            'FORSCHUNG '          ,
            'MUENCHEN            ' ,
            '01120003'           ;
            > ;
/*          */
INSERT INTO ABTEILUNG VALUES (&V1.*) ;
DISPLAY FORM LINE RETURN &V1 ;
/*-----*/
COMMIT WORK ;
END PROC;

```

2.6.4 DRIVE DDL program for access permissions and foreign keys

The following DRIVE program "LOAD_FOREIGNKEY" grants all necessary privileges and generates foreign keys (see page 11).

```

PROC "LOAD_FOREIGNKEY" ;
/* */
/* GRANTING OF THE PRIVILEGES SELECT, DELETE, INSERT, UPDATE and */
/* REFERENCES TO ALL SQL USERS OF THE DATABASE */
/* */
EXEC 'SET SESSION AUTHORIZATION ''PERSONALLEITER''';
EXEC 'SET CATALOG ''PERSONALVERWALTUNG''';
EXEC 'SET SCHEMA ''STAMMDATEN''';
/* */
EXEC 'GRANT ALL PRIVILEGES ON TABLE MITARBEITER TO PUBLIC ' ;
EXEC 'GRANT ALL PRIVILEGES ON TABLE ABTEILUNG TO PUBLIC ' ;
EXEC 'COMMIT WORK' ;
/* */
EXEC 'SET SESSION AUTHORIZATION ''PROJEKTLEITER''';
EXEC 'SET CATALOG ''PERSONALVERWALTUNG''';
EXEC 'SET SCHEMA ''PROJEKTDATEN''';
/* */
EXEC 'GRANT ALL PRIVILEGES ON TABLE PROJEKT TO PUBLIC ' ;
EXEC 'COMMIT WORK' ;
/* */
EXEC 'COMMIT WORK' ;
/* */
END PROC;

```

```

/* DEFINE FOREIGN KEY FOR KEY PROJEKT.PROJEKT_NR AND FOR      */
/* KEY ABTEILUNG.ABTEILUNG_NR                                  */
/* */
EXEC 'SET SESSION AUTHORIZATION 'PERSONALLEITER'' ;
EXEC 'SET CATALOG 'PERSONALVERWALTUNG'' ;
EXEC 'SET SCHEMA 'STAMMDATEN'' ;
/* */
EXEC 'ALTER TABLE MITARBEITER ADD CONSTRAINT "PRJ_CONSTRAINT"  '||'|
      FOREIGN KEY ("PROJ_MIT")                                '||'|
      REFERENCES "PROJEKTDATEN"."PROJEKT" ("PROJEKT_NR")' ;
/* */
EXEC 'ALTER TABLE MITARBEITER ADD CONSTRAINT "ABT_CONSTRAINT"  '||'|
      FOREIGN KEY ("ABT_MIT_NR")                              '||'|
      REFERENCES "STAMMDATEN"."ABTEILUNG" ("ABTEILUNG_NR")' ;
/* */
EXEC 'COMMIT WORK' ;
END PROC;

```

2.6.5 Sample programs

The DRIVE V1.1 programs (see the manual “DRIVE Programming Language V1.1”) also execute under version 2.1 if you define the SQL environment with the following statements:

```

PARAMETER DYNAMIC AUTHORIZATION=PERSONALLEITER;
PARAMETER DYNAMIC CATALOG=PERSONALVERWALTUNG;
PARAMETER DYNAMIC SCHEMA=STAMMDATEN;

```

These **PARAMETER** statements are valid for static and dynamic statements if no **OPTION** statement with these operations has been included in the programs. Otherwise, the **OPTION** statement must be specified for the static statements and **SET** statements for the dynamic statements in order to set **AUTHORIZATION**, **CATALOG** and **SCHEMA**.

For remote access, you must also set the database system, e.g. with the statement:

```
OPTION DBSYSTEM=SESAMSQL
```

Program name: MITARBEITER.HAUPT

```

PROCEDURE MITARBEITER;
COPY MITVAR;
DECLARE VARIABLE &ENDE2 CHAR(1);
DCL SCREEN FHSBILD;
DECLARE FORM NEUBILD
  TTITLE NL 1,
    TAB 30,'M I T A R B E I T E R',NL 2,
    ' NEUAUFNAHME',
    TAB 60,'FORMULAR 02',NL 1,
    ' BEENDEN ( E ) : ',RETURN &ENDE2,NL 1,
    ' ','-'(70),NL 3,
    ' ABT_MIT_NR      : ',RETURN &ABT_MIT_NR,NL 1,
    ' NACHNAME       : ',RETURN &ENACHNAME,NL 1,
    ' VORNAME        : ',RETURN &EVORNAME,NL 1,
    ' LAND           : ',RETURN &ELAND,NL 1,
    ' STRASSE        : ',RETURN &ESTRASSE,NL 1,
    ' PLZ            : ',RETURN &EPLZ,NL 1,
    ' ORT            : ',RETURN &EORT,NL 1,
    ' GEHALT         : ',RETURN &EGEHALT,NL 1,
    ' SPRACHEN       : ',RETURN &ESPRACHEN(1),' ','',
    '                : ',RETURN &ESPRACHEN(2),' ','',
    '                : ',RETURN &ESPRACHEN(3),' ','',
    '                : ',RETURN &ESPRACHEN(4),' ','',NL 1,
    ' ABT_LEITER     : ',RETURN &EABT_LEITER,NL 1,
    ' PROJ_MIT       : ',RETURN &EPROJ_MIT,NL 1
  BTITLE ' ','-'(70),NL 1;
SUBPROCEDURE MITARBEITERAUFNAHME;
SET &ENDE2 = ' ';
CYCLE;
  DISPLAY NEUBILD;
  IF &ENDE2 = 'E'
    THEN BREAK SUBPROCEDURE;
  END IF;
  INSERT INTO MITARBEITER VALUES ( &ABT_MIT_NR,*,&AUFNAHMESATZ.*);
  IF &DML_STATE = 'SQL ERROR' THEN
    DISPLAY FORM &DML_STATE,
      ': DER DATENSATZ WURDE NICHT AUFGENOMMEN (' ,
      &SQL_STATE, ') - DUE-TASTE';
  ELSE IF &DML_STATE <> 'OK' THEN
    SEND MESSAGE 'FEHLER ',&SQL_STATE,
      ' BEI DATENSATZAUFNAHME - DUE-TASTE';
    ELSE COMMIT WORK;
    SEND MESSAGE
      'DER DATENSATZ WURDE AUFGENOMMEN - DUE-TASTE';
  END IF;
END IF;

```

```

END CYCLE;
END SUBPROCEDURE;
WHenever &DML_STATE CONTINUE;
CYCLE;
  DISPLAY FHSBILD;
  IF &WAHL = '0' THEN BREAK CYCLE;
  END IF;
  IF &WAHL = '1' THEN CALL MITARBEITERAUFNAHME; SET &WAHL = ' ';
  END IF;
  IF &WAHL = '2' THEN CALL MITKORR; SET &WAHL = ' ';
  END IF;
  IF &WAHL = '3' THEN CALL MITLOES; SET &WAHL = ' ';
  END IF;
  IF &WAHL = '4' THEN CALL MITANZ; SET &WAHL = ' ';
  END IF;
  IF &WAHL = '5' THEN CALL MITDRUCK; SET &WAHL = ' ';
  END IF;
END CYCLE;
ROLLBACK WORK;
END PROCEDURE;

```

Name of the COPY member: MITVAR

```

DECLARE VARIABLE &FRAGE PERMANENT CHAR(1) INIT 'N';
DECLARE MITARBEITER CURSOR FOR S ALL * FROM MITARBEITER;
DECLARE DRUCKCURSOR CURSOR FOR S ABT_MIT_NR,LFD_NR,NACHNAME,VORNAME,
      GEHALT,ABT_LEITER,PROJ_MIT FROM MITARBEITER;
DECLARE VARIABLE &MITARBEITERSATZ LIKE CURSOR MITARBEITER,
      &INDEX INT INIT 1,
      &NUMMER INT,
      &VGL CHAR(4),
      &DATUM DATE
      MASK 'Q(10)'' ,DER ''ZD''.'''R(10)''.'''YYYY'',
      &ZEIT TIME
      MASK 'ZH'' UHR ''ZI'',
1 &SAETZE,
2 AABT_MIT_NR CHAR(4) INIT '0000',
2 AUFNAHMESATZ,
3 ENACHNAME CHAR(20),
3 EVORNAME CHAR(20) ,
3 ELAND CHAR(3),
3 ESTRASSE CHAR(26),
3 EPLZ CHAR(10),
3 EORT CHAR(20),
3 EGEHALT NUM(7,2) CHECK &EGEHALT < 20000 MESSAGE
  '          DAS GEHALT IST ZU HOCH ! ',
3 ESPRACHEN(4) CHAR(3) INIT '---'

```

```

CHECK &ESPRACHEN <> ' ' MESSAGE
'      GEBEN SIE ''---'' ALS LEERSTELLE EIN',
3 EABT_LEITER CHAR(8),
3 EPROJ_MIT CHAR(6) INIT '-----'
CHECK &EPROJ_MIT <> ' ' MESSAGE
'      GEBEN SIE ''-----'' ALS LEERSTELLE EIN',
2 DRUCKSATZ,
3 ABT_MIT_NR_ CHAR(4) REDEFINES &AABT_MIT_NR,
3 LFD_NR_ INT,
3 NACHNAME_ CHAR(20) ,
3 VORNAME_ CHAR(20) ,
3 GEHALT_ NUM(7,2) MASK 'ZZZZ9P99'' DM ''',
3 ABT_LEITER_ CHAR(8) ,
3 PROJ_MIT_ CHAR(6) ,
&ABFRAGEBEFEHL CHAR(15) INIT 'DISPLAY FORM ',
&ABFRAGETEXT1 CHAR(74) INIT
'NL 14,TAB 5, 'GEBEN SIE DIE ABT_MIT_NR EIN : ',RETURN &VGL,',
&ABFRAGETEXT2 CHAR(74) INIT
'NL 2,TAB 5, 'GEBEN SIE DIE LAUFENDE NUMMER EIN : ',RETURN &NUMMER;',
&FEHLERMELDUNG CHAR(74) INIT
'SEND MESSAGE '' DER DATENSATZ IST NICHT VORHANDEN. DUE-TASTE'';',
&DRUCKTEXT CHAR(74) INIT
'NL 15,TAB 15, 'SOLL JETZT GEDRUCKT WERDEN ? (J/N) : ',RETURN &FRAGE';

```


Name of the external subprogram: MITKORR

```

PROCEDURE MITARBEITERKORREKTUR;
COPY MITVAR;
DECLARE FORM KORRBILD
  TTITLE NL 1,
    TAB 30,'M I T A R B E I T E R',NL 2,
    ' KORREKTUR VON MITARBEITERDATEN',TAB 60,'FORMULAR 03',NL 2,
    ' ','-'(70),NL 1,
    ' ABT_MIT_NR      : ',&ABT_MIT_NR,NL 1,
    ' LFD_NR         : ',&LFD_NR,NL 1,
    ' NACHNAME       : ',RETURN &NACHNAME,NL 1,
    ' VORNAME        : ',RETURN &VORNAME,NL 1,
    ' LAND           : ',RETURN &LAND,NL 1,
    ' STRASSE        : ',RETURN &STRASSE,NL 1,
    ' PLZ            : ',RETURN &PLZ,NL 1,
    ' ORT            : ',RETURN &ORT,NL 1,
    ' GEHALT         : ',RETURN &GEHALT,NL 1,
    ' SPRACHEN       : ',RETURN &SPRACHEN(1),' ','',
                        RETURN &SPRACHEN(2),' ','',
                        RETURN &SPRACHEN(3),' ','',
                        RETURN &SPRACHEN(4),' ','',NL 1,
    ' ABT_LEITER     : ',RETURN &ABT_LEITER,NL 1,
    ' PROJ_MIT       : ',RETURN &PROJ_MIT,NL 1
  BTITLE ' ','-'(70),NL 1;
WHENEVER &DML_STATE CONTINUE;
EXEC CONCAT(&ABFRAGEBEFEHL,CONCAT(&ABFRAGETEXT1,&ABFRAGETEXT2));
SELECT ALL * INTO &MITARBEITERSATZ.* FROM MITARBEITER
  WHERE (ABT_MIT_NR = &VGL) AND (LFD_NR = &NUMMER);
IF &DML_STATE <> 'OK' THEN
  SEND MESSAGE 'FEHLER ',&SQL_STATE;
  EXEC &FEHLERMELDUNG;
  SET &VGL = ' ';
  BREAK PROCEDURE;
END IF;
CYCLE WHILE &INDEX < 5;
  IF &SPRACHEN(&INDEX) IS NULL THEN
    SET &SPRACHEN(&INDEX) = '----';
  END IF;
  SET &INDEX = &INDEX + 1;
END CYCLE;
SET &INDEX = 1;
IF &PROJ_MIT IS NULL THEN
  SET &PROJ_MIT = '-----';
END IF;

```

```

DISPLAY KORRBILD;
UPDATE MITARBEITER
  SET  NACHNAME  = &NACHNAME,
       VORNAME   = &VORNAME,
       LAND      = &LAND,
       STRASSE   = &STRASSE,
       PLZ       = &PLZ,
       ORT       = &ORT,
       GEHALT    = &GEHALT,
       SPRACHEN(1)= &SPRACHEN(1),
       SPRACHEN(2)= &SPRACHEN(2),
       SPRACHEN(3)= &SPRACHEN(3),
       SPRACHEN(4)= &SPRACHEN(4),
       ABT_LEITER = &ABT_LEITER,
       PROJ_MIT  = &PROJ_MIT
  WHERE (ABT_MIT_NR = &VGL) AND (LFD_NR = &NUMMER);
IF &DML_STATE <> 'OK' THEN
  ROLLBACK WORK;
  SEND MESSAGE 'FEHLER ',&SQL_STATE;
ELSE COMMIT WORK;
END IF;
END PROCEDURE;

```

Name of the external subprogram: MITLOES

```

PROCEDURE MITARBEITERLOESCHEN ;
COPY MITVAR;
DECLARE FORM DELBILD
  TTITLE NL 1,
  TAB 30,'M I T A R B E I T E R',NL 2,
  ' FOLGENDER MITARBEITERSATZ',TAB 60,'FORMULAR 04',NL 1,
  ' SOLL GELOESCHT WERDEN : ',NL 2,
  ' ABT_MIT_NR      : ',&ABT_MIT_NR,NL 1,
  ' LFD_NR         : ',&LFD_NR,NL 1,
  ' NACHNAME       : ',&NACHNAME,NL 1,
  ' VORNAME        : ',&VORNAME,NL 1,
  ' LAND           : ',&LAND,NL 1,
  ' STRASSE        : ',&STRASSE,NL 1,
  ' PLZ            : ',&PLZ,NL 1,
  ' ORT            : ',&ORT,NL 1,
  ' GEHALT         : ',&GEHALT,NL 1,
  ' SPRACHEN       : ',&SPRACHEN(1),',',',',
                    &SPRACHEN(2),',',',',
                    &SPRACHEN(3),',',',',
                    &SPRACHEN(4),',',',',NL 1,
  ' ABT_LEITER     : ',&ABT_LEITER,NL 1,
  ' PROJ_MIT       : ',&PROJ_MIT,NL 1

```

```

BTITLE ' ','-'(70),NL 1,
      TAB 50,'LOESCHEN ( J/N ) : ',RETURN &FRAGE;
WHENEVER &DML_STATE CONTINUE;
EXEC CONCAT(&ABFRAGEBEFEHL,CONCAT(&ABFRAGETEXT1,&ABFRAGETEXT2));
SELECT ALL * INTO &MITARBEITERSATZ.* FROM MITARBEITER
      WHERE (ABT_MIT_NR = &VGL) AND (LFD_NR = &NUMMER);
ROLLBACK WORK;
IF &DML_STATE <> 'OK' THEN
  SEND MESSAGE 'FEHLER ',&SQL_STATE;
  EXEC &FEHLERMELDUNG;
  BREAK PROCEDURE;
END IF;
DISPLAY DELBILD;
IF &FRAGE = 'J' THEN
  DELETE FROM MITARBEITER WHERE (ABT_MIT_NR = &VGL)
      AND (LFD_NR = &NUMMER);
  IF &DML_STATE <> 'OK' THEN
    ROLLBACK WORK;
    SEND MESSAGE 'FEHLER ',&SQL_STATE;
  ELSE COMMIT WORK;
  END IF;
END IF;
SET &FRAGE = 'N';
END PROCEDURE;

```

Name of the external subprogram: MITANZ

```

PROCEDURE MITARBEITERANZEIGEN;
DECLARE VARIABLE &L(4) CHAR(1),
      &G(4) CHAR(1),
      &H1(4) CHAR(3),
      &G_UNTER NUM(7,2) ,
      &G_OBER NUM(7,2) ,
      &INDEX NUM(1) INIT 1,
      &HLAND CHAR(3),
      &SUCHE1 CHAR(60),
      &SUCHE2 CHAR(60),
      &SUCHE CHAR (120),
      &CURSORDEC1 CHAR(70) INIT
'DECLARE ANZ CURSOR FOR S LAND,NACHNAME,VORNAME,GEHALT ',
      &CURSORDEC2 CHAR(40) INIT
'FROM MITARBEITER WHERE ',
      &CURSORDEC CHAR(110),
1 &ANZEIGESATZ,
2 LAND CHAR(3),
2 VORNAME CHAR(20),
2 NACHNAME CHAR(20),

```

```

                2 GEHALT NUM(7,2);
DECLARE ANZ CURSOR;
DECLARE FORM MITARBEITERABFRAGE
    TTITLE NL 1,
        TAB 30,'M I T A R B E I T E R',NL 4,
        TAB 10,'KREUZEN SIE ZUTREFFENDES AN :',NL 3,
        TAB 10,'DEUTSCHLAND ',RETURN &L(1),
        TAB 35,'GEHALT ZWISCHEN 2000 UND 4000 ',RETURN &G(1),NL 2,
        TAB 10,'USA ',RETURN &L(2),
        TAB 35,'GEHALT ZWISCHEN 4000 UND 6000 ',RETURN &G(2),NL 2,
        TAB 10,'SCHWEIZ ',RETURN &L(3),
        TAB 35,'GEHALT ZWISCHEN 6000 UND 8000 ',RETURN &G(3),NL 2,
        TAB 10,'ENGLAND ',RETURN &L(4),
        TAB 35,'GEHALT UEBER 8000 ',RETURN &G(4)
    BTITLE ' ','-'(70),NL 1;
DECLARE FORM MITARBEITERAUSGABE
    TTITLE NL 1,
        ' '(30),'MITARBEITERANZEIGE',NL 1,
        ' '(29),'-'(20)
    BTITLE '='(80);
DISPLAY MITARBEITERABFRAGE;
SET &H1(1) = 'FRG';
SET &H1(2) = 'USA';
SET &H1(3) = 'CH';
SET &H1(4) = 'ENG';
CYCLE WHILE &INDEX < 5;
    IF &L(&INDEX) <> ' ' THEN
        SET &HLAND = &H1(&INDEX);
    END IF;
    IF &G(&INDEX) <> ' ' THEN
        SET &G_UNTER = &INDEX * 2000;
        SET &G_OBER = &G_UNTER +2000;
    END IF;
    SET &INDEX = &INDEX + 1;
END CYCLE;
IF &G_OBER = 10000 THEN
    SET &G_OBER =20000;
END IF;
SET &SUCHE1 = ' (LAND = &HLAND)';
SET &SUCHE2 = ' (GEHALT >= &G_UNTER) AND (GEHALT <= &G_OBER)';
IF (&HLAND = ' ') AND (&G_UNTER <> 0) THEN
    SET &SUCHE = &SUCHE2;
END IF;
IF (&HLAND <> ' ') AND (&G_UNTER = 0) THEN
    SET &SUCHE = &SUCHE1;
END IF;

```

```

IF (&HLAND <> ' ') AND (&G_UNTER <> 0) THEN
    SET &SUCHE = CONCAT(&SUCHE1,CONCAT(' AND ',&SUCHE2));
END IF;
IF (&HLAND = ' ') AND (&G_UNTER = 0) THEN
    SET &SUCHE = ' (GEHALT > 1) ';
END IF;
SET &CURSORDEC = CONCAT(&CURSORDEC1,&CURSORDEC2);
EXEC CONCAT (&CURSORDEC,CONCAT(&SUCHE,','));
CYCLE ANZ INTO &ANZEIGESATZ.*;
    IF &DML_STATE <> 'OK' THEN
        ROLLBACK WORK;
        SEND MESSAGE 'FEHLER ',&SQL_STATE;
        BREAK PROCEDURE;
    END IF;
    FILL MITARBEITERAUSGABE TABLE VALUES &ANZEIGESATZ;
END CYCLE;
DROP CURSOR ANZ;
COMMIT WORK;
DISPLAY MITARBEITERAUSGABE;
END PROCEDURE;

```

Name of the external subprogram: MITDRUCK

```

PROCEDURE MITARBEITERDRUCK;
COPY MITVAR;
DECLARE LIST MITARBEITERAUSGABE
    TTITLE &DATUM,TAB 37,&ZEIT,TAB 110,'SEITE:',&PAGES,NL 3,
        TAB 60,'MITARBEITERLISTE',NL 1,
        TAB 59,'-'(18),NL 2
    BTITLE NL 2,'='(123);
FILL MITARBEITERAUSGABE TABLE NAMES &DRUCKSATZ;
CYCLE DRUCKCURSOR INTO &DRUCKSATZ.*;
    FILL MITARBEITERAUSGABE TABLE VALUES &DRUCKSATZ;
END CYCLE;
COMMIT WORK;
DISPLAY MITARBEITERAUSGABE;
EXEC CONCAT(&ABFRAGEBEFEHL,&DRUCKTEXT);
IF &FRAGE = 'J' THEN
    SYSTEM 'PRINT *SYSLST';
    SEND MESSAGE 'DIE MITARBEITERLISTE WIRD AUSGEDRUCKT. DUE-TASTE';
    SET &FRAGE = 'N';
ELSE
    SEND MESSAGE
        'DIE MITARBEITERLISTE WIRD AM ENDE DER BS2000-SITZUNG GEDRUCKT.';
END IF;
END PROCEDURE;

```

2.7 Pragmas

This section describes

- how pragmas can be used
- the difference between their use in DRIVE/WINDOWS, and in ESQL/COBOL programs and in the Utility Monitor
- the syntax used to specify pragmas in DRIVE/WINDOWS
- how DRIVE/WINDOWS handles errors in the context of pragmas

2.7.1 Application possibilities and advantages

The SESAM V2 user can use pragmas to

- activate block mode, i.e. specify the maximum number of rows in a (static, dynamic or variable) cursor table that can be read “in advance” by a FETCH statement in order to accelerate execution of subsequent FETCH statements considerably.



This functionality is only available if you are using SESAM/SQL V2.1 (or higher). SESAM/SQL V2.0 does not support block mode, which means that use of the block mode pragma clause PREFETCH in DRIVE/WINDOWS will result in an error (see section “Error handling” on page 66).

- output a readable representation of the internal access plan of the SQL optimizer to a file for new-style DML statements (SELECT or cursor processing, INSERT, UPDATE, DELETE)
- influence the execution rule (SQL access plan) for processing a new-style DML statement.
- specify the isolation level for database accesses by a new-style DML statement independent of the isolation level of the transaction in which the statement is executed.
- insert new oldest-style columns into an oldest-style table (CALL DML table) (see the utility statement MIGRATE and the DDL statement ALTER TABLE).



However, because DRIVE/WINDOWS does not currently support DDL statements for CALL DML tables, it cannot currently support this functionality.

- process base tables in the state “check pending” (see the SESAM V2 manual “SQL Language Reference Manual, Part 2: Utilities” [19]).



However, because DRIVE/WINDOWS does not currently support utility statements, it is not possible for a DRIVE user to fulfill the requirement for this processing (i.e. establish the state “check pending” in the current SQL session), which means that DRIVE/WINDOWS cannot currently support this functionality.

2.7.2 Differences in syntax compared with ESQL/COBOL and the Utility Monitor

While in ESQL/COBOL and in the Utility Monitor, pragmas are entered as special SQL comments together with the SQL statement that they are to influence, in DRIVE/WINDOWS they are declared using a separate DRIVE SQL statement, the PRAGMA statement. A PRAGMA statement is a program statement that is interpreted at compilation time. In the case of a static PRAGMA statement, this is when explicit source compilation (COMPILE statement) or implicit source compilation (DO or CALL statement: preliminary stage of procedure execution) is performed. In the case of a dynamic PRAGMA statement, this is when implicit statement compilation (EXECUTE statement: generation and compilation as a preliminary stage of statement execution) is performed. Refer to the description of the PRAGMA statement on page 119 for a complete description of how pragmas function.

2.7.3 Static and dynamic pragmas

Static PRAGMA statement only affect static SQL statements, while dynamic PRAGMA statements only affect dynamic SQL statements.

2.7.4 Pragma clauses

Because pragmas can only be declared via the PRAGMA statement

```
PRAGMA ' { pragma-clause }, ... '
```

in DRIVE/WINDOWS, the following is a description of pragma clauses rather than pragmas, unlike in the SESAM V2 manuals.

PREFETCH pragma clause

The PREFETCH pragma clause has the syntax:

```
PREFETCH n
```

You can use this clause to activate block mode by assigning the clause to a (static or dynamic) DECLARE CURSOR statement. When the first FETCH statement after OPEN is executed for this cursor, SESAM tries to read up to *n* rows in the cursor table “in advance” in order to accelerate subsequent FETCH statements. This functionality is identical with that of the PREFETCH clause within the DECLARE CURSOR statement (see description on page 88).

Block mode functionality is only effective if you are working with a SESAM V2.1 or higher. If you are working with SESAM/SQL V2.0, the PREFETCH clause within a DECLARE CURSOR statement and the PREFETCH pragma clause result in DRIVE errors: the clauses are permitted by DRIVE/WINDOWS (no DRIVE compilation error), but result in SESAM warnings and thus in DRIVE errors

- during SESAM precompilation at compilation time (static cursors and pragmas) and
- during SESAM preparation at compilation time (dynamic cursors and pragmas and variable cursors).

EXPLAIN pragma clause

SESAM V2 creates an internal evaluation rule, the SQL access plan, for most new-style SQL statements. The SQL optimizer ensures that an especially efficient access plan, in which as few system resources are used as possible, is created for DML statements (SELECT or cursor processing, INSERT, UPDATE, DELETE).

You can use the Explain components of the optimizer to determine the individual steps in which a DML statement is processed. If you declare an EXPLAIN pragma clause before the statement, you are provided with a readable representation of the internal access plan in a file. The EXPLAIN pragma clause has the following syntax:

```
EXPLAIN INTO filename
```

The output of the Explain components and the information contained in the output is described in chapter 7 in the SESAM V2 “Performance” manual [45].

Pragma clauses for influencing the access plan

You can influence the execution plan (SQL access plan) for processing a new-style DML statement (SELECT or cursor processing, INSERT, UPDATE, DELETE) with the following clauses:

- Pragma clause IGNORE INDEX
OPTIMIZATION LEVEL *n*
- Pragma clause OPTIMIZATION LEVEL
IGNORE INDEX *index_name*
- Pragma clause SIMPLIFICATION
SIMPLIFICATION { ON | OFF }

(see the SESAM V2 “Performance“ manual [45]).

Pragma clause ISOLATION LEVEL

The pragma clause ISOLATION LEVEL has the following syntax:

```
ISOLATION LEVEL { READ UNCOMMITTED |  
                  READ COMMITTED   |  
                  REPEATABLE READ  |  
                  SERIALIZABLE     }
```

You can use this clause to specify the isolation level for database accesses of a new-style DML statement independent of the isolation level of the transaction in which the statement is executed.

Pragma clause DATA TYPE

The pragma clause DATA TYPE has the following syntax:

```
DATA TYPE OLDEST
```

You can use this clause to insert columns of the type “oldest-style” into CALL DML tables. Use of this clause, however, leads to an error with &DML_STATE = 'SQL ERROR' in DRIVE/WINDOWS.

CHECK pragma clause

The CHECK pragma clause has the following syntax:

```
CHECK { ON | OFF }
```

The default value is ON. Under certain conditions, you can use the clause 'CHECK OFF' to access base tables that are in the state “check pending” with DML statements (see SESAM V2 “SQL Language Reference Manual, Part 2: Utilities” [19]). Use of this clause, however, leads to an error with &DML_STATE = 'SQL ERROR' in DRIVE/WINDOWS.

2.7.5 Error handling

When using pragmas, the following errors may occur:

- The syntax of PRAGMA statement is incorrect.

In this case, DRIVE/WINDOWS reports a syntax error during compilation of the PRAGMA statement. In the case of a dynamic PRAGMA statement, &ERROR is then assigned the value 'SYNTAX ERROR'.

Otherwise, the PRAGMA statement takes effect (so-called DRIVE effect) when the next SQL statement is executed and the contents of the specified literal are passed to SESAM as an SQL comment with the prefix %PRAGMA (see the description on page 119 for a detailed explanation of how the PRAGMA statement functions). Errors may occur that SESAM detects and which mean that the statement has no affect in SESAM:

- The syntax of the contents of the literal in the PRAGMA statement is incorrect.

In this case, SESAM reports an SQLSTATE of the class 01 (warning) to DRIVE/WINDOWS and precompiles (static pragma) or prepares (dynamic pragma) the SQL statement without the pragma. If a SESAM error occurs while this is being done, the warning and thus the pragma error are lost.

- The syntax of the contents of the literal in the PRAGMA statement is correct, but the pragma clause is assigned to an SQL statement that it cannot influence (see description of the individual clauses).

This is the case, for example, if the PREFETCH pragma clause is used with SESAM/SQL V2.0.

SESAM reports an SQLSTATE of the class 01 (warning) to DRIVE/WINDOWS and precompiles (static pragma) or prepares (dynamic pragma) the SQL statement without the pragma.

- The syntax of the contents of the literal in the PRAGMA statement is correct and the pragma clause is assigned to an SQL statement that it can influence in SESAM, but this influence is hindered for some other reason (see description of the individual clauses).

This is the case, for example, if a BS2000 file that cannot be shared or which has not been cataloged as SHAREABLE is specified in the EXPLAIN pragma clause.

In this case, SESAM also reports an SQLSTATE of the class 01 (warning) to DRIVE/WINDOWS and precompiles or prepares the SQL statement without the pragma.

DRIVE/WINDOWS now converts all the warnings issued by SESAM (SQLSTATE class 01) into DRIVE SQL errors with `&DML_STATE = 'SQL ERROR'` and `&ERROR = 'OK'`.

At compilation time (static pragmas), this means that the compilation of the SQL statement involved is considered errored and the warning reported by SESAM is included in the compiler listing as a DRIVE SQL error (message number DRI0536). The SQL statement that was precompiled by SESAM without the pragma is therefore not taken over by DRIVE/WINDOWS.

At execution time (dynamic pragmas), this means that all SESAM warnings can be handled by WHENEVER in program (or debugging) mode. This means, in particular, that all pragma errors can also be handled by WHENEVER (see the “DRIVE Programming Language” manual [2], chapter 4, “Programming logic”). The following must, however, be observed:

- If no WHENEVER action is defined for the error event `&DML_STATE = 'SQL ERROR'` or the action BREAK is defined, DRIVE/WINDOWS aborts the program (or branches to the end breakpoint).

If a transaction is open at this time, it is rolled back by DRIVE/WINDOWS. This means, in particular, that the SQL statement involved is rolled back if it has been executed by SESAM without the pragma.

- If the WHENEVER action CONTINUE is defined for `&DML_STATE = 'SQL ERROR'`, DRIVE/WINDOWS continues the program with the SQL or DRIVE statement that follows the SQL statement involved.

If a COMMIT WORK statement is subsequently executed, execution of the SQL statement involved by SESAM without PRAGMA is committed.

- If calling an internal subprogram is defined as the WHENEVER action for `&DML_STATE = 'SQL ERROR'`, how the SESAM warning issued because of a pragma error is handled depends on the logic of the subprogram:
 - If (after various DRIVE statements for error handling, if necessary) the transaction is rolled back (ROLLBACK WORK), execution of the SQL statement involved by SESAM without PRAGMA is also rolled back.
 - If the transaction is committed (COMMIT WORK), execution of the SQL statement involved by SESAM with PRAGMA is committed.
 - If the internal subprogram is terminated or aborted without ROLLBACK WORD or COMMIT WORK, the aforementioned applies to the WHENEVER action CONTINUE.

3 DRIVE SQL statements

This chapter provides you with a description of two classes of SQL statements:

- All SQL statements from SESAM/SQL V2 that DRIVE/WINDOWS supports, with mention of any restrictions or extensions.
- SQL statements from SESAM/SQL V1 that DRIVE/WINDOWS continues to support for reasons of compatibility or performance (DROP statements in particular).

In ESQL/COBOL programs and in the Utility Monitor, SQL statements do not end with a semicolon. In DRIVE/WINDOWS, this is only possible in interactive mode and for dynamic program statements. In DRIVE programs, static SQL statements must always end with a semicolon.

In ESQL/COBOL, SQL statements are enclosed by the keywords “EXEC SQL” and “END-EXEC”. In command files for the Utility Monitor, SQL statements start with the keyword SQL; continuation lines must be linked with the “-” character (this character must be doubled if used within SQL expressions). This is not necessary in DRIVE/WINDOWS: the SQL database language is completely integrated in the DRIVE language, a fourth generation programming language.

In ESQL/COBOL programs and Utility Monitor command files, comment lines start with the “*” character. Comments can start with the string “--” within and before SQL statements. The end of a comment is the end of the line. In DRIVE/WINDOWS, comments start with “/*” and must end with “*/”.

Normally, DRIVE/WINDOWS can determine whether SQL statements violate any syntax rules, and these violations are treated like analysis errors in DRIVE statements (see the “DRIVE Programming Language” manual [2], chapter 4, “Programming logic”). This means, in this case, that the violation involves a DRIVE rule, i.e. a violation of statement syntax (use of keywords and the way in which they are written, the order of clauses, etc.), which is indicated by an appropriate DRIVE error message. In some SQL statements, syntax violations can be reported by SESAM with an appropriate SQLSTATE (SQLSTATE classes 42 and 01, in particular). In this case, a violation of a SESAM rule has occurred (semantic dependencies, context violations, invalid SQL metadata, missing access permissions, invalid or incorrectly used pragma clauses).

ALTER TABLE - Alter base table

You use ALTER TABLE to modify an existing base table. You can add or update columns, or you can add or delete integrity constraints.

If you are using a CALL DML table, you can only update columns. The restrictions that apply to CALL DML tables are described below.

The BASE_TABLES view in the INFORMATION_SCHEMA provides you with information on which base tables have been defined (see chapter 8, “Information schemas”, in the “SESAM/SQL-Server, SQL Reference Manual, Part 1”).

The current authorization identifier must own the schema to which the base table belongs.



This statement can destroy declaration statements in a DRIVE program. A static cursor must be created in the declaration section of the DRIVE program. If an ALTER TABLE statement in the body of the DRIVE program updates the table for which the cursor was declared, the cursor can no longer be accessed.

DRIVE/WINDOWS provides limited support for the ALTER TABLE statement of SESAM/SQL:

- no columns can be added to CALL DML tables
- only one column can be added or updated

The corresponding possibilities for SESAM/SQL represent an extension of the SQL2 standard [47].

ALTER TABLE table

```
{ ADD [ COLUMN ] column_definition |
ALTER [ COLUMN ] column
    { DROP DEFAULT |
      SET basic_data_type |
      SET default } |
ADD [ CONSTRAINT integrity_constraint_name ] table_constraint |
DROP CONSTRAINT integrity_constraint_name RESTRICT }
```

```
default::= DEFAULT { literal |  
                    CURRENT DATE |  
                    CURRENT TIME |  
                    CURRENT TIMESTAMP |  
                    [ CURRENT ] USER | SYSTEM USER |  
                    NULL }
```

table

Name of a base table (see metavariable *table_specification*, especially for full and partial qualification).

ADD [COLUMN] *column_definition*

Adds new columns to the base table. The new columns are added to the end of the base table. *column_definition* defines the columns.

column_definition can only include a DEFAULT clause if the table is empty. You cannot define a primary key constraint in *column_definition*.

If the table already contains rows, the NULL value is entered in the new column. Any defined integrity constraints are checked when this is done.

ALTER [COLUMN] *column*

Name of the column to be modified.

The column cannot occur in views, indexes or integrity constraints. All the temporary views based on the table are deleted.

DROP DEFAULT

Deletes the default value for the column.

SET *basic_data_type*

New data type of the column.

The dimension of a multiple column cannot be modified.

basic_data_type can only be CHARACTER, VARCHAR or NUMERIC. For CHARACTER and VARCHAR, the new (maximum) length cannot be less than the old length. For NUMERIC the new number of digits cannot be smaller than the old number of digits, and the number of decimal places must stay the same.

column cannot be referenced in a view definition, in an integrity constraint, or in an index definition.

SET *default*

Defines a new SQL default value for the column.

- *column* cannot be a multiple column.
- *column* cannot be a CALL DML column.
- *default* must conform to the assignment rules for default values see section 4.4.2, “Default values for table columns”, in the “SESAM/SQL-Server, SQL Reference Manual, Part 1”).

The default is evaluated when a row is inserted or updated, and the default value is used for *column*.

ADD CONSTRAINT clause

Adds an integrity constraint to the base table.

CONSTRAINT *integrity_constraint_name*

Assigns a name to the integrity constraint. The unqualified name of the integrity constraint must be unique within the schema. You can qualify the name of the integrity constraint with a database and schema name. The database and schema name must be the same as the database and schema name of the base table to which the integrity constraint is being added.

CONSTRAINT *integrity_constraint_name* omitted:

The integrity constraint is assigned a name according to the following pattern:

{ UN | FK | CH } *integrity_constraint_name*

where UN stands for UNIQUE, FK for FOREIGN KEY and CH for CHECK.

integrity_constraint_name is a 16-digit integer (time stamp).

table_constraint

Specifies an integrity constraint for the table (see metavariable *table-constraint*).

table-constraint cannot define a primary key constraint.

DROP CONSTRAINT *integrity_constraint_name* RESTRICT

Deletes the integrity constraint *integrity_constraint_name*.

You cannot delete the primary key constraint on a table or the uniqueness constraint on a column if a referential constraint on another table references this column.

Special considerations for CALL DML tables

The ALTER TABLE statement for CALL DML tables must take the following restrictions into account:

- Only the specification ALTER [COLUMN] *column* SET *basic_data_type* is permitted. *column* cannot be a multiple column.
- Only the data types CHARACTER and NUMERIC are permitted for *basic_data_type*.

Example

The example below deletes the NOT NULL integrity constraint on the column `company` of the `customers` table. The name of the integrity constraint is in the `CHECK_CONSTRAINTS` view of the `INFORMATION-SCHEMA`.

```
ALTER TABLE customer
    DROP CONSTRAINT customer.company_notnull RESTRICT;
```

The following DRIVE-DDL program inserts one column and updates two columns.

```
OPTION AUTHORIZATION = x;
OPTION CATALOG = y;
/* Load column in SQL table */
ALTER TABLE A.T1
    ADD COLUMN C1 CHAR (20) NOT NULL;
/* Update default value in SQL table */
/* Before: Default: 'OLD' */
ALTER TABLE A.T2
    ALTER COLUMN C2 SET DEFAULT 'NEW';
/* Update column in CALL DML table */
/* Before: CHAR (24) */
ALTER TABLE A.T3
    ALTER COLUMN C3 SET CHAR (30);
```

CLOSE - Close cursor

You use CLOSE to close a cursor you declared with the DECLARE statement and opened with OPEN or RESTORE.

The cursor description (see the DECLARE statement) is retained. You can save the current cursor position with STORE before the cursor is closed.

You may close and open any number of times. An open cursor is also closed at the end of a transaction.

A CLOSE statement followed by an OPEN for the same cursor is, for example, useful if you used variables in the *cursor_description* when you declared the cursor.

cursor_description is updated with the variable values valid at the time that the OPEN statement is issued, and the cursor table is filled with the current data from the database.

In program mode, cursors can only be referenced in the source file in which they were declared with a DECLARE statement.

CLOSE *cursor*

cursor

Name of the cursor to be closed.

COMMIT WORK - Terminate transaction

You use COMMIT WORK to terminate a transaction and commit the updates performed on the database during the transaction. The updated data is then available to all other transactions.

COMMIT WORK also commits all values set since the end of the last transaction with the SET and PARAMETER DYNAMIC statements and their operands CATALOG/SCHEMA/AUTHORIZATION. This commits the SQL environment in dynamic programs for subsequent transactions.

A new transaction is started by the first SQL statement after COMMIT WORK that initiates a transaction (see below).

COMMIT WORK closes all cursors opened during the transaction. A cursor defined with TEMPORARY is deleted at a higher program level when the next COMMIT WORK statement is issued. You can save the cursor positions of non-PREFETCH cursors with the STORE statement.

For information on the rules that apply to distributed applications, refer to the “DRIVE Programming Language” manual [2], chapters 12, “Distributed applications”, and 13, “Distributed transaction processing”.

```
COMMIT [ WORK ] [ WITH { display | send message | stop } ]
```

WITH

WITH allows you to specify a statement that is executed after the end of the transaction.

If you do not specify WITH in an interactive program in the operating mode with the TP monitor (UTM operation), the transaction is terminated and the program run is continued in the subsequent subprogram without output to the screen.

WITH may only be specified in program mode.

display

Display a form. The following statement may be used:

- DISPLAY *screen_form* (see the “Directory of DRIVE Statements” [3], DISPLAY *screen_form* statement).
- DISPLAY *form_name* (see the “Directory of DRIVE Statements” [3], DISPLAY *form_name*).
- DISPLAY FORM (see the “Directory of DRIVE Statements” [3], DISPLAY FORM statement).

send message

Send messages (see the “Directory of DRIVE Statements” [3], SEND MESSAGE statement).

stop

Terminate the DRIVE session (see the “Directory of DRIVE Statements” [3], STOP statement).

Transaction

You start a transaction with any SQL statement that initiates a transaction. All subsequent SQL statements up to the next COMMIT WORK or ROLLBACK WORK statement belong to one transaction. COMMIT WORK or ROLLBACK WORK terminates the transaction.

Initiating a transaction

The following DRIVE-SQL statements do not initiate a transaction:

- static CREATE TEMPORARY VIEW (not executable)
- static DECLARE (not executable)
- PERMIT
- PRAGMA (not executable)
- SET CATALOG
- SET SCHEMA
- SET SESSION AUTHORIZATION
- SET TRANSACTION
- WHENEVER (not executable)

The EXECUTE statement only initiate a transaction if the dynamic statement to be executed initiates a transaction.

All other SQL statements initiate a transaction if no transaction is open when they are executed.

Statements within a transaction

The following statements cannot be executed within a transaction:

- PERMIT
- SET SESSION AUTHORIZATION
- SET TRANSACTION

You cannot execute or prepare an SQL statement that manipulates data (query, update) in a transaction in which an SQL statement for defining or managing schemas is executed. This means that you cannot mix DDL and DML statements in a transaction.

Effects of COMMIT WORK

COMMIT WORK affects the subsequent transactions, as well as the open cursors and the defaults in the transaction.

Effect on subsequent transactions

COMMIT WORK work sets the isolation or consistency level and the transaction mode, which were set for the transaction with the SET TRANSACTION statement, back to their default values. Any subsequent transaction therefore works the default isolation or consistency level and transaction mode if they are not changed again with SET TRANSACTION.

Effect on cursors

COMMIT WORK closes all the cursors opened in the transaction. If you want to save the cursor position beyond the end of the transaction, you can save the position with the STORE statement and restore it later with RESTORE, provided that the cursor is not a PREFETCH cursor.

Cursors defined within the current transaction (see the DECLARE statement) are committed provided that their definitions are not canceled (see the DROP CURSOR statement). Deletions of cursor definitions are committed.

Effect on temporary views

Temporary view defined within the current transaction (see the CREATE TEMPORARY VIEW statement) are committed provided that their definitions are not deleted (see the DROP TEMPORARY VIEW statement). Deletions of view definitions are committed.

Effect on defaults

Default values defined for dynamic programs with SET CATALOG, SET SCHEMA and SESSION AUTHORIZATION are committed after COMMIT WORK.

Behavior of SESAM/SQL in the event of an error

If a transaction cannot be terminated normally because of an error, SESAM/SQL rolls back the complete transaction. Refer to ROLLBACK WORK for information on which database objects are affected.

Example

The example below shows you how to process a cursor row by row using a loop subject to transaction management. Cursor processing within a loop with COMMIT WORK is only permitted if you saved the cursor with STORE and restored it with RESTORE.

```
DECLARE c1 ...;
...
CYCLE c1 INTO &var.*;
...
STORE c1;
COMMIT WORK;
RESTORE c1;
...
END CYCLE;
```

CREATE SCHEMA - Create schema

You use CREATE SCHEMA to create a schema. At the same time you can define tables, views and privileges. You can also modify the schema later with the appropriate CREATE, ALTER, GRANT, DROP and REVOKE statements.

The current authorization identifier must have the special privilege CREATE SCHEMA.



DRIVE/WINDOWS provides limited support for the CREATE SCHEMA statement of SESAM: No *create_index_definition* can be used because DRIVE/WINDOWS does not support the SSL statement CREATE INDEX.

CREATE SCHEMA

```
{ [ catalog . ] schema [ AUTHORIZATION authorization_id ] |
  AUTHORIZATION authorization_id }

[ { create_table_definition |
  create_view_definition |
  grant_definition } ... ]
```

schema

Name of the schema. The unqualified schema name must be unique within the database. You can qualify the schema name with a database name.

schema omitted:

The name of the authorization identifier in the AUTHORIZATION clause is used as the schema name.

AUTHORIZATION *authorization_id*

The authorization identifier owns the schema.

This authorization identifier is also used as the name of the schema if you do not specify a schema name. The authorization identifier can up to 18 characters long.

AUTHORIZATION *authorization_identifier* omitted:

In the case of a static CREATE SCHEMA statement, the authorization identifier specified via OPTION AUTHORIZATION owns the schema. For a dynamic CREATE SCHEMA statement, the authorization identifier specified in the last SET SESSION AUTHORIZATION statement owns the schema (see also PARAMETER DYNAMIC AUTHORIZATION for information on setting default values).

create/grant_definitions

If you use unqualified table and index names in the CREATE and GRANT statements, the names are automatically qualified with the database and schema name of the schema.

create_table_definition

CREATE TABLE statement, without a concluding semicolon, that creates a base table for the schema. The table name must be unqualified or can only be qualified with the database and schema names from the CREATE SCHEMA statement.

create_view_definition

CREATE VIEW statement, without a concluding semicolon, that creates a view for the schema. The view name must be unqualified or can only be qualified with the database and schema names from the CREATE SCHEMA statement.

grant_definition

GRANT statement that grants privileges for a base table or a view of the schema (DML rights and referential rights). You cannot use the GRANT statement to grant special privileges.

create/grant_definitions omitted:

An empty schema is created.

How CREATE SCHEMA functions

CREATE TABLE, CREATE VIEW and GRANT definitions that are specified in the CREATE SCHEMA statement are executed in the order in which they are specified. You must therefore place definitions that reference existing tables or views after the definition that creates these tables or views.

Example

The example below creates the schema `andromeda`, defines a table and grants privileges for the schema.

```
CREATE SCHEMA andromeda
CREATE TABLE telephone_list
    (name CHARACTER (25),
    telephone CHARACTER (15),
    fax CHARACTER (15))
GRANT ALL PRIVILEGES ON telephone_list TO hugh;
```


CREATE TABLE - Create base table

You use CREATE TABLE to create a base table in which the data is permanently stored. SQL tables can only be processed with SQL, not with CALL DML.

The current authorization identifier must own the schema to which the table belongs. If you specify the space for the base table, the current authorization identifier must own the space.



DRIVE/WINDOWS provides limited support for the CREATE TABLE statement of SESAM/SQL: no CALL DML tables can be created.

```
CREATE TABLE table
```

```
( { column_definition |
  [ CONSTRAINT integrity_constraint_name ] table_constraint },... )

[ USING SPACE space ]
```

table

Name of the new base table. The unqualified table name must be different from all the other base table names and view names in the schema and must be different from the unqualified name of any temporary views. You can qualify the table name with a database and schema name (see also metavariable *table_specification*).

column_definition

Defines columns for the base table (see also metavariable *column_definition*).

You must define at least one column. A base table can have up to 26,134 columns of any type except VARCHAR and up to 1000 columns of the type VARCHAR.

The current authorization identifier is granted all table privileges for the defined columns.

CONSTRAINT *integrity_constraint_name*

Assigns an integrity constraint name to the table constraint. The unqualified name of the integrity constraint must be unique within the schema. You can qualify the name of the integrity constraint with a database and schema name. The database and schema name must be the same as the database and schema name of the base table for which the integrity condition is defined.

CONSTRAINT *integrity_constraint_name* omitted:

The integrity constraint is assigned a name according to the following pattern:

```
{ UN | FK | CH } integrity_constraint_name
```

where UN stands for UNIQUE, FK for FOREIGN KEY and CH for CHECK.

integrity_constraint_name is a 16-digit integer (time stamp).

table_constraint

Defines an integrity constraint for the table (see metavariable *table_constraint*).

USING SPACE *space*

Name of the space in which that table is to be stored. The space must already be defined for the database to which the table belongs. *space* can be up to 18 characters long. You can qualify the space name with the database name. This database name must be the same as the database name of the base table.

USING SPACE *space* omitted:

The table is stored in the default space of the current authorization identifier on the storage group D0STOGROUP.

The default space is D0*authorization_identifier* with the first 10 characters of the authorization identifier. If this space does not yet exist, it is created if the current authorization identifier has been granted the special privilege USAGE for the storage group D0STOGROUP (see the “SESAM/SQL-Server SQL Reference Manual, Part 1” [18] and Part 2 [19]).

Example

The example below shows the CREATE TABLE statement for the `orders` table in the demonstration database.

```
CREATE TABLE orders
(
  order_num    INTEGER CONSTRAINT order_num_primary PRIMARY KEY,
  cust_num     INTEGER CONSTRAINT o_cust_num_notnull NOT NULL,
  contact_num  INTEGER,
  order_date   DATE DEFAULT CURRENT_DATE,
  order_text   CHARACTER (30),
  actual       DATE,
  target       DATE,
  order_stat   INTEGER DEFAULT 1 CONSTRAINT order_stat_notnull NOT NULL,
  CONSTRAINT o_cust_num_ref_customers FOREIGN KEY (cust_num)
    REFERENCES customers,
  CONSTRAINT contact_num_ref_contacts FOREIGN KEY (contact_num)
    REFERENCES contacts,
  CONSTRAINT order_stat_ref_ordstat FOREIGN KEY (order_stat)
    REFERENCES ordstat(order_stat_num)
);
```

CREATE TEMPORARY VIEW - Declare temporary view

You use CREATE TEMPORARY VIEW to declare a temporary view that can be used within the source file in which it was declared or in interactive mode. A temporary view is a named database query that is stored for the duration of the DRIVE session or until an appropriate DROP TEMPORARY VIEW statement is issued. See the “DRIVE Programming Language” manual [2], chapter 10, “Transaction concept”, for information on the scope of validity and life of temporary views.

The statement declaring a static temporary view is only permitted at the beginning of a program, i.e. before the processing statements. You can define dynamic temporary views at execution time with EXECUTE.

The current authorization identifier at the time at which the source program is compiled is the owner of the static temporary view (see OPTION AUTHORIZATION). In particular, all the static temporary views in a compilation unit have the same owner. The owner of a dynamic temporary view is the current authorization identifier at the time at which the dynamic CREATE TEMPORARY VIEW statement is executed.

A temporary view is assigned to the default catalog that has been set by CREATE TEMPORARY VIEW and which is valid at compilation time (static view) or at execution time (dynamic view) (see OPTION CATALOG, SET CATALOG and PARAMETER DYNAMIC CATALOG) or to the catalog referenced in *query_expression* by means of direct qualification. The latter takes precedence over the former.



In a DRIVE session (TIAM session or UTM conversation), only dynamic temporary views can be declared for an SQL user per SESAM database. If, after a successful dynamic CREATE TEMPORARY VIEW statement, the SQL user is changed (this requires that the transaction be terminated beforehand), the new SQL user can only declare dynamic temporary views in a different catalog. Otherwise, DRIVE/WINDOWS issues the following error message:

```
DRI0536 42SQ6 -127 SCHEMA catalog.MODULES FOR USER authorization
CANNOT BE ACCESSED.
```

catalog is the name of the SESAM database for which dynamic temporary views declared by a different SQL user in the same DRIVE session already exist. *MODULE* is the reserved name of the schema to which all the dynamic temporary views in *catalog* are assigned. *authorization* is the name of the new SQL user who is not able to declare any dynamic temporary views in *catalog* as long as the schema *MODULE* is not empty (see the DROP TEMPORARY VIEW statement).

When the statement that references the temporary view is executed, the current authorization identifier must have the SELECT privilege for the tables used in *query_expression* if these tables belong to a different schema owner.

A temporary view ceases to be valid

- when the program is terminated (the life of a temporary view ends when the application is terminated)
- if the program is aborted
- when DRIVE is terminated (STOP)
- if DROP TEMPORARY VIEW *view* or DROP TEMPORARY VIEWS (only in interactive mode or within the EXECUTE statement if the view was also declared using EXECUTE).

```
CREATE TEMPORARY VIEW temp_view_name [ ( { column },... ) ]
    AS query_expression
```

temp_view_name

Name of the new temporary view. The name of a static temporary view cannot be longer than 24 characters. The name of a dynamic temporary program view or a temporary dialog view cannot be longer than 31 characters.

The name of a temporary view must be unique within the current compilation unit.

temp_view_name cannot be “PLAM_DIRECTORY”.

(*column*,...)

Name of the column in the view. You only need to specify the view columns if the columns of the underlying table are ambiguous or if there are derived columns that do not have names.

(*column*,...) omitted:

The column names returned by the query expression are used.

AS *query_expression*

Query expression that selects the columns and rows from the base table to create the new view. The columns of the views have the same data type as the underlying columns in the query expression. *query_expression* cannot reference a temporary or permanent view.

You cannot include *variable* in the query expression. If the columns in the view are named, the number of columns in the derived table of the query expression must be the same as the number of named columns.

Updatable temporary view

A temporary view is updatable (with INSERT, UPDATE or DELETE) if the underlying query expression is updatable (see metavariable *query_expression*).

Privileges for the temporary view

The current authorization identifier is granted the SELECT privilege for the temporary view. If the view is updatable, the current authorization identifier is granted the privileges INSERT, UPDATE and DELETE if it has been granted these privileges for the underlying base table.

All accesses to temporary views that reference tables in one database must be performed using the same current authorization identifier.

Restrictions for temporary views

New SQL applications should use views rather than temporary views since temporary views have the following disadvantages:

- Temporary views can only reference base tables not views.
- Temporary views are only valid within a single compilation unit.
- Temporary views can only be used with the current authorization identifier (no GRANT statement for temporary views).
- In programs with multiple SQL users, the use of dynamic temporary views is restricted to a great extent (see above).
- The INFORMATION_SCHEMA does not include a description of the temporary views.

Example

The example below defines a temporary view that contains the completed orders from the base table orders.

```
CREATE TEMPORARY VIEW complete
  AS SELECT * FROM orders
  WHERE actual IS NOT NULL;
```

CREATE VIEW - Create view

You use `CREATE VIEW` to create a (permanent) view. A view is a table defined by a query expression that is not evaluated until the view is used. Permanent views are stored in the database and are part of the metadata of the database. `CREATE VIEW` is an executable statement that must be located in the body of `DRIVE` programs.

The current authorization identifier must own the schema for which the view is created and must have the `SELECT` privilege for the tables used if it does not own the tables.

```
CREATE VIEW table [ ( column,... ) ]
    AS query_expression
    [ WITH CHECK OPTION ]
```

table

Name of the new (permanent) view. The unqualified view name must be different from all the other names of base tables and views of the schema. You can qualify the view name with a database and schema name (see metavariable *table_specification*).

(column,...)

Name of the column in the view. You only need to name the view columns if the columns of the underlying table are ambiguous or if there are derived columns without a name.

(column,...) omitted:

The column names returned by the query expression are used.

AS *query_expression*

Query expression that selects the columns and rows for the new view from existing base tables and permanent views (see metavariables *query_expression* and *sql_expression*). The columns in the view have the same data type as the underlying columns in the query expression.

The tables named in the query expression must belong to the same database as the view. You cannot include *variable* in the query expression. If the columns in the view are named, the number of columns in the derived table of the query expression must be the same as the number of named columns.

WITH CHECK OPTION

Rows that you insert or update via the view are checked to see if they satisfy the constraint defined in the query expression. Rows that do not satisfy the condition are rejected. The view must be updatable.

The query expression can only include multiple columns in the SELECT clause, not in the WHERE clause.

WITH CHECK OPTION omitted:

If the view is updatable, you can insert or update rows in the view that do not satisfy the condition in the query expression. These rows cannot subsequently be accessed via the view.

Updatable view

A view is updatable (with INSERT, UPDATE or DELETE) if the underlying query expression is updatable (see metavariable *query_expression*).

Privileges for the view

The current authorization identifier is granted the SELECT privilege for the view. It is only extended GRANT authorization, which allows it to grant this privilege to other users, if it has GRANT authorization for the SELECT privilege for all the tables used.

If the view is updatable, the current authorization identifier is granted the privileges INSERT, UPDATE and DELETE if it has been granted these privileges for the underlying base table. It is only extended GRANT authorization, which allows it to grant these privileges to other users, if it has GRANT authorization for the appropriate privilege of the underlying base table.

Example

The example below defines a view containing the completed orders in the base table orders.

```
CREATE VIEW complete
  AS SELECT * FROM orders
  WHERE actual IS NOT NULL;
```

DECLARE - Declare cursor

You use DECLARE to define a cursor. You can use the cursor to access the individual rows in a derived table. The current row on which the cursor is positioned can be read. If the cursor is updatable, you can also update and delete rows.

A static cursor declaration must physically precede any statement that uses the cursor in the program text. All the statements that use this cursor must be located in the same compilation unit.

The DECLARE statement for a static cursor is not an executable statement. This means that the statement declaring a static cursor is only permitted at the beginning of the program, i.e. before the processing statements.

You must specify *cursor_description* if you are declaring a dynamic cursor (see the EXECUTE statement in the “Directory of DRIVE Statements” [3]).

In DRIVE/WINDOWS, up to 20 dynamic and variable cursors are permitted. The DRIVE program is aborted if more cursors are declared. You can prevent the program from being aborted with WHENEVER &DML_STATE IN ('TOO MANY CURSORS') (see WHENEVER and the “DRIVE Programming Language” manual [2], chapter 3.1.2, “System variables”).

Scope of validity of a cursor:

A cursor ceases to be valid

- when the program is terminated (the life of a cursor ends when the application or transaction is terminated)
- if the program is aborted
- when DRIVE is terminated (STOP)
- if DROP CURSOR *cursor* (DRIVE statement, only in interactive mode, dynamically or for variable cursors).
- DROP CURSORS (DRIVE statement, only in interactive mode or dynamically)

When you switch from DRIVE interactive mode to program mode, the cursor definition remains valid, but the cursor position is lost because no transaction can be open when this switch takes place. You can, however, save the cursor position with STORE, provided that the cursor involved is not a PREFETCH cursor.

In DRIVE program mode, a cursor defined with PERMANENT remains valid beyond the end of a program invoked with CALL, and its position is retained.

A cursor defined with TEMPORARY is closed when the program is terminated and deleted if a COMMIT WORK statement is issued on a higher program level.

A cursor always ceases to be valid in program mode when you switch to interactive mode, if the program is aborted, and when DRIVE is terminated (STOP or COMMIT WORK WITH STOP).

```

DECLARE cursor [ { PERMANENT | TEMPORARY } ]
              [ SCROLL ] [ PREFETCH n ] CURSOR [ FOR cursor_description ]

cursor_description ::=
    query_expression
    [ ORDER BY { { column | column(pos_no) | column_number }
    [ { ASCENDING | DESCENDING } ] },... ]
    [ FOR UPDATE [ OF { column },... ] ]

n ::= unsigned_integer

pos_no ::= unsigned_integer

column_no ::= unsigned_integer

```

cursor

Name of the cursor. The name of a static cursor cannot be longer than 11 characters. The name of a dynamic or variable program cursor or a dialog cursor cannot be longer than 18 characters. You cannot define more than one cursor with the same name within a compilation unit. You cannot define two cursors with the same name on a single program level or in interactive mode. The scope of validity of the cursor is limited to the compilation unit in which the cursor is defined.

PERMANENT

PERMANENT can only be specified within programs called using CALL.

The position of the cursor is retained after a program invoked with CALL is terminated, provided that no COMMIT WORK statement was executed in the called program or in the calling program between the CALLS.

When the program is invoked with CALL runs for the first time, the cursor must be opened with the OPEN statement. Whenever it executes subsequently, no OPEN statement may be issued for that cursor.

The calling program must not contain a COMMIT WORK statement.

TEMPORARY

TEMPORARY is the default value.

TEMPORARY can only be specified in programs called using CALL.

The cursor is closed at the end of the CALLED program and its position is lost (end of the scope of validity of the cursor). The cursor is deleted (end of the life of the cursor) when the next COMMIT WORK statement is issued at a higher program level.

SCROLL

You can position the cursor on any row in the derived table and in any order with FETCH NEXT/PRIOR/FIRST/LAST/RELATIVE/ABSOLUTE.

You can only specify SCROLL if no FOR UPDATE clause was defined in the cursor description of *cursor*.

SCROLL omitted:

You can only position the cursor on the next row. Only the position specification NEXT is permitted for FETCH.

PREFETCH

The PREFETCH clause increases performance by activating block mode.

Instead of the PREFETCH clause, you can also use a PRAGMA statement with the pragma clause PREFETCH to activate block mode (see section "Pragmas" on page 62). Both ways of activating block mode are functionally equivalent to each other. A prerequisite for its use is that you are working with a SESAM/SQL Version 2.1 or higher.



If you are working with SESAM/SQL V2.0, use of the PREFETCH clause or PREFETCH pragma will result in SESAM errors (SQLSTATE class 01).

A cursor for which block mode is activated using one of the above-mentioned methods is referred to as a **PREFETCH cursor**.

The following statements are not permitted for a PREFETCH cursor:

FETCH PRIOR, FIRST, LAST, RELATIVE, ABSOLUTE (only positioning with FETCH NEXT is permitted)

STORE and RESTORE

DELETE... WHERE CURRENT OF...

UPDATE... WHERE CURRENT OF...

n

Block factor *n-1* indicates the number of records that SESAM is to read into a buffer when the first FETCH statement after OPEN is executed (block). A subsequent FETCH statement does not have to access the database. *n* is an integer of the type SMALLINT. *n* must be greater than or equal to 2 and less than or equal to 32000. If *l* is the sum of the lengths of all the selected row elements, then *n * l* should be less than or equal to 30000. If you are outputting to the screen, the number of rows that can be displayed on a screen can be used as a guideline. Depending on *l*, less than *n-1* rows may be read into the block buffer.

FOR clause

The FOR clause can only be omitted in program mode in a static cursor declaration. If it is omitted, a **variable cursor** is declared. This type of cursor is not made known to the database system until a subsequent dynamic declaration with a FOR clause is executed. In the case of a dynamic declaration with EXECUTE, you must specify a FOR clause. Except for the FOR clause, the static and dynamic declarations for a cursor must be the same. The block factor n within a PREFETCH clause can, however, vary. You can specify any of the other cursor statements (OPEN, FETCH, CLOSE, DROP CURSOR, STORE, RESTORE, UPDATE, DELETE, CYCLE) statically for the variable cursor. This leads to an improvement in performance (see the “DRIVE Programming Language” manual [2], chapter 4.6.1, “Dynamic SQL statements”).

cursor_description

Declares a static or dynamic cursor.

cursor_description defines the derived table and the attributes of the cursor. The earliest point at which a row in the derived table can be selected is when you open the cursor with OPEN. The latest point at which a row can be selected is when you execute a FETCH statement.

query_expression

Query expression for selecting rows and columns from base tables or views.

The values of the variables in *query_expression* are not determined until the cursor is opened. The literals CURRENT_USER and SYSTEM_USER and time functions that are used in *query_expression* are not evaluated until the cursor is opened.

ORDER BY clause

The ORDER BY clause indicates the columns according to which the derived table is to be sorted. First of all, the rows are sorted according to the values in the column specified. If two or more rows have the same values in that column according to the comparison rules (see metavariable *predicate*), these rows are sorted according to the values in the second sort column and so on. In SESAM/SQL, NULL values are considered smaller than all non-NULL values for sorting purposes.

The order of rows with the same value in all the sort columns is undefined.

ORDER BY omitted:

The order of the rows in the cursor table is undefined.

column

Name of the column according to which the table is to be sorted. The column must be part of the derived table created by *query_expression*.

You can specify an atomic column for *column*. The column name cannot be qualified with a table specification and cannot include a range.

column(pos_no)

Element of a multiple column that is to be taken as the basis for the sorting operation. The column element must be part of the derived table created by *query_expression*.

pos_no is an unsigned integer indicating the position number of the column element in the multiple column.

column_number

Number of the column to be used as the basis for sorting.

column_number is an unsigned integer where

$1 \leq \text{column_number} \leq \text{number of derived columns}$.

By specifying a column number, you can also use columns that do not have a name, or which do not have a unique name, as the basis for sorting.

column_number can be an atomic column or a multiple column with the dimension 1.

ASCENDING

ASCENDING is the default value.

The values in the column involved are sorted in ascending order.

DESCENDING

The values in the column involved are sorted in descending order.

FOR UPDATE

For static or dynamic cursors.

You can only use the FOR UPDATE clause for an updatable cursor (see below). In particular, the FOR UPDATE and ORDER BY clauses are mutually exclusive. You use a FOR UPDATE clause to specify which columns in the underlying table can be updated via the cursor with UPDATE...WHERE CURRENT OF.

You can only specify FOR UPDATE if you have not included a SCROLL clause in the cursor description of *cursor*.

FOR UPDATE is not permitted for PREFETCH cursors.

FOR UPDATE omitted:

If the cursor is updatable, you can update all the columns of the underlying table with UPDATE...WHERE CURRENT.

OF {*column*},...

Only the specified columns can be updated with UPDATE...WHERE CURRENT OF. For *column*, specify the name of a column in the table that the updatable cursor references. *column* is the unqualified name of the column in the underlying table, regardless of whether a new column name was defined in the query expression of the cursor description.

OF *column*,... omitted:

Each column in the underlying table can be updated with UPDATE...WHERE CURRENT OF.

Updatable cursor

Only updatable cursors can be used with the UPDATE... WHERE CURRENT OF... or DELETE... WHERE CURRENT OF statements to perform updates or deletions. A cursor is updatable if its cursor description is updatable, i.e. the underlying query expression is updatable, and no ORDER BY clause is specified (see metavariable *query_expression*). No SCROLL clause can be specified in the cursor declaration. If you specify the PREFETCH clause, an updatable cursor cannot be used for updating or deleting.

Example

This example is a cursor declaration with a variable cursor description. A static cursor is specified with a DECLARE ... CURSOR statement without a FOR clause.

```
DECLARE cur_disp1 SCROLL CURSOR;
```

An EXECUTE statement is used to declare the cursor description dynamically at execution time. (The expression in quotes can be up to 256 characters long, otherwise you will have to use CONCAT. DRIVE/WINDOWS includes the blanks for indenting the code in the count.)

```
EXECUTE 'DECLARE cur_disp SCROLL CURSOR FOR '||
        'SELECT country, last_name, first_name, salary '||
        'FROM db_employee '||
        'WHERE (country = &hcountry) ' ||
        ' AND (salry >= &s_lower) AND (salary <= &s_upper);';
```

All statements that reference the cursor can be specified statically in the program.

```

CYCLE cur_disp INTO &disp_row.*;
DISPLAY FORM LINE &disp_row;
END CYCLE;
DROP CURSOR cur_disp;
...

COMMIT WORK;

EXECUTE 'DECLARE cur_update CURSOR FOR ' ||
        'SELECT country, last_name, first_name, salary ' ||
        'FROM db_employee ' ||
        'WHERE (country = &hcountry) AND
        (salary = &highest);';

```

A COMMIT WORK should be included between the DROP statement and an EXECUTE 'DECLARE ...' statement on the same cursor name (in accordance with static cursor declaration without a FOR clause).

Example

An updatable cursor *cur* is declared. The underlying table is *tab*. Only column *col* in table *tab* can be updated via cursor *cur*. To do this, a FOR UPDATE clause with the column name *col* is specified in the cursor description.

```

DECLARE cur CURSOR FOR
    SELECT corr.col AS column FROM tab AS corr
    FOR UPDATE OF col;

```

The unqualified, original column name *col* is used in the FOR UPDATE clause although the column is renamed in the SELECT list, and the table is renamed in the FROM clause.

Example

A static cursor is declared with an input variable.

```

DECLARE cur_order CURSOR FOR
    SELECT order_num, order_date, order_text, order_stat
    FROM orders
    WHERE cust_num >= &CUST_NUM;

```

The cursor description with the current value of &CUST_NUM is evaluated for OPEN *cur_order*. When RESTORE *cur_order* is executed, the cursor description of the last OPEN *cur_order* statement remains valid.

DELETE - Delete rows

You use DELETE to delete rows from a table.

If you want to delete a row from the specified table, you must own the table or have the DELETE privilege for this table. In addition, the transaction mode of the current transaction must be READ WRITE.

If integrity constraints have been defined for the table or columns involved, these are checked after the delete operation has been performed. If the integrity constraint has been violated, the deletion is canceled and an appropriate SQLSTATE set.

```
DELETE FROM table [ WHERE { condition | CURRENT OF cursor } ]
```

table

Name of the table from which rows are to be deleted. The table can be a base table, an updatable view or an updatable temporary view (see metavariable *table_specification*). Static temporary views cannot be specified for dynamic cursors.

WHERE clause

Indicates which rows are to be deleted.

WHERE *condition* deletes a set of rows (multiple DELETE statement), WHERE CURRENT OF *cursor* deletes a single row.

WHERE omitted:

All the rows in the table are deleted.

condition

Condition that the rows to be deleted must satisfy. A row is only deleted if it satisfies the specified *condition*.

Column specifications in *condition* that are outside of subqueries can only reference the specified *table*.

Subqueries in *condition* cannot reference the base table from which the rows are to be deleted either directly or indirectly.

CURRENT OF *cursor*

Name of the cursor used to select the rows to be deleted. The cursor must be updatable (see the DECLARE statement), and *table* must be the underlying table.

The cursor must be defined in the same compilation unit and must be opened with OPEN or RESTORE and positioned on a row in the derived table with FETCH before the DELETE statement is executed.

DELETE deletes the row at the current cursor position from *table*.

After DELETE, the cursor is positioned before the next row in the derived table or after the last row if the end of the table has been reached. If you want to execute another DELETE...WHERE CURRENT OF statement, you must first position the cursor on a row in the derived table with FETCH.

DELETE is not permitted if *cursor* is a PREFETCH cursor.

DELETE and transaction management

SQL only allows you to delete rows within transactions. You can control the effect of the deletion operation on a transaction by defining an isolation level with SET TRANSACTION for concurrent transactions. If an error occurs during the execution of the DELETE statement, any deletions already performed are canceled.

Examples

1. All customers situated in Hanover are to be deleted from the `customers` table.

```
DELETE FROM customers WHERE city = 'Hanover'
```

2. In the example below, a cursor is used to delete customers situated in Hanover from the `customers` table.

```
DECLARE cur_customers CURSOR FOR
  SELECT cust_num, company, city FROM customers
  WHERE city = 'Hanover'
  FOR UPDATE;
```

```
OPEN cur_customers;
```

All the rows found can be deleted with a series of FETCH and DELETE statements.

```
FETCH cur_customers INTO &Cust_num, &Company, &City;
DELETE FROM customers WHERE CURRENT OF cur_customers;
```


DROP CURSOR - Release cursor description

You use DROP CURSOR or DROP CURSORS in DRIVE interactive mode or within EXECUTE (only if the cursor was also declared within EXECUTE) to release a cursor. You can also use the DROP CURSOR statement in program mode for variable cursors (see DECLARE statement).

DROP CURSOR(S) is subject to transaction management, i.e. if you specify ROLLBACK WORK, the cursors specified in DROP CURSOR(S) are not released.

You can use DECLARE... CURSOR several times within a transaction for the same cursor if you execute a DROP CURSOR statement between DECLARE... CURSOR statements. If DROP CURSOR and DECLARE ... CURSOR statements for the same cursor occur in a transaction, the first of these statements cannot be DROP CURSOR.

You can only delete a variable cursor explicitly (static or dynamic) with DROP CURSOR *cursor*. If you execute DROP for a variable cursor, the variable cursor description (FOR clause) is deleted. This means that you can declare the cursor description of the cursor again dynamically after a DROP statement. A COMMIT WORK statement should be included between DROP and dynamic declarations.

```
DROP { CURSOR cursor | CURSORS }
```

cursor

The specified *cursor* is released. If the cursor is open, an implicit CLOSE is executed.

CURSORS

All the cursors defined in interactive mode or all non-variable cursors defined with EXECUTE at the same program level are released.

The DROP CURSORS statement is equivalent to a sequence of DROP CURSOR statements in which all the cursor names are specified explicitly.

Example

```
DECLARE c1 CURSOR;  
EXECUTE 'DECLARE c1 CURSOR FOR ' || &SEARCH1;  
...  
DROP CURSOR c1;  
COMMIT WORK;  
EXECUTE 'DECLARE c1 CURSOR FOR ' || &SEARCH2;
```

DROP SCHEMA - Delete schema

You use DROP SCHEMA to delete an empty database schema. You must delete all the base tables, integrity constraints and views of the schema beforehand.

The SCHEMAATA view of the INFORMATION_SCHEMA provides you with information on which schemas have been defined (see chapter 8, "Information schemas", in the "SESAM/SQL-Server SQL Reference Manual, Part 1" [18]).

The current authorization identifier must own the schema.

```
DROP SCHEMA [ catalog . ] schema RESTRICT
```

schema

Name of the schema. The schema must be empty.

You can qualify the name of the schema with a database name.

DROP TABLE - Delete base table

You use DROP TABLE to delete a base table and the associated indexes, provided that there are no rows in the table. You cannot delete a base table if it is used in a view or in an integrity constraint. All the temporary views defined for the base table are deleted.

The BASE_TABLES view of the INFORMATION_SCHEMA provides you with information on which base tables have been defined (see chapter 8, “Information schemas”, in the “SESAM/SQL-Server SQL Reference Manual, Part 1” [18]).

The current authorization identifier must own the schema to which the table belongs.



This statement can destroy declaration statements in a DRIVE program.

DROP TABLE *table* RESTRICT

table

Name of the base table to be deleted. The table must be empty. You can qualify the name of the table to be deleted with a database and schema name.

DROP TEMPORARY VIEW - Delete temporary view

You use DROP TEMPORARY VIEW to delete the definition of a temporary view. All the dynamic cursors that reference this view are deleted implicitly along with the view.

This statement is only permitted as a static statement in interactive mode. As a dynamic statement in program mode, it must refer to a dynamically defined view (with EXECUTE) of the appropriate DRIVE program.

The current authorization identifier must own the temporary view.

```
DROP TEMPORARY { VIEW table | VIEWS }
```

table

Name of the temporary view to be deleted.

VIEWS

All the views defined in interactive mode or all the view defined at the same program level with EXECUTE are released together with the cursors associated with them.

The DROP TEMPORARY VIEWS statement is equivalent to a sequence of DROP TEMPORARY VIEW statements in which all the view names are specified explicitly. DROP TEMPORARY VIEWS is therefore only executed if the current authorization identifier owns all the views.

DROP VIEW - Delete view

You use DROP VIEW to delete the definition of a (permanent) view. You cannot delete a view if it is used in another view definition.

The VIEWS view of the INFORMATION_SCHEMA provides you with information on which views have been defined. Information on the tables a view uses is provided in the VIEW_TABLE_USAGE view of the INFORMATION_SCHEMA (see chapter 8, “Information schemas”, in the “SESAM/SQL-Server SQL Reference Manual, Part 1” [18]).

The current authorization identifier must own the schema to which the view belongs.

```
DROP VIEW table RESTRICT
```

table

Name of the view to be deleted.

FETCH - Position cursor and read row

You use FETCH to position a cursor. The new cursor position is either on a row, before the first row or after the last row of the cursor table. If the new cursor position is on a row in the cursor table, this row is the current row and the column values of this row are read.

In program mode, the column values in the current row are passed to the variable(s) specified (see INTO clause). In interactive mode, they are displayed on the screen (see chapter 6, “Interactive data access”, in the “DRIVE Programming System” manual [1]).

If no row is read for FETCH because the specified position does not exist, an appropriate SQLSTATE is set, which can be handled with WHENEVER &DML_STATE IN ('TABLE END').

If you declare a cursor with SCROLL, the cursor can be positioned with FETCH on any row in the cursor table and in any order. If you do not specify SCROLL, the declared cursor can only be positioned on the next row (FETCH NEXT...).

The cursor declaration with DECLARE must be located in the same compilation unit.

The cursor must already have been opened or restored (see the OPEN and RESTORE statements).

There must be no backup status of the cursor created with a STORE statement when the FETCH statement is executed.

If block mode has been declared for the cursor (see DECLARE ... CURSOR FOR ... and PRAGMA), you can only position the cursor with FETCH NEXT. If a (static) FETCH [NEXT] statement has already been executed for a static PREFETCH cursor, only this exact FETCH statement is subsequently permitted until the next CLOSE or COMMIT WORK, i.e. the same statement in a loop (see the CYCLE statement in the “Directory of DRIVE Statements” [3]) or in an internal subprogram (see the SUBPROCEDURE statement in the “Directory of DRIVE Statements” [3]).

```

FETCH [ { NEXT | PRIOR | FIRST | LAST | RELATIVE n | ABSOLUTE n } ]
      [ FROM ] cursor

      [ INTO { variable },... ]

      n ::= { [ { + | - } ] unsigned_integer | variable }

```

NEXT

NEXT is the default value.

Positions the cursor on the next row in the cursor table. If you declared the cursor without SCROLL or PREFETCH, you can only use the NEXT clause.

If the cursor is located on the last row in the cursor table, it is positioned after the last row. If it is already positioned after the last row, its position remains unchanged.

PRIOR

Positions the cursor on the preceding row of the cursor table.

If the cursor is positioned on the first row of the cursor table, it is positioned before the first row. If it is already positioned in front of the first row, its position remains unchanged.

You can only specify PRIOR if you declared the cursor with SCROLL.

FIRST

Positions the cursor on the first row of the cursor table or before the first row if the cursor table is empty.

You can only specify FIRST if you declared the cursor with SCROLL.

LAST

Positions the cursor on the last row of the cursor table or after the last row if the cursor table is empty.

You can only specify LAST if you declared the cursor with SCROLL.

ABSOLUTE n

Specify the position of the cursor. You can only specify ABSOLUTE if you declared the cursor with SCROLL.

For n you can specify an integer or a variable of the DRIVE data type INTEGER or SMALLINT. The cursor position is determined by the value of n as follows:

- >0 The cursor is positioned on the n th row of the cursor table or after the last row if $n >$ number of rows in the cursor table.
- 0 The cursor is positioned before the first row of the cursor table.
- <0 The cursor is positioned on the $(N+1-|n|)$ th row of the cursor table, where N is the number of rows in the cursor table. If $|n| > N$, the cursor is positioned before the first row.

Example:

FETCH ABSOLUTE -1 and FETCH LAST are equivalent.

RELATIVE *n*

Position of the cursor relative to its current position. You can only specify RELATIVE if you declared the cursor with SCROLL.

For *n* you can specify an integer literal or a variable of the type INT or SMALLINT. The cursor position is determined by the value of *n* as follows:

- >0 The cursor is positioned on the row that is *n* rows after its current position. If the new position is greater than the number of rows in the cursor table, the cursor is positioned after the last row.
- 0 The cursor position remains unchanged.
- <0 The cursor is positioned on the row that is *n* rows in front of its actual position. If the new position is ≤ 1 , the cursor is positioned before the first row.

[FROM] *cursor*

Name of the cursor.

INTO clause

Indicates where the values read are to be stored. You must specify the clause in program mode. This clause is not permitted in interactive mode.

variable

Name of a variable to be assigned a column value from the derived row.

The data type of a variable must be compatible with the data type of the corresponding output value. If an output value is an aggregate with several elements, the corresponding variable must be a vector with the same number of elements.

The number of specified variables must match the number of columns in the SELECT list of the cursor description. If this is not the case, DRIVE/WINDOWS issues error message DRI0504. The value of the *n*th column in the SELECT list, which may be the NULL value, is assigned to the *n*th variable in the INTO clause.

Behavior of SESAM/SQL in the event of an error

If an error occurs when a value is read (e.g. numeric value is too big for the target data type), the cursor is moved to its new position but the assigned values are undefined.

In the event of other errors (e.g. incompatible data types), the position of the cursor remains unchanged and no values are read.

Examples

Example of FETCH NEXT in program mode:

```
FETCH cur_order
  INTO &ORDER_NUM,
      &ORDER_DATE,
      &ORDER_TEXT,
      &ORDER_STAT;
```

Example of FETCH with a static PREFETCH cursor:

```
DCL cur_order CURSOR PREFETCH...
      FOR SELECT order_num, order_date, order_text, order_stat
      FROM order
      WHERE cust_num >= &CUST_NUM;

...

CYCLE cur_order INTO &ORDER_NUM,
                    &ORDER_DATE,
                    &ORDER_TEXT,
                    &ORDER_STAT;

...

END CYCLE;
```

GRANT - Grant privileges

You use GRANT to grant table and column privileges for base tables and views (DML rights and referential rights), and special privileges for databases. If the GRANT statement is included in a CREATE SCHEMA statement, you cannot grant special privileges with GRANT. The privileges can be granted to SQL users specified by means of authorization identifier or to all users.

The current authorization identifier must be authorized to grant the specified privileges:

- It is the authorization identifier of the universal user, i.e. the owner of the database.
- It is the owner of the schema to which the table belongs.
- It has GRANT authorization for granting the privileges to other users.

Information on which authorization identifiers are schema owners is stored in the SCHEMATA view. The TABLE_PRIVILEGES, COLUMN_PRIVILEGES and CATALOG_PRIVILEGES provide you with information on whether the authorization identifier has GRANT authorization for a certain privilege (see chapter 8, “Information schemas”, in the “SESAM/SQL-Server SQL Reference Manual, Part 1” [18])

Only the authorization identifier that granted a privilege can revoke that privilege.



DRIVE/WINDOWS provides limited support for the GRANT statement of SESAM/SQL:

- all table and column privileges, as a whole or individually
- all special privileges for databases as a whole
- the special privilege CREATE SCHEMA

The GRANT statement has two formats: one for granting table and column privileges and one for granting special privileges.

GRANT format for table and column privileges:

```
GRANT { ALL PRIVILEGES | { table_and_column_privilege },... }
ON [ TABLE ] table
TO { PUBLIC | { authorization_id },... }
[ WITH GRANT OPTION ]
table_and_column_privilege ::= { SELECT | DELETE | INSERT |
                                UPDATE [ ( { column },... ) ] |
                                REFERENCES [ ( { column },... ) ] }
```

ALL PRIVILEGES

All the table and column privileges that the current authorization identifier can grant are granted. ALL PRIVILEGES comprises the privileges SELECT, DELETE, INSERT, UPDATE and REFERENCES.

table_and_column_privilege

The table and column privileges are granted individually. You can specify more than one privilege.

SELECT

Privilege that allows rows in the table to be read.

DELETE

Privilege that allows rows to be deleted from the table.

INSERT

Privilege that allows rows to be inserted into the table.

UPDATE [(*column*,...)]

Privilege that allows rows in the table to be updated.

The update operation can be limited to the specified columns. *column* must be the unqualified name of a column in the specified table. You can specify more than one column.

(*column*,...) omitted:

All the columns in the table can be updated including columns inserted later.

REFERENCES [(column,...)]

Privilege that allows the definition of referential constraints that reference the table (see CREATE TABLE and ALTER TABLE statements).

The reference can be limited to the specified columns. *column* must be the unqualified name of a column in the specified table. You can specify more than one column.

(*column,...*) omitted:

All the columns in the table can be referenced including columns inserted later.

ON [TABLE] *table*

Name of the table for which you want to grant privileges.

If you use the GRANT statement in a CREATE SCHEMA statement, you can only qualify the table name with the database and schema name from the CREATE SCHEMA statement.

The table can be a base table or a view. You can only grant the SELECT privilege for a view that cannot be updated.

TO PUBLIC

The privileges are extended to all authorization identifiers. Each authorization identifier is granted the privileges extended to PUBLIC in addition to its own privileges. These privileges are also extended to any authorization identifiers added later.

TO {*authorization_id*},...

The privileges are granted to *authorization_id*. You may specify more than one authorization identifier. The authorization identifier can be up to 18 characters long.

WITH GRANT OPTION

The specified authorization identifier(s) is granted not only the specified privileges but also GRANT authorization. This means that the authorization identifier(s) is authorized to grant the privileges it has been extended to other authorization identifiers. Once these privileges have been granted, they cannot be revoked. You cannot specify the WITH GRANT OPTION clause together with PUBLIC.

WITH GRANT OPTION omitted:

The specified authorization identifier(s) cannot grant the privileges it has been extended to other authorization identifiers.

GRANT format for special privileges:

```
GRANT { ALL SPECIAL PRIVILEGES | CREATE SCHEMA }  
  
ON CATALOG catalog  
  
TO { PUBLIC | { authorization_id },... }  
  
[ WITH GRANT OPTION ]
```

ALL SPECIAL PRIVILEGES

All the special privileges that the current authorization identifier can grant for the database specified in the ON clause are granted. ALL SPECIAL PRIVILEGES comprises the special privileges CREATE SCHEMA (see below), CREATE USER, CREATE STOGROUP and UTILITY (see chapter 8, "Information schemas", in the "SESAM/SQL-Server SQL Reference Manual, Part 1" [18]).

CREATE SCHEMA

Special privilege that permits definition of database schemas.

ON CATALOG *catalog*

Name of the database for which you are granting special privileges.

TO PUBLIC

The privileges are extended to all authorization identifiers. Each authorization identifier is granted the privileges extended to PUBLIC in addition to its own privileges. These privileges are also extended to any authorization identifiers added later.

TO {*authorization_id*},...

The privileges are granted to *authorization_id*. You may specify more than one authorization identifier. The authorization identifier can be up to 18 characters long.

WITH GRANT OPTION

The specified authorization identifier(s) is granted not only the specified privileges but also GRANT authorization. This means that the authorization identifier(s) is authorized to grant the privileges it has been extended to other authorization identifiers. Once these special privileges have been granted, they cannot be revoked. You cannot specify the WITH GRANT OPTION clause together with PUBLIC.

WITH GRANT OPTION omitted:

The specified authorization identifier(s) cannot grant the privileges it has been extended to other authorization identifiers.

Example

The first GRANT statement grants several table privileges, the second grants the special privilege CREATE SCHEMA to an existing authorization identifier.

```
GRANT SELECT,INSERT,UPDATE ON TABLE telephone_list TO bertha;
```

```
GRANT CREATE SCHEMA ON CATALOG my_db TO hugh;
```

INSERT - Insert rows in table

You use INSERT to insert one or more rows in an existing table.

If you want to insert a row in the specified table, you must either own the table or have the INSERT privilege for the table. In addition, the transaction mode of the current transaction must be READ WRITE.

The literals CURRENT USER or USER, and SYSTEM USER and the time functions CURRENT DATE, CURRENT TIME and CURRENT TIMESTAMP in the INSERT statement (and in the defaults) are evaluated once, and the calculated values are valid for all insertions.

If integrity constraints have been defined for the table or the columns involved, these are checked after the row(s) has been inserted. If an integrity constraint has been violated, the insertion is canceled and an appropriate SQLSTATE set.

```
INSERT INTO table
```

```

{ [ ( { column | column(pos_no) | column(min-max) },... ) ]
  { VALUES ( { sql_expression | DEFAULT | NULL | * },... ) |
    VALUES { sql_expression | DEFAULT | NULL | * } |
    query_expression } |
  DEFAULT VALUES }

```

```
[ RETURN INTO variable ]
```

table

Name of the table into which the rows are to be inserted. The table can be a base table, an updatable view, or an updatable temporary view (see metavariable *table_specification*).

column

Atomic column into which a value is to be inserted.

column is a column in the specified table. You cannot qualify the column name with the name of a table. The order in which you specify the columns does not have to be the same as the order of the columns in the table.

You can only specify an atomic column once in the column list.

column(pos_no)

Element of a multiple column that is to receive the value.

The multiple column must be part of the table. If other elements of a multiple column are specified, each column element with a smaller position number must also be specified. Each element can only be specified once.

pos_no is an unsigned integer ≥ 1 .

column(min..max)

Range of column elements in a multiple column that are to be assigned values. The multiple column must be part of the table. The range specified must be selected in such a way that each column element with a position number between 1 and the largest position number specified only occurs once.

min and *max* are unsigned integers ≥ 1 ; *max* must be $\geq min$.

No column specification:

The following specifications supply values for all the columns in the table. The order of columns specified for CREATE TABLE, ALTER TABLE or CREATE VIEW or CREATE TEMPORARY VIEW is valid.

VALUES clause

Indicates the individual values for the previously specified column or for all columns. If only one value is to be entered, you can omit the parentheses.

You can use this form of the INSERT statements to insert a single row into *table*.

If a list of columns has been specified, the VALUES clause must include an appropriate value for each column in the list. Columns not specified are set to the default value if one has been defined or, if this is not the case, to the NULL value.

If there is no list of columns, you must specify an appropriate value for each column in the table.

The assignment rules described in the "SESAM/SQL-Server SQL Reference Manual, Part 1" [18], section 4.4.1, "Entering values in table columns", apply to the assignment of values.

The *n*th value in the VALUES list is assigned to the *n*th column in the column list, if a column list was specified. Otherwise it is assigned to the *n*th column of the table.

sql_expression

Expression whose value is assigned to a column. The value of the expression must be compatible with the data type of the column.

If *sql_expression* is a variable, you can also specify a vector. In this case, the column must be a multiple column and the number of elements in the vector must be the same as the number of column elements.

Subqueries in *sql_expression* cannot refer directly or indirectly to the base table into which the rows are inserted.

If *sql_expression* is an aggregate (see metavariable *value*) that is to be assigned to a multiple column, the number of values must be the same as the number of column elements, and the data type of each component of the aggregate must be compatible with the data type of the target column.

DEFAULT

Only for an atomic column.

The corresponding column is assigned the default value. The default was specified when the column was defined. If no default has been defined, the column is assigned the NULL value.

NULL

Only for an atomic column.

The corresponding column is assigned the NULL value.

*

Only for an atomic column.

Value specification for a column for which SESAM/SQL determines the value (count column).

The column must have an integer or fixed-point data type (SMALLINT, INT, DECIMAL, NUMERIC) and must be part of a primary key. The column cannot be used in a referential constraint or in a check constraint on *table*. SESAM/SQL assigns a value to the column that ensures that the primary key within the table remains unique.

* can only be specified once in the VALUES clause.

query_expression

The values to be inserted are specified via a query expression.

You can use this form of the INSERT statement to insert several rows into *table* (multiple INSERT statement).

query expression is a query expression whose derived table contains the rows to be inserted. If *query expression* returns an empty table, no rows are inserted and an appropriate SQLSTATE is set, which can be handled with WHENEVER &DML_STATE IN ('TABLE END').

If a list of columns has been specified, the number of columns in the derived table must be the same as the number of columns in the list. Columns not specified are set to the default value if one has been defined or, if this is not the case, to the NULL value.

If no list of columns is specified, an appropriate column must exist in the derived table for each column in the table.

The value in the *n*th derived column is assigned to the *n*th column in the column list, if a column list was specified. Otherwise it is assigned to the *n*th column of the table.

Each value must be compatible with the data type of the corresponding column.

You cannot specify a table that references the base table into which you are inserting rows in the FROM clauses or subqueries of the query expression. In particular, you cannot specify *table*.

DEFAULT VALUES

Insert a row into *table* that consists only of the column-specific default values (see metavariable *column-definition*).

The column for which a default value has been defined are assigned the default value. Columns for which there is no default are assigned the NULL value.

RETURN INTO

The value of the column assigned a value by SESAM/SQL because * was specified in the VALUES clause is stored in a variable.

You can only use the RETURN INTO clause, if the VALUES clause contains an asterisk (*).

variable

Name of the variable into which the value of the count column is entered. *variable* must have an integer or floating-point data type (SMALLINT, INT, DECIMAL, NUMERIC).

Inserting values for multiple columns

In the case of a multiple column, you can insert values for individual column elements or for ranges of elements.

An element of a multiple column is identified by its position number in the multiple column.

A range of elements in a multiple column is identified by the position numbers of the first and last element in the range.



The position of an element in a multiple column can change (see the UPDATE statement).

INSERT and integrity constraints

By specifying integrity constraints when you define the base table, you can restrict the range of values for the corresponding column. The values specified in the INSERT statement must satisfy the defined integrity constraint.

INSERT and transaction management

INSERT initiates a transaction if no transaction is open. If you define an isolation level for concurrent transactions, you can control how the INSERT statement affects these transactions.

If an error occurs during insertion, any rows that have already been inserted are removed.

Example

You want to insert a new row in the `orders` table.

```
INSERT INTO orders (order_num, cust_num, contact_num, order_text)
VALUES (500, 105, 35, 'Consultancy');
```

OPEN - Open cursor

You use OPEN to open a cursor declared with DECLARE.

- The variables in the cursor description are evaluated.
- The literals CURRENT USER or USER and SYSTEM USER, as well as the time functions CURRENT DATE, CURRENT TIME and CURRENT TIMESTAMP in the cursor description are evaluated.

All the values returned contain the same date and/or time. These values are valid for the cursor table as long as the cursor is open, and if the cursor is reopened with RESTORE.

After the OPEN statement, the cursor is positioned before the first row in the derived table, even if the previous cursor position was saved with STORE. A previously saved cursor position cannot be restored with RESTORE after an OPEN statement.

A cursor can only be referenced in the compilation unit in which it was declared with DECLARE. A static cursor declaration with DECLARE must physically precede the OPEN statement in the program text.

In the case of a dynamic cursor, the cursor description must be declared before the OPEN statement is executed.

The cursor must be closed.

You can close an open cursor with one of the following statements:

- CLOSE
- COMMIT WORK
- DROP CURSOR
- DROP TEMPORARY VIEW, if the cursor references the temporary view involved

OPEN cursor

cursor

Name of the cursor to be opened.

Example

You want to read all orders completed before 1.1.1994.

```
DECLARE cur_complete CURSOR FOR
  SELECT cust_num,order_text FROM orders WHERE actual < '1994-01-01';
OPEN cur_complete;
```

PERMIT - Specify user identification for old style

In order to allow programs created with SESAM/SQL V1.x to run without you having to modify them, the PERMIT statement is still allowed. Execution of a PERMIT statement in new-style operation does not, however, have any effect. A SESAM/SQL V1.x new-style program can only be executed successfully under the current version of SESAM/SQL if the current authorization identifier is set with a valid system entry (see OPTION AUTHORIZATION, SET AUTHORIZATION and PARAMETER DYNAMIC AUTHORIZATION) for which the appropriate privileges for the referenced SQL and CALL DML/SQL tables have been defined with GRANT.

Old-style procedures still need a password for accessing password-protected CALL DML tables (see the section dealing with SESAM password protection in the “SESAM/SQL V1, Creation and Maintenance” manual [46]). This user identification can be made available in DRIVE/WINDOWS V1.1 with the PERMIT statement for old-style or mixed operation. The first DO or CALL on a SESAM old-style procedure during the DRIVE session (TIAM session or UTM conversation) causes the last PERMIT password entered to be passed to the old-style runtime system. This password is then valid for all old-style accesses during the DRIVE session. This means, in particular, that in mixed mode various password-protected CALL DML tables can only be accessed with one password, valid for the entire session.

You must recompile a SESAM/SQL V1.x program before you run it under the current version of SESAM/SQL (see also the manuals “SESAM/SQL-Server, Migrating SESAM Databases and Applications to SESAM/SQL-Server” [22] and “DRIVE Programming System” [13]).

The PERMIT statement does not initiate a transaction.

```
PERMIT SCHEMA = { table | variable } [ PASSWORD = value ]
```

table

Unqualified name of a CALL DML table (see metavariable *table_specification*).



The names of the CALL DML tables that can be accessed can be determined from the DBH start statement ADD-OLD-TABLE-CATALOG-LIST.

variable

See metavariable *variable*. Contains the unqualified name of a CALL DML table.

PASSWORD = *variable*

Value assignment for a password (see metavariable *value*). *value* must be an alphanumeric variable or a literal that contains the password. The password can be up to three characters long and must observe SESAM conventions. In program mode, *value* must be specified as a variable.

PASSWORD omitted:

The default declaration: PASSWORD = ' ' is valid.

Example

```
PERMIT SCHEMA = "GEN-CALLRBT-2"  
      PASSWORD = 'XX3';
```

PRAGMA - Declare pragma clauses

You use PRAGMA to declare pragma clauses. DRIVE/WINDOWS converts the declared clauses into pragmas for SESAM/SQL, i.e. special SQL comments that can be used to influence or monitor the execution of SQL statements. You will find a brief description of the application possibilities and advantages of pragmas in section "Pragmas" on page 62. Refer to the SESAM/SQL-Server manuals [18] and [45] for more detailed information.

The PRAGMA statement is not executable and therefore does not initiate a transaction.

The PRAGMA statement is only permitted in program mode. A static PRAGMA statement can be included anywhere between PROCEDURE and END PROCEDURE. A dynamic PRAGMA statement can be included wherever EXECUTE is permitted. A static PRAGMA is evaluated at compilation time (i.e. is not executable) and in DRIVE/WINDOWS influences the next static SQL statement in the program text and only this statement. In DRIVE, this causes the SQL statement is prefixed with the special comment

```
--%PRAGMA { pragma_clause },..."
```

The effect of the PRAGMA statement in DRIVE therefore corresponds to a conversion of the pragma clauses into SESAM-compliant usage as in ESQL/COBOL programs and in the Utility Monitor.

Whether the effect in DRIVE leads to an effect in SESAM depends on whether all the pragma clauses influence the SQL statement (see below). A dynamic PRAGMA statement is evaluated at execution time and in DRIVE/WINDOWS influences the next chronological SQL statement and only this statement. At the user interface, the effect in DRIVE corresponds to that of static PRAGMA statements. Static PRAGMA statements therefore only influence static SQL statements, and dynamic PRAGMA statements only influence dynamic SQL statements.

```
PRAGMA literal
```

```
literal ::= '{ pragma_clause },...'
```

```
pragma_clause ::= { PREFETCH n |
                  EXPLAIN INTO file |
                  IGNORE INDEX index_name |
                  OPTIMIZATION LEVEL n |
                  SIMPLIFICATION { ON | OFF } |
                  ISOLATION LEVEL
                  { READ UNCOMMITTED |
                    READ COMMITTED |
                    REPEATABLE READ |
                    SERIALIZABLE } |
                  DATA TYPE OLDEST |
                  CHECK { ON | OFF } }
```

literal

An alphanumeric literal containing a list of pragma clauses. DRIVE/WINDOWS does not check the contents of the literal.

pragma_clause

Syntactically valid pragma clause for SESAM/SQL.

PREFETCH only has an effect in SESAM for DECLARE statements and indirectly for the corresponding FETCH statements. EXPLAIN INTO, IGNORE INDEX, OPTIMIZATION LEVEL, SIMPLIFICATION and ISOLATION LEVEL only influence the statements DECLARE, SELECT, INSERT, UPDATE and DELETE in SESAM.

DATA TYPE and CHECK do not have any effect on SESAM in DRIVE/WINDOWS.



If a *literal* in a pragma clause does not have any effect in SESAM, an error with &DML_STATE = 'SQL ERROR' and &SQL_CLASS = '01' occurs in DRIVE/WINDOWS when the corresponding SQL statement is compiled (static PRAGMA) or executed (dynamic PRAGMA).

PREFETCH pragma clause

PREFETCH controls the block mode of the SQL statement FETCH (position cursor). Block mode accelerates execution of the FETCH statement. It is only effective if FETCH is used to position the cursor on the next row in the cursor table (FETCH NEXT).

You can use the PREFETCH pragma to activate block mode and specify a blocking factor (n). When the first FETCH NEXT... statement is executed, the column values of the current row are read and the next $n-1$ rows of the associated cursor table are stored in a buffer. When the next $n-1$ FETCH NEXT... statements that relate to the same cursor are executed, the next row can be accessed directly, without DBH contact.

If the cursor description of a DECLARE statement for static or dynamic cursors includes a FOR UPDATE clause, the PREFETCH pragma is ignored (i.e. it does not have any effect in SESAM), and block mode is not activated.

PREFETCH n

n

Blocking factor. You must specify the blocking factor as an unsigned integer (of the type SMALLINT).

If the blocking factor (n) has a value > 0 , up to $n-1$ rows in the specified cursor table are stored in a buffer. If the blocking factor is the value 0, the PREFETCH pragma has no effect. This means that you can activate the effect of the pragma and thus block mode by specifying a value > 0 for n and deactivate it by specifying the value 0.

The following restrictions apply if block mode has been activated:

Only the FETCH NEXT statement is permitted for the PREFETCH cursor in the same compilation unit. The following SQL statements are no longer executable:

- UPDATE ... WHERE CURRENT OF *cursor*
- DELETE ... WHERE CURRENT OF *cursor*
- STORE *cursor*
- FETCH *cursor* with a cursor position that is different to NEXT
- FETCH *cursor* with an INTO clause that is different to the first FETCH NEXT statement if *cursor* is static.

EXPLAIN pragma clause

EXPLAIN is used to output the access plan selected by the optimizer. You can only use this pragma if the current authorization identifier has the special privilege UTILITY (see the “SESAM/SQL-Server Language Reference Manual, Part 1” [18]).

This pragma is only effective in the following SQL statements in SESAM:

- DECLARE
- DELETE
- INSERT
- SELECT
- UPDATE

This pragma is only effective in a static statement if you precompile the program while the database is online. This is always the case in DRIVE/WINDOWS.

EXPLAIN INTO *file*

file

Name of the SAM file into which the explanation is to be output. If the file already exists, the explanation is appended to the file.

If *file* includes a BS2000 user ID, this user ID is used. If not, the ID of the Database Handler for the database referenced in the SQL statement is used. In both cases the DBH must have write permission for the file.

You specify an alphanumeric literal for *file*.

In the case of dynamic statements, the explanation is output when the EXECUTE statement is executed. For static statements, the explanation is output during precompilation.

The explanation comprises the SQL statement and an edited representation of the access plan. The representation of access plans is described in the “SESAM/SQL-Server Performance” manual [45].

You can display the contents of the file with SHOW-FILE. If you want to read the file with EDT, you must enter the following command:

```
SET-FILE-LINK LINK-NAME=EDTSAM, FILE-NAME=file, . . . , BUFFER-LENGTH=(STD, 2), . . .
```

As of EDT Version 16.5A, you can also enter:

```
@OPEN F=file, TYPE=CATALOG or @OP F=file, T=C
```

Example

```
SET &explainfile = '$YDRI20.EXPLAINFILE';
/* The file $YDRI20.EXPLAINFILE must be cataloged with */
/* USER-ACC = ALL-USERS */
EXECUTE 'SET SESSION AUTHORIZATION '"DRI-USER1" ''';
/* The authorization identifier DRI-USER1 must have the special */
/* privilege UTILITY for the default database */
EXECUTE 'PRAGMA 'EXPLAIN INTO '|| &explainfile || ''';
DISPLAY FORM 'Output of the SQL access plan in file ' || &explainfile;
EXECUTE 'DECLARE C1 CURSOR FOR cursor_description';
```

ISOLATION LEVEL pragma clause

ISOLATION LEVEL determines the isolation level for database accesses performed by an SQL statement.

This pragma is only effective in the following SQL statements in SESAM:

- DECLARE
- DELETE
- INSERT
- SELECT
- UPDATE

```
ISOLATION LEVEL
    { READ UNCOMMITTED |
      READ COMMITTED |
      REPEATABLE READ |
      SERIALIZABLE }
```

The isolation levels are described under the SET TRANSACTION statement.

If you have specified the ISOLATION LEVEL pragma, any database access performed in connection with this statement takes place under this isolation level.



If you specify a lower isolation level than specified for the transaction, the isolation level defined for the transaction is no longer guaranteed.

IGNORE INDEX pragma clause

The specified index is ignored when the join order and join algorithm are specified and when the optimum access path (to base relations) is selected.

```
IGNORE INDEX index
```

OPTIMIZATION LEVEL pragma clause

The option n controls the number of plan alternatives that can be generated and evaluated during access path selection.

OPTIMIZATION LEVEL n

Firstly, all plan variants are examined, and only the most favorable variants that in general promise an improvement of the evaluation costs are selected by means of heuristic techniques. The number of plan variants that are followed up on depends on the value of n . The value of n can be between 1 and 10; the default value is 9. If a value $n \leq 5$ is specified, only one plan variant is examined in each optimization step.

A distinction is made between the following levels:

- $n \leq 9$

Various join orders are considered.

- $n \leq 8$

In the case of a nested loop join, this statement checks whether switching the two join partners would be advantageous.

- $n \leq 7$

Execution of sort minimization.

- $n \leq 6$

In the case of join optimization, not only the sort merge but also the nested loop join is considered.

When selecting the access path, all the possibilities for achieving the required sort are considered (physical sorting in the DBH kernel, sorting via index scan).

- $n \leq 5$

Execution of subquery optimization and storage of intermediate result relations that are needed more than once.

- $n \leq 4$

Execution of the range construction, i.e. several atomic predicates on the same column are grouped together as one index access.

SIMPLIFICATION pragma clause

All the optimization techniques for simplification (part of the algebraic optimization) are activated or deactivated, i.e. the optimization steps outlined in these techniques are either executed in their entirety (ON), or none of them are executed (OFF).

```
SIMPLIFICATION { ON | OFF }
```

DATA TYPE pragma clause

The DATA TYPE pragma indicates that a column can only be created in the attribute format for CALL DML tables.

This clause only has an effect in SESAM if it is specified in the ALTER TABLE ... ADD COLUMN statement and the table is a CALL DML table.

```
DATA TYPE OLDEST
```



Because the ADD COLUMN clause is not allowed for ALTER TABLE involving CALL DML tables in DRIVE/WINDOWS, this clause has no SESAM effect in DRIVE/WINDOWS.

CHECK pragma clause

The pragma clause CHECK has no SESAM effect in DRIVE/WINDOWS.

RESTORE - Restore cursor

You use RESTORE to open a cursor saved with STORE.

The cursor is opened with the same cursor description as for the last OPEN. If variables have been updated in the meantime, this does not have any effect on the resulting derived table.

If the literals CURRENT USER or USER and SYSTEM USER or the time functions CURRENT DATE, CURRENT TIME and/or CURRENT TIMESTAMP are included in the cursor description, they are not reevaluated.

A cursor position saved with STORE can be lost if, in the same or a different transaction, rows starting at the stored position have been deleted in the meantime, or the row on which the cursor was positioned has been updated in such a way that it no longer belongs to the cursor table.

If no cursor position has been saved for the cursor, the cursor is not opened and an appropriate SQLSTATE is set.

Otherwise, the cursor is opened and the cursor position restored. If you want to delete (DELETE ... WHERE CURRENT OF) or update (UPDATE ... WHERE CURRENT OF) a row, the cursor must be positioned on the row with FETCH.

After the RESTORE statement has been executed, all the information on this cursor that has been saved with STORE is deleted. You must save the cursor position again with STORE before a new RESTORE statement can be executed.

The cursor to be restored must be saved with STORE during the same DRIVE session (dialog cursor) or in the same program run (program cursor) and must be closed when RESTORE is executed. The transactions containing the STORE and RESTORE statements must have the same isolation or consistency levels (see SET TRANSACTION statement).

If a dynamic cursor is no longer declared for SESAM when the RESTORE statement is executed, it is declared dynamically by DRIVE/WINDOWS. However, because the cursor is not open, SESAM sets an appropriate SQLSTATE when the RESTORE statement is executed.

RESTORE cursor

cursor

Name of the cursor to be restored.

Processing the cursor after RESTORE

After a RESTORE statement, you must position the cursor on a row with FETCH.

Example

FETCH NEXT positions to the next row in the cursor table.

Only then can the cursor be accessed with an UPDATE or DELETE statement.

REVOKE - Revoke privileges

You use REVOKE to revoke table and column privileges or special privileges from one or more, or from all, authorization identifiers. Temporary views belonging to the authorization identifiers that are based on the table are deleted.

Only the authorization identifier that granted a privilege can revoke that privilege from an authorization identifier (see the GRANT statement). If the privileges have been granted to other users, they cannot be revoked. A privilege used as the basis for defining a view or for a referential constraint cannot be revoked.

The TABLE_PRIVILEGES, COLUMN_PRIVILEGES and CATALOG_PRIVILEGES views of the INFORMATION_SCHEMA provide you with information on the privileges assigned to the authorization identifiers (see chapter 8, “Information schemas”, in the “SESAM/SQL-Server SQL Reference Manual, Part 1” [18]).

The REVOKE statement has two formats: one format for table and column privileges and another for special privileges.

REVOKE format for table and column privileges:

```

REVOKE { ALL PRIVILEGES | { table_and_column_privilege },... }
ON [ TABLE ] table
FROM { PUBLIC | { authorization_id },... } RESTRICT

table_and_column_privilege ::= { SELECT |
                                DELETE |
                                INSERT |
                                UPDATE [ ( { column },... ) ] |
                                REFERENCES [ ( { column },... ) ] }

```

ALL PRIVILEGES

All the table privileges that the current authorization identifier can revoke are revoked. ALL PRIVILEGES comprises the privileges SELECT, DELETE, INSERT, UPDATE and REFERENCES.

table_and_column_privilege

The table and column privileges are revoked individually. You can specify more than one privilege.

SELECT

Privilege that allows rows in the table to be read.

DELETE

Privilege that allows rows to be deleted from the table.

INSERT

Privilege that allows rows to be inserted into the table.

UPDATE [(*column*,...)]

Privilege that allows rows in the table to be updated.

The revoke operation can be limited to the specified columns. *column* must be the unqualified name of a column in the specified table. You can specify more than one column.

(*column*,...) omitted:

The privilege for updating all the columns in the table is revoked.

REFERENCES [(*column*,...)]

Privilege that allows definition of referential constraints that reference the table.

The revoke operation can be limited to the specified columns. *column* must be the unqualified name of a column in the specified table. You can specify more than one column.

(*column*,...) omitted:

The privilege for referencing all the columns in the table is revoked.

ON [TABLE] *table*

Name of the table for which you want to revoke privileges.

The table can be a base table or a view. In the case of a view that cannot be updated, you can only revoke the SELECT privilege.

FROM PUBLIC

The privileges are revoked from all authorization identifiers. The individual privileges of the individual authorization identifiers are not affected.

FROM *authorization_id*

The privileges are revoked from the user with the authorization identifier *authorization_id*. You may specify more than one authorization identifier.

REVOKE format for special privileges:

```
REVOKE { ALL SPECIAL PRIVILEGES | CREATE SCHEMA }  
ON CATALOG catalog  
FROM { PUBLIC | { authorization_id },... } RESTRICT
```

ALL SPECIAL PRIVILEGES

All the special privileges that the current authorization identifier can revoke are revoked (see the GRANT statement).

CREATE SCHEMA

Revokes the special privilege that allows you to define database schemas.

FROM PUBLIC

The special privileges are revoked from all authorization identifiers. The individual privileges of the individual authorization identifiers are not affected.

FROM {*authorization_id*},...

The special privileges are revoked from the user with the authorization identifier *authorization_id*. You may specify more than one authorization identifier.

ON CATALOG *catalog*

Name of the database for which special privileges are to be revoked.

Example

The following REVOKE statement revokes the UPDATE privilege for all the columns in the table `telephone_list`.

```
REVOKE UPDATE ON TABLE telephone_list FROM bertha RESTRICT;
```

ROLLBACK WORK - Roll back transaction

You use ROLLBACK WORK to terminate a transaction and undo all the updates performed since the end of the last SQL transaction.

ROLLBACK WORK undoes the following updates:

- all cursors opened in the transaction are closed
- updated data in SQL schemas
- dynamic statements and cursor descriptions
- cursor positions saved with STORE
- SET statements of a (dynamic) program (AUTHORIZATION, CATALOG, SCHEMA) not committed with COMMIT WORK
- the user specification specified with PERMIT for old-style procedures

In interactive mode, ROLLBACK WORK does not roll back the parameter values for SESAM or for DRIVE/WINDOWS (see PARAMETER DYNAMIC statement).

The SET TRANSACTION cannot be rolled back.

Refer to the “DRIVE Programming Language” manual, chapters 12, “Distributed applications”, and 13, “Distributed transaction processing”, for the rules governing distributed applications.

The first error-free SQL statement that initiates a transaction executed after ROLLBACK WORK starts a new transaction.

```
ROLLBACK [ WORK ] [ WITH RESET ]
```

WITH RESET

WITH RESET is only permitted in program mode and refers only to DRIVE control.

The program is rolled back to the status of the last COMMIT WORK and continues with the statement that follows this COMMIT WORK (same behavior as for an internal ROLLBACK WORK). The contents of the DRIVE variables is reset to the values valid at the time of the COMMIT WORK. In addition, in SINIX and MS-Windows, all windows open on the graphical user interface are closed after this COMMIT WORK (see the “DRIVE Programming Language” manual [2], chapter 10, “Transaction concept”). If no COMMIT WORK has been executed since the program was started, the program is aborted and an error message issued.

As far as the database is concerned, the statements ROLLBACK WORK and ROLLBACK WORK WITH RESET are identical.



If the ROLLBACK WORK WITH RESET statement is specified without a condition, there is a danger of an endless loop since the DRIVE program is continued with the statement that follows the last COMMIT WORK statement.

Implicit execution of ROLLBACK WORK

SESAM/SQL rolls back a transaction by implicitly executing a ROLLBACK WORK statement if one of the following situations occur:

- An unrecoverable error occurs in the current transaction.
- The specified isolation level cannot be ensured for two or more transactions that access certain SQL data concurrently (see also the SESAM/SQL-Server “Core Manual” [20]).
- A transaction is interrupted for a long time and is using resources required by other transactions (see also the SESAM/SQL-Server “Core Manual” [20]).

The effect is the same as if ROLLBACK WORK WITH RESET were called explicitly.

SELECT - Read individual rows

You use `SELECT` to read exactly one row in a table. The column values read are stored in variables.

If a derived table would contain more than one row, the `SELECT` statement does not read a row and an appropriate `SQLSTATE` is set. If you want to read derived tables with more than one row, you must use a cursor.

In order to execute a `SELECT` statement, you must own the table in which you are querying values, or you must have the `SELECT` privilege for the referenced table.

In program mode, `SELECT` transfers the column values to the variables specified with `INTO`.

In interactive mode, `SELECT` displays the column values on the screen appropriately according to their type (see the “DRIVE Programming Language” manual [2], chapter 3, “Using variables and constants”). For information on `NULL` value representation on the screen see the “DRIVE Programming Language” manual [2], chapter 8, “Processing databases”).

You can use the `FROM` clause and, if appropriate, the `WHERE` clause to create a join, i.e. link two or more tables with each other and construct the Cartesian product of all the tables involved (see also *join_expression*).

```
SELECT [ { ALL | DISTINCT } ] select_list
      [ INTO { variable },... ]
      FROM table_specification,...
      [ WHERE condition ]
      [ GROUP BY column,... ]
      [ HAVING condition ]
```

With the exception of the `INTO` clause, the clauses of the `SELECT` statement are defined exactly as they are for the `SELECT` expression (see metavariable *select_expression*).

INTO

In a `SELECT` statement in program mode, you must specify the variables that are to be assigned the column values of the derived row in the `INTO` clause.

variable

Name of a variable that is assigned a column value from the derived row or specification of a component or list of components of a structured variable (see metavariable *variable*).

The data type of a variable must be compatible with the data type of the corresponding derived column. If a derived column is an aggregate with several elements, the corresponding variable must be a vector with the same number of elements.

The number of specified variables must be the same as the number of columns in the *select_list* of the SELECT statement. The value of the *n*th column in the *select_list* is assigned to the *n*th variable in the INTO clause. If the value to be assigned is the NULL value, the variable is set to NULL.

If there is no derived row or more than one derived row, none of the variables are set.

If there is no derived row, an SQLSTATE is set which can be handled with WHENEVER &DML_STATE IN ('TABLE END'). If there is more than one derived row, an SQLSTATE is set which can be handled with WHENEVER &DML_STATE IN ('SQL ERROR').

Example

You want to read the name and address of the company with the customer number 100 and store the information in the variables `company`, `zip`, `city` and `street`.

Because the customer number is unique in the `customers` table, you can be sure that the query will return only one row.

```
SELECT company, zip, city, street
INTO &Company,&Zip,&City,&Street
FROM customers
WHERE cust_num=100;
```

SET CATALOG - Set default database name

You use SET CATALOG to define the default database name for unqualified schema names that occur in subsequent EXECUTE statements. The default database name set with OPTION CATALOG continues to be used to qualify unqualified schema names for all static SQL statements. Until the time that the first SET CATALOG (or SET SCHEMA) statement is executed, the database name specified with PARAMETER DYNAMIC CATALOG is used as the default database name for all dynamic statements.

The default database name you set with SET CATALOG is valid until a new database name is set with SET CATALOG or SET SCHEMA or until the end of the application. In interactive mode, the DRIVE default set with PARAMETER DYNAMIC CATALOG is always valid. Refer to chapter “Working with SESAM/SQL V2” on page 25 for information on the interaction between the SET CATALOG statement and the statements OPTION CATALOG and PARAMETER DYNAMIC CATALOG.

The SET CATALOG statement is only permitted in program mode and does not initiate a transaction. It is also permitted within transactions and, in order to illustrate its exclusive effect on dynamic SQL statements, should only be used as a dynamic statement

SET CATALOG *default_catalog*

```
default_catalog ::= { alphanumeric_literal | variable }
```

default_catalog

Name of the database to act as the default for the current program run (up to 18 characters long).

alphanumeric_literal

The database name is specified as an alphanumeric literal.

variable

The database name is specified as an alphanumeric variable of the type CHARACTER or VARCHAR. The variable cannot be a vector and cannot be assigned the NULL value.



The SET CATALOG statement is used in dynamic programs. You can use it to change the database name specific to the DRIVE session. All subsequent SQL statements in the dynamic program can only refer to this database name.

Scope of validity of the default database name:

Up to the end of the application at the latest (transition to interactive mode) or until the next SET CATALOG statement in the same program or in another program, which can also be executed within the transaction.

The statement is only permitted in program mode.

This statement should be specified in dynamic form (EXECUTE) since it only affects dynamic statements. Static statements use the database name specified as the default in the compiler options.

Example

```
EXECUTE 'SET CATALOG ''my_db''';
```

Subsequent dynamic SQL statements use the database name 'my_db' to qualify unqualified schema names.

SET SCHEMA - Set default schema name

You use SET SCHEMA to define the default schema name for the unqualified names of integrity constraints, indexes and tables that occur in subsequent EXECUTE statements. The default schema name set with OPTION SCHEMA continues to be used to qualify the names of integrity constraints, indexes and tables for all static SQL statements. Until the time that the first SET SCHEMA statement is executed, the schema name set with PARAMETER DYNAMIC SCHEMA is used as the default schema name for all dynamic statements.

The default schema name defined with SET SCHEMA is valid until a new schema name is set with SET SCHEMA or until the end of the application. In interactive mode, the DRIVE default set with PARAMETER DYNAMIC SCHEMA is always valid. Refer to chapter "Working with SESAM/SQL V2" on page 25 for information on the interaction between the SET SCHEMA statement and the statements OPTION SCHEMA and PARAMETER DYNAMIC SCHEMA.

If an unqualified schema name is qualified with a database name in the SET SCHEMA statement, the default database name set with PARAMETER DYNAMIC CATALOG is also changed for all subsequent dynamic SQL statements.

The SET SCHEMA statement is only permitted in program mode and does not initiate a transaction. It is also permitted within transactions and, in order to illustrate its exclusive effect on dynamic SQL statements, should only be used as a dynamic statement

SET SCHEMA default_schema

```
default_schema ::= { alphanumeric_literal | variable }
```

default_schema

Name of the default schema for the current DRIVE session (up to 31 characters long). You can qualify the unqualified schema name with a database name.

If you qualify the schema name with a database name, this database name is used as the default database name as if it had been set with SET CATALOG.

alphanumeric_literal

The schema name is specified as an alphanumeric literal.

variable

The schema name is specified as an alphanumeric variable of the type CHARACTER or VARCHAR. The variable cannot be a vector and cannot be assigned the NULL value.



The SET SCHEMA statement is used in dynamic programs. You can use it to change the schema name specific to the DRIVE session. All subsequent SQL statements in the dynamic program can only refer to this schema name.

Scope of validity of the default schema name:

Up to the end of the application at the latest (transition to interactive mode) or until the next SET SCHEMA statement in the same program or in another program, which can also be executed within the transaction.

The statement is only permitted in program mode.

This statement should be specified in dynamic form (EXECUTE) since it only affects dynamic statements. Static statements use the schema name set with the compiler option as the default.

Example

```
EXECUTE 'SET SCHEMA ''my_db.andromeda''';
```

Subsequent dynamic SQL statements use the schema name 'my_andromeda' and the database name 'my_db' for qualifying the names of SQL objects.

SET SESSION AUTHORIZATION - Define authorization identifier

You use SET SESSION AUTHORIZATION to define the current authorization identifier for the dynamic SQL statement in a DRIVE application:

A SET SESSION statement overwrites the following values:

- the default value for SESAM (D0USER)
- the default set with PARAMETER DYNAMIC SESSION AUTHORIZATION, but only until the end of the application (transition to interactive mode)
- the value set with the last SET SESSION AUTHORIZATION statement

You define the current authorization identifier for the dynamic SQL statements of a DRIVE application with either the PARAMETER DYNAMIC AUTHORIZATION or SET SESSION AUTHORIZATION statement. If no authorization identifier is defined in either of these statements, the default authorization identifier set by SESAM, D0USER, is used as the current authorization identifier for the DRIVE application.

The SET SESSION AUTHORIZATION is only permitted in program mode and only within an EXECUTE statement. It does not initiate a transaction and can therefore only be used outside of an SQL transaction.



SET SESSION statements cannot change the authorization identifiers specified with OPTION AUTHORIZATION, i.e. they have no effect on static SQL statements.

```
SET SESSION AUTHORIZATION new_authorization_id
```

```
new_authorization_id := { alphanumeric_literal | variable }
```

new_authorization_id

Name of the authorization identifier that is to be valid for the current program run. The new authorization identifier is valid until the next SET SESSION AUTHORIZATION statement or until the end of the application (transition to interactive mode). The authorization identifier can be up to 18 characters long.

literal

The new authorization identifier is specified as an alphanumeric literal.

variable

The new authorization identifier is specified as an alphanumeric variable. The variable cannot be a vector and cannot be assigned the NULL value.



The SET SESSION AUTHORIZATION statement is used in dynamic programs. You can use it to change the authorization identifier specific to the DRIVE session. All subsequent SQL statements in the dynamic program are only executed for this authorization identifier.

Scope of validity of the current authorization identifier:

Up to the end of the application at the latest (transition to interactive mode) or until the next SET SESSION AUTHORIZATION statement in the same program or in another program, which can only be executed outside of a transaction.

This statement is only permitted in program mode and only within an EXECUTE statement, i.e. it can not be included in the program as a static statement. It must be specified in dynamic form (EXECUTE) since it only affects dynamic statements in DRIVE/WINDOWS. Static statements use the authorization identifier defined in the compiler options as the current authorization identifier.

Example

You want to define a new authorization identifier for dynamic SQL statements for the DRIVE application. The current UTM or BS2000 user must have a system entry with this authorization identifier.

```
EXECUTE 'SET SESSION AUTHORIZATION ''bertha''';
```

All subsequent dynamic SQL statements are executed with the authorization identifier 'bertha'.

SET TRANSACTION - Define transaction attributes

You can use SET TRANSACTION to set the isolation or consistency level and transaction mode for the subsequent SQL transaction.

The isolation or consistency level of a transaction specifies to what degree read operations on rows in the transaction are affected by write accesses in a concurrent transaction.

The transaction mode allows you to specify whether table rows can only be read or can also be updated in the subsequent transaction.



If you define an isolation or consistency level, you also influence the degree of concurrency and thus performance: the fewer phenomena you permit, the lesser the degree of concurrency.

The settings effected by SET TRANSACTION are only valid for the transaction that immediately follows this statement. After the transaction has ended, the default values are again valid.

The SET TRANSACTION statement does not initiate a transaction and can only be used outside an SQL transaction.

```
SET TRANSACTION { level [ [ , ] transaction_mode ] |
                 transaction_mode [ [ , ] level ] }
```

```
level ::= { ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED |
                             REPEATABLE READ | SERIALIZABLE } |
           CONSISTENCY LEVEL consistency_level }
```

```
transaction_mode ::= { READ ONLY | READ WRITE }
```

As in earlier SESAM/SQL versions, you can omit the comma between the two specifications. If, however, you want your application to be portable, you must include the comma.

ISOLATION LEVEL

Sets the isolation level.

If several transactions work with the same tables simultaneously, the following phenomena can occur in which the read accesses in one transaction are affected by the simultaneous write access of another transaction. By specifying an isolation level, you determine which of these phenomena you want to permit in the subsequent SQL transaction.

The following phenomena are of importance:

- dirty read:
A transaction updates a row or inserts a new row. A second transaction reads this row before the first transaction has committed the update. If the first transaction is rolled back, the second transaction has read a row that was never committed.
- non-repeatable read:
A transaction reads a row. Before this transaction is terminated, a second transaction updates or deletes this row and commits the update. If the first transaction then tries to read this row again, either different values will be returned, or an error occurs because the row has been deleted in the meantime. In other words, the result of the second read operation is different to the result of the first.
- phantom:
A transaction reads rows that satisfy a certain search condition. A second transaction subsequently inserts rows that also satisfy this search condition. If the first transaction repeats the query, the derived table includes the new rows.

READ UNCOMMITTED

Isolation level that offers the least protection against concurrent transactions. All the above-mentioned phenomena are possible. In the subsequent SQL transaction, rows can be read that have not yet been committed and these rows can be updated after they have been read.

You cannot specify READ UNCOMMITTED if, at the same time, you specify the transaction mode READ WRITE.

READ COMMITTED

The phenomena “non-repeatable read” and “phantom” can occur. In the subsequent SQL transaction, rows that have been read can be updated by other transaction after they have been read. No rows are read that have not yet been committed

REPEATABLE READ

The phenomenon “phantom” can occur. The phenomena “non-repeatable read” and “dirty read” are not possible.

SERIALIZABLE

Complete protection against concurrent transactions is ensured. The phenomena “dirty read”, “non-repeatable read” and “phantom” cannot occur. The subsequent transaction is unaware of the existence of concurrent transactions.

CONSISTENCY LEVEL

For reasons of upward compatibility with earlier versions, SESAM/SQL provides the parameter **CONSISTENCY LEVEL** as an alternative to isolation level. This means that you define a consistency level which, like the isolation level, determines whether the phenomena “dirty read”, “non-repeatable read” and “phantom” can occur.

consistency_level

Unsigned integer, where $0 \leq \textit{consistency_level} \leq 4$.

Level	Locks set	Rows read
0	Rows read are not locked against updating by other transactions	All rows including those locked against updating by other transactions
1	Rows read are locked against updating by other transactions (until the end of the transaction) unless they are already locked	like 0
2	like 0	Only the rows that other transaction have not locked against updating
3	Rows read are locked against updating by other transactions (until the end of the transaction)	like 2
4	Rows read are locked just as for level 3. The lock against updating by other transactions for non-existent rows ensures that rows cannot be inserted by other transactions.	like 2

The following table indicates the correlation between isolation and consistency level and which phenomena can occur at the different consistency and isolation levels.

Isolation level	Consistency level	Dirty read	Non-repeatable read	Phantom
READ UNCOMMITTED	0	x	x	x
-	1	x	x 1)	x
READ COMMITTED	2	-	x	x
REPEATABLE READ	3	-	-	x
SERIALIZABLE	4	-	-	-

To 1)

The phenomenon “non-repeatable read” can only occur for rows previously read with a dirty read.

READ ONLY

Sets the transaction mode READ ONLY.

Only read database accesses are permitted within the transaction. READ ONLY is the default value for the isolation level READ UNCOMMITTED and the consistency levels 0 and 1.

READ WRITE

Sets the transaction mode READ WRITE.

Only read and write database accesses are possible in the transaction. READ WRITE is the default value for the isolation levels READ COMMITTED, REPEATABLE READ and SERIALIZABLE and for the consistency levels 2, 3 and 4.

You cannot specify READ WRITE if you specify the isolation level READ UNCOMMITTED.

Default values

If an isolation or consistency level entry exists in a user-specific configuration file (see the SESAM/SQL-Server “Core Manual” [20]), this value is used as the default. If this is not the case, the isolation level SERIALIZABLE, the consistency level 4, and the transaction mode READ WRITE are the default values.

STORE - Save cursor position

You use STORE to save the current cursor position.

At the end of a transaction, all open cursors are closed. If you want to be able to access the contents of the derived table in the subsequent transaction, you must save the current cursor position with STORE before the end of the transaction. A cursor saved with STORE can be restored with the RESTORE statement.

FETCH cannot be used after STORE.

A cursor position saved with STORE is canceled by the RESTORE and OPEN statements.

The cursor must be open and positioned on a row. STORE does not close the cursor. The saved cursor position is saved until the end of the DRIVE session at the latest (dialog cursor) or until the end of the program run (program cursor).

STORE is not permitted for a PREFETCH cursor.

STORE *cursor*

cursor

Name of the cursor whose position is to be stored.

The call overwrites any cursor position for the same cursor previously saved with STORE.

UPDATE - Update column values

You use UPDATE to update the column values of rows in a table.

The literals CURRENT USER or USER and SYSTEM USER, as well as the time functions CURRENT DATE, CURRENT TIME and CURRENT TIMESTAMP in the UPDATE statement (and in default values) are evaluated once, and the values calculated are valid for all updates.

If you want to update a row in the specified table, you must own the table or have the UPDATE privilege for each of the columns to be updated. Furthermore, the transaction mode of the current transaction must be READ WRITE.

If integrity constraints have been defined for the table or the columns involved, these are checked after the update operation. If an integrity constraint has been violated, the updates are canceled and an appropriate SQLSTATE set.

```
UPDATE table SET {
    { column | column(pos_no) | column(min-max) }
    = { sql_expression | DEFAULT | NULL }
    },...
[ WHERE { condition | CURRENT OF cursor } ]
```

table

Name of the table containing the rows you want to update. The table can be a base table, an updatable view or an updatable temporary view (see metavariable *table_specification*). Static temporary views cannot be specified for dynamic cursors.

column

Name of an atomic column whose contents you want to update. The column must be part of the table. You cannot qualify the name of the column with a table specification. You can only specify a column once in an UPDATE statement.

column(pos_no)

Element of a multiple column containing the value you want to update.

The multiple column must be part of the table. If you specify several elements in a multiple column, the range of elements specified must be contiguous. Each element can only be specified once.

pos_no is an unsigned integer ≥ 1 .

column(min..max)

Range of column elements in a multiple column that are to be assigned values. The multiple column must be part of the table. If you specify several elements in a multiple column, the range of elements specified must be contiguous. Each element can only be specified once.

min and *max* are unsigned integers ≥ 1 ; *max* must be $\geq min$.

sql_expression

Expression whose value is to be assigned to the preceding column. The value of the expression must be compatible with the data type of the column.

If *sql_expression* is a variable, you can also specify a vector. If you do so, the column must be a multiple column and the number of elements in the vector must be the same as the number of column elements.

If *sql_expression* is an aggregate (see metavariable *value*) that is to be assigned to a multiple column, the number of values must be the same as the number of column elements, and the data type of each component of the aggregate must be compatible with the data type of the target column.

The following restrictions apply to *sql_expression*:

- Neither the underlying base table for *table* nor a view of this base table can be included in the FROM clause of a subquery in *sql_expression*.
- Set functions (AVG, MAX, MIN, SUM, COUNT) are not permitted.

DEFAULT

Only for atomic columns.

The associated column is supplied with the default value if a default value has been defined for the column (see metavariable *column_definition*), otherwise it is assigned the NULL value.

NULL

The preceding column is assigned the NULL value.

WHERE clause

The WHERE clause indicates the rows to be updated.

WHERE *condition* updates a number of rows (multiple UPDATE statement), WHERE CURRENT OF *cursor* updates a single row.

WHERE not specified:

All the rows in the table are updated.

condition

Condition that the rows to be updated must satisfy. A row is only updated if it satisfies the specified condition.

The following restrictions apply to *condition*:

- Column specifications in *condition* outside of subqueries can only reference the specified table.
- Neither the underlying base table for *table* nor a view of this base table can be included in the FROM clause of a subquery included in *condition*.

If no row satisfies *condition*, no row is updated and an SQLSTATE is set which can be handled with WHENEVER &DML_STATE IN ('TABLE END').

CURRENT OF *cursor*

Name of the cursor used to determine the row to be updated. *table* must be the table specified in the first FROM clause of the cursor description.

The cursor must satisfy the following conditions:

- The cursor must reference *table*.
- The cursor must be updatable.
- The cursor must be declared in the same compilation unit, it must be open and positioned on a row in the table with FETCH when the UPDATE statement is executed.

UPDATE updates the row indicated by *cursor*.

UPDATE is not permitted if *cursor* is a PREFETCH cursor.

If *cursor* was declared with the FOR UPDATE clause and column specifications, only the columns specified in that clause can be updated.

The UPDATE statement does not influence the position of the cursor. If you want to update the next row in the derived table, you must position the cursor on this row with FETCH.

Updating the values in a multiple column

In the case of a multiple column, you can update values for individual column elements or for ranges of elements.

An element of a multiple column is identified by its position number in the multiple column.

A range of elements in a multiple column is identified by the position numbers of the first and last element in the range.



The position of an element in a multiple column can change. If an element with a low position number is set to the NULL value, all subsequent elements are shifted to the left and the NULL value added to the end. In this respect, there may be a difference between updating a multiple column with a single UPDATE statement and a list of SET clauses or with a series of UPDATE statements each with a single SET clause.

UPDATE and integrity constraints

By specifying integrity constraints when you define the base table, you can restrict the possible contents of *table*. After the UPDATE statement has been executed, the contents of *table* must satisfy the defined integrity constraints.

UPDATE and updatable permanent views

If CHECK OPTION is specified in the definition of an updatable view, only rows that satisfy the query expression in the view definition can be inserted in the view.

INSERT and transaction management

UPDATE initiates a transaction if no transaction is open. If you define an isolation level for concurrent transactions, you can control how the UPDATE statement affects these transactions (see the SET TRANSACTION statement).

If an error occurs during an update operation, any updates that have already been performed are canceled.

Examples

1. You want to increase the minimum stock level of all items to 20.

```
UPDATE items SET min_stock = 20
           WHERE min_stock < 20;
```

2. You want to update the minimum stock level using a cursor:

```
DECLARE cur_items CURSOR FOR
           SELECT min_stock FROM items
           WHERE min_stock < 20
           FOR UPDATE;
OPEN cur_items;
```

You can update the rows involved with a series of FETCH and UPDATE statements:

```
FETCH cur_items INTO &Min_stock;
UPDATE items SET min_stock = 20
           WHERE CURRENT OF cur_items;
```

3. You want to update the intensity of the individual color components for the color orange in the table color_tab. The column rgb for the color intensity is a multiple column:

```
UPDATE color_tab SET rgb(1-3) = <0.8, 0.4, 0>
           WHERE color_name = 'orange';
```

WHENEVER - Define error handling

You use **WHENEVER**, which is only permitted in program mode and is not executable, to query the entries of the **DRIVE** system variable **&DML_STATE** (= **&ERROR_STATE.DML_STATE**) and define error exits (see also the “Directory of **DRIVE** Statements” [3], **WHENEVER** statement, and the “**DRIVE** Programming Language” manual [2], sections 3.1.2, “System variables”, and 4.2, “Error recovery, end criteria”). After execution of a (static or dynamic) SQL statement and after execution of a **CYCLE cursor INTO ...** statement and the associated **END CYCLE** statement, **&DML_STATE** either contains the entry 'OK' or a specific error status (see below).

In the database system **SESAM/SQL V2**, **WHENEVER** defines the reaction to execution of these statements if they are terminated with an **SQLSTATE** $\langle \rangle$ '00000' (SQL statement executed successfully), $\langle \rangle$ '01SA1' (row locked by foreign transaction that is still open) or '02000' (no row read or updated). **SQLSTATE** 00000 is not an error and cannot be redefined as an error. The other two **SQLSTATE**s are predefined as not being errors, but can be redefined as errors (see below).

You can specify the **WHENEVER** statement more than once in a **DRIVE** program. It must be defined in the declaration section of the program after the definitions of internal subprograms (see the “Directory of **DRIVE** Statements” [3], **SUBPROCEDURE** statement). You use **WHENEVER** to specify which error exits (exception handling) are to be executed in which (or all) error situations (exception conditions). If several **WHENEVER** statements have been specified for an error situation, i.e. **&DML_STATE** was assigned a certain value $\langle \rangle$ 'OK', the last statement entered is used.

```
WHENEVER &DML_STATE [ IN ( status,... ]
                { CONTINUE | CALL subprog_name | BREAK }
```

&DML_STATE

DRIVE system variable of the type **CHAR(16)** that is assigned a value by **DRIVE/WINDOWS** each time an SQL statement is executed.

IN clause

Specification of an exception condition or a list of exception conditions. The exception condition `&DML_STATE IN (status)` is satisfied if `&DML_STATE` is assigned the value *status*. A list of exception conditions is satisfied if `&DML_STATE` is assigned one of the values specified in the list.

```
status ::= { 'TABLE END' | 'DIRTY READ' | 'SQL ERROR' |
             'CURSOR SQL ERROR' | 'TOO MANY CURSORS' |
             'TEMP SYS ERROR' | 'ACC SYS ERROR' |
             'ADMIN SYS ERROR' | 'LIMIT REACHED' }
```

If the IN clause is omitted, this is the same as specifying all possible exception conditions.

```
{ CONTINUE | CALL subprog_name | BREAK }
```

Definition of an error exit (exception handling) for the specified exception condition(s).

CONTINUE

If (one of) the specified exception condition(s) occurs, the program is continued.

CALL *subprog_name*

If (one of) the specified exception condition(s) occurs, the subprogram *subprog_name* is called. If, while *subprog_name* is being processed, another error occurs for which an error exit has been defined with WHENEVER, the program is aborted (no error propagation).

BREAK

If (one of) the specified exception condition(s) occurs, the program is aborted.



The error exit CONTINUED is the default for the exception conditions `&DML_STATE IN ('TABLE END')` and `&DML_STATE IN ('DIRTY READ')`. The error exit BREAK is the default error exit for all other exception conditions.

Mapping SQLCODEs to &DML_STATE entries

SQLCODEs are mapped to &DML_STATEs in a way that is compatible with SESAM/SQL V1 and DRIVE/WINDOWS V1.1 (see chapter 6, “Error messages”, in the present manual, and section 3.1.2, “System variables”, in the “DRIVE Programming Language” manual [2]).

Mapping SQLSTATEs to &DML_STATE entries

The SQLSTATE 00000 is mapped to &DML_STATE = 'OK'.

The SQLSTATE 02000 is mapped to &DML_STATE = 'TABLE END'.

The SQLSTATE 01SA1 is mapped to &DML_STATE = 'DIRTY READ'.

The SQLSTATE 24SA5 is mapped to &DML_STATE = 'CURSOR SQL ERROR'.

The SQLSTATEs 25SA3 and 25SA5 are mapped to &DML_STATE = 'SQL ERROR'.

The SQLSTATEs 91SA3 and 91SA5 are mapped to &DML_STATE = 'LIMIT REACHED'.

All other SQLSTATEs are mapped to &DML_STATE = 'SQL ERROR', provided that &DML_STATE is not assigned a different value due to an SQLCODE reported at the same time and its mapping (see above).



The SQLSTATE of SESAM/SQL V2 is stored in the DRIVE system variable &SQL_STATE (= &ERROR_STATE.SQL_STATE). &SQL_STATE is a structured variable that divides the SQLSTATE into the error class &SQL_CLASS (= &SQL_STATE.SQL_CLASS) and the subclass &SUB_CLASS (= &SQL_STATE.SUB_CLASS), both of which can be referenced individually.

Therefore, each SQLSTATE can be subject to specific error handling in a WHENEVER subprogram *subprog_name* for the exception condition &DML_STATE IN ('SQL ERROR'). In particular, the current SQL transaction can be rolled back (ROLLBACK), the DRIVE transaction rolled back (ROLLBACK WITH RESET), the program aborted (BREAK) or continued (END SUBPROC) for certain SQLSTATEs.

Meaning of the individual &DML_STATE entries

'OK'

SQL statement executed successfully.



The condition &DML_STATE IN ('OK') or = 'OK' is not an exception condition. However, if necessary, it can be used in IF or CASE statements after SQL statements or in CYCLE statements. For productive use, however, it is recommended that you use WHENEVER statements and subprograms for error handling.

'TABLE END'

No row read or modified.

Meaning

No row was read by a FETCH or SELECT statement,
no row was deleted by a DELETE statement,
no row was updated by an UPDATE statement,
no row was inserted by an INSERT statement.

Response

The DRIVE program must be modified if necessary.



If an error exit <> CONTINUE was specified for 'TABLE END' in a WHENEVER statement, this is not executed if the error situation 'TABLE END' occurs in a CYCLE *cursor* INTO ... statement.

'DIRTY READ'

Row locked by a foreign transaction that is still open.

Meaning

The row that has been output contains data that may have been updated by a foreign transaction that is still open. A “dirty read” phenomenon may have occurred (see the “DRIVE Programming Language” manual [2], chapter 10, “Transaction concept”). This situation is only possible for consistency level 0 or 1, i.e. isolation level READ UNCOMMITTED.

Response

The SQL statement must be repeated if necessary and/or a different consistency level defined, if appropriate (see SET TRANSACTION statement or ISOLATION LEVEL pragma clause).

'SQL ERROR'

SQL not execute successfully or executed without pragma influence.

Meaning

The exact meaning is the same as that recommended by SESAM for the corresponding SQLSTATE xxSxx (enter the SYSTEM 'HELP SEWxxxx' statement).

Response

The recommended response is the same as that recommended by SESAM for the corresponding SQLSTATE xxSxx (enter the SYSTEM 'HELP SEWxxxx' statement).

'CURSOR SQL ERROR'

Cursor is not saved.

Meaning

The cursor was not saved with a STORE statement or has been canceled in the mean time because the transaction has been rolled back.

Response

Modify DRIVE program.

'TOO MANY CURSORS'

Permitted DRIVE system limit for the number of dynamic cursor declarations has been exceeded.

Meaning

20 dynamic or variable program cursors or dialog cursors have already been declared.

Response

Delete several cursors (see DROP statement) or abort DRIVE program (see BREAK statement).

'TEMP SYS ERROR'

Temporary obstacle for SESAM-RTS or SESAM-DBH

Meaning

see 'SQL ERROR'

Response

see 'SQL ERROR'

'ACC SYS ERROR'

Incompatibilities within an SQL schema.

Meaning

see 'SQL ERROR'

Response

see 'SQL ERROR'

'ADMIN SYS ERROR'

SESAM administrator intervention required.

Meaning

see 'SQL ERROR'

Response

see 'SQL ERROR'

'LIMIT REACHED'

SESAM system limit reached.

Meaning

see 'SQL ERROR'

Response

see 'SQL ERROR'



In the event of the error situation 'LIMIT REACHED', an SQLSTATE does not necessarily exist.

4 DRIVE SQL metavariables

SQL is fully integrated in the DRIVE language. This means that a number of DRIVE metavariables can be used in SQL statements (e.g. conditions and variables, see *condition* and *variable*). The SQL dialect supported by the database system involved always includes restrictions and extensions with regard to the SQL standard, which can result in differences compared to the DRIVE metavariables. Only if you want to reference SQL objects (e.g. tables and columns, see *table_specification* and *sql_expression*) will you need metavariables that can only be used in SQL statements but not in DRIVE statements. This chapter provides you with a description of SQL metavariables and DRIVE SQL metavariables. The description of DRIVE SQL metavariables includes the section from the complete description in the “Directory of DRIVE Statements” [3] that is relevant to the support of SESAM/SQL V2.

The following metavariables can only be used in SQL statements (SQL metavariables):

- *table*
- *column_definition*
- *basic_data_type* (also in DRIVE statements)
- *default*
- *column_constraint*
- *table_constraint*
- *query_expression*
 - *table_specification*
 - SELECT expressions (see *select_expression*)
 - projections (see *select_list*)
 - FROM clause
 - WHERE clause
 - GROUP BY clause
 - HAVING clause
 - join expressions (see *join_expression*)
 - unions
 - subqueries (see *subquery*).

The following metavariables can be used in SQL statements and, if necessary, in DRIVE statements (DRIVE SQL metavariables):

- *sql_expression*
 - values (see *value*)
 - literals (see *literal*)
 - alphanumeric literals
 - numeric literals
 - time literals
 - variables (see *variable*)
 - simple variables
 - structured variables
 - indexed variables
 - aggregates
 - three time functions (see *time_function*)
 - column references “[*table.*] *column* [({*pos_no* | *min-max*})]”⁴
 - six set functions (see *set_function*)
 - subqueries (see *subquery*)
 - two user functions (SQL user and system entry)
 - operators (unary +, -, binary +, -, *, /, ||)
- *condition*
 - seven groups of predicates (see *predicate*)
 - conjunctions, disjunctions, negations

These metavariables represent a consistent extension of the DRIVE metavariables *expression* and *condition* in accordance with the “Directory of DRIVE Statements” [3]. Only if column references, set functions, subqueries or user functions occur in *sql_expression* and *condition* are you dealing with a DRIVE SQL metavariable that is not a DRIVE metavariable, and which therefore can only be used in SQL statements.

In the case of *query_expression* and *sql_expression*, DRIVE/WINDOWS V2.1 supports the full functionality of SESAM/SQL V2.0 but not the extensions provided in Version 2.1.

query_expression

In SESAM/SQL, query expressions are the most important means of querying data.

You use query expressions to select rows and columns from base tables, views and temporary views. The rows found constitute the derived table.

A query expression is part of an SQL statement. A query expression can occur in subqueries or in any of the following SQL statements:

CREATE TEMPORARY VIEW	Define a temporary view
CREATE VIEW	Define a view
DECLARE	Declare a cursor
INSERT	Insert rows in a table

If you want to use a query expression in an SQL statement, you must own the table referenced with the query expression or have SELECT privilege for the table involved.

```
query_expression ::= { select_expression | expression | ( query_expression ) }
                  [ UNION [ ALL ] query_expression ]
```

select_expression

SELECT expression, see metavariable *select_expression*.

join_expression

Join expression, see metavariable *join_expression*.

(query_expression)

subquery, see metavariable *subquery*.

UNION

The UNION clause combines two query expressions. The derived table contains all the rows that occur in the first or second derived table. You can combine more than two derived tables if you use the UNION clause several times.

If you want to combine query expressions using UNION, the following conditions must be satisfied:

- The derived tables of both UNION operands must have the same number of columns and the data types of the corresponding columns must be compatible. The data types of the derived columns are determined according to the rules explained further below.

If the corresponding columns in both source tables have the same names, the derived column is given this name. Otherwise, the name of the derived column is undefined.

- Only atomic columns may be selected.

Query expressions combined with the UNION clause cannot be updated.

ALL

Duplicate rows in the derived table are retained.

ALL omitted:

Duplicate rows are removed (duplicate elimination).

Updatability of a query expression

A query expression is updatable if the following conditions are fulfilled:

- The query expression does not contain a join expression.
- The query expression does not contain a UNION clause.
- Only column names can be specified in *select_list*. Other elements of *sql_expression*, e.g. subqueries, set functions, time functions or literals, are not permitted. Atomic columns cannot be specified more than once. Subranges of multiple columns cannot overlap.
- Only a table or updatable subquery can be specified in the FROM clause. If a table is specified, it must be a base table or an updatable view.
- No subquery can occur in the WHERE clause.
- The keyword DISTINCT cannot be specified.
- The SELECT expression cannot include a GROUP BY or HAVING clause.

Data type of the derived column for UNION

If two query expressions are combined with UNION, the data type of the derived column is determined by applying the following rules:

- Both source columns are of the type CHAR:
The derived column is of the type CHAR with the longer of the two lengths.
- One source column is of the type VARCHAR and the other source column is of the type CHAR or VARCHAR:
The derived column is of the type VARCHAR with the greater length or greater maximum length.

- Both source columns are an integer or fixed-point type (INT, SMALLINT, NUM, DEC):
The data type is an integer or fixed-point type.
 - The number of digits to the right of the decimal point is the greater of the two values of the source columns.
 - The total number of significant digits is the greater of the two values plus the greater of the two values for the number of digits after the decimal point of the source column. The maximum number of digits is, however, 31.
- One source column is of a floating-point type (REAL, DOUBLE, FLOAT), the other is of any numeric data type:
The derived column is of the type DOUBLE PRECISION.
- Both source columns have a date and time data type:
Both column must have the same date and time data type and the derived column also has this data type.

condition

Conditions are used to restrict the number of rows affected by a table operation. Only the rows that satisfy the specified *condition* are taken into account. You may specify *condition* for DELETE, UPDATE and SELECT and when joining tables (join expression). You can specify *condition* in table and column constraints in order to formulate CHECK clauses as integrity constraints (see metavariables *column_constraint* and *table_constraint*).

You define a *condition* in a WHERE, HAVING, ON or CHECK clause, which may be used in the following statements or query expressions:

- WHERE clause
 - DELETE statement
 - SELECT statement
 - SELECT expression for CREATE VIEW, CREATE TEMPORARY VIEW, DECLARE, INSERT
 - UPDATE statement
- HAVING clause
 - SELECT statement
 - SELECT expression for CREATE VIEW, CREATE TEMPORARY VIEW, DECLARE, INSERT
- ON clause in join expression
- CHECK clause in the CREATE TABLE or ALTER TABLE statement

condition consists of predicates and can include logical operators. The predicates are the operands of the logical operators.

condition is evaluated by applying the operators to the results of the operands. The result is one of the truth values **true**, **false** or **unknown**.

```
condition ::= { [ condition AND ] { [ NOT ] { predicate | (condition) } } |
              condition OR { [ NOT ] { { predicate | (condition) } } }
```

predicate

Specification of a predicate (see page 185).

(*condition*)

Nested condition for creating a condition with more than one logical operator or predicate with precedence specifications.

AND

Logical AND (conjunction).

Result

Op1 AND Op2		Op2		
		true	false	unknown
Op2	true	true	false	unknown
	false	false	false	false
	unknown	unknown	false	unknown

Table 1: Logical operator AND

OR

Logical OR (disjunction).

Result

Op1 OR Op2		Op1		
		true	false	unknown
Op2	true	true	true	true
	false	true	false	unknown
	unknown	true	unknown	unknown

Table 2: Logical operator OR

NOT

Logical negation.

Result

		NOT Op
Op	true	false
	false	true
	unknown	unknown

Table 3: Logical operator NOT

Precedence

- Expressions enclosed in parentheses have highest precedence.
- NOT takes precedence over AND and OR.
- AND takes precedence over OR.
- Operators with the same precedence level are applied from left to right.



If *condition* assumes the truth value **unknown** or **false** in an IF statement (see the “Directory of DRIVE Statements” [3]), you branch to the ELSE path.

If *condition* assumes the truth value **unknown** or **false** in a CYCLE statement, execution of the statement is terminated.

join_expression

A join links the data from several tables. A table can also be joined to itself.

A join is created by linking two or more tables with each other. There are two ways of creating a join:

- with a join expression (especially as part of a FROM clause) which is used to
 1. create the Cartesian product of all the tables involved
 2. select rows from the Cartesian product of the joined tables by specifying join conditions.

All the rows in the Cartesian product that satisfy the join conditions are included in the derived table.

- without a join expression: in a *select_expression* or in a SELECT statement using the FROM clause (contains the tables involved) and, if necessary, the WHERE clause (contains join conditions).

Because of the evaluation rules for *select_expression* (see page 202), the result of this possibility is the same as using a join expression.

A join expression consists of the tables to be joined, the desired join operation and a join condition.

A join expression can be specified

- as a query expression in an SQL statement
- In the FROM clause of a *select_expression* or SELECT statement
- in a subquery in *select_list* and HAVING clause

The derived table of a join expression cannot be updated.

The result of a join expression consists of one, two or three partial results.

First of all, all the rows that do not satisfy the join condition in the ON clause are removed from the Cartesian product of both tables.

In the case of an INNER join, the remaining rows constitute the result. For other join types, they constitute the first partial result.

In the case of a LEFT OUTER join, all the rows in the first table specification for which there is no matching row in the second table specification are multiplied with a non-existent NULL row from the second table, i.e. a NULL value is added to the first table for each row for which there is no match. The first, i.e. left, table is referred to as the dominant table.

In the case of a RIGHT OUTER join, the same procedure is followed but the roles of the first and second tables are switched. The second, i.e. right, table is the dominant table. Rows determined in this way constitute the second partial result.

In the case of a FULL OUTER join, both procedures are performed and return the second and third partial results.

All the partial results are then combined to create the result of the outer join.

```
join_expression ::=
    { table_specification [ { INNER | { LEFT | RIGHT | FULL } [ OUTER ] } ]
      JOIN table_specification ON condition |
      ( join_expression ) }
```

table_specification

Specification of a table from which data is to be read.

INNER

INNER is the default value.

INNER operator for creating an inner join. In an inner join, the derived table only contains the rows of the Cartesian product that satisfy the join condition (see above).

{ LEFT | RIGHT | FULL | [OUTER] }

Operators for creating an outer join. A table that is part of an outer join cannot include multiple columns.

In an outer join, the type of outer join defines the dominant table(s) (see above).

If a row in the dominant table does not satisfy the join condition, the row is nevertheless included in the derived table. The derived column that references the other table is set to NULL values.

LEFT

The table to the left of the LEFT operator is the dominant table.

RIGHT

The table to the right of the RIGHT operator is the dominant table.

FULL

The table to the left and the right of the FULL operator are both dominant tables. FULL joins the tables created with LEFT and RIGHT.

condition

Condition to be used as the join condition for joining the specified tables (see metavariable *condition*).

The following applies to any column specified in *condition*:

The column must either be part of one of the tables to be joined or, in the case of subqueries, part of one of the tables from a higher-level *select_expression*.

If a set function occurs in *condition*, one of the following conditions must be satisfied:

- The set function is part of a subquery.
- The join expression is in a *select_list* or HAVING clause, and the column specified in the argument of the set function is an external reference (see metavariable *set_function*).

(join_expression)

Nested join expression for creating a join from more than two tables with precedence specification.

literal

With the exception of the NULL value, literals exist for each group of values. The following diagram provides you with an overview:

```
literal ::=  
  { char_literal | num_literal | date_time_literal }
```



It is not possible to specify a hexadecimal literal (X'string'[(n)]) in SESAM V2 as it is in SESAM V1. Instead, you must assign the desired hexadecimal value to an appropriate DRIVE variable and then use this variable instead of the literal.

char_literal - Alphanumeric literal

The syntax for an alphanumeric literal is defined as follows:

```
char_literal ::= 'string'  
string ::= [ character ] ...
```

string

Any string. *string* must be enclosed in single quotes ('). If a single quote is used in *string*, it must be specified twice. The pair of single quote characters is considered a single character. *string* can be empty or can contain up to 256 characters. Strings with the length 0 are permitted as literals although it is not possible to define a data type CHARACTER(0).

character

Any character (EBCDI code in BS2000, ASCII code in SINIX and in MS-Windows).



The specification of a repetition factor permitted in DRIVE statements is not possible in SQL statements.

num_literal - Numeric literals

The syntax for numeric literals is defined as follows:

```

num_literal ::= { integer | fixed_point_number | floating_point_number | $PI }
integer ::= [ { + | - } ] unsigned_integer
fixed_point_number ::= [ { + | - } ] unsigned_integer [ .unsigned_integer ]
floating_point_number ::= fixed_point_number E [ { + | - } ] unsigned_integer
unsigned_integer ::= digit...

```

Integers and fixed-point literals can have up to 31 digits.

The data type of the literal is integer, fixed-point number or floating-point number with the specified number of digits to the right and left of the decimal point.

\$PI

Abbreviation of the number 3,141592653 ... (circumference of a circle divided by its diameter).

digit

Decimal digit 0 to 9.



Specification of a dot after *unsigned_integer* is not permitted in DRIVE/WINDOWS for *integer*.

For *fixed_point_number*, specification of a dot in DRIVE/WINDOWS is only permitted if *unsigned_integer* is specified twice. You cannot include any blanks in *floating_point_number*.

date_time_literal - Time literals

The syntax for time literals is defined as follows:

```
date_time_literal ::=
    { DATE (year-month-day) |
      TIME (hour:minute:second [.fraction_of_second ] ) |
      TIMESTAMP (year-month-day hour:minute:second [.fraction_of_second ] ) }
```

DATE

Date. The data type is DATE.

TIME

Time. The data type is TIME or TIME(3). The data type TIME, i.e. leaving out the fractions of a second, is not permitted in SQL statements.

TIMESTAMP

Time stamp. The data type is TIMESTAMP(3).

year

Four-digit unsigned integer between 0001 and 9999 indicating the year.

month

Two-digit unsigned integer between 01 and 12 indicating the month.

day

Two-digit unsigned integer between 01 and 31 (corresponding to the month and year) indicating the day.

hour

Two-digit unsigned integer between 00 and 23 indicating the hour.

minute

Two-digit unsigned integer between 00 and 59 indicating the minute.

second

Unsigned fixed-point number between 00 and 59 that indicates the seconds. The specification of up to two leap seconds (values 60 and 61) is not permitted in DRIVE/WINDOWS.

fraction_of_second

Three-digit unsigned integer between 000 and 999 that indicates the fractions of a second.

The specification of dates "B.C." is therefore not possible.

A date specification must observe the rules of the Gregorian calendar even if the date involved is before introduction of the Gregorian calendar. In particular, the days between DATE(1582-10-05) and DATE(1582-10-14) do not therefore exist.



The separators between the component values must be specified exactly as stated below:

hyphen "-" between year, month and day

blank " " between day and hour

colon ":" between hour, minutes and seconds

period "." between seconds and fractions of a second.

set_function

Set functions return the average, count, maximum value, minimum value or sum of a set of values or the number of rows in a derived table.

Set functions are only permitted in SQL statements.

```
set_function ::= { { AVG | COUNT | MAX | MIN | SUM }
                  ( [ { ALL | DISTINCT } ] sql_expression ) |
                  COUNT(*) }
```

ALL

ALL is the default value.

All values are taken into account, including duplicate values

DISTINCT

Only unique values are taken into account. Duplicate values are ignored.

For the functions MAX() and MIN(), DISTINCT has no effect on the result.

sql_expression

Expression determining the values in the set (see metavariable *sql_expression*).

The *sql_expression* for each set function except for COUNT(*) can have a certain data type. The permitted data type(s) for each function is specified in the function description.

The following restrictions apply to *sql_expression*:

- *sql_expression* cannot include any multiple columns.
- *sql_expression* cannot include any set functions.
- *sql_expression* cannot include any subqueries.
- If a column name in *sql_expression* specifies a column of a higher-level query expression (**external reference**), *sql_expression* may only include this column name.

In this case, the set function must satisfy one of the following conditions:

- The set function is included in a *select_list*.
- The set function is included in a subquery of a HAVING clause. The column name must indicate a column of the *select_expression* that contains a HAVING clause.

AVG() - Calculate arithmetic average

AVG() calculates the average of a set of numeric values. NULL values are ignored.

```
AVG( [ { ALL | DISTINCT } ] sql_expression )
```

ALL

ALL is the default value.

All values are taken into account, including duplicate values.

DISTINCT

Only unique values are included in the calculation. Duplicate values are ignored.

sql_expression

Numeric expression.

Result

Without GROUP BY clause:

Returns the arithmetic average of all the values in the specified *sql_expression*.

With GROUP BY clause:

Returns the arithmetic average per group of all the values for this group.

If the set of values returned by *sql_expression* is empty, the result or the result for this group is the NULL value.

Data type: like *sql_expression* with the following number of digits:

- Integer or fixed-point number:

The total number of significant digits is 31, the number of digits to the right of the decimal point is $31-t+r$.

t and r are the total number of significant digits and the number of digits after the decimal point, respectively, in *sql_expression*.

- Floating-point number:

The total number of significant digits is 21 binary digits for REAL numbers and 53 binary digits for DOUBLE PRECISION.

Examples

1. SELECT without GROUP BY:

Calculate the average price of the services in the table `service`:

```
SELECT AVG(service_price) FROM service;
```

```
783.33
```

If you enter a row in the table that contains the NULL value in the column `service_price`, the result does not change.

2. SELECT with GROUP BY:

The average price is calculated for each order number:

```
DECLARE ... CURSOR FOR SELECT order_num, AVG(service_price)
FROM service
GROUP BY order_num;
```

```
order_num
200        1026
211        662.5
250        662.5
```

COUNT(*) - Count table rows

COUNT(*) counts the rows in a table. Rows containing NULL values are included in the count. COUNT(*) is only permitted in the *select_list* of a *select_expression* (see metavariable *select_expression*).

COUNT(*)

Result

Without GROUP BY clause:

Returns the number of rows in the derived table of the corresponding *select_expression* (or corresponding SELECT statement). Duplicate rows and rows containing only NULL values are included.

With GROUP BY clause:

Returns the number of rows per group for each group in the derived table.

Data type: numeric (integer) with 31 digits.

Examples

1. SELECT without GROUP BY:

Query the number of customers living in Munich in the table customers:

```
SELECT COUNT(*) FROM customers WHERE city='Munich';
```

```
3
```

2. SELECT with GROUP BY:

Count the customers for each city:

```
DECLARE ... CURSOR FOR SELECT city, COUNT(*)
FROM customers
GROUP BY city;
```

```
city
Berlin          1
Berne 33        1
Hanover         1
Moenchengladbach 1
Munich          3
New York, NY    1
```


COUNT() - Count elements

COUNT() counts the elements in a set of values. NULL values are not included in the count.

```
COUNT( [ { ALL | DISTINCT } ] sql_expression )
```

ALL

ALL is the default value.

All values are taken into account, including duplicate values.

DISTINCT

Only unique values are taken into account. Duplicate values are ignored.

sql_expression

Numeric expression, alphanumeric expression or time value expression.

Result

Without GROUP BY clause:

Number of values in the set returned by *sql_expression*.

With GROUP BY clause:

Returns the number of values in each group.

Data type: numeric (integer) with 31 digits.

Examples

1. SELECT without GROUP BY:

Determine the number of different service descriptions in the table `service`:

```
SELECT COUNT(DISTINCT service_text) FROM service;
```

```
7
```

2. SELECT with GROUP BY:

Count the number of different services for each order number:

```
DECLARE ... CURSOR FOR SELECT order_num, COUNT(DISTINCT service_text)
FROM service
GROUP BY order_num;
```

order_num	
200	2
211	4
260	2

MAX() - Determine largest value

MAX() determines the largest value in a set of values. NULL values are ignored.

The comparison of values (with comparable data types) is described under the metavariablen *condition*.

```
MAX( [ { ALL | DISTINCT } ] sql_expression )
```

ALL is the default value.

DISTINCT

DISTINCT can be specified but has no effect on the result.

sql_expression

Numeric expression, alphanumeric expression or time value expression.

Result

Without GROUP BY clause:

Determines the largest value in the set of values returned by *sql_expression*.

With GROUP BY clause:

Returns the largest value of each group.

If the set of values returned by *sql_expression* is empty, the result or the result for this group is the NULL value.

Data type: like *sql_expression*

Examples

1. SELECT without GROUP BY:

Query the highest service price for order 211 in the table *service*:

```
SELECT MAX(service_price) FROM service WHERE order_num=211
```

```
1200
```

2. DECLARE ... CURSOR FOR SELECT with GROUP BY:
Determine the highest service price for each order number:

```
DECLARE... CURSOR FOR SELECT order_num, MAX(service_price)
FROM service
GROUP BY order_num;
```

order_num	
200	1600
211	1200
250	1200

MIN() - Determine lowest value

MIN() determines the smallest element in a set of values. NULL values are ignored.

The comparison of values (with comparable data types) is described under the metavariablen *condition*.

```
MIN( [ { ALL | DISTINCT } ] sql_expression )
```

ALL is the default value.

DISTINCT

DISTINCT can be specified but has no effect on the result.

sql_expression

Numeric expression, alphanumeric expression or time value expression.

Result

Without GROUP BY clause:

Determines the lowest value in the set of values returned by *sql_expression*.

With GROUP BY clause:

Returns the lowest value of each group.

If the set of values returned by *sql_expression* is empty, the result or result for this group is the NULL value.

Data type: like *sql_expression*.

Examples

1. SELECT without GROUP BY:

Query the lowest service price for order 211 in the table `service`:

```
SELECT MIN(service_price) FROM service WHERE order_num=211;
```

```
50
```

2. SELECT with GROUP BY:

Determine the lowest service price for each order number:

```
DECLARE ... CURSOR FOR SELECT order_num, MIN(service_price)
FROM service
GROUP BY order_num;
```

order_num	
200	75
211	50
250	125

SUM() - Calculate sum

SUM() calculates the sum of all the values in a set. NULL values are ignored.

```
SUM( [ { ALL | DISTINCT } ] sql_expression )
```

ALL

ALL is the default value.

All values are taken into account, including duplicate values.

DISTINCT

Only unique values are taken into account. Duplicate values are ignored.

sql_expression

Numeric expression.

Result

Without GROUP BY clause:

Calculates the sum of the values returned by *sql_expression*.

With GROUP BY clause:

Returns the sum of the values in the derived column of each group.

If the set of values returned by *sql_expression* is empty, the result or the result for this group is the NULL value.

Data type: like *sql_expression* with the following number of digits:

- Integer or fixed-point number:
The total number of significant digits is 31, the number of digits to the right of the decimal point remains the same.
- Floating-point number:
The total number of significant digits corresponds to 21 binary digits for REAL numbers and 53 for DOUBLE PRECISION.

If the sum of the values is too large for this data type, a SESAM error message is issued.

Example

Calculate the sum of the parts for each item number in the table purpose:

```
DECLARE ... CURSOR FOR SELECT item_num, SUM(number)
FROM purpose
GROUP BY item_num;
```

item_num	
1	4
120	27
200	20

predicate - Specify predicate

predicate consists of operands and operators. *predicate* can be grouped together as follows according to the operator involved:

- comparison of two values
- comparison with a derived column (only possible in SQL statements)
- range query
- element query
- pattern comparison (only possible in SQL statements)
- NULL value comparison
- existence query (only possible in SQL statements)

predicate returns the truth value **true**, **false** or **unknown**. The value of *predicate* is calculated by calculating the values of the operands and applying the appropriate operators to the calculated values.

The diagram below describes the syntax of all the groups of *predicate*.

```

predicate ::=
{ sql_expression { = | < | > | <= | >= | <> } sql_expression |
  sql_expression { = | < | > | <= | >= | <> } { ANY | SOME | ALL }
  subquery |
  sql_expression [NOT] BETWEEN sql_expression AND sql_expression |
  sql_expression [NOT] IN { subquery | (sql_expression,sql_expression,...) } |
  [ table . ] { column | column(pos_no) | column(min-max) } [ NOT ] LIKE
  patter [ ESCAPE character ] |
  [ table . ] { column | column(pos_no) | column(min-max) }
  IS [ NOT ] NULL |
  EXISTS subquery }

```

The predicates are described in the order in which they are listed in the overview:

- comparison of two values
- comparison with a derived column
- range query
- element query
- pattern comparison
- NULL value comparison
- existence query

Comparing two values

Two operands with comparable data types are compared as indicated by the specified comparison operator.

`sql_expression { = | < | > | <= | >= | <> } sql_expression2`

sql_expression1, *sql_expression2*

Operands for comparison.

The value for *sql_expression1* and *sql_expression2* must either be an atomic value or the name of a multiple column. If the operand is a multiple column, the column specification cannot be an external reference (see metavariable *set_function*).

If *sql_expression1* is an atomic value, *sql_expression1* and *sql_expression2* must be of the same *basic_data_type*.

If *sql_expression1* is structured, only the comparison operators equal to (=) and not equal to (<>) are valid. Exception: multiple columns.

Comparison operator

- = Compare whether two values are the same
- < Compare whether one value is smaller than the other
- > Compare whether one value is greater than the other
- <= Compare whether one value is smaller than or equal to the other
- >= Compare whether one value is greater than or equal to the other
- <> Compare whether two values are not equal

Result

operand1 is an atomic value:

Unknown if at least one operand is the NULL value.

True if both operands are non-NULL values and the comparison holds true.

False in all other cases.

operand1 is a multiple column:

Each occurrence of *sql_expression1* is compared with *sql_expression2*. The comparison results are ORed.

Example

If X is a multiple column with 3 elements, the comparison

```
X(1-3) >= 13
```

is equivalent to the following comparisons:

```
X(1) >= 13 OR
```

```
X(2) >= 13 OR
```

```
X(3) >= 13
```

Comparison rules

The way in which a comparison operation is performed depends on the data type of the operands. The following is an overview of the rules governing comparison.

NULL values

If an operand is the NULL value, all comparisons return the truth value **unknown**.

Alphanumeric values

Two alphanumeric values are compared from left to right character by character. If the two values have different lengths, the shorter string is padded on the right with blanks so that both values have the same length.

Two strings are identical if each has the same character at the same position.

If two strings are not identical, the EBCDIC code of the first two differing characters determines which string is greater or smaller. This is valid without restriction in BS2000. On the MS-Windows and SINIX platforms, this holds true within SQL statements because the comparison in BS2000 is made by SESAM. Within DRIVE statements, on the other hand, the comparison is performed by DRIVE/WINDOWS on the client platform, meaning that the ASCII code determines the comparison results.

Numeric values

Two numeric values are the same if they are both 0, or if they have the same sign and the same value.

Time values

Dates, times and time stamps can be compared. The data type of both operands must be the same.

- One date is greater than another if it is a later date.
- One time is greater than another if it is a later point in time.
- One time stamp is greater than another if either the date is later or, if the date is the same, the time is later.

Examples

1. Comparing alphanumeric values:

Select the customers from the table `customers` that come from Munich, and include the customer information:

```
DECLARE ... CURSOR FOR SELECT company, cust_info, city
FROM customers
WHERE city = 'Munich';
...

```

company	cust_info	city
Siemens AG	Electrical	Munich
Login GmbH	PC networks	Munich
Plenzer Trading	Fruit market	Munich

2. Comparison with subquery that returns an atomic value:

Select the items that need the greatest number of part 501 from the table `purpose`:

```
SELECT item_num
FROM purpose
WHERE part = 501
AND
number = (SELECT MAX(number)
FROM purpose
WHERE part = 501);

```

Because the maximum is always a unique value, the subquery returns an atomic value and can be specified in the comparison as an operand.

item_num
200

Comparison with derived column

sql_expression is compared with the values of a derived column that is the result of a subquery.

This predicate is only possible within SQL statements.

```
sql_expression { = | < | > | <= | >= | <> } { ANY | SOME | ALL } subquery
```

sql_expression

Operand for the comparison.

The value of *sql_expression* must be an atomic value.

Comparison operator

- = Compare whether two values are the same
- < Compare whether one value is smaller than the other
- > Compare whether one value is greater than the other
- <= Compare whether one value is smaller than or equal to the other
- >= Compare whether one value is greater than or equal to the other
- <> Compare whether two values are not equal

subquery

Subquery that returns a single-column table.

sql_expression and the values returned by the subquery must have compatible data types.

Result

ANY

SOME

True if the comparison with at least one value in the derived column is **true**.

False if the derived column is empty, or if the comparison with all the values in the derived column is **false**.

Unknown in all other cases.

ALL

True if the derived column is empty, or if the comparison with all the values in the derived column is **true**.

False if the comparison with at least one value in the derived column is **false**.

Unknown in all other cases.

Example

From the table `purpose`, select the items that have a part whose total number is greater than the total number of all the parts of the item with the item number 1.

```
DECLARE ... CURSOR FOR SELECT item_num
      FROM purpose
      WHERE number
      > ALL (SELECT number
            FROM purpose
            WHERE item_num = 1);
```

```
item_num
  120
  200
```

Range queries

In a range query, the value of the first *sql_expression* is checked to see if it lies in the range of values specified by the second and third *sql_expression*.

□ □

```
sql_expression1 [ NOT ] BETWEEN sql_expression2 AND sql_expression3
```

sql_expression1, *sql_expression2*, *sql_expression3*

Operands for the comparison.

The value of the first *sql_expression* must be an atomic value or the name of a multiple column. If *sql_expression* is a multiple column, the column specification cannot be an external reference (see metavariable *set_function*).

The values of the second and third *sql_expression* must be atomic values.

The operands must have compatible data types.

Result

sql_expression1 is an atomic value:

Without NOT:

identical to:

$(sql_expression1 \geq sql_expression2) \text{ AND } (sql_expression1 \leq sql_expression3)$

With NOT:

identical to:

$\text{NOT } (sql_expression1 \text{ BETWEEN } sql_expression2 \text{ AND } sql_expression3)$

sql_expression1 is a multiple column:

- The range query is performed for each occurrence of *sql_expression1*.
- The individual results are ORed.

Example

If X is a multiple column with 3 elements, the range query

```
X(1-3) BETWEEN 13 AND 20
```

is equivalent to the following range queries:

```
X(1) BETWEEN 13 AND 20 OR
```

```
X(2) BETWEEN 13 AND 20 OR
```

```
X(3) BETWEEN 13 AND 20
```


Examples

1. Numeric range:

Select all the items from the table `items` whose price is between 50 and 100 dollars. Include the item name in the output:

```
SELECT item_num, item_name, price
FROM items
WHERE price BETWEEN 50.00 AND 100.00;
```

item_num	item_name	price
200	handlebars	60.00

2. Range of dates:

Select all the orders placed in December 1990 from the `orders` table. Include the order number, customer number, order date and order text in the output:

```
DECLARE ... CURSOR FOR SELECT order_num, cust_num, order_text,
order_date
FROM orders
WHERE order_date
BETWEEN DATE(1990-12-01) AND DATE(1990-12-31);
...
```

order_num	cust_num	order_text	order_date
210	106	Customer administration	1990-12-13
211	106	Database CUSTOMERS	1990-12-29

Element queries

An element query checks whether a value is one of the elements in a set.

```
sql_expression1 [NOT] IN { subquery | (sql_expression2,sql_expression3,...) }
```

sql_expression1

Expression for the comparison.

sql_expression1 must be either an atomic value or the name of a multiple column. If the *sql_expression1* is a multiple column, the column specification cannot be an external reference (see metavariable *set_function*), and you cannot specify a subquery as the second operand.

subquery

Subquery that returns a single-column table.

sql_expression1 and the subsequent expressions *sql_expression2*, *sql_expression3*,... or the values resulting from the subquery must have compatible data types.

(*sql_expression2*, *sql_expression3*,...) cannot include any multiple expressions.

Result

sql_expression1 is an atomic value:

Without NOT:

True if the comparison with at least one expression or value from the subquery is **true**.

False if the comparison with all the expressions or all the values from the subquery are **false**, or if the derived column of the subquery is empty.

Unknown in all other cases.

With NOT:

identical to:

NOT (*sql_expression* IN *subquery*) or NOT (*sql_expression1* IN (*sql_expression2*,...))

sql_expression1 is a multiple column:

- The element query is performed for each occurrence of *sql_expression1*.
- The individual results are ORed.

Example

If X is a multiple column with 3 elements, the element query

```
X(1-3) IN (13, 20, 30)
```

is equivalent to the following element queries:

```
X(1) IN (13, 20, 30) OR
X(2) IN (13, 20, 30) OR
X(3) IN (13, 20, 30)
```

Examples

1. Element query with alphanumeric values:
Select the customers from Munich or Berlin from the customers table. Include the customer information in the output:

```
DECLARE ... CURSOR FOR SELECT company, cust_info, city
      FROM customer
      WHERE city IN ('Munich','Berlin');
...

```

company	cust_info	city
Siemens AG	Electrical	Munich
Login GmbH	PC networks	Munich
Plenzer Trading	Fruit market	Munich
Freddys Fishery	Unit retail	Berlin

2. Element query with derived column.
Select the orders for which no training was performed from the orders and service tables:

```
SELECT cust_num
      FROM orders
      WHERE order_num
      NOT IN (SELECT order_num
            FROM service
            WHERE service_text = 'Training');
```

cust_num
106

Pattern comparison

In a pattern comparison, an alphanumeric value is checked to see if it matches a specified pattern. A pattern is a string that, in addition to normal characters, can also include placeholders and escape characters.

A placeholder represents one or more characters. A placeholder can also occur in a pattern as a normal character if its special meaning is canceled with the escape character. You can define the escape character with the ESCAPE clause.

This predicate is only permitted within SQL statements.

```
[ table . ] { column | column(pos_no) | column(min-max) }
           [ NOT ] LIKE pattern [ESCAPE character ]
```

```
pattern ::= value
```

```
character ::= value
```

table

Name of the table that contains *column*. If a correlation name is defined for the table, you specify the correlation name instead of the table name.

column

Name of a column from which the values are taken. The data type of the column must be alphanumeric.

pos_no

Unsigned integer.

column is a multiple column. *column* cannot be a column from a higher-level query expression.

The value is the value of the $(pos_no - col_{min} + 1)$ th element of *column*.

If *column* is not a multiple column, *pos_no* is smaller than *col_{min}* or *pos_no* is greater than *col_{max}*, a SESAM error message is issued.

col_{min} and *col_{min}* are the smallest and largest position numbers of the multiple column.

min-max

Unsigned integers.

column is a multiple column. *column* cannot be a column from a higher-level query expression.

The value is the aggregate of the column elements ($min-col_{min}+1$) to ($max-col_{min}+1$).

If *column* is not a multiple column, *min* is not smaller than *max*, *min* is smaller than col_{min} or *max* is greater than col_{max} , a SESAM error message is issued.

col_{min} and col_{min} are the smallest and largest position numbers of the multiple column.

pos_no or *min-max* omitted:

column cannot be a multiple column.

pattern

Alphanumeric value to which the value from *column* is to be matched. *pattern* can include the following:

- normal characters (i.e. without placeholders and escape characters)
- placeholders

Placeholder	Meaning
_ (underscore)	any character
%	any (including empty) sequence of characters

Table 4: Placeholders in a pattern comparison

- escape characters (followed by placeholders or escape characters)

Blanks in *pattern*, even at the beginning or end, are part of the pattern.

ESCAPE clause

You use the ESCAPE clause to define an escape character. If you place an escape character in front of a placeholder, the placeholder loses its function as a placeholder and is interpreted instead as a normal character. You can also use the escape character to cancel the special meaning of the escape character and use it as a normal character.

character

Alphanumeric character with a length 1.

In this comparison, *character* acts as an escape character.

ESCAPE omitted:

No escape character is defined.

Result

column is an atomic column:

Unknown if the value of *column*, *pattern* or *character* is the NULL value, otherwise:

Without NOT:

True if the placeholders in *pattern* produce a value that is the same as the value from *column* and has the same length.

False in all other cases

With NOT:

True if the placeholders in *pattern* do not result in a value that matches the value from *column* and has the same length.

False in all other cases.

column is a multiple column:

- The pattern comparison is performed for every occurrence in *column*.
- The individual results are ORed.

Examples

1. Select all the contact people from the `contacts` table whose first name starts with Ro:

```
DECLARE ... CURSOR FOR SELECT fname, lname
FROM contacts
WHERE fname LIKE 'Ro%';
...
```

fname	lname
Roland	Loetzerich
Robert	Heinlein

2. The following statement selects all strings that start with the underscore character and end with at least one blank from an alphanumeric column `col` from a table `tab`:

```
SELECT * FROM tab
WHERE col LIKE '@_%' ESCAPE '@'
```

Comparison with the NULL value

A comparison is performed to check whether a column contains the NULL value.

```
[ table . ] { column | column(pos_no) | column(min-max) }
           IS [ NOT ] NULL
```

table

Name of the table that contains *column*. If a correlation name is defined for the table, you specify the correlation name instead of the table name.

column

Name of a column from which the values are to be taken.

pos_no

Unsigned integer.

column is a multiple column. *column* cannot be a column from a higher-level query expression.

The value is the value of the $(pos_no - col_{min} + 1)$ th element of *column*.

If *column* is not a multiple column, *pos_no* is smaller than col_{min} or *pos_no* is greater than col_{max} , a SESAM error message is issued.

col_{min} and col_{min} are the smallest and largest position numbers of the multiple column.

min-max

Unsigned integers.

column is a multiple column. *column* cannot be a column from a higher-level query expression.

The value is the aggregate of the column elements $(min - col_{min} + 1)$ to $(max - col_{min} + 1)$.

If *column* is not a multiple column, *min* is not smaller than *max*, *min* is smaller than col_{min} , or *max* is greater than col_{max} , a SESAM error message is issued.

col_{min} and col_{min} are the smallest and largest position numbers of the multiple column.

pos_no or *min-max* omitted:

column cannot be a multiple column.

Result

column is an atomic column:

Without NOT:

True if the value of *column* is the NULL value.

False in all other cases.

With NOT:

True if the value in *column* is not the NULL value.

False in all other cases.

column is a multiple column:

Without NOT:

True if at least one occurrence of *column* is the NULL value.

False in all other cases.

With NOT:

True if at least one occurrence *column* is not the NULL value.

False in all other cases.

Example

Select the orders from the `orders` table that have not yet been dealt with completely, i.e. for which the actual date is the NULL value. The order text and the target date should also be output.

```
DECLARE ... CURSOR FOR SELECT order_num, order_text, target
      FROM orders
      WHERE actual IS NULL;
```

order_num	order_text	target
250	Mail merge intro	1991-03-01
251	Customer administration	1991-05-01
300	Network test/comparison	
305	Staff training	1991-02-27



The specified form of the predicate is only permitted in SQL statements. In DRIVE statements, a *value* can be specified instead of a column reference (see the “Directory of DRIVE Statements” [3], metavariable *expression*).

Existence queries

An existence query checks whether a derived table is empty.

This predicate is only permitted within SQL statements.

EXISTS subquery

subquery

Subquery that returns a derived table.

Result

True if the derived table is not empty.

False if the derived table is empty.

Example

Select the customers that have not placed an order from the `customers` table:

```
DECLARE ... CURSOR FOR SELECT company
    FROM customer
    WHERE NOT EXISTS
        (SELECT order_num
         FROM order
         WHERE order.cust_num = customer.cust_num);
...

```

```
company
Siemens AG
Plenzer Trading
Freddys Fishery
Externa & Co Kg

```

select_expression

select_expression ::=

```
SELECT [ { ALL | DISTINCT } ] select_list
```

```
FROM table_specification,...
```

```
[ WHERE condition ]
```

```
[ GROUP BY column,... ]
```

```
[ HAVING condition ]
```

```
select_list ::= { * | { table.* | sql_expression [ [ AS ] column ] },... }
```

ALL

ALL is the default value.

Duplicate records are retained in the derived table.

DISTINCT

Duplicate rows are removed (duplicate elimination).

DISTINCT can only be used once on any level of a SELECT query. You cannot specify the following, for example:

```
SELECT DISTINCT COUNT(DISTINCT ...) ...
```

You can create a join using the FROM clause and, if appropriate, the WHERE clause. This means that you can join two or more tables and create the Cartesian product of all the tables involved (see also metavariable *join_expression*).

The following applies to all clauses:

- The clauses must be specified in the given order.
- Column names must be unique. If a column name occurs in several tables, you must qualify the column name with the table name. If you rename a table using a correlation name for the duration of the SELECT statement (see metavariable *table_specification*), you must use only the correlation name.

Example

```
SELECT o.cust_num, s.service_price
FROM orders o, service s
WHERE o.order_num=s.order_num;
```

Evaluation of SELECT expressions

SELECT expressions are evaluated in the following order:

1. The Cartesian product from all the table specifications in the FROM clause is created (see metavariable *table_specification*: there are base tables, view tables, temporary view tables, derived tables and join tables).
2. If a WHERE clause is specified, the WHERE condition is applied to all the rows of the Cartesian product. The rows for which the condition returns the value **true** are selected.
3. If a GROUP BY clause is specified, the rows determined in point 2 or point 1 (if no WHERE clauses is specified) are combined into groups. All the rows comprising a group have the same value in the column specified in the GROUP BY clause.
4. If a HAVING clause is specified, the HAVING condition is applied to all the groups. The groups that satisfy the condition are selected. If no GROUP BY clause is specified, all the previously selected rows are considered a group.
5. If *select_list* includes a set function and the derived table has not yet been divided into groups, all the rows in the derived table are combined to form a group.
6. If the derived table has been divided into (one or more) groups, *select_list* is evaluated for each group.

If the derived table has not been divided into groups, *select_list* is evaluated for each derived row.
7. If DISTINCT is specified, duplicate rows are removed.

The resulting rows then form the derived table of the *select_expression*.

select_list - Select derived columns

You determine the columns in the derived table with the *select_list*.

```
select_list ::= { * | { table.* | sql_expression [ [ AS ] column ] }, ... }
```

*

Select all columns. The order and the names of the columns in the table specified in the FROM clause are used. If several tables are involved, the order of the tables in the FROM clause is used.

*table.**

All the columns in *table* are selected. *table* must be included in the FROM clause. The order and the names of the columns in *table* are used.

sql_expression

Expression denoting a derived column. If *sql_expression* contains a column specification, the table to which the column belongs must be included in the FROM clause of this or a higher level *select_expression*.



The names of the columns in *select_list* must be unique. If you join tables and these base tables have columns with identical names, you must qualify the names using the table or correlation name so that they can be uniquely identified.

If a set function (AVG, COUNT, MAX, MIN, SUM) occurs in a column selection, the following restriction applies:

Only column names that are specified in the GROUP BY clause or which are arguments in the set function can be included in *select_list*.

[AS] *column*

Name of the derived column specified with *sql_expression*.

Example

```
SELECT order_num AS order_no, COUNT(*) AS total FROM orders
GROUP BY order_num
```

order_no	total
...	...

column omitted:

If *sql_expression* is a column name, the derived column is assigned this name. Otherwise, the column name is not defined.

Example

```
SELECT order_num, COUNT(*) FROM orders GROUP BY order_num
```

order_num
...

Columns in the derived table

The order of the columns in the derived table corresponds to the order of the columns in *select_list*.

The attributes of a derived column (data type, length, precision, digits to the right of the decimal point) are either taken from the underlying column or result from the specified expression.

The NULL value is permitted for a derived column if one of the following conditions is satisfied:

- The NULL value is permitted for the source column.
- *sql_expression* contains an indicator variable, a subquery, SYSTEM USER or one of the set functions AVG, MAX, MIN or SUM.

FROM clause - Specify tables

You use the FROM clause to specify the tables from which data is to be selected.

In order to read the specified tables, you must either own these tables or have SELECT permission.

```
FROM table_specification,...
```

table_specification

Specification of a table from which data is to be read. You can only specify tables located in the same database. You can therefore qualify all the table names in the FROM clause with, at the most, one database name that is the same for all the tables. If you do not qualify all the table names with such a mutual database name, you can only use the default database name to qualify the table names (see OPTION CATALOG for static SQL statements and SET CATALOG and SET SCHEMA for dynamic SQL statements).

A table name *table* in a *table_specification* is referred to as being “synonym-free” if no *correlation_name* for *table* is specified in the *table_specification*. All correlation names and all synonym-free table names in the FROM clause must be different.

WHERE clause - Select derived rows

You use the WHERE clause to specify a condition for selecting the rows for the derived table. The derived table contains only the rows that satisfy the condition (i.e. the condition is **true**). Rows for which the condition returns the value **false** or **unknown** are not included in the derived table.

WHERE condition

condition

Condition that the selected rows must satisfy (see metavariable *condition*).

GROUP BY clause - Group derived rows

You use the GROUP BY clause to combine table rows into groups. Two rows belong to the same group if, for each grouping column, the values in both rows are the same with regard to the comparison rules (see metavariable *predicate*), or both values are the NULL value.

The derived table contains a row for each group.

GROUP BY column,...

column

Grouping column. *column* must be part of a table that was specified in the FROM clause. Ambiguous column names must be qualified with the table name. If you declared a correlation name for the table involved in the FROM clause, you must use this name to qualify the column names.

Multiple columns cannot be used as the grouping column.

Effect of the GROUP BY clause

If you specify the GROUP BY clause, only columns listed in GROUP BY or which are arguments in a set function can be included in *select_list*.

Set functions for columns of a grouped table are evaluated for each group.

How are groups created?

- A group is a set of rows that all have the same values in each specified grouping column according to the comparison rules (see above).
- Rows that have the NULL value in the same column and the same values in the other columns also constitute a group.

Example

You want to list the average amount of `vat` for each order number:

```
DECLARE ... CURSOR FOR SELECT order_num, AVG(vat_rate) AS vat
FROM service
GROUP BY order_num;
```

...

order_num	
200	0.14
211	0.06
250	0.07

HAVING clause - Select groups

You use the HAVING clause to specify conditions for selecting groups. If a group satisfies the specified condition, the row for that group is included in the derived table. If no GROUP BY clause is specified, all the rows are considered one group.

HAVING condition

condition

Condition to be satisfied by a group (see metavariable *condition*).

Unlike a WHERE search condition, which is evaluated for each row in a table, the HAVING search condition is evaluated once for each group.

A column name in *condition* must satisfy one of the following conditions:

- The column is included in the GROUP BY clause.
- The column name is an argument in a set function (AVG(), SUM(), ...). If the column name is also included in *select_list*, it can only occur there as an argument of a set function.
- The column name occurs in a subquery. If the column name references the table in the FROM clause, it must be included in the GROUP BY clause or be the argument in a set function.
- The column name is part of a table from a higher-level *select_expression*.

Example

You want to display the latest service provided for each order, but only if it was provided after 1.1.1991:

```
SELECT order_num, MAX(service_date)
FROM service
GROUP BY order_num
HAVING MAX(service_date) > DATE'1991-01-01';
```

column_constraint

When a base table is created or updated (CREATE TABLE, ALTER TABLE), column constraints can be specified in the column definitions for the individual columns. The column cannot be a multiple column.

A column constraint is an integrity constraint on a single column. All the values in the column must satisfy the integrity constraint.

```
column_constraint ::=
```

```
{ NOT NULL |  
  UNIQUE |  
  PRIMARY KEY |  
  CHECK ( condition ) |  
  REFERENCES table [ ( column ) ]
```

NOT NULL

Non-NULL constraint.

The column cannot contain any NULL values.

The NOT NULL constraint is stored as a check constraint (*column* IS NOT NULL).

UNIQUE

Uniqueness constraint.

Non-NULL column values must be unique.

The column must not be longer than 256 characters.

PRIMARY KEY

Primary key constraint.

The column is the primary key of the table. The values in the column must be unique. Only one primary key can be defined for each table.

The column cannot have the data type VARCHAR. The length of the column must be between 4 and 256 characters.

The NOT NULL constraint applies implicitly to a primary key column.

CHECK (*condition*)

Check constraint.

Each value in the column must satisfy the *condition*. The following restrictions apply to *condition*:

- *condition* cannot contain any variables
- *condition* cannot contain any set functions
- *condition* cannot contain any subqueries, i.e. *condition* can only reference the column of the table to which the column constraint belongs
- *condition* cannot contain a time function
- *condition* cannot contain USER, CURRENT USER or SYSTEM USER.

REFERENCES

Referential constraint.

The column of the referencing table can only contain a non-NULL value if the same value is included in the referenced column of the referenced table:

The current authorization identifier must have the REFERENCES privilege for the referenced columns.

table

Name of the referenced base table.

The referenced base table must be an SQL table. The name of the referenced base table can be qualified by a database or schema name. The database name must be the same as the database name of the referencing table.

(*column*)

Name of the referenced column.

The referenced column must be defined with UNIQUE or PRIMARY KEY. The referenced column cannot be a multiple column. Referencing column and referenced column must have exactly the same data type.

(*column*) omitted:

The primary key of the referenced column is used as the referenced column. Referencing column and referenced column must have exactly the same data type.

column_definition

When a base table is created or updated (CREATE TABLE, ALTER TABLE), the column definition defines the name and the attributes of a column.

SESAM/SQL distinguishes between atomic and multiple columns. In an atomic column, exactly one value can be stored in each row. In a multiple column, several values of the same type can be stored in each row.

A base table can contain a maximum of 26,134 columns of any data type except VARCHAR. It can contain up to 1000 VARCHAR columns.

```
column_definition ::=
    column [ ( dimension ) ] { basic_data_type | FLOAT ( precision ) }
                                     [ default ]
    [ [ CONSTRAINT integrity_constraint_name ] column_constraint ... ]

dimension ::= unsigned_integer

default ::= DEFAULT { literal |
                    CURRENT DATE |
                    CURRENT TIME |
                    CURRENT TIMESTAMP |
                    [ CURRENT ] USER |
                    SYSTEM USER |
                    NULL }
```

column

Name of the column. The column name must be unique within the base table.

dimension

Unsigned integer between 1 and 255. *dimension* indicates the number of column elements in a multiple column.

dimension omitted: The column is an atomic column.

dimension cannot be specified for VARCHAR (*length*) and CHAR VARYING (*length*).

basic_data_type

Data type for the column (see the “Directory of DRIVE Statements” [3], metavariable *basic_data_type*).

For CHAR, the length must be less than 257.

For the data types NUMERIC and DECIMAL, *precision* must not exceed 31.

The default value for *precision* is 1.

The abbreviation SMINT is not permitted or the data type SMALLINT.

The abbreviation NUM is not permitted for the data type NUMERIC.

The data type EXTENDED DECIMAL or XDEC is not permitted.

FLOAT indicates the data type FLOAT(1) (and not DOUBLE PRECISION).

The data type TIME is not permitted (only TIME(3)).

The data type INTERVAL is not permitted.

user_type is not permitted.

FLOAT (*precision*)

For the data type FLOAT, *precision* indicates the binary length of the mantissa. It must be greater than 0 and smaller than 54. The range of values for FLOAT corresponds to that for REAL if *precision* is less than or equal to 21. Otherwise, it corresponds to that of DOUBLE PRECISION. The default value for *precision* is 1.

default

Defines an SQL default value that is entered in the column if a row is inserted or updated and no value is specified for the column, not even the NULL value.

- *column* cannot be a multiple column.
- *column* cannot be a CALL DML column.
- *default* must observe the assignment rules for default values (see the “SESAM/SQL-Server Language Reference manual, Part 1” [18], section 4.4.1, “Entering values in table columns”).
- The default value must satisfy the column constraint.

The default setting is evaluated when a row is inserted or updated and the default value is to be used for *column*.

default omitted:

There is no SQL default value.

The NULL value is entered in columns without a NOT NULL constraint.

For reasons of compatibility with the SQL standard, it is recommended that you specify a *length* greater or equal to 128 for the data types CHAR(*length*) or VARCHAR(*length*) for CURRENT USER or SYSTEM USER.

[CONSTRAINT *integrity_constraint_name*] *column_constraint*

Defines an integrity constraint for the column. Integrity constraints must not be specified for multiple columns.

[CONSTRAINT *integrity_constraint_name*] *column_constraint* omitted:
No column constraint defined.

CONSTRAINT *integrity_constraint_name*

Defines a name for the integrity constraint. The unqualified name of the integrity constraint must be unique within the schema. The name of the integrity constraint can be qualified with a database and schema name. This database and schema name must be the same as the database and schema name of the base table for which the integrity constraint is generated.

CONSTRAINT *integrity_constraint_name* omitted:

The integrity constraint is assigned a name according to the following pattern:

{ UN | PK | FK | CH } *integrity_constraint_number*

where UN stands for UNIQUE, PK for PRIMARY KEY, FK for FOREIGN KEY and CH for CHECK. *integrity_constraint_number* is a 16-digit number (time stamp). The NOT NULL constraint is stored as a check constraint.

column_constraint

Indicates an integrity constraint that the column must satisfy (see metavariable *column_constraint*).

sql_expression

sql_expression returns a value. *sql_expression* can occur in:

- column selection (*select_expression*, SELECT statement)
- predicates in conditions (e.g. WHERE clause, HAVING clause)
- assignments (INSERT, UPDATE statement).

sql_expression consists of operands and can include operators. If an operand is the NULL value, the total result is also the NULL value. Otherwise the operators are used on the results of the operands. The result of the evaluation is an alphanumeric, numeric or a time value.

The operands are not evaluated in a predefined order. In certain cases, a partial expression is not calculated if it is not required for calculating the total result.

```
sql_expression ::=
    { value |
      [ table. ] { column | column(pos_no) | column(min-max) } |
      { + | - } sql_expression |
      sql_expression { * | / | + | - | || } sql_expression |
      ( sql_expression ) |
      subquery |
      set_function |
      time_function |
      [ CURRENT ] USER | SYSTEM USER }
```

```
pos_no ::= unsigned_integer
```

```
min ::= unsigned_integer
```

```
max ::= unsigned_integer
```

value

Alphanumeric value, numeric value or time value (see metavariable *value*).

table

Name of the table containing *column*. If a correlation name has been defined for the table, you must specify the correlation name instead of the table name (see metavariable *table_specification*).

column

Name of the column from which the values are to be taken (column reference).

pos_no

Unsigned integer.

The value is taken from the $(pos_no - col_{min} + 1)$ th column element of the multiple column *column* and can be used as an atomic value.

If *column* is not a multiple column, *pos_no* is smaller than col_{min} or *pos_no* is greater than col_{max} , a SESAM error message is issued.

col_{min} and col_{max} are the smallest and largest position numbers of the multiple column.

min-max

Unsigned integers.

The value is the aggregate from the column elements $(min - col_{min} + 1)$ to $(max - col_{min} + 1)$ of the multiple column *column*.

If *column* is not a multiple column, *min* is not smaller than *max*, *min* is smaller than col_{min} or *max* is greater than col_{max} , a SESAM error message is issued.

col_{min} and col_{max} are the smallest and largest position numbers of the multiple column.

pos_no or *min-max* omitted:

column must not be a multiple column.

– *sql_expression*, + *sql_expression*

“–” changes the sign, i.e. the value of *sql_expression* is negated. “+” does not change the value of *sql_expression*. *sql_expression* must be numeric and cannot be a multiple value with a dimension > 1. *sql_expression* cannot start with “+” or “–”.

The following variants for numeric values and expressions, in particular, are therefore permitted:

{ + | – } { column | value | set_function | (sql_expression) }

sql_expression { + | – | * | / | || } *sql_expression*

Indicates the arithmetic operations addition, subtraction, multiplication and division, as well as concatenation. For the arithmetic operations +, –, * and /, both operands must be numeric. For concatenation (||), both expressions must be alphanumeric. None of the operands may be a structured variable or an aggregate with more than one component.

If *a* and *b* are of the data type CHARACTER, the result has the data type CHARACTER with the length $l_a + l_b$. The maximum number of characters is, however, 256.

If a or b is of the type VARCHAR, the result has the data type VARCHAR with a length of l_a+l_b . The maximum number of characters is, however, 32000.

l_a and l_b are the lengths of a and b .

If a result of the type CHARACTER is longer than 256 characters, a SESAM error message is issued.

If a result of the type VARCHAR is longer than 32000 characters, the string is truncated from the right to a length of 32000. If characters that are not blanks are removed, a SESAM error message is issued.

$a * b$

Multiply a with b .

The expressions a and b must be numeric.

If a and b are integers or fixed-point numbers, the result is an integer or fixed-point number with t_a+t_b significant digits. The maximum number of digits is, however, 31. The number of digits to the right of the decimal point is r_a+r_b , with a maximum number of 31 digits.

t_a and t_b are the total number of significant digits for a and b .

r_a and r_b are the number of digits to the right of the decimal point for a and b respectively.

If a or b is a floating-point numbers, the result is a floating-point number with a total number of significant digits of 24 bits for REAL numbers and 56 bits for DOUBLE PRECISION numbers.

If the result value is too big for the resulting data type, a SESAM error message is issued. If the total number of significant digits is too big, the number is rounded.

a / b

Divide a by b .

The expressions a and b must be numeric.

If a and b are integers or fixed-point numbers, the result is an integer or fixed-point number with 31 significant digits. The number of digits to the right of the decimal point is $31-l_a-r_b$, at least however 0.

l_a is the number of digits to the left of the decimal point for a .

r_b is the number of digits to the right of the decimal point for b .

If a or b is a floating-point number, the result is a floating-point number with a total number of significant digits of 24 bits for REAL numbers and 56 bits for DOUBLE PRECISION numbers.

If the result value is too big for the resulting data type or the value of b is 0, a SESAM error message is issued. If the total number of significant digits is too big, the number is rounded.

$a + b$

Add a and b .

The expressions a and b must be numeric.

If a and b are integers or fixed-point numbers, the result is an integer or fixed-point number with $l_{max} + r_{max} + 1$ significant digits. The maximum number of digits is, however, 31. The number of digits to the right of the decimal point is r_{max} .

l_{max} is the larger of the two numbers of digits to the left of the decimal point for a and b .

r_{max} is the larger of the two numbers of digits to the right of the decimal point for a and b .

If a or b is a floating-point number, the result is a floating-point number with a total number of significant digits of 24 bits for REAL numbers and 56 bits for DOUBLE PRECISION numbers.

If the result value is too big for the resulting data type, a SESAM error message is issued. If the total number of significant digits is too big, the number is rounded.

 $a - b$

Subtract b from a .

The expressions a and b must be numeric.

If a and b are integers or fixed-point numbers, the result is an integer or fixed-point number with $l_{max} + r_{max} + 1$ significant digits. The maximum number of digits is, however, 31. The number of digits to the right of the decimal point is r_{max} .

l_{max} is the larger of the two numbers of digits to the left of the decimal point for a and b .

r_{max} is the larger of the two numbers of digits to the right of the decimal point for a and b .

If a or b is a floating-point number, the result is a floating-point number with a total number of significant digits of 24 bits for REAL numbers and 56 bits for DOUBLE PRECISION numbers.

If the result value is too big for the resulting data type, a SESAM error message is issued. If the total number of significant digits is too big, the number is rounded.

 $a || b$

Concatenate a and b .

The expressions a and b must be alphanumeric.

If a and b are of the type CHARACTER, the resulting data type is CHARACTER with a length of $l_a + l_b$. The maximum number of characters is, however, 256.

If a or b is of the type VARCHAR, the result has the data type VARCHAR with a length of l_a+l_b . The maximum number of characters is, however, 32000.
 l_a and l_b are the lengths of a and b .

If a result of the type CHARACTER is longer than 256 characters, a SESAM error message is issued.

If a result of the type VARCHAR is longer than 32000 characters, the string is truncated from the right to a length of 32000 characters. If characters are removed that are not blanks, a SESAM error message is issued.

(sql_expression)

You can use parentheses to group parts of expression together to form a unit, thus changing the order in which the arithmetic expressions are evaluated. Parentheses must be set according to the rules for algebra.

subquery

Subquery (see metavariable *subquery*) that returns exactly one value.

set_function

Set function (see metavariable *set_function*). *sql_expression* indicates the value that this function returns

set_function may only occur in the *select_list* a SELECT statement or a *select_expression*.

time_function

Time function (see metavariable *time_function*).

[CURRENT] USER | SYSTEM USER

Specification of SESAM user functions. The result of [CURRENT] USER is the current authorization identifier (see CREATE USER statement). It is an alphanumeric literal of the type CHARACTER (18).

The result of SYSTEM USER is the name of the current system user. The name is constructed from the host name, the UTM application name (or blanks) and the UTM or BS2000 user ID (see CREATE SYSTEM_USER statement). It is an alphanumeric literal of the type CHARACTER(24).

Example

```
CREATE TABLE current_users (counter INTEGER PRIMARY KEY,  
                             sqluser CHARACTER (18),  
                             systemuser CHARACTER (24),  
                             stamptime TIMESTAMP (3));  
  
COMMIT WORK;  
  
INSERT INTO current_users VALUES (*,  
                                   CURRENT USER,  
                                   CURRENT SYSTEM USER,  
                                   CURRENT TIMESTAMP)  
RETURN INTO &counter;
```

Precedence

- Expressions enclosed in parentheses have highest precedence.
- Monadic operators take precedence over dyadic operators.
- The operators for multiplication (*) and division (/) take precedence over the operators for addition (+) and subtraction (-).
- Operators for multiplication all have the same precedence level.
- Operators for addition all have the same precedence level.
- Operators with the same precedence level applied from left to right.



If *sql_expression* does not contain column references, set functions, subqueries or functions, this metavariable is identical to the corresponding part of the DRIVE metavariable *expression* (see the “Directory of DRIVE Statements” [3]). Nevertheless, *sql_expression* within an SQL statement is always calculated by SESAM. DRIVE only calculates *expression* within DRIVE statements.

table_specification

You use *table_specification* to specify a table.

```
table_specification ::= { table [ [ AS ] correlation_name [ (column,...) ] ] |
                        subquery [ AS ] correlation_name [ (column,...) ] |
                        join_expression }
```

```
table ::= { [ [ catalog. ] unqual_schema_name. ] unqual_base_table_name |
           [ [ catalog. ] unqual_schema_name. ] unqual_view_name |
           temp_view_name }
```

table

Name of a base table, view or temporary view.

catalog

Name of a database (catalog space *catalog.CATALOG* and user spaces *catalog.spacename*). A database name can be up to 18 characters long (see the CREATE CATALOG statement in the SESAM manual [19]).

unqual_schema_name

Name of a schema. The unqualified schema name must be unique within the database. An unqualified schema name can be up to 31 characters long (see the CREATE SCHEMA statement).

unqual_base_table_name

Name of a base table. The unqualified name of a base table must be unique within the base table and (permanent) view names of your schema. An unqualified base table name can be up to 31 characters long (see the CREATE TABLE statement).

unqual_view_name

Name of a permanent view. The unqualified name of a view must be unique within the base table and view names of its schema. An unqualified view name can be up to 31 characters long (see the CREATE VIEW statement).

temp_view_name

Name of a temporary view. The name of a temporary program view must be unique within the compilation unit (DRIVE program). The name of a temporary dialog view must be unique within the DRIVE session or UTM conversation. A temporary view name can be up to 31 characters long; the name of a static temporary program view can be up to 24 characters long (see the CREATE TEMPORARY VIEW statement).

The same table can occur more than once in a table specification in *query_expression*. The correlation names of tables must be used to make a distinction between the different occurrences of the same table.

correlation_name

Table name used in *query_expression* as a new name for the table (correlation name or synonym). A correlation name can be up to 18 characters long.

The *correlation_name* must be used to qualify the column name in every column specification that references this instance of the table.

The new name must be unique, i.e. *correlation_name* may only occur once in a table specification of this *query_expression*.

You must give a table a new name if the columns in the table cannot otherwise be identified uniquely in *query_expression*.

In addition, you may give a table a new name in order to formulate *query_expression* so that it is more easily understood or as an abbreviation of long names.

Example

Joining a table with itself:

```
SELECT a.company, b.company           /* Query customer who lives */
   FROM customers AS a, customers AS b
   WHERE a.city = b.city             /* in the same city and */
   AND a.cust_num < b.cust_num      /* avoid duplicates */
```

column,...

Column name that is used within *query_expression* as the new name for the column of the corresponding table.

If you rename a column, you must assign all the columns in the table a new name.

column is the new name of the column and must be unique within the table specified by *correlation name*. This column can only be referenced with the new name in this *query_expression*.

The columns of a derived table must be renamed if the column names of the table upon which it is based are not unique, or if the derived columns are to be referenced using names that have been assigned internally.

Example

You want to give the columns in the WAREHOUSE table new, more informative names:

```
SELECT * FROM warehouse w (item_number, current_stock, location)
  WHERE location = 'Parts warehouse (east)'
```

column,... omitted:

The column names of the associated table are valid. These could be names that are assigned internally, which cannot be referenced in *query_expression*.

subquery

The table is the derived table that results from evaluating the subquery (see metavariable *subquery*).

join_expression

Join expression that determines the tables from which the data is to be selected (see metavariable *join_expression*).

Underlying tables

Depending on the specification made in the table specification, the underlying table is defined as follows:

Specification in table specification	Underlying table
Base table	Base table
View or temporary view	Base tables which the (temporary) view references directly or indirectly
Subquery	The table upon which the subquery is based (see metavariables <i>select_expression</i> and <i>query_expression</i>)
Join expression	The table upon which the table specification in the <i>join_expression</i> is based

table_constraint

When a base table is created or updated (CREATE TABLE, ALTER TABLE), table constraints can be specified. A table constraint is an integrity constraint on one or more columns in the base table. None of the columns may be a multiple column.

```
table_constraint ::=
```

```
{ UNIQUE ( { column,... } ) |
  PRIMARY KEY ( { column,... } ) |
  FOREIGN KEY ( { column,... } ) REFERENCES table [ ( { column },... ) ] |
  CHECK ( condition )
```

UNIQUE (*column*,...)

Uniqueness constraint.

The set of column values that are not equal to NULL must be unique.

The sum of the lengths of the columns plus the total number of columns cannot exceed 256.

PRIMARY KEY (*column*,...)

Primary key constraint.

The specified column constitutes the primary key of the table. The set of column values must be unique. Only one primary key can be defined for each table.

None of the columns can be VARCHAR columns. The sum of the column lengths must be between 4 and 256 characters.

The NOT NULL constraint applies implicitly to the primary key columns.

FOREIGN KEY ... REFERENCES

Referential constraint.

The referencing columns (FOREIGN KEY clause) can only contain a set of values that does not include any NULL values if the set of values also occurs in the referenced columns (REFERENCES clause).

You must specify the same number of columns in the referencing and referenced table. The data types of the corresponding columns must be exactly the same.

The current authorization identifier must have the REFERENCES privilege for the referenced column.

FOREIGN KEY (*column*,...)

Columns of the referencing table whose sets of values should be contained in the referenced base table.

REFERENCES *table*

Name of the referenced base table.

The referenced base table must be an SQL table. The name of the referenced base table can be qualified with a database or schema name. The database name must be the same as the database name of the referencing table.

(*column,...*)

Names of the referenced columns.

A uniqueness or primary key constraint that uses the same columns and the same order must be defined for these columns. None of the columns may be a multiple column.

(*column,...*) omitted:

The primary key of the referenced table is used as the referenced column.

CHECK (*condition*)

Check constraint.

The condition *condition* must be satisfied for each row in the table. The following restrictions apply to *condition*.

- *condition* cannot include any variables.
- *condition* cannot include any set functions.
- *condition* cannot include any subqueries, i.e. *condition* can only reference columns of the table to which the column constraint belongs.
- *condition* cannot include a time function.
- *condition* cannot include USER, CURRENT USER or SYSTEM USER.

subquery

A subquery is the specification of a table as the derived table of a *query_expression* that can be used in

- expressions:
The subquery must return a single-column derived table with a maximum of one row. The value of the subquery is then the value in the derived table or the NULL value if the derived table is empty.
- predicates:
In the predicates ANY, SOME, ALL and IN, the subquery must return a single-column derived table. In the predicate EXISTS, the subquery can return any derived table.
- the FROM clause in SELECT expressions:
The subquery returns a derived table.
- join expressions:
The subquery returns a derived table.

A subquery is always enclosed in parentheses.

```
subquery ::= (query_expression)
```

query_expression

Query expression that returns the derived table.

In subqueries that are not specified in the predicate EXISTS or in a FROM clause or in a *join_expression*, the derived table can only contain an atomic column or multiple columns with the dimension 1.

variable

variable specifies a simple variable or a component of a structured variable. The list of all components that can be located on the next level can be specified with the partial qualification “*”. In DRIVE/WINDOWS *variable* has the following syntax (see also the “Directory of DRIVE Statements” [3]):

```
variable ::= { &varname1 [ suffix ] |
              &varname2 { ( index1, index2 ) |
                          ( index1, range2 ) |
                          ( range1, index2 ) }

suffix ::= { group_component | index_component }

group_component ::= . { * | component [ suffix ] }

index_component ::= { ( { index | range } ) }

index ::= unsigned_integer

range ::= index1 - index2
```

variable

Variables are used in an SQL statement to accept values from the database (output variable) or to store values in the database. They are also used to supply values required in calculations and conditions (input variables). The data types of columns and the associated variables must always be compatible.

varname1

Name of a simple variable or the first qualification of the components of a structured variable. *varname1* can be up to 31 characters long.

You will find a detailed explanation of the syntax under the metavariable *variable* in the “Directory of DRIVE Statements” [3].

suffix

Further qualification of the components of a structured variable.

.*

Abbreviated notation for the list of all variable components that are located as components on the next lower level.

component

Name of a component, i.e. part of a structured variable or the unqualified name of a table column (see the DECLARE VARIABLE... LIKE CURSOR/TABLE statement).

component can be up to 31 characters long.

varname2

Name of a matrix. *varname2* can be up to 31 characters long.



DRIVE/WINDOWS allows you create variables whose structure and names correspond to those of a table or cursor using the DECLARE VARIABLE ... LIKE statement.

value

You can specify a value either as a literal or via a variable. *value* defines the data value for a variable or a component of a variable. Each variable and each table column can be assigned the NULL value if no NOT NULL constraint was declared in the definition (DECLARE VARIABLE, CREATE/ALTER TABLE) (see the DRIVE statement SET and the SQL statements INSERT and UPDATE).

```
value ::= { literal | variable | aggregate }
```

```
aggregate ::= < { value | NULL }, ... >
```

literal

Alphanumeric literal, numeric literal or time literal (see metavariable *literal*).

variable

Name of a variable that contains the value (see metavariable *variable*).

aggregate

Specifies an aggregate, i.e. a structured value whose components are defined by *value* or NULL. *aggregate* cannot contain more than 255 components. An aggregate is also called a multiple value because it specifies the values for multiple columns. The number of components of an aggregate is also referred to accordingly as the dimension (see also metavariable *column_definition*).

Example

```
CREATE TABLE demo (demo (5) INTEGER);
COMMIT WORK;
INSERT INTO demo (demo (2-4)) VALUES (<13,NULL,67>);
```

The multiple column `demo` subsequently contains the multiple value `<NULL,13,NULL,67,NULL>`.

value

You may specify literals and variables for *value*, i.e. nested aggregates are not permitted. In the case of structured variables, you can only reference the lowest structure, but not the entire structure. In the case of vectors, only one component can be referenced.

NULL

The appropriate component of *aggregate* is assigned the NULL value. The NULL value can be assigned to any variable (SET statement) and any column in a table (INSERT and UPDATE statements).

time_function

Time functions return the current date and/or time:

```
time_function ::= { CURRENT DATE | CURRENT TIME | CURRENT TIMESTAMP }
```

CURRENT DATE

Returns the current date. **Data type:** DATE.

CURRENT TIME

Returns the current time. **Data type:** TIME(3).

CURRENT TIMESTAMP

Returns the current time stamp. **Data type:** TIMESTAMP(3).

If several time functions are included in an SQL statement, SESAM executes them all simultaneously. This is also true of all time functions that are evaluated as the result of the statement:

- time functions in the DEFAULT clause of the column definition if the default value is used
- time functions that occur in the SELECT expression of a view or temporary view if the view or temporary view is referenced

Time functions within DRIVE statements are executed by DRIVE/WINDOWS.

Time functions in dynamic statements and in cursor descriptions are evaluated when the EXECUTE statement is performed.

All the values that are returned have the same data and/or time. Therefore, you cannot use time functions to determine execution times within an SQL or DRIVE statement. You can, however, use time functions to determine the **approximate** time that it will take for individual statements or statement blocks to execute.

Examples

Example 1

```
DCL VAR &vorher TIMESTAMP(3),
      &nachher TIMESTAMP(3),
      &dauer INTERVAL FRACTIONS;
SET &vorher=CURRENT TIMESTAMP;
INSERT INTO demo VALUES (<1,2,3,4,5>);
SET &nachher=CURRENT TIMESTAMP;
SET &dauer=&nachher - &vorher;
DISPLAY FORM 'Duration of INSERT statement', NL 1,
      'in milliseconds:', &dauer;
```

Example 2

```
DCL VAR &vorher1 TIMESTAMP(3),
      &vorher2 TIMESTAMP(3),
      &nachher1 TIMESTAMP(3),
      &nachher2 TIMESTAMP(3),
      &dauer1 INTERVAL FRACTIONS,
      &dauer2 INTERVAL FRACTIONS,
      &lauf SMALLINT,
      &SCHLUESSEL CHAR(6),
      ARTNAM CHAR(10);
DCL C1 CURSOR FOR S SCHLUESSEL, ARTNAM FROM FIRMA;
DCL C2 CURSOR PREFETCH 200
      FOR S SCHLUESSEL, ARTNAM FROM FIRMA;
DCL VAR &V LIKE CURSOR C1;
CYCLE FOR &lauf = 1 TO 10;
      SET &vorher1 = CURRENT TIMESTAMP;
      CYCLE C1 INTO &V.*;
      END CYCLE;
      SET &nachher1 = CURRENT TIMESTAMP,
```

```
        SET &dauer1 = &dauer1 + (&nachher1 - vorher1);
        END CYCLE;
SET &dauer1 = &dauer1/10;
CYCLE FOR &lauf = 1 TO 10;
        SET &vorher2 = CURRENT TIMESTAMP;
        CYCLE C2 INTO &V.*;
        END CYCLE;
        SET &nachher2 = CURRENT TIMESTAMP,
        SET &dauer2 = &dauer2 + (&nachher2 - vorher2);
END CYCLE;
SET &dauer2 = &dauer2/10;
DISPLAY FORM 'average processing duration for cursor table', NL1,
        'with ', &anzahl, 'rows with the length 16', NL1,
' a) without PREFETCH: ', &dauer1, 'milliseconds', NL1,
' b) with PREFETCH 200: ', &dauer2, 'milliseconds';
```

5 Syntax overview

5.1 Statements

ALTER TABLE - Alter base table

```
ALTER TABLE table
```

```
{ ADD [ COLUMN ] column_definition |
```

```
  ALTER [ COLUMN ] column
    { DROP DEFAULT |
      SET basic_data_type |
      SET default } |
```

```
  ADD [ CONSTRAINT integrity_constraint_name ] table_constraint |
```

```
  DROP CONSTRAINT integrity_constraint_name RESTRICT }
```

```
default::= DEFAULT { literal |
                    CURRENT DATE |
                    CURRENT TIME |
                    CURRENT TIMESTAMP |
                    [ CURRENT ] USER | SYSTEM USER |
                    NULL }
```

CLOSE - Close cursor

CLOSE cursor

COMMIT WORK - Terminate transaction

COMMIT [WORK] [WITH { display | send message | stop }]

CREATE SCHEMA - Create schema

CREATE SCHEMA
 { [catalog .] schema [AUTHORIZATION authorization_id] |
 AUTHORIZATION authorization_id }

 [{ create_table_definition |
 create_view_definition |
 grant_definition } ...]

CREATE TABLE - Create base table

CREATE TABLE table
 ({ column_definition |
 [CONSTRAINT integrity_constraint_name] table_constraint },...)

 [USING SPACE space]

CREATE TEMPORARY VIEW - Declare temporary view

```
CREATE TEMPORARY VIEW temp_view_name [ ( { column },... ) ]  
    AS query_expression
```

CREATE VIEW - Create view

```
CREATE VIEW table [ ( column,... ) ]  
    AS query_expression  
    [ WITH CHECK OPTION ]
```

DECLARE - Declare cursor

```
DECLARE cursor [ { PERMANENT | TEMPORARY } ]  
    [ SCROLL ] [ PREFETCH n ] CURSOR [ FOR cursor_description ]
```

```
    cursor_description ::=  
        query_expression  
        [ ORDER BY { { column | column(pos_no) | column_number }  
        [ { ASCENDING | DESCENDING } ] },... ]  
        [ FOR UPDATE [ OF { column },... ] ]
```

```
n ::= unsigned_integer
```

```
pos_no ::= unsigned_integer
```

```
column_no ::= unsigned_integer
```

DELETE - Delete rows

```
DELETE FROM table [ WHERE { condition | CURRENT OF cursor } ]
```

DROP CURSOR - Release cursor description

```
DROP { CURSOR cursor | CURSORS }
```

DROP SCHEMA - Delete schema

```
DROP SCHEMA [ catalog . ] schema RESTRICT
```

DROP TABLE - Delete base table

```
DROP TABLE table RESTRICT
```

DROP TEMPORARY VIEW - Delete temporary view

```
DROP TEMPORARY { VIEW table | VIEWS }
```

DROP VIEW - Delete view

```
DROP VIEW table RESTRICT
```

FETCH - Position cursor and read row

```
FETCH [ { NEXT | PRIOR | FIRST | LAST | RELATIVE n | ABSOLUTE n } ]  
      [ FROM ] cursor
```

```
[ INTO { variable },... ]
```

```
n ::= { [ { + | - } ] unsigned_integer | variable }
```

GRANT - Grant privileges

GRANT format for table and column privileges:

```
GRANT { ALL PRIVILEGES | { table_and_column_privilege },... }
```

```
ON [ TABLE ] table
```

```
TO { PUBLIC | { authorization_id },... }
```

```
[ WITH GRANT OPTION ]
```

```
table_and_column_privilege ::= { SELECT | DELETE | INSERT |  
                                UPDATE [ ( { column },... ) ] |  
                                REFERENCES [ ( { column },... ) ] }
```

GRANT format for special privileges:

```
GRANT { ALL SPECIAL PRIVILEGES | CREATE SCHEMA }  
  
ON CATALOG catalog  
  
TO { PUBLIC | { authorization_id },... }  
  
[ WITH GRANT OPTION ]
```

INSERT - Insert rows in table

```
INSERT INTO table  
  
  { [ ( { column | column(pos_no) | column(min-max) },... ) ]  
  
    { VALUES ( { sql_expression | DEFAULT | NULL | * },... ) |  
      VALUES { sql_expression | DEFAULT | NULL | * } |  
      query_expression } |  
  
  DEFAULT VALUES }  
  
[ RETURN INTO variable ]
```

OPEN - Open cursor

```
OPEN cursor
```

PERMIT - Specify user identification for old style

```
PERMIT SCHEMA = { table | variable } [ PASSWORD = value ]
```

PRAGMA - Declare pragma clauses

```
PRAGMA literal
```

```
literal ::= '{ pragma_clause },...'
```

```
pragma_clause ::= { PREFETCH n |  
                  EXPLAIN INTO file |  
                  IGNORE INDEX index_name |  
                  OPTIMIZATION LEVEL n |  
                  SIMPLIFICATION { ON | OFF } |  
                  ISOLATION LEVEL  
                    { READ UNCOMMITTED |  
                      READ COMMITTED |  
                      REPEATABLE READ |  
                      SERIALIZABLE } |  
                  DATA TYPE OLDEST |  
                  CHECK { ON | OFF } }
```

RESTORE - Restore cursor

```
RESTORE cursor
```

REVOKE - Revoke privileges

REVOKE format for table and column privileges:

```
REVOKE { ALL PRIVILEGES | { table_and_column_privilege },... }  
ON [ TABLE ] table  
FROM { PUBLIC | { authorization_id },... } RESTRICT  
  
table_and_column_privilege ::= { SELECT |  
                                DELETE |  
                                INSERT |  
                                UPDATE [ ( { column },... ) ] |  
                                REFERENCES [ ( { column },... ) ] }
```

REVOKE format for special privileges:

```
REVOKE { ALL SPECIAL PRIVILEGES | CREATE SCHEMA }  
ON CATALOG catalog  
FROM { PUBLIC | { authorization_id },... } RESTRICT
```

ROLLBACK WORK - Roll back transaction

```
ROLLBACK [ WORK ] [ WITH RESET ]
```

SELECT - Read individual rows

```
SELECT [ { ALL | DISTINCT } ] select_list
    [ INTO { variable },... ]
    FROM table_specification,...
    [ WHERE condition ]
    [ GROUP BY column,... ]
    [ HAVING condition ]
```

SET CATALOG - Set default database name

```
SET CATALOG default_catalog

    default_catalog::= { alphanumeric_literal | variable }
```

SET SCHEMA - Set default schema name

```
SET SCHEMA default_schema

    default_schema::= { alphanumeric_literal | variable }
```

SET SESSION AUTHORIZATION - Define authorization identifier

```
SET SESSION AUTHORIZATION new_authorization_id

    new_authorization_id ::= { alphanumeric_literal | variable }
```

SET TRANSACTION - Define transaction attributes

```
SET TRANSACTION { level [ [ , ] transaction_mode ] |
                  transaction_mode [ [ , ] level ] }

level ::= { ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED |
                             REPEATABLE READ | SERIALIZABLE } |
           CONSISTENCY LEVEL consistency_level }

transaction_mode ::= { READ ONLY | READ WRITE }
```

STORE - Save cursor position

```
STORE cursor
```

UPDATE - Update column values

```
UPDATE table SET {  
    { column | column(pos_no) | column(min-max) }  
    = { sql_expression | DEFAULT | NULL }  
    },...  
[ WHERE { condition | CURRENT OF cursor } ]
```

WHENEVER - Define error handling

```
WHENEVER &DML_STATE [ IN ( status,... )  
    { CONTINUE | CALL subprog_name | BREAK }
```

5.2 Metavariables

query_expression

```
query_expression ::= { select_expression | expression | ( query_expression ) }  
                  [ UNION [ ALL ] query_expression ]
```

condition

```
condition ::= { [ condition AND ] { [ NOT ] { predicate | (condition) } } |  
              condition OR { [ NOT ] { { predicate | (condition) } } }
```

join_expression

```
join_expression ::=  
  { table_specification [ { INNER | { LEFT | RIGHT | FULL } [ OUTER ] } ]  
    JOIN table_specification ON condition |  
    ( join_expression ) }
```

literal

```
literal ::=
  { char_literal | num_literal | date_time_literal }
```

```
char_literal ::= 'string'
string ::= [ character ] ...
```

```
num_literal ::= { integer | fixed_point_number | floating_point_number | $PI }
integer ::= [ { + | - } ] unsigned_integer
fixed_point_number ::= [ { + | - } ] unsigned_integer [ .unsigned_integer ]
floating_point_number ::= fixed_point_number E [ { + | - } ] unsigned_integer
unsigned_integer ::= digit...
```

```
date_time_literal ::=
  { DATE (year-month-day) |
    TIME (hour:minute:second [.fraction_of_second ] ) |
    TIMESTAMP (year-month-day hour:minute:second [.fraction_of_second ] ) }
```

set_function

```
set_function ::= { { AVG | COUNT | MAX | MIN | SUM }
                  ( [ { ALL | DISTINCT } ] sql_expression ) |
                  COUNT(*) }
```

predicate

predicate::=

```
{ sql_expression { = | < | > | <= | >= | <> } sql_expression |
  sql_expression { = | < | > | <= | >= | <> } { ANY | SOME | ALL }
  subquery |
  sql_expression [NOT] BETWEEN sql_expression AND sql_expression |
  sql_expression [NOT] IN { subquery | (sql_expression,sql_expression,...) } |
  [ table . ] { column | column(pos_no) | column(min-max) } [ NOT ] LIKE
  patter [ ESCAPE character ] |
  [ table . ] { column | column(pos_no) | column(min-max) }
  IS [ NOT ] NULL |
  EXISTS subquery }
```

select_expression

select_expression::=

```
SELECT [ { ALL | DISTINCT } ] select_list
  FROM table_specification,...
  [ WHERE condition ]
  [ GROUP BY column,... ]
  [ HAVING condition ]

select_list ::= { * | { table.* | sql_expression [ [ AS ] column ] },... }
```

column_constraint

column_constraint ::=

```
{ NOT NULL |
  UNIQUE |
  PRIMARY KEY |
  CHECK ( condition ) |
  REFERENCES table [ ( column ) ] }
```

column_definition

column_definition ::=

```
column [ ( dimension ) ] { basic_data_type | FLOAT ( precision ) }
                               [ default ]
```

```
[ [ CONSTRAINT integrity_constraint_name ] column_constraint ... ]
```

dimension ::= unsigned_integer

```
default ::= DEFAULT { literal |
                     CURRENT DATE |
                     CURRENT TIME |
                     CURRENT TIMESTAMP |
                     [ CURRENT ] USER |
                     SYSTEM USER |
                     NULL }
```

sql_expression

```
sql_expression ::=
    { value |
      [ table. ] { column | column(pos_no) | column(min-max) } |
      { + | - } sql_expression |
      sql_expression { * | / | + | - | || } sql_expression |
      ( sql_expression ) |
      subquery |
      set_function |
      time_function |
      [ CURRENT ] USER | SYSTEM USER }
```

```
pos_no ::= unsigned_integer
```

```
min ::= unsigned_integer
```

```
max ::= unsigned_integer
```

table_specification

```
table_specification ::= { table [ [ AS ] correlation_name [ (column,...) ] ] |
  subquery [ AS ] correlation_name [ (column,...) ] |
  join_expression }
```

```
table ::= { [ [ catalog. ] unqual_schema_name. ] unqual_base_table_name |
  [ [ catalog. ] unqual_schema_name. ] unqual_view_name |
  temp_view_name }
```

table_constraint

```
table_constraint ::=
  { UNIQUE ( { column,... } ) |
    PRIMARY KEY ( { column,... } ) |
    FOREIGN KEY ( { column,... } ) REFERENCES table [ ( { column },... ) ] |
    CHECK ( condition ) }
```

subquery

```
subquery ::= ( query_expression )
```

variable

```
variable ::= { &varname1 [ suffix ] |
              &varname2 { ( index1, index2 ) |
                          ( index1, range2 ) |
                          ( range1, index2 ) }

suffix ::= { group_component | index_component }

group_component ::= . { * | component [ suffix ] }

index_component ::= { ( { index | range } ) }

index ::= unsigned_integer

range ::= index1 - index2 }
```

value

value ::= { literal | variable | aggregate }

aggregate ::= < { value | NULL }, ... >

time_function

time_function ::= { CURRENT DATE | CURRENT TIME | CURRENT TIMESTAMP }

6 Error messages

This chapter provides you with a description of

- the mapping of the SQLCODEs from SESAM V2 to the &DML_STATE of DRIVE/WINDOWS. This mapping is compatible with SESAM V1 or DRIVE/WINDOWS V1.1, and DRIVE V6.x
- the SQLSTATE classes of SESAM V2 that are relevant to DRIVE/WINDOWS

Refer to the WHENEVER statement in section “Pragmas” on page 62 and the “DRIVE Programming Language” manual [2], section 4.2, "Error recovery, end criteria" for more detailed information on error handling.

6.1 Mapping the SESAM SQLCODEs to &DML_STATE

In this section the SQLCODES of SESAM V2 are mapped to the &DML_STATE of DRIVE/WINDOWS (upward compatibility), and the associated SQLSTATES are listed.

SQLCODEs from the language definition

Meaning	SQLCODE	&DML_STATE	SQLSTATES
table-end-reached	100	TABLE END	02000
warning	50	OK	01xxx
dirty-read	10	DIRTY READ	01SA1
sql-ok	0	OK	00000
access-right-conflict	-22	SQL ERROR	42SQC
syntax-error	-110	<drive sys error>	42xxx (36 possible SQLSTATES)
schema-name-ambiguous	-118	SQL ERROR	42SND, 42SNN
table-not-primitive	-121	SQL ERROR	42S01, 42SQJ
table-not-defined	-127	SQL ERROR	42SH3, 42SNI, 42SQ6, 42SQQ

Meaning	SQLCODE	&DML_STATE	SQLSTATEs
table-name-ambiguous	-128	SQL ERROR	42SA2, 42SA4, 42SQI
col-or-comp-error	-131	SQL ERROR	42SOG, 42SP3, 42SR6
col-or-comp-spec-error	-135	SQL ERROR	42SAK, 42SAS, 42SO2, 42SQ2, 42SQ4, 42SQ5
col-or-comp-not-defined	-137	SQL ERROR	42SNF, 42SNG, 42SNM, 42S00
col-or-comp-ambiguous	-138	SQL ERROR	42SA1, 42SI8, 42SI9, 42SN5
cursor-already-closed	-141	SQL ERROR	24SA2
cursor-already-open	-142	SQL ERROR	24SA1, 24SA6
cursor-not-positioned	-143	SQL ERROR	24SA3
cursor-position-error	-144	CURSOR SQL ERROR	24SA5
cursor-not-defined	-147	SQL ERROR	34SA1, 42SF1
cursor-name-ambiguous	-148	SQL ERROR	42SF0
null-error	-210	SQL ERROR	23SA3, 23SA4
unique-error	-220	SQL ERROR	23SA2, 23SA5
no-indicator-variable	-310	SQL ERROR	22002
more-than-one-hit	-320	SQL ERROR	21000
value-list-error	-330	SQL ERROR	42SAT, 42SOC, 42SQH
value-error	-335	SQL ERROR	22023, 22SA2, 42SC9
set-function-not-allowed	-338	SQL ERROR	42SBR, 42SNH, 42SNO
value-overflow	-340	SQL ERROR	22001, 22003, 22012
type-error	-345	SQL ERROR	07SA6, 22019, 42SAX, 42SAY, 42SBO, 42SBX, 42SBY, 42SN2, 42S0Y
type-mismatch	-350	SQL ERROR	22005, 42SAW, 42SBS, 42SQ7, 42SQ9, 42SR1
argument-error	-365	SQL ERROR	42SAA
like-error	-370	SQL ERROR	22025, 42SAL, 42SAM, 42SBP
default-value-not-allowed	-372	SQL ERROR	22SA3
search-cond-error	-380	SQL ERROR	42SBU

Meaning	SQLCODE	&DML_STATE	SQLSTATEs
view-column-list-missing	-390	SQL ERROR	42SOF, 42SR3
group-restriction-error	-420	SQL ERROR	42SNK
into-clause-error	-440	SQL ERROR	42SCP, 42SQT
error-in-sql-stmt-processing	-550	SQL ERROR	42xxx (well over 200 possible SQLSTATEs)
error-in-dml-stmt-processing	-560	SQL ERROR	22018, 23SA0, 23SA1, 28000, 3D000, 3F000, 42SH4...42SH8, 42SNL, 44000
stmt-not-allowed	-600	SQL ERROR	24SA4, 25SA2, 25SA4
fetch-orientation-error	-630	SQL ERROR	42SF2
syntax-value-violated	-650	SQL ERROR	22020, 22021, 42SA8, 42SAB, 42SAD, 42SAE, 42SAH, 42SBW, 42SF3, 42SF4, 42SL0...42SL6, 42SM1...42SM4, 42SN3, 42SOE, 42SOP, 42SR9

System-specific SQLCODEs

<session cancelled> indicates the internal &DML_STATE entry 'SESSION CANCELLED'.

<db not available> indicates the internal &DML_STATE entry 'DB NOT AVAILABLE'.

<drive sys error> indicates a DRIVE/WINDOWS system error. In this case, DRIVE/WINDOWS issues error message DRI0078 and generates diagnostic documentation.

Temporary obstacles

Meaning	SQLCODE	&DML_STATE	SQLSTATEs
temp-access-restriction	-700	TEMP SYS ERROR	81SC9
stmt-cancelled	-701	TEMP SYS ERROR	81SA2, 81SD2
sort-option-restriction	-702	TEMP SYS ERROR	81SA3, 81SS0...81SS5, 91SCF, 91SCG
temp-system-limit	-710	TEMP SYS ERROR	81SC7, 91SA0, 91SA2, 91SA4, 91SA6, 91SA8, 91SA9, 91SAA, 91SAB, 91SAC, 91SAG, 91SAJ, 91SAL, 91SC5, 91SC6, 91SC7, 91SCB, 91SCC, 91SCD, 91SCE, 91SCH, 91SCJ, 91SCK, 91SS0...91SS7

Incompatibilities within the SQL schema

Meaning	SQLCODE	&DML_STATE	SQLSTATEs
db-open-error	-775	ACC SYS ERROR	42SC7, 42SN7, 55SA1, 55SAA, 55SAE, 81SA4, 81SA5

Administrator intervention required

Meaning	SQLCODE	&DML_STATE	SQLSTATEs
session-unknown	-740	<session cancelled>	81SP2
access-restriction	-800	ADMIN SYS ERROR	81SA6, 81SB2, 81SCA
update-restriction	-810	ADMIN SYS ERROR	25SA1, 81SB0, 81SB1
system-abnormally-down	-830	<db not available>	81SB5, 81SC3
component-not-available	-840	ACC SYS ERROR	81SC6
transaction-start-not-allowed	-850	ADMIN SYS ERROR	81SA1, 81SP3
configuration-file-error	-860	ACC SYS ERROR	81SB4, 81SC0, 81SC1

Programming errors in the DB system or at the DB interface

Meaning	SQLCODE	&DML_STATE	SQLSTATEs
ta-stmt-not-allowd	-654	<drive sys error>	42SH2
program-error	-900	<session cancelled>	42SC1, 55SA7, 81SC4, 81SC5
integrity-error	-910	ACC SYS ERROR	81SA7, 81SA8, 81SA9, 91SAD
system-limit-exceeded	-920	ACC SYS ERROR	91SA1, 91SA7, 91SAE, 91SAF, 91SAH, 91SAI, 91SAK, 91SB1...91SB4, 91SC0...91SC4, 91SC8, 91SC9, 91SCA, 91SCI, 91SR0, 91SR2, 91SR3, 91SU0...91SU4
not-implemented	-940	SQL ERROR	42SS0
illegal-order-of-state- ments	-990	SQL ERROR	25SA3, 25SA5
representation-error	-990	<drive sys error>	-
output-too-long	-990	<drive sys error>	-

Internal rollback for the ("CANCEL WORK")

If the transaction is rolled back internally by the database system, an internal SQLCODE between -1000 and -2000 is generated by adding -1000 to the actual SQLCODE. In this case, DRIVE/WINDOWS generates the internal &DML_STATE entry 'TA CANCELLED' and afterwards behaves as it does for an external ROLLBACK WORK WITH RESET. The SQLCODE -1830 is an exception for which DRIVE/WINDOWS generates the internal &DML_STATE entry 'DB NOT AVAILABLE'.

Error for dynamic SQL statements (EXECUTE)

Errors in dynamic SQL statements (SQLSTATE class 07) normally result in DRIVE system errors because DRIVE/WINDOWS cannot offer these statements at the user interface, but rather generates them internally within the framework of the DRIVE EXECUTE statement.

Meaning	SQLCODE	&DML_STATE	SQLSTATEs
undefined-descriptor-area-item	100	<drive sys error>	02SA1
error-in-dynamic-stmt	-650	<drive sys error>	-
invalid-stmt-identifier	-651	<drive sys error>	-
exec-or-open-not-possible	-652	<drive sys error>	-
error-in-using-clause	-653	<drive sys error>	-
descriptor-area-name-invalid	-661	<drive sys error>	-
allocated-descriptor-area	-662	<drive sys error>	-
not-allocated-descriptor-area	-663	<drive sys error>	-
error-in-descriptor-area-item	-664	<drive sys error>	-
configuration-error	-690	<drive sys error>	-
session-limit-exceeded	-950	<drive sys error>	-
too-many-host-variables	-960	<drive sys error>	-
error-in-descriptor-area-size	-965	LIMIT REACHED	91SA3, 91SA5

6.2 SQLSTATE classes

The mapping of SQLSTATES to &DML_STATE entries is described under the WHENEVER statement. The following SQLSTATE classes of SESAM/SQL V2 are relevant for DRIVE/WINDOWS:

&SQL_STATE	Meaning
00xxx	execution successful
01xxx	warning
02xxx	no data
21xxx	set limit violation
22xxx	data error
23xxx	integrity constraint violation
24xxx	illegal cursor status or cursor operation
25xxx	illegal transaction status
26xxx	illegal or invalid SQL statement name
28xxx	illegal authorization identifier
2Dxxx	illegal means of terminating transaction
34xxx	illegal or invalid cursor name
3Dxxx	illegal catalog name
3Fxxx	illegal schema name
40xxx	transaction rolled back
42xxx	syntax error or no access permission: this class contains over 300 subclasses
44xxx	"CHECK OPTION" violation
55xxx	BS2000 error messages
56xxx	BS2000 restrictions
81xxx	environment error
91xxx	insufficient resources
95xxx	errored transaction status

The SESAM/SQL-Server "Messages" manual [24] contains a list of all SQLSTATES and the associated SQLCODEs.

Related publications

[1] **DRIVE/WINDOWS V1.1 (BS2000)**

Programming System

User Guide

Target group

Application programmers

Contents

- Introduction to the programming system DRIVE/WINDOWS
- Explanation of the functions available in interactive mode
- Installation
- DRIVE/WINDOWS generation and administration

[2] **DRIVE/WINDOWS (BS2000)**

Programming Language

Reference Guide

Target group

Application programmers

Contents

Description of program creation including alpha screen forms, as well as the use fo DRIVE list forms and the report generator.

[3] **DRIVE/WINDOWS (BS2000)**

System Directory of DRIVE Statements

Reference Manual

Target group

Applications programmers

Contents

Syntax and range of functions of all DRIVE statements. DRIVE messages and keywords.

- [4] **DRIVE/WINDOWS (SINIX)**
Directory of DRIVE SQL Statements for SESAM V1.x
Reference Manual
- Target group*
Application programmers
- Contents*
A concise description of the syntax and scope of functions of all the DRIVE SQL statements for SESAM V1.x.
- [5] **DRIVE/WINDOWS (SINIX)**
Directory of DRIVE SQL Statements for SESAM V2.x
Reference Manual
- Target group*
Application programmers
- Contents*
A concise description of the syntax and scope of functions of all the DRIVE SQL statements for SESAM V2.x.
- [6] **DRIVE/WINDOWS (SINIX)**
Directory of DRIVE SQL Statements for UDS
Reference Manual
- Target group*
Application programmers
- Contents*
A concise description of the syntax and scope of functions of all the DRIVE SQL statements for UDS.
- [7] **DRIVE/WINDOWS V2.0 (MS-Windows)**
Software Production Environment (SPE)
User Guide
- Target group*
Application programmers.
- Contents*
The manual describes the functions of the software production environment (desktop), how to prepare DRIVE/WINDOWS for use, remote access to BS2000 and SINIX databases and client/server applications.

- [8] **DRIVE/WINDOWS V2.0** (MS-Windows)
Programming Language
Reference Manual
- Target group*
Application programmers.
- Contents*
The manual describes the creation of programs, including window and client/server applications.
- [9] **DRIVE/WINDOWS V2.0** (MS-Windows)
System Directory
Reference Manual
- Target group*
Application programmers.
- Contents*
The manual describes the syntax and functions of all statements, messages and keywords of DRIVE/WINDOWS.
- [10] **DRIVE/WINDOWS** (SINIX)
Software Production Environment (SPE)
User Guide
- Target group*
Application programmers
- Contents*
The functions available in the software production environment (desktop) and in expert mode. Setting up DRIVE/WINDOWS, including remote access to BS2000 databases and generating applications for BS2000.
- [11] **DRIVE/WINDOWS** (SINIX)
Programming Language
Reference Manual
- Target group*
Application programmers
- Contents*
The creation of programs, including graphical and alpha screen forms, as well as list forms using DRIVE and the report generator.

- [12] **DRIVE/WINDOWS** (SINIX)
System Directory
Reference Manual
- Target group*
Application programmers
- Contents*
The syntax and scope of functions of all DRIVE statements, as well as all DRIVE messages and keywords.
- [13] **DRIVE/WINDOWS** (SINIX)
Directory of DRIVE SQL Statements for INFORMIX
Reference Manual
- Target group*
Application programmers
- Contents*
A concise description of the syntax and scope of functions of all the DRIVE SQL statements for INFORMIX.
- [14] **DRIVE V5.1** (BS2000)
Part 1: User's Guide
- Target group*
- Users in non-dp departments
 - Applications programmers
- Contents*
- General overview of the DRIVE system in old style
 - Description of the DRICE components
 - Introduction to DRIVE application using worked examples
 - DRIVE generation and administration in UTM operation
- [15] **DRIVE V5.1** (BS2000)
Part 2: System Directory
- Target group*
- Users in non-dp departments
 - Applications programmers
- Contents*
- Syntax and scope of functions of all DRIVE statements in old style
 - DRIVE messages and keywords

- [16] **DRIVE/WINDOWS-COMP** (BS2000)
User Guide
- Contents*
The differences concerning the DRIVE V6.0 language, and the compilation process. Generating and starting TIAM and UTM applications with compiled DRIVE objects, with special consideration of mixed version operation.
- [17] **SQL for SESAM/SQL**
Language Reference Manual
- Target group*
Programmers who want to access SESAM databases using SQL statements.
- Contents*
SQL statements available for accessing SESAM databases.
- [18] **SESAM/SQL-Server** (BS2000/OSD)
SQL Reference Manual Part 1: SQL Statements
User Guide
- Target group*
The manual is intended for all users who wish to process an SESAM/SQL database by means of SESAM/SQL statements.
- Contents*
The manual describes how to embed SQL statements in COBOL, and the SQL language constructs. The entire set of SQL statements is listed in an alphabetical directory.
- [19] **SESAM/SQL-Server** (BS2000/OSD)
SQL Reference Manual Part 2: Utilities
User Guide
- Target group*
The manual is intended for all users responsible for SESAM/SQL database administration.
- Contents*
An alphabetical directory of all utility statements, i.e. statements in SQL syntax implementing the SESAM/SQL utility functions.
- [20] **SESAM/SQL-Server** (BS2000/OSD)
Core Manual
User Guide
- Target group*
The manual is intended for all users and to anyone seeking information on SESAM/SQL.
- Contents*
The manual gives an overview of the database system. It describes the basic concepts. It is the foundation for understanding the other SESAM/SQL manuals.

- [21] **SESAM/SQL-Server** (BS2000/OSD)
Utility Monitor
User Guide
- Target group*
The manual is intended for SESAM/SQL-Server database and system administrators.
- Contents*
The manual describes the utility monitor. The utility monitor can be used to administer the database and the system. One aspect covered is its interactive menu interface.
- [22] **SESAM/SQL-Server** (BS2000/OSD)
Migrating SESAM Databases and Applications to SESAM/SQL-Server
User Guide
- Target group*
Users of SESAM/SQL-Server.
- Contents*
This manual gives an overview of the new concepts and functions. Its primary subject is, however, the difference between the previous and the new SESAM/SQL version(s). It contains all the information a user may require to migrate to SESAM/SQL-Server V2.0.
- [23] **SESAM/SQL-Server** (BS2000/OSD)
CALL DML Applications
User Guide
- Target group*
SESAM application programmers
- Contents*
- CALL DML statements for processing SESAM databases using application programs
 - Transaction mode with UTM and DCAM
 - Utility routines SEDI61 and SEDI63 for data retrieval and direct updating
 - Notes on using both CALL DML and SQL modes
- [24] **SESAM/SQL-Server** (BS2000/OSD)
Messages
User Guide
- Target group*
All users of SESAM/SQL.
- Contents*
All SESAM/SQL messages, sorted by message number.

- [25] **SQL for UDS/SQL**
Language Reference Manual
- Target group*
Programmers who want to access UDS databases using SQL statements.
- Contents*
SQL statements available for accessing UDS databases.
- [26] **UDS/SQL (BS2000)**
Administration and Operation
User Guide
- Target group*
Database administrators
- Contents*
All features comprising the management and operation of the database, such as database saving, processing, restructuring, as well as outputting database information and checking the consistency of the database.
- Applications*
Database operation by the database administrator
- [27] **UDS/SQL (BS2000)**
Creation and Restructuring
User Guide
- Target group*
Database administrators
- Contents*
- Overview of the files required by UDS
 - UDS utility routines required for database creation
 - Utility routines required for restructuring
- Applications*
Database creation by the database administrator
- [28] **IFG for FHS (TRANSDATA)**
User Guide
- Target group*
Terminal users, application engineers and programmers
- Contents*
The Interactive Format Generator (IFG) is a system that permits simple, user-friendly generation and management of formats at a terminal. In conjunction with FHS, these formats can be used on the host computer. This user guide describes how formats are generated, modified and managed, plus also the new functions of IFG V8.1.

- [29] **FHS (TRANSDATA)**
User Guide
Target group
Programmers
Contents
Program interfaces of FHS for TIAM, DCAM and UTM applications. Generation, application and management of formats.
- [30] **UTM (TRANSDATA, BS2000)**
Generating and Administering Applications
User Guide
Target group
– System administrators
– UTM administrators
Contents
– Creation, generation and operation of UTM applications
– Working with UTM messages and error codes
Applications
BS2000 transaction processing
- [31] **UTM (TRANSDATA)**
Programming Applications
User's Guide
Target group
Programmers of UTM applications
Contents
– Language-independent description of the KDCS program interface
– Structure of UTM programs
– KDCS calls
– Testing UTM applications
– All the information required by programmers of UTM applications
Applications
BS2000 transaction processing
- [32] **UTM(SINIX)**
Formatting System
Target group
UTM(SINIX) users who wish to use formats, C programmers and COBOL programmers
Contents
How to use the FORMANT format handler in UTM(SINIX) program units, create formats, convert formats from BS2000 to/from SINIX.

[33] EDT V16.5A (BS2000/OSD)

Statements
User Guide

Target group

EDT newcomers and EDT users

Contents

Processing of SAM and ISAM files and elements from program libraries and POSIX files.

[34] LMS (BS2000)

ISP Format
Reference Manual

Target group

BS2000 users

Contents

Description of the LMS statements in ISP format for creating and managing PLAM libraries and the members these contain.

Frequent applications are illustrated by means of examples.

[35] BS2000/OSD-BC

Commands, Volume1 - 3

Target group

The manual addresses both nonprivileged BS2000/OSD users and system support.

Contents

This manual contains BS2000/OSD commands (basic configuration and selected products) with the functionality for all privileges. The introduction provides information on command input.

[36] BS2000/OSD-BC V2.0

System Installation
User Guide

Target group

BS2000/OSD system administration

Contents

This manual describes

- the generation of the hardware and software configuration with UGEN
- the following installation services:
 - disk organization with MPVS
 - program system SIR
 - volume installation with SIR
 - configuration update (CONFUPD)
 - utility routine IOCFCOPY

- [37] **BS2000/OSD-BC V2.0A**
DMS Introductory Guide
User Guide

Target group

The manual addresses both nonprivileged users and systems support.

Contents

The manual describes file processing in BS2000, focussing on:

- file and catalog management
- files and data media
- file and data protection
- OPEN, CLOSE and EOV processing
- DMS access methods (SAM, ISAM ...).

- [38] **BS2000**
Introductory Guide for System Users
User's Guide

Target group

BS2000 users

Contents

- Introduction to BS2000
- Description of the most frequent user commands
- Introduction to using the utility routines and software products EDT, SORT, ARCHIVE, TSOSLNK, LMS and PERCON
- Notes for the programmer

Applications

BS2000 interactive mode and batch mode

- [39] **FORMANT (SINIX)**
Reference Manual

Target group

- C programmers
- COBOL programmers
- Application designers

Contents

Formant is a mask control program for all SINIX systems. The manual contains:

- Introduction to FORMANT
- Description of FORMANTGEN
- Description of user interface
- Program interfaces in C and COBOL
- Programming examples

- [40] **OMNIS (TRANSDATA, BS2000)**
Administration and Programming
User Guide
- Target group*
- OMNIS administrators
 - Programmers
- Contents*
- Introduction to OMNIS administration, the OMNIS utility routines and the application interface for extending the OMNIS functionality
- Applications*
- Software development
 - Application scheduling
- [41] **DRIVE/WINDOWS-COMP (SINIX)**
Compiler
User Guide
- Target group*
- Applications programmers and system administrators
- Contents*
- Description of the compilation process using the DRIVE Compiler.
- [42] **INFORMIX-NET V4.0 (SINIX)**
INFORMIX-STAR V4.0 (SINIX)
User Guide
- Target group*
- INFORMIX users
 - System administrators
- Contents*
- This manual describes how to use the INFORMIX-NET and INFORMIX-STAR products. Using these products, INFORMIX applications can generate and process databases on foreign computers from local computers.
- [43] **DRIVE/WINDOWS V1.1**
(SINIX)
Supplement
User Guide
- Target group*
- Application programmers
- Contents*
- The manual contains the functional changes included in DRIVE/WINDOWS (SINIX) V1.1. If this supplement is to be used, the manuals of version 1.0 are also required.

- [44] **SESAM/SQL-Server** (BS2000/OSD)
Messages
User Guide
Target group
All users of SESAM/SQL.
Contents
All SESAM/SQL messages, sorted by message number.
- [45] **SESAM/SQL-Server** (BS2000/OSD)
Performance
User Guide
Target group
Experienced users of SESAM/SQL.
Contents
The manual covers how to recognize bottlenecks in the behavior of SESAM/SQL and how to remedy this behavior.
- [46] **SESAM/SQL** (BS2000)
Creation and Maintenance
User's Guide
Target group
Database administrators
Contents
– Creation and maintenance of SESAM databases using the database administration monitor SESASB
– Shadow database operation

Other publications

- [47] International Organization for Standardization (ISO):
Database Language SQL
ISO/IEC 9075:1992

Ordering manuals

The manuals listed above and the corresponding order numbers can be found in the Siemens Nixdorf List of Publications. New publications are described in the Druckschriften-Neuerscheinungen (New Publications).

You can arrange to have both of these sent to you regularly by having your name placed on the appropriate mailing list. Please apply to your local office, where you can also order the manuals.

Index

&DML_STATE IN (status) 153

. * 228

A

ABSOLUTE (clause) 103

ACC SYS ERROR 153

access permission

granting 11

ADD COLUMN (clause) 71

ADD CONSTRAINT (clause) 72

addition 219

ADD-SQL-CATALOG-LIST 25

ADMIN SYS ERROR 153

aggregate 230

ALL (clause)

AVG() 174

COUNT() 177

MAX() 179

MIN() 181

set function 173

SUM() 183

UNION (clause) 160

ALL (predicate) 191

select_expression 202

ALL PRIVILEGES (clause) 107, 129

ALL SPECIAL PRIVILEGES (clause) 109, 131

alphanumeric literal 4, 169

alphanumeric values

comparison 188

ALTER COLUMN (clause) 71

ALTER TABLE 70

AND (operator) 163

ANY (predicate) 190

argument

function 173

arithmetic average
 AVG() 174
arithmetic operators 5
AS (clause) 84, 86
AUTHORIZATION (clause) 79
AUTHORIZATION (parameterization) 36
authorization identifier 29
 =SQL user 10, 11, 15
 =SQL user name 29
 for SESAM database 36, 38
 ROLLBACK WORK 29
AVG() 174

B

base table
 create 81
 delete 99
 modify 70
blank 5
block mode
 PREFETCH (DECLARE statement) 90
 PREFETCH (pragma clause) 62, 64, 121

C

calculate
 predicate 185
calculate sum
 SUM() 183
CALL DML 15
 PERMIT 15
 table 73
cancel
 characters in literals 4
catalog 7
CATALOG (parameterization) 37
catalog setting
 in foreign environment 39
character
 comment 5
 enclose in quotes 4
charliteral 169
CHECK (clause) 212, 226
CHECK (pragma clause) 126
CLOSE 74

- close
 - cursor 74
- column 8
 - add 70
 - for CALL DML table 126
 - update 70
 - update contents (UPDATE) 147
- column constraint 211
- column definition 213
- column number 92
- column privilege 106
- combine
 - query expressions 159
- comma 5
- comments 5
- COMMIT WORK 75
 - SET (database environment) 27
- compare
 - NULL value 188
- compare two values
 - predicate 187
- comparison
 - alphanumeric value 188
 - numeric value 188
 - operators 5
 - time value 189
 - with derived column 190
 - with NULL value 199
- comparison operation 188
- comparison operator 187
- comparison rules 188
- compilation
 - authorization identifier 29
 - controlling 38
 - database contact 28
 - transaction 28
- compiler options 38
 - static programs 28
- component 228
- concatenation 219
- concatenation operator 5
- condition 162
 - CHECK (clause) 162
 - GROUP clause 210

- HAVING (clause) 162
- ON (clause) 162
- operator 162
- precedence 164
- predicate 162
- WHERE (clause) 162, 207
- CONSISTENCY LEVEL (clause) 144
- constants 4
- constraint 8
- CONSTRAINT (clause) 81, 214
- CONTINUE (WHENEVER) 153
- control
 - compilation run 38
- correlation
 - name 223
 - table 202
- count elements
 - COUNT() 177
- count rows
 - COUNT(*) 176
- count table rows
 - COUNT(*) 176
- COUNT() 177
- CREATE CATALOG 47
- CREATE SCHEMA 49, 79
- CREATE SCHEMA privilege 109
- CREATE TABLE 81
- CREATE TEMPORARY VIEW 83
- CREATE USER 48
- CREATE VIEW 86
- CURRENT DATE 232
- current date
 - CURRENT DATE 232
- CURRENT OF (clause) 95, 149
- current row (FETCH) 102
- CURRENT TIME 232
- current time
 - CURRENT TIME 232
- current time stamp
 - CURRENT TIMESTAMP 232
- CURRENT TIMESTAMP 232
- cursor
 - close 74
 - declare 88

FOR UPDATE (DECLARE) 92
open 116
ORDER BY 91
query expression 91
restore 127
SCROLL 90
updatable 93
variable 91
cursor description 91
cursor position 102
 save (STORE) 146
CURSOR SQL ERROR 153
cursor statements for variable cursors 14

D

data type
 modify 71
 UNION (clause) 160
DATA TYPE (pragma clause) 126
database contact
 compilation 28
database environment
 parameter settings 26
 PARAMETER statement 36
 SET 26
DATE 171
date/time literal 4
 date_time_literal 171
DDL statements 14
DECLARE 88
DECLARE CURSOR statement
 PREFETCH clause 64
DECLARE VARIABLE ...LIKE 14
DEFAULT (clause) 113, 148, 214
default catalog
 SQL environment 15
default schema
 SQL environment 15
DEFAULT VALUES (clause) 114
define
 default value 72
 transaction mode (SET TRANSACTION) 142
 variable 228
DELETE 95

- delete
 - base table 99
 - row 95
 - schema 98
 - temporary view 100
 - view 101
- DELETE (clause) 107
- DELETE privilege 130
- delimiters 5
- derived column
 - data type for UNION 160
 - select 204
- derived row
 - group 208
 - select 207
- derived table 159
- determine largest value
 - MAX() 179
- determine lowest value
 - MIN() 181
- differences SESAM V1/V2
 - for DRIVE access 24
- DIRTY READ 153
- DISTINCT clause
 - AVG() 174
 - COUNT() 177
 - MAX() 179
 - MIN() 181
 - select_expression 202
 - set function 173
 - SUM() 183
- division 218
- DML statements 15
- dominant table 166
- DRIVE compilation
 - static program 15
- DRIVE extensions to SQL standard 13
- DRIVE operating modes
 - SESAM access 13
- DRIVE session/UTM conversation
 - SQL user 23
- DROP CONSTRAINT (clause) 72
- DROP DEFAULT (clause) 71
- DROP SCHEMA 98

DROP TABLE 99
DROP VIEW 101
duplicate
 COUNT() 177
DYNAMIC 36
dynamic
 new-style program 14
 SQL statement 13
dynamic program
 changing SQL user 16

E

elements
 count 177
error handling
 define (WHENEVER) 152
 define actions 152
 pragmas 66
ESCAPE (predicate) 197
escape character 196
evaluate
 precedence 185
evaluation
 select_expression 203
example
 table Abteilung 42
 table Mitarbeiter 43
 table Projekt 44
exception conditions for error exit 153
execution
 authorization identifier 29
existence query 201
EXISTS (predicate) 201
expression
 priority 221
 sql_expression 216

F

FIRST (clause) 103
FOR (clause) 91
FOR UPDATE
 cursor (DECLARE) 92
FOREIGN KEY clause 225
FOREIGN KEY constraint 9

FROM (clause)
 FETCH 104
 SELECT 206
FROM (predicate)
 select_expression 206
FROM PUBLIC (clause) 130
FULL OUTER (clause) 166
function
 AVG() 174
 COUNT() 177
 CURRENT DATE 232
 CURRENT TIME 232
 CURRENT TIMESTAMP 232
 MAX() 179
 MIN() 181
 set 173
 SUM() 183
 time 232
function argument 173

G

general
 new-style program 14
GRANT 11, 15, 106
GRANT authorization 108, 109
group 208
GROUP BY (clause) 208
grouping
 derived rows 208

H

hexadecimal literal 4

I

IGNORE INDEX (pragma clause) 124
IN (clause) (WHENEVER) 153
incompatibilities
 DRIVE access to SESAM V1/V2 24
INNER (clause) 166
INSERT 111
insert
 row 111
INSERT (clause) 107
INSERT privilege 130

- integrity constraint
 - add 70, 72
 - delete 70, 72
- internal access plan
 - output 62
- interpretation
 - PRAGMA statement 63
- interval literal 4
- INTO (clause) 104, 134
- IS NULL (predicate) 199
- ISOLATION LEVEL
 - SET TRANSACTION 143
- isolation level
 - for statement (pragma clause) 62, 65
 - set (SET TRANSACTION) 142
- ISOLATION LEVEL (pragma clause) 124

J

- join_expression 165

K

- keyword 3

L

- LAST (clause) 103
- LEFT OUTER (clause) 166
- LIMIT REACHED 153
- literal 168
 - alphanumeric 4, 169
 - date/time 4, 171
 - enclose characters in quotes 4
 - hexadecimal 4
 - interval 4
 - numeric 4, 170
- literals 4
- logical operator 162
 - AND 163
 - NOT 163
 - OR 163

M

- MAX() 179
- metadata
 - via SELECT on system table 40

- via SHOW 40
- via Utility Monitor 40
- metavariables 1, 4
- MIN() 181
- mixed mode
 - SESAM access 13
- modify
 - base table 70
 - data type 71
- multiplication 218
- multi-SQL-user program
 - TA profile for general programs 23

N

- name 3
 - partially qualified (variable) 228
 - with special characters 3
- name qualification
 - of tables and columns 8
- naming conventions 3
- new-style transaction
 - DML statements 15
- NEXT (clause) 103
- NOT (operator) 163
- NOT NULL clause 211
- notational conventions 5
- NULL (INSERT) 113
- NULL (UPDATE) 148
- NULL value
 - compare 188
 - expression 216
- numeric literal 4
 - numliteral 170
- numeric values
 - comparison 188
- numliteral
 - \$PI specification 170

O

- old- and new-style
 - SESAM access 13
 - transaction 15
- oldest-style table
 - extending (pragma clause) 62, 65

- old-style
 - SESAM access 13
- ON CATALOG (clause) 109, 131
- ON TABLE (clause) 108, 130
- OPEN 116
- open
 - cursor 116
- operand
 - expression 216
- operator
 - AND 163
 - comparison 187
 - condition 162
 - expression 216
 - logical 162
 - NOT 163
 - OR 163
 - predicate 185
- OPTIMIZATION LEVEL (pragma clause) 125
- optimizer
 - output access plan (pragma clause) 62
- optimizer access plan
 - influencing (pragma clause) 62
- OPTION statement 28, 38
- OR (operator) 163
- ORDER BY (clause) 91
- organization of a database
 - logical 7
- OUTER (clause) 166

P

- parameter settings
 - dynamic statements 27
 - programs 27
- PARAMETER statement 26, 36
- parameters
 - for dynamic programs 36
 - for static programs 36
- partially qualified name (variable) 228
- performance
 - improved 90, 91
- PERMIT 117
 - CALL DML 15
- physical organization

- of a database 9
- statements for 10
- placeholder
 - pattern comparison 196
- position
 - cursor 102
 - cursor (FETCH) 102
- pragma clause
 - CHECK 66, 126
 - DATA TYPE 65, 126
 - EXPLAIN
 - read SQL access plan (pragma clause) 64
 - IGNORE INDEX 65, 124
 - ISOLATION LEVEL 65, 124
 - OPTIMIZATION LEVEL 65, 125
 - PREFETCH 66, 121
 - SIMPLIFICATION 65
- PRAGMA statement 63, 119
 - interpreting 63
 - static/dynamic 63
- pragmas
 - application possibilities 62
 - error handling 66
- precedence
 - condition 164
- precompiled statements (SESAM) 15
- predicate 185
 - ALL 191
 - ANY 190
 - BETWEEN 192
 - calculate 185
 - compare two values 187
 - comparison with derived column 190
 - comparison with NULL value 199
 - condition 162
 - element query 194
 - evaluate 185
 - existence query 201
 - EXISTS 201
 - IN 194
 - IS NULL 199
 - LIKE 196
 - operator 185

- pattern comparison 196
- range query 192
- SOME 190
- PREFETCH
 - DECLARE CURSOR statement 64, 90
 - pragma clause 64
- PREFETCH (pragma clause) 121
- PRIMARY KEY (clause) 211, 225
- PRIMARY KEY constraint 9
- PRIOR (clause) 103
- priority
 - expression 221
- privilege
 - grant 106
 - revoke 129
- program communication
 - dynamic-dynamic 20
 - dynamic-static 22
 - general-general 22
 - new-/old-style 22
 - static-dynamic 21
 - static-static 19
- program compilation
 - controlling 38
- programming recommendations 23

Q

- query expressions
 - combine 159
- query_expression 159
 - updatable 160

R

- read
 - row 102
- READ COMMITTED (clause) 143
- READ ONLY (clause) 145
- READ UNCOMMITTED (clause) 143
- READ WRITE (clause) 145
- REFERENCES privilege 108, 130
- RELATIVE (clause) 104
- renaming
 - table 202
- REPEATABLE READ (clause) 143

RESTORE 127
restore
 cursor 127
RETURN INTO (clause) 114
REVOKE 129
RIGHT OUTER (clause) 166
roll back
 transaction 132
ROLLBACK WORK 132
 authorization identifier 29
 SET (database environment) 27
row
 count 176
 current (FETCH) 102
 delete (DELETE) 95
 insert 111
 insert (INSERT) 111
 read 102

S

sample database
 structure of 45
schema 8
 create 79
 delete 98
SCHEMA (compiler option) 39
SCHEMA (parameterization) 37
schema definition 39
schema setting
 in foreign environment 39
scroll cursor 90
SELECT 134
 INTO clause 134
 read individual rows 134
SELECT (clause) 107
SELECT privilege 129
SELECT/FROM 206
select_expression
 evaluation 203
select_list (SELECT list) 204
 see select_expression and SELECT statement 157
SERIALIZABLE (clause) 144
SESAM access
 DRIVE operating modes 13

- old- and new-style 13
- SET
 - CATALOG/SCHEMA/AUTHORIZATION 26
- set
 - isolation level 142
- SET (database environment)
 - COMMIT WORK 27
 - ROLLBACK WORK 27
- SET CATALOG 136
- set function
 - AVG() 174
 - COUNT() 177
 - MAX() 179
 - MIN() 181
 - set_function 173
 - SUM() 183
- SET SCHEMA 138
- SET SESSION AUTHORIZATION 140
- SET TRANSACTION 142
- set_function (set function) 173
- SHOW
 - output metadata 40
- simple variable 228
- single-SQL-user program 23
- single-SQL-user session 23
- SOME (predicate) 190
- space 9
- special characters
 - in names 3
- special privilege 109
 - grant 106
 - revoke 131
- specify
 - table 206
- SQL environment 15
 - compiler listing 16
 - debugging mode 17
 - dynamic programs 16
 - static programs 15
- SQL ERROR 153
- SQL objects
 - access to 10
 - logical 7
- SQL standard

- DRIVE extensions 13
- entry/intermediate/full level 12
- SQL user 25
 - =authorization identifier 10, 11, 15
 - access permission 15
 - check during data access 25
 - DRIVE session/UTM conversation 23
- SQL user name
 - =authorization identifier 29
- sql_expression
 - addition 219
 - assignment 216
 - calculating 216
 - column 217
 - column selection 216
 - concatenation 219
 - division 218
 - expression 216
 - multiplication 218
 - NULL value 216
 - operand 216
 - operator 216
 - predicate 216
 - set function 220
 - subquery 220
 - subtraction 219
 - value 216
- SQLERROR (WHENEVER) 152
- SQLSTATE 1
- SSL statements 9, 14
- statements
 - for session control 14
 - for transaction management 14
- static
 - new-style program 14, 15
 - SQL statement 13
- storage group 9
- STORE 146
- store
 - cursor position 146
- structure
 - of sample database 45
 - of statements 2
- subquery 227

- expression 227
- FROM (clause) 227
- join expression 227
- predicate 227
- subtraction 219
- SUM() 183
- supported interfaces for SESAM 14

T

- table 8
 - correlation name 202, 223
 - dominant 166
 - renaming 202
 - specify 206
- TABLE (clause) 81
- TABLE END 153
- table privilege 106
- table_constraint 225
- table_specification 222
- TEMP SYS ERROR 153
- temporary view
 - declare 83
- terminate
 - transaction 75
- TIME 171
- time function
 - CURRENT DATE 232
 - CURRENT TIME 232
 - CURRENT TIMESTAMP 232
- time value
 - comparison 189
 - separators 172
- time_function 232
- TIMESTAMP 171
- TO PUBLIC (clause) 108, 109
- TOO MANY CURSORS 153
- transaction
 - old- and new-style 15
 - roll back (ROLLBACK WORK) 132
 - terminate 75
- transaction mode
 - specify (SET TRANSACTION) 142
- transaction profile
 - static program 15, 16

truth value

condition 162

predicate 185

U

UDL statements 14

UNION (clause) 159

UNIQUE (clause) 211, 225

UNIQUE constraint 9

updatability

query_expression 160

updatable

cursor 93

view 87

UPDATE 147

update

column value 147

UPDATE privilege 107, 130

user setting

in foreign environment 39

USING SPACE (clause) 82

utility statements 14

V

value

specify 230

variable 3

VALUES (clause) 112

variable 228

cursor 91

defining 228

simple 228

value 3

view

create 86

declare 84

delete 101

temporary - delete 100

updatable 87

W

WHENEVER 152

WHERE (clause) 95, 148

WITH CHECK OPTION (clause) 87

WITH GRANT OPTION (clause) 108, 109

Contents

1	Preface	1
1.1	Summary of contents	1
1.2	Structure of DRIVE SQL statements	2
1.3	Notational conventions	5
2	Working with SESAM/SQL V2	7
2.1	Organization of a SESAM V2 database using an example	7
2.1.1	Terminology for logical organization	7
2.1.2	Terminology for physical organization	9
2.2	Database structure and migration of SESAM V1 tables	10
2.2.1	DRIVE requirements for SESAM migration	12
2.3	DRIVE program access to SESAM	12
2.3.1	SQL language resources in new style	14
2.3.2	Static programs	15
2.3.3	Dynamic programs	16
2.3.4	Program communication	17
2.3.5	Programming recommendations	23
2.3.6	Incompatibilities	24
2.4	Defining the database environment	25
2.4.1	SET CATALOG/SCHEMA/SESSION AUTHORIZATION	26
2.4.2	PARAMETER DYNAMIC CATALOG/SCHEMA/AUTHORIZATION	26
2.4.3	OPTION CATALOG/SCHEMA/AUTHORIZATION	28
2.4.4	Changing the database and schema	28
2.4.5	Current authorization identifier for compilation and execution	29
2.4.6	Examples of database environments	30
2.4.6.1	SESAM database and DRIVE programs	30
2.4.6.2	TIAM operation	31
2.4.6.3	UTM operation	34
2.5	DRIVE statements for SESAM V2	36
2.5.1	PARAMETER DYNAMIC - Define dynamic parameters	36
2.5.2	OPTION - Control program compilation	38
2.5.3	SESAM V2 settings in foreign environments	39
2.5.4	SHOW - Output information about metadata	40
2.6	Examples and sample database	42
2.6.1	Sample tables before and after migration	42
2.6.2	Command file for migration	47

2.6.3	DRIVE DDL programs for the table ABTEILUNG	50
2.6.4	DRIVE DDL program for access permissions and foreign keys	52
2.6.5	Sample programs	53
2.7	Pragmas	62
2.7.1	Application possibilities and advantages	62
2.7.2	Differences in syntax compared with ESQL/COBOL and the Utility Monitor	63
2.7.3	Static and dynamic pragmas	63
2.7.4	Pragma clauses	63
2.7.5	Error handling	66
3	DRIVE SQL statements	69
	ALTER TABLE - Alter base table	70
	CLOSE - Close cursor	74
	COMMIT WORK - Terminate transaction	75
	CREATE SCHEMA - Create schema	79
	CREATE TABLE - Create base table	81
	CREATE TEMPORARY VIEW - Declare temporary view	83
	CREATE VIEW - Create view	86
	DECLARE - Declare cursor	88
	DELETE - Delete rows	95
	DROP CURSOR - Release cursor description	97
	DROP SCHEMA - Delete schema	98
	DROP TABLE - Delete base table	99
	DROP TEMPORARY VIEW - Delete temporary view	100
	DROP VIEW - Delete view	101
	FETCH - Position cursor and read row	102
	GRANT - Grant privileges	106
	INSERT - Insert rows in table	111
	OPEN - Open cursor	116
	PERMIT - Specify user identification for old style	117
	PRAGMA - Declare pragma clauses	119
	PREFETCH pragma clause	121
	EXPLAIN pragma clause	122
	ISOLATION LEVEL pragma clause	124
	IGNORE INDEX pragma clause	124
	OPTIMIZATION LEVEL pragma clause	125
	SIMPLIFICATION pragma clause	126
	DATA TYPE pragma clause	126
	CHECK pragma clause	126
	RESTORE - Restore cursor	127
	REVOKE - Revoke privileges	129
	ROLLBACK WORK - Roll back transaction	132
	SELECT - Read individual rows	134
	SET CATALOG - Set default database name	136

SET SCHEMA - Set default schema name 138

SET SESSION AUTHORIZATION - Define authorization identifier 140

SET TRANSACTION - Define transaction attributes 142

STORE - Save cursor position 146

UPDATE - Update column values 147

WHENEVER - Define error handling 152

4 DRIVE SQL metavariables 157

 query_expression 159

 condition 162

 join_expression 165

 literal 168

 char_literal - Alphanumeric literal 169

 num_literal - Numeric literals 170

 date_time_literal - Time literals 171

 set_function 173

 AVG() - Calculate arithmetic average 174

 COUNT(*) - Count table rows 176

 COUNT() - Count elements 177

 MAX() - Determine largest value 179

 MIN() - Determine lowest value 181

 SUM() - Calculate sum 183

 predicate - Specify predicate 185

 Comparing two values 187

 Comparison with derived column 190

 Range queries 192

 Element queries 194

 Pattern comparison 196

 Comparison with the NULL value 199

 Existence queries 201

 select_expression 202

 select_list - Select derived columns 204

 FROM clause - Specify tables 206

 WHERE clause - Select derived rows 207

 GROUP BY clause - Group derived rows 208

 HAVING clause - Select groups 210

 column_constraint 211

 column_definition 213

 sql_expression 216

 table_specification 222

 table_constraint 225

 subquery 227

 variable 228

 value 230

	time_function	232
5	Syntax overview	235
5.1	Statements	235
	ALTER TABLE - Alter base table	235
	CLOSE - Close cursor	236
	COMMIT WORK - Terminate transaction	236
	CREATE SCHEMA - Create schema	236
	CREATE TABLE - Create base table	236
	CREATE TEMPORARY VIEW - Declare temporary view	237
	CREATE VIEW - Create view	237
	DECLARE - Declare cursor	237
	DELETE - Delete rows	238
	DROP CURSOR - Release cursor description	238
	DROP SCHEMA - Delete schema	238
	DROP TABLE - Delete base table	238
	DROP TEMPORARY VIEW - Delete temporary view	238
	DROP VIEW - Delete view	239
	FETCH - Position cursor and read row	239
	GRANT - Grant privileges	239
	INSERT - Insert rows in table	240
	OPEN - Open cursor	240
	PERMIT - Specify user identification for old style	241
	PRAGMA - Declare pragma clauses	241
	RESTORE - Restore cursor	241
	REVOKE - Revoke privileges	242
	ROLLBACK WORK - Roll back transaction	242
	SELECT - Read individual rows	243
	SET CATALOG - Set default database name	243
	SET SCHEMA - Set default schema name	243
	SET SESSION AUTHORIZATION - Define authorization identifier	244
	SET TRANSACTION - Define transaction attributes	244
	STORE - Save cursor position	244
	UPDATE - Update column values	245
	WHENEVER - Define error handling	245
5.2	Metavariables	246
	query_expression	246
	condition	246
	join_expression	246
	literal	247
	set_function	247
	predicate	248
	select_expression	248
	column_constraint	249

	column_definition	249
	sql_expression	250
	table_specification	250
	table_constraint	251
	subquery	251
	variable	251
	value	252
	time_function	252
6	Error messages	253
6.1	Mapping the SESAM SQLCODEs to &DML_STATE	253
	SQLCODEs from the language definition	253
	System-specific SQLCODEs	256
6.2	SQLSTATE classes	259
	Related publications	261
	Index	275

DRIVE/WINDOWS V2.1 (BS2000/OSD)

Directory of DRIVE SQL Statements for SESAM/SQL 2

Reference Manual

Target Group

The manual is aimed at programmers who develop DRIVE applications or components of client-server applications using DRIVE/WINDOWS on BS2000 computers and SESAM/SQL Server V2 as a database.

Contents

The manual describes all DRIVE SQL statements for SESAM/SQL 2 in alphabetical order together with their syntax and a description of their functional scope.

Edition: February 1996

File: DRV_SES2.PDF

BS2000 and DRIVE are registered trademarks of Siemens Nixdorf Informationssysteme AG

Copyright © Siemens Nixdorf Informationssysteme AG, 1996.

All rights are reserved

Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufactures.



Information on this document

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.

This document from the document archive refers to a product version which was released a considerable time ago or which is no longer marketed.

Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.

Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format ...@ts.fujitsu.com.

The Internet pages of Fujitsu Technology Solutions are available at

[http://ts.fujitsu.com/...](http://ts.fujitsu.com/)

and the user documentation at <http://manuals.ts.fujitsu.com>.

Copyright Fujitsu Technology Solutions, 2009

Hinweise zum vorliegenden Dokument

Zum 1. April 2009 ist Fujitsu Siemens Computers in den alleinigen Besitz von Fujitsu übergegangen. Diese neue Tochtergesellschaft von Fujitsu trägt seitdem den Namen Fujitsu Technology Solutions.

Das vorliegende Dokument aus dem Dokumentenarchiv bezieht sich auf eine bereits vor längerer Zeit freigegebene oder nicht mehr im Vertrieb befindliche Produktversion.

Bitte beachten Sie, dass alle Firmenbezüge und Copyrights im vorliegenden Dokument rechtlich auf Fujitsu Technology Solutions übergegangen sind.

Kontakt- und Supportadressen werden nun von Fujitsu Technology Solutions angeboten und haben die Form ...@ts.fujitsu.com.

Die Internetseiten von Fujitsu Technology Solutions finden Sie unter

[http://de.ts.fujitsu.com/...](http://de.ts.fujitsu.com/), und unter <http://manuals.ts.fujitsu.com> finden Sie die Benutzerdokumentation.

Copyright Fujitsu Technology Solutions, 2009