REFERENCE MANUAL | PUBLIC

SAP Adaptive Server Enterprise 16.0 SP03

Document Version: 1.0 – 2018-08-15

SAP ASE SQLScript Reference

THE BEST RUN SAP

Content

1 SAP ASE SQLScript

SAP ASE includes support for the SQLScript structured query language and extensions used by SAP HANA by adding a SQLScript parser, allowing you to build application logic using SAP HANA SQLScript syntax directly in SAP ASE.

SAP ASE servers can include databases that use the SQL dialect native to SAP HANA or SAP ASE. Having a common language between these systems allows you to take advantage of the benefits of both SAP databases, and allows you to build applications once and deploy them on both systems.

The SAP ASE SQLScript parser supports a subset of the SQLScript statements supported by SAP HANA. In addition to the language changes, there are a number of features added so that the SAP ASE SQLScript database is largely compatible with SAP HANA SQLScript, including:

- Schemas
- Exception handling
- Sequences
- Local variables
- Table UDFs
- Cursor parameters
- `order by` clauses in subqueries
- Multiple returns from scalar functions
- Upsert
- Scalar functions (GREATEST and LEAST)

The extensions include:

- Data extension, which allows the definition of table types without specifying actual tables
- Functional extension, which allows the definition of (side effect-free) functions, which can be used to express and encapsulate complex data flows
- Procedural extension, which provides imperative constructs executed in the context of the database process

This document lists those SQLScript extensions that SAP ASE supports, which allows you to create scripts or applications for your SAP ASE databases in SQLScript.

SAP ASE supports only a subset of SAP HANA SQLScript; when you issue SQLScript statements that are not supported in SAP ASE, the server returns an error message.

1.1 Considerations for Using SAP ASE SQLScript Databases

There are a number of things to consider when you are using SQLScript databases.

> **i Note**
>
> SAP ASE SQLScript database refers to a database supporting SAP HANA SQLScript and SQL extensions. In this document, the term SQLScript is also used to describe SQLScript in a stored procedure and SQL extensions for ad-hoc queries.

- Server-level DBA administration jobs are not fully supported in SQLScript database, execute them from the Transact-SQL master database.
- SAP ASE databases are either Transact-SQL databases or SQLScript databases, they cannot use both parsers for the same database.
- Only Transact-SQL statements can be used in Transact-SQL databases, and only SQLScript statements can be used in SQLScript databases.
- Although individual databases must use either Transact-SQL or SQLScript, the server can contain both SQLScript and Transact-SQL parsed databases.
- You cannot alter existing Transact-SQL databases to use the SQLScript parser.
- SQLScript does not allow multiple statements in a batch. Instead, you must include multiple statements in a compound statement:

  ```
  DO
  BEGIN
  <SQL statement>;
  END
  ```

1.2 Compatibility and Operating Systems

SAP ASE SQLScript compatibility and operating system support.

- SAP ASE SQLScript statements and functions are compatible with SAP HANA version 1.0 SPS12.
- Linux x86_64 is supported for Redhat and SUSE.

1.3 Working with Schemas

Schemas are containers that hold database objects such as tables, views, and stored procedures. SAP ASE SQLScript database schemas are compatible with SAP HANA schemas.

SQLScript databases can have multiple schemas. Every object in the database must belong to a schema, but an object cannot belong to more than one schema. You create the schema before you create the objects that occupy the schema. You cannot create two tables with the same name inside a single schema, even if they are created by different users.

Creating a Schema

Use the `create schema` or `create user` commands to create schemas. `create schema` creates a schema with the specified name; `create user` creates a default schema for the user, with the same name as the user (this user must have `create schema` privileges).

This example creates the `ASE_schema`:

```
create schema ASE_schema
```

This creates a user named `user_bob`, and also creates a schema named `user_bob`:

```
create user user_bob password really_secret
```

To create objects that use a schema that is different from the one configured for your session, include the alternative schema name as the prefix when you create and access the object. For example, if the current session schema is `ASE_schema`, create a table named `new_table` that uses the `other_schema` schema, by issuing:

```
create table other_schema.t1 (c int)
```

You must include the `<schema_name>` prefix when you access objects in other schemas. For example, to insert data into the `t1` table from the `ASE_schema`, issue:

```
insert into other_schema.t1 values (10)
```

And to select data from `t1` while using the `ASE_schema`:

```
select * from schema1.t1
c
-----------
10
```

If you do not specify a schema when you create or access an object, SQLScript uses the default schema.

Setting a Schema

Use the `set schema` command to set the schema for the current session:

```
set schema <schema_name>
```

Once you have set the schema for the session, all the objects created in the session belong to this schema, unless you execute another `set schema` command, or you explicitly specify the schema name as the prefix of an object name when you create it.

All database objects you create are part of this schema until you change schemas or you explicitly specify otherwise. To view the currently set schema, issue:

```
select current_schema from dummy

 ------------------------------------------
 ASE_schema
(1 row affected)
```

Dropping a Schema

Use `drop schema` to remove a schema. However, the command fails if the schema is not empty (that is, it contains tables, index, and so on). Instead, drop all the objects belonging to the schema, then issue `drop schema`, or use `drop schema cascade` to drop the schema and dependent objects. For example:

```
drop table other_schema.t1
go
drop schema other_schema
go
```

Related Information

SET SCHEMA Statement (Session Management) [page 258]
CREATE SCHEMA Statement (Data Definition) [page 186]
CREATE TABLE Statement (Data Definition) [page 191]
CURRENT_SCHEMA Function (Miscellaneous) [page 81]
DROP SCHEMA Statement (Data Definition) [page 222]
Supported SQLScript Procedures [page 17]
GRANT Statement (Access Control) [page 232]
Security [page 21]

1.4 Working with Sequences

A sequence is a database object used to generate unique integers by multiple users.

Sequences generate a series of incrementing or decrementing numbers based on the property settings for the sequence. There are two operations for sequence objects: getting the current value of a sequence and getting the next value of a sequence.

Sequences are not associated with tables; they are used by applications. Sequences provide an easy way to generate the unique values that applications use (for example, to identify a table row or a field). You can set options in the sequence specification that control the start and end point of the sequence, the size of the increment size, or the minimum and maximum allowed value. You can also specify whether the sequence should recycle values when it reaches the maximum value specified.

The sequence of numeric values is generated in an ascending or descending order at a defined increment interval. For example, applications can use numbers generated by a sequence can be used to identify the rows and columns of a table, providing unique primary keys.

The relationship between sequences and tables is controlled by the application. Applications can reference a sequence object and coordinate the values across multiple rows and tables.

You can use database sequences to perform these operations:

- Generate unique, primary key values. For example, to identify the rows and columns of a table.
- Coordinate keys across multiple rows or tables.

Creating a Sequence

Use the `CREATE SEQUENCE` command to create a sequence. This example creates a sequence named `seq1` that starts with the value 3, and increments this value by 3 with each iteration:

```
create sequence seq1 start with 3 increment by 3
```

The `INSERT` and `SELECT` commands allow you to include the `CURRVAL` and `NEXTVAL` operators to display or insert the sequence of numbers. The `CURRVAL` operator displays the current value generated by a sequence, and the `NEXTVAL` generates the next value in the defined sequence. Once a user increases the sequence value by issuing the `NEXTVAL` parameter, the current value of the sequence is changed immediately for other users.

You can use `CURRVAL` only after calling `NEXTVAL`. The `NEXTVAL` call can be during another, or a new, session.

To use the `NEXTVAL` parameter to determine the next value in the sequence, issue:

```
select seq1.nextval from dummy
nextval
--------------------
         3
```

This example matches the current value of the sequence:

```
select seq1.currval from dummy
currval
---------------------
         3
```

The value for `currval` is changed after calling `NEXTVAL`. For example, calling `seq.currval` returns 6:

```
select seq1.nextval from dummy
nextval
--------------------
         6
```

Altering a Sequence

Use the `ALTER SEQUENCE` command to change an existing sequence. This changes the sequence to increment the value by 2 instead of 3:

```
alter sequence seq1 increment by 2
```

When you select the next value from the sequence, it is incremented to a value of 8 instead of 9:

```
select seq1.nextval from dummy
nextval
--------------------
          8
```

Include the `NEXTVAL` operator with the `INSERT` command to populate a table:

```
create table tab1 (a int);
insert into tab1 values (seq1.nextval);
insert into tab1 values (seq1.nextval);
insert into tab1 values (seq1.nextval);
select * from tab1;
          a
--------------------
          10
          12
          14
```

Include the `NEXTVAL` operator with the `CREATE TABLE` command to populate a table with the sequential values:

```
create table tab2 as (select seq1.nextval as i, a from tab1) with data;
(4 rows affected)
select * from tab2;
go
     i            a
----------- -----------
      16          10
      18          12
      20          14
```

Use the `NEXTVAL` operator with the `UPDATE` command to update a table with new sequential numbers:

```
update tab2 set i = seq1.nextval;
select * from tab2;
     i              a
------------- -----------
       22          10
       24          12
       26          14
```

Dropping a Sequence

Use the `DROP SEQUENCE` command to remove the sequence:

```
drop sequence seq1
```

Related Information

CREATE SEQUENCE Statement (Data Definition) [page 187]
ALTER SEQUENCE Statement (Data Definition) [page 142]
DROP SEQUENCE Statement (Data Definition) [page 223]
GRANT Statement (Access Control) [page 232]
REVOKE Statement (Access Control) [page 246]
Security [page 21]

1.4.1 Sequence Within a Table as an Identity Column

Provide sequence information when you create or alter a table to specify it as an identity column.

Specifying `<identity>` in the `CREATE TABLE` and `ALTER TABLE` commands allows you to indicate that the column has the IDENTITY property.

Each table in a database can have one IDENTITY column with a data type of:

- Exact numeric and a scale of 0; or
- Any of the integer data types, including `signed` or `unsigned bigint`, `int`, `smallint`, or `tinyint`.

Restrictions for IDENTITY columns:

- A table can only have one IDENTITY column.
- You cannot modity the IDENTITY column using the `ALTER TABLE` command.
- Temporary table does not support IDENTITY column.
- You cannot update IDENTITY columns, and they do not allow nulls.
- IDENTITY columns are used to store sequential numbers—such as invoice numbers or employee numbers —that are generated automatically by the SAP ASE server. The value of the IDENTITY column uniquely identifies each row in a table.

Syntax

Use the following `CREATE TABLE` syntax to specify sequence information for the IDENTITY column.

Syntax

```
CREATE [<table_type>] TABLE <table_name> <table_contents_source>
<table_contents_source> ::= (<table_element>, ...)
<table_element> ::= <column_definition> [<column_constraint>] |
<table_constraint>
```

The syntax for `<column_definition>` is:

```
<column_definition> ::= <column_name> {<data_type> | <lob_data_type>}
    <col_gen_as_ident>
```

The syntax for `<col_gen_as_ident>` is:

```
<col_gen_as_ident> ::=
 GENERATED {ALWAYS | BY DEFAULT} AS IDENTITY [(<sequence_option>)]
```

The syntax for `<sequence_options>` is:

```
<sequence_option> ::=
 <sequence_param_list>
 | RESET BY <subquery>
 | <sequence_param_list> RESET BY <subquery>
```

Use the following `ALTER TABLE` syntax to add an IDENTITY column and specify sequence information for it.

Syntax

```
ALTER TABLE ADD ({<column_definition>}[, …] )
```

The syntax for `<column_definition>` is:

```
<column_definition> ::= <column_name>
   { <data_type> | <lob_data_type>}
    <col_gen_as_ident>
```

The syntax for `<col_gen_as_ident>` is:

```
<col_gen_as_ident> ::= GENERATED {ALWAYS | BY DEFAULT} AS IDENTITY
[(<sequence_option>)]
```

The syntax for `<sequence_option>` is:

```
<sequence_option> ::= {
   <sequence_param_list>
   | RESET BY <subquery>
   | <sequence_param_list>
  RESET BY <subquery>
}
```

The values you provide for `<sequence_options>` should match those you specified in the `CREATE SEQUENCE` statement's `<sequence_parameter_list>`.

Example

This creates the table `tab1`:

```
create table tab1 (id int generated always as identity(start with 100) primary
key, name varchar(30))
insert into tab1(name) values ('Andy')
go
select * from tab1
go
id          name
----------- ------------------------------------------------------------
100         Andy
```

This alters a table by adding an IDENTITY column:

```
create table tab1 (id int primary key, name varchar(30));
    insert into tab1 values (1, 'Mary');
    insert into tab1 values (2, 'Bob');
alter table tab1 add (refid int generated always as identity(start with 3));
    select * from tab1;

id    name    refid
--- -------- --------
1     Mary     3
2     Bob      4
```

Related Information

CREATE TABLE Statement (Data Definition) [page 191]
ALTER TABLE Statement (Data Definition) [page 145]
CREATE SEQUENCE Statement (Data Definition) [page 187]

1.5 Using the CALL Statement to Issue Procedures

SQLScript databases require that you include the CALL statement when you issue a stored procedure.

i Note

All SQLScript statements must be ended with a semicolon ";" in the stored procedure.

The syntax is:

```
CALL <procedure_name>
```

For example, to issue a stored procedure named `get_price` for the `pubs2.titles` table located in `schema1`, issue:

```
CALL get_price ('schema1.titles')
title                                        price         advance
```

```
 -------------------------------------    ------------  ----------------------
 The Busy Executive's Database Guide            19.99                 5,000.00
 Cooking with Computers: Surreptitio            11.95                 5,000.00
```

To display all the permissions granted on a particular schema, issue:

```
CALL ss_sp_helprotect('schema1')
go
grantor   grantee     type        action       object     column   predicate
grantable
------ ---------- -------------- ---------- ----------- ---------- ----------
---------
dbo       sso          Grant       Create        All         NULL         0
FALSE
bob      mary         Grant       Select       schema1    All          0        TRUE
```

To issue a user-defined procedure named `user_proc1`, issue:

```
CALL user_proc1
```

CALL requires that you include the procedure parameters in single quotes and within parenthesis. The syntax is:

```
CALL procedure_name ('<parameter>')
```

For example, to run the `sp_helprotect` procedure against the T-SQL version of the `pubs2.titles` table in a standard SAP ASE database, issue:

```
CALL ss_sp_helprotect pubs2.titles
```

However, to issue the same system procedure from a SQLScript-enabled `pubs2` database (which requires that you include the schema name), issue:

```
CALL ss_sp_helprotect ('<schema_name>.titles')
```

Related Information

CALL Statement (Procedural) [page 164]

1.5.1 Supported SQLScript Procedures

You cannot run T-SQL system procedures in a SQLScript database. However, you can run some system procedure from the master database to perform operations on the SQLScript database (for example, `sp_dboption`).

To execute stored procedures, issue the procedure with the CALL command. For example, to issue `sp_help` for the `pubs2.titles` table, located in schema1, issue:

```
CALL sp_help ('schema1.titles')
go
Name      Owner      Object_type     Object_status       Create_date
```

```
------ --------- -------------- ---------------- -------------------
titles    dbo      user table       -- none --    Jan 24 2017 9:45AM
. . .
(return status = 0)
```

To display all the permissions granted on a particular schema, issue:

```
CALL sp_helprotect schema1
go
grantor   grantee     type       action       object    column   predicate
grantable
------ --------- -------------- ---------- ----------- --------- ----------
---------
dbo       sso         Grant      Create       All        NULL         0
FALSE
bob       mary        Grant      Select       schema1    All          0
TRUE
```

When you create a new user, a default schema is created with the same name as the user. This example creates a user named "joe" and also creates a new default schema called "joe":

```
CALL create user joe
go
```

To displays identifiers that are SQLScript reserved words, issue:

```
CALL sp_checkreswords
go
Reserved Words Used as Database Object Names for Database,sqlscript_DB1.
 Upgrade renames sysobjects.schema to sysobjects.schemacnt.
 Database-wide Objects
 ---------------------
 Found no reserved words used as database object names.
(return status = 0)
```

See SAP ASE SQLScript Procedures [page 46] for a description of the procedures that are specific to SQLScript and are installed when you run `installSQLScriptProc`.

Related Information

Creating a SQLScript Database [page 19]
SAP ASE SQLScript Procedures [page 46]

1.6 Creating a SQLScript Database

You must create a SQLScript-enabled database to use the SAP ASE SQLScript syntax.

Procedure

1. If necessary, use `disk init` to create a device for the SQLScript database. There are no SQLScript parameters required to create a device for this database.
2. Create an SAP ASE SQLScript database from the master database using the `create database … for sqlscript` command. The syntax is:

   ```
   create database <dbname>
   [on {default | database_device} [= <size>]
   [, database_device [= <size>]]...]
   [log on database_device [= <size>]
   [, database_device [= <size>]]...]
   for sqlscript
   ```

   For example, this creates the `sqlscipt_DB` on the `sqlscript_dev` device:

   ```
   use master
   go
   create database sqlscript_DB
   on sqlscript_dev = '300M'
   for sqlscript
   go
   ```
3. (Optional) Install the auditing system. For more information, see Security Administration Guide > Auditing > Managing the Audit System > Install the Audit System.
   - To set auditing options, see ss_sp_audit [page 46].
   - To show audit records, see ss_sp_show_audit_record [page 60].
   - To truncate the audit table, see ss_sp_truncate_audit_table [page 61].
4. Using `isql` from the command line, execute the `installSQLScriptProc` installation script to install the SQLScript-related system procedures. Use the `-D` parameter to pipe this script into the SQLScript database. The syntax is:

   ```
   $SYBASE/$SYBASE_OCS/bin/isql -U<user_name> -P<password> -S<server_name> -
   D<dbname> -i$SYBASE/$SYBASE_ASE/scripts/installSQLScriptProc
   ```

   A successful `installSQLScriptProc` run looks similar to:

   ```
   $SYBASE/$SYBASE_OCS/bin/isql -Usa -P -Sbig_server -Dsqlscript_DB -iASE-16_0/
   scripts/installSQLScriptProc
   DBCC execution completed. If DBCC printed error messages, contact a user with
   System Administrator (SA) role.
   DBCC execution completed. If DBCC printed error messages, contact a user with
   System Administrator (SA) role.
   (1 row affected)
   (1 row affected)
   (1 row affected)
   (1 row affected)
   (1 row affected)
   DBCC execution completed. If DBCC printed error messages, contact a user with
   ```

```
System Administrator (SA) role.
installing procedure ss_sp_helptext
DBCC execution completed. If DBCC printed error messages, contact a user with
System Administrator (SA) role.
(1 row affected)
DBCC execution completed. If DBCC printed error messages, contact a user with
System Administrator (SA) role.
installing procedure ss_sp_help_allobjs
DBCC execution completed. If DBCC printed error messages, contact a user with
System Administrator (SA) role.
(1 row affected)
DBCC execution completed. If DBCC printed error messages, contact a user with
System Administrator (SA) role.
installing procedure ss_sp_odbc_datatype_info
DBCC execution completed. If DBCC printed error messages, contact a user with
System Administrator (SA) role.
(1 row affected)
```

5. (Optional) Configure the monitoring tables. Enable these configuration parameters:
    - `enable monitoring`
    - `statement statistics active`
    - `statement pipe active`
    - `per object statistics active`

    Set this configuration parameter to a value appropriate for your site:
    - `statement pipe max messages`
6. Create a schema (that is, a container that holds database objects such as tables, views, and stored procedures) using the `create schema` or `create user` commands. Create the schema in an SQLScript database before you create the objects that the schema holds.

    When you create a new user, a default schema creates a user with the same name. A default dbo schema is automatically created for the sa user. You should create schemas to group objects.

    This example creates the `ASE_schema`:

    ```
    create schema ASE_schema
    ```

    This creates a user named `user_bob`, and also creates a schema named `user_bob`:

    ```
    create user user_bob password really_secret
    ```
7. Access the SAP ASE SQLScript database and verify that the database is created correctly.

    For example, issue this from a non-ODBC or non-JDBC based application (which do not allow you to switch between two different types of databases):

    ```
    USE <dbname>
    SELECT CURRENT_DATABASE() FROM DUMMY
    ```

    If the database is created correctly, the database name is returned:

    Sample Code

    ```
    USE sqlscript_DB
    GO
    SELECT CURRENT_DATABASE() FROM DUMMY
    GO
     -------------------------------------
     sqlscript_DB
    ```

```
(1 row affected)
```

8. Issue the `set schema` command to use the schema you created earlier for your current session. This sets the session's schema to `ASE_schema`:

   ```
   set schema ASE_schema
   ```

   Unless you specify a new schema, all subsequent objects you create during this session will use this schema.

Related Information

CREATE SCHEMA Statement (Data Definition) [page 186]
SET SCHEMA Statement (Session Management) [page 258]

1.7 Security

User Management

The following DDLs are supported on SQLScript databases to manage users.

Command	Syntax
CREATE USER	`CREATE USER <username> PASSWORD <password>`
ALTER USER	`ALTER USER <username> PASSWORD <newpassword>`
DROP USER	`DROP USER <username>`

Privileges and Roles

SAP ASE SQLScript supports three types of grants and revokes: system privileges, object privileges, schema privileges.

System Privileges

System privileges protect the execution of tasks that are not restricted to a particular schema or any other database object. The tasks that users with these privileges can perform affect the database as a whole, such as creation of a user or granting of roles.

Privilege	Privilege Task
CREATE SCHEMA	Create schemas.
USER ADMIN	Create, drop, or alter user accounts.
ROLE ADMIN	Create, drop, grant, or revoke roles.

Schema Privileges

You can grant schema privileges on a particular schema and govern the tasks that affect one particular schema. For example, you can grant a user the privilege to create a table that is inside a particular schema.

Privilege	Privilege Task
SELECT	Select any table, view, sequence in the schema.
UPDATE	Update any table in the schema.
DELETE	Delete from any table in the schema.
INSERT	Insert into any table in the schema.
EXECUTE	Execute any procedure or function in the schema.
CREATE ANY	Create any object within a schema. Objects that can be created are tables, views, procedures, sequences, and functions.
REFERENCES	Create foreign key reference on any table in the schema.

Object Privileges

Object privileges can be granted on a particular object and control the tasks performed on that object. For example, you can grant a user SELECT on a particular table, or EXECUTE on a particular procedure.

Privilege	Privilege Task	Applicable Object
SELECT	Select from tables, views, or sequences	Tables, views, sequences
UPDATE	Update tables	Tables
DELETE	Delete from tables	Tables
INSERT	Insert into tables	Tables
EXECUTE	Execute procedures and functions	Procedures, functions
REFERENCES	Create foreign key references	Tables
ALL PRIVILEGES	Grant select, update, delete, insert, and references	Tables and views

Roles

Roles offer a way to grant and revoke a set of privileges to and from a user. For example, you can create a role named `user_management_role`, with the USER ADMIN or ROLE ADMIN privilege. The role can then be granted by the DBA to a user who is allowed to perform these user management tasks, or the role can be revoked and granted to another user, allowing a level of indirection between users and privileges for better management.

Roles can be created and dropped as follows:

- CREATE ROLE `<rolename>`
- DROP ROLE `<rolename>`

Grant Statement

Use the `GRANT` statement to grant roles and privileges to users, or other roles. Privileges can also be granted with the `WITH GRANT OPTION` clause, which allows the grantee to grant or revoke the permission to and from others. For system privileges, the same can be achieved with the `WITH ADMIN OPTION` clause.

Grant Privilege	Syntax

Grant privileges to users or roles	`GRANT USER ADMIN TO <username>`
	`GRANT USER ADMIN, ROLE ADMIN TO <rolename>`

Grant role to users or roles	`GRANT <rolename> to <username>`
	`GRANT <role1> to <role2>`

Grant schema permissions	`GRANT SELECT ON SCHEMA <schemaname> TO <username>`
	`GRANT CREATE ANY ON SCHEMA <schemaname> TO <username>`

Grant object permissions	`GRANT DELETE ON <tablename> TO <username>`
	`GRANT EXECUTE ON <procedurename> TO <username>`

Grant `WITH GRANT OPTION`	`GRANT USER ADMIN TO <username> WITH ADMIN OPTION`
	`GRANT SELECT ON SCHEMA <schemaname> TO <username> WITH GRANT OPTION`
	`GRANT EXECUTE ON <procedurename> TO <username> WITH GRANT OPTION`

i Note

You cannot grant roles that give the grantee the ability to grant or revoke the role.

Revoke Statement

Use the REVOKE statement to revoke roles and privileges from users or other roles.

Revoke Privilege	Syntax

Revoke privileges from users or roles	`REVOKE USER ADMIN FROM <username>`
	`REVOKE USER ADMIN, ROLE ADMIN FROM <rolename>`

Revoke role from users or roles	`REVOKE <rolename> FROM <username>`
	`REVOKE <role1> from <role2>`

Revoke schema permissions	`REVOKE SELECT ON SCHEMA <schemaname> FROM <username>`
	`REVOKE CREATE ANY ON SCHEMA <schemaname> FROM <username>`

Revoke object permissions	`REVOKE DELETE ON <tablename> FROM <username>`
	`REVOKE EXECUTE ON <procedurename> FROM <username>`

Reporting Procedure

The `ss_sp_helprotect` procedure allows you to query granted roles and privileges. For more information, see ss_sp_helprotect [page 55].

Auditing

Configure auditing policies for a SQLScript database using `ss_sp_audit`, `ss_sp_show_audit_record`, and `ss_sp_truncate_audit_table`. For more information, see SAP ASE SQLScript Procedures [page 46].

Related Information

GRANT Statement (Access Control) [page 232]
REVOKE Statement (Access Control) [page 246]
Working with Sequences [page 11]
Working with Schemas [page 9]
CREATE SCHEMA Statement (Data Definition) [page 186]
CREATE SEQUENCE Statement (Data Definition) [page 187]
DROP SCHEMA Statement (Data Definition) [page 222]
DROP SEQUENCE Statement (Data Definition) [page 223]

1.8 Using BCP with SQLScript Databases

Use the `bcp` utility to copy data in and out of SQLScript-enabled databases.

Issuing `bcp` against a SQLScript database requires that you include the schema name with the table name. The syntax is as follows, where `<schema_name>` is the name of the schema to which the table belongs:

```
bcp <database_name>.<schema_name>.<table_name> in|out
```

For example, this example copies the `titles` table out of the SQLScript-enabled database `pubs2_sql` into a file named `pub_out`:

```
bcp pubs2_sql.schema1.titles out pub_out -Ujoe -Pjoe123 -Sbigserver -I$SYBASE/
interfaces -c
```

The `<schema_name>` parameter is not needed when copying data from the output file into a table. This example copies data from the `pub_out` file into the `pubs2.titles` table:

```
bcp pubs2_sql..titles in pub_out -Ujoe -Pjoe123 -Sbigserver -I$SYBASE/interfaces
-c -Y
```

You cannot include partition information when you are copying data in and out of SQLScript-enabled databases.

2 Data Types

SQLScript supports the following data types:

SQLScript Data Types Support:

- Datetime Data Types [page 26]
- Numeric Data Types [page 29]
- Character String Data Types [page 31]
- Large Object (LOB) Data Types [page 32]
- Binary Data Types [page 34]
- Boolean Data Type [page 34]
- Table Types [page 35]

SQLScript does not support these data types:

- Spatial data types are not supported.
- Array (multi-valued data types) are not supported.

2.1 Datetime Data Types

Datetime data types are used to store date and time information.

Data Type	Description
`DATE`	The DATE data type consists of year, month, and day information to represent a date value. The default format for the DATE data type is YYYY-MM-DD. YYYY represents the year, MM represents the month, and DD represents the day. The range of the date value is between 0001-01-01 and 9999-12-31.
`TIME`	The TIME data type consists of hour, minute, and second information to represent a time value. The default format for the TIME data type is HH24:MI:SS. HH24 represents the hour from 0 to 24, MI represents the minute from 0 to 59, and SS represents the second from 0 to 59.

Data Type	Description
`SECONDDATE`	The SECONDDATE data type consists of year, month, day, hour, minute and second information to represent a date with a time value. The default format for the SECONDDATE data type is YYYY-MM-DD HH24:MI:SS. YYYY represents the year, MM represents the month, DD represents the day, HH24 represents hours, MI represents minutes, and SS represents seconds. The range of the date value is between 0001-01-01 00:00:01 and 9999-12-31 24:00:00.
`TIMESTAMP`	The TIMESTAMP data type consists of date and time information. Its default format, YYYY-MM-DD HH24:MI:SS`<n>`, represents the fractional seconds, where `<n>` indicates the number of digits in fractional part. The range of the time stamp value is between 0001-01-01 00:00:00.0000000 and 9999-12-31 23:59:59.9999999.

The EMPTY value refers to the lowest possible value of each datetime type and ensures compatibility with ABAP.

Date Formats

Use the following datetime formats when parsing a string into a datetime type and converting a datetime type value into a string value. The format for TIMESTAMP is the combination of DATE and TIME with additional support for fractional seconds. An empty date (0000-00-00) is a special value in SAP HANA. Even though an empty date looks like a NULL or unknown value, it is not. For example, the empty date can be represented as `''`, which behaves like an empty string. It also satisfies a IS NOT NULL predicate. Use the following statements to test the behavior of the empty date value:

```
CREATE TABLE T (A INT, B DATE, C DATE);
INSERT INTO T VALUES (1, '', '0001-01-01');
INSERT INTO T VALUES (2, '0000-00-00', '0001-01-01');
INSERT INTO T VALUES (3, '0000-00-00', '0001-01-01');
INSERT INTO T VALUES (4, '0001-01-01', '0001-01-01');
INSERT INTO T VALUES (5, NULL, '0001-01-01');
SELECT * FROM T WHERE B = '00000000';
SELECT * FROM T WHERE B = '';
SELECT * FROM T WHERE B IS NOT NULL;
SELECT * FROM T;
SELECT DAYS_BETWEEN(B, C) FROM T;
```

Table 1: Supported Formats for DATE

Format	Description	Examples
YYYY-MM-DD	Default format.	`INSERT INTO my_tbl VALUES ('1957-06-13');`

Format	Description	Examples

YYYY/MM/DD
 	
YYYY/MM-DD
 	
YYYY-MM/DD
	YYYY is from 0001 to 9999, MM is from 1 to 12, and DD is from 1 to 31. If year has less than four digits, month has less than two digits, or day has less than two digits, then values are padded by one or more zeros. For example, a two-digit year like 45 is saved as year 0045, while a one-digit month like 9 is saved as 09, and a one-digit day like 2 is saved as 02.	`INSERT INTO my_tbl VALUES ('1957-06-13');`
		
		`INSERT INTO my_tbl VALUES ('1957/06/13');`
		
		`INSERT INTO my_tbl VALUES ('1957/06-13');`
		
		`INSERT INTO my_tbl VALUES ('1957-06/13');`

YYYYMMDD	The ABAP data type, DATS format.	`INSERT INTO my_tbl VALUES ('19570613');`
MON	The abbreviated name of month. For example, JAN or DEC.	`INSERT INTO my_tbl VALUES (TO_DATE('2040-Jan-10', 'YYYY-MON-DD'));`

MONTH	The name of month. For example, JANUARY - DECEMBER.	`INSERT INTO my_tbl VALUES (TO_DATE('2040-January-10', 'YYYY-MONTH-DD'));`
		
		`INSERT INTO my_tbl VALUES (TO_DATE('January-10', 'MONTH-DD'));`

RM	The Roman numeral month (I-XII; JAN = I).	`INSERT INTO my_tbl VALUES (TO_DATE('2040-I-10', 'YYYY-RM-DD'));`

Time Formats

Table 2: Supported Formats for TIME

Format	Description	Examples
HH24:MI:SS	The default format.	

Format	Description	Examples

HH:MI[:SS][AM|PM]
HH12:MI[:SS][AM|PM]
HH24:MI[:SS]
	HH is from 0 to 23. MI is from 0 to 59. SS is from 0 to 59. FFF is from 0 to 999.
	If a one-digit hour, minute, or second is specified, then 0 is also inserted into the value. For example, 9:9:9 is saved as 09:09:09.
	HH12 indicates a 12-hour clock. HH24 indicates a 24-hour clock.
	Specify AM or PM as a suffix to indicate that the time value is before or after midday.
		`INSERT INTO my_tbl VALUES ('23:59:59');`
		`INSERT INTO my_tbl VALUES ('3:47:39 AM');`
		`INSERT INTO my_tbl VALUES ('9:9:9 AM');`
		`INSERT INTO my_tbl VALUES (TO_TIME('11:59:59'));`

Timestamp Formats

Table 3: Supported Formats for TIMESTAMP

Format	Description
YYYY-MM-DD HH24:MI:SS.FF7	The default format.

2.2 Numeric Data Types

Use numeric data types to store numeric information.

i Note

SAP ASE restrictions:

- The SMALLDECIMAL numeric data type is not supported.
- The precision difference of decimal data type is different between SAP ASE and SAP HANA.

Each supported numeric type has a maximum and minimum value. A numeric overflow exception is thrown if a value is smaller than the minimum value or greater than the maximum value. In order to comply with IEEE754, NaN and infinity are not supported . -0.0 is stored as +0.0.

Floating-point data types are stored in the system using binary numbers. The fractional part of these numbers is represented using a combination of 1/2, 1/4, 1/8, 1/16, and so on. For this reason, they cannot completely represent rational numbers with fractional digits. For example, because 0.1 cannot be represented exactly by combining these binary fractions, you obtain inaccurate results when using a floating-point data type. This is the correct behavior for these data types. The following example demonstrates the behavior and returns `4.6999999999` for DOUBLE_SUM:

```
SELECT TO_DOUBLE(0.1) + TO_DOUBLE(4.6) AS DOUBLE_SUM FROM DUMMY;
```

The supported data types are:

Data Type	Description
TINYINT	The TINYINT data type stores an 8-bit unsigned integer. The minimum value is 0 and the maximum value is 255.
INTEGER	The INTEGER data type stores a 32-bit signed integer. The minimum value is -2,147,483,648 and the maximum value is 2,147,483,647.
BIGINT	The BIGINT data type stores a 64-bit signed integer. The minimum value is -9,223,372,036,854,775,808 and the maximum value is 9,223,372,036,854,775,807.
DECIMAL(`<precision>`, `<scale>`) or DEC(`<p>`,`<s>`)	DECIMAL(`<p>`, `<s>`) is the SQL standard notation for fixed-point decimals. `<p>` specifies precision, or the number of total digits (the sum of whole digits and fractional digits). `<s>` denotes scale, or the number of fractional digits. If a column is defined as DECIMAL(5, 4) for example, the numbers 3.14, 3.1415, 3.141592 are stored in the column as 3.1400, 3.1415, 3.1415, retaining the specified precision(5) and scale(4). `<precision>` specifies the maximum number of decimal digits that can be stored in the column. It includes all digits, both to the right and to the left of the decimal point. You can specify precisions ranging from 1 digit to 38 digits or use the default precision of 18 digits. `<scale>` specifies the maximum number of digits that can be stored to the right of the decimal point. The scale must be less than or equal to the precision. You can specify a scale ranging from 0 digits to 38 digits, or use the default scale of 0 digits. If precision and scale are not specified, then DECIMAL becomes a floating-point decimal number. In this case, precision and scale can vary within the range of 1 to 34 for precision and -6,111 to 6,176 for scale, depending on the stored value. For example, 0.0000001234 (1234E-10) has precision 4 and scale 10. 1.0000001234 (10000001234E-10) has precision 11 and scale 10. The value 1234000000 (1234E6) has precision 4 and scale -6.
REAL	The REAL data type specifies a single-precision, 32-bit floating-point number.

Data Type	Description
DOUBLE	The DOUBLE data type specifies a double-precision, 64-bit floating-point number. The minimum value is -1.7976931348623157E308 and the maximum value is 1.7976931348623157E308 . The smallest positive DOUBLE value is 2.2250738585072014E-308 and the largest negative DOUBLE value is -2.2250738585072014E-308.
FLOAT(`<n>`)	The FLOAT(`<n>`) data type specifies a 32-bit or 64-bit real number, where `<n>` specifies the number of significant bits and can range between 1 and 53. If you use the FLOAT(`<n>`) data type, and `<n>` is smaller than 25, then the 32-bit REAL data type is used. If `<n>` is greater than or equal to 25, or if `<n>` is not declared, then the 64-bit DOUBLE data type is used.

2.3 Character String Data Types

Character string data types are used to store values that contain character strings. VARCHAR data types contain 7-bit ASCII character strings, and NVARCHAR are used for storing Unicode character strings.

> i Note
>
> SAP ASE restrictions:
>
> - The SHORTTEXT character string data type is not supported.
>
> > i Note
> >
> > The SAP HANA database does not officially support the CHAR and NCHAR data types. Even though they are available for use, they are only for legacy support and consistent behavior is not guaranteed for values of these types between a row table and a column table. Use VARCHAR and NVARCHAR instead.

Collation expressions are not supported, and values of string data type are compared using a binary comparison.

Character string data types in SAP HANA use 7-bit ASCII. Extended ASCII characters are converted into corresponding Unicode characters. If the data includes anything other than 7-bit ASCII characters, use Unicode character string types, such as NVARCHAR and NCLOB.

Data Types	Description
VARCHAR	The VARCHAR(<n>) data type specifies a variable-length character string, where <n> indicates the maximum length in bytes and is an integer between 1 and 5000. If the length is not specified, the default is 1. If the VARCHAR(<n>) data type is used in a DML query, for example `CAST (A as VARCHAR(n))`, then <n> indicates the maximum length of the string in characters. Use VARCHAR with 7-bit ASCII character-based strings only. For data containing other characters, use the NVARCHAR data type instead.
NVARCHAR	The NVARCHAR(<n>) data type specifies a variable-length Unicode character set string, where <n> indicates the maximum length in characters and is an integer between 1 and 5000. If the length is not specified, then the default is 1.
ALPHANUM	The ALPHANUM(<n>) data type specifies a variable-length character string that contains alpha-numeric characters, where <n> indicates the maximum length and is an integer between 1 and 127. Sorting among values of type ALPHANUM is performed in alpha-representation. In the case of a purely numeric value, this means that the value can be considered as an alpha value with leading zeros.

2.4 Large Object (LOB) Data Types

LOB (large objects) data types, such as CLOB, NCLOB, and BLOB, are used to store a large amount of data, such as text documents and images. The maximum size for a LOB is 2 GB.

i Note

SAP ASE does not support BINTEXT:

Data Type	Description
BLOB	The BLOB data type is used to store large amounts of binary data. BLOB values can be converted to VARBINARY. The SAP HANA BLOB data type is mapped to SAP ASE IMAGE data type.

Data Type	Description
CLOB	The CLOB data type is used to store large amounts of 7-bit ASCII character data. CLOB values can be converted to VAR-CHAR. The SAP HANA CLOB and BCLOB data types are mapped the SAP ASE TEXT data type.
NCLOB	The NCLOB data type is used to store a large Unicode char-acter object. NCLOB values can be converted to NVARCHAR. The SAP HANA NCLOB data types is mapped the SAP ASE TEXT data type.
TEXT	The SAP ASE SQLScript database only supports row tables. The SAP HANA TEXT data type is mapped the SAP ASE TEXT data type. Selecting a TEXT column yields a column of type TEXT. A value of type TEXT cannot be converted implicitly to a value of type (N)VARCHAR, and string functions (UPPER, LOWER, and so on) cannot be applied directly to a value of type TEXT. Explicit conversion from a value of type TEXT to a value of type (N)VARCHAR is allowed. String functions can therefore be applied to the converted value. For columns of type TEXT, the LIKE predicate is not sup-ported.

LOB types support the following operations:

- The LENGTH() function for values of type CLOB/NCLOB/BLOB, which returns the LOB length in bytes.
- The SUBSTR() function for values of type CLOB/NCLOB, which returns the substring of a (N)CLOB value.
- The COALESCE() function.
- The LIKE and CONTAINS predicate for values of type CLOB/NCLOB.
- The IS NULL predicate for values of type CLOB/NCLOB/BLOB.

LOB data types have the following restrictions:

- LOB columns cannot appear in ORDER BY or GROUP BY clauses.
- LOB columns cannot appear in FROM clauses as join predicates.
- LOB columns cannot appear in WHERE clauses as predicates other than LIKE (meaning that no comparison is allowed).
- LOB columns cannot appear in SELECT clauses as aggregate function arguments.
- LOB columns cannot appear in SELECT DISTINCT clauses.
- LOB columns cannot be used in set operations such as EXCEPT. UNION ALL is an exception.
- LOB columns cannot be used as primary keys.
- LOB columns cannot be used in CREATE INDEX statements.
- LOB columns cannot be used in statistics update statements.

2.5 Binary Data Types

Binary types are used to store bytes of binary data.

A value of type binary can be converted to a value of type (N)VARCHAR if its size is smaller than or equal to 8192. It can therefore be used like a value of type (N)VARCHAR except for full text search operations and numeric operations.

The VARBINARY(`<n>`) data type is used to store binary data of a specified maximum length in bytes, where `<n>` indicates the maximum length and is an integer between 1 and 5000. If the length is not specified, then the default is 1.

2.6 Boolean Data Type

The BOOLEAN data type stores the TRUE and FALSE Boolean values.

i Note

SAP ASE does not support the UNKNOWN value.

When the client does not support a boolean type, it returns 1 for TRUE and 0 for FALSE.

The following example returns TRUE or 1 for boolean:

```
CREATE TABLE TEST (A BOOLEAN);
INSERT INTO TEST VALUES (TRUE);
INSERT INTO TEST VALUES (FALSE);
SELECT A "boolean" FROM TEST WHERE A = TRUE;
```

Although predicates and Boolean expressions can both have the same values (TRUE and FALSE), they are not the same. Therefore, you cannot use Boolean type comparisons to compare predicates, or use predicates where boolean expressions should be used.

For example, the following statement does not work:

```
SELECT * FROM DUMMY WHERE ( A>B ) = ( C>D );
```

The following statement is the correct way to achieve the results desired from the statement above:

```
SELECT * FROM DUMMY WHERE
   case when ( A>B ) then TRUE when NOT ( A>B ) then FALSE else NULL end=
   case when ( C>D ) then TRUE when NOT ( C>D ) then FALSE else NULL end;
```

2.7 Table Types

SAP ASE SQLScript supports both scalar SQL data types and user-defined tabular types for tabular values. The table type is specified using a list of attribute names and primitive data types. For each table type, attributes must have unique names.

Example

This example creates a table type called `tab_type` that is used in a procedure:

```
CREATE TYPE tab_type AS TABLE (I INT, A VARCHAR)
go
CREATE TABLE mytab(I INT, A VARCHAR)
go
INSERT INTO mytab values(1, 'Have9try')
go
CREATE PROCEDURE test_table
AS
BEGIN
DECLARE tab tab_type;
tab = SELECT * FROM mytab;
SELECT * FROM :tab;
END;
go
```

When you issue `test_table`, you can refer to the created table type in SQLScript procedures and functions:

```
CALL test_table
go
I           A
----------- -
1
```

Related Information

CREATE TYPE Statement (Procedural) [page 210]
DROP TYPE Statement (Procedural) [page 229]

3 Expressions

An expression is a clause that can be evaluated to return values, and is a combination of one or more constants, literals, functions, column identifiers, or variables, separated by operators.

> **i Note**
>
> SAP ASE does not support:
>
> - JSON object expressions.
> - The following for aggregate expressions:
>   - CORR
>   - CORR_SPEARMAN
>   - MEDIAN
>   - STRING_AGG

Syntax

```
<expression> ::= <case_expression>
               | <function_expression>
               | <aggregate_expression>
               | <json_object_expression>
               | (<expression> )
               | ( <subquery> )
               | - <expression>
               | <expression> <operator> <expression>
               | <variable_name>
               | <constant>
               | [<correlation_name>.]<column_name>
```

Case Expressions

A case expression allows the user to use if-then-else logic without using procedures in SQL statements.

Syntax

```
<case_expression> ::= <simple_case_expression> | <search_case_expression>
<simple_case_expression> ::=
            CASE <expression>
            WHEN <expression> THEN <expression>
            [{ WHEN <expression> THEN <expression>}…]
```

```
             [ ELSE <expression>]
             END
<search_case_expression> > ::=
             CASE
             WHEN <condition> THEN <expression>
             [{ WHEN <condition> THEN <expression>}…]
             [ ELSE <expression>]
             END
<condition> ::= <condition> OR <condition> | <condition> AND <condition>
               | NOT <condition> | ( <condition> ) | <predicate>
```

If the expression following the CASE statement is equal to the expression following the WHEN statement, the expression following the THEN statement is returned. Otherwise, the expression following the ELSE statement is returned if it exists.

Function Expressions

SQL built-in functions can be used as expressions.

Syntax

```
 <function_expression> ::= <function_name> ( <expression> [{, <expression>}...] )
```

Aggregate Expressions

An aggregate expression uses an aggregate function to calculate a single value from the values of multiple rows in one or more columns.

Syntax

```
 <aggregate_expression> ::= COUNT(*) | COUNT ( DISTINCT <expression> ) |
<agg_name> (  [ ALL | DISTINCT ] <expression> ) )
        <agg_name> ::= COUNT | MIN | MEDIAN | MAX | SUM | AVG | STDDEV | VAR  |
STDDEV_POP | VAR_POP | STDDEV_SAMP | VAR_SAMP
        <delimiter> ::= <string_constant>
        <aggregate_order_by_clause> ::= ORDER BY <expression> [ ASC | DESC ]
```

SAP ASE unsupported options:

- `<expression>` does not support more than one parameter.
- `<delimiter>`
- `<aggregate_order_by_clause>`
- NULLS FIRST
- NULLS LAST

Aggregate name	Description
COUNT	Counts the number of rows returned by a query. COUNT(*) returns the number of rows, regardless of the value of those rows and including duplicate values. COUNT(`<expression>`) returns the number of non-NULL values for that expression returned by the query. COUNT(DISTINCT `<expression>`) returns the number of distinct values for that expressions returned by the query, excluding rows with all NULL values for that expression.
MIN	Returns the minimum value of the expression.
MAX	Returns the maximum value of the expression.
SUM	Returns the sum of the expression.
AVG	Returns the arithmetical mean of the expression.
STDDEV	Returns the standard deviation of the given expression as the square root of the VAR function.
STDDEV_POP	Returns the standard deviation of the given expression as the square root of the VAR_POP function.
STDDEV_SAMP	Returns the standard deviation of the given expression as the square root of the VAR_SAMP function.
VAR	Returns the variance of the given expression as the square of the standard deviation.
VAR_POP	Returns the population variance of expression as the sum of squares of the difference of `<expression>` from the mean of `<expression>`, divided by the number of rows remain-ing.
VAR_SAMP	Returns the sample variance of expression as the sum of squares of the difference of `<expression>` from the mean of `<expression>`, divided by the number of rows remaining minus 1 (one).This function is similar to VAR, the only differ-ence is that it returns NULL when the number of rows is 1.

Table 4: Result Type of Numeric Aggregate Expressions

Aggregate Name	tinyint	smallint	integer	bigint	decimal(p,s)	decimal	real	double
COUNT	bigint	bigint	bigint	bigint	bigint	bigint	bigint	bigint
MIN	tinyint	smallint	integer	bigint	decimal(p,s)	decimal	real	double

Aggregate Name	tinyint	smallint	integer	bigint	decimal(p,s)	decimal	real	double
MAX	tinyint	smallint	integer	bigint	decimal(p,s)	decimal	real	double
SUM	integer	integer	integer	bigint	decimal(p',s)*	decimal	real	double
AVG	decimal(9,6)	decimal(11,6)	decimal(16,6)	decimal(25,6)	decimal(p,s)	decimal	real	double
STDDEV	decimal(9,6)	decimal(11,6)	decimal(16,6)	decimal(25,6)	decimal(p,s)	decimal	real	double
VAR	decimal(9,6)	decimal(11,6)	decimal(16,6)	decimal(25,6)	decimal(p,s)	decimal	real	double

* p' is determined by the following rule: p' = 18 when p <= 18, p' = 28 when p <= 28 and p' = 38 when p <= 38

Subqueries in Expressions

A subquery is a SELECT statement enclosed in parentheses. The SELECT statement can contain no more than one select list item. When used as an expression, a scalar subquery can only return a zero or a single value. The subquery may include an `order by` clause, but it must appear in a noncorrelated subquery with a `LIMIT` clause.

Syntax

```
<scalar_subquery_expression> ::= (<subquery>)
```

In the SELECT list of the top level SELECT, or in the SET clause of an UPDATE statement, you can use a scalar subquery anywhere where you can use a column name. You cannot, however, use the scalar subquery inside the GROUP BY clause.

The following statement returns the number of employees in each department for example, grouped by department name:

```
SELECT DepartmentName, COUNT(*), 'out of',
(SELECT COUNT(*) FROM Employees)
FROM Departments AS D, Employees AS E
WHERE D.DepartmentID = E.DepartmentID
GROUP BY DepartmentName;
```

4 Predicates

A predicate is specified by combining one or more expressions or logical operators, and returns the TRUE or FALSE logical values.

SQLScript Predicates Support:

- Comparison Predicates [page 40]
- LIKE Predicate [page 41]
- BETWEEN Predicate [page 42]
- NULL Predicate [page 43]
- EXISTS Predicate [page 45]
- IN Predicate [page 44]

Unsupported Predicate

The CONTAINS predicate is not supported.

4.1 Comparison Predicates

Two values are compared using comparison predicates, and the comparison returns true, false, or unknown.

Syntax

```
<comparison_predicate> ::=  <expression> { = | != | <> | > | < | >= | <= }
                            [ ANY | SOME | ALL ] { <subquery> }
```

Syntax Elements

```
<expression_list> ::= <expression> [{, <expression>}...]
```

An <expression> is a scalar subquery, or a simple expression such as a character, a date or a number:

```
ANY | SOME
```

Specifies that the comparison returns true if the comparison of the <expression> and at least one value returned by the <subquery> or <expression_list> is true.

If the <subquery> or <expression_list> is empty, the comparison returns false.

```
ALL
```

Specifies that the comparison returns true if the comparison of all values returned by the <subquery> is true. If the <subquery> is empty, the comparison returns true.

4.2 LIKE Predicate

The LIKE predicate performs a comparison to see if a character string matches, or does not match, a specified pattern.

Syntax

```
<like_predicate> ::= <source_expression> [NOT] LIKE <pattern_expression>
   [ESCAPE <escape_expression>]
```

Syntax Elements

source_expression Specifies the character string in which to search for <pattern_expression>. Specifying NOT inverts the operation of the LIKE predicate.
pattern_expression Specifies the pattern to search for in <source_expression>.
escape_expression

Specifies the escape character used in comparison string <pattern_expression>, if any.

Description

The LIKE predicate performs string comparisons: <source_expression> is tested for the pattern contained in <pattern_expression>. Assuming NOT is not set, LIKE returns true if the value of <pattern_expression> is found in <source_expression>.

<pattern_expression> can use wildcard characters (%) and (_). The percentage sign (%) wildcard matches zero or more characters. The underscore (_) wildcard matches exactly one character.

To match a percentage sign or underscore with the LIKE predicate, place an escape character in front of the wildcard character. Use ESCAPE `<escape_expression>` to specify the escape character you are using. For example, `LIKE 'data_%' ESCAPE '_'` matches the string **`data%`**, and `LIKE 'data__%' ESCAPE '_'` (that is, two underscores, followed by a percent sign) matches a string that starts with **`'data_'`**

The underscore (_) and percentage sign (%) are ASCII characters.

An expression is either a simple expression such as a character, a date or a number, or it can be a scalar subquery.

4.3 BETWEEN Predicate

The BETWEEN predicate compares a value with a list of values within the specified range and returns true or false.

Syntax

```
<between_predicate> ::= <expression> [NOT] BETWEEN <lower_expression> AND
<upper_expression>
```

Syntax Elements

expression

The value to search for in the specified list of values.

lower_expression

An expression that sets the lower bound of the value list to compare `<expression>` to.

upper_expression

An expression that sets the upper bound of the value list to compare `<expression>` to.

NOT

Inverts the operation of the BETWEEN predicate: returns TRUE when `<expression>` is not in the range of values specified between `<lower_expression>` and `<upper_expression>`, including equal to `<lower_expression>` and `<upper_expression>`.

Description

The range predicate returns true if `<expression1>` is within the range specified by `<lower_expression>` and `<upper_expression>`, and NOT is not specified.

i Note

TRUE will only be returned if `<lower_expression>` has a value less than or equal to `<upper_expression>`.

An expression is either a scalar subquery or a simple expression such as a character, a date or a number.

4.4 NULL Predicate

The NULL predicate compares the value of an expression with NULL.

Syntax

```
<null_predicate> ::= <expression> IS [NOT] NULL
```

Syntax Elements

expression

An expression is a scalar subquery, or a simple expression such as a character, a date or a number.

IS NULL

Returns true if the value of `<expression>` is NULL.

IS NOT NULL

Returns true if the value of `<expression>` is not NULL.

Description

Performs a comparison of the value of an expression with NULL.

4.5 IN Predicate

The IN predicate searches for a value in a set of values and returns true or false.

i Note

SAP ASE does not support an `IN` predicate with a set of values. For example, the following is not supported:

```
create table t11(a int, b int, c int); select * from t11 where (a,b) in
(select 1, 3 from dummy);
```

Syntax

```
<in_predicate> ::= <expression1> [NOT] IN { <expression> | <subquery> }
```

Syntax Elements

expression1

The value to search for in the set.

expression

Specifies an expression in which to to search for `<expression1>`.

subquery

Specifies a subquery in which to search for `<expression1>`.

NOT

Inverts the operation of the IN predicate: true is returned if `<expression1>` is *not* found in the specified set.

Description

True will be returned if the value of `<expression1>` is found in the `<expression>` or `<subquery>`.

An expression is a scalar subquery, or simple expression such as a character, a date or a number.

4.6 EXISTS Predicate

The EXISTS predicate tests for the presence of a value in a set and returns either true or false.

Syntax

```
<exists_predicate> ::= [NOT] EXISTS ( <subquery> )
```

Syntax Elements

NOT

Inverts the operation of the EXISTS predicate: true is returned when the `<subquery>` returns an empty result set and false is returned when the `<subquery>` returns a result set.

subquery

Specifies what to test for.

Description

Returns true if the `<subquery>` returns a result set that is not empty and returns false if the `<subquery>` returns an empty result set.

5 SAP ASE SQLScript Procedures

The installSQLScriptProc script installs a number of system procedures for managing and monitoring the SQLScript database. Do not use the T-SQL system procedures (for example, sp_who) within a SQLScript-enabled database.

The SQLScript system procedures include:

- ss_sp_audit [page 46]
- ss_sp_help [page 51]
- ss_sp_help_allobjs [page 54]
- ss_sp_helprotect [page 55]
- ss_sp_helptext [page 57]
- ss_sp_help_usage [page 58]
- ss_sp_odbc_datatype_info [page 59]
- ss_sp_show_audit_record [page 60]
- ss_sp_truncate_audit_table [page 61]

5.1 ss_sp_audit

Configures auditing policies for a SQLScript database.

Running ss_sp_audit requires that you enable auditing and create the sybsecurity database. For more information, see Security Administration Guide > Auditing.

i Note

Not all audit options are applicable for SQLScript databases.

Syntax

```
call ss_sp_audit (<option>, <login_role_name>, <object_name> [,<setting>])
```

Parameters

- **option** is a global, user-specific, database-specific, or object-specific option.

```
  { "adhoc" | "all" | "allow" | "alter" | "autotuning_rule" | "bcp" | "bind"
| "cluster" | "cmdtext"
  | "config_history" | "create" | "dbaccess" | "dbcc" | "delete" | "deny" |
"disk" | "drop" | "dump"
  | "dump_config" | "encryption_key" | "errors" | "errorlog" |
"exec_procedure" | "exec_trigger"
  | "func_obj_access" | "func_dbaccess" | "grant" | "insert" | "install" |
"load" | "login" | "login_admin"
  | "login_locked" | "logout" | "mount" | "password" | "quiesce" |
"reference" | "remove" | "revoke"
  | "role" | "role_locked" | "rpc" | "security" | "security_profile" |
"select" | "setuser" | "sproc_auth"
  | "table_access" | "thread_pool" | "transfer_table" | "truncate" | "unbind"
| "unmount"
  | "update" | "view_access" },
```

Table 5: Auditing Options

Option	Description
`adhoc`	Allows users to use `sp_addauditrecord` to add their own user-defined audit records to the audit trail.
`all`	Audits all actions.
`allow`	Audits the command `allow`.
`alter`	Audits the execution of the commands: `alter database`, `alter index`, `alter role`, `alter table`, `alter...modify owner` (including `alter encryption key modify owner`)
`bcp`	Audits the execution of the `bcp in` utility.
`bind`	Audits the execution of: `sp_bindefault`, `sp_bindmsg`, `sp_bindrule`
`cluster`	Audits cluster commands.
`cmdtext`	Audits the SQL text entered by a user. System stored procedures and command password parameters can be replaced with a fixed-length string of asterisks for security purposes.
`config_history`	Audits configuration history.
`create`	Audits the create commands: `create database`, `create default`, `create function`, `create index`, `create procedure`, `create role`, `create rule`, `create service`, `create table`, `create trigger`, `create view`
`dbaccess`	Audits access to the database from another database.
`dbcc`	Audits the execution of all `dbcc` commands.
`delete`	Audits the deletion of rows from a table or view.
`deny`	Audits the `deny` command.
`disk`	Audits the execution of these commands: `disk init`, `disk mirror`, `disk refit`, `disk reinit`, `disk remirror`, `disk resize`, `disk unmirror`

Option	Description
drop	Audits the executions of the commands: drop database, drop default, drop function, drop index, drop procedure, drop role, drop rule, drop service, drop table, drop trigger, drop view, sp_dropmessage
dump	Audits the execution of the commands: dump database, dump database...cumulative, dump transaction
dump_config	Audits the execution of the dump configuration tocommand.
encryption_key	Audits the execution of the commands: alter encryption key, create encryption key, drop encryption key, sp_encryption
errors	Audits fatal error and non-fatal errors.
errorlog	Audits changes to the error log.
exec_procedure	Audits the execution of a stored procedure.
exec_trigger	Audits any command that fires the trigger.
func_dbaccess	Audits access to a database using built-in functions.
func_obj_access	Audits access to an object using built-in functions.
grant	Audits the execution of the grant, and grant rolecommands.
insert	Audits the insertion of rows into a table or view using the insert command.
install	Audits the installation of Java classes using the installjava command.
load	load database, Audits the execution of the commands: load database..cumulative load transaction
login	Audits the execution of Audits a login attempt to the server for all logins or roles, a specific login, or a specific role (both system defined and user defined roles).
login_admin	Audits the execution of the commands alter login, create login, drop login by login administrators.
login_locked	Audits the host name and network IP address when a login account is locked due to exceeding the configured number of failed login attempts.
logout	Audits any logout from an SAP ASE server.
mount	Audits the execution of the mount database command.
password	Audits the events for global password and login policy options.
quiesce	Audits the execution of the quiesce database, prepare database commands.
reference	Audits the references between tables using the create table or alter table commands.
remove	Audits the removal of Java classes.
revoke	Audits the execution of the revoke, revoke role commands.

Option	Description
`role`	Audits the execution of the `alter role`, `create role`, `drop role`, `grant role`, `revoke role`, `set role` commands.
`role_locked`	Audits the `alter role..lock` command.
`rpc`	Audits remote procedure calls (either in or out)
`security`	Audits the server-wide security-relevant events.
`security_profile`	Audits the `alter login profile`, `create login profile`, `drop login profile` commands.
`select`	Audits the execution of the `select` command for a table or view.
`setuser`	Audits the execution of the `setuser` command.
`sproc_auth`	Audits the authorization checks that are done inside system stored procedures.
`table_access`	Audits access to any table by a specific user for the `select`, `delete`, `update`, or `insert` commands.
`thread_pool`	Audits the execution of the `alter thread pool`, `create thread pool`, `drop thread pool` commands.
`transfer_table`	Audits the execution of the `transfer table` command.
`truncate`	Audits the execution of the `truncate table` command.
`unbind`	Audits the execution of the `sp_unbindefault`, `sp_unbindrule`, `sp_unbindmsg` commands.
`unmount`	Audits the execution of the `unmount database` commands.
`update`	Audits updates to rows in a table or view.
`view_access`	Audits the access to any view by a specific user using the `select`, `delete`, `insert`, or `update` commands.

- **login_role_name** determines the logins or roles to audit.

```
{ "all" | <login_name> | <role_name> },
```

  all all roles and logins.

  login_name login to audit when the first parameter `<option>` is `all`. You can audit login attempts of the specified login when you use the global option `login` as the first parameter for `<option>`.

  role_name logins granted to the specified role (both system and user define roles) when the first parameter `<option>` is `all`. You can audit login attempts of the logins granted to the role when you use the global option `login` for `<option>`.

- **object_name** valid objects to audit.

```
{ "all" | "default table" | "default view" | "default procedure" | "default
trigger"  | <object_name> }
```

  all all valid objects.

default table is a valid value when you specify `delete`, `insert`, `select`, or `update` for `<option>` as the first parameter.

default view is a valid value when you specify `delete`, `insert`, `select`, or `update` for `<option>` as the first parameter.

default procedure is valid when you specify the `exec_procedure` for `<option>` as the first parameter.

default trigger is valid when you specify the `exec_trigger` for `<option>` as the first parameter.

object_name is the name of a specific object to be audited. Valid values depend on the value you specified for the global option.

You can specify the object name and include the owner's name if you do not own the object. For example, to audit a table named `inventory` that is owned by Joe, you would specify `joe.inventory` for `<object_name>`.

- **setting** determines the settings for the audit events. The server generates audit records for events controlled by this option, whether the event passes or fails permission checks.

```
[, "off" | "on" | "pass" | "fail" ]
```

off deactivates auditing for the specified option.

on activates auditing for the specified option.

pass activates auditing for events that pass permission checks.

fail activates auditing for events that fail permission checks.

- **restart** If the audit process is forced to terminate due to an error, you can manually restart auditing by using the `restart` option.

The audit process can be restarted provided that no audit was currently running, but that the audit process has been configured to run by entering `sp_configure “auditing” 1`.

Examples

Example 1

Configures auditing for `select` commands on table `t1` (contained in schema `s1`):

```
call dbo.ss_sp_audit('select','all','s1.t1','on')
Audit option has been changed and has taken effect immediately.
(return status = 0)
```

Example 2

Configures auditing for `update` commands on table `t1` (contained in schema `s1`) that have passed the permission check:

```
call dbo.ss_sp_audit('update', 'all', 's1.t1', 'pass')
Audit option has been changed and has taken effect immediately.
(return status = 0)
```

Example 3

Configures auditing for all failed executions by users with role `role1`:

```
call dbo.ss_sp_audit('all', 'role1', 'all', 'fail')
```

```
Audit option has been changed and has taken effect immediately.
(return status = 0)
```

Usage

- `sp_audit` determines what is audited when auditing is enabled. No actual auditing takes place until you use `sp_configure` to set the `auditing` parameter to `on`, at which point, all auditing options that have been configured with `sp_audit` take effect. For more information, see `sp_configure`.
- If the object does not belong to your default schema, then qualify the object with the schema name as follows:

  ```
  "<schemaname>.<objname>"
  ```
- The configuration parameters that control auditing are:
  - `auditing` – enables or disables auditing for the server.
  - `audit queue size` – establishes the size of the audit queue.
  - `current audit table` – sets the current audit table. The SAP ASE server writes all audit records to that table.
  - `suspend auditing when full` – controls the behavior of the audit process when an audit device becomes full.

  All auditing configuration parameters are dynamic and take effect immediately.
- If you do not specify a value for the fourth parameter, SAP ASE displays the current auditing setting for the option. If you specify `pass` for an option and later specify `fail` for the same option, or vice versa, the result is equivalent to specifying `on`. The SAP ASE server generates audit records regardless of whether events pass or fail permission checks.
  - `on` or `off` – apply to all auditing options
  - `pass` and `fail` – apply to all options except `cmdtext`, `errors`, and `adhoc`. For these options, only `on` or `off` applies. The initial, default value of all options is `off`. If you select the `cmdtext` option to either `pass` or `fail`, the SAP ASE server replaces the value with `on`.

5.2 ss_sp_help

Reports information about a database object (any object listed in `sysobjects`) and about system- or user-defined data types, as well as user-defined functions, computed columns, and function-based indexes. Columns display `optimistic_index_lock`.

Syntax

```
ss_sp_help ([<objname> [, <print_option>]])
```

Parameters

- **objname** is the name of any object. You cannot specify database names. `<objname>` can include tables, views, precomputed result sets, stored procedures, logs, rules, defaults, triggers, referential constraints, encryption keys, predicates, and check constraints, but refers to tables when you enable `optimistic_index_lock`. Use owner names if the object owner is not the user running the command and is not the database owner.
- **print_option** display the option. One of the following:
  - `help` – prints the help information.
  - `terse` – (tables only) displays an abridged list of object properties in tabular format.
  - `null` or '' – is the default option.

  .

Examples

Example 1

Displays a list of objects in `sysobjects` and displays each object's name, owner, and object type. Also displays a list of each user-defined data type in `systypes`, indicating the data type name, storage type, length, null type, default name, and rule name. Null type is 0 (null values not allowed) or 1 (null values allowed):

```
call ss_sp_help('','help')
Usage of ss_sp_help
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
---------------------------------------------
ss_sp_help : displays the information for given database object
Usage of ss_sp_help
----------------
call ss_sp_help(objname, printopt)
objname: Object name in current database. Object could be of any type
that ASE supports.
printopt: Support following options:
terse - display terse information for table objname
Valid only when @objname is of type table
help - display usage of sp_help
Examples
1) "call ss_sp_help(null, 'help')"
Display usage of sp_help.
2) "call ss_sp_help(objname, 'help')"
Display usage of sp_help.
3) "call ss_sp_help(objname)"
Display information of object "objname".
4) "call ss_sp_help(objname, 'terse')"
Display terse information for table objname.
(1 row affected)
(return status = 0)
```

Example 2

Print information about the `tab1` table:

```
call ss_sp_help('tab1','')
-----------------------------
This table is VOLATILE table.
(1 row affected)
Name      Owner   Schema_name Object_type Object_status   Create_date
-------- ------- ----------- ----------- --------------- -------------------
tab1      dbo       dbo      user table -- none --      Mar 21 2017 3:11PM
(1 row affected)
Column_name   Type  Length Object_storage Prec Scale Nulls Not_compressed
Identity
----------- ------- ------ -------------- ---- ----- ----- --------------
---------
id            int     4     in row        NULL  NULL    0       0           0
name        varchar  255    in row        NULL  NULL    0       0           0
(2 rows affected)
(return status = 0)
```

Example 3

Uses the terse parameter to report abridged information about table tab1:

```
call ss_sp_help('tab1','terse')
-----------------------------
This table is VOLATILE table.
(1 row affected)
Name      Owner   Schema_name Object_type Object_status   Create_date
-------- ------- ----------- ----------- --------------- -------------------
tab1      dbo      dbo       user table   -- none --     Mar 21 2017 3:11PM
(1 row affected)
Column_description
------------------------------------
"id" int not null in row
"name" varchar (255) not null in row
(2 rows affected)
```

Usage

- `ss_sp_help` looks for an object in the current database only.
- `ss_sp_help` works on temporary tables if you issue it from `tempdb`.
- Columns with the IDENTITY property have an "Identity" value of 1; others have an "Identity" value of 0. In the example that shows the printed output of tab1, there are no IDENTITY columns.
- `ss_sp_help` lists any indexes on a table, including indexes created by defining unique or primary key constraints in the `create table` or `alter table` statements. It also lists any attributes associated with those indexes. However, `ss_sp_help` does not describe any information about the integrity constraints defined for a table. Use `sp_helpconstraint` for information about any integrity constraints.
- `ss_sp_help` displays the locking scheme, which can be set with `create table` and changed with `alter table`.
- When Component Integration Services is enabled, `ss_sp_help` displays information on the storage location of remote objects.
- `ss_sp_help` displays information about encryption keys. When a key name is specified as the parameter to `ss_sp_help`, the command lists the key's name, owner, object type, and creation date.

- For tables enabled for in-memory row storage, `ss_sp_help` reports on row caching (even if it is temporarily disabled), and snapshot isolation.
- `ss_sp_help <tablename>` indicates if a column is encrypted, including the name of the decrypt default on the column, if one exists.

5.3 ss_sp_help_allobjs

This procedure typically displays information about the name, owner, schema and type for all objects in the database:
Displays all of the objects for the specified schema.

Syntax

```
ss_sp_help_allobjs
```

Examples

Returns Object Information

In general, this procedure returns four types information of the objects, name, owner, schema and type.

```
call ss_sp_help_allobjs
Name              Owner    SchemaName  Object_type
---------------- -----    ----------  ------------
tab1              dbo dbo  user        table
tab2              dbo dbo  user        table
tab67             dbo dbo  user        table
sysalternates     dbo dbo  system      table
sysattributes     dbo dbo  system      table
syscolumns        dbo dbo  system      table
syscomments       dbo dbo  system      table
sysconstraints    dbo dbo  system      table
sysdams           dbo dbo  system      table
sysdepends        dbo dbo  system      table
```

5.4 ss_sp_helprotect

Reports on granted roles and privileges.

Syntax

```
ss_sp_helprotect ([<name>[, <username>[,'grant'| <role_name>[,
<permission_name>]]]]])
```

Parameters

- **name** the name of user or object whose grants are to be reported.
- **username** the name of the user or role in the current database.
- **grant** displays the privileges granted on `<name>` to `<username>` with `grant` option.
- **role_name** lists privileges granted through `<role_name>`.
- **permission_name** allows `ss_sp_helprotect` to provide information (grantor name, grantee name, table or column name, grantability) for any specific permission granted in a given database.

Examples

Example 1

In this example, `s1` is a schema, and contains table `t1`, and includes these grants:

- CREATE ANY on `s1` to `procuser1`
- SELECT and DELETE on `t1` to `procuser1` and `procuser2`, respectively.
- DELETE to `procuser2`

CREATE ANY is not shown as a single grant. Instead all the underlying grants (CREATE FUNCTION, CREATE PROCEDURE, and so on) are displayed:

```
call ss_sp_helprotect('s1')
grantor     grantee        type       action                 object
column    predicate    grantable
---------- -------------- ---------- ----------------------- --------
------------ ----------- -----------
dbo        procuser1       Grant      Create Function          s1
All         NULL     FALSE
dbo        procuser1       Grant      Create Procedure         s1
All         NULL     FALSE
dbo        procuser1       Grant      Create Sequence          s1
All         NULL     FALSE
dbo        procuser1       Grant      Create Table             s1
All         NULL     FALSE
dbo        procuser1       Grant      Create View              s1
All         NULL     FALSE
```

```
dbo        procuser1       Grant      Select                    s1.t1
All         NULL     FALSE
dbo        procuser2       Grant      Delete                    s1.t1
All         NULL     TRUE
 (1 row affected)
 (return status = 0)
```

Example 2

In this example, `procuser1` was granted:

- CREATE ANY on schema `s1`
- USER ADMIN system privilege.

`procuser1` has received other grants indirectly by their public membership:

```
call ss_sp_helprotect('procuser1')
grantor     grantee        type       action                 object
column    predicate    grantable
---------- -------------- ---------- ----------------------- --------
------------ ----------- -----------
dbo       procuser1       Grant      Create Function        s1
All       NULL      FALSE
dbo       procuser1       Grant      Create Procedure       s1
All       NULL      FALSE
dbo       procuser1       Grant      Create Sequence        s1
All       NULL      FALSE
dbo       procuser1       Grant      Create Table           s1
All       NULL      FALSE
dbo       procuser1       Grant      Create View            s1
All       NULL      FALSE
dbo       procuser1       Grant      Select                 dbo.seq1
All       NULL      FALSE
dbo       procuser1       Grant      Select                 s1.t1
All       NULL      FALSE
dbo       procuser1       Grant      User Admin
All       NULL      FALSE
dbo       public          Grant      Execute                dbo.ss_sp_audit
All       NULL      FALSE
dbo       public          Grant      Execute                dbo.ss_sp_help
All       NULL      FALSE
...
```

Include `user admin` as the last parameter to determine if `procuser1` was granted user admin privileges:

```
call ss_sp_helprotect('procuser1', null, null, null, 'user admin')
grantor     grantee        type       action                 object
column    predicate    grantable
---------- -------------- ---------- ----------------------- --------
------------ ----------- -----------
dbo            procuser1          Grant      User Admin
All         NULL      FALSE
```

Example 4

Lists the grants on table `t1` contained in schema `s1`:

```
call ss_sp_helprotect('s1.t1')
grantor     grantee        type       action                 object
column    predicate    grantable
---------- -------------- ---------- ----------------------- --------
------------ ----------- -----------
dbo         procuser1       Grant      Select                  s1.t1
All       NULL        FALSE
dbo         procuser2       Grant      Delete                  s1.t1
All       NULL         TRUE
```

Example 5

List the grants on table `t1` contained in schema `s1` with `grant` option:

```
1> call ss_sp_helprotect('s1.t1', null, 'grant')
2> go
grantor     grantee        type       action                  object
column    predicate    grantable
---------- -------------- ---------- ----------------------- --------
------------ ----------- -----------
dbo          procuser2      Grant      Delete                  s1.t1
All         NULL          TRUE
```

Example 6

In this example, `procuser2` is granted select permission on sequence `seq1` contained in the default schema of dbo. Below lists `grant` on sequence `seq1` contained in the default schema for dbo:

```
call ss_sp_helprotect('seq1')
grantor     grantee        type       action                  object
column    predicate    grantable
---------- -------------- ---------- ----------------------- --------
------------ ----------- -----------
dbo          procuser1      Grant      Select                 dbo.seq1
All         NULL        FALSE
```

Example 7

A non-DBO user must qualify the `ss_sp_helprotect` as `dbo.ss_sp_helprotect` to execute:

```
call dbo.ss_sp_helprotect('s1')
grantor     grantee        type       action                  object
column    predicate    grantable
---------- -------------- ---------- ----------------------- --------
------------ ----------- -----------
dbo         procuser1       Grant      Create Function           s1
All         NULL       FALSE
dbo         procuser1       Grant      Create Procedure          s1
All         NULL       FALSE
dbo         procuser1       Grant      Create Sequence           s1
All         NULL       FALSE
dbo         procuser1       Grant      Create Table              s1
All         NULL       FALSE
dbo         procuser1       Grant      Create View               s1
All         NULL       FALSE
dbo         procuser1       Grant      Select                    s1.t1
All         NULL       FALSE
dbo         procuser2       Grant      Delete                    s1.t1
All         NULL       TRUE
```

5.5 ss_sp_helptext

Displays source text.

Displays the source text of a compiled object, as well as the text for user-defined functions, computed columns, or function-based index definitions.

Syntax

```
ss_sp_helptext (<objname>)
```

Parameters

- **objname** is the name of the compiled object for which the source text is to be displayed. The compiled object must be in the current database.

Examples

Displays Text for Procedure

Creates procedure `proc1` then displays the text:

```
create procedure proc1 as
   begin
     select 1 from dummy;
   end
go
call ss_sp_helptext('proc1')
go
# Lines of Text
----------------
1
(1 row affected)
text

----------------------------------------------------------
create procedure proc1 as
begin
select 1 from dummy;
end
(1 row affected)
(return status = 0)
```

5.6 ss_sp_help_usage

Generates help information for the `ss_sp_help` stored procedure.

Syntax

```
ss_sp_help_usage
```

Examples

Example 1

Displays information about a given object:

```
call ss_sp_help_usage
Usage of ss_sp_help
----------------------------------------------------------------
ss_sp_help : displays the information for given database object
Usage of ss_sp_help
----------------
call ss_sp_help(objname, printopt)
objname: Object name in current database. Object could be of any type
that ASE supports.
printopt: Support following options:
terse - display terse information for table objname
Valid only when @objname is of type table
help - display usage of sp_help
Examples
1) "call ss_sp_help(null, 'help')"
Display usage of sp_help.
2) "call ss_sp_help(objname, 'help')"
Display usage of sp_help.
3) "call ss_sp_help(objname)"
Display information of object "objname".
4) "call ss_sp_help(objname, 'terse')"
Display terse information for table objname.
(1 row affected)
```

5.7 ss_sp_odbc_datatype_info

Returns the differences between system data types and their ODBC driver equivalent (requires an ODBC driver).

Returns the differences between system data types and their ODBC driver equivalent (requires an ODBC driver). For example, the system `image` data type is changed to `blob` for ODBC drivers.

Syntax

```
ss_sp_odbc_datatype_info (<data_type>, <odbc_ver>)
```

Parameters

- **data_type** The integer value of the system data type.
- **odbc_ver** The ODBC driver version of the data type.

Examples

Display ODBC Data Type Equivalents

Displays the ODBC data type equivalents for the current database:

```
call ss_sp_odbc_datatype_info
TYPE_NAME DATA_TYPE COLUMN_SIZE LITERAL_PREFIX LITERAL_SUFFIX CREATE_PARAMS
NULLABLE CASE_SENSITIVE SEARCHABLE UNSIGNED_ATTRIBUTE FIXED_PREC_SCALE
AUTO_UNIQUE_VALUE LOCAL_TYPE_NAME MINIMUM_SCALE MAXIMUM_SCALE
SQL_DATA_TYPE SQL_DATETIME_SUB NUM_PREC_RADIX INTERVAL_PRECISION
---------- --------- ----------- -------------- -------------- -------------
-------- -------------- ---------- ------------------ ----------------
----------------- --------------- ------------- -------------
------------- ---------------- -------------- ------------------
nclob      -10       2147483647   ' '            NULL          1
1        1              NULL       0                 NULL
          unitext  NULL           NULL          -10
NULL          NULL             NULL
boolean      -7       1           NULL           NULL          NULL
0        0              2          NULL              0
NULL          bit              NULL             NULL
            -7       NULL      10              NULL
```

5.8 ss_sp_show_audit_record

Selects audit records from the audit table.

To use `ss_sp_show_audit_record`, you must first enable auditing and create the `sybsecurity` database. For more information, see *Security Administration Guide* > *Auditing*.

Syntax

```
call ss_sp_show_audit_record()
```

Examples

Displays Audit Records

Displays the current audit records for the database:

```
call ss_sp_show_audit_record()
go
event eventmod spid eventtime sequence suid dbid objid xactid loginname
dbname objname objowner extrainfo nodeid
----- -------- ---- --------- -------- ---- ---- ----- ------ ---------
------ ------- -------- --------- ------
 62       1     45   Mar 28 2017  2:21PM        1           1
```

```
          31515          80 NULL
         sa
         sybsecurity
         sysaudits_01
         dbo
         sa_role sso_role oper_role sybase_ts_role mon_role; ; ; ; ; ; sa/ase;
           NULL
 61        1     45   Mar 28 2017  2:21PM       1           1
          31515          80 NULL
         sa
         sybsecurity
         sysaudits_01
         dbo
         sa_role sso_role oper_role sybase_ts_role mon_role; ; ; ; ; ; sa/
ase;
           NULL
```

5.9 ss_sp_truncate_audit_table

Truncates the audit tables.

Syntax

```
call ss_sp_truncate_audit_table()
```

Examples

Truncates the Audit Table

```
call ss_sp_truncate_audit_table()
(return status = 0)
```

Issuing `ss_sp_show_audit_record` indicates that the audit tables are now empty:

```
call ss_sp_show_audit_record()
event eventmod spid eventtime sequence suid dbid objid xactid loginname
dbname objname objowner extrainfo nodeid
----- -------- ---- --------- -------- ---- ---- ----- ------ ---------
------ ------- -------- --------- ------

(0 rows affected)
(return status = 0)
```

6 SAP ASE SQLScript Functions

The SAP ASE SQLScript database supports a number of functions:

The SAP ASE SQLScript database supports these functions

- Aggregate Functions [page 62]
- Data Type Conversion Functions [page 63]
- Datetime Functions [page 63]
- Numeric Functions [page 65]
- Miscellaneous Functions [page 66]
- String Functions [page 67]

The SAP ASE SQLScript database does not support these functions

- Window functions
- Security functions
- Series data functions
- Hierarchy functions

6.1 Aggregate Functions

Aggregate functions are analytic functions that calculate an aggregate value based on a group of rows.

The SAP ASE SQLScript database supports the following:

- STDDEV_POP Function (Aggregate) [page 113]
- STDDEV_SAMP Function (Aggregate) [page 114]
- VAR_POP Function (Aggregate) [page 133]
- VAR_SAMP Function (Aggregate) [page 133]

The SAP ASE SQLScript database does not support the following:

- `AUTO_CORR`
- `CORR`
- `CORR_SPEARMAN`
- `CROSS_CORR`
- `DFT`
- `FIRST_VALUE`
- `LAST_VALUE`
- `MEDIAN`
- `NTH_VALUE`
- `STRING_AGG`

6.2 Data Type Conversion Functions

Data type conversion functions convert data from one data type to another data type.

Data type conversion functions are used to convert function arguments from one data type to another, or to test whether a conversion is possible.

> **Note**
>
> In both implicit and explicit numeric type conversions, these functions always truncate the least significant digits toward zero.

The SAP ASE SQLScript database supports the following:

- CAST Function (Data Type Conversion) [page 75]
- TO_ALPHANUM Function (Data Type Conversion) [page 116]
- TO_BIGINT Function (Data Type Conversion) [page 117]
- TO_BINARY Function (Data Type Conversion) [page 118]
- TO_BLOB Function (Data Type Conversion) [page 118]
- TO_CLOB Function (Data Type Conversion) [page 119]
- TO_DATS Function (Data Type Conversion) [page 121]
- TO_DOUBLE Function (Data Type Conversion) [page 121]
- TO_FIXEDCHAR Function (Data Type Conversion) [page 122]
- TO_INT Function (Data Type Conversion) [page 122]
- TO_INTEGER Function (Data Type Conversion) [page 123]
- TO_NCLOB Function (Data Type Function) [page 124]
- TO_NVARCHAR Function (Data Type Conversion) [page 124]
- TO_REAL Function (Data Type Conversion) [page 125]
- TO_SECONDDATE Function (Data Type Conversion) [page 126]
- TO_SMALLINT Function (Data Type Conversion) [page 127]
- TO_TIME Function (Data Type Conversion) [page 127]
- TO_TINYINT Function (Data Type Conversion) [page 129]
- TO_VARCHAR Function (Data Type Conversion) [page 130]

The SAP ASE SQLScript database offers limited support for the following:

- TO_DATE Function (Data Type Conversion) [page 119]
- TO_TIMESTAMP Function (Data Type Conversion) [page 128]
- TO_TIME Function (Data Type Conversion) [page 127]

The SAP ASE SQLScript database does not support the `TO_SMALLDECIMAL` data type function.

6.3 Datetime Functions

Date and time functions perform operations on date and time data types or return date or time information.

The SAP ASE SQLScript database supports the following:

- ADD_DAYS Function (Datetime) [page 69]
- ADD_MONTHS Function (Datetime) [page 70]
- ADD_SECONDS Function (Datetime) [page 70]
- ADD_YEARS Function (Datetime) [page 71]
- CURRENT_DATE Function (Datetime) [page 80]
- CURRENT_TIME Function (Datetime) [page 82]
- CURRENT_TIMESTAMP Function (Datetime) [page 82]
- DAYNAME Function (Datetime) [page 84]
- DAYOFMONTH Function (Datetime) [page 84]
- DAYOFYEAR Function (Datetime) [page 85]
- DAYS_BETWEEN Function (Datetime) [page 86]
- EXTRACT Function (Datetime) [page 86]
- HOUR Function (Datetime) [page 89]
- MINUTE Function (Datetime) [page 98]
- MONTH Function (Datetime) [page 100]
- MONTHNAME Function (Datetime) [page 100]
- NANO100_BETWEEN Function (Datetime) [page 101]
- NEXT_DAY Function (Datetime) [page 101]
- NOW Function (Datetime) [page 102]
- SECOND Function (Datetime) [page 109]
- SECONDS_BETWEEN Function (Datetime) [page 109]
- WEEK Function (Datetime) [page 134]
- WEEKDAY Function (Datetime) [page 135]
- YEAR Function (Datetime) [page 135]

The SAP ASE SQLScript database does not support the following:

- `ADD_MONTHS_LAST`
- `ADD_WORKDAYS`
- `CURRENT_UTCDATE`
- `CURRENT_UTCTIME`
- `CURRENT_UTCTIMESTAMP`
- `ISOWEEK`
- `LAST_DAY`
- `LOCALTOUTC`
- `MONTHS_BETWEEN`
- `QUARTER`
- `UTCTOLOCAL`
- `WORKDAYS_BETWEEN`
- `YEARS_BETWEEN`

6.4 Numeric Functions

Numeric functions perform mathematical operations on numerical data types or return numeric information.

Number functions take numeric values, or strings with numeric characters, as inputs and return numeric values. When strings with numeric characters are provided as inputs, implicit conversion from a string to a number is performed automatically before results are computed.

The SAP ASE SQLScript database supports the following:

- ABS Function (Numeric) [page 68]
- ACOS Function (Numeric) [page 69]
- ASIN Function (Numeric) [page 72]
- ATAN Function (Numeric) [page 73]
- ATAN2 Function (Numeric) [page 73]
- BITAND Function (Numeric) [page 74]
- CEIL Function (Numeric) [page 76]
- COS Function (Numeric) [page 79]
- COT Function (Numeric) [page 79]
- EXP Function (Numeric) [page 87]
- FLOOR Function (Numeric) [page 87]
- LN Function (Numeric) [page 93]
- MOD Function (Numeric) [page 99]
- POWER Function (Numeric) [page 103]
- RAND Function (Numeric) [page 104]
- SIGN Function (Numeric) [page 112]
- SIN Function (Numeric) [page 112]
- SQRT Function (Numeric) [page 113]
- TAN Function (Numeric) [page 116]

The SAP ASE SQLScript database offers limited support for the following:

- ROUND Function (Numeric) [page 106]

The SAP ASE SQLScript database does not support the following:

- `BITCOUNT`
- `BITNOT`
- `BITOR`
- `BITSET`
- `BITUNSET`
- `BITXOR`
- `COSH`
- `LOG`
- `RAND_SECURE`
- `SINH`
- `TANH`
- `UMINUS`

6.5 Miscellaneous Functions

SAP HANA supports many functions that return system values and perform various operations on values, expressions, and return values of other functions.

The SAP ASE SQLScript database supports the following:

- COALESCE Function (Miscellaneous) [page 77]
- CURRENT_SCHEMA Function (Miscellaneous) [page 81]
- CURRENT_USER Function (Miscellaneous) [page 83]
- IFNULL Function (Miscellaneous) [page 90]
- NULLIF Function (Miscellaneous) [page 103]
- SESSION_CONTEXT Function (Miscellaneous) [page 110]
- SESSION_USER Function (Miscellaneous) [page 111]

The SAP ASE SQLScript database offers limited support for the following:

- CURRENT_IDENTITY_VALUE Function (Miscellaneous) [page 80]
- GREATEST Function (Miscellaneous) [page 88]
- LEAST Function (Miscellaneous) [page 91]

The SAP ASE SQLScript database does not support the following:

- `CARDINALITY`
- `CONVERT_CURRENCY`
- `CONVERT_UNIT`
- `CURRENT_CONNECTION`
- `CURRENT_OBJECT_SCHEMA`
- `CURRENT_TRANSACTION_ISOLATION_LEVEL`
- `CURRENT_UPDATE_STATEMENT_SEQUENCE`
- `CURRENT_UPDATE_TRANSACTION`
- `GROUPING`
- `GROUPING_ID`
- `HASH_SHA256`
- `HASH_MD5`
- `JSON_QUERY`
- `JSON_TABLE`
- `JSON_VALUE`
- `MAP`
- `MEMBER_AT`
- `NEWUID`
- `RESULT_CACHE_ID`
- `RESULT_CACHE_REFESH_TIME`
- `SUBARRAY`
- `SYSUUID`
- `TRIM_ARRAY`
- `WIDTH_BUCKET`

- XMLEXTRACT
- XMLEXTRACTVALUE

6.6 String Functions

String functions perform extraction and manipulation on strings, or return information about strings.

The SAP ASE SQLScript database supports the following:

- ASCII Function (String) [page 72]
- CHAR Function (String) [page 76]
- CONCAT Function (String) [page 78]
- HEXTOBIN Function (String) [page 89]
- LCASE Function (String) [page 91]
- LEFT Function (String) [page 92]
- LENGTH Function (String) [page 93]
- LOWER Function (String) [page 96]
- LPAD Function (String) [page 97]
- REPLACE Function (String) [page 105]
- RIGHT Function (String) [page 105]
- RPAD Function (String) [page 107]
- SUBSTRING Function (String) [page 115]
- TRIM Function (String) [page 131]
- UCASE Function (String) [page 131]
- UPPER Function (String) [page 132]

The SAP ASE SQLScript database offers limited support for the following:

- LTRIM Function (String) [page 97]
- RTRIM Function (String) [page 108]

The SAP ASE SQLScript database does not support the following:

- `ABAP_UPPER`
- `ABAP_LOWER`
- `BINTOHEX`
- `BINTONHEX`
- `BINTOSTR`
- `HAMMING_DISTANCE`
- `LIKE_REGEXPR`
- `LOCATE`
- `LOCATE_REGEXPR`
- `NCHAR`
- `OCCURRENCES_REGEXPR`
- `REPLACE_REGEXPR`
- `STRTOBIN`

- `SUBSTR_AFTER`
- `SUBSTR_BEFORE`
- `SUBSTRING_REGEXPR`
- `UNICODE`

6.7 Alphabetical List Of Functions

6.7.1 ABS Function (Numeric)

Returns the absolute value of a numeric argument.

Syntax

```
ABS (<n>)
```

Description

Returns the absolute value of the numeric argument `<n>`.

Example

The following example returns the value `1` for **`"absolute"`**:

```
SELECT ABS (-1) "absolute" FROM DUMMY;
```

6.7.2 ACOS Function (Numeric)

Returns the arc-cosine, in radians, of a numeric argument between -1 and 1.

Syntax

```
ACOS (n)
```

Description

Returns the arc-cosine, in radians, of the numeric argument n between -1 and 1.

Example

The following example returns the value `1.0471975511965979` for "acos":

```
SELECT ACOS (0.5) "acos" FROM DUMMY;
```

6.7.3 ADD_DAYS Function (Datetime)

Computes the specified date plus the specified days.

Syntax

```
ADD_DAYS (<d>, <n>)
```

Description

Computes the date d plus n days.

Example

The following example increments the date value **2009-12-05** by **30** days, and returns the value `2010-01-04`:

```
SELECT ADD_DAYS (TO_DATE ('2009-12-05', 'YYYY-MM-DD'), 30) "add days" FROM DUMMY;
```

6.7.4 ADD_MONTHS Function (Datetime)

Computes the specified date plus the specified months.

Syntax

```
ADD_MONTHS (<d>, <n>)
```

Description

Computes the date d plus n months.

Example

The following example increments the date **2009-12-05** by **1** month, and returns the value `2010-01-05`:

```
SELECT ADD_MONTHS (TO_DATE ('2009-12-05', 'YYYY-MM-DD'), 1) "add months" FROM
DUMMY;
```

6.7.5 ADD_SECONDS Function (Datetime)

Computes the specified time plus the specified seconds.

Syntax

```
ADD_SECONDS (<t>, <n>)
```

Description

Computes the time `<t>` plus `<n>` seconds.

Example

The example increments the TIMESTAMP **`2012-01-01 23:30:45`** by **`60*30`** seconds, and returns the value `2012-01-02 00:00:45.0`:

```
SELECT ADD_SECONDS (TO_TIMESTAMP ('2012-01-01 23:30:45'), 60*30) "add seconds"
FROM DUMMY;
```

6.7.6 ADD_YEARS Function (Datetime)

Computes the specified date plus the specified years.

Syntax

```
ADD_YEARS (<d>, <n>)
```

Description

Computes the date d plus n years.

Example

The following example increments the specified date value **`2009-12-05`** by **`1`** year, and returns the value `2010-12-05`:

```
SELECT ADD_YEARS (TO_DATE ('2009-12-05', 'YYYY-MM-DD'), 1) "add years" FROM
DUMMY;
```

6.7.7 ASCII Function (String)

Returns the integer ASCII value of the first byte in a specified string.

Syntax

```
ASCII(<c>)
```

Description

Returns the integer ASCII value of the first byte in a string `<c>`.

Example

This example converts the first character of the string **`Ant`** into an numeric ASCII value and returns the value `65`:

```
SELECT ASCII('Ant') "ascii" FROM DUMMY;
```

6.7.8 ASIN Function (Numeric)

Returns the arc-sine, in radians, of a numeric argument.

Syntax

```
ASIN (<n>)
```

Description

Returns the arc-sine, in radians, of the numeric argument `<n>` between -1 and 1.

Example

The following example returns the value `0.5235987755982989` for **"asin"**:

```
SELECT ASIN (0.5) "asin" FROM DUMMY;
```

6.7.9 ATAN Function (Numeric)

Returns the arc-tangent, in radians, of a numeric argument.

Syntax

```
ATAN (<n>)
```

Description

Returns the arc-tangent, in radians, of the numeric argument `<n>`. The range of `<n>` is unlimited.

Example

The following example returns the value `0.4636476090008061` for **"atan"**:

```
SELECT ATAN (0.5) "atan" FROM DUMMY;
```

6.7.10 ATAN2 Function (Numeric)

Returns the arc-tangent, in radians, of the ratio of two numbers.

Syntax

```
ATAN2 (<n>, <m>)
```

Description

Returns the arc-tangent, in radians, of the ratio of two numbers `<n>` and `<m>`.

Example

The following example returns the value `0.4636476090008061` for **"atan2"**:

```
SELECT ATAN2 (1.0, 2.0) "atan2" FROM DUMMY;
```

6.7.11 BITAND Function (Numeric)

Performs an AND operation on the bits of two arguments.

Syntax

```
BITAND (<n>, <m>)
```

Description

Performs an AND operation on the bits of the arguments `<n>` and `<m>`, where `<n>` and `<m>` must be non-negative INTEGER or VARBINARY values. The BITAND function returns a result along the argument's type.

Example

The following example returns the value `123` for **"bitand"**:

```
SELECT BITAND (255, 123) "bitand" FROM DUMMY;
```

6.7.12 CAST Function (Data Type Conversion)

Returns the value of an expression converted to a supplied data type.

Syntax

```
CAST (<expression> AS <data_type>)
```

Syntax Elements

expression

Specifies the expression to be converted.

data_type

Specifies the target data type.

```
<data_type> ::=
 TINYINT
 | SMALLINT
 | INTEGER
 | BIGINT
 | DECIMAL
 | SMALLDECIMAL
 | REAL
 | DOUBLE
 | ALPHANUM
 | VARCHAR
 | NVARCHAR
 | DAYDATE
 | DATE
 | TIME
 | SECONDDATE
 | TIMESTAMP
```

Description

Returns the value of an expression converted to a supplied data type.

Examples

The following example converts the value **7** to the VARCHAR value `7`.

```
SELECT CAST (7 AS VARCHAR) "cast" FROM DUMMY;
```

The following example converts the value **`10.5`** to the INTEGER value `10`, truncating the mantissa.

```
SELECT CAST (10.5 AS INTEGER) "cast" FROM DUMMY;
```

6.7.13 CEIL Function (Numeric)

Returns the first integer that is greater than or equal to the specified value.

Syntax

```
CEIL (<n>)
```

Description

Returns the first integer that is greater than or equal to the value of `<n>`.

Example

The following example returns the value `15` for **`"ceiling"`**:

```
SELECT CEIL (14.5) "ceiling" FROM DUMMY;
```

6.7.14 CHAR Function (String)

Returns the character that has the ASCII value of the specified number.

Syntax

```
CHAR (<n>)
```

Description

Returns the character that has the ASCII value of the number `<v>`.

Example

This example converts three ASCII values into characters and concatenates the results, returning the string `Ant`.

```
SELECT CHAR (65) || CHAR (110) || CHAR (116) "character" FROM DUMMY;
```

6.7.15 COALESCE Function (Miscellaneous)

Returns the first non-NULL expression from a specified list.

Syntax

```
COALESCE (expression_list)
```

Description

Returns the first non-NULL expression from a list. At least two expressions must be contained in expression_list, and all expressions must be comparable. The result is NULL if all the arguments are NULL.

Example

```
CREATE TABLE coalesce_example (ID INT PRIMARY KEY, A REAL, B REAL);
 INSERT INTO coalesce_example VALUES(1, 100, 80);
 INSERT INTO coalesce_example VALUES(2, NULL, 63);
 INSERT INTO coalesce_example VALUES(3, NULL, NULL);
 SELECT id, a, b, COALESCE (a, b*1.1, 50.0) "coalesce" FROM coalesce_example;
```

ID	A	B	coalesce
1	100.0	80.0	100.0

ID	A	B	coalesce
2	NULL	63.0	69.30000305175781
3	NULL	NULL	50.0

6.7.16 CONCAT Function (String)

Returns a combined string consisting of two specified strings.

Syntax

```
CONCAT (<str1>, <str2>)
```

Description

Returns a combined string consisting of `<str1>` followed by `<str2>`. The concatenation operator (||) is identical to this function.

The maximum length of the concatenated string is 8,388,607. If a string length is longer than the maximum length, an exception will be thrown. Exceptionally, an implicit truncation is done when converting a `(N)CLOB` typed value with a size greater than the maximum length of a `(N)VARCHAR` typed value.

Example

This example concatenates the specified string arguments and returns the value `Cat`:

```
SELECT CONCAT ('C', 'at') "concat" FROM DUMMY;
```

6.7.17 COS Function (Numeric)

Returns the cosine of the angle, in radians, for the specified argument.

Syntax

```
COS (<n>)
```

Description

Returns the cosine of the angle `<n>`, in radians.

Example

The following example returns the value `1.0` for **`"cos"`**:

```
SELECT COS (0.0) "cos" FROM DUMMY;
```

6.7.18 COT Function (Numeric)

Computes the cotangent of a specified number.

Syntax

```
COT (<n>)
```

Description

Computes the cotangent of a number `<n>`, where `<n>` is an angle expressed in radians.

Example

The following example returns the value `-0.8950829176379128` for `"cot"`:

```
SELECT COT (40) "cot" FROM DUMMY;
```

6.7.19 CURRENT_DATE Function (Datetime)

Returns the current local system date.

Syntax

```
CURRENT_DATE
```

Description

Returns the current local system date.

The usage of local timestamps is discouraged. It is a best practice to use UTC times instead. The use of local times or conversion between local time zones might require additional handling in application code.

Example

The following example returns the value `2010-01-11` for the current local system date:

```
SELECT CURRENT_DATE "current date" FROM DUMMY;
```

6.7.20 CURRENT_IDENTITY_VALUE Function (Miscellaneous)

Returns a BIGINT value representing the latest inserted identity value in the current session.

Syntax Elements

```
CURRENT_IDENTITY_VALUE()
```

Description

Returns a BIGINT value representing the latest inserted identity value in the current session. If no identity value was inserted in the current session, then NULL is returned.

Example

The following example creates the table `test` and inserts a row with the identity 101. The identity value is then read and returned.

```
CREATE COLUMN TABLE test (objectid BIGINT generated by default as identity
(start with 101 increment by 1)
   NOT NULL, col2 integer, primary key(objectid));
INSERT INTO test (col2) VALUES ( 1 );
SELECT CURRENT_IDENTITY_VALUE() "current identity value" FROM test;
```

6.7.21 CURRENT_SCHEMA Function (Miscellaneous)

Returns a string containing the current schema name.

Syntax

```
CURRENT_SCHEMA
```

Description

Returns a string containing the current schema name.

Example

```
SELECT CURRENT_SCHEMA "current schema" FROM DUMMY;

 current schema
 SYSTEM
```

6.7.22 CURRENT_TIME Function (Datetime)

Returns the local system time.

Syntax

```
CURRENT_TIME
```

Description

Returns the current local system time.

The usage of local timestamps is discouraged. It is a best practice to use UTC times instead. The use of local times or conversion between local time zones might require additional handling in application code.

Example

The following example returns the value `17:37:37`, reflect to reflect the current local system time:

```
SELECT CURRENT_TIME "current time" FROM DUMMY;
```

6.7.23 CURRENT_TIMESTAMP Function (Datetime)

Returns the current local system timestamp information.

Syntax

```
CURRENT_TIMESTAMP
```

Description

Returns the current local system timestamp information.

The usage of local timestamps is discouraged. It is a best practice to use UTC times instead. The use of local times or conversion between local time zones might require additional handling in application code.

Example

The following example returns the value `2010-01-11 17:38:48.802` as the current timestamp::

```
SELECT CURRENT_TIMESTAMP "current timestamp" FROM DUMMY;
```

6.7.24 CURRENT_USER Function (Miscellaneous)

Returns the current user name at the current statement context.

Syntax

```
CURRENT_USER
```

Description

Returns the current user name at the current statement context. This is the user name that is currently at the top of authorization stack.

Example

```
-- example showing basic function operation using SYSTEM user
 SELECT CURRENT_USER "current user" FROM DUMMY;

 current user
 SYSTEM
 -- definer-mode procedure declared by USER_A
 CREATE PROCEDURE USER_A.PROC1 LANGUAGE SQLSCRIPT SQL SECURITY DEFINER AS
 BEGIN
     SELECT CURRENT_USER "current user" FROM DUMMY;
 END;

 -- USER_B executing USER_A.PROC1
 CALL USER_A.PROC1;
 current user
 USER_A
 -- invoker-mode procedure declared by USER_A
 CREATE PROCEDURE USER_A.PROC2 LANGUAGE SQLSCRIPT SQL SECURITY INVOKER AS
```

```
BEGIN
    SELECT CURRENT_USER "current user" FROM DUMMY;
END;

-- USER_B is executing USER_A.PROC
CALL USER_A.PROC2;
current user
USER_B
```

6.7.25 DAYNAME Function (Datetime)

Returns the weekday for the specified date.

Syntax

```
DAYNAME (<d>)
```

Description

Returns the weekday in English for date `<d>`.

Example

The following example returns `Monday` as the week day for the specified date:

```
SELECT DAYNAME ('2011-05-30') "dayname" FROM DUMMY;
```

6.7.26 DAYOFMONTH Function (Datetime)

Returns the day of the month for the specified date.

Syntax

```
DAYOFMONTH (<d>)
```

Description

Returns an integer the day of the month for date `<d>`.

Example

The following example returns `30` as the number for the day of the month for the specified date:

```
SELECT DAYOFMONTH ('2011-05-30') "dayofmonth" FROM DUMMY;
```

6.7.27 DAYOFYEAR Function (Datetime)

Returns an integer representation of the day of the year for the specified date.

Syntax

```
DAYOFYEAR (<d>)
```

Description

Returns an integer representation of the day of the year for date `<d>`.

Example

The following example returns the value `150` as the day of the year for the specified date:

```
SELECT DAYOFYEAR ('2011-05-30') "dayofyear" FROM DUMMY;
```

6.7.28 DAYS_BETWEEN Function (Datetime)

Computes the number of days between d1 and d2.

Syntax

```
DAYS_BETWEEN (<d1>, <d2>)
```

Description

Computes the number of days between `<d1>` and `<d2>`.

Example

The following example returns the value `31` for days between the two dates specified:

```
SELECT DAYS_BETWEEN (TO_DATE ('2009-12-05', 'YYYY-MM-DD'), TO_DATE('2010-01-05',
'YYYY-MM-DD')) "days between" FROM DUMMY;
```

6.7.29 EXTRACT Function (Datetime)

Finds and returns the value of a specified datetime field from a specified date.

Syntax

```
EXTRACT ({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND} FROM <d>)
```

Description

Finds and returns the value of a specified datetime field from date `<d>`.

Example

The following example returns the value `2010` for the year extracted from the specified date:

```
SELECT EXTRACT (YEAR FROM TO_DATE ('2010-01-04', 'YYYY-MM-DD')) "extract" FROM
DUMMY;
```

6.7.30 EXP Function (Numeric)

Returns the result of the base of the natural logarithms e raised to the power of the specified argument.

Syntax

```
EXP (<n>)
```

Description

Returns the result of the base of the natural logarithms e raised to the power of the argument `<n>`.

Example

The following example returns the value `2.718281828459045` for **`"exp"`**:

```
SELECT EXP (1.0) "exp" FROM DUMMY;
```

6.7.31 FLOOR Function (Numeric)

Returns the largest integer that is not greater than the specified numeric argument.

Syntax

```
FLOOR (<n>)
```

Description

Returns the largest integer that is not greater than the numeric argument `<n>`.

Example

The following example returns the value `14` for "floor":

```
SELECT FLOOR (14.5) "floor" FROM DUMMY;
```

6.7.32 GREATEST Function (Miscellaneous)

Returns the greatest value among the specified arguments.

> **i Note**
>
> SAP ASE restrictions:
>
> Only the input data types INT, CHAR and VACHAR are supported.

Syntax

```
GREATEST (<argument> [{, <argument>}...])
```

Description

Returns the greatest value among the arguments (n1, n2, ...).

Example

```
SELECT GREATEST ('aa', 'ab', 'ba', 'bb') "greatest" FROM DUMMY;

 greatest
 bb
```

6.7.33 HEXTOBIN Function (String)

Converts a hexadecimal value to a binary value.

Syntax

```
HEXTOBIN (<value>)
```

Description

Converts a hexadecimal value to a binary value.

Example

The following example converts the hexadecimal value **`1a`** to the BINARY value `1A`:

```
SELECT HEXTOBIN ('1a') "hextobin" FROM DUMMY;
```

6.7.34 HOUR Function (Datetime)

Returns an integer representation of the hour for the specified time.

Syntax

```
HOUR (<t>)
```

Description

Returns an integer representation of the hour for time `<t>`.

Example

The following example returns the hour `12`:

```
SELECT HOUR ('12:34:56') "hour" FROM DUMMY;
```

6.7.35 IFNULL Function (Miscellaneous)

Returns the first non-NULL input expression.

Syntax

```
IFNULL (expression1, expression2)
```

Description

Returns the first not NULL input expression.

- Returns expression1 if expression1 is not NULL.
- Returns expression2 if expression1 is NULL.
- Returns NULL if both input expressions are NULL.

Example

The following query returns `diff`:

```
SELECT IFNULL ('diff', 'same') "ifnull" FROM DUMMY
```

The following query returns `same`:

```
SELECT IFNULL (NULL, 'same') "ifnull" FROM DUMMY
```

The following query returns `NULL`:

```
SELECT IFNULL (NULL, NULL) "ifnull" FROM DUMMY
```

6.7.36 LCASE Function (String)

Converts all characters in a string to lowercase.

Syntax

```
LCASE(<str>)
```

Description

Converts all characters in string `<str>` to lowercase.

The `LCASE` function is identical to the `LOWER` function.

Example

This example converts all characters of the given string **`TesT`** to lowercase and returns the value `test`.

```
SELECT LCASE ('TesT') "lcase" FROM DUMMY;
```

6.7.37 LEAST Function (Miscellaneous)

Returns the lesser value of two specified arguments.

> **i Note**
>
> SAP ASE restrictions:
>
> Only the input data types INT, CHAR and VACHAR are supported.

Syntax

```
LEAST (<n1> [, <n2>]...)
```

Description

Returns the lesser value of the arguments (`<n1>`, `<n2>`...).

Example

The following query returns `aa`.

```
SELECT LEAST('aa', 'ab', 'ba', 'bb') "least" FROM DUMMY;
```

6.7.38 LEFT Function (String)

Returns the first characters/bytes from the beginning of a string.

Syntax

```
LEFT (<str>, <n>)
```

Description

Returns the first `<n>` characters/bytes from the beginning of string `<str>`.

Returns an empty string value if `<n>` is less than 1.

Returns string `<str>` without blank padding if the value of `<n>` is greater than the length of string `<str>`.

Example

The following example returns the leftmost **`3`** characters of the given string (**`Hel`**):

```
SELECT LEFT ('Hello', 3) "left" FROM DUMMY;
```

The following example returns the given string (**`Hello`**) because the value **`10`** exceeds the string length:

```
SELECT LEFT ('Hello', 10) "left" FROM DUMMY;
```

6.7.39 LENGTH Function (String)

Returns the number of characters in a string.

Syntax

```
LENGTH(<str>)
```

Description

Returns the number of characters in string `<str>`.

In the case that the type of str is a `VARCHAR`, the `LENGTH(<str>)` will not return the number of bytes. In this case the number of characters like `NVARCHAR`-typed strings are returned instead.

Supplementary plane Unicode characters, each of which occupies 6 bytes in CESU-8 encoding, are counted as two characters.

Example

This example returns the number of characters (`14`) contained in the given string:

```
SELECT LENGTH ('length in char') "length" FROM DUMMY;
```

6.7.40 LN Function (Numeric)

Returns the natural logarithm of the specified argument.

Syntax

```
LN (<n>)
```

Description

Returns the natural logarithm of the numeric argument `<n>`.

Example

This example returns the value `2.1972245773362196` for **`"ln"`**:

```
SELECT LN (9) "ln" FROM DUMMY;
```

6.7.41 LOCATE Function (String)

Returns the position of a substring within a string.

> i Note
>
> SAP ASE restrictions:
>
> Supported input types for `<haystack>` are VARCHAR, UNIVARCHAR, and VARBINARY. TEXT and IMAGE data types are not supported.

Syntax

```
LOCATE (<haystack>, <needle>, <start_position>, <occurrences>)
```

Description

Returns the position of a substring `<needle>` within a string `<haystack>`.

Returns 0 if `<needle>` is not found within `<haystack>`, or if `<occurrences>` is set to less than 1.

Returns NULL if `<haystack>` or `<needle>` is NULL.

Return 0 if `<start_position>` is negative

- If `<start_position>` is not specified or is 0, the search starts at the beginning of the string `<haystack>`.
- If `<occurrences>` is not specified, the first matched position will be returned.

Examples

The following example returns `1` because `<needle>` is an empty string:

```
SELECT locate ('this is a test', '') "locate" from dummy;
 locate
 -----------
           1
```

The following example returns `1` because `<needle>` matches the beginning part of `<haystack>`:

```
SELECT locate ('this is a test', 'this') "locate" from dummy;
locate
 -----------
           1
```

The following example returns `0` because `<needle>` is not found within `<haystack>`:

```
SELECT locate ('this is a test', 'dummy') "locate" from dummy;
 locate
 -----------
          0
```

In the following example, the search starts at the beginning of the string `<haystack>` because the `<start_position>` is not specified:

```
SELECT locate ('this is a test', 'is') "locate" from dummy;
 locate
 -----------
           3
```

In the following example, the second matched position is returned because the search starts at the beginning of the string `<haystack>` (`<start_position>` is specified as a positive value) and the first `<start_position>` number of characters are skipped:

```
SELECT locate ('this is a test', 'is', 5) "locate" from dummy;
 locate
 -----------
           6
```

In the following examples, the second matched position is returned because `<occurrences>` is set to 2:

```
select locate ('this is a test', 'is', 0, 2) "locate" from dummy;
 locate
 -----------
           6
```

```
1> create table test (
2> c1 int,
3> c2 varchar(20) null,
4> c3 int null,
5> c4 int null)
1> insert into test values(1, 'this is a test', NULL, NULL);
(1 row affected)
1> create function f1 (a int, b int default 1)
2> returns x int
3> as
4> begin
5>      x = a + b;
6> end
```

```
1> select locate (c2, 'is', (select count(*) from test)) from test;
 -----------
           3
```

```
1> select locate (c2, 'is', f1(1,2), f1(1,1)) from test;
 -----------
           6
```

6.7.42 LOWER Function (String)

Converts all characters in a string to lowercase.

Syntax

```
LOWER(<str>)
```

Description

Converts all characters in string `<str>` to lowercase.

The LOWER function is identical to the LCASE function.

Example

This example converts the given string **`AnT`** to lowercase, and returns the value `and`:

```
SELECT LOWER ('AnT') "lower" FROM DUMMY;
```

6.7.43 LPAD Function (String)

Pads the start of string with spaces to make a string a specified number of characters in length.

Syntax

```
LPAD (<str>, <n> [, <pattern>])
```

Description

Pads the start of string `<str>` with spaces to make a string of `<n>` characters in length. If the `<pattern>` argument is provided, string `<str>` will be padded using sequences of these characters until the required length is met.

Example

The following example pads the start of string **`end`** with the pattern **`12345`** to make a string of **`15`** characters in length, and returns the value `123451234512end`:

```
SELECT LPAD ('end', 15, '12345') "lpad" FROM DUMMY;
```

6.7.44 LTRIM Function (String)

Returns a string, trimmed of all leading spaces.

i Note

SAP ASE restrictions:

- LTRIM supports a single character for the second parameter.
- SAP ASE treats `<remove_set>` as a single character rather than a set of characters.

Syntax

```
LTRIM (<str> [, <remove_set>])
```

Description

Returns string `<str>`, trimmed of all leading spaces. If `<remove_set>` is specified, LTRIM removes all the characters contained in this set from the start of string `<str>`. This process continues until a character that is not in `<remove_set>` is reached.

`<remove_set>` is treated as a single character.

Example

This example removes all leading **`a`** characters from the given string:

```
SELECT LTRIM ('babababAabend','a') "ltrim" FROM DUMMY;
```

6.7.45 MINUTE Function (Datetime)

Returns an integer representation of the minute for the specified time.

Syntax

```
MINUTE (<t>)
```

Description

Returns an integer representation of the minute for time `<t>`.

Example

The following example returns the value `34` as the minute from the specified time:

```
SELECT MINUTE ('12:34:56') "minute" FROM DUMMY;
```

6.7.46 MOD Function (Numeric)

Returns the remainder of a specified number divided by a specified divisor.

Syntax

```
MOD (<n>, <d>)
```

Description

Returns the remainder of a number `<n>` divided by a divisor `<d>`.

When `<n>` is negative this function acts differently to the standard computational modulo operation.

The following explains example of what the MOD function returns as the result:

- If `<d>` is zero, then `<n>` is returned.
- If `<n>` is greater than 0 and `<n>` is less than `<d>`, then `<n>` is returned.
- If `<n>` is less than 0 and `<n>` is greater than `<d>`, then `<n>` is returned.
- In other case that of those mentioned above, remainder of the absolute value of `<n>` divided by the absolute value of `<d>` is used to calculate remainder. If `<n>` is less than 0, then the returned remainder from MOD is a negative number, and if `<n>` is greater than 0, then the returned remainder from MOD is a positive number.

Example

The following example returns the value `3` for **`"modulus"`**:

```
SELECT MOD (15, 4) "modulus" FROM DUMMY;
```

The following example returns the value `-3` for **`"modulus"`**:

```
SELECT MOD (-15, 4) "modulus" FROM DUMMY;
```

6.7.47 MONTH Function (Datetime)

Returns the number of the month from the specified date.

Syntax

```
MONTH(<d>)
```

Description

Returns the number of the month from date `<d>`.

Example

The following example returns the value `5` as the month for the date specified:

```
SELECT MONTH ('2011-05-30') "month" FROM DUMMY;
```

6.7.48 MONTHNAME Function (Datetime)

Returns the name of the month for the specified date.

Syntax

```
MONTHNAME(<d>)
```

Description

Returns the name of the month in English for date `<d>`.

Example

The following example returns the value `MAY` the month name of the specified date:

```
SELECT MONTHNAME ('2011-05-30') "monthname" FROM DUMMY;
```

6.7.49 NANO100_BETWEEN Function (Datetime)

Computes the time difference between two dates.

Syntax

```
NANO100_BETWEEN (<d1>, <d2>)
```

Description

Computes the time difference between date arguments `<d1>` and `<d2>`, to the precision of 0.1 microseconds.

Example

The following example returns `864000000000` as the time difference between the two specified dates:

```
SELECT NANO100_BETWEEN ('2013-01-30', '2013-01-31') "nano100 between" FROM DUMMY;
```

6.7.50 NEXT_DAY Function (Datetime)

Returns the date of the next day after the specified date.

Syntax

```
NEXT_DAY (<d>)
```

Description

Returns the date of the next day after date `<d>`.

Example

The following example returns `2010-01-01` as the next day after the specified date:

```
SELECT NEXT_DAY (TO_DATE ('2009-12-31', 'YYYY-MM-DD')) "next day" FROM DUMMY;
```

6.7.51 NOW Function (Datetime)

Returns the current timestamp.

Syntax

```
NOW ()
```

Description

Returns the current timestamp.

Example

The following example returns the value `2010-01-01 16:34:19.894` for the current timestamp:

```
SELECT NOW () "now" FROM DUMMY;
```

6.7.52 NULLIF Function (Miscellaneous)

Determines whether two expressions are equal.

Syntax

```
NULLIF (expression1, expression2)
```

Description

NULLIF compares the values of two expressions and returns NULL if the first expression equals the second expression.

- If expression1 does not equal expression2, NULLIF returns expression1.
- If expression2 is NULL, NULLIF returns expression1.

Example

The following query returns `diff`.

```
SELECT NULLIF ('diff', 'same') "nullif" FROM DUMMY;
```

The following query returns `NULL`.

```
SELECT NULLIF('same', 'same') "nullif" FROM DUMMY;
```

6.7.53 POWER Function (Numeric)

Calculates a specified base number raised to the power of a specified exponent.

Syntax

```
POWER (<b>, <e>)
```

Description

Calculates the base number `<b>` raised to the power of an exponent `<e>`.

Example

The following example returns the value `1024.0` for **`"power"`**:

```
SELECT POWER (2, 10) "power" FROM DUMMY;
```

6.7.54 RAND Function (Numeric)

Returns a pseudo-random value in the range of 0 to less than 1.0.

Syntax

```
RAND()
```

Description

Returns a value type of DOUBLE.

The resulting values are not safe for cryptographic or security purposes.

Example

The following example returns the pseudo-random DOUBLE value `0.34130480715134404`:

```
SELECT RAND() FROM DUMMY;
```

6.7.55 REPLACE Function (String)

Searches a string for all occurrences of a specified string and replaces them with another specified string.

Syntax

```
REPLACE (<original_string>, <search_string>, <replace_string>)
```

Description

Searches in `<original_string>` for all occurrences of `<search_string>` and replaces them with `<replace_string>`.

- If `<original_string>` is an empty string, then the result will be an empty string.
- If two overlapping substrings match the `<search_string>` in the `<original_string>`, then only the first occurrence will be replaced.
- If `<original_string>` does not contain any occurrence of `<search_string>`, then `<original_string>` will be returned unchanged.
- If `<original_string>`, `<search_string>`, or `<replace_string>` are NULL then NULL is returned.

Example

The following example changes all occurences of **`DOWN`** in the given string to **`UP`** and returns the value `UPGRADE UPWARD`:

```
SELECT REPLACE ('DOWNGRADE DOWNWARD','DOWN', 'UP') "replace" FROM DUMMY;
```

6.7.56 RIGHT Function (String)

Returns the rightmost specified characters or bytes of a string.

Syntax

```
RIGHT(<str>, <n>)
```

Description

Returns the rightmost `<n>` characters/bytes of string `<str>`.

Returns an empty string value of `<n>` is less than 1.

Returns string `<str>` without blank padding if the value of `<n>` is greater than the length of string `<str>`.

Example

The following example returns the rightmost **3** characters of the given string (**789**):

```
SELECT RIGHT('HI0123456789', 3) "right" FROM DUMMY;
```

The following example returns the entire specified string because the value **20** exceeds the string length:

```
SELECT RIGHT('HI0123456789', 20) "right" FROM DUMMY;
```

6.7.57 ROUND Function (Numeric)

Rounds the specified argument to the specified amount of places after the decimal point.

i Note

SAP ASE restrictions:

- The `<rounding_mode>` parameter is not supported.

Syntax

```
ROUND (<n> [, <pos>])
```

Syntax Elements

n

Specifies the numeric argument to round.

pos

Specifies the amount of places after the decimal point to round `<n>` to.

Description

Rounds argument `<n>` to the specified amount of places (`<pos>`) after the decimal point.

When using ROUND with floating point types REAL and DOUBLE, the precision of the numeric representation can affect the result. In this case, the actual number of digits after the decimal point is influenced by the precision of the type used. For example, the result of the following statement is is rounded up to the next round figure. If the value is precisely in halfway between two rounded values, it is rounded up away from zero (commercial rounding). `399.710000` not 399.71:

```
SELECT ROUND(TO_REAL(399.71429443359375),2) from DUMMY;
```

Examples

The following example returns the value `16.2`:

```
SELECT ROUND (16.16, 1) "round" FROM DUMMY;
```

The following example returns the value `20` :

```
SELECT ROUND (16.16, -1) "round" FROM DUMMY;
```

6.7.58 RPAD Function (String)

Pads the end of a string with spaces to make a string a specified number of characters in length.

Syntax

```
RPAD (<str>, <n> [, <pattern>])
```

Description

Pads the end of string `<str>` with spaces to make a string of `<n>` characters in length. If the pattern argument is provided, the string `<str>` will be padded using sequences of the given characters until the required length is met.

Example

The following example pads the end of string **`end`** with the pattern **`12345`** to make a string of **`15`** characters in length and returns the value `end123451234512`:

```
SELECT RPAD ('end', 15, '12345') "right padded" FROM DUMMY;
```

6.7.59 RTRIM Function (String)

Returns a string trimmed of all trailing spaces.

> i Note
>
> SAP ASE restrictions:
>
> - RTRIM supports a single character for the second parameter.
> - SAP ASE treats `<remove_set>` as a single character rather than a set of characters.

Syntax

```
RTRIM (<str> [,<remove_set> ])
```

Description

Returns a string trimmed of all trailing spaces. If `<remove_set>` is specified, then `RTRIM` removes all the characters contained in the specified set from the end of the string. This process continues until a character that is not in the `<remove_set>` has been reached.

`<remove_set>` is treated as a single character.

Example

This example removes all trailing **`a`** and characters from the given string:

```
SELECT RTRIM ('endabAabbabab','a') "rtrim" FROM DUMMY;
```

6.7.60 SECOND Function (Datetime)

Returns a value of the seconds for a given time.

Syntax

```
SECOND (<t>)
```

Description

Returns a value of the seconds for a given time.

Subseconds are included for TIMESTAMP datatypes.

Examples

The following example returns the value `56` for the second of the specified time:

```
SELECT SECOND ('12:34:56') "second" FROM DUMMY;
```

The following example returns the value `56.789` for the subseconds of the specified time:

```
SELECT SECOND ('2014-03-25 12:34:56.789') "subseconds" FROM DUMMY;
```

6.7.61 SECONDS_BETWEEN Function (Datetime)

Computes the number of seconds between two specified dates.

Syntax

```
SECONDS_BETWEEN (<d1>, <d2>)
```

Description

Computes the number of seconds between date arguments `<d1>` and `<d2>`, which is semantically equal to (`<d2>` - `<d1>`).

Example

The following example returns the value `2678400` as the seconds between the two specified dates:

```
SELECT SECONDS_BETWEEN ('2009-12-05', '2010-01-05') "seconds between" FROM DUMMY;
```

6.7.62 SESSION_CONTEXT Function (Miscellaneous)

Returns the value of session_variable assigned to the current user.

Syntax

```
SESSION_CONTEXT(<session_variable>)
```

Description

The `<session_variable>` accessed can either be predefined or user-defined. Predefined session variables that can be set by the client are APPLICATION, APPLICATIONUSER, and TRACEPROFILE.

Session variables can be defined or modified using SET `<variable_name>` = `<value>` command, and unset using UNSET `<variable_name>`.

SESSION_CONTEXT returns an NVARCHAR with a maximum length of 512 characters.

Example

The following query returns the value `HDBStudio`:

```
SELECT SESSION_CONTEXT('APPLICATION') "session context" FROM DUMMY;
```

6.7.63 SESSION_USER Function (Miscellaneous)

Returns the user name of the current session.

Syntax

```
SESSION_USER
```

Description

Returns the user name of the current session.

Example

The following query returns `SYSTEM`:

```
SELECT SESSION_USER "session user" FROM DUMMY;
```

Consider the following definer-mode procedure that is declared by USER_A:

```
CREATE PROCEDURE USER_A.PROC1 LANGUAGE SQLSCRIPT SQL SECURITY DEFINER AS
 BEGIN
     SELECT SESSION_USER "session user" FROM DUMMY;
 END;
```

The following query returns `USER_B` when USER_B executes USER_A.PROC:

```
CALL USER_A.PROC1;
```

Consider the following invoker-mode procedure that is declared by USER_A:

```
CREATE PROCEDURE USER_A.PROC2 LANGUAGE SQLSCRIPT SQL SECURITY INVOKER AS
 BEGIN
     SELECT SESSION_USER "session user" FROM DUMMY;
 END;
```

The following query returns `USER_B` when USER_B executes USER_A.PROC:

```
CALL USER_A.PROC2;
```

6.7.64 SIGN Function (Numeric)

Returns the sign (positive or negative) of the specified numeric argument.

Syntax

```
SIGN (<n>)
```

Description

Returns the sign (positive or negative) of the numeric argument `<n>`.

Returns 1 if `<n>` is a positive value, -1 if `<n>` is a negative value, 0 if `<n>` is equal to zero, and NULL if `<n>` is equal to NULL.

Example

The following example returns the value `-1` for **`"sign"`**:

```
SELECT SIGN (-15) "sign" FROM DUMMY;
```

6.7.65 SIN Function (Numeric)

Returns the sine of an angle expressed in radians.

Syntax

```
SIN (<n>)
```

Description

Returns the sine of `<n>`, where `<n>` is an angle expressed in radians.

Example

The following example returns the value `1.0` for `"sine"`:

```
SELECT SIN ( 3.141592653589793/2) "sine" FROM DUMMY;
```

6.7.66 SQRT Function (Numeric)

Returns the square root of the specified argument.

Syntax

```
SQRT (n)
```

Description

Returns the square root of the numeric argument `<n>`.

Example

The following example returns the value `1.4142135623730951` for "sqrt":

```
SELECT SQRT (2) "sqrt" FROM DUMMY;
```

6.7.67 STDDEV_POP Function (Aggregate)

Returns the standard deviation of a given expression as the square root of VAR_POP function.

Syntax

```
STDDEV_POP(<expression>)
```

Description

Returns the standard deviation of the given expression as the square root of VAR_POP function.

Examples

The following examples returns `0` for the standard deviation of the specified expression:

```
CREATE ROW TABLE RTABLE (A INT);
INSERT INTO RTABLE VALUES (1);
SELECT STDDEV_POP(A) "STDDEVPOP" FROM RTABLE;
```

The following examples returns `0.5` for the standard deviation of the specified expression:

```
INSERT INTO RTABLE VALUES (2);
SELECT STDDEV_POP(A) "STDDEVPOP" FROM RTABLE;
```

6.7.68 STDDEV_SAMP Function (Aggregate)

Returns the standard deviation of the given expression as the square root of VAR_SAMP function.

Syntax

```
STDDEV_SAMP(<expression>)
```

Description

Returns the standard deviation of the given expression as the square root of VAR_SAMP function.

Examples

The following example returns `NULL` for the standard deviation of the specified expression, as the square root of VAR_SAMP function:

```
CREATE ROW TABLE RTABLE (A INT);
INSERT INTO RTABLE VALUES (1);
SELECT STDDEV_SAMP(A) "STDDEVSAMP" FROM RTABLE;
```

The following example returns `0.707107` for the standard deviation of the specified expression, as the square root of VAR_SAMP function:

```
INSERT INTO RTABLE VALUES (2);
SELECT STDDEV_SAMP(A) "STDDEVSAMP" FROM RTABLE;
```

6.7.69 SUBSTRING Function (String)

Returns a substring of a specified string starting from a specified position within the string.

Syntax

```
SUBSTRING (<str>, <start_position> [, <string_length>])
```

Description

Returns a substring of string `<str>` starting from `<start_position>` within the string. SUBSTRING can return the remaining part of a string from the `<start_position>` or, optionally, a number of characters set by the `<string_length>` parameter:

- If `<start_position>` is less than 0, then it is considered to be 1.
- If `<string_length>` is less than 1, then an empty string is returned.
- If `<string_length>` is greater than the length of remaining part of `<str>`, then the remaining part is returned without blank padding.

Example

The following example selects two characters from the string **`1234567890`** starting at position **`4`**, and returns the value `45`:

```
SELECT SUBSTRING ('1234567890',4,2) "substring" FROM DUMMY;
```

6.7.70 TAN Function (Numeric)

Returns the tangent of a specified number, where the argument is an angle expressed in radians.

Syntax

```
TAN (<n>)
```

Description

Returns the tangent of `<n>`, where `<n>` is an angle expressed in radians.

Example

The following example returns the value `0.0` for **`"tan"`**:

```
SELECT TAN (0.0) "tan" FROM DUMMY;
```

6.7.71 TO_ALPHANUM Function (Data Type Conversion)

Converts a given value to an ALPHANUM data type.

Syntax

```
TO_ALPHANUM (<value>)
```

Description

Converts a given value to an ALPHANUM data type.

Example

The following example converts the value **10** to the ALPHANUM value `10`.

```
SELECT TO_ALPHANUM ('10') "to alphanum" FROM DUMMY;
```

6.7.72 TO_BIGINT Function (Data Type Conversion)

Converts a value to a BIGINT data type.

Syntax

```
TO_BIGINT (<value>)
```

Description

Converts a value to a BIGINT data type.

If the input value has a mantissa, these digits are truncated during the conversion process.

Examples

The following example converts the value **10** to a BIGINT value `10`:

```
SELECT TO_BIGINT ('10') "to bigint" FROM DUMMY;
```

The following example converts the value **10** to a BIGINT value `10`, truncating the mantissa:

```
SELECT TO_BIGINT (10.5) "to bigint" FROM DUMMY;
```

6.7.73 TO_BINARY Function (Data Type Conversion)

Converts a value to a BINARY data type.

Syntax

```
TO_BINARY (<value>)
```

Description

Converts a value to a BINARY data type.

Example

The following example converts the value **abc** to the BINARY value `616263`.

```
SELECT TO_BINARY ('abc') "to binary" FROM DUMMY;
```

6.7.74 TO_BLOB Function (Data Type Conversion)

Converts a binary string to a BLOB data type.

Syntax

```
TO_BLOB (<value>)
```

Description

Converts a value to a BLOB data type. `<value>` must be a binary string.

Example

The following example converts the value **`abcde`** to the BLOB value `abcde`:

```
SELECT TO_BLOB (TO_BINARY('abcde')) "to blob" FROM DUMMY;
```

6.7.75 TO_CLOB Function (Data Type Conversion)

Converts a value to a CLOB data type.

Syntax

```
TO_CLOB (<value>)
```

Description

Converts a value to a CLOB data type.

Example

The following example converts the value **`TO_CLOB converts the value to a CLOB data type`**, to the CLOB value `TO_CLOB converts the value to a CLOB data type`.

```
SELECT TO_CLOB ('TO_CLOB converts the value to a CLOB data type') "to clob" FROM
DUMMY;
```

6.7.76 TO_DATE Function (Data Type Conversion)

Converts a date string into a DATE data type.

Syntax

```
TO_DATE (<d> [, <format>])
```

Description

Converts the date string `<d>` into a DATE data type.

If the `<format>` specifier is omitted, the conversion is performed using the date format model.

SAP ASE SQLScript supports these formats:

- YYYY-MM-DD
- YYYY/MM/DD
- YYYYMMDD
- YYYY/MM-DD
- YYYY-MM/DD
- YYYY-MON-DD
- YYYY/MON/DD
- YYYYMONDD
- YYYY/MON-DD
- YYYY-MON/DD
- YYYY-MONTH-DD
- YYYY/MONTH/DD
- YYYYMONTHDD
- YYYY/MONTH-DD
- YYYY-MONTH/DD
- YYYY-RM-DD
- YYYY/RM/DD
- YYYYRMDD
- YYYY/RM-DD
- YYYY-RM/DD

Example

The following example converts the string **`2010-01-12`** to a DATE value with the format **`YYYY-MM-DD`**, and returns the value `2010-01-12` (or another format like `Jan 12, 2010`, depending on your date display settings):

```
SELECT TO_DATE('2010-01-12', 'YYYY-MM-DD') "to date" FROM DUMMY;
```

6.7.77 TO_DATS Function (Data Type Conversion)

Converts a date string into an ABAP DATE string.

Syntax

```
TO_DATS (<d>)
```

Description

Converts a date string d into an ABAP DATE string with format 'YYYYMMDD'.

Example

The following example converts the value **`2010-01-12`** the ABAP DATE string `20100112`:

```
SELECT TO_DATS ('2010-01-12') "abap date" FROM DUMMY;
```

6.7.78 TO_DOUBLE Function (Data Type Conversion)

Converts a value to a DOUBLE data type.

Syntax

```
TO_DOUBLE (<value>)
```

Description

Converts a specified value to a DOUBLE (double precision) data type.

Example

The following example multiplies the value **15.12** by **3** and converts the result to a DOUBLE, returning the value 45.36:

```
SELECT 3*TO_DOUBLE ('15.12') "to double" FROM DUMMY;
```

6.7.79 TO_FIXEDCHAR Function (Data Type Conversion)

Converts a specified number of characters of a string starting at the first character in the string.

Syntax

```
TO_FIXEDCHAR (<string>, <size>)
```

Description

Converts the specified string to a CHAR value of fixed size as specified by `<size>`, starting at the first character. `<size>` cannot be a variable.

Example

The following example converts the value **Ant** to a CHAR of length **2**, and returns the value An.

```
SELECT TO_FIXEDCHAR ('Ant', 2) "to_fixedchar" FROM DUMMY;
```

6.7.80 TO_INT Function (Data Type Conversion)

Converts a value to an INT data type.

Syntax

```
TO_INT (<value>)
```

Description

Converts a value to an INT data type.

If the input value has a mantissa, the mantissa is truncated during the conversion process.

Examples

The following example converts the value **`10`** to the INT value `10`:

```
SELECT TO_INT('10') "to int" FROM DUMMY;
```

The following example converts the value **`10.5`** to the INT value `10`, truncating the mantissa.

```
SELECT TO_INT(10.5) "to int" FROM DUMMY;
```

6.7.81 TO_INTEGER Function (Data Type Conversion)

Converts the value to an INTEGER data type.

Syntax

```
TO_INTEGER (<value>)
```

Description

Converts the value to an INTEGER data type.

If the input value has a mantissa, these digits are truncated during the conversion process.

Examples

The following example converts the value **`10`** to the INTEGER value `10`:

```
SELECT TO_INTEGER ('10') "to int" FROM DUMMY;
```

The following example converts the value **`10.5`** to the INTEGER value `10`, truncating the mantissa:

```
SELECT TO_INTEGER(10.5) "to int" FROM DUMMY;
```

6.7.82 TO_NCLOB Function (Data Type Function)

Converts a value to a NCLOB data type.

Syntax

```
TO_NCLOB (<value>)
```

Description

Converts a value to a NCLOB data type.

Example

The following example converts the value **`TO_NCLOB converts the value to a NCLOB data type`** to the NCLOB value `TO_NCLOB converts the value to a NCLOB data type`.

```
SELECT TO_NCLOB ('TO_NCLOB converts the value to a NCLOB data type') "to nclob"
FROM DUMMY;
```

6.7.83 TO_NVARCHAR Function (Data Type Conversion)

Converts the specified value to a NVARCHAR unicode character data type.

i Note

SAP ASE restrictions:

- The `<format>` parameter is not supported.

Syntax

```
TO_NVARCHAR (<value>)
```

Description

Converts the value to a NVARCHAR unicode character data type.

The following data types can be converted to NVARCHAR using the TO_NVARCHAR function:

- ALPHANUM, BIGINT, DATE, DECIMAL, DOUBLE, FIXED12, FIXED16, FIXED8, INTEGER, REAL, SECONDDATE, SMALLDECIMAL, SMALLINT, TIME, TIMESTAMP, TINYINT, VARBINARY, VARCHAR.
- CLOB, NCLOB, TEXT (an exception is thrown if the value is longer than maximum length of NVARCHAR).

Example

The following example coverts the value **`2009/12/31`** to the NVARCHAR value `09-12-31`:

```
SELECT TO_NVARCHAR(TO_DATE('2009/12/31')) "to nvarchar" FROM DUMMY;
```

6.7.84 TO_REAL Function (Data Type Conversion)

Converts a value to a REAL data type.

Syntax

```
TO_REAL (<value>)
```

Description

Converts a value to a REAL (single precision) data type.

Example

The following converts the value `15.12` to a REAL value, and multiplies it by `3` to return the value `45.36000061035156`.

```
SELECT 3*TO_REAL ('15.12') "to real" FROM DUMMY;
```

6.7.85 TO_SECONDDATE Function (Data Type Conversion)

Converts a date string `<d>` into a SECONDDATE data type.

i Note

SAP ASE restrictions:

- The FORMAT option is not supported.

Syntax

```
TO_SECONDDATE (<d>)
```

Description

Converts a date string `<d>` into a SECONDDATE data type.

Example

The following example converts the value **`2010-01-11 13:30:00`** to a SECONDDATE data type and returns the value `2010-01-11 13:30:00.0`:

```
SELECT TO_SECONDDATE ('2010-01-11 13:30:00') "to seconddate" FROM DUMMY;
```

6.7.86 TO_SMALLINT Function (Data Type Conversion)

Converts a value to a SMALLINT data type.

Syntax

```
TO_SMALLINT (<value>)
```

Description

Converts a value to a SMALLINT data type.

If the input value has a mantissa, these digits are truncated during the conversion process.

Examples

The following example converts the value **10** to a SMALLINT and returns the value 10:

```
SELECT TO_SMALLINT ('10') "to smallint" FROM DUMMY;
```

The following example converts the value **10.5** to a SMALLINT and returns the value 10, truncating the mantissa:

```
SELECT TO_SMALLINT(10.5) "to smallint" FROM DUMMY;
```

6.7.87 TO_TIME Function (Data Type Conversion)

Converts a time string into a TIME data type.

i Note

SAP ASE restrictions:

- The FORMAT option is not supported.

Syntax

```
TO_TIME (<t>)
```

Description

Converts a time string `<t>` into the TIME data type.

Example

The following example converts the value **`08:30 AM`** to a TIME value and returns the value `08:30:00`:

```
SELECT TO_TIME ('08:30 AM') "to time" FROM DUMMY;
```

6.7.88 TO_TIMESTAMP Function (Data Type Conversion)

Converts a date string to a TIMESTAMP data type.

Syntax

```
TO_TIMESTAMP (<d> [, <format>])
```

Description

Converts date string `<d>` to the TIMESTAMP data type.

If the `<format>` specifier is omitted, then the conversion is performed using the date format model.

SAP ASE SQLScript supports these formats:

- YYYY-MM-DD HH24:MI:SS
- YYYY/MM/DD HH24:MI:SS
- YYYYMMDD HH24:MI:SS
- YYYY/MM-DD HH24:MI:SS
- YYYY-MM/DD HH24:MI:SS
- YYYY-MON-DD HH24:MI:SS
- YYYY/MON/DD HH24:MI:SS
- YYYYMONDD HH24:MI:SS
- YYYY/MON-DD HH24:MI:SS
- YYYY-MON/DD HH24:MI:SS
- YYYY-MONTH-DD HH24:MI:SS
- YYYY/MONTH/DD HH24:MI:SS

- YYYYMONTHDD HH24:MI:SS
- YYYY/MONTH-DD HH24:MI:SS
- YYYY-MONTH/DD HH24:MI:SS
- YYYY-RM-DD HH24:MI:SS
- YYYY/RM/DD HH24:MI:SS
- YYYYRMDD HH24:MI:SS
- YYYY/RM-DD HH24:MI:SS
- YYYY-RM/DD HH24:MI:SS

Example

The following example converts the value **`2010-01-11 13:30:00`** to the TIMESTAMP value `2010-01-11 13:30:00.0` using the format **`YYYY-MM-DD HH24:MI:SS`**:

```
SELECT TO_TIMESTAMP ('2010-01-11 13:30:00', 'YYYY-MM-DD HH24:MI:SS') "to
timestamp" FROM DUMMY;
```

6.7.89 TO_TINYINT Function (Data Type Conversion)

Converts a value to a TINYINT data type.

Syntax

```
TO_TINYINT (<value>)
```

Description

Converts the value to a TINYINT data type.

If the input value has a mantissa, these digits are truncated during the conversion process.

Examples

The following example converts the value **`10`** to the TINYINT value `10`.

```
SELECT TO_TINYINT ('10') "to tinyint" FROM DUMMY;
```

The following example converts the value **10.5** to the TINYINT value 10, truncating the mantissa.

```
SELECT TO_TINYINT(10.5) "to tinyint" FROM DUMMY;
```

6.7.90 TO_VARCHAR Function (Data Type Conversion)

Converts a given value to a VARCHAR character data type.

> i Note
>
> SAP ASE restrictions:
>
> - The FORMAT option is not supported.

Syntax

```
TO_VARCHAR (<value>)
```

Description

Converts a given value to a VARCHAR character data type.

Example

The following example converts the value **2009-12-31** to a date value and then converts it again to a VARCHAR type, and returns the value 2009/12/31:

```
SELECT TO_VARCHAR (TO_DATE('2009-12-31')) "to char" FROM DUMMY;
```

6.7.91 TRIM Function (String)

Returns a string after removing leading and trailing spaces.

Syntax

```
TRIM ([[LEADING | TRAILING | BOTH] <trim_char> FROM] <str> )
```

Description

Returns string `<str>` after removing leading and trailing spaces. The trimming operation is carried out either from the start (`LEADING`), end (`TRAILING`) or both (`BOTH`) ends of string `<str>`.

- If either `<str>` or `<trim_char>` are a null values, then a `NULL` is returned.
- If no options are specified, `TRIM` removes both the leading and trailing substring `<trim_char>` from string `<str>`.
- If you do not specify `<trim_char>`, then a single blank space is used.

Example

The following example removes the character **a** both at the beginning and the end of the specified string and returns the value `123456789`:

```
SELECT TRIM ('a' FROM 'aaa123456789aa') "trim both" FROM DUMMY;
```

The following example removes the character **a** at the begin of the specified string, and returns the value `123456789aa`:

```
SELECT TRIM (LEADING 'a' FROM 'aaa123456789aa') "trim leading" FROM DUMMY;
```

6.7.92 UCASE Function (String)

Converts all characters in the specified string to uppercase.

Syntax

```
UCASE (<str>)
```

Description

Converts all characters in string `<str>` to uppercase.

The UCASE function is identical to the UPPER function.

Example

This example converts the given string to uppercase, and returns the value `ANT`:

```
SELECT UCASE ('Ant') "ucase" FROM DUMMY;
```

6.7.93 UPPER Function (String)

Converts all characters in a string to uppercase.

Syntax

```
UPPER (<str>)
```

Description

Converts all characters in string `<str>` to uppercase.

The UPPER function is identical to the UCASE function.

Example

This example converts the given string **`Ant`** to uppercase and returns the value `ANT`:

```
SELECT UPPER ('Ant') "uppercase" FROM DUMMY;
```

6.7.94 VAR_POP Function (Aggregate)

Returns the population variance of an expression.

Syntax

```
VAR_POP(<expression>)
```

Description

Returns the population variance of the expression as the sum of squares of the difference of `<expression>` from the mean of `<expression>`, divided by the number of rows remaining.

Examples

The following example returns `0` as the population variance for the specified expression:

```
CREATE ROW TABLE RTABLE (A INT);
INSERT INTO RTABLE VALUES (1);
SELECT VAR_POP(A) "VARPOP" FROM RTABLE;
```

The following example returns `0.25` as the population variance for the specified expression:

```
INSERT INTO RTABLE VALUES (2);
SELECT VAR_POP(A) "VARPOP" FROM RTABLE;
```

6.7.95 VAR_SAMP Function (Aggregate)

Returns the sample variance of an expression.

Syntax

```
VAR_SAMP(<expression>)
```

Description

Returns the sample variance of the expression as the sum of squares of the difference of `<expression>` from the mean of `<expression>`, divided by the number of rows remaining minus 1 (one). This functions is similar to VAR, the only difference is that it returns NULL when the number of rows is 1.

Examples

The following example returns `NULL` as the sample variance of the specified expression:

```
CREATE ROW TABLE RTABLE (A INT);
INSERT INTO RTABLE VALUES (1);SELECT VAR_SAMP(A)
"VARSAMP" FROM RTABLE;
```

The following example returns 0.5 as the sample variance of the specified expression:

```
INSERT INTO RTABLE VALUES (2);
SELECT VAR_SAMP(A) "VARSAMP" FROM RTABLE;
```

6.7.96 WEEK Function (Datetime)

Returns the week number of the specified date.

Syntax

```
WEEK (<d>)
```

Description

Returns the week number of date `<d>`.

Example

The following example returns the value `23` for the week number of the specified date:

```
SELECT  WEEK (TO_DATE('2011-05-30', 'YYYY-MM-DD')) "week" FROM DUMMY;
```

6.7.97 WEEKDAY Function (Datetime)

Returns the day of the week for the specified date.

Syntax

```
WEEKDAY (<d>)
```

Description

Returns an integer representation of the day of the week for date `<d>`. The return value ranges from 0 to 6, representing Monday(0) through to Sunday(6).

Example

The following example returns the value `4` for as the week day of the specified date:

```
SELECT WEEKDAY (TO_DATE ('2010-12-31', 'YYYY-MM-DD')) "week day" FROM DUMMY;
```

6.7.98 YEAR Function (Datetime)

Returns the year number of a specified date.

Syntax

```
YEAR (<d>)
```

Description

Returns the year number of date `<d>`.

Example

The following example returns the value `2011` for the year of the specified date:

```
SELECT YEAR (TO_DATE ('2011-05-30', 'YYYY-MM-DD')) "year" FROM DUMMY;
```

7 SAP ASE SQLScript Statements

SAP ASE supports SQL extensions outside a stored procedure.

SQLScript Statement Support:

SAP ASE SQLScript supports:

- Access Control Statements [page 137]
- Data Definition Statements [page 138]
- Data Manipulation Statements [page 139]
- Procedural Statements [page 140]
- Session Management Statements [page 140]
- Transaction Management Statements [page 141]

However, SAP ASE SQLScript does not support data import and data export statement.

7.1 Access Control Statements

Access control statements enable database administrators to create, alter, and drop access to the database.

These SQLScript statements are fully supported in the SAP ASE SQLScript database:

- DROP ROLE Statement (Access Control) [page 221]
- DROP USER Statement (Access Control) [page 230]

These SQLScript statements have limited support in the SAP ASE SQLScript database:

- ALTER USER Statement (Access Control) [page 163]
- CREATE ROLE Statement (Access Control) [page 185]
- CREATE USER Statement (Access Control) [page 211]
- GRANT Statement (Access Control) [page 232]
- REVOKE Statement (Access Control) [page 246]

These SQLScript statements are not supported in the SAP ASE SQLScript database:

- `ALTER AUDIT POLICY`
- `ALTER CREDENTIAL`
- `ALTER REMOTE SOURCE`
- `ALTER ROLE`
- `ALTER SAML PROVIDER`
- `CREATE AUDIT POLICY`

- `CREATE CREDENTIAL`
- `CREATE REMOTE SOURCE`
- `CREATE SAML PROVIDER`
- `DROP AUDIT POLICY`
- `DROP CREDENTIAL`
- `DROP REMOTE SOURCE`
- `DROP SAML PROVIDER`

7.2 Data Definition Statements

Data definition statements define structures.

SQLScript Supported Statements:

These SQLScript statements are fully supported in the SAP ASE SQLScript database:

- CREATE SCHEMA Statement (Data Definition) [page 186]
- CREATE SEQUENCE Statement (Data Definition) [page 187]
- DROP INDEX Statement (Data Definition) [page 219]
- DROP SCHEMA Statement (Data Definition) [page 222]
- DROP SEQUENCE Statement (Data Definition) [page 223]
- DROP TABLE Statement (Data Definition) [page 227]
- DROP TRIGGER Statement (Data Definition) [page 228]
- DROP VIEW Statement (Data Definition) [page 231]

SQLScript Supported Statements with Restrictions:

These SQLScript statements have limited support in the SAP ASE SQLScript database:

- ALTER SEQUENCE Statement (Data Definition) [page 142]
- ALTER TABLE Statement (Data Definition) [page 145]
- CREATE INDEX Statement (Data Definition) [page 175]
- CREATE TABLE Statement (Data Definition) [page 191]
- CREATE TRIGGER Statement (Data Definition) [page 199]
- REFRESH STATISTICS Statement (Data Definition) [page 238]
- CREATE VIEW Statement (Data Definition) [page 212]
- RENAME COLUMN Statement (Data Definition) [page 240]
- RENAME TABLE Statement (Data Definition) [page 241]

SQLScript Statements that are Not Supported:

These SQLScript statements are not supported in the SAP ASE SQLScript database:

- `ALTER FULLTEXT INDEX`
- `ALTER INDEX`
- `ALTER STATISTICS`
- `ALTER VIEW`
- `ALTER VIRTUAL TABLE`
- `COMMENT ON`
- `CREATE GRAPH WORKSPACE`
- `CREATE FULLTEXT INDEX`
- `CREATE STATISTICS`
- `CREATE SYNONYM`
- `CREATE VIRTUAL TABLE`
- `DROP FULLTEXT INDEX`
- `DROP GRAPH WORKSPACE`
- `DROP STATISTICS`
- `DROP SYNONYM`
- `RENAME INDEX`

7.3 Data Manipulation Statements

The following statements enable you to manipulate data within the database.

This SQLScript statement is fully supported in the SAP ASE SQLScript database:

- TRUNCATE TABLE Statement (Data Manipulation) [page 262]

This SQLScript statements have limited support in the SAP ASE SQLScript database:

- DELETE Statement (Data Manipulation) [page 214]
- INSERT Statement (Data Manipulation) [page 236]
- SELECT Statement (Data Manipulation) [page 249]
- UPDATE Statement (Data Manipulation) [page 264]
- REPLACE / UPSERT Statement (Data Manipulation) [page 243]

These SQLScript statements are not supported in the SAP ASE SQLScript database:

- `EXPLAIN PLAN`
- `LOAD`
- `MERGE INTO`
- `UNLOAD`

7.4 Procedural Statements

Procedural statements manage system and user-defined procedures for the database.

These SQLScript statements are fully supported in the SAP ASE SQLScript database:

- CREATE TYPE Statement (Procedural) [page 210]
- DROP FUNCTION Statement (Procedural) [page 218]
- DROP TYPE Statement (Procedural) [page 229]
- DROP PROCEDURE Statement (Procedural) [page 220]

These SQLScript statements have limited support in the SAP ASE SQLScript database:

- ALTER PROCEDURE Statement (Procedural) [page 141]
- CREATE FUNCTION Statement (Procedural) [page 168]
- CALL Statement (Procedural) [page 164]
- DO BEGIN ... END Statement (Procedural) [page 216]
- CREATE PROCEDURE Statement (Procedural) [page 176]
- SET Statement (Procedural) [page 257]

These SQLScript statements are not supported in the SAP ASE SQLScript database:

- `ALTER FUNCTION`
- `CREATE VIRTUAL FUNCTION`
- `CREATE VIRTUAL PRCEDURE`

7.5 Session Management Statements

The following SQL statements manage database sessions.

These SQLScript statements are fully supported in the SAP ASE SQLScript database:

- SET [SESSION] Statement (Session Management) [page 259]
- SET SCHEMA Statement (Session Management) [page 258]
- UNSET [SESSION] Statement (Session Management) [page 263]

These SQLScript statements are not supported in the SAP ASE SQLScript database:

- `CONNECT`
- `SET HISTORY SESSION`

7.6 Transaction Management Statements

The following SQL statements manage transactions in the database.

These SQLScript statements are fully supported in the SAP ASE SQLScript database:

- COMMIT Statement (Transaction Management) [page 168]
- ROLLBACK Statement (Transaction Management) [page 248]

These SQLScript statements have limited support in the SAP ASE SQLScript database:

- SET TRANSACTION Statement (Transaction Management) [page 260]

These SQLScript statements are not supported in the SAP ASE SQLScript database:

- `LOCK TABLE`
- `SET TRANSACTION AUTOCOMMIT DDL`

7.7 Alphabetical List of Statements

Below is a list of available statements, including any restrictions for the SAP ASE environment.

7.7.1 ALTER PROCEDURE Statement (Procedural)

Alters a procedure or manually triggers a recompilation of a procedure by generating an updated execution plan.

> i Note
>
> SAP ASE restrictions:
>
> - The `SEQUENTIAL EXECUTION` option is not supported.
> - Only `SQLScript` is supported for the `LANGUAGE` option.
> - The `WITH PLAN` option is not supported.
> - The `DEFAULT SCHEMA` option is not supported.
> - The `READS SQL DATA` option is not supported.
> - The `BEGIN … END` option is not supported.

Syntax

```
ALTER PROCEDURE <proc_name> RECOMPILE
```

Syntax Elements

proc_name

Specifies the procedure to be altered, with an optional schema name.

```
<proc_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

Description

For full description of the clauses for the ALTER PROCEDURE statement, including examples, refer to the CREATE PROCEDURE and ALTER PROCEDURE statements in *the SAP HANA SQLScript Guide*.

Example

Trigger the recompilation of the `my_proc` procedure:

```
ALTER PROCEDURE my_proc RECOMPILE;
```

7.7.2 ALTER SEQUENCE Statement (Data Definition)

Alters the parameters of a sequence generator.

i Note

The implementation of CACHE and NO CACHE differs in SAP HANA and SAP ASE. For SAP ASE, the sequence value is stored in disk. For NO CACHE sequence, the value is written to disk each time when NEXTVAL is called. But for CACHE sequence, the value is not written to disk every time NEXTVAL is called. The value is written in memory when NEXTVAL is called until it reaches the cache size, and then it flushes the value to disk.

Syntax

```
ALTER SEQUENCE <sequence_name> [<restart_with>] [<parameter_list>] [RESET BY
<reset_by_subquery>]
```

Syntax Elements

sequence_name

Specifies the name of the sequence to be altered with optional schema name.

```
<sequence_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

restart_with

(Optionally) specifies the starting value of the sequence:

```
<restart_with> ::= RESTART WITH <restart_value>
<restart_value> ::= <signed_integer>
```

`<restart_value>` can be between -4611686018427387904 and 4611686018427387902.

The current value of the sequence is used if you do not specify a value for the RESTART WITH clause.

parameter_list

(Optional) specifies the sequence parameter list:

```
<parameter_list> ::=
<sequence_parameter> [{, <sequence_parameter>}...]
```

sequence_parameter

Specifies a set of parameters that can be used with ALTER SEQUENCE.

```
<sequence_parameter> ::=
 INCREMENT BY <increment_value>
 | MAXVALUE <max_value>
 | NO MAXVALUE
 | MINVALUE <min_value>
 | NO MINVALUE
 | CYCLE
 | NO CYCLE
 | CACHE <cache_size>
 | NO CACHE
```

INCREMENT BY increment_value

Specifies the amount the next sequence value is incremented from the last value assigned.

```
INCREMENT BY <increment_value>
<increment_value> ::= <signed_integer>
```

The default is 1. Specify a negative value to generate a descending sequence. An error is returned if the INCREMENT BY value is 0.

MAXVALUE max_value

Specifies the maximum value that can be generated by the sequence.

```
MAXVALUE <max_value>
<max_value> ::= <signed_integer>
```

`<max_value>` must be between -4611686018427387903 and 4611686018427387902.

NO MAXVALUE

Specifies the maximum value for an ascending sequence is 4611686018427387903, and the maximum value for a descending sequence is -1.

MINVALUE min_value

Specifies the minimum value that a sequence can generate.

```
MINVALUE <min_value>
<min_value> ::= <signed_integer>
```

`<min_value>` must be between -4611686018427387904 and 4611686018427387902.

NO MINVALUE

Specifies that the minimum value for an ascending sequence is 1 and the minimum value for a descending sequence is -4611686018427387903.

CYCLE

Specifies that the sequence number is restarted after it reaches its maximum or minimum value.

NO CYCLE

Specifies that the sequence number is not restarted after it reaches its maximum or minimum value.

CACHE cache_size

Specifies the cache size with which a range of sequence numbers are cached in memory. `<cache_size>` must be unsigned integer. An error is returned if the CACHE is less than 2 or greater than `min(MAXVALUE - MINVALUE, 30000). 2 <= cache_size <= min(MAXVALUE - MINVALUE, 30000).`

NO CACHE

Specifies that the sequence number is not cached in memory. This is the default behavior.

reset_by_subquery

During a restart of the database, the system automatically executes the RESET BY statement and the sequence value is restarted with the value determined from the RESET BY subquery. This setting is optional.

```
<reset_by_subquery> ::= <subquery>
```

The sequence value is stored persistently in database if RESET BY is not specified. During the restart of the database, the next value of the sequence is generated from the saved sequence value.

Description

The ALTER SEQUENCE statement alters the parameters of a sequence generator.

Examples

Example 1

Create table A and a sequence seq. Sequence seq, when reset, starts from the value of the select statement shown:

```
CREATE TABLE A (a INT);
CREATE SEQUENCE seq RESET BY SELECT IFNULL(MAX(a), 0) + 1 FROM A;
```

Change the starting sequence value of sequence seq to 2:

```
ALTER SEQUENCE seq RESTART WITH 2;
```

Change the maximum value of sequence s to 100, and specify that it does not have a minimum value:

```
ALTER SEQUENCE seq MAXVALUE 100 NO MINVALUE;
```

Change the incremental value of sequence seq to 3, and specify that the sequence does not restart upon reaching its maximum or minimum value:

```
ALTER SEQUENCE seq INCREMENT BY 3 NO CYCLE;
```

Example 2

Create table B, with column a. You create a sequence s1 with a reset-by subquery based on table B:

```
CREATE TABLE B (a INT);
CREATE SEQUENCE s1 RESET BY SELECT IFNULL(MAX(a), 0) + 1 FROM B;
```

Change the reset-by subquery of sequence s1 to the maximum value contained in column a of table B:

```
ALTER SEQUENCE s1 RESET BY SELECT MAX(a) FROM B;
```

Related Information

Working with Sequences [page 11]
DROP SEQUENCE Statement (Data Definition) [page 223]
CREATE SEQUENCE Statement (Data Definition) [page 187]

7.7.3 ALTER TABLE Statement (Data Definition)

Alters the definition of a table.

i Note

SAP ASE restrictions:

The following clauses are not supported in SAP ASE SQLscript:

- `<add_replica_clause>`
- `<asynchronous_replica_clause>`
- `<auto_merge_option>`

- <default_value>
- <delta_log_option>
- <drop_primary_key_clause>
- <drop_replica_clause>
- <move_clause>
- <move_replica_clause>
- <persistent_merge_option>
- <preload clause>
- <reclaim_data_space_clause>
- <schema_flexibility_option>
- <series_reorganize_clause>
- <set_group_option>
- <set_row_order>
- <table_conversion_clause>
- <trigger_option>
- <unload_priority>
- <unset_group_option>
- <unset_row_order>
- <unused_retention_period>
- <using_extended_storage_clause>

Syntax

```
ALTER TABLE <table_name>
                 [<add_column_clause>]
                 [<drop_column_clause>]
                 [<alter_column_clause>]
                 [<add_constraint_clause>]
                 [<drop_constraint_clause>]
                 [<add_range_partition_clause>]
                 [<drop_range_partition_clause>]
                 [<merge_partition_clause>]
                 [<partition_clause>]
                 [<alter_partition_attributes_clause> ]
```

Syntax Elements

table_name

Specifies the identifier of the table to be altered, with optional schema name.

```
<table_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <identifier>
```

add_column_clause

```
<add_column_clause> ::=
ADD ( {<column_definition> [<column_constraint_short>]} [, …] )
```

drop_column_clause

Removes one or more columns from the specified table.

```
<drop_column_clause> ::= DROP ( <column_name>[,...] )
```

alter_column_clause

Alters one or more column definitions.

```
<alter_column_clause> ::=
 ALTER ( {<column_definition> [<column_constraint>]}  [,…] )
```

Restrictions:

- For column table, only increasing the size of a column data type is allowed to prevent data loss. For example, changing from NVARCHAR(20) to NVARCHAR(10) or from INTEGER to TINYINT raises an error.
- For row table, only increasing the size of VARCHAR and NVARCHAR type column is allowed. Other data type changes are not allowed.
- ALTER does not currently follow data type conversion rules.
- Adding NOT NULL constraint to an existing column is allowed if either of the following cases are true:
  - The table is empty.
  - The default value is specified when the table contains data.
  - The table does not contain a NULL-value in that column.
- You can only alter the comment for a column while making another change to the column (such as changing the data type).

column_definition

Specifies a definition for a column:

```
<column_definition> ::= <column_name>
   { <data_type> | <lob_data_type>}
   [<ddic_data_type>]
   [<default_value_clause>]
   [<col_gen_as_expression> | <col_gen_as_ident>]
   [<col_calculated_field>]
   [<schema_flexibility>]
   [<fuzzy_search_index>]
   [<fuzzy_search_mode>]
   [<load_unit>]
```

data_type

Specify one of the available data types:

```
<data_type> ::=
 DATE
 | TIME
 | SECONDDATE
 | TIMESTAMP
 | TINYINT
```

```
  | SMALLINT
  | INTEGER
  | BIGINT
  | REAL
  | DOUBLE
  | TEXT
  | BINTEXT
  | VARCHAR [ (<unsigned_integer>) ]
  | NVARCHAR [ (<unsigned_integer>) ]
  | ALPHANUM [ (<unsigned_integer>) ]
  | VARBINARY [ (<unsigned_integer>) ]
  | SHORTTEXT [ (<unsigned_integer>) ]
  | DECIMAL [ (<unsigned_integer> [, <unsigned_integer> ]) ]
  | FLOAT [ (<unsigned_integer>) ] | BOOLEAN
```

lob_data_type

Specifies a LOB data type:

```
<lob_data_type> ::=
<lob_type_name> [MEMORY THRESHOLD <memory_threshold_value>]
<lob_type_name> ::=
 BLOB
  | CLOB
  | NCLOB
```

`<memory_threshold_value>` controls whether LOB data should be stored in memory, according to the following conditions:

- If `<memory_threshold_value>` is not provided, then the LOB data is stored in memory by default.
- If `<memory_threshold_value>` is provided and its LOB size is bigger than the memory threshold value, then LOB data is stored on disk.
- If `<memory_threshold_value>` is provided and its LOB size is equal or less than the memory threshold value, then LOB data is stored in memory.
- If `<memory_threshold_value>` is NULL, then all LOB data is stored in memory.
- If `<memory_threshold_value>` is 0, then all LOB data is stored on disk.

ddic_data_type

Specifies a DDIC data type:

```
<ddic_data_type> ::=
 DDIC_ACCP | DDIC_ALNM | DDIC_CHAR | DDIC_CDAY | DDIC_CLNT | DDIC_CUKY
| DDIC_CURR | DDIC_D16D
  | DDIC_D34D | DDIC_D16R | DDIC_D34R | DDIC_D16S | DDIC_D34S |
DDIC_DATS | DDIC_DAY  | DDIC_DEC
  | DDIC_FLTP | DDIC_GUID | DDIC_INT1 | DDIC_INT2 | DDIC_INT4 |
DDIC_INT8 | DDIC_LANG | DDIC_LCHR
  | DDIC_MIN  | DDIC_MON  | DDIC_LRAW | DDIC_NUMC | DDIC_PREC |
DDIC_QUAN | DDIC_RAW  | DDIC_RSTR
  | DDIC_SEC  | DDIC_SRST | DDIC_SSTR | DDIC_STRG | DDIC_STXT |
DDIC_TIMS | DDIC_UNIT | DDIC_UTCM
  | DDIC_UTCL | DDIC_UTCS | DDIC_TEXT | DDIC_VARC | DDIC_WEEK
```

default_value_clause

Specifies a value to be assigned to the column if an INSERT statement does not provide a value for the column:

```
<default_value_clause> ::= DEFAULT <default_value_exp>
<default_value_exp> ::=
 NULL
 | <string_literal>
 | <signed_numeric_literal>
 | <unsigned_numeric_literal>
 | <datetime_value_function>
<datetime_value_function> ::=
CURRENT_DATE
 | CURRENT_TIME
 | CURRENT_TIMESTAMP
 | CURRENT_UTCDATE
 | CURRENT_UTCTIME
 | CURRENT_UTCTIMESTAMP
```

col_gen_as_ident

Specifies an identity column:

```
<col_gen_as_ident> ::= GENERATED {ALWAYS | BY DEFAULT} AS IDENTITY
[(<sequence_option>)]
<sequence_option> ::= {
   <sequence_param_list>
   | RESET BY <subquery>
   | <sequence_param_list>
   RESET BY <subquery>
}
<sequence_param_list> ::= <sequence_parameter>[{,
<sequence_parameter>}...]
<sequence_parameter> ::=
 START WITH <start_value>
 | INCREMENT BY <increment_value>
 | MAXVALUE <max_value>
 | NO MAXVALUE
 | MINVALUE <min_value>
 | NO MINVALUE
 | CYCLE
 | NO CYCLE
 | CACHE <cache_size>
 | NO CACHE
 RESET BY <subquery>
```

If ALWAYS is specified, then values are always generated. If BY DEFAULT is specified, then values are generated by default.

For more information on sequence parameters, see the CREATE SEQUENCE statement.

For more information on subqueries, see the SELECT statement.

column_constraint

The column constraint rules:

```
<column_constraint> ::=
 NULL
 | NOT NULL
 | [<constraint_name_definition>] <unique_specification>
 | [<constraint_name_definition>] <references_specification>
```

NULL

If NULL is specified it is not considered a constraint, then it represents that a column that may contain a null value. The default is NULL.

NOT NULL

The NOT NULL constraint prohibits a column value from being NULL:

constraint_name_definition

```
<constraint_name_definition> ::= CONSTRAINT <constraint_name>
<constraint_name> ::= <identifier>
```

unique_specification

Specifies unique constraints:

```
<unique_specification> ::=
 UNIQUE [<unique_tree_type_index>]
 | PRIMARY KEY [<unique_tree_type_index>
 | <unique_inverted_type_index>]
```

UNIQUE specifies a column as a unique key. A composite unique key enables the specification of multiple columns as a unique key. With a unique constraint, multiple rows cannot have the same value in the same column.

A PRIMARY KEY constraint is a combination of a NOT NULL constraint and a UNIQUE constraint. It prohibits multiple rows from having the same value in the same column.

If you do not specify the index type, then the SAP HANA database automatically selects an index type as follows:

Index type	Criteria
CPBTREE	- character string types - binary string types - decimal types - when the constraint is a composite key - when the constraint is a non-unique constraint
BTREE	All other cases than specified for CPBTREE

`<unique_inverted_type_index>` specifies the inverted index type. INVERTED HASH encodes the composite key in a condensed manner and allows for faster equality queries over the composite keys, as well as reduced memory requirements for storing the composite key. INVERTED HASH should not be used as a composite type in cases where range queries or similarity queries on the composite keys are a significant part of the workload, VALUE should be used instead. INVERTED VALUE is the default composite key type.

references_specification

For `<references_specification>`, see References Specification [page 152].

column_constraint_short

The description of the syntax for <column_constraint_short> and <column_constraint> are the same except that <column_constraint_short><unique_specification>, and is used specifically when adding a column (does not include syntax for specifying <add_column_clause>):

```
<column_constraint_short> does ::=
 NULL
 | NOT NULL
 | [<constraint_name_definition>] <references_specification>
```

add_constraint_clause

Adds a table constraint.

```
<add_constraint_clause> ::=
 ADD [CONSTRAINT <constraint_name>] <table_constraint>
<constraint_name> ::= <identifier>
```

table_constraint

Specifies either a unique constraint, a referential constraint, or a check constraint:

```
<table_constraint> ::=
 <unique_constraint_definition>
 | <referential_constraint_definition>
 | <check_constraint_definition>
```

unique_constraint_definition

Specifies a unique constraint:

```
<unique_constraint_definition> ::=
<unique_specification> (<unique_column_name_list>)
```

For more information about unique constraints, see the CREATE TABLE statement.

unique_column_name_list

Specifies the unique column name list which can have one or more column names.

```
<unique_column_name_list> ::=
 <unique_column_name>[{, <unique_column_name>}...]
<unique_column_name> ::= <identifier>
```

referential_constraint_definition

Specifies a referential constraint:

```
<referential_constraint_definition> ::=
FOREIGN KEY (<referencing_column_name_list>) <references_specification>
```

Foreign key constraints over unique key columns are not supported. Self-referencing foreign key constrains are supported.

referencing_column_name_list

Specifies the referencing column name list, which can have one or more column names:

```
<referencing_column_name_list> ::= <referencing_column_name>[{,
<referencing_column_name>}...]
```

referencing_column_name

The identifier of a referencing column:

```
<referencing_column_name> ::= <identifier>
```

references_specification

Specifies the referenced table, with optional column name list and trigger action.:

```
<references_specification> ::=
REFERENCES <referenced_table> [(<referenced_column_name_list>)]
[<referential_triggered_action>]
```

If `<referenced_column_name_list>` is specified, then there is a one-to-one correspondence between `<column_name>` of `<column_definition>` and `<referenced_column_name>`. If it is not specified, then there is a one-to-one correspondence between `<column_name>` of `<column_definition>` and the column name of the referenced table's primary key.

referenced_column_name_list

Specifies the referenced column name list, which can have one or more column names:

```
<referenced_column_name_list> ::= <referenced_column_name>[{,
<referenced_column_name>}...]
```

referenced_table

Specifies the identifier of a table to be referenced:

```
<referenced_table> ::= <identifier>
```

referenced_column_name

Specifies the identifier of the column name to be referenced:

```
<referenced_column_name> ::= <identifier>
```

referential_triggered_action

Specifies an update rule with optional delete rule or a delete rule with optional update rule:

```
<referential_triggered_action> ::=
   <update_rule> [<delete_rule>]
   | <delete_rule> [<update_rule>]
```

The order that you specify the rules in provides an order of precedence for execution:

update_rule

Specifies the behavior to perform when data in the column is updated:

```
<update_rule> ::= ON UPDATE <referential_action>
<referential_action> ::= RESTRICT | CASCADE | SET NULL | SET DEFAULT
```

Action Name	Update Action
RESTRICT	Any updates to a referenced table are prohibited if there are any matched records in the referencing table. This is the default action.
CASCADE	If a record is updated in the referenced table, then the corresponding records in the referencing table are also updated with the same values.
SET NULL	If a record is updated in the referenced table, then the corresponding records in the referencing table are also updated with null values.
SET DEFAULT	If a record is updated in the referenced table, then the corresponding records in the referencing table are also updated with their default values.

delete_rule

Specifies the behavior to perform when data in the column is deleted:

```
<delete_rule> ::= ON DELETE <referential_action>
```

The following DELETE referential actions are possible:

Action Name	Delete Action
RESTRICT	Any deletions to a referenced table are prohibited if there are any matched records in the referencing table. This is the default action.
CASCADE	If a record in the referenced table is deleted, then the corresponding records in the referencing table are also deleted.
SET NULL	If a record in the referenced table is deleted, then the corresponding records in the referencing table are set to null.
SET DEFAULT	If a record in the referenced table is deleted, then the corresponding records in the referencing table are set to their default values.

check_constraint_definition

Specifies a condition to check for in the column:

```
<check_constraint_definition> ::= CHECK (<search_condition>)
```

A table check constraint is satisfied if `<search_condition>` evaluates to true.

drop_primary_key_clause

Drops the primary key constraint.

```
<drop_primary_key_clause> ::= DROP PRIMARY KEY
```

drop_constraint_clause

Drops a unique or referential constraint.

```
<drop_constraint_clause> ::= DROP CONSTRAINT <constraint_name>
<constraint_name> ::= <identifier>
```

table_conversion_clause

Converts the table storage from ROW to COLUMN or from COLUMN to ROW.

```
<table_conversion_clause> ::= [ALTER TYPE] {
   ROW [THREADS <number_of_threads>]
   | COLUMN [<column_store_mode>] [THREADS <number_of_threads> [BATCH
<batch_size>]]
}
```

column_store_mode

Specifies which column to use for a destination column table:

```
<column_store_mode> ::= DELTA | MAIN
```

The default is DELTA.

THREADS number_of_threads

Specifies how many parallel execution threads should be used for the table conversion:

```
THREADS <number_of_threads>
 <number_of_threads> ::= <unsigned_integer>
```

The optimal value for the number of threads is the number of available CPU cores. If THREADS is not provided, then the default value of the number of CPU cores specified in the `indexserver.ini` file is used.

BATCH batch_size

Specifies the number of rows to be inserted in a batch:

```
BATCH <batch_size>
 <batch_size> ::= <unsigned_integer>
```

If BATCH is not specified, then the default value of 2,000,000 is used. Inserts into column tables are immediately committed after every `<batch_size>` records have been inserted. You can only use the BATCH option when a table is converted from ROW to COLUMN storage.

move_clause

Moves a partition:

```
<move_clause> ::=
 MOVE [PARTITION <partition_number>] TO <indexserver_host_port> [PHYSICAL]
 | MOVE [PARTITION <partition_number>] PHYSICAL
```

PARTITION partition_number

Moves a table to another location in a distributed environment:

```
PARTITION <partition_number>
<partition_number> ::= <unsigned_integer>
```

When using the MOVE command without the PHYSICAL option, all table data (excluding Hybrid Lob on disk) is directly moved to the target host (node). Using the PHYSICAL option, the Hybrid Lobs on disk are also moved to the target host.

For partitioned tables, specifies the partition to be moved. An error is returned if you try to move a partitioned table without specifying `<partition_number>`.

indexserver_host_port

Specifies the internal indexserver host name and port number where the table is to be moved:

```
<indexserver_host_port> ::= 'host_name:port_number'
```

PHYSICAL

Specifies that a column store table's persistence storage be moved immediately to the target host. Row store tables are always moved immediately, and does not need PHYSICAL.

If you do not specify PHYSICAL, then the table move creates a link inside the new host persistence pointing to the old host persistence. The link is removed on the next merge or upon execution of another move operation not using the TO `<indexserver_host_port>` clause.

add_range_partition_clause

Adds a partition for tables partitioned with RANGE, HASH RANGE, ROUNDROBIN RANGE, RANGE RANGE:

```
<add_range_partition_clause> ::= ADD [(<partition_expression>)]
<range_partition_clause>
```

partition_expression

Specifies how to segregate data into partitions:

```
<partition_expression> ::=
 <column_name> [AS <data_type_partition_expression>]
 | YEAR(<column_name>)
 | MONTH(<column_name>)
<column_name> ::= <identifier>
<data_type_partition_expression> ::= INT | DATE
```

range_partition_clause

The range specifier for a new partition:

```
<range_partition_clause> ::= {<from_to_spec> | <single_spec>} [,
<partition_others>]
<from_to_spec> ::= PARTITION <lower_value> <= VALUES < <upper_value>
<lower_value> ::= <string_literal> | <numeric_literal>
<upper_value> ::= <string_literal> | <numeric_literal>
<single_spec> ::= PARTITION VALUE = <target_value>
<target_value> ::= <string_literal> | <numeric_literal>
<partition_others> ::= PARTITION OTHERS [DYNAMIC [THRESHOLD
<threshold_count>] ]
```

`<from_to_spec>` specifies a partition by using lower and upper values of a `<partition_expression>`.

`<single_spec>` specifies a partition using a single value of a `<partition_expression>`.

`<partition_others>` specifies other supported partitioning options. The DYNAMIC keyword applied to PARTITION OTHERS enables dynamic range partitioning on the table. Without the DYNAMIC keyword, dynamic range is deactivated if applicable. The optional `<threshold_count>` sets the table's dynamic range threshold limit and can be used only in conjunction with DYNAMIC. For syntax details, see `<partition range specifier>` [page 157].

drop_range_partition_clause

Drops a partition for tables partitioned with RANGE, HASH RANGE, ROUNDROBIN RANGE, RANGE RANGE:

```
<drop_range_partition_clause> ::= DROP <range_partition_clause>
```

If `<partition_expression>` is specified, then the corresponding partition is dropped.

merge_partition_clause

Merges all parts of a partitioned table into a non-partitioned table:

```
<merge_partition_clause> ::= MERGE PARTITIONS
```

partition_clause

Partitions a table using the selected rules:

```
<partition_clause> ::=
 PARTITION BY <hash_partition> [, <range_partition> | , <hash_partition>]
 | PARTITION BY <range_partition> [,<range_partition>]
 | PARTITION BY <roundrobin_partition> [,<range_partition>]
```

hash_partition

Partitions the created table using a hash partitioning scheme:

```
<hash_partition> ::=
 HASH (<partition_expression> [{<partition_expression>,}...])
 PARTITIONS {<num_partitions> | GET_NUM_SERVERS()}
```

GET_NUM_SERVERS() returns the number of servers/partitions according to table placement.

range_partition

Partitions the created table using a range partitioning scheme:

```
<range_partition> ::=
 RANGE (<partition_expression>) (<range_spec>, ...)
```

partition_expression

Specifies how to segregate data into partitions:

```
<partition_expression> ::=
 <column_name> [AS <data_type_partition_expression>]
 | YEAR(<column_name>)
 | MONTH(<column_name>)

<dynamic_range_data_type_partition_expression> ::= INT | DATE
```

data_type_partition_expression

The supported data types for the column used in the dynamic range partitioning are:

- INT, VARCHAR for the dynamic data type INTEGER
- DATE, TIMESTAMP, and SECONDDATE for the dynamic range data type DATE

dynamic_range_data_type_partition_expression

This construct allows dynamic range to treat the partitioning column as the specified data type. It can be used only in conjunction with DYNAMIC and is called dynamic range data type.

range_spec

The range specifier for a partition. Ranges can be specified for positive values:

```
<range_spec> ::=
   {<from_to_spec> | <single_spec>}
   [{ {<from_to_spec> | <single_spec>},} ...]
   [, PARTITION OTHERS]
```

from_to_spec

Specifies a partition by using lower and upper values of a `<partition_expression>`:

```
<from_to_spec> ::= PARTITION <lower_value> <= VALUES < <upper_value>
<lower_value> ::= <string_literal> | <numeric_literal>
<upper_value> ::= <string_literal> | <numeric_literal>
```

single_spec

Specifies a partition using a single value of a `<partition_expression>`:

```
<single_spec> ::= PARTITION VALUE = <target_value>
<target_value> ::= <string_literal> | <numeric_literal>
```

PARTITION OTHERS

Specifies that all other values that are not covered by the partition specification are gathered into one partition.

roundrobin_partition

Partitions the created table by using a round robin partitioning scheme:

```
<roundrobin_partition> ::=
   ROUNDROBIN PARTITIONS {<num_partitions> | GET_NUM_SERVERS()}
   [, <range_partition>]
```

GET_NUM_SERVERS() returns the number of servers/partitions according to table placement.

`<num_partitions>` specifies the number of partitions to be created for the table.

alter_partition_attributes_clause Defines attributes on the partition:

```
<alter_partition_attributes_clause> ::=
    FOR NON CURRENT PARTITIONS UNIQUE CONSTRAINTS { ON | OFF }
    |  FOR DEFAULT STORAGE {ALL | NON CURRENT } PARTITIONS { PAGE | COLUMN }
LOADABLE
```

synchronous_replica_clause

Adds, drops, or moves a synchronous replica:

```
<synchronous_replica_clause> ::=
 <add_synchronous_replica>
 | <drop_synchronous_replica>
```

add_synchronous_replica_clause

Adds a synchronous replica:

```
<add_synchronous_replica_clause> ::= ADD <replica_clause>
<replica_clause> ::=
 [SYNCHRONOUS] [<column_or_row>] REPLICA [<replica_partition>] AT
{<num_replicas> LOCATIONS | <replica_locations>}
<column_or_row> ::= COLUMN | ROW
<replica_partition> ::= <partition_clause>
<num_replicas> ::= ALL | <unsigned_integer>
<replica_locations> ::= [LOCATION] {<indexserver_host_port> |
(<indexserver_host_port>, ...)}
<indexserver_host_port> ::= 'host_name:port_number'
```

Replica type is specified by `<column_or_row>` if you do not specify it, then the type is the same as that of table. Row table can have partitioned column replica specified by `<replica_partition>`:

- ALL LOCATIONS adds replicas in all index servers. `<unsigned_integer>`
- LOCATIONS adds replicas in the specified number of index servers.
- `<replica_locations>` adds replicas in the specified index server.

drop_synchronous_replica_clause

Drops a synchronous replica at the specified location(s):

```
<drop_synchronous_replica_clause> ::= DROP REPLICA AT { {<num_replicas> |
ALL} LOCATIONS | <replica_locations>}
```

- ALL LOCATIONS drops replicas in all indexservers.
- `<num_replicas>` drops replicas in the specified number of indexservers:

  ```
  <num_replicas> ::= <unsigned_integer>
  ```

- `<replica_locations>` drops replicas in the specified indexserver:

asynchronous_replica_clause

Adds, drops, enables or disables an asynchronous replica.:

```
<asynchronous_replica_clause> ::=
 <add_asynchronous_replica_clause>
 | <drop_asynchronous_replica_clause>
<partition_number> ::= <unsigned_integer>
```

i Note

Asynchronous table replication works within a single SAP HANA scale-out landscape. Do not replicate a table across two different SAP HANA landscapes or between SAP HANA and other DBMS instances.

- You can distribute source tables to multiple nodes, but a source table and its replica cannot be located on the same node.
- You cannot create a replica table on the master indexserver.
- Only one identical replica table can exist in a replica node. Regarding partitioned tables, only one identical replica table's partition can exist in a replica node.
- You cannot set these table types as a replication table: history table, flexible table, temporary table, proxy table, or extended storage table.
- A source table and its replica table have an identical table structure. For partitioned tables, its source and replica have identical partitioning specifications.

- You cannot execute write and DDL operations on the replica table.
- A “DDL autocommit off” transaction cannot execute a write operation on a replica table after a DDL operation on any table in the same transaction boundary.
- The duration of the replication activation depends on the size of the table to be replicated. For large tables, you should use the use EXPORT/IMPORT statements.
- Global temporary table data in all sessions should be empty in order to execute ALTER TABLE.

add_asynchronous_replica_clause

Creates the additional replica table for `<source_schema_name>.<source_table_name>`:

```
<add_asynchronous_replica_clause> ::=
   <source_schema_name>.<source_table_name>
   ADD ASYNCHRONOUS REPLICA [AT '<replica_host>:<replica_port>' ]
<source_schema_name> ::= <identifier>
<source_table_name> ::= <identifier>
<replica_host> ::= <hostname>
<replica_port> ::= <port_number>
```

Add the replica table on the specified location with the AT clause. If you don't specify it, then the replica table is added based on the table placement rule. If no table placement rule is also defined, then the replica table is added one of the slave nodes, which is not a source table location and no replica table exists. If there are no more replica nodes to add a replica table, then an error is returned. This command does not activate its replication (that is, the replica is created as an empty state). To activate its replication, execute ALTER SYSTEM ENABLE ALL ASYNCHRONOUS TABLE REPLICAS.

drop_asynchronous_replica_clause

Drops table replicas table for the replica specified:

```
<drop_asynchronous_replica_clause> ::=
   <source_schema_name>.<source_table_name>
   DROP REPLICA AT { ALL LOCATIONS | ' <replica_host>:<replica_port>'}
<source_schema_name> ::= <identifier>
<source_table_name> ::= <identifier>
<replica_host> ::= <hostname>
<replica_port> ::= <port_number>
```

ALL LOCATIONS drops replicas in all indexservers. A specified indexserver drops the replica from that location only.

move_replica_clause

Moves a replica from one index server to another:

```
<move_replica_clause> ::=
 MOVE REPLICA [PARTITION <partition_number> FROM [LOCATION] <from_location>
TO [LOCATION] <to_location>

<from_location> ::= <indexserver_host_port>
<to_location> ::= <indexserver_host_port>
<indexserver_host_port> ::= '<host_name>:<port_number>'
```

persistent_merge_option

Enables or disables persistent merging:

```
<persistent_merge_option> ::= {ENABLE | DISABLE} PERSISTENT MERGE
```

- (Default) When enabled, the merge-manager uses persistent merges for the given table.

- When disabled, the merge-manager uses main-memory merges instead of persistent merges for the given table.

delta_log_option

Enables or disables delta logging for table:

```
<delta_log_option> ::= {ENABLE | DISABLE} DELTA LOG
```

After enabling, perform a savepoint to be certain that all data is persisted, and perform a data backup, to make sure the data is recoverable..

If logging is disabled, then log entries do not persist for this table. Changes to this table are only written to the data store when a savepoint is carried out. This can cause loss of committed transaction should the indexserver terminate. In the case of a termination, you must truncate this table and insert all data again. Use this option only during initial load.

auto_merge_option

Enables or disables automatic delta merge on the specified table:

```
<auto_merge_option> ::= {ENABLE | DISABLE} AUTOMERGE
```

The delta merge feature is only supported on column store tables.

unload_priority

Sets the priority of table to be unloaded from memory:

```
<unload_priority_option> ::= UNLOAD PRIORITY <unload_priority>
<unload_priority> ::= <digit>
```

`<unload_priority>` can be 0 ~ 9, where 0 means not-unloadable and 9 means earliest unload.

schema_flexibility_option

Enables, disables, or alters schema flexibility for the specified table:

```
<schema_flexibility_option> ::=
 { {ENABLE | DISABLE} SCHEMA FLEXIBILITY}
 | ALTER SCHEMA FLEXIBILITY <flexibility_option> [<flexibility_option>...]
```

ALTER SCHEMA FLEXIBILITY

Alters the schema flexibility for the specified table:

```
ALTER SCHEMA FLEXIBILITY <flexibility_option> [<flexibility_option> ...]
```

All schema flexibility options that are listed in the CREATE TABLE statement in the WITH SCHEMA FLEXIBILITY section can be used here.

trigger_option

Enables or disables the specified trigger on the specified table:

```
<trigger_option> ::= {ENABLE | DISABLE} TRIGGER [<trigger_name>]
```

If you do not specify `<trigger_name>`, then this setting enables or disables all triggers on the specified table.

set_group_option

Sets one or more table group option for the table. Table group settings are stored in the TABLE_GROUPS system view:

```
<set_group_options> ::= SET <option> [...]
<option> ::=
 GROUP TYPE <identifier>
 | GROUP SUBTYPE <identifier>
 | GROUP NAME <identifier>
 | GROUP LEAD
```

unset_group_option

Unsets (removes) all table group attributes for the table (type, subtype, name, and so on):

```
<unset_group_option> ::= UNSET GROUP
```

set_row_order

Sets the row order and type:

```
<set_row_order> ::= SET <row_order_definition>
<row_order_definition> ::= <row_order_specification>
(<unique_column_name_list>)
<row_order_specification> ::= ROW ORDER [BY VALUE]
<unique_column_name_list> ::= <unique_column_name>[{,
<unique_column_name>}...]
```

Rows are sorted exactly by value for a given set of columns. BY VALUE is the default.

unset_row_order

Unsets the row order and type:

```
<unset_row_order> ::= UNSET ROW ORDER
```

unused_retention_period

Unloads a table or table partition from memory if it was not accessed for the number of seconds defined as the retention period:

```
<unused_retention_period> ::= UNUSED RETENTION PERIOD { <retention_period> |
(<retention_period>, ...)}
<retention_period> ::= <unsigned_integer>
```

A single value specifies the retention period at table-level, including all of the table partitions. Multiple values specify specific retention periods for each of the table partitions. If multiple values are specified, then the number of values must match the number of table partitions.

The default value and behavior of UNUSED_RETENTION_PERIOD is 0 (inactive). To activate UNUSED_RETENTION_PERIOD, change the value to something other than 0. However, use of the UNUSED_RETENTION_PERIOD setting on a table requires that the global unused_retention_period setting in the `global.ini` configuration file be set to something other than 0.

reclaim_data_space_clause

Reclaims data space of the specified table:

```
<reclaim_data_space_clause> ::= RECLAIM DATA SPACE
```

This statement has the same effect as defragmentation by restructuring table data space. However, this behavior does not always guarantee that restructured data space uses less memory than before. In some optimized cases, restructured data space uses much memory.

series_reorganize_clause

Reorganizes a series table:

```
<series_reorganize_clause> ::=
   SERIES REORGANIZE [PART <int_const_partnum>]
   [LIMIT <inst_const_reorganize_limit>]
```

All table partitions are reorganized by default. If an error occurs during the reorganize operation, then it is returned from the blocking statement.

By default, the statement blocks until all reorganized work is completed.

PART int_const_partnum

Reorganizes the specified partition.

LIMIT inst_const_reorganize_limit

Limits the number of rows processed by the statement.

Description

This feature is used with restrictions or is extended by the following SAP HANA Dynamic Tiering option.

Examples

- Alter table t adding a new column c:

```
ALTER TABLE t ADD (c NVARCHAR(10) DEFAULT 'NCHAR');
```

Create a primary key constraint, prim_key, on columns a and b of table t:

```
ALTER TABLE t ADD CONSTRAINT prim_key PRIMARY KEY (a, b);
```

Change the table type of table t to COLUMN storage:

```
ALTER TABLE t COLUMN;
```

Set the preload flags of column b and c on table t:

```
ALTER TABLE t PRELOAD (b, c);
```

Partition table t with a RANGE partition, and then add an additional partition:

```
ALTER TABLE t PARTITION BY RANGE (a) (PARTITION VALUE = 1, PARTITION OTHERS);
 ALTER TABLE t ADD PARTITION 2 <= VALUES < 10;
```

Disable delta logging of table t:

```
ALTER TABLE t DISABLE DELTA LOG;
```

Change the unload priority of table t to 2:

```
ALTER TABLE t UNLOAD PRIORITY 2;
```

- Create table R. Then you alter table R adding a unique constraint UK:

```
CREATE TABLE R (A INT PRIMARY KEY, B NVARCHAR(10));
 ALTER TABLE R ADD CONSTRAINT UK UNIQUE (B);
```

Drop the unique constraint UK from table R:

```
ALTER TABLE R DROP CONSTRAINT UK;
```

Create table S. You add a referential constraint FK to table S that references column A of table R:

```
CREATE TABLE S (FA INT, B NVARCHAR(10));
 ALTER TABLE S ADD CONSTRAINT FK FOREIGN KEY(FA) REFERENCES R(A);
```

7.7.4 ALTER USER Statement (Access Control)

Modifies the database user.

i Note

The `ALTER USER` statement only supports modifying a users' password.

Syntax

```
ALTER USER <user_name> {<password_val_option>}
```

Syntax Elements

user_name

Specifies the username of the user to be modified:

```
<user_name> ::= <unicode_name>
```

password

Specifies the user password:

```
<password_val_option> ::=
PASSWORD <password>
```

The password must follow the rules of password for SAP ASE server..

Description

`<user_name>` must specify an existing database user.

The database administrator or a user with USER ADMIN system privilege can use ALTER USER statement

Examples

Alter a users password:

```
ALTER USER new_user PASSWORD newPassword123;
```

7.7.5 CALL Statement (Procedural)

Calls a procedure defined using the CREATE PROCEDURE statement..

i Note

SAP ASE restrictions:

- SAP ASE SQLScript does not support DEBUG mode.
- The OUT and INOUT parameters are treated the same in SAP ASE.
- Executing a stored procedure in a CALL statement with parameters values represented as question marks (`call example_proc(?, ?)`) at the ODBC or JDBC driver side is supported if the parameter is a scalar type. However, if the parameter is a table type, then it is not supported.

Syntax

```
CALL <proc_name> (<param_list>) [WITH OVERVIEW]
```

Syntax Elements

proc_name

Specifies the procedure to be called, and, optionally, specifies a schema name for the identifier:

```
<proc_name> ::= [<schema_name>.]<identifier>
```

param_list

Specifies the procedure parameters:

```
<param_list> ::= <proc_param>[{, <proc_param>}...]
```

proc_param

Specifies procedure parameters:

```
<proc_param> ::=
 <identifier>
 | <string_literal>
 | <unsigned_integer>
 | <signed_integer>
 | <signed_numeric_literal>
 | <unsigned_numeric_literal>
 | <expressions>
 | <question_mark>
```

Parameters passed to a procedure are scalar constants and can be passed either as IN, OUT, or INOUT parameters:

- IN – Defines an input parameter. That is, the value of the variable transferred to the database procedure when the procedure is called. This example defines the input parameter as `v1` with a data type of `int`:

  ```
  in v1 int
  ```

- OUT or INOUT – Defines output parameters that combines the IN and OUT functions. That is, it passes data in and returns data out from the procedure. Supports only a scalar type. In this example, the contents of the `mytab` table are used as the input parameter:

  ```
  create procedure Example_proc(out IN_1 mytab, in v1 int) as
  begin
  IN_1 = select * from mytab;
  ```

Scalar parameters are assumed to be NOT NULL. Arguments for IN parameters of table type can either be physical tables or views.

WITH OVERVIEW

Determines that the result of a procedure call is stored in a physical table. Only takes effect on parameters with OUT mode.

Including the WITH OVERVIEW parameter returns the result of an output variable on the specified table. Scalar outputs are represented as temporary tables with a single cell. WITH OVERVIEW inserts the result set tuples of the procedure into the provided tables when you pass existing tables to the output parameters. CALL generates temporary tables that contain the result set when you pass a question mark to the output parameters These tables are dropped automatically once the database session is closed.

Description

When the procedure is defined with parameters, provide proper data according to these parameters when you issue the CALL command since they are not visible in the signature of the procedure. TABLE type allows you to transfer data between tables when it is used for procedure IN/OUT parameters.

CALL syntax behaves consistent with the SQL standard semantics when executed by a client; for example, Java clients can call a procedure using a JDBC `CallableStatement`. Scalar output variables are a scalar value that can be retrieved from the callable statement directly.

Unquoted identifiers are implicitly treated as upper case. Quoting identifiers respect capitalization and allow for using white spaces which are normally not allowed in SQL identifiers.

Example 1: Calling a Procedure Using an Existing Table as the Input Parameter

- Example 1 – Creates a table named `mytab`, inserts data, and then creates the `Example_p1` procedure with an input table type parameter (`in v0 mytable`) using an existing table (`mytab`). The data in `mytab` is assigned to the input parameter (`v0`) and passed to the `Example_p1` procedure:

```
create table mytab(a int, b int);
go
insert into mytab values (3, 30);
go
insert into mytab values (2, 20);
go
insert into mytab values (1, 10);
go
create procedure Example_p1(in v0 mytab) as
begin
select * from :v0;
end;
go
call Example_p1(mytab)
go
(3 rows affected)
a b
----------- -----------
3 30
2 20
1 10
(3 rows affected)
(return status = 0
```

- Example 2 – Transfers data between tables. Create the `tab1` table:

```
create table tab1(a int, b int, c int, d int);
go
insert into tab1 values(1, 2, 3, 4);
go
insert into tab1 values(5, 6, 7, 8);
go
create table new_tab(c1 int, c2 int);
go
```

1) Create the `Example_transfer` procedure:

```
create procedure Example_transfer(out v0 new_tab, in v1 tab1) as
begin
v0 = select a, c from :v1;
end;
go
```

2) Transfer data between `tab1` and `new_tab`:

```
CALL Example_transfer(new_tab, tab1) with overview
go
(0 rows affected)
(2 rows affected)
(return status = 0)
select * from new_tab
go
c1 c2
----------- -----------
1           3
5           7
(2 rows affected)
```

- Example 3 – (Calling a procedure with a table type as an output parameter) These examples use this table and procedure:

```
create table mytab(a int, b int);
go
insert into mytab values (1, 2);
go
insert into mytab values (3, 4);
go
create procedure Example_proc(out v0 mytab, in v1 int) as
begin
v0 = select * from mytab;
end;
go
```

When you invoke a procedure with an output parameter using a table type, and the output table type parameter uses the following:

  - An existing table with the `with overview` option, the data assigned to the output table table within the procedure is passed from the procedure and stored in the existing table. For example:

```
call Example_proc(mytab, 1) with overview
go
(2 rows affected)
(return status = 0)
select * from mytab
go
a b
----------- -----------
1 2
3 4
1 2
3 4
(4 rows affected)
```

  - An existing table name but does not include the `with overview` parameter, CALL returns an an error message. For example:

```
call Example_proc(mytab, 1)
go
Msg 101, Level 15, State 9:
```

```
Line 1:
Line 1: SQL syntax error.
```

7.7.6 COMMIT Statement (Transaction Management)

Makes changes to the database permanent, or terminates a user-defined transaction.

Syntax

```
COMMIT
```

Description

The system supports transactional consistency that guarantees the current job to be either completely applied to the system or disposed of.

If a user wants to apply the current job to the system persistently, then the user should issue a COMMIT command. If a COMMIT command is issued and successfully processed, then any change on the system that the current transaction has done is applied to the system and the change is visible to other jobs that start in the future. A job that has already been committed a via COMMIT command cannot be reverted.

In a distributed system, a standard two-phase-commit protocol is complied. In the first phase, transaction coordinator consults every participant whether if it is ready to commit, and sends the result to the participants in the second phase. The COMMIT command only works with an autocommit-disabled session.

Example

```
COMMIT;
```

7.7.7 CREATE FUNCTION Statement (Procedural)

Creates read-only functions.

i Note

SAP ASE restrictions:

- Only `SQLSCRIPT` can be used for the `LANGUAGE` option.

- The `WITH CACHE RETENTION` option is not supported.

Syntax

```
CREATE FUNCTION <function_name> [(<parameter_clause>)] RETURNS <return_type>

   [SQL SECURITY <mode>]
   [DEFAULT SCHEMA <default_schema_name>]
   AS
      {
      BEGIN
         <function_body>
      END
      | HEADER ONLY
      }
```

Syntax Elements

function_name

Specifies the function name, and optionally, a schema name:

```
<function_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <identifier>
```

parameter_clause

Specifies the input parameters for the function:

```
<parameter_clause> ::= <parameter> [{,<parameter>}...]
```

parameter

Specifies a function parameter with its data type:

```
<parameter> ::= [IN] <param_name> <datatype>
```

For user-defined functions:

```
<parameter> ::= [IN] <param_name> <datatype> [DEFAULT <value>]
```

param_name

Specifies the variable name for a parameter:

```
<param_name> ::= <identifier>
```

datatype

Specifies the data type of the parameter (can be the default value):

```
<datatype> ::= (<sql_type> | <table_type> |  <table_type_definition>)
DEFAULT (<value> | <table_name>)
```

Scalar user-defined functions only support primitive SQL types as input.

sql_type

Specifies the primitive SQL type.

Scalar user-defined functions supported types:

```
<sql_type> ::=
 DATE
 | TIME
 | TIMESTAMP
 | SECONDDATE
 | TINYINT
 | SMALLINT
 | INTEGER
 | BIGINT
 | DECIMAL
 | SMALLDECIMAL
 | REAL
 | DOUBLE
 | VARCHAR
 | NVARCHAR
```

Table user-defined functions supported types:

```
<sql_type> ::=
 DATE
 | TIME
 | TIMESTAMP
 | SECONDDATE
 | TINYINT
 | SMALLINT
 | INTEGER
 | BIGINT
 | DECIMAL
 | SMALLDECIMAL
 | REAL
 | DOUBLE
 | VARCHAR
 | NVARCHAR
 | ALPHANUM
 | VARBINARY
 | CLOB
 | NCLOB
 | BLOB
```

table_type

Specifies the table type:

```
<table_type> ::= <identifier>
```

To look at a table type previously defined with the CREATE TYPE command, see CREATE TYPE.

table_type_definition

Specifies a table type that is implicitly defined within the signature:

```
<table_type_definition> ::= TABLE (<column_list_definition>)
<column_list_definition> ::= <column_elem>[{, <column_elem>}...]
<column_elem> ::= <column_name> <data_type>
<column_name> ::= <identifier>
```

return_type

Specifies the return type:

```
<return_type> ::= <return_parameter_list> | <return_table_type>
```

Table functions must return a table whose type is defined by `<return_table_type>`. Scalar functions must return scalar values specified in `<return_parameter_list>`:

return_parameter_list

Specifies the output parameters:

```
<return_parameter_list> ::= <return_parameter>[{, <return_parameter>}...]
<return_parameter> ::= <parameter_name> <sql_type>
```

return_table_type

Specifies the structure of the returned table data:

```
<return_table_type> ::= TABLE ( <ret_column_list> ) |<table_type>
```

Table user-defined functions support RETURN `<table_type>`

ret_column_list

Specifies the list of columns returned from the function:

```
<ret_column_list> ::= <ret_column_elem>[{, <ret_column_elem>}...]
```

ret_column_elem

Specifies the name of the column element with its associated data type:

```
<ret_column_elem> ::= <column_name> <sql_type>
<column_name> ::= <identifier>
```

lang

Specifies the programming language used in the function:

```
LANGUAGE <lang>
<lang> ::= SQLSCRIPT
```

You can only use SQLScript functions.

mode

Specifies the security mode of the function:

```
SQL SECURITY <mode>
<mode> ::= DEFINER | INVOKER
```

DEFINER

Performs the execution of the function with the privileges of the definer of the function. DEFINER is the default for table user-defined functions.

INVOKER

Performs the execution of the function with the privileges of the invoker of the function. Invoker is the default for scalar user-defined functions.

default_schema_name

Specifies the schema for unqualified objects in the function body:

```
<default_schema_name> ::= <identifier>
```

If you do not specify `<default_schema_name>`, the `<current_schema>` of the session is used.

DETERMINISTIC

For use with scalar functions, DETERMINISTIC specifies that the function always returns the same result any time it is called with a specific set of input parameters. Deterministic functions offer the advantage that they do not need to be recalculated every the result every time; you can refer to the cached result.

function_body

Specifies the main body of the table functions and scalar functions:

```
<function_body> ::= <scalar_function_body>|<table_function_body>
<scalar_function_body> ::= [DECLARE <func_var>] <proc_assign>
<table_function_body> ::= [<func_block_decl_list>]
 [<func_handler_list>]
 <func_stmt_list>
 <func_return_statement>
```

Because the function is flagged as read-only, neither DDL or DML statements (INSERT, UPDATE, and DELETE) are allowed in the function body. Scalar functions do not support table-typed input variables or table operations in the function body.

For the definition of `<proc_assign>`, see the CREATE PROCEDURE statement.

func_block_decl_list

Specifies one or more local variables that are associated with a scalar type or an array type:

```
<func_block_decl_list> ::=
 DECLARE {
    <func_var>
    |<func_cursor>
    |<func_condition>
   }
<func_var> ::=
 <variable_name_list> [CONSTANT] { <sql_type> | <array_datatype> }
   [NOT NULL][<func_default>];
<array_datatype> ::= <sql_type> ARRAY [ := <array_constructor> ]
<array_constructor> ::= ARRAY ( <expression> [{,<expression>}...] )
<func_default> ::= { DEFAULT | := } <func_expr>
<func_expr> ::= !!An element of the type specified by <sql_type>
```

An array type has `<type>` as its element type. An array has a range from 1 to 2,147,483,647, which is the limitation of underlying structure.

Assign default values by specifying `<expression>`.

func_handler_list

Specifies a list of function handlers:

```
<func_handler_list> ::= <proc_handler_list>
```

See the CREATE PROCEDURE statement for more information about this clause.

func_stmt_list

Specifies statements for the functions:

```
<func_stmt_list> ::=
 <func_stmt>
 | <func_stmt_list> <func_stmt>
<func_stmt> ::=
 <proc_block>
 | <proc_assign>
 | <proc_single_assign>
 | <proc_if>
 | <proc_while>
 | <proc_for>
 | <proc_foreach>
 | <proc_exit>
 | <proc_signal>
 | <proc_resignal>
 | <proc_open>
 | <proc_fetch>
 | <proc_close>
```

For further information about the definitions in `<func_stmt>`, see the CREATE PROCEDURE statement.

func_return_statement

Specifies the return statement for the function:

```
<func_return_statement> ::= RETURN <function_return_expr>
<func_return_expr> ::= <table_variable> | <subquery>
```

Table functions must contain a return statement.

Description

There are two kinds of user-defined function (UDF): table and scalar. They differ according to the input or output parameter, supported functionality in the body, and the way they are consumed in SQL statements.

User-defined functions are read-only functions that are free of side effects: DDL nor DML statements are not allowed within the function body.

The CREATE FUNCTION statement creates read-only functions that are free of side effects. Neither DDL or DML statements (INSERT, UPDATE, and DELETE) is allowed in the function body. Also, other functions or procedures selected or called from the body of the function must be read only.

When you specify HEADERS ONLY, only the properties of the function are created along with the OID and no function dependencies exist. Once the headers are replaced with full function definitions, dependencies are created while the function OID remains the same. Dependencies appear in the OBJECT_DEPENDENCIES system view. HEADERS ONLY is useful when you need to create dependent functions.

Examples

Example 1 – Creates a table function:

```
CREATE FUNCTION scale (val INT)
```

```
 RETURNS TABLE (a INT, b INT) LANGUAGE SQLSCRIPT AS
 BEGIN
     RETURN SELECT a, :val * b AS  b FROM mytab;
 END;
```

Example 2 – Uses the scale function like a table. For example:

```
SELECT * FROM scale(10);
 SELECT * FROM scale(10) AS a, scale(10) AS b WHERE a.a =  b.a;
```

Example 3 – Creates a scalar function.

```
CREATE FUNCTION func_add_mul(x Double, y Double)
 RETURNS result_add Double, result_mul Double
 LANGUAGE SQLSCRIPT READS SQL DATA AS
 BEGIN
     result_add := :x + :y;
     result_mul := :x * :y;
 END;
```

Example 4 – Uses the `func_add_mul` function like a built-in function. For example:

```
CREATE TABLE TAB (a Double, b Double);
 INSERT INTO TAB VALUES (1.0, 2.0);
 INSERT INTO TAB VALUES (3.0, 4.0);

SELECT a, b, func_add_mul(a, b).result_add AS ADD, func_add_mul(a, b).result_mul
AS MUL FROM TAB ORDER BY a;
```

The SELECT statement returns the following results:

A	B	ADD	MUL
1	2	3	2
3	4	7	12

Example 5 – Creates a function `func_mul` that is assigned to a scalar variable in the `func_mul_wrapper` function:

```
CREATE FUNCTION func_mul(input1 INT)
 RETURNS output1 INT LANGUAGE SQLSCRIPT AS
 BEGIN
     output1 := :input1 * :input1;
 END;
 CREATE FUNCTION func_mul_wrapper(input1 INT)
 RETURNS output1 INT LANGUAGE SQLSCRIPT AS
 BEGIN
     output1 := func_mul(:input1);
 END;
 SELECT func_mul_wrapper(2) as RESULT FROM DUMMY;
```

The SELECT statement returns 4.

Example 6 – Creates the function FuncHeader as HEADER only. Later you use ALTER FUNCTION to replace the header with the full function definitions:

```
CREATE FUNCTION FuncHeader (input1 integer) RETURNS  output1  integer AS HEADER
ONLY;
ALTER FUNCTION FuncHeader (input1 integer) RETURNS  output1  integer
```

```
AS
BEGIN
  output1 := :input1 * :input1;
END;
```

7.7.8 CREATE INDEX Statement (Data Definition)

Creates an index on a table column.

> i Note
>
> SAP ASE restrictions:
>
> The following SQLScript index types are not supported.
>
> - TREE
> - BTREE
> - CPBTEE
> - INVERTED
> - INVERTED HASH
> - INVERTED VALUE
> - FULL
> - NOWAIT
> - ONLINE

Syntax

```
CREATE [UNIQUE] INDEX <index_name> ON <table_name>
    (<column_name_order_entry>[{, <column_name_order_entry>}...])
    [<global_index_order>]
    [<fillfactor>]
```

Syntax Elements

UNIQUE

Defines that a unique index should be created. A duplicates check occurs when the index is created and when a record is added to the table.

index_name

Specifies the name of the index to be created, with optional schema name.

```
<index_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <identifier>
```

column_name_order_entry

Specifies the columns to be used in the index, with an optional ordering.

```
<column_name_order_entry> ::= <column_name> [<column_order>]
```

column_order

Specifies whether the column index should be created in ascending or descending order. The default ordering is ASC.

```
<column_order> ::= ASC | DESC
```

global_index_order

Specifies the index ordering for all columns in the index.

```
<global_index_order> ::= ASC | DESC
```

> i Note
>
> `<column_order>` and `<global_index_order>` cannot be used when `<global_index_order>` is used.

fillfactor

Specifies how each node of a new index is filled. It is a percentage value of integer from 50 to 100, and the default value is 90.

```
<fillfactor> ::= FILLFACTOR <unsigned_integer>
```

Description

Creates an index on a table column.

Example

This example creates table `t`, then adds index `idx` on column `b` of table `t` with ascending order:

```
CREATE TABLE t (a INT, b NVARCHAR(10), c NVARCHAR(20)); CREATE INDEX idx ON t(b);
```

7.7.9 CREATE PROCEDURE Statement (Procedural)

Creates a procedure that uses the `SQLScript` programming language.

> i Note
>
> SAP ASE restrictions:

- The SEQUENTIAL EXECUTION option is not supported.
- The WITH RESULT VIEW option is not supported.
- The only LANGUAGE option supported is SQLScript.

Syntax

```
CREATE PROCEDURE <proc_name> [(<parameter_clause>)]
    [LANGUAGE <lang>]
    [SQL SECURITY <mode>]
    [DEFAULT SCHEMA <default_schema_name>]
    [READS SQL DATA ] AS
       {BEGIN
           <procedure_body>
        END
        | HEADER ONLY
       }
```

Syntax Elements

proc_name

The procedure name, with optional schema name:

```
<proc_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

parameter_clause

Specifies the input and output parameters for the procedure:

```
<parameter_clause> ::= <parameter> [,...]
```

parameter

Specifies a procedure parameter with its associated data type:

```
<parameter> ::= [<param_inout>] <param_name> <datatype>
```

param_inout

Specifies the parameter type:

```
<param_inout> ::= IN | OUT | INOUT
```

The default is IN. Input and output parameters must be explicitly typed; no un-typed tables are supported.

param_name

Specifies the parameter name:

```
<param_name> ::= <identifier>
```

datatype

Specifies the SQL type for the parameter:

```
<datatype> ::= <sql_type> | <table_type> | <table_type_definition>
```

The input and output parameters of a procedure can have any of the primitive SQL types or a table type. INOUT parameters can only be of scalar type.

sql_type

Specifies the data type of the variable:

```
<sql_type> ::=
 DATE
 | TIME
 | SECONDDATE
 | TIMESTAMP
 | TINYINT
 | SMALLINT
 | INTEGER
 | BIGINT
 | REAL
 | DOUBLE
 | BINTEXT
 | VARCHAR [ (<unsigned_integer>) ]
 | NVARCHAR [ (<unsigned_integer>) ]
 | ALPHANUM [ (<unsigned_integer>) ]
 | VARBINARY [ (<unsigned_integer>) ]
 | DECIMAL [ (<unsigned_integer> [, <unsigned_integer> ]) ]
```

table_type

Specifies a table type that was previously defined with the CREATE TYPE command:

```
<table_type> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

table_type_definition

Specifies a table type that is implicitly defined within the signature:

```
<table_type_definition> ::= TABLE (<column_list_definition>)
<column_list_definition> ::= <column_elem>[,...]
<column_elem> ::= <column_name> <data_type>
<column_name> ::= <identifier>
```

LANGUAGE

Specifies the programming language that is used in the procedure:

```
LANGUAGE <lang>
<lang> ::= SQLSCRIPT
```

Only `SQLSCRIPT` is supported.

SQL SECURITY

Specifies the security mode for the procedure:

```
SQL SECURITY <mode>
<mode> ::= DEFINER | INVOKER
```

DEFINER

(Default) Specifies that the procedure is executed with the privileges of the user who defined the procedure.

INVOKER

Specifies that the procedure is executed with the privileges of the user who invoked the procedure.

DEFAULT SCHEMA

Specifies the schema for unqualified objects in the procedure body:

```
DEFAULT SCHEMA <default_schema_name>
<default_schema_name> ::= <identifier>
```

If nothing is specified, then the current_schema of the session is used.

READS SQL DATA

Specifies that the procedure is read-only and side-effect free; it does not make modifications to the database data or its structure.

The procedure cannot contain DDL or DML statements, and it only calls other read-only procedures. The advantage of using this parameter is that certain optimizations are available for read-only procedures.

procedure_body

Defines the main body of the procedure according to the programming language selected:

```
<procedure_body> :=
 [<proc_decl_list>] [<proc_handler_list>] <proc_stmt_list>
```

proc_decl_list

Declares the condition handler:

```
<proc_decl_list> ::= <proc_decl> [{<proc_decl>}…]
<proc_decl> ::=
DECLARE {
   <proc_variable>
   |<proc_table_variable>
   |<proc_cursor>
   |<proc_condition>
  };
<proc_table_variable> ::= <variable_name_list> <table_type_definition>
<proc_variable>::=
 <variable_name_list> [CONSTANT] {<sql_type>|<array_datatype>}
   [NOT NULL] [<proc_default>]
<variable_name_list> ::= <variable_name>[{, <variable_name>}...]
<column_list_elements> ::= (<column_definition>[{,<column_definition>}...])
<array_datatype> ::= <sql_type> ARRAY [ := <array_constructor> ]
<array_constructor> ::= ARRAY (<expression> [ { , <expression> }...] )
<proc_default> ::= (DEFAULT | ':=' ) <value>|<expression>
<value> ::= !! An element of the type specified by <type> or an expression
<proc_cursor> ::=
 CURSOR <cursor_name> [ ( proc_cursor_param_list ) ] FOR <subquery>;
<proc_cursor_param_list> ::= <proc_cursor_param>
[{,<proc_cursor_param>}...]
<variable_name> ::= <identifier>
<cursor_name> ::= <identifier>
<proc_cursor_param> ::= <param_name> <datatype>
<proc_condition> ::=
 <variable_name> CONDITION | <variable_name> CONDITION FOR <sql_error_code>
```

proc_handler_list

Declares exception handlers to catch SQL exceptions:

```
<proc_handler_list> ::= <proc_handler> [{, <proc_handler>}...]
<proc_handler> ::= DECLARE EXIT HANDLER FOR <proc_condition_value_list>
<proc_stmt>;
```

proc_condition_value_list

Specifies one or more condition values:

```
<proc_condition_value_list> ::=
 <proc_condition_value> [{, <proc_condition_value>}...]
```

proc_condition_value

Specifies a specific error code number or condition name declared on condition variable:

```
<proc_condition_value> ::=
 SQLEXCEPTION
 | SQLWARNING
 | <sql_error_code>
 | <condition_name>
```

proc_stmt_list

Specifies statements for the procedure body:

```
<proc_stmt_list> := {<proc_stmt>}...
<proc_stmt> ::=
 <proc_block>
 | <proc_assign>
 | <proc_single_assign>
 | <proc_if>
 | <proc_loop>
 | <proc_while>
 | <proc_for>
 | <proc_foreach>
 | <proc_exit>
 | <proc_continue>
 | <proc_signal>
 | <proc_resignal>
 | <proc_sql>
 | <proc_open>
 | <proc_fetch>
 | <proc_close>
 | <proc_call>
 | <proc_exec>
 | <proc_return>
```

proc_block

Nests a section of the procedure by using BEGIN and END terminals:

```
<proc_block> ::=
 BEGIN <proc_block_option>
   [<proc_decl_list>]
   [<proc_handler_list>]
   <proc_stmt_list>
 END;
```

```
<proc_block_option> ::=
 [SEQUENTIAL EXECUTION] [AUTONOMOUS TRANSACTION]
 | [AUTONOMOUS TRANSACTION] [SEQUENTIAL EXECUTION]
```

The autonomous transaction is independent from the main procedure. Changes made and committed by an autonomous transaction can be stored in persistency regardless of commit/rollback of the main procedure transaction. The end of the autonomous transaction block has an implicit commit.

proc_assign

Assigns values to variables:

```
<proc_assign> ::=
 <variable_name> := { <expression> | <array_function> };
 | <variable_name> '[' <expression> ']' := <expression>;
```

`<expression>` is either a simple expression such as a character, a date, or a number, or it can be a scalar function.

proc_single_assign

Specifies data assignment for table type variables, and binds the tabular result of a valid SELECT statement on the right-hand side to the table-type variable on the left-hand side:

```
<proc_single_assign> ::=
 <variable_name> = <subquery>
 |  <variable_name> = <proc_ce_call>
 |  <variable_name> = <proc_apply_filter>
 |  <variable_name> = <unnest_function>
 |  <variable_name> = <match_merge_op>
```

```
<proc_multi_assign> ::= (<var_name_list>) := <function_expression>
```

`<function_expression>` is a scalar user-defined function and the number of elements in `<var_name_list>` must be equal to the number of output parameters of the scalar user-defined function.

proc_if

Controls the execution flow with conditionals:

```
<proc_if> ::=
 IF <condition> THEN [SEQUENTIAL EXECUTION]
    [<proc_decl_list>]
    [<proc_handler_list>]
    <proc_stmt_list>
    [<proc_elsif_list>]
    [<proc_else>]
 END IF;
<proc_elsif_list> ::=
 ELSEIF <condition> THEN [SEQUENTIAL EXECUTION]
   [<proc_decl_list>]
   [<proc_handler_list>]
   <proc_stmt_list>
<proc_else> ::=
 ELSE [SEQUENTIAL EXECUTION]
   [<proc_decl_list>]
   [<proc_handler_list>]
   <proc_stmt_list>
```

proc_loop

Uses a loop to repeatedly execute a set of statements:

```
<proc_loop> ::=
 LOOP [SEQUENTIAL EXECUTION]
    [<proc_decl_list>]
    [<proc_handler_list>]
    <proc_stmt_list>
 END LOOP;
```

proc_while

Specifies that a set of trigger statements is repeatedly called while a condition is true:

```
<proc_while> ::=
 WHILE <condition> DO [SEQUENTIAL EXECUTION]
    [<proc_decl_list>]
    [<proc_handler_list>]
    <proc_stmt_list>
 END WHILE;
```

proc_for

Specifies that a FOR - IN loop iterates over a set of data:

```
<proc_for> ::=
 FOR <column_name> IN [ REVERSE ] <expression> [...]  <expression>
   DO [SEQUENTIAL EXECUTION]
     [<proc_decl_list>]
     [<proc_handler_list>]
     <proc_stmt_list>
 END FOR;
```

proc_foreach

Specifies that FOR - EACH loops to iterate over all elements in a set of data:

```
<proc_foreach> ::=
 FOR <column_name> AS <column_name> [<open_param_list>]
   DO [SEQUENTIAL EXECUTION]
     [<proc_decl_list>]
     [<proc_handler_list>]
     <proc_stmt_list>
 END FOR;
<open_param_list> ::= ( <expression> [ { , <expression> }...] )
```

proc_exit

Terminates a loop:

```
<proc_exit> ::= BREAK;
```

proc_continue

Skips a current loop iteration and continues with the next value:

```
<proc_continue> ::= CONTINUE;
```

proc_signal

Explicitly raises an exception from within your trigger procedures:

```
<proc_signal> ::=
 SIGNAL <signal_value> [<set_signal_info>];
```

proc_resignal

Raises an exception on the action statement in an exception handler:

```
<proc_resignal> ::=
 RESIGNAL [<signal_value>] [<set_signal_info>];
```

If you do not specify an error code, RESIGNAL throws the caught exception.

signal_value

Specifies a SIGNAL or RESIGNAL for a signal name or an SQL error code:

```
<signal_value> ::= <signal_name> | <sql_error_code>
<signal_name> ::= <identifier>
<sql_error_code> ::= <unsigned_integer>
```

set_signal_info

Specifies that an error message is delivered to users when a specified error is thrown during procedure execution:

```
<set_signal_info> ::= SET MESSAGE_TEXT = '<message_string>'
<message_string> ::= <any_character>
```

proc_sql

Specifies a subquery:

```
<proc_sql> ::=
 <subquery>
 | <select_into_stmt>
 | <insert_stmt>
 | <delete_stmt>
 | <update_stmt>
 | <replace_stmt>
 | <call_stmt>
 | <create_table>
 | <drop_table>
```

select_into_stmt

Specifies a query:

```
<select_into_stmt> ::=
 SELECT <select_list> INTO <var_name_list> <from_clause>
   [<where_clause>]
   [<group_by_clause>]
   [<having_clause>]
   [{<set_operator> <subquery>, ... }]
   [<order_by_clause>]
   [<limit>];
```

For information on `<select_list>`, `<from_clause>`, `<where_clause>`, `<group_by_clause>`, `<having_clause>`, `<set_operator>`, `<subquery>`, `<order_by_clause>`, and `<limit>`, see the SELECT statement.

var_name_list

Specifies a variable:

```
<var_name_list> ::= <var_name>[{, <var_name>}...]
<var_name> ::= <identifier>
```

`<var_name>` is scalar variable. You can assign selected item value to this scalar variable.

proc_open

Controls cursor operations:

```
<proc_open> ::=
 OPEN <cursor_name> [ <open_param_list>];
<proc_fetch> ::=
 FETCH <cursor_name> INTO <column_name_list>;
<proc_close> ::= CLOSE <cursor_name>;
```

proc_exec

Makes a dynamic SQL call:

```
<proc_exec> ::= {EXEC | EXECUTE IMMEDIATE} <proc_expr>;
```

proc_return

Returns a value from a procedure:

```
<proc_return> ::= RETURN [<proc_expr>];
```

Description

The CREATE PROCEDURE statement creates a procedure using the specified programming language `<lang>`.

Examples

Example - Creating an SQL Procedure

Create an SQLScript procedure with the following definition.

```
CREATE PROCEDURE orchestrationProc
 LANGUAGE SQLSCRIPT AS
 BEGIN
   DECLARE v_id BIGINT;
   DECLARE v_name VARCHAR(30);
   DECLARE  v_pmnt BIGINT;
   DECLARE v_msg VARCHAR(200);
   DECLARE CURSOR c_cursor1 (p_payment BIGINT) FOR
     SELECT id, name, payment FROM control_tab
       WHERE payment > :p_payment ORDER BY id ASC;
   CALL init_proc();
   OPEN c_cursor1(250000);
   FETCH c_cursor1 INTO v_id, v_name, v_pmnt; v_msg := :v_name || ' (id '
|| :v_id || ') earns ' || :v_pmnt || ' $.';
   CALL ins_msg_proc(:v_msg);
   CLOSE c_cursor1;
 END;
```

The procedure features a number of imperative constructs including the use of a cursor (with associated state) and local scalar variables with assignments.

7.7.10 CREATE ROLE Statement (Access Control)

Creates a new role.

Syntax

```
CREATE ROLE <role_name>
```

Syntax Elements

role_name

Specifies the name of the role to be created with optional schema name:

```
<role_name> ::= <identifier>
```

The name must not be identical to the name of an existing user or role.

Description

The database administrator and users with ROLE ADMIN privilege can create roles.

If you want to allow several database users to perform the same actions, then create a role, grant the needed privileges to this role, and then grant the role to the database users.

You can retrieve the role information from `sysusers` and `sysroles`. For example:

```
select name from sysusers su, sysroles sr where su.uid = sr.lrid
```

Example

Creates a role with the name role_for_work_on_my_schema:

```
CREATE ROLE role_for_work_on_my_schema;
```

7.7.11 CREATE SCHEMA Statement (Data Definition)

Creates a schema in the current database.

Syntax

```
CREATE SCHEMA <schema_name> [OWNED BY <user_name>]
```

Syntax Elements

schema_name

Specifies the name of the schema:

```
<schema_name> ::= <unicode_name>
```

Restrictions:

- The schema name that starts with "#" or includes unicode string of 3-byte CESU-8 is not supported.
- If the schema name comprises only digits, the maximum length is 78.

user_name

Specifies the name of the schema owner. If omitted, the current user is the owner of the schema:

```
<user_name> ::= <unicode_name>
```

Description

The CREATE SCHEMA statement creates a schema in the current database.

These users can create a schema:

- Users with sa_role
- dbo users
- Users with the CREATE SCHEMA privilege
- Users with the permission to create schema can do so for themselves or for others using the OWNED BY clause.

Examples

Example 1 – Creates the ASE_schema:

```
CREATE SCHEMA ASE_schema
```

Example 2 – Creates a schema named `my_schema`, which is owned by the user `user_bob`:

```
CREATE SCHEMA my_schema OWNED BY user_bob
```

Related Information

Working with Schemas [page 9]
DROP SCHEMA Statement (Data Definition) [page 222]

7.7.12 CREATE SEQUENCE Statement (Data Definition)

Creates a sequence that generates primary key values that are unique across multiple tables, and for generating default values for a table.

Syntax

```
CREATE SEQUENCE <sequence_name> [<sequence_parameter_list>]
   [RESET BY <subquery>]
```

Syntax Elements

sequence_name

Specifies the name of the sequence to be created, with optional schema name:

```
<sequence_name> ::= [<schema_name>.]<identifier>
```

sequence_parameter_list

Defines one or more sequence parameters:

```
<sequence_parameter_list> ::=
<sequence_parameter> [{<sequence_parameter>}...]
```

Where <sequence_parameter> is one of:

```
<sequence_parameter> ::=
 START WITH <start_value>
 | INCREMENT BY <increment_value>
 | MAXVALUE <max_value>
 | NO MAXVALUE
 | MINVALUE <min_value>
 | NO MINVALUE
 | CYCLE
 | NO CYCLE
 | CACHE <cache_size>
 | NO CACHE
```

START WITH start_value

Specifies the starting sequence value.

```
START WITH <start_value>
<start_value> ::= <signed_integer>
```

If you do not specify a value for the START WITH clause, then MINVALUE is used for ascending sequences and MAXVALUE is used for descending sequences.

INCREMENT BY increment_value

Specifies the amount that the next sequence value is incremented from the last value assigned:

```
INCREMENT BY <increment_value>
<increment_value> ::= <signed_integer>
```

The default is 1. Specify a negative value to generate a descending sequence. An error is returned if the INCREMENT BY value is 0.

MAXVALUE max_value

Specifies the maximum value that can be generated by the sequence:

```
MAXVALUE <max_value>
<max_value> ::= <signed_integer>
```

<max_value> must be between -4611686018427387903 and 4611686018427387902.

NO MAXVALUE

When you use NO MAXVALUE directive, the maximum value for an ascending sequence is 4611686018427387903 and the maximum value for a descending sequence is -1.

MINVALUE min_value

Specifies the minimum value that a sequence can generate:

```
MINVALUE <min_value>
<min_value> ::= <signed_integer>
```

<min_value> must be between -4611686018427387904 and 4611686018427387902.

NO MINVALUE

When you use NO MINVALUE directive, the minimum value for an ascending sequence is 1 and the minimum value for a descending sequence is -4611686018427387903.

CYCLE

When you use CYCLE directive, the sequence number restarts after it reaches its maximum or minimum value.

NO CYCLE

Specifies the default option. When you use NO CYCLE directive, the sequence number does not restart after it reaches its maximum or minimum value.

CACHE cache_size

Specifies the cache size with which a range of sequence numbers are cached in memory:

```
CACHE <cache_size>
<cache_size> ::= <unsigned_integer>
```

`<cache_size>` is an unsigned integer. An error is returned if the CACHE is less than 2 or greater than min(MAXVALUE - MINVALUE, 30000). 2 <= `<cache_size>` <= min(MAXVALUE - MINVALUE, 30000).

NO CACHE

Specifies the default option. When you use NO CACHE directive, the sequence number is not cached in memory:

RESET BY subquery

During the restart of the database, the database automatically executes the `<subquery>`, and the sequence value is restarted with the returned value.

If you do not specify RESET BY, then the sequence value is stored persistently in database. During the restart of the database, the next value of the sequence is generated from the saved sequence value.

For more information on subqueries, see the SELECT statement.

Description

A sequence generates unique integers for use by multiple users.

Use CURRVAL (for example, `SELECT <sequence_name>.CURRVAL FROM ...`) to get the current value of the sequence and NEXTVAL to get the next value of the sequence. CURRVAL is only valid after the next call of NEXTVAL.

Valid `nextval` and `currval` Uses	Invalid `nextval` and `currval` Uses
• In the `select` clause of a `select` statement that is not involved in the creation of a view, or a subquery that is part of a `delete`, `select`, or `update`. • In the `select` list of a subquery in an `insert` statement. • In the `values` clause of an `insert` statement. • In the `set` clause of an `update` statement.	• In the `where` clause of a `select` statement. • In subqueries in `delete`, `select`, or `update`. • In a `select` statements for creating a view. • In a `select` statements with the `distinct` operator. • In a `select` statements with a `group by` clause. • In a `select` statement that involves a `set` operation (`union`, `intersect`, or `minus`). • As a default value of a column in a `create table` or `alter table` statement.

Examples

Example 1 – This example creates a sequence, **mySequence**, starting it at 1000, and then queries the NEXTVAL value for the sequence. After you create a sequence, you cannot query the value of CURRVAL until the value is populated by querying the NEXTVAL value:

```
CREATE SEQUENCE mySequence START WITH 1000 INCREMENT BY 1;
SELECT mySequence.NEXTVAL FROM DUMMY;
```

MYSEQUENCE.NEXTVAL
1,000

Example 2 – Execute the following statement to view the value of CURRVAL:

```
SELECT mySequence.CURRVAL FROM DUMMY;
```

MYSEQUENCE.CURRVAL
1,000

Example 3 – Create a table, **myTable** with a column definition that references **mySequence.NEXTVAL**, and then query the contents of the table:

```
CREATE TABLE myTable (a INT);
INSERT INTO myTable VALUES (mySequence.NEXTVAL);
SELECT * FROM myTable;
```

A
1,001

Example 4 – Now insert another row into **myTable** and query the contents of the table again:

```
INSERT INTO myTable VALUES (mySequence.NEXTVAL);
SELECT * FROM myTable;
```

A
1,001
1,002

Related Information

Working with Sequences [page 11]
ALTER SEQUENCE Statement (Data Definition) [page 142]
DROP SEQUENCE Statement (Data Definition) [page 223]

7.7.13 CREATE TABLE Statement (Data Definition)

Creates a new table in the database.

i Note

SAP ASE restrictions:

- The following clauses are not supported in SAP ASE SQLScript:
    - `<auto_merge_option>`
    - `<group_option>`
    - `<location_clause>`
    - `<logging_option>`
    - `<replica_clause>`
    - `<schema_flexibility_option>`
    - `<series_clause>`
    - `<unload_priority_clause>`
    - `<unused_retention_period_option>`
    - `<with_association_clause>`
    - `<remote_property_clause>`
    - `<table_comment_string>`
    - `<asynchronous_replica_clause>`
- SAP ASE does not support the following table types:
    - COLUMN
    - HISTORY COLUMN
    - GLOBAL TEMPORARY COLUMN
    - LOCAL TEMPORARY COLUMN

Syntax

```
CREATE [<table_type>] TABLE <table_name>
                [<table_contents_source>]
                [<partition_clause>]

  |CREATE VIRTUAL TABLE <table_name> <remote_location_clause>
```

Syntax Elements

table_type

Defines the type of table storage organization.This feature is used with restrictions and/or is extended by the following:

```
<table_type> ::=
ROW
| GLOBAL TEMPORARY
| LOCAL TEMPORARY
```

ROW

(Default) If the majority of table access involves selecting a few records with all attributes selected, then ROW-based storage is preferable. The SAP HANA database uses a combination of table types to enable storage and interpretation in both ROW and COLUMN forms.

LOCAL TEMPORARY

Specifies that the table definition and data are visible only to the current session.

The table is truncated at the end of the session. Metadata exists for the duration of the session and is session specific, meaning only the session owner of the local temporary table can see it. Data in a local temporary table is session specific, meaning only the session owner of the local temporary table can insert/read/truncate the data. The data exists for the duration of the session, and data from the local temporary table is automatically dropped when the session is terminated.

Supported operations on local temporary tables are:

- Create without a primary key
- Truncate
- Drop
- Select
- Select into or insert
- Delete
- Update
- Upsert or replace

table_name

Specifies the table name:

```
<table_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

table_contents_source

Specifies the source from which the table definition is derived:

```
<table_contents_source> ::=
 (<table_element>, ...) [<with_association_clause>]
 | <like_table_clause>
 | [(<column_name>, ...)] <as_table_subquery>
```

table_element

Defines a table column with associated column or table constraint.

```
<table_element> ::=
 <column_definition> [<column_constraint>]
 | <table_constraint>
```

column_definition

Defines a table column:

```
<column_definition> ::= <column_name> <data_type>
    [<default_value_clause>]
    [<col_gen_as_ident>]
```

SAP ASE does not support the following options:

- `<lob_data_type>`
- `<column_store_data_type>`
- `<ddic_data_type>`
- `<col_gen_as_expression>`
- `<col_calculated_field>`
- `<schema_flexibility>`
- `<fuzzy_search_index>`
- `<fuzzy_search_mode>`
- `<load_unit>`
- `<column_comment_string>`

column_constraint

See column constraint [page 194].

table_constraint

See table constraint [page 194].

column_name

Specifies the column name.

```
<column_name> ::= <identifier>
```

data_type

Specifies the colun data type:

```
<data_type> ::=
 DATE
 | TIME
 | SECONDDATE
 | TIMESTAMP
 | TINYINT
 | SMALLINT
 | INTEGER
 | BIGINT
 | SMALLDECIMAL
 | REAL
 | DOUBLE
 | TEXT
 | BINTEXT
 | VARCHAR [ (<unsigned_integer>) ]
 | NVARCHAR [ (<unsigned_integer>) ]
 | ALPHANUM [ (<unsigned_integer>) ]
 | VARBINARY [ (<unsigned_integer>) ]
 | SHORTTEXT [ (<unsigned_integer>) ]
 | DECIMAL [ (<unsigned_integer> [, <unsigned_integer> ]) ]
 | FLOAT [ (<unsigned_integer>) ]
 | BOOLEAN
```

default_value_clause

Specifies a value to be assigned to the column if an INSERT statement does not provide a value for the column:

```
<default_value_clause> ::= DEFAULT <default_value_exp>
```

```
<default_value_exp> ::=
 NULL
 | <string_literal>
 | <signed_numeric_literal>
 | <unsigned_numeric_literal>
 | <datetime_value_function>
<datetime_value_function> ::=
 CURRENT_DATE
 | CURRENT_TIME
 | CURRENT_TIMESTAMP
 | CURRENT_UTCDATE
 | CURRENT_UTCTIME
 | CURRENT_UTCTIMESTAMP
```

column_constraint

Specifies the column constraint rules:

```
<column_constraint> ::=
 NULL
 | NOT NULL
 | [<constraint_name_definition>] <unique_specification>
 | [<constraint_name_definition>] <references_specification>
```

NULL

Allows NULL values in the column. If NULL is specified it is not considered a constraint, it represents that a column that may contain a null value. The default is NULL.

NOT NULL

Prohibits NULL values in the column.

unique_specification

Specifies unique constraints.

```
<unique_specification> ::=
 UNIQUE [<unique_tree_type_index>]
 | PRIMARY KEY [<unique_tree_type_index>
]
```

If the index type is omitted, the SAP ASE database chooses the appropriate index by considering the column data type. If you do not specify the index type, the SAP HANA database automatically selects an index type as follows.

UNIQUE

Specifies a column as a unique key. A composite unique key enables the specification of multiple columns as a unique key. With a unique constraint, multiple rows cannot have the same value in the same column. A UNIQUE column can contain multiple NULL values.

PRIMARY KEY Specifies a primary key constraint, which is a combination of a NOT NULL constraint and a UNIQUE constraint.:

references_specification

For `<references_specification>`, see References Specification [page 195].

table_constraint

The table constraint can be either a unique constraint, a referential constraint, or a check constraint:

```
<table_constraint> ::=
 <unique_constraint_definition>
 | <referential_constraint_definition>
 | <check_constraint_definition>
```

unique_constraint_definition

The unique specification:

```
<unique_constraint_definition> ::=
 <unique_specification> (<unique_column_name_list>)
```

unique_column_name_list

Specifies the unique column name list of one or more column names:

```
<unique_column_name_list> ::=
 <unique_column_name>[{, <unique_column_name>}...]
```

unique_column_name

Specifies a column name identifier:

```
<unique_column_name> ::= <identifier>
```

referential_constraint_definition

Specifies a referential constraint:

```
<referential_constraint_definition> ::=
 FOREIGN KEY (<referencing_column_name_list>) <references_specification>
```

Foreign key constraints over unique key columns are not supported. Self-referencing foreign key constraints are supported.

referencing_column_name_list

Specifies that the referencing column name list that can have one or more column names:

```
<referencing_column_name_list> ::=
 <referencing_column_name>[{, <referencing_column_name>}...]
```

referencing_column_name

The identifier of a referencing column:

```
<referencing_column_name> ::= <identifier>
```

references_specification

Specifies the referenced table, with optional column=name list and trigger action:

```
<references_specification> ::=
  REFERENCES <referenced_table> [(<referenced_column_name_list>)]
  [<referential_triggered_action>]
```

If `<referenced_column_name_list>` is specified, then there is a one-to-one correspondence between `<column_name>` of `<column_definition>` (see column definition [page 192]) and `<referenced_column_name>`. If not specified, then there is a one-to-one correspondence between `<column_name>` of `<column_definition>` and the column name of the referenced table's primary key.

referenced_column_name_list

Specifies the referenced column name list, which can have one or more column names.

```
<referenced_column_name_list> ::=
 <referenced_column_name>[{, <referenced_column_name>}...]
```

referenced_table

Specifies the identifier of a table to be referenced.

```
<referenced_table> ::= <identifier>
```

like_table_clause

Creates a table that has the same definition as `<like_table_name>` unless a corresponding property or WITHOUT option is specified.

```
<like_table_clause> ::=
 LIKE <like_table_name> [WITH NO DATA] [<like_without_option>]
```

All the column definitions with constraints, default values, and other properties, such as generated columns, schema flexibility, and so on, are copied from the table `<like_table_name>`. All table constraints, indexes, fulltext indexes, and the table location are also copied.

like_table_name

Specifies the name of the table being duplicated.

```
<like_table_name> ::=
 [[<database_name>.]<schema_name>.]<table_name>
<table_name> ::= <identifier>
```

Schema name and database name are optional. A database name can only be specified in a multitenant database container system where cross-database access is enabled for the given database name.

like_without_option

Specifies which properties are not copied from the `<like_table_name>` table. For example, when WITHOUT PARTITION is specified, the table is not partitioned.

as_table_subquery

Creates a table and fills it with the data computed by the `<subquery>`:

```
<as_table_subquery> ::= AS (<subquery>) [WITH DATA]
```

Only NOT NULL constraints are copied by this clause. If column_names are specified, then specified column_names override the column names from `<subquery>`. For more information about subqueries, see the SELECT statement topic.

unload_priority_clause

Specifies the priority of the table to be unloaded from memory:

```
<unload_priority_clause> ::= UNLOAD PRIORITY <unload_priority>
```

`<unload_priority>` is a number from 0 and 9, inclusive, where 0 means not-unloadable and 9 means earliest unload.

partition_clause

Partitions the table by using the specified rules:

```
<partition_clause> ::=
 PARTITION BY <hash_partition> [, <range_partition> | , <hash_partition>]
 | PARTITION BY <range_partition> [,<range_partition>]
 | PARTITION BY <roundrobin_partition> [,<range_partition>]
```

Partitioning by date values requires the format YYYYMMDD, YYYY-MM-DD, or YYYY/MM/DD. Date is the most granular date time value that you can use for partitioning.

You can find more information about table partitioning in the *SAP HANA Administration Guide*.

hash_partition

Partitions the created table by using a hash partitioning scheme:

```
<hash_partition> ::=
 HASH (<partition_expression> [{<partition_expression>,}...])
   PARTITIONS {<num_partitions> | GET_NUM_SERVERS()}
```

range_partition

Partitions the created table by using a range partitioning scheme:

```
<range_partition> ::=
 RANGE (<partition_expression>) (<range_spec>, ...)
```

roundrobin_partition

Partitions the created table by using a round-robin partitioning scheme:

```
<roundrobin_partition> ::=
 ROUNDROBIN PARTITIONS {<num_partitions> | GET_NUM_SERVERS()} [,
<range_partition>]
```

GET_NUM_SERVERS()

Returns the number of servers/partitions according to table placement. You can find more information on table placement in the *SAP HANA Administration Guide*.

range_spec

Specifies the range specifier for a partition. Ranges can be specified for positive values:

```
<range_spec> ::=
 {<from_to_spec> | <single_spec>}
   [{ {<from_to_spec> | <single_spec>},} ...]
   [, PARTITION OTHERS]
```

from_to_spec

Specifies a partition by using the lower and upper values of a `<partition_expression>`:

```
<from_to_spec> ::=
 PARTITION <lower_value> <= VALUES < <upper_value>
```

single_spec

Specifies a partition by using a single value of a `<partition_expression>`:

```
<single_spec> ::=
 PARTITION VALUE = <target_value>
```

PARTITION OTHERS

Specifies that all other values that are not covered by the partition specification are gathered into one partition.

partition_expression

Declares the specifier that segregates data into partitions:

```
<partition_expression> ::=
 <column_name>
 | YEAR(<column_name>)
 | MONTH(<column_name>)
```

lower_value

Specifies the lower value of a partition specifier:

```
<lower_value> ::=
 <string_literal>
 |<unsigned_numeric_literal>
```

upper_value

Specifies the upper value of a partition specifier:

```
<upper_value> ::=
 <string_literal>
 |<unsigned_numeric_literal>
```

target_value

Specifies the target value of a single partition specifier:

```
<target_value> ::=
 <string_literal>
 | <unsigned_numeric_literal>
```

num_partitions

Specifies the number of partitions to create for the table:

```
<num_partitions> ::= <unsigned_integer>
```

group_option_list

Specifies the group type, subtype, and name, and whether the table is the leader of the table group:

```
<group_option_list> ::= <group_option> [<group_option> ...]
<group_option> ::=
 GROUP TYPE <identifier>
 | GROUP SUBTYPE <identifier>
 | GROUP NAME <identifier> [ GROUP LEAD ]
```

GROUP LEAD sets the table as the leader of the table group it belongs to.

remote_location_clause

Identifies a remote object (table or view) from an existing remote source:

```
<remote_location_clause> ::=
 AT <source_name>..<owner>.<identifier>
<source_name> ::= <identifier>
<owner> ::= <identifier>
```

Description

The CREATE TABLE statement creates a table that does not contain data.

The CREATE VIRTUAL TABLE is an extension of the CREATE TABLE statement. It provides a way to access an existing table on a remote source from an SAP ASE instance. The list of remote columns is automatically imported into the virtual table.

> **i Note**
>
> To use CREATE TABLE, enable the `select into` database option. For example: `sp_dboption <dbname>,'select into', true`

Examples

Example 1 – Creates table A that has INTEGER-type for columns A and B. Column A has a primary key constraint:

```
CREATE TABLE A (A INT PRIMARY KEY, B INT);
```

Example 2 – Creates table C2 that has the same column data types and NOT NULL constraints as table A:

```
CREATE TABLE C2 AS (SELECT * FROM A)
```

Example 3 – Creates table F that has a foreign key that references column A of table R:

```
CREATE TABLE R (A INT PRIMARY KEY, B NVARCHAR(10));
 CREATE TABLE F (FK INT, B NVARCHAR(10), FOREIGN KEY(FK) REFERENCES R);
```

Example 4 – Creates a self-referencing foreign key:

```
CREATE TABLE SELF(A INTEGER PRIMARY KEY, B INTEGER);
ALTER TABLE SELF ADD CONSTRAINT FK_T1 FOREIGN KEY(B) REFERENCES SELF(A);
```

Related Information

Sequence Within a Table as an Identity Column [page 14]

7.7.14 CREATE TRIGGER Statement (Data Definition)

Creates a trigger on a table or view.

> **i Note**
>
> SAP ASE restrictions:

- The AFTER value and INSTEAD OF value for <trigger_action_time> are not supported.

Syntax

```
CREATE TRIGGER <trigger_name> <trigger_action_time> <trigger_event_list>
   ON <subject_table_name>
   [REFERENCING <transition_list>]
   [<for_each_row>]
   [ <trigger_order_clause> ]
      BEGIN
         [<trigger_decl_list>]
         [<proc_handler_list>]
        <trigger_stmt_list>
      END
```

Syntax Elements

trigger_name

Specifies the name of the trigger to be created, with optional schema name:

```
<trigger_name> ::= [<schema_name>.]<identifier>

<schema_name> ::= <unicode_name>
```

trigger_action_time

Specifies when the trigger action should occur:

```
<trigger_action_time> ::=
 BEFORE
```

BEFORE

Specifies that the trigger is executed before the DML operation on a table.

trigger_event_list

Specifies the data modification command that activates the trigger action:

```
<trigger_event_list> ::=
 <trigger_event>
 | <trigger_event_list> OR <trigger_event>

<trigger_event> ::=
 INSERT
 | DELETE
 | UPDATE [[EXCEPT] OF <column_name_list>]

<column_name_list> ::= <column_name> [{, <column_name>}...]

<column_name> ::= <identifier>
```

If <column_name_list> is provided with the UPDATE OF clause, then the UPDATE trigger only fires if a column specified in the list is updated. If <column_name_list> is provided with the UPDATE EXCEPT OF clause, then the UPDATE trigger is not fired if a column specified in the list is updated. The UPDATE trigger is fired regardless of the UPDATE EXCEPT OF clause if columns that are not specified in the list are updated together.

subject_table_name

Specifies the subject table name:

```
<subject_table_name> ::= <identifier>
```

REFERENCING transition_list

When a trigger transition variable is declared, the trigger can access records that are being changed by the DML triggering the trigger.

transition_list

Specifies one or more transition list entries:

```
<transition_list> ::= <transition> [{, <transition>}...]
```

transition

Specifies a transition variable or table variable:

```
<transition> ::= <transition_variable> | <transition_table>
```

transition_variable

During row-level trigger execution, `<trans_var_name>.<column_name>` represents a single record from the corresponding column that is being changed by the DML:

```
<transition_variable> ::= {OLD | NEW} ROW [AS] <trans_var_name>

<trans_var_name> ::= <identifier>
```

OLD

Specifies that you can access the old row of the DML in the trigger. This is the row that is replaced by an update or a deleted old row. UPDATE triggers and DELETE triggers can have OLD ROW transition variables or OLD TABLE transition table variables.

NEW

Specifies that you can access the new record of the DML in the trigger. This is the row that is inserted or the new updated row.

UPDATE triggers and INSERT triggers can have NEW ROW transition variables or NEW TABLE transition table variables.

for_each_row

Specifies whether the trigger is called in a row-wise or statement-wise fashion:

```
<for_each_row> ::= FOR EACH { ROW | STATEMENT }
```

The default is ROW.

ROW

Specifies that a row-level trigger is used. This is fired once for each row affected by the triggering event.

trigger_order_clause

Specifies the order in which the triggers are executed:

```
<trigger_order_clause> ::= FOLLOWS | PRECEDES <trigger_name_list>
<trigger_name_list> ::= <trigger_name> [,...]
```

trigger_decl_list

Specifies the trigger declaration:

```
<trigger_decl_list> ::= {DECLARE <trigger_decl>}...
```

trigger_decl

Declares trigger variables or trigger conditions:

```
<trigger_decl> ::=
 <trigger_var_decl>
 | <trigger_condition_decl>
```

The declared variable can be used as a scalar value assignment or referenced in a trigger SQL statement.

trigger_var_decl

Specifies the trigger variable declaration:

```
<trigger_var_decl> ::=
 <var_name> [CONSTANT] <data_type> [<not_null>] [<trigger_default_assign>];
```

var_name

Specifies the identifier of the trigger variable:

```
<var_name> ::= <identifier>
```

CONSTANT

Specifies that you cannot change the variable during trigger execution.

data_type

Specifies the data type of the trigger variable:

```
<data_type> ::=
 DATE
 | TIME
 | SECONDDATE
 | TIMESTAMP
 | TINYINT
 | SMALLINT
 | INTEGER
 | BIGINT
 | SMALLDECIMAL
 | DECIMAL
 | REAL
 | DOUBLE
 | VARCHAR
 | NVARCHAR
 | ALPHANUM
 | SHORTTEXT
 | VARBINARY
 | BLOB
 | CLOB
 | NCLOB
 | TEXT
```

```
  | BINTEXT
```

not_null

Specifies the not null condition:

```
<not_null> ::= NOT NULL
```

trigger_default_assign

Specifies the default value of the trigger variable:

```
<trigger_default_assign> ::= DEFAULT <expression> | := <expression>
```

trigger_condition_decl

Specifies the condition handler declaration:

```
<trigger_condition_decl> ::=
 <condition_name> CONDITION;
 | <condition_name> CONDITION FOR <sql_error_code>;
```

condition_name

Specifies a declared condition name that you can reference in an exception handler:

```
<condition_name> ::= <identifier>
```

sql_error_code

Specifies the error code for exception handling:

```
<sql_error_code> ::= SQL_ERROR_CODE <int_const>
```

proc_handler_list

Declares exception handlers to catch SQL exceptions:

```
<proc_handler_list> ::= {<proc_handler>}...

<proc_handler> ::=
 DECLARE EXIT HANDLER FOR <proc_condition_value_list> <trigger_stmt>
```

proc_condition_value_list

Specifies one or more condition values:

```
<proc_condition_value_list> ::=
 <proc_condition_value> [{, <proc_condition_value>}]
```

proc_condition_value

```
<proc_condition_value> ::=
 SQLEXCEPTION
 | SQLWARNING
 | <sql_error_code>
 | <condition_name>
```

You can use a specific error code number or condition name declared on a condition variable.

Description

A trigger is special kind of stored procedure that automatically executes when an event occurs on a given table or view. The CREATE TRIGGER command defines a set of statements that are executed when a given operation (INSERT, UPDATE, DELETE) takes place on a given subject table or subject view. Only database users with the TRIGGER privilege for the given `<subject_table_name>` are allowed to create a trigger for that table or view.

The following limitations apply to triggers:

- Statement-level triggers are only supported in row-store tables. You also cannot convert a row table with a statement trigger into column table by using ALTER TABLE.
- Modification (any INSERT, UPDATE, DELETE, REPLACE) of a subject table that a trigger is defined on is not allowed in the trigger body.
- The trigger on a partitioned table cannot access the subject table, while a trigger on a non-partitioned table can execute the SELECT statement on its subject table.
- When a subject table is accessed in a trigger body, it does not always show row-wise result in case of batch updates due to performance reasons.
- The maximum trigger number per single table and per DML is 1024, which means a table can have maximum 1024 INSERT triggers, 1024 UPDATE triggers, and 1024 DELETE triggers at the same time.
- In the case of multiple triggers, trigger execution order is not guaranteed. To ensure a certain set of trigger execution order, use a single unified trigger instead of multiple triggers.
- Procedures calls inside trigger action body are supported. The procedure that is called in a trigger only can have SQL statements which are suitable for a trigger body.
- Transition variable modifications ares allowed in the BEFORE trigger. Internal column ($trexkey$, $rowid$, concat column) or generated column modifications are not allowed.
- Trigger actions that are unsupported include:
  - Result-set assignments like selecting a result set assignment into a table type
  - Dynamic SQL executions like building SQL statements dynamically at the runtime of a SQLScript

Examples

Example 1 – Basic trigger usage

Create a table that the trigger is created for:

```
CREATE TABLE TARGET ( A INT);
```

Create a table that the trigger accesses and modifies:

```
CREATE TABLE SAMPLE ( A INT);
```

Create the following trigger:

```
CREATE TRIGGER TEST_TRIGGER
 AFTER INSERT ON TARGET FOR EACH ROW
 BEGIN
     DECLARE SAMPLE_COUNT INT;
     SELECT COUNT(*) INTO SAMPLE_COUNT FROM SAMPLE;
     IF :SAMPLE_COUNT = 0
     THEN
       INSERT INTO SAMPLE VALUES(5);
     ELSEIF :SAMPLE_COUNT = 1
     THEN
       INSERT INTO SAMPLE VALUES(6);
     END IF;
 END;
```

TEST_TRIGGER is executed after any record insert execution for TARGET table:

```
INSERT INTO TARGET VALUES (1);
 SELECT * FROM SAMPLE;
```

The SELECT statement returns 5, since the SAMPLE table record count is zero at the first insert attempt and the trigger TEST_TRIGGER inserts 5 into the SAMPLE table:

```
INSERT INTO TARGET VALUES (2);
 SELECT * FROM SAMPLE;
```

The SELECT statement returns 6, since on the second insertion to the TARGET table, the trigger inserts 6 into the SAMPLE table because its count is now two.

Example 2 – Trigger ordering

Create the trigger TRIGGER_TEST_1 that is executed for each row following the TRIGGER_TEST_2 and TRIGGER_TEST_3 triggers:

```
CREATE TRIGGER TRIGGER_TEST_1
BEFORE INSERT ON MY_TAB
FOR EACH ROW FOLLOWS TRIGGER_TEST_2, TRIGGER_TEST_3
BEGIN
     DECLARE SAMPLE_COUNT INT;
     SELECT COUNT(*) INTO SAMPLE_COUNT FROM SAMPLE;
     IF :SAMPLE_COUNT = 0
     THEN
       INSERT INTO SAMPLE VALUES(5);
     ELSEIF :SAMPLE_COUNT = 1
     THEN
       INSERT INTO SAMPLE VALUES(6);
     END IF;
 END;
```

Example 3 – Trigger with AFTER UPDATE and a WHILE loop

```
CREATE TABLE TARGET ( A INT);
 CREATE TABLE SAMPLE ( A INT);
 CREATE TRIGGER TEST_TRIGGER_WHILE_UPDATE
 AFTER UPDATE ON TARGET
 BEGIN
     DECLARE found INT := 1;
     DECLARE val INT := 1;
     WHILE :found <> 0 DO
         SELECT count(*) INTO found FROM sample WHERE a = :val;
         IF :found = 0 THEN
             INSERT INTO sample VALUES(:val);
         END IF;
         val := :val + 1;
     END WHILE;
 END;
```

Example 4 – UPDATE trigger with column list

```
CREATE TABLE TARGET ( A INT, B INT);
CREATE TABLE SAMPLE ( A INT);
CREATE TRIGGER TEST_TRIGGER_UPDATE_COLUMN_LIST
AFTER UPDATE OF B ON TARGET
BEGIN
    INSERT INTO SAMPLE VALUES(1);
END;
INSERT INTO TARGET VALUES(1, 1);
UPDATE TARGET SET A = 0;
 -- does not fire the trigger
SELECT COUNT(*) FROM SAMPLE;
```

The SELECT statement returns 0.

```
UPDATE TARGET SET B = 0; -- does fire the trigger
SELECT COUNT(*) FROM SAMPLE;
```

The SELECT statement returns 1.

Example 5 – UPDATE EXCEPT OF trigger with column list

```
CREATE TABLE TARGET ( A INT, B INT);
CREATE TABLE SAMPLE ( A INT);
CREATE TRIGGER TEST_TRIGGER_UPDATE_EXCEPT_OF_COLUMN_LIST
AFTER UPDATE EXCEPT OF B ON TARGET
BEGIN
    INSERT INTO SAMPLE VALUES(1);
END;
INSERT INTO TARGET VALUES(1, 1);
UPDATE TARGET SET B = 0; -- does not fire the trigger
SELECT COUNT(*) FROM SAMPLE;
```

The SELECT statement returns 0.

```
UPDATE TARGET SET A = 0; -- does fire the trigger
SELECT COUNT(*) FROM SAMPLE;
```

The SELECT statement returns 1.

```
UPDATE TARGET SET A = 2, B = 2; -- does fire the trigger
SELECT COUNT(*) FROM SAMPLE;
```

The SELECT statement returns 2.

Example 6 – Trigger with an AFTER INSERT and FOR loop

```
CREATE TABLE TARGET ( A INT);
 CREATE TABLE control_tab(id INT PRIMARY KEY, name VARCHAR(30), payment
INT);
 CREATE TABLE message_box(message VARCHAR(200), log_time
TIMESTAMP);
 CREATE TRIGGER TEST_TRIGGER_FOR_INSERT
 AFTER INSERT ON TARGET
 BEGIN
     DECLARE v_id        INT := 0;
     DECLARE v_name      VARCHAR(20) := '';
     DECLARE v_pay       INT := 0;
     DECLARE v_msg       VARCHAR(200) := '';
     DELETE FROM message_box;
     FOR v_id IN 100 .. 103 DO
         SELECT name, payment INTO v_name, v_pay FROM control_tab WHERE id
= :v_id;
         v_msg := :v_name || ' has ' || TO_VARCHAR(:v_pay);
         INSERT INTO message_box VALUES (:v_msg, CURRENT_TIMESTAMP);
     END FOR;
 END;
```

Example 7 – Trigger with EXIT HANDLER examples

```
CREATE TABLE TARGET ( A INT);
 CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);

 CREATE TRIGGER MYTRIG_SQLEXCEPTION
 AFTER INSERT ON TARGET
 BEGIN
     DECLARE EXIT HANDLER FOR SQLEXCEPTION RESIGNAL;
     INSERT INTO MYTAB VALUES (1);
     INSERT INTO MYTAB VALUES (1);  -- expected unique violation error: 301
     -- not reached
 END;

 CREATE TRIGGER MYTRIG_SQL_ERROR_CODE
 AFTER UPDATE ON TARGET
 BEGIN
     DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 301 RESIGNAL;
     INSERT INTO MYTAB VALUES (1);
     INSERT INTO MYTAB VALUES (1);  -- expected unique violation error: 301
     -- not reached
 END;

 CREATE TRIGGER MYTRIG_CONDITION
 AFTER DELETE ON TARGET
 BEGIN
     DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 301;
     DECLARE EXIT HANDLER FOR MYCOND RESIGNAL;
     INSERT INTO MYTAB VALUES (1);
     INSERT INTO MYTAB VALUES (1);  -- expected unique violation error: 301
     -- not reached
 END;
```

Example 8 – Trigger with SIGNAL/RESIGNAL examples

```
CREATE TABLE TARGET ( A INT);
 CREATE TABLE MYTAB (I INTEGER PRIMARY KEY);
 CREATE TABLE MYTAB_TRIGGER_ERR (err_code INTEGER, err_msg VARCHAR(30));
 CREATE TRIGGER MYTRIG_SIGNAL
 AFTER INSERT ON TARGET
 BEGIN
     DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 10001;
     DECLARE EXIT HANDLER FOR MYCOND INSERT INTO MYTAB_TRIGGER_ERR VALUES
(::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE);
     INSERT INTO MYTAB VALUES (1);
     SIGNAL MYCOND SET MESSAGE_TEXT = 'my error message1';  -- signal immediately
     -- not reached
     INSERT INTO MYTAB VALUES (2);
 END;
 INSERT INTO SUBJECT VALUES (1)
 SELECT * FROM MYTAB_TRIGGER_ERR;
 (10001, my error message1)

 SELECT * FROM MYTAB;
```

The SELECT statement returns 1.

```
CREATE TRIGGER MYTRIG_RESIGNAL
 AFTER UPDATE ON TARGET
 BEGIN
     DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 10002;
     DECLARE EXIT HANDLER FOR MYCOND RESIGNAL;  -- 2. throws error with error
code 10002
```

```
     INSERT INTO MYTAB VALUES (1);
     SIGNAL MYCOND SET MESSAGE_TEXT = 'my error message2';  -- 1. signal
immediately
     -- not reached
     INSERT INTO MYTAB VALUES (2);
 END;
 UPDATE SUBJECT SET A = 100 WHERE A = 1;
 ERR-10002
 ERR-MSG:user-defined error: "TRIGGER_EXCEPTION"."MYTRIG_RESIGNAL": line 4 col
37 (at pos 210): [10002] (range 3) user-defined error exception: my error
message2
```

Example 9 – Transition variable

```
CREATE TABLE TARGET ( A INT, B VARCHAR(10));
 CREATE TABLE SAMPLE_OLD ( A INT, B VARCHAR(10));
 CREATE TABLE SAMPLE_NEW ( A INT, B VARCHAR(10));
 CREATE TABLE SAMPLE ( A INT, B VARCHAR(10));
 INSERT INTO TARGET VALUES ( 1, 'oldvalue');
 INSERT INTO TARGET VALUES ( 2, 'oldvalue');
 INSERT INTO TARGET VALUES ( 5, 'oldvalue');
 CREATE TRIGGER TEST_TRIGGER_VAR_UPDATE
 AFTER UPDATE ON TARGET
 REFERENCING NEW ROW mynewrow, OLD ROW myoldrow
 FOR EACH ROW
 BEGIN
  INSERT INTO SAMPLE_new VALUES(:mynewrow.a, :mynewrow.b);
  INSERT INTO SAMPLE_old VALUES(:myoldrow.a, :myoldrow.b);
  INSERT INTO SAMPLE VALUES(0, 'trigger');
 END;
 UPDATE TARGET SET b = 'newvalue' WHERE A < 3;
 SELECT * FROM TARGET;
 1, 'newvalue'
 2, 'newvalue'
 5, 'oldvalue'
 SELECT * FROM SAMPLE_NEW;
 1, 'newvalue'
 2, 'newvalue'
 SELECT * FROM SAMPLE_OLD;
 1, 'oldvalue'
 2, 'oldvalue'
 SELECT * FROM SAMPLE;
 0, 'trigger'
 0, 'trigger'
```

Example 10 – Transition table

```
CREATE TABLE TARGET ( A INT, B VARCHAR(10));
 CREATE TABLE SAMPLE_OLD ( A INT, B VARCHAR(10));
 CREATE TABLE SAMPLE_NEW ( A INT, B VARCHAR(10));
 CREATE TABLE SAMPLE ( A INT, B VARCHAR(10));
 INSERT INTO TARGET VALUES ( 1, 'oldvalue');
 INSERT INTO TARGET VALUES ( 2, 'oldvalue');
 INSERT INTO TARGET VALUES ( 5, 'oldvalue');
 CREATE TRIGGER TEST_TRIGGER_TAB_UPDATE
 AFTER UPDATE ON TARGET
 REFERENCING NEW TABLE mynewtab, OLD TABLE myoldtab
 FOR EACH STATEMENT
 BEGIN
  INSERT INTO SAMPLE_new SELECT * FROM :mynewtab;
  INSERT INTO SAMPLE_old SELECT * FROM :myoldtab;
  INSERT INTO SAMPLE VALUES(0, 'trigger');
 END;
 UPDATE TARGET SET b = 'newvalue' WHERE A < 3;
 SELECT * FROM TARGET;
 1, 'newvalue'
 2, 'newvalue'
 5, 'oldvalue'
 SELECT * FROM SAMPLE_NEW;
 1, 'newvalue'
 2, 'newvalue'
 SELECT * FROM SAMPLE_OLD;
 1, 'oldvalue'
 2, 'oldvalue'
 SELECT * FROM SAMPLE;
 0, 'trigger'
CREATE TABLE T1(A INTEGER PRIMARY KEY, B INTEGER);
CREATE COLUMN TABLE T2(A INTEGER PRIMARY KEY, C INTEGER);
CREATE VIEW V1 AS (SELECT T1.A A, B, C, T1.A+10 D FROM T1, T2 WHERE T1.A = T2.A);
CREATE TRIGGER TR1 INSTEAD OF INSERT ON V1 REFERENCING NEW ROW NEW
BEGIN
    INSERT INTO T1 VALUES(:NEW.A, :NEW.B);
    INSERT INTO T2 VALUES(:NEW.A, :NEW.C);
END;
INSERT INTO V1(A,B,C) VALUES(1,2,3);
SELECT FROM T1 ORDER BY A, B;
```

The SELECT statement returns 1, 2.

```
SELECT FROM T2 ORDER BY A, C;
```

The SELECT statement returns 1, 3.

```
SELECT FROM V1 ORDER BY A, B, C, D;
```

The SELECT statement returns 1, 2, 3, 11

```
CREATE TRIGGER TR2 INSTEAD OF UPDATE ON V1 REFERENCING OLD ROW OLD, NEW ROW NEW
BEGIN
    UPDATE T1 SET A=:NEW.A, B=:NEW.B WHERE A=:OLD.A;
    UPDATE T2 SET A=:NEW.A, C=:NEW.C WHERE A=:OLD.A;
END;
UPDATE V1 SET A = 5, B = 6, C = 7 WHERE A = 1;
SELECT FROM T1 ORDER BY A, B;
```

The SELECT statement returns 5, 6.

```
SELECT FROM T2 ORDER BY A, C;
```

The SELECT statement returns 5, 7.

```
SELECT FROM V1 ORDER BY A, B, C, D;
```

The SELECT statement returns 5, 6, 7, 11.

```
CREATE TRIGGER TR3 INSTEAD OF DELETE ON V1 REFERENCING OLD ROW OLD
BEGIN
    DELETE FROM T1 WHERE A = :OLD.A;
    DELETE FROM T2 WHERE A = :OLD.A;
END;
DELETE FROM V1;
SELECT FROM T1 ORDER BY A, B;
SELECT FROM T2 ORDER BY A, C;
SELECT FROM V1 ORDER BY A, B, C, D;
```

7.7.15 CREATE TYPE Statement (Procedural)

Defines tape type for tabular data used in CREATE PROCEDURE and CREATE FUNCTION commands.

Syntax

```
CREATE TYPE <type_name>
   AS TABLE (<column_definition> [{null | not null }] [ ,...n ])
```

Syntax Elements

type_name

Identifies the table type to be created and, optionally, in which schema the creation should take place:

```
<type_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

column_definition

Defines a table column:

```
<column_definition> ::=
 <column_name> <data_type>
```

column_name

Specifies the table column name:

```
<column_name> ::= <identifier>
```

data_type

Specifies the data type for the column. System data types are allowed:

```
<data_type> ::=
 DATE | TIME | SECONDDATE | TIMESTAMP | TINYINT
 | SMALLINT | INTEGER | BIGINT | DECIMAL
 | REAL | DOUBLE | VARCHAR | NVARCHAR | ALPHANUM
 | SHORTTEXT | VARBINARY | BLOB | CLOB | NCLOB
 | TEXT | BINTEXT
```

Description

The syntax for defining table types follows the SQL syntax for defining new types. The table type is specified using a list of attribute names and primitive data types. For each table type, attributes must have unique names.

The name length is limited to 255 characters.

DML statements are not allowed for table type.

Constraints cannot be used, with the exception of NULL/NOT NULL.

Example

Creates a table type called tt_publishers:

```
CREATE TYPE tab_type AS TABLE (I INT, A VARCHAR)
CREATE TABLE mytab(I INT, A VARCHAR)
INSERT INTO mytab values(1, 'Have9try')
CREATE PROCEDURE test_table
AS
BEGIN
DECLARE tab tab_type;
tab = SELECT * FROM mytab;
SELECT * FROM :tab;
END;
go
CALL test_table
go
I           A
----------- -
1           H
(1 row affected)
(return status = 0)
```

7.7.16 CREATE USER Statement (Access Control)

Creates a new database user.

i Note

SAP ASE restrictions:

- The `CREATE USER` statement only supports creating database users specifying a users' password.

Syntax

```
CREATE USER <user_name>  { <password_option> }
```

Syntax Elements

user_name

Specifies the user name of the user to be created:

```
<user_name> ::= <unicode_name>
```

password_option

Specifies the user password of the user being created:

```
<password_option> ::= PASSWORD <password>
```

> i Note
>
> The password rules include a minimal password length and a definition of which character types (lower, upper, digit, special characters) have to be part of the password.

Description

The database administrator or user with USER ADMIN system privilege can create users.

The specified user name cannot be identical to the name of an existing user, role, or schema.

Users in the database can be authenticated by internal authentication mechanism using a password.

Example

The following code example shows you how to create a user **`T12345`** with a password **`Password123`**:

```
CREATE USER T12345 PASSWORD Password123
```

7.7.17 CREATE VIEW Statement (Data Definition)

Creates a view on the database.

> i Note
>
> SAP ASE restrictions:
>
> - The `<subquery>` clause of the CREATE VIEW command does not support the `<limit_clause>` and `<order_by_clause>` clauses.

Syntax

```
CREATE VIEW <view_name>
```

```
    [(<column_name_list>)] AS <subquery>
```

Syntax Elements

view_name

Creates the specified view, with an optional schema name:

```
<view_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

column_name_list

Specifies the column names for the view:

```
<column_name_list> ::= <column_name>[{, <column_name>}...]
<column_name> ::= <identifier>
```

When a column name is specified along with the view name, a query result is displayed with that column name. If a column name is omitted, then a query result gives an appropriate name to the column automatically. The number of column names has to be the same as the number of columns returned from `<subquery>`.

Description

The CREATE VIEW statement effectively creates a virtual table based on the results of an SQL statement. It is not a real table as it does not contain any data.

Update operations on views are supported if the following conditions are met:

- Each column in the view maps to the column of a single table.
- If a column in a view base table has a NOT NULL constraint without a default value, then the column must be included in view columns so that inserts can be performed.
- The view cannot contain an aggregate or analytic function in a SELECT list. For example, the following functions are not allowed:
    - TOP, SET, or DISTINCT operator in a SELECT list.
    - ORDER BY clause.
    - LIMIT clause.
- The view cannot contain a subquery in a SELECT list.
- The view cannot contain a sequence value (CURRVAL, NEXTVAL).
- The view cannot contain a column view as the base view.

If the base views or tables are updatable, then a view on the base views or tables can also be updatable if the above conditions are met.

Examples

Example 1 – Creates table A:

```
CREATE TABLE A (A INT PRIMARY KEY, B INT);
```

Example 2 – Creates a view, v, that selects all records from table A:

```
CREATE VIEW v AS SELECT * FROM A;
```

7.7.18 DELETE Statement (Data Manipulation)

Deletes records from a table where a specified condition is met.

i Note

SAP ASE restrictions:

- The `HISTORY` parameter and `<hint_clause>` are not supported in SAP ASE SQLScript.

Syntax

```
DELETE  FROM <table_name> [WHERE <condition>]
```

Syntax Elements

FROM table_name

Specifies the table name where the deletion is to occur, with optional schema name:

```
<table_name> ::= [<schema_name>.]<identifier>

<schema_name> ::= <unicode_name>
```

WHERE condition

Specifies the conditions where the command should be performed:

```
<condition> ::=
 <condition> OR <condition>
 | <condition> AND <condition>
 | NOT <condition>
 | ( <condition> )
 | <predicate>
```

Description

If the WHERE clause is omitted, then DELETE removes all records from a table.

When using the DELETE HISTORY command, time travel queries referencing the deleted rows may still access these rows. To physically delete these rows, you must execute the following statements:

```
ALTER SYSTEM RECLAIM VERSION SPACE;
```

In some cases even the execution of the two statements above may not lead to physical deletion.

To check whether the rows are physically deleted, execute the following statement:

```
SELECT * FROM <table_name> WHERE <condition>
```

Examples

Example 1 - Standard delete operation

Create a table T and insert some data:

```
CREATE TABLE T (KEY INT PRIMARY KEY, VAL INT);
 INSERT INTO T VALUES (1, 1);
 INSERT INTO T VALUES (2, 2);
 INSERT INTO T VALUES (3, 3);
```

Delete from table T where the key column is equal to 1;:

```
DELETE FROM T WHERE KEY = 1;
```

After executing the above query, the contents of table T are as follows, showing that one row was deleted from the table:

KEY	VAL
2	2
3	3

Delete from the table using an array value:

```
DELETE FROM T1 WHERE 3 MEMBER OF C1;
```

7.7.19 DO BEGIN ... END Statement (Procedural)

Executes an anonymous block a single time.

i Note

SAP ASE restrictions:

- The `SEQUENTIAL EXECUTION` option is not supported.

Syntax

```
DO
   BEGIN
     <procedure_body>
   END
```

Syntax Elements

procedure_body

Specifies the anonymous block logic.

For more information about this clause, see the CREATE PROCEDURE statement.

Description

An anonymous block gets executed only a single time. All SQLScript statements supported in procedures are also supported in anonymous blocks. Compared to procedures, an anonymous block has no corresponding object created in the metadata catalog.

Anonymous blocks are defined and executed in a single step. Therefore, lifecycle handling like CREATE or DROP is not needed. Anonymous blocks neither have parameters nor container-specific properties (such as the language or the security mode). The body of an anonymous block is similar to a procedure body.

To return a result set, use a SELECT statement, as anonymous blocks do not have any parameters defined.

Examples

Example 1 – Executes an anonymous block that creates a table and inserts values into that table:

```
DO
BEGIN
```

```
    DECLARE I INTEGER;
    CREATE TABLE TAB1 (I INTEGER);
    FOR I IN 1..10 DO
        INSERT INTO TAB1 VALUES (:I);
    END FOR;
END;
```

Example 2 – Executes an anonymous block and return the result set by using a SELECT statement:

```
DO
BEGIN
    T1 = SELECT I, 10 AS J FROM TAB;
    T2 = SELECT I, 20 AS K FROM TAB;
    T3 = SELECT J, K FROM :T1, :T2 WHERE :T1.I = :T2.I;
    SELECT * FROM :T3;
END;
```

Example 3 – Executes an anonymous block that calls another procedure:

```
DO
BEGIN
    T1 = SELECT * FROM TAB;
    CALL PROC3(:T1, :T2);
    SELECT * FROM :T2;
END;
```

Example 4 – Executes an anonymous block that includes an exception handler:

```
DO
BEGIN
    DECLARE I, J INTEGER;
    BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
        IF ::SQL_ERROR_CODE = 288 THEN
            DROP TABLE TAB;
            CREATE TABLE TAB (I INTEGER PRIMARY KEY);
        ELSE
            RESIGNAL;
        END IF;
        CREATE TABLE TAB (I INTEGER PRIMARY KEY);
    END;
    FOR I in 1..3 DO
        INSERT INTO TAB VALUES (:I);
    END FOR;
    IF :J <> 3 THEN
        SIGNAL SQL_ERROR_CODE 10001;
    END IF;
END;
```

Example 5 – Executes an anonymous block that uses COMMIT and ROLLBACK:

```
DO
BEGIN
    CREATE TABLE TAB2 (K INT);
    COMMIT;
    DROP TABLE TAB;
    CREATE TABLE TAB (J INT);
    ROLLBACK;
    DELETE FROM TAB;
END;
```

Example 6 – Executes an anonymous block that calls a procedure to process the selected data:

```
DO
```

```
  BEGIN
    T1 = SELECT I, 10 AS J FROM TAB;
    T2 = SELECT I, 20 AS K FROM TAB;
    T3 = SELECT J, K FROM :T1, :T2 WHERE :T1.I = :T2.I;
    CALL PROC3(:T3, T4);
    SELECT * FROM :T4;
  END;
```

Exmaple 7 – Executes an anonymous block that runs a loop to insert data into a table:

```
DO
  BEGIN
    DECLARE I INTEGER;
    FOR I in 1..3 DO
        INSERT INTO TAB VALUES (:I);
      END FOR;
  END;
```

7.7.20 DROP FUNCTION Statement (Procedural)

Deletes a function from the database.

Syntax

```
DROP FUNCTION <func_name> [<drop_option>]
```

Syntax Elements

func_name

Specifies the name of the function to be dropped, with optional schema name.

```
<func_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

drop_option

Specifies whether a cascaded drop is performed.

```
<drop_option> ::= CASCADE | RESTRICT
```

When `<drop_option>` is not specified, a non-cascaded drop is performed. This only drops the specified function; dependent objects of the function are invalidated but not dropped.

The invalidated objects can be revalidated when an object that has same schema and object name is created.

CASCADE

Drops the function and dependent objects.

RESTRICT

Drops the function only when dependent objects do not exist. If this drop option is used and a dependent object exists an error is thrown.

Description

Drops a function created using CREATE FUNCTION from the database catalog.

Examples

Drop a function called my_func from the database using a non-cascaded drop.

```
DROP FUNCTION my_func;
```

7.7.21 DROP INDEX Statement (Data Definition)

Removes an index.

Syntax

```
DROP INDEX <index_name>
```

Syntax Elements

index_name

Drops the specified index, with the optional schema name:

```
<index_name> ::= [<schema_name>].<table_name>.<identifier>
```

Description

The DROP INDEX statement removes an index.

Examples

Example 1 – Creates table A and then add an index, i, on column b of table A:

```
CREATE TABLE A (a INT, b NVARCHAR(10), c NVARCHAR(20));
 CREATE INDEX i ON A(b);
```

Example 2 – Drops index i:

```
DROP INDEX i;
```

7.7.22 DROP PROCEDURE Statement (Procedural)

Deletes a procedure from the database.

Syntax

```
DROP PROCEDURE <proc_name> [<drop_option>]
```

Syntax Elements

proc_name

Specifies the name of the procedure to be dropped, and optionally, a schema name.

```
<proc_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

drop_option

Specifies whether a cascaded drop is performed.

```
<drop_option> ::= CASCADE | RESTRICT
```

When `<drop_option>` is not specified, then a non-cascaded drop be performed. Only the specified procedure is dropped; dependent objects of the procedure are invalidated, but they are not dropped.

The invalidated objects can be revalidated when an object that has same schema and object name is created.

CASCADE

Drops the procedure and dependent objects.

RESTRICT

Drops the procedure only when dependent objects do not exist. If this drop option is used and a dependent object exists, then an error is thrown.

Description

Drops a procedure created using CREATE PROCEDURE from the database catalog.

Examples

Drop a procedure called my_proc from the database using a non-cascaded drop.

```
DROP PROCEDURE my_proc;
```

7.7.23 DROP ROLE Statement (Access Control)

Drops a role.

Syntax

```
DROP ROLE <role_name>
```

Syntax Elements

role_name

Drops the specified role, with the optional schema name:

```
<role_name> ::= <identifier>
```

`<role_name>` must specify an existing role.

Description

The database administrator or a user with ROLE ADMIN system privilege can drop a role.

If a role has been granted to a user or another role, then it is revoked when the role is dropped. Revoking a role may lead to making some views inaccessible or making procedures not executable, which occurs if a view or procedures depends on any of the privileges of the dropped role.

Examples

Example 1 – Creates a role with the name role_for_work_on_my_schema:

```
CREATE ROLE role_for_work_on_my_schema;
```

Example 2 – Drops the role_for_work_on_my_schema role:

```
DROP ROLE role_for_work_on_my_schema;
```

7.7.24 DROP SCHEMA Statement (Data Definition)

Removes a schema.

Syntax

```
DROP SCHEMA <schema_name> [<drop_option>]
```

Syntax Elements

schema_name

Drops the specified schema:

```
<schema_name> ::= <unicode_name>
```

drop_option

Specifies the drop option to use:

```
<drop_option> ::= CASCADE | RESTRICT
```

CASCADE drops the schema and dependent objects. RESTRICT drops the schema only when dependent objects do not exist. If this drop option is used and a dependent object exists, then an error is thrown.

If you do not specify `<drop_option>`, then a non-cascaded drop is performed, which only drops the specified schema. Dependent objects of the schema are invalidated but not dropped.

Invalidated objects can be revalidated when an object that has same schema name is created.

Description

The DROP SCHEMA statement removes a schema.

Examples

Example 1 – Creates a schema named my_schema and a table named my_schema.t:

```
CREATE SCHEMA my_schema;
 CREATE TABLE my_schema.t (a INT);
```

Example 2 – Drops my_schema with a CASCADE option:

```
DROP SCHEMA my_schema CASCADE;
```

Permissions

- - Users can drop schema that they own
  - Users with sa_role can drop schemas owned by anyone

7.7.25 DROP SEQUENCE Statement (Data Definition)

Removes a sequence.

Syntax

```
DROP SEQUENCE <sequence_name> [<drop_option>]
```

Syntax Elements

sequence_name

Drops the specified sequence, with the optional schema name:

```
<sequence_name> ::= [<schema_name>.]<identifier>
```

drop_option

Specifies the drop option to use:

```
<drop_option> ::= CASCADE | RESTRICT
```

CASCADE drops the sequence and dependent objects. RESTRICT drops the sequence only when dependent objects do not exist. If this drop option is used and a dependent object exists, then an error is thrown.

If you do not specify <drop_option>, then a non-cascaded drop is performed, which only drops the specified sequence. Dependent objects of the sequence are invalidated but not dropped.

Description

Invalidated objects can be re-validated when an object that has same schema and object name is created.

Example

Example 1 – Creates a sequence named seq:

```
CREATE SEQUENCE seq START WITH 11;
```

Example 2 – Drops the sequence named seq:

```
DROP SEQUENCE seq;
```

Related Information

ALTER SEQUENCE Statement (Data Definition) [page 142]
CREATE SEQUENCE Statement (Data Definition) [page 187]
Working with Sequences [page 11]

7.7.26 DROP STATISTICS (Data Definition)

Drops data statistic objects the query optimizer uses to make decisions for query plans.

SAP ASE restrictions:

- The HAVING <match_properties> option is not supported.

Syntax

```
DROP STATISTICS ON <data_sources>
```

Syntax Element

ON data_sources

Specifies the data source(s) of the data statistics objects to be dropped:

```
<data_sources> ::=
 <table_name> [(<column_name>{, <column_name>}...)]
```

Description

You can specify multiple filtering clauses to refine the list of data statistics objects to drop.

Examples

Example 1 – Drop statistics from the `TEST` table:

```
DROP STATISTICS ON TEST;
```

Example 2 – Drop statistics from the `TEST.big_column` column:

```
DROP STATISTICS ON TEST(big_column);
```

Example 3 – Drops statistics from test, and columns `a` and `b`:

```
1> insert into test values(1,1)
2> go
(1 row affected)
1> refresh statistics on test
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    1
(1 row affected)
1> drop statistics on test
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    0
(1 row affected)
1> insert into test values(1,1)
2> go
(1 row affected)
1> refresh statistics on test(a,b)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    5
```

```
(1 row affected)
1> drop statistics on test
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    0
(1 row affected)
1> insert into test values(1,1)
2> go
(1 row affected)
1> refresh statistics on test(a,b)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    5
(1 row affected)
1> drop statistics on test(b)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    5
(1 row affected)
1> drop statistics on test(a)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    2
(1 row affected)
1> drop statistics on test(a,b)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    1
(1 row affected)
1> drop statistics on test
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    0
(1 row affected)
```

7.7.27 DROP TABLE Statement (Data Definition)

Removes a table from the database.

Syntax

```
DROP TABLE <table_name> [<drop_option>]
```

Syntax Elements

table_name

Drops the specified table, with the optional schema name:

```
<table_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

drop_option

Specifies the drop option to use:

```
<drop_option> ::= CASCADE | RESTRICT
```

CASCADE drops the table and dependent objects. RESTRICT drops the table only when dependent objects do not exist. If this drop option is used and a dependent object exists, then an error is thrown.

If you do not specify `<drop_option>`, then a non-cascaded drop is performed which only drops the specified table. Dependent objects of the table are invalidated but not dropped.

The invalidated objects can be re-validated when an object that has same schema and object name is created.

Description

The DROP TABLE statement removes a table from the database.

Example

Creates table A and then drop it immediately afterward:

```
CREATE TABLE A(C INT);
 DROP TABLE A;
```

7.7.28 DROP TRIGGER Statement (Data Definition)

Deletes a trigger.

Syntax

```
DROP TRIGGER <trigger_name> [<drop_option>]
```

Syntax Elements

trigger_name

Drops the specified trigger, with the optional schema name.

```
<trigger_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

drop_option

Specifies the drop option to use.

```
<drop_option> ::= CASCADE | RESTRICT
```

CASCADE drops the trigger and dependent objects. RESTRICT drops the trigger only when dependent objects do not exist. If this drop option is used and a dependent object exists, then an error is thrown.

If `<drop_option>` is not specified, then a non-cascaded drop is performed which only drops the specified trigger. Dependent objects of the trigger are invalidated but not dropped.

Description

Only database users with the TRIGGER privilege for the table on which the trigger was defined are allowed to drop a trigger for that table.

Invalidated objects can be re-validated when an object that has same schema and object name is created.

Example

Create two tables, target and sample, and a trigger called test.

```
CREATE TABLE target ( A INT);
 CREATE TABLE sample ( A INT);
 CREATE TRIGGER test
```

```
  AFTER UPDATE ON target
  BEGIN
    INSERT INTO sample VALUES(3);
  END;
```

Drop the trigger called test.

```
DROP TRIGGER test;
```

7.7.29 DROP TYPE Statement (Procedural)

Removes a user-defined table type.

Syntax

```
DROP TYPE <type_name> [<drop_option>]
```

Syntax Elements

type_name

Specifies the table type to be dropped with optional schema name.

```
<type_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

drop_option

Specifies how the type is dropped.

```
<drop_option> ::= CASCADE | RESTRICT
```

When `<drop_option>` is not specified, a non-cascaded drop be performed. Only the specified type is dropped; dependent objects of the type are invalidated, but they are not dropped.

The invalidated objects can be revalidated when an object that has same schema and object name is created.

Description

The DROP TYPE statement removes a user-defined table type.

Example

Create a table type called my_type.

```
CREATE TYPE "my_type" AS TABLE ( "column_a" DOUBLE );
```

Drop the my_type table type.

```
DROP TYPE "my_type";
```

7.7.30 DROP USER Statement (Access Control)

Deletes a database user.

Syntax

```
DROP USER <user_name> [<drop_option>]
```

Syntax Elements

user_name

Drops the specified user:

```
<user_name> ::= <unicode_name>
```

`<user_name>` must specify an existing database user.

drop_option

Specifies the drop option to use:

```
<drop_option> ::= CASCADE | RESTRICT
```

When you do not specify `<drop_option>`, RESTRICT is used by default.

CASCADE drops the user and performs the following actions:

- Drops the user's home schema and other schemas belonging to the user together with all objects stored in them regardless of which user created them.
- Drops objects owned by the user, even if they are part of another schema.
- Drops objects that are dependent on deleted objects.
- Revokes privileges on deleted objects.
- Revokes privileges granted by the deleted user.

Revoked privileges may cause further revokes if the privileges have been granted further.

RESTRICT drops the user only if they are not the owner of any other object than their home schema and other schemas created by the user. RESTRICT prevents the user from being dropped if schemas owned by the user contain objects owned by other users.

Description

The database administrator or a user with USER ADMIN system privilege can drop a user.

Users created by the deleted user and roles created by them are not deleted.

It is possible to delete a user even if an open session for this user exists.

Examples

Example 1 – Creates a new user called new_user:

```
CREATE USER new_user PASSWORD Password1;
```

Example 2 – Drops the user new_user you created in the previous step:

```
DROP USER new_user CASCADE;
```

7.7.31 DROP VIEW Statement (Data Definition)

Removes a view from the database.

Syntax

```
DROP VIEW <view_name> [<drop_option>]
```

Syntax Elements

view_name

Drops the specified view, with the optional schema name.

```
<view_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

drop_option

Specifies the drop option to use.

```
<drop_option> ::= CASCADE | RESTRICT
```

CASCADE drops the view and dependent objects. RESTRICT drops the view only when dependent objects do not exist. If this drop option is used and a dependent object exists, then an error is thrown.

If `<drop_option>` is not specified, then a non-cascaded drop is performed which only drops the specified view. Dependent objects of the view are invalidated but not dropped.

The invalidated objects can be re-validated when an object that has same schema and object name is created.

Description

The DROP VIEW statement removes a view from the database.

Example

Create table t, and then a view, v, that selects all records from table t.

```
CREATE TABLE t (a INT);
 CREATE VIEW v AS SELECT * FROM t;
```

Drop view v from the database.

```
DROP VIEW v;
```

7.7.32 GRANT Statement (Access Control)

Grants privileges to users and roles.

i Note

SAP ASE restrictions:

- The SOURCE and STRUCTURED privileges are not supported.

Syntax

Grant system privileges:

```
GRANT <system_privilege>[{, <system_privilege>}...] TO <grantee> [WITH ADMIN
OPTION]
```

Grant schema privileges:

```
GRANT <schema_privilege>[{, <schema_privilege>}...] ON SCHEMA <schema_name> TO
<grantee> [WITH GRANT OPTION]
```

Grant object privileges:

```
GRANT <object_privilege>[{, <object_privilege>}...] ON <object_name> TO
<grantee> [WITH GRANT OPTION]
```

Grant role to a user:

```
GRANT <role_name> to <grantee>
```

Syntax Elements

system_privilege

Grants the specified system privilege:

```
<system_privilege> ::=
| CREATE SCHEMA
| USER ADMIN
| ROLE ADMIN
```

- CREATE SCHEMA – allows the grantee to create schemas for themselves and others.
- USER ADMIN – allows the grantee to create, alter, and drop users.
- ROLE ADMIN – allows the grantee to create, drop, grant, and revoke roles.

schema_privilege

Grants the specified privilege on a schema to the grantee:

```
<schema_privilege> ::=
| SELECT
| UPDATE
| DELETE
| INSERT
| REFERENCES
| EXECUTE
| CREATE ANY
```

object_privilege

Restricts accessing and modifying database objects:

```
<object_privilege> ::=
| SELECT
| UPDATE
| DELETE
| INSERT
| REFERENCES
| EXECUTE
| ALL PRIVILEGES
```

object_name

```
<object_name> ::= <table_name>
```

```
| <view_name>
| <sequence_name>
| <procedure_name>
| <function_name>
```

The following table describes the privileges, on which the object can be granted.

Privilege	Object Types
SELECT	Tables, views, sequences
UPDATE	Tables
DELETE	Tables
INSERT	Tables
EXECUTE	Procedures, functions
REFERENCES	Tables
ALL PRIVILEGES	Tables and views

grantee

Specifies the user or role to which the privilege is being granted:

```
<grantee> :: =  PUBLIC | <user_name> | <role_name>
```

```
<user_name> ::= <unicode_name>
```

```
<role_name> ::= <identifier>
```

Where:

- PUBLIC specifies all users. For object access permissions, PUBLIC excludes the object owner. For object creation permissions PUBLIC excludes the database owner
- `<user_name>` specifies the grantee's user name.
- `<role_name>` specifies the grantee role name.

Special considerations:

- If a privilege or role is granted to a role, then all users granted that role have the specified privilege or role.
- A role is a named collection of privileges and can be granted to either a user or another role.
- To allow several database users to perform the same actions:
  1. Create a role
  2. Grant the needed privileges to this role
  3. Grant the role to the different database users
- Privileges cascade when you grant roles to other roles. That is, if you grant a role (role_R) to a role or user (use_G), this user has all the privileges directly granted to role_R and all privileges granted to roles that have been granted to role_R.

WITH ADMIN OPTION and WITH GRANT OPTION

Specifies that the granted privileges can be granted further by the specified user or by users having the specified role.

role_name

Specifies the role to which you are granting to a specific users.

Description

- GRANT grants privileges to users and roles. GRANT is also used to grant roles to users and other roles.
- The specified users, roles, and objects privileges must exist before they can be used in the GRANT command.
- A user need both the privilege and permissions required to grant that privilege to use the GRANT command to grant privileges to other users and roles.
- The SYSTEM user has all system privileges and the role PUBLIC. All other users have the role PUBLIC. These privileges and roles cannot be revoked.
- For tables created by users, users have all privileges and may grant all privileges further to other users and roles.
- The owner of the dependent object does not have a complete set of privileges for objects that are dependent on other objects, similar to views being dependent on tables. This can occur if the user do not have the privileges on the underlying objects on which their object depends.
- Users can have privileges on an object, but may not have sufficient privileges to grant them to other users and roles.

Examples

Example 1 – Creates a schema called my_schema:

```
CREATE SCHEMA my_schema;
```

Example 2 – Creates a table named work_done in the my_schema:

```
CREATE TABLE my_schema.my_schoolwork_done (t TIMESTAMP, user NVARCHAR (256),
work_done VARCHAR (256));
```

Example 3 – Creates a new user named worker with password His_Password_1:

```
CREATE USER worker PASSWORD His_Password_1;
```

Example 4 – Creates role called role_for_work_on_my_schema:

```
CREATE ROLE role_for_work_on_my_schema;
```

Example 5 – Grants the SELECT privilege on any object in my_schema to the role_for_work_on_my_schema:

```
GRANT SELECT ON SCHEMA my_schema TO role_for_work_on_my_schema;
```

Example 6 – Grants the INSERT privilege for the work_done table to the role_for_work_on_my_schema:

```
GRANT INSERT ON my_schema.my_schoolwork_done TO role_for_work_on_my_schema;
```

Example 7 – Grants DELETE privilege for this table to the worker user:

```
GRANT DELETE ON my_schema.my_schoolwork_done TO worker;
```

Example 8 – Grants the worker user the privilege to create any kind of object in the my_schema:

```
GRANT CREATE ANY ON SCHEMA my_schema TO worker;
```

The result of the above examples is that the worker user has the privilege to SELECT all tables, views and sequences in schema my_schema, to INSERT into and DELETE from table my_schoolwork_done and to create objects in schema my_schema.

Related Information

Security [page 21]

7.7.33 INSERT Statement (Data Manipulation)

Adds a record to a table.

> i Note
>
> SAP ASE restrictions:
>
> - The `<hint_clause>` is not supported.

Syntax

```
INSERT INTO <table_name> [<column_list_clause>] { <value_list_clause> <subquery>}
```

Syntax Elements

table_name

Specifies the table where the insert is to be performed, with optional schema name:

```
<table_name> ::= [ <schema_name>. ]<identifier>

<schema_name> ::= <unicode_name>
```

column_list_clause

Specifies a list of column identifiers, ordered in the order of values in the `<value_list_clause>` or `<subquery>`:

```
<column_list_clause> ::= ( <column_name>, ... )
<column_name> ::= <identifier>
```

If the column list is omitted, then the database performs the insert using all the columns in the table. A column that is not included in the column list is filled using the column's default value.

value_list_clause

Specifies a list of values, or expressions evaluating to values, that are inserted into the table:

```
<value_list_clause> ::= VALUES ( <expression>, ... )
```

subquery

Specifies the subquery.

For information on subqueries, see the SELECT statement.

Description

The values to be inserted can either be values, expressions or the result of a subquery. If the subquery used does not return any records, then the database does not insert any records into the table.

Always define the `<column_list_clause>`. This practice helps to protect your INSERT queries from damaging data if the target table schema is modified.

Example

Create table T:

```
CREATE TABLE T (KEY INT PRIMARY KEY, VAL1 INT, VAL2 NVARCHAR(20));
```

Insert a row into table T.

```
INSERT INTO T VALUES (1, 1, 'The first');
```

KEY	VAL1	VAL2
1	1	The first

Insert a new row into table T using column list to specify which columns should receive the input values.

```
INSERT INTO T (KEY, VAL2) VALUES (2,3);
```

KEY	VAL1	VAL2
1	1	The first
2	0	NULL

Insert a row into table T using a subquery.

```
INSERT INTO T SELECT 3, 3, 'The third' FROM DUMMY;
```

KEY	VAL1	VAL2
1	1	The first
2	0	3
3	3	The third

7.7.34 REFRESH STATISTICS Statement (Data Definition)

Refreshes data statistic objects that the query optimizer uses to make better decisions for query plans.

SAP ASE restrictions:

- The data statistic type RECORD is not supported

Syntax

```
REFRESH STATISTICS ON <table_name> {column_name}
```

Syntax Element

table_name

Refreshes statistics for the specified table name, with the optional schema name:

```
<table_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

column_name

Refreshes statistics for the specified column name:

```
<column_name> ::= <identifier>
```

Description

The REFRESH STATISTICS statement rebuilds data statistics objects.

Example

Refreshes the histograms on table columns T.a, T.b, and T.c:

```
REFRESH STATISTICS ON T(a,b,c) TYPE HISTOGRAM;
```

Insert, drop, and refresh statistics:

```
1> insert into test values(1,1)
2> go
(1 row affected)
1> refresh statistics on test
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    1
(1 row affected)
1> drop statistics on test
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    0
(1 row affected)
1> insert into test values(1,1)
2> go
(1 row affected)
1> refresh statistics on test(a,b)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    5
(1 row affected)
1> drop statistics on test
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    0
(1 row affected)
1> insert into test values(1,1)
2> go
(1 row affected)
1> refresh statistics on test(a,b)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    5
(1 row affected)
```

```
1> drop statistics on test(b)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    5
(1 row affected)
1> drop statistics on test(a)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    2
(1 row affected)
1> drop statistics on test(a,b)
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    1
(1 row affected)
1> drop statistics on test
2> go
1> select count(*) from sysstatistics where id=(select id from sysobjects where
name='test')
2> go
 --------------------
                    0
(1 row affected)
```

7.7.35 RENAME COLUMN Statement (Data Definition)

Changes the name of a column.

i Note

SAP ASE restrictions:

- Renaming the columns of tables that are in other schemas is not supported.

Syntax

```
RENAME COLUMN <table_name>.<old_column_name> TO <new_column_name>
```

Syntax Elements

table_name

Specifies the name of the table where the column is to be renamed:

```
<table_name> ::= <identifier>
```

old_column_name

Specifies the old column name:

```
<old_column_name> ::= <identifier>
```

new_column_name

Specifies the new column name:

```
<new_column_name> ::= <identifier>
```

Description

Changes the name of a column.

Examples

Example 1 – Creates table tab with two columns named A and B:

```
CREATE TABLE tab (A INT PRIMARY KEY, B INT);
```

Example 2 – Displays the column names for table tab stored in the SAP HANA database:

```
SELECT COLUMN_NAME, POSITION FROM TABLE_COLUMNS WHERE SCHEMA_NAME =
CURRENT_SCHEMA AND TABLE_NAME = 'tab' ORDER BY POSITION;
```

Example 3 – Renames column A to C:

```
RENAME COLUMN tab.A TO C;
```

Example 4 – Displays the column names for table tab after the renaming:

```
SELECT COLUMN_NAME, POSITION FROM TABLE_COLUMNS WHERE SCHEMA_NAME =
CURRENT_SCHEMA AND TABLE_NAME = 'tab' ORDER BY POSITION;
```

7.7.36 RENAME TABLE Statement (Data Definition)

Changes the name of a table.

> i Note
>
> SAP ASE restrictions:

- The `<current_table_name>` cannot contain a schema name, because renaming a table to a different schema is not allowed. For example, you cannot use `RENAME TABLE schema1.tab1 to tab2`.

Syntax

```
RENAME TABLE <current_table_name> TO <new_table_name>
```

Syntax Elements

current_table_name

Specifies the current name of the table, with optional schema name.

```
<current_table_name> ::= [<schema_name>.]<identifier>
```

new_table_name

Specifies the new name for the table.

```
<new_table_name> ::= [<schema_name>.]<identifier>
```

Description

The RENAME TABLE statement changes the name of a table.

Example

Example 1 - Renaming a table in the current schema

Create table A in the current schema:

```
CREATE TABLE A (A INT PRIMARY KEY, B INT);
```

Show a list of table names in the current schema:

```
select name
    from dbo.sysobjects
    where schemaid=(select id from dbo.sysobjects
        where name=CURRENT_SCHEMA)
```

Rename table A to a new name B.

```
RENAME TABLE A TO B;
```

Example 2 - Renaming a table in another schema

Create schema mySchema and then create a table A in the new schema:

```
CREATE SCHEMA mySchema;
 CREATE TABLE mySchema.A (A INT PRIMARY KEY, B INT);
```

Show the list of table names in schema mySchema:

```
select name
    from dbo.sysobjects
    where schemaid=(select id from dbo.sysobjects
        where name=mySchema)
```

Rename table A in mySchema to B:

```
RENAME TABLE mySchema.A TO B;
```

Show a list of table names in schema mySchema after the renaming process:

```
select name
    from dbo.sysobjects
    where schemaid=(select id from dbo.sysobjects
        where name=mySchema)
```

Example 3 - Renaming a table schema

Rename the schema name of table B from mySchema to mySchema2:

```
RENAME TABLE mySchema.B TO mySchema2.B;
```

Show the list of table names in schema mySchema2.B:

```
select name
    from dbo.sysobjects
    where schemaid=(select id from dbo.sysobjects
        where name=mySchema2)
```

7.7.37 REPLACE / UPSERT Statement (Data Manipulation)

Updates rows in a table or inserts new rows.

i Note

SAP ASE restrictions:

- The WHERE clause is not supported.
- The WITH PRIMARY KEY option is not supported.

Syntax

```
UPSERT  <table_name> [ <column_list_clause> ]
   {
     <value_list_clause>
     | <subquery>
   }
REPLACE <table_name> [ <column_list_clause> ]
   {
     <value_list_clause>
     | <subquery>
   }
```

Syntax Elements

table_name

Specifies the table where UPSERT or REPLACE is performed, with the optional schema name:

```
<table_name> ::= [ <schema_name>. ]<identifier>
<schema_name> ::= <unicode_name>
```

column_list_clause

Specifies a list of column identifiers, ordered in the order of values in the `<value_list_clause>` or `<subquery>`:

```
<column_list_clause> ::= ( <column_name>, ... )
<column_name> ::= <identifier>
```

value_list_clause

Specifies a list of values, or expressions evaluating to values, to be inserted or updated into the table:

```
<value_list_clause> ::= VALUES ( <expression>, ... )
```

subquery

For information on subqueries, see the SELECT statement.

Description

When this command is used without a subquery it functions in a similar way to the INSERT or UPDATE statements.

Executing the UPSERT (or REPLACE) statement with a subquery functions is to INSERT or UPDATE with an array value construction by subquery.

Example

This example shows creating a table, and then using UPSERT to insert a new value if the conditions (a=1 OR b=2 OR c=3) are false:

```
1> create table t1 (a int,b int,c int)
2> go
1> upsert t1(a,b,c) values(1,2,3)
2> go
(1 row affected)
1> select * from t1
2> go
a           b           c
----------- ----------- -----------
      1          2          3
(1 row affected)
```

This example shows creating a table, and then using UPSERT to update with a new value if the conditions (a=1 OR b=1 OR c=1)) are true:

```
1> create table t1 (a int,b int,c int)
2> go
1> insert into t1(a,b,c) values(1,2,3)
2> go
1> upsert t1(a,b,c) values(1,1,1)
2> go
(1 row affected)
1> select * from t1
2> go
a           b           c
----------- ----------- -----------
    1           1           1
(1 row affected)
```

This example shows creating a source and target table, inserting data into source table, then inserting the data from the source into the target using an array value construction with a subquery:

```
create table source (a int,b int,c int)
create table target (a int,b int,c int)

insert into source values(1,1,1)
insert into source values(2,2,2)
insert into source values(3,3,3)
1> select * from source
2> go
a           b           c
----------- ----------- -----------
     1           1           1
     2           2           2
     3           3           3
(3 rows affected)
1> select * from target
2> go
a           b           c
----------- ----------- -----------
(0 rows affected)
1> upsert target (a,b,c) select a,b,c from source
2> go
(3 rows affected)
1> select * from target
2> go
a           b           c
----------- ----------- -----------
     1           1           1
```

```
      2              2              2
      3              3              3
(3 rows affected)
```

7.7.38 REVOKE Statement (Access Control)

Revokes permissions from users, roles, or groups.

i Note

SAP ASE restrictions:

These system privileges are not supported:

- ON REMOTE SOURCE FROM
- STRUCTURED PRIVILEGE FROM
- ADAPTER ADMIN

Syntax

Revoke system privileges:

```
REVOKE <system_privilege>,... FROM <grantee>
```

Revoke schema privileges:

```
REVOKE <schema_privilege>,... ON SCHEMA <schema_name> FROM <grantee>
```

Revoke object privileges:

```
REVOKE <object_privilege>,... ON <object_name> FROM <grantee>
```

Revoke roles:

```
REVOKE <role_name>,... FROM <grantee>
```

Syntax Elements

system_privilege

Revokes the specified system privilege:

```
<system_privilege> ::=
| CREATE SCHEMA
| USER ADMIN
| ROLE ADMIN
```

schema_privilege

Revokes the specified privilege on a schema from the grantee:

```
<schema_privilege> ::=
| SELECT
| UPDATE
| DELETE
| INSERT
| REFERENCES
| EXECUTE
| CREATE ANY
```

object_privilege

Restricts accessing and modifying database objects:

```
<object_privilege> ::=
| SELECT
| UPDATE
| DELETE
| INSERT
| REFERENCES
| EXECUTE
| ALL PRIVILEGES
```

object_name

```
<object_name> ::= <table_name>
| <view_name>
| <sequence_name>
| <procedure_name>
| <function name>
```

```
<table_name> ::= [<schema_name>.]<identifier>
```

```
<view_name> ::= [<schema_name>.]<identifier>
```

```
<procedure_name> ::= [<schema_name>.]<identifier>
```

```
<sequence_name> ::= [<schema_name>.]<identifier>
```

```
<function_name> ::= [<schema_name>.]<identifier>
```

Where:

- `<table_name>` - is the name of the table from which you are revoking permissions
- `<view_name>` - is the name of the view from which you are revoking permissions
- `<procedure_name>` - is the name of the stored procedure from which you are revoking permissions
- `<sequence_name>` - is the name of an sequence from which you are revoking permissions
- `<function_name>` - is the name of an function from which you are revoking permissions

FROM {PUBLIC | NAME_LIST | ROLE_LIST} Indicates that you are revoking permissions from the specified object.

PUBLIC Is all users. For object access permissions, PUBLIC excludes the object owner. For object creation permissions, public excludes the database owner.

NAME_LIST A list of user names, separated by commas.

ROLE_LIST A list of the name of system or user-defined roles from whom you are revoking the permission, and cannot be a variable..

Description

Either a user with ROLE ADMIN privilege or a user who has been granted the privilege with ADMIN OPTION can revoke the privilege.

By default, the SYSTEM user has all the system privileges and the PUBLIC role, and all other users have the PUBLIC role. These privileges and roles cannot be revoked.

If a user has been granted a role, then it is not possible to revoke a subset of the privileges belonging to that role. To grant a subset of privileges contained within a role, revoke the entire role and grant the required privileges separately instead.

Revoking a privilege or role can lead to some views becoming inaccessible or procedures that the user can no longer execute. This occurs if a view or procedures depends on a revoked privilege or on one of the privileges the role had.

Example

Revokes SELECT permissions from my_schema:

```
REVOKE SELECT ON SCHEMA my_schema FROM role_for_work_on_my_schema;
```

Related Information

Security [page 21]

7.7.39 ROLLBACK Statement (Transaction Management)

Changes made during the current transaction are reverted and the current database session is set to an idle state.

Syntax

```
ROLLBACK
```

Description

The database supports transactional consistency, which guarantees that a transaction is completely applied to the system or disposed. During a transaction, data manipulation language (DML) modifications to the database can be explicitly reverted via ROLLBACK command. After ROLLBACK is issued, changes made during the current transaction are reverted, and the current database session is set to an idle state. ROLLBACK only works with an autocommit-disabled session.

If you attempt to use the ROLLBACK statements in an autocommit-enabled session, then nothing occurs, as transactions are automatically committed to the database.

Example

Before attempting to execute this example, ensure that you are using an autocommit-disabled session.

Create a table T:

```
CREATE TABLE T (KEY INT PRIMARY KEY, VAL INT);
```

Insert three rows into table T:

```
INSERT INTO T VALUES (1, 1);
 INSERT INTO T VALUES (2, 2);
 INSERT INTO T VALUES (3, 3);
```

Roll back the current transaction.

```
ROLLBACK;
```

Select the data in table T:

```
SELECT * FROM T;
```

The SELECT command returns an empty table. The data definition language (DDL) command used to create the table persisted, but the DML used to create the table data has been rolled back.

7.7.40 SELECT Statement (Data Manipulation)

Retrieves rows from database objects.

i Note

SAP ASE restrictions:

- These clauses are not supported:
    - `<with_clause>`
    - `time_travel`
    - `<hint_clause>`

    ◦ TABLESAMPLE clause in <subquery>
- The following grouping sets for <group_by_clause> are not supported: GROUPING SETS, ROLLUP, CUBE, BEST 1, WITH SUBTOTAL, WITH BALANCE, WITH TOTAL, TEXT_FILTER , FILL UP [SORT MATCHES TO TOP], and STRUCTURED RESULT [WITH OVERVIEW].
- Multiple result sets are not supported.
- Set operators UNION DISTINCT, INTERSECT DISTINCT, and EXCEPT DISTINCT are not supported
- <order_by_clause> is only supported in main query and IN and EXISTS predicates.
- <limit> cannot be used with TOP at the same time. Only one of them can be used in a single query.
- <set_operators> cannot be used with the subquery. They can only be used with the main query.

Syntax

```
<select_statement> ::=
 <subquery> [ <for_update> ]
 |   ( <subquery> ) [ <for_update> ]

<subquery> ::= <select_clause> <from_clause> [<where_clause>]
 [<group_by_clause>]
 [<having_clause>]
 [<set_operator> <subquery> [{, <set_operator> <subquery>}...]]
 [<order_by_clause>]
 [<limit>]
```

Syntax Elements

for_update

Locks the selected records so that other users cannot lock or change the records until end of this transaction:

```
<for_update> ::= FOR UPDATE [OF <update_column_name_list>][ <wait_nowait> ]
<wait_nowait> ::= WAIT <unsigned_integer> | NOWAIT
<update_column_name_list>> ::= ( <column_name>[{, <column_name>}...] )
```

FOR UPDATE supports row, column, and virtual tables created from SAP HANA remote sources.

<wait_nowait> specifies when to return an error if a lock cannot be obtained on a record. If WAIT <unsigned_integer> is specified, then the statement waits up to the specified number of seconds for each record. If the statement fails to acquire a lock after waiting, then it returns an error message indicating a timeout. If NOWAIT is specified, and the statement fails to acquire record locks, then an error is returned indicating that the resource is busy. WAIT 0 is equivalent to NOWAIT. If you do not specify <wait_nowait>, then the default behavior reflects the lock_wait_timeout transaction timeout setting located in indexserver.ini.

If a SAP HANA query using virtual tables cannot be entirely delegated to the remote source, the query returns the error message indicating that the FOR UPDATE feature is not supported.

A list of columns specifies which table or view in the FROM clause should be locked. If there is only a single table or view, then FOR UPDATE is sufficient. However, if there are two objects (such as tables, views, of subqueries), use OF.

A table used in a subquery cannot be locked. Also only one table/view can be currently locked. The lock is released when the corresponding transaction is finished by commit or rollback.

select_clause

Specifies an output to be returned either to the caller or to an outer `<select_clause>` if one exists:

```
<select_clause> ::=
 SELECT [TOP <unsigned_integer>] [ ALL | DISTINCT ] <select_list>
```

TOP

Specifies that the first `<unsigned_integer>` records from the SQL statement should be returned:

```
TOP <unsigned_integer>
```

DISTINCT

Specifies that only one copy of each set of duplicate records selected should be returned.

ALL

(Default) Specifies that all rows selected, including all copies of duplicates, should be returned.

select_list

Retrieves the specified list of columns:

```
<select_list> ::= <select_item>[{, <select_item>}... ]
<select_item> ::=
 [<table_name>.]{<asterisk> | <expression>} [ AS <column_alias> ]
<table_name> ::= [[<database_name>.]<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
<column_alias> ::= <identifier>
<asterisk> ::= *
```

`<select_item>` specifies the details of the columns to be retrieved.

`<table_name>` specifies the table that is the source for the selected columns, with the optional schema name.

`<column_alias>` specifies an alias for a column.

`<asterisk>` (*) selects all columns from all tables or views listed in the `<from_clause>`. If you provide a schema name or a table name with an asterisk (*), then it limits the scope of the result set to the specified table.

from_clause

Specifies inputs such as tables, views, and subqueries to be used in the SELECT statement:

```
<from_clause> ::= FROM <table> [{, <table>}...]
<table> ::=
 <table_name> [ <partition_restriction> ] [ [AS] <table_alias> ]
[<tablesample_clause>]
 | <subquery> [ [AS] <table_alias> ]
 | <joined_table>
 | <collection_derived_table>
 | <associated_table_expression>
```

<table> specifies the table, subquery or join which is the data source for the query.

partition_restriction

Restricts the query to the specified partition(s) of a partitioned table:

```
<> ::=
 PARTITION ( <partition_number> [{, <partition_number>}...] )
<partition_number> ::= <unsigned_integer>
```

If no partition is specified, then all partitions are the source of the SELECT.
<partition_restriction> can be obtained from the M_CS_PARTITIONS system view. This syntax is available for partitioned tables only.

table_alias

Specifies the alias identifier for a table, subquery, or joined table:

```
<table_alias> ::= <identifier>
```

tablesample_clause

Applies the SELECT statement to a random sample of the table data:

```
<tablesample_clause> ::=
 TABLESAMPLE [BERNOULLI | SYSTEM] (<sample_size>)
<sample_size> ::= <exact_numeric_literal>
```

<sample_size> specifies the percentage of the table to be returned as a sample.

Values must be greater than 0 and less than or equal to 100. (100 returns the complete table). The goal of the TABLESAMPLE operator is to allow queries to be executed over ad-hoc random samples of tables. Samples are computed uniformly over rows in a columnar base table. Samples can either be uniform random samples (SYSTEM sampling) or uniform and independent random samples (BERNOULLI sampling).

SYSTEM sampling exploits the lack of the independence requirement by sampling blocks of rows at a time, while SYSTEM sampling offers performance advantages over BERNOULLI sampling, it comes at the cost of larger variance in sample size.

joined_table

Specifies how tables are joined:

```
<joined_table> ::=
 <table> [<join_type>] JOIN <table> ON <predicate>
 | <table> CROSS JOIN <table>
 | <joined_table>
```

ON <predicate> specifies a join predicate.

CROSS JOIN specifies that a cross join should be performed. A cross join produces the cross-product of two tables:

```
<join_type> ::=
 INNER  | { LEFT | RIGHT | FULL } [OUTER]
```

<join_type> specifies the type of join to be performed:

- INNER performs an inner join.

- LEFT defines a left outer join.
- RIGHT defines a right outer join.
- FULL indicates a full outer join.
- OUTER performs an outer join.

collection_derived_table

Specifies a collection-derived table.

```
<collection_derived_table> ::=
 UNNEST ( <collection_value_expression> [{,
<collection_value_expression>}...] )
   [WITH ORDINALITY] AS <table_alias> [ <derived_column_list> ]
<collection_value_expression> ::=
 ARRAY ( <expression> [{, <expression>}...] | <column_name> )
<derived_column_list> ::= ( <column_name>, ... )
```

Collection-derived tables are similar to lateral tables in that they can contain a reference to another element of the `<from_clause>` that has been specified before them. This reference has inner join characteristics, because the referenced element of the `<from_clause>` is used as a parameter to execute the lateral subquery.

Collection values (array) can be interpreted as the columns of a table by "turning them 90 degrees clockwise".

You can use an array constructor or a field name of ARRAY type.

`<derived_column_list>` specifies a list of column identifiers, sorted in the order of values in the collection value expressions. The column list can be omitted.

associated_table_expression

Specifies a table that is associated with other tables:

```
<associated_table_expression> ::=
 <associated_table_name> : <associated_column>[.<associated_column2>[...]]
```

`<associated_column>` specifies a column that is associated with `<associated_table_name>`.

where_clause

Specifies predicates on inputs in the FROM clause:

```
<where_clause> ::= WHERE <condition>
<condition> ::=
 <condition> OR <condition>
 | <condition> AND <condition>
 | NOT <condition>
 | ( <condition> )
 | <predicate>

<predicate> ::=
 <comparison_predicate>
 | <range_predicate>
 | <in_predicate>
 | <exist_predicate>
 | <like_predicate>
 | <null_predicate>

<comparison_predicate> ::=
 <expression> { = | != | <> | > | < | >= | <= } [ ANY | SOME | ALL ]
({<expression_list>
 | <subquery>})
```

```
<range_predicate> ::=
 <expression> [NOT] BETWEEN <expression> AND <expression>

<in_predicate> ::=
 <expression> [NOT] IN ( { <expression_list> | <subquery> } )

<exist_predicate> ::=
 [NOT] EXISTS ( <subquery> )

<like_predicate> ::=
 <expression> [NOT] LIKE <expression> [ESCAPE <expression>]

<null_predicate> ::=
 <expression> IS [NOT] NULL
<expression_list> ::= <expression> [{, <expression>}...]
```

GROUP BY

Groups the selected rows based on the values in the specified columns:

```
<group_by_clause> ::= GROUP BY <group_by_expression_list>
<group_by_expression_list> ::=
 { <expression>|<grouping_set> } [{, <expression>|<grouping_set>} ...]
<grouping_expression_list> ::= <grouping_expression> [{,
<grouping_expression>}...]
<grouping_expression> ::=
 <expression>
 | ( <expression> [{, <expression>}...] )
 | ( ( <expression> [{, <expression>}...] ) <order_by_clause> )
```

LIMIT

Returns the first `<unsigned_integer>` grouped records after skipping OFFSET`<unsigned_integer>` for each grouping set:

```
LIMIT <unsigned_integer> [OFFSET <unsigned_integer>]
```

FILL UP

Returns not only matched grouped records, but also non-matched records.

STRUCTURED RESULT

Returns results as temporary tables. For each grouping set a single temporary table is created.

PREFIX

Specifies a prefix for naming the temporary tables:

```
PREFIX <prefix_table_name>
<prefix_table_name> ::= #<identifier>
```

`<prefix_table_name>` must start with "#", which means a temporary table. If you do not specify PREFIX, the default prefix is "#GN" followed by a non-negative integer number. See 'Return Format' below.

Related Functions:

- GROUPING_ID(`<column_name_list>`) function returns an integer number to identify which grouping set each grouped record belongs to.
- GROUPING(`<column>`) function returns 1 if the column is used for grouping. Otherwise it returns 0.

- TEXT_FILTER (<grouping_column>) function, which is used with TEXT_FILTER, FILL UP, and SORT MATCHES TO TOP, displays matching values or NULL. NULL is displayed for non-matching values when FILL UP option is specified.

HAVING

Selects the specified groups that satisfy the predicates.

```
<having_clause> ::= HAVING <condition>
```

If this clause is omitted, all groups are selected.

SET OPERATORS

Enables multiple SELECT statements to be combined but return only one result set.

```
<set_operator> ::=
 UNION [ ALL | DISTINCT ]
 | INTERSECT [DISTINCT]
 | EXCEPT [DISTINCT]
```

UNION ALL

Selects all records from all SELECT statements. Duplicates are not removed.

ORDER BY

Sorts records by expressions or positions:

```
<order_by_clause> ::=
 ORDER BY { <order_by_expression>, ... }
<order_by_expression> ::=
 <expression> [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
 | <position> [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]
<position> ::= <unsigned_integer>
```

ASC sorts records in ascending order and DESC sorts records in descending order. The default is ASC.

`[ NULLS FIRST | NULLS LAST ]` specifies where in the result set NULL values should appear. For ascending ordering NULL values are returned first by default. For descending ordering, NULL values are returned last. You can override this behavior using NULLS FIRST or NULLS LAST to explicitly specify NULL value ordering.

`<position>` uses the entries in the select list to define the ordering required. For example:

```
SELECT col1, col2 FROM t ORDER BY 2;
```

ORDER BY 2 indicates that ordering should be undertaken using the second expression in the select list, which in this case is col2.

LIMIT

Limits the number of output records:

```
<limit> ::=
 LIMIT <unsigned_integer> [ OFFSET <unsigned_integer> ]
```

The following example returns the first 3 records after skipping 5 records:

```
LIMIT 3 [OFFSET 5]
```

Examples

Example 1: WAIT and NOWAIT

Creates and then selects from table x:

```
CREATE column table x ( a int, b int );
INSERT INTO x values (1,1);
INSERT INTO x values (2,2);
CREATE column table y ( a int, b int );
INSERT INTO y values (1,1);
INSERT INTO y values (2,2);
SELECT * FROM x WHERE a=1 FOR UPDATE WAIT 1; --> OK
SELECT * FROM y WHERE b=1 FOR UPDATE NOWAIT; --> OK
COMMIT;
SELECT * FROM x WHERE a=1 FOR UPDATE OF x.c; --> error because c column does not
exist in table x.
SELECT * FROM x WHERE a=1 FOR UPDATE OF x.a; --> OK
COMMIT;
SELECT * FROM x, y WHERE a=1 FOR UPDATE; --> error because "OF" is not used to
specify which one of x and y is locked.
SELECT * FROM x, y WHERE a=1 FOR UPDATE OF y.b; --> OK
COMMIT;
SELECT * FROM x, (SELECT * FROM y) WHERE a=1 FOR UPDATE OF y.b;  --> error
because y is inside subquery
COMMIT;
```

Example 2: GROUP BY

Creates column table t1 and populates it with some example data:

```
create column table t1 ( id int primary key, customer varchar(5), year int,
product varchar(5), sales int );
 insert into t1 values(1, 'C1', 2009, 'P1', 100);
 insert into t1 values(2, 'C1', 2009, 'P2', 200);
 insert into t1 values(3, 'C1', 2010, 'P1', 50);
 insert into t1 values(4, 'C1', 2010, 'P2', 150);
 insert into t1 values(5, 'C2', 2009, 'P1', 200);
 insert into t1 values(6, 'C2', 2009, 'P2', 300);
 insert into t1 values(7, 'C2', 2010, 'P1', 100);
 insert into t1 values(8, 'C2', 2010, 'P2', 150);
```

Example 3: LIMIT

Limits the number of records to a maximum 2 for each group:

```
select customer, year, product, sum(sales)
     from t1
     group by grouping sets LIMIT 2
     (
      (customer, year),
      (product)
     );
```

For the (customer, year) group, the number of records are 4, so only first 2 records are returned. For the (product) group, the number of records are 2, in this case all the records are returned.

Example 4: FILL UP

Uses FILL UP to return both matched and non-matched records with <filterspec>. The following query returns six records, whereas the previous example only returned three:

```
select customer, year, product, sum(sales), text_filter(customer),
text_filter(product)
     from t1
     group by grouping sets TEXT_FILTER '*2' FILL UP
     (
      (customer, year),
      (product)
     );
```

Example 5: TABLESAMPLE

Selects a random sample of 1% of the employee table and within that sample count the number of managers:

```
SELECT count(*), avg(salary)
    FROM employee TABLESAMPLE SYSTEM (1)
    WHERE employee.type = ‘manager’;
```

Example 6: PARTITION

Selects the data contained in partitions 1, 3 and 4 of table `T0`:

```
SELECT * FROM T0 PARTITION (1, 3, 4);
```

Example 7: Derived table expression

When there are built-in, aggregate, UDF functions, or subqueries without an alias name present in the derived table expression, use:

```
select * from t join (select a, max(a), sum(b) from t) s on t.a = s.a
```

In following, `max(a)` and `sum(b)` are present in a derived table expression select list without alias names. Column `max(a)` is titled as `UNKNOWN.Col_1` and column `sum(b)` is titled as `UNKNOWN.Col_2`. For example:

```
select * from t join (select a, max(a), sum(b) from t) s on t.a = s.a
a       b     a    UNKNOWN.Col_1 UNKNOWN.Col_2
---- ----- ------- ------------- -------------
1      2      1      10            22
10     20     10     10            22
(2 rows affected)
```

7.7.41 SET Statement (Procedural)

The set command is used in the SAP ASE SQLScript database by prefixing the option name with `tsql_`.

Syntax

```
SET <option_name>
```

Syntax Elements

option_name

Specifies the name of the SAP ASE `set` command.

Description

The SET command is not used directly in SQLScript. The option is prefixed with `tsql`.

Example

Enables showplan:

```
set tsql showplan on;
```

7.7.42 SET SCHEMA Statement (Session Management)

Changes the current schema for the session.

Syntax

```
SET SCHEMA <schema_name>
```

Syntax Elements

schema_name

Specifies the name of the schema that the session should change to.

```
<schema_name> ::= <unicode_name>
```

Description

The current schema is used when SQL statements use database object names, such as table names, that are not prefixed with a schema name.

Example

Create a new schema called MY_SCHEMA.

```
CREATE SCHEMA MY_SCHEMA;
```

Change the current schema of the session to MY_SCHEMA.

```
SET SCHEMA MY_SCHEMA;
```

7.7.43 SET [SESSION] Statement (Session Management)

Sets variables for your database session.

Syntax

```
SET [SESSION] <key> = <value>
```

Syntax Elements

key

Specifies the key of a session variable:

```
<key> ::= <string_literal>
```

The maximum length of the key is 5000 characters.

value

Specifies the value for a session variable:

```
<value> ::= <string_literal>
```

The maximum length of the value is 5000 characters.

Description

The maximum number of session variables is 1024 per session by default. A different number of allowed session variables can be configured in `max_session_variables` in the `session` section of `indexserver.ini`.

Session variables can be retrieved by using the SESSION_CONTEXT function. They can be unset by using the UNSET [SESSION] statement.

Examples

Set the session variable MY_VAR to abc:

```
SET 'MY_VAR' = 'abc';
```

Select the variable MY_VAR from the current session:

```
SELECT SESSION_CONTEXT('MY_VAR') FROM DUMMY;
```

or:

```
SELECT * FROM SYS.M_SESSION_CONTEXT WHERE CONNECTION_ID = CURRENT_CONNECTION;
```

Unset the session variable MY_VAR:

```
UNSET 'MY_VAR';
```

7.7.44 SET TRANSACTION Statement (Transaction Management)

Implements updates by inserting new versions of data and not by overwriting existing records.

Syntax

```
SET TRANSACTION  <isolation_level>
```

Syntax Elements

isolation_level

Sets the statement level read consistency of the data in the database:

```
<isolation_level> ::= ISOLATION LEVEL <level>
<level> ::=
 READ COMMITTED
 | REPEATABLE READ
 | SERIALIZABLE
```

If `<isolation_level>` is omitted, then the default is READ COMMITTED.

The READ COMMITTED isolation level provides statement-level read consistency during a transaction. Each statement in a transaction uses the committed state of the data in the database as the execution of the statement begins. This means that each statement in the same transaction may see varying snapshots of the data in the database as they are executed. This is because data can be committed during the transaction.

The REPEATABLE READ or SERIALIZABLE isolation levels provide transaction level snapshot isolation. All statements of a transaction use the same snapshot of the database data. This snapshot contains all changes that were committed at the time the transaction started along with the changes made by the transaction itself.

Description

The database uses multi-version concurrency control (MVCC) to ensure consistent read operations. Concurrent read operations have a consistent view of the database data without blocking concurrent write operations. Updates are implemented by inserting new versions of data and not by overwriting existing records.

The isolation level specified determines the lock operation type that is used. The system supports both statement level snapshot isolation and transaction level snapshot isolation.

- For statement snapshot isolation use level READ COMMITTED.
- For transaction snapshot isolation use REPEATABLE READ or SERIALIZABLE.

During a transaction when rows are inserted, updated, or deleted, the system sets exclusive locks on the affected rows for the duration of the transaction. The system also sets shared locks on the affected tables for the duration of the transaction. This guarantees that the table is not dropped or altered while rows of the table are being updated. The database releases these locks at the end of the transaction.

Reading a row does not set any locks on either tables or rows within the database regardless of the isolation level used.

Example

Sets the transaction isolation level to READ COMMITTED to provide statement level read consistency during the current transaction:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

7.7.45 TRUNCATE TABLE Statement (Data Manipulation)

Deletes all rows from a table.

Syntax

```
TRUNCATE TABLE <table_name>
```

Syntax Elements

table_name

Truncates the specified table, with the optional schema name:

```
<table_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

Description

Deletes all rows from a table. TRUNCATE is faster than DELETE FROM when deleting all records from a table, but TRUNCATE cannot be rolled back. .

HISTORY tables can also be truncated by using this command. All parts of the history table (main, delta, history main, and history delta) are truncated, and the content is permanently deleted.

TRUNCATE is also supported in DDL auto commit mode OFF, except for local and global temporary column tables. A TRUNCATE on temporary column tables cannot be rolled back, but it does not commit changes on other database objects.

Example

Create table A:

```
CREATE TABLE A (A INT PRIMARY KEY, B INT);
```

Truncate the contents of table A:

```
TRUNCATE TABLE A;
```

7.7.46 UNSET [SESSION] Statement (Session Management)

Unsets session variables of the current session.

Syntax

```
UNSET [SESSION] <key>
```

Syntax Elements

key

Specifies the key of a session variable:

```
<key> ::= <string_literal>
```

The maximum length of the key is 32 characters.

Description

Using UNSET [SESSION], you can unset session variables for the current session.

Example

Set the session variable MY_VAR to abc:

```
SET 'MY_VAR' = 'abc';
```

Select the variable MY_VAR from the current session:

```
SELECT SESSION_CONTEXT('MY_VAR') FROM DUMMY;
```

Unset the session variable MY_VAR:

```
UNSET 'MY_VAR';
```

7.7.47 UPDATE Statement (Data Manipulation)

Changes the values of the records of a table.

> **i Note**
>
> SAP ASE restrictions:
>
> - The `<hint_clause>` and `<top_clause>` clauses are not supported.
> - The array column type is not supported.

Syntax

```
UPDATE  <table_name> [<alias_name>] [<partition_restriction>] <set_clause>
   [<from_clause>]
   [ WHERE <condition> ]
```

Syntax Elements

table_name

Specifies the table where the UPDATE is performed, with the optional schema name.

```
<table_name> ::= [<schema_name>.]<identifier>
<schema_name> ::= <unicode_name>
```

alias_name

Specifies the alias that can be used to refer to the table defined by `<table_name>`.

```
<alias_name> ::= [AS] <identifier>
```

partition_restriction

For a partitioned table, `<partition_restriction>` specifies the partition in which the target updated rows are located.

```
<partition_restriction> ::= PARTITION ( <partition_number> )
<partition_number> ::= <unsigned_integer>
```

If you do not specify a partition restriction, then the database checks all partitions for the update. But if there is partition restriction, only the rows in specified partition are updated.

Limitations:

- An UPDATE statement with a partition restriction is blocked if the following are true:
    - Partitioning key column update
    - UNIQUE constraint column update

    - PRIMARY KEY column update
- Updates that change the updated rows partition are not allowed.

set_clause

Specifies the column names and associated values to be set by the UPDATE statement:

```
<set_clause> ::=
 SET {<column_name> = <expression>
 | ( <with_clause> <subquery> ) },...
```

For the definitions of the `<with_clause>` and `<subquery>`, see the SELECT statement. The `<with_clause>` can be used with column names, not with column lists.

from_clause

Specifies inputs such as tables to be used in the UPDATE statement:

```
<from_clause> ::= FROM {<table_name>} [, {<table_name>}]...
<table_name> ::= <identifier>
```

The object to be updated can be specified as an alias in the `<from_clause>`.

WHERE condition

Specifies the conditions where the command should be performed.

```
<condition> ::=
 <condition> OR <condition>
 | <condition> AND <condition>
 | NOT <condition>
 | ( <condition> )
 | <predicate>
```

For more information on predicates, see the Predicates topic in this guide

Description

When this command is used with the FROM clause, the UPDATE FROM query must include the updated table name in the FROM clause. Or if an alias exists, the alias should exist in the FROM clause. Otherwise it throws an 'invalid table name' error.

If the WHERE condition is used and is true for a specific row, then the an update is performed on that row. If the WHERE clause is omitted, then the UPDATE command updates all records of a table.

Examples

Creates table T, and insert two rows into it:

```
CREATE TABLE T (KEY INT PRIMARY KEY, VAL INT);
 INSERT INTO T VALUES (1, 1);
 INSERT INTO T VALUES (2, 2);
```

Updates the rows of table T if the condition in the WHERE clause is true:

```
UPDATE T SET VAL = VAL + 1 WHERE KEY = 1;
```

KEY	VAL
1	2
2	2

Updates all rows of table T because a where clause is not specified as part of the update statement:

```
UPDATE T SET VAL = KEY + 10;
```

KEY	VAL
1	11
2	12

Creates table T2, and insert two rows into it:

```
CREATE TABLE T2 (KEY INT PRIMARY KEY, VAR INT);
 INSERT INTO T2 VALUES (1, 2);
 INSERT INTO T2 VALUES (3, 6);
```

Updates the values of table T by joining the target table T with table T2:

```
UPDATE T SET VAL = T2.VAR FROM T, T2 WHERE T.KEY = T2.KEY;
```

KEY	VAL
1	2
2	2

Updates the table T with an aliased table in the FROM clause by joining the target table T with table T2 which also has an alias:

```
UPDATE T A SET VAL = B.VAR FROM T A, T2 B WHERE A.KEY = B.KEY;
```

KEY	VAL
1	2
2	2

The updated table should exist in the FROM clause.

If you update table using the FROM clause like the example below, ambiguity of the target table in the FROM clause is allowed:

KEY	VAL
1	2
2	2

You can specify the partition that you want to update with partition restriction clause:

```
CREATE COLUMN TABLE PARTTAB1 (A INT, B INT, C INT) PARTITION BY RANGE(A)
(PARTITION 0 <= VALUES <3, PARTITION OTHERS);
 INSERT INTO PARTTAB1 VALUES (1,1,1);
 INSERT INTO PARTTAB1 VALUES (2,2,2);
 INSERT INTO PARTTAB1 VALUES (3,3,3);
 UPDATE PARTTAB1 PARTITION(1) SET B = 10; -- only records (1,1,1) and (2,2,2)
are updated.
```

```
UPDATE T SET VAL = B.VAL FROM T A, T B WHERE A.KEY = B.KEY; --> error due to
ambiguity of table T.
```

Update a chunk of 100,000 records of the table **`testtab`** that contains 'XXX' in the **`request`** column and the value 0 in the **`updated`** column. For each row processed, the **`updated`** column is set to 1:

```
update top 100000 testtab set updated = 1 where request = 'XXX' and updated = 0
```

8 SAP ASE Declarative SQLScript Logic

Declaritive SQLScript logic includes table parameters, local table variables, table variable type definitions, and binding table and referencing variables.

8.1 Table Parameter

Syntax

```
[IN|OUT] <param_name> {<table_type>|<table_type_definition>}
<table_type> ::= <identifier>
<table_type_definition> ::= TABLE(<column_list_elements>)
```

Description

Table parameters that are defined in the Signature are either input or output. They must be typed explicitly. This can be done either by using a table type previously defined with the `CREATE TYPE` command or by writing it directly in the signature without any previously defined table type.

Example

```
(IN inputVar TABLE(I INT),OUT outputVar TABLE (I INT, J DOUBLE))
```

Defines the tabular structure directly in the signature.

```
(IN inputVar tableType, OUT outputVar outputTableType)
```

Using previously defined `tableType` and `outputTableType` table types.

The advantage of previously defined table type is that it can be reused by other procedure and functions. The disadvantage is that you must take care of its lifecycle.

The advantage of a table variable structure that you directly define in the signature is that you do not need to take care of its lifecycle. In this case, the disadvantage is that it cannot be reused.

8.2 Local Table Variables

In SQLScript, the local table variable can be declared in a procedure or a user defined function.

Local table variables are, as the name suggests, variables with a reference to tabular data structure. This data structure originates from an SQL query.

8.3 Table Variable Type Definition

The type of a table variable in the body of a procedure or table function is either derived from the SQL Query oryou can declare it explicitly.

If the table variable derived its type from the SQL query, the SQLScript compiler determines the type from the first assignments of that variable. This provides a great deal of flexibility. One disadvantage, however, is that it generates many type conversions in the background because the derived table type does not always match the typed table parameters at the signature. This can lead to unnecessary conversion costs.

Another disadvantage is the cost for unnecessary internal statement compilation to derive the types.

To avoid this unnecessary cost, you can declare the type of a table variable explicitly.

Signature

```
DECLARE <sql_identifier> [{,<sql_identifier> }...] {TABLE
(<column_list_definition>)|<table_type>}
```

Local table variables are declared using the DECLARE keyword. For the referenced type you can either use previously declared table type or use the inplace type definition TABLE (`<column_list_definition>`). The next example illustrate both variants:

```
DECLARE temp TABLE (n int);
DECLARE temp MY_TABLE_TYPE;
```

Description

Local table variables are declared using the `DECLARE` keyword. You can reference a table variable `temp` by using `:temp`. The `<sql_identifier>` must be unique among all other scalar variables and table variables in the same code block. You can, however, use names that are identical to the name of another variable in a different code block. Additionally, you can reference these identifiers only in their local scope:

```
CREATE PROCEDURE exampleExplicit (OUT outTab TABLE(n int))
LANGUAGE SQLScript READS SQL DATA AS
BEGIN
```

```
     DECLARE temp TABLE (n int);
     temp = SELECT 1 as n FROM DUMMY ;
     BEGIN
        DECLARE temp TABLE (n int);
        temp = SELECT 2 as n FROM DUMMY ;
        outTab = Select * from :temp;
    END;
    outTab = Select * from :temp;
END;
call exampleExplicit(?);
```

In each block there are table variables declared with identical names. However, since the last assignment to the output parameter `<outTab>` can only have the reference of variable `<temp>` declared in the same block, the result is as follows:

```
 N
----
 1
```

```
CREATE PROCEDURE exampleDerived (OUT outTab TABLE(n int))
LANGUAGE SQLScript READS SQL DATA
AS
BEGIN
    temp = SELECT 1 as n FROM DUMMY ;
    BEGIN
        temp = SELECT 2 as n FROM DUMMY ;
        outTab = Select * from :temp;
    END;
    outTab = Select * from :temp;
END;
call exampleDerived (?);
```

In this code example, there is no explicit table variable declaration where done. It means the `<temp>` variable is visible among all blocks. For this reason, the result is as follows:

```
 N
----
 2
```

For every assignment of the explicit declared table variable, the derived column names and types on the right side are checked against the explicit declared type on the left side.

Another difference, compared to derived types, is that a reference to a table variable without assignment leads to an error during compile time:

```
BEGIN
    DECLARE a TABLE (i DECIMAL(2,1), j INTEGER);
    IF :num = 4
    THEN
        a = SELECT i, j FROM tab;
    END IF;
END;
```

The example above sends an error because table variable `<a>` is unassigned if `<:num>` is not 4. In comparison, the derived table variable type approach would send an error at runtime, but only if `<:num>` is not 4.

The following table shows the differences:

	Derived Type	Explicitly Declared
Create new variable	First SQL query assignment `tmp = select * from table;`	Table variable declaration in a block: `DECLARE tmp TABLE(i int);`
Variable scope	Global scope (global here means only within a procedure or function), regardless of the block where it was first declared	Available in declared block only. Variable hiding is applied.
Unassigned variable check	Passes the compile phase even though the variable can be unassigned in some cases. Error when unassigned variable is used in execution time. (Success if the variable was assigned in execution time)	An error in the compile phase if there is a possibility of a reference to an unassigned table variable. Checks only when a table variable is used. If a procedure passed compile phase, there is no possibility of a null referencing error during execution.

i Note

The NOT NULL option is not supported (see also the information about scalar variable declaration).

8.4 Binding Table Variables

Table variables are bound using the equality operator. This operator binds the result of a valid `SELECT` statement on the right side to an intermediate variable or an output parameter on the left side.

Statements on the right side can refer to input parameters or intermediate result variables bound by other statements. Recursion is not possible; that means cyclic dependencies that result from the intermediate result assignments or from calling other functions are not allowed.

This is an example of binding table variables.

1. Prepare tables `mytab` and `mytab2` and populate data:

```
create table mytab(a int, b int);
go
insert into mytab values (1, 2);
go
insert into mytab values (3, 4);
go
insert into mytab values (5, 6);
go
create table mytab2(a int, b int);
go
insert into mytab2 values (1, 2);
go
```

2. Assign to declared the table variable:

```
create procedure Example_p1
as begin
declare t1 table(c int, d tinyint);
t1 = select a, b from mytab;
select * from :t1;
end;
go
call Example_p1
go
c d
----------- ---
1 2
3 4
5 6
(3 rows affected)
(return status = 0)
```

3. Assign to the undefined variable:

```
create procedure Example_p2
as begin
t1 = select a, b from mytab;
select * from :t1;
end;
call Example_p2
go
a b
----------- -----------
1 2
3 4
5 6
(3 rows affected)
(return status = 0)
```

4. Truncate the table variable when it is used again:

```
create procedure Example_p3
as begin
t1 = select a, b from mytab;
t1 = select c, d from mytab2;
select * from :t1; --t1 only has records from mytab2
end;
call Example_p3
go
a b
----------- -----------
1 2
(1 row affected)
(return status = 0)
```

8.5 Referencing Variables

Bound variables are referenced by their name (for example, `<var>`). In the variable reference, the variable name is prefixed by `<:>`, such as `<:var>`. The procedure or table function describes a dataflow graph using statements and variables that connect the statements. The order in which statements are written in a body can differ from the order in which statements are evaluated. In case a table variable is bound multiple times, the order of these bindings is consistent with the order they appear in the body. Additionally, statements are only evaluated if the variables that are bound by the statement are consumed by another subsequent statement. Consequently, statements with results that are not consumed are removed during optimization.

Example

```
<lt_expensive_books> = SELECT title, price, crcy FROM :<it_books>
                     WHERE price > <:minPrice> AND crcy = <:currency>;
```

In this assignment, the variable `<lt_expensive_books>` is bound. The `<:it_books>` variable in the `FROM` clause refers to an `IN` parameter of a table type. It would also be possible to consume variables of type table in the `FROM` clause which were bound by an earlier statement. `<:minPrice>` and `<:currency>` refer to `IN` parameters of a scalar type.

SQLScript supports references on table variables column names. For example:

```
create procedure my_var(in v0 table(a int, b int), in v1 int) as
begin
select :v0.a from :v0 where :v0.b = 4;
end;
go
```

9 SAP ASE Imperative SQLScript Logic

SQLScript arrays, which are an indexed collection of elements of a single data type, are not supported in SAP ASE.

9.1 Local Scalar Variables

i Note

SAP ASE restrictions:

- The `NOT NULL` option is not supported.
- For `<proc_default>`, only the `CONSTANT` value is supported. `EXPRESSION` is not supported.

Syntax

```
DECLARE <sql_identifier> <type> [<proc_default>]
```

Syntax Elements

Default value expression assignment:

```
<proc_default> ::= (DEFAULT | =) <value>;
```

The value to be assigned to the variable:

```
<value>   !!= An element of the type specified by <type>
```

Description

Local variables are declared using the DECLARE keyword and, optionally, be initialized with their declaration. By default, scalar variables are initialized with NULL. You can reference a scalar variable `<var>` in the same way as `<:var>`, described above.

→ Tip

Use `:var` to access the value of the variable. Use `var` in your code to assign a value to the variable.

You can assign a value multiple times. Each assignment overwriting the previous value stored in the scalar variable. Assignment is performed using the = operator.

Although the `:=` operator does not generate an error, you should only use the = operator to define scalar variables.

Example

This example shows the various ways of making declarations and assignments:

```
CREATE PROCEDURE proc (OUT z INT) LANGUAGE SQLSCRIPT READS SQL DATA
AS
BEGIN
    DECLARE a int;
    DECLARE b int = 0;
    DECLARE c int DEFAULT 0;

       t = select * from baseTable ;
       select count(*) into a from :t;
       b = :a + 1;
       z = :b + :c;
end;
```

9.2 Global Session Variables

You can use global session variables in SQLScript to share a scalar value between procedures and functions that are running in the same session. The value of a global session variable is not visible from another session.

To set the value of a global session variable, use:

```
SET <key> = <value>;
```

While `<key>` can only be a constant string, `<value>` can be any expression, scalar variable, or function that returns a value that can convert to a string. Both have maximum length of 5000 characters. You cannot explicity type the session variable, which is a STRING data type. If `<value>` is not a STRING data type, it is implicitly converted to STRING.

This example shows how to set the value of a session variable in a procedure:

```
CREATE PROCEDURE CHANGE_SESSION_VAR (IN NEW_VALUE NVARCHAR(50))
AS
BEGIN
    SET 'MY_VAR' = :NEW_VALUE;
END
```

The following example uses the SESSION_CONTEXT (`<key>`) function to retrieve the session variable, which in this case is 'MY_VAR':

```
CREATE FUNCTION GET_VALUE ()
RETURNS var NVARCHAR(50)
AS
BEGIN
    var = SESSION_CONTEXT('MY_VAR');
END;
```

> i Note
>
> You cannot use SET `<key>` = `<value>` in functions and procedures flagged as READ ONLY (scalar and table functions are implicitly READ ONLY).

The maximum number of session variables can be configured with the configuration parameter max_session_variables under the section session (min=1, max=UINT32_MAX) . The default is 1024.

Session variables are null by default and can be reset to null using UNSET `<key>`.

9.3 Variable Scope Nesting

SQLScript supports local variable declarations in a nested block. Local variables are only visible in the scope of the block in which they are defined.

You may define local variables inside LOOP, WHILE, FOR, or IF-ELSE control structures. For example:

```
CREATE PROCEDURE nested_block(OUT val INT)
    AS
    BEGIN
        DECLARE a INT = 1;
        BEGIN
            DECLARE a INT = 2;
            BEGIN
            DECLARE a INT;
                a = 3;
            END;
            val = a;
        END;
    END;
```

When you call this procedure, the result is:

```
call nested_block(?)
--> OUT:[2]
```

In this result, the innermost nested block value of 3 has not been passed to the `val` variable. The following example redefines the procedure without the innermost `DECLARE` statement:

```
DROP PROCEDURE nested_block;
CREATE PROCEDURE nested_block(OUT val INT)
    AS
    BEGIN
        DECLARE a INT = 1;
        BEGIN
            DECLARE a INT = 2;
            BEGIN
                a = 3;
            END;
            val = a;
        END;
    END;
```

The results of the modified procedure are :

```
call nested_block(?)
--> OUT:[3]
```

The results show that the innermost nested block uses the variable declared in the second-level nested block.

Local Variables in Control Structures

This example uses conditionals:

```
CREATE FUNCTION MYDIV(a INT, b INT) RETURNS x INT
    AS
    BEGIN
        IF b = 0 THEN
        SIGNAL SQL_ERROR_CODE 20001 SET MESSAGE_TEXT='devided by zero';
        END IF;
        x = a /b;
    END;
```

```
CREATE PROCEDURE nested_block_if(IN inval INT, OUT val INT)
    AS
    BEGIN
        DECLARE a INT = 1;
        DECLARE v INT = 0;
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
            BEGIN
            val = :a;
            END;
            v = MYDIV(1, 1-:inval);
            IF :a = 1 THEN
                DECLARE a INT = 2;
                DECLARE EXIT HANDLER FOR SQLEXCEPTION
                BEGIN
                    val = :a;
                END;
                v = MYDIV(1, 2-:inval);
                IF :a = 2 THEN
                    DECLARE a INT = 3;
                    DECLARE EXIT HANDLER FOR SQLEXCEPTION
                        BEGIN
                            val = :a;
                        END;
                        v = MYDIV(1, 3-:inval);
                END IF;
                v = MYDIV(1, 4-:inval);
            END IF;
            v = MYDIV(1, 5-:inval);
    END;
call nested_block_if(1, ?)
-->OUT:[1]
call nested_block_if(2, ?)
-->OUT:[2]
call nested_block_if(3, ?)
```

```
-->OUT:[3]
call nested_block_if(4, ?)
--> OUT:[2]
call nested_block_if(5, ?)
--> OUT:[1]
```

This example uses a while loop:

```
CREATE PROCEDURE nested_block_while(OUT val INT)
    AS
    BEGIN
        DECLARE v int = 2;
        val = 0;
         WHILE v > 0
         DO
                DECLARE a INT = 0;
                a = :a + 1;
                val = :val + :a;
                v = :v - 1;
          END WHILE;
    END;
call nested_block_while(?)
--> OUT:[2]
```

This example uses a for loop:

```
CREATE TABLE mytab1(a int);
CREATE TABLE mytab2(a int);
CREATE TABLE mytab3(a int);
INSERT INTO mytab1 VALUES(1);
INSERT INTO mytab2 VALUES(2);
INSERT INTO mytab3 VALUES(3);
CREATE PROCEDURE nested_block_for(IN inval INT, OUT val INT)
    AS
    BEGIN
        DECLARE a1 int default 0;
        DECLARE a2 int default 0;
        DECLARE a3 int default 0;
        DECLARE v1 int default 1;
        DECLARE v2 int default 1;
        DECLARE v3 int default 1;
        DECLARE CURSOR C FOR SELECT * FROM mytab1;
        FOR R as C DO
            DECLARE CURSOR C FOR SELECT * FROM mytab2;
            a1 = :a1 + R.a;
            FOR R as C DO
                 DECLARE CURSOR C FOR SELECT * FROM mytab3;
                 a2 = :a2 + R.a;
                       FOR R as C DO
                       a3 = :a3 + R.a;
                       END FOR;
              END FOR;
        END FOR;
    IF inval = 1 THEN
        val = :a1;
    ELSEIF inval = 2 THEN
        val = :a2;
    ELSEIF inval = 3 THEN
        val = :a3;
    END IF;
END;
call nested_block_for(1, ?)
--> OUT:[1]
call nested_block_for(2, ?)
--> OUT:[2]
call nested_block_for(3, ?)
```

```
--> OUT:[3]
```

This example uses a loop.

The example below uses tables and values created in the `For Loop` example above.

```
CREATE PROCEDURE nested_block_loop(IN inval INT, OUT val INT)
    AS
    BEGIN
        DECLARE a1 int;
        DECLARE a2 int;
        DECLARE a3 int;
        DECLARE v1 int default 1;
        DECLARE v2 int default 1;
        DECLARE v3 int default 1;
        DECLARE CURSOR C FOR SELECT * FROM mytab1;
        OPEN C;
        FETCH C into a1;
        CLOSE C;
        LOOP
            DECLARE CURSOR C FOR SELECT * FROM mytab2;
            OPEN C;
            FETCH C into a2;
            CLOSE C;
                LOOP
                    DECLARE CURSOR C FOR SELECT * FROM mytab3;
                    OPEN C;
                    FETCH C INTO a3;
                    CLOSE C;
                    IF :v2 = 1 THEN
                        BREAK;
                       END IF;
                END LOOP;
                IF :v1 = 1 THEN
                    BREAK;
                END IF;
        END LOOP;
            IF :inval = 1 THEN
                  val = :a1;
            ELSEIF :inval = 2 THEN
                  val = :a2;
            ELSEIF :inval = 3 THEN
                  val = :a3;
            END IF;
    END;
call nested_block_loop(1, ?)
--> OUT:[1]
call nested_block_loop(2, ?)
--> OUT:[2]
call nested_block_loop(3, ?)
--> OUT:[3]
```

9.4 Control Structures

9.4.1 Conditionals

The `IF` statement consists of a Boolean expression `<bool_expr1>`. If this expression evaluates to true then the statements `<then_stmts1>` in the mandatory `THEN` block are executed. The `IF` statement ends with `END IF`. The remaining parts are optional.

The `IF` statement consists of a Boolean expression `<bool_expr1>`. If this expression evaluates to true then the statements `<then_stmts1>` in the mandatory `THEN` block are executed. The `IF` statement ends with `END IF`. The remaining parts are optional.

If the Boolean expression `<bool_expr1>` does not evaluate to true the `ELSE`-branch is evaluated. The statements`<else_stmts3>` are executed without further checks. After an else branch no further `ELSE` branch or `ELSEIF` branch is allowed.

Alternatively, when `ELSEIF` is used instead of `ELSE` a further Boolean expression >`<then_stmts2>` are executed. In this manner an arbitrary number of `ELSEIF` clauses can be added.

This statement can be used to simulate the switch-case statement known from many programming languages.

Syntax:

```
IF <bool_expr1>
THEN
    <then_stmts1>
[{ELSEIF <bool_expr2>
THEN
    <then_stmts2>}...]
[ELSE
    <else_stmts3>]
END IF
```

Syntax elements:

```
<bool_expr1>   ::= <condition>
<bool_expr2>   ::= <condition>
<condition>    ::= <comparison> | <null_check>
<comparison>   ::= <comp_val> <comparator> <comp_val>
<null_check>   ::= <comp_val> IS [NOT] NULL
```

Tests if `<comp_val>` is `NULL` or `nullNOT NULL`.

i Note

`NULL` is the default value for all local variables.

See *Example 2* for an example use of this comparison.

```
<comparator>   ::= < | > | = | <= | >= | !=
<comp_val>     ::= <scalar_expression>|<scalar_udf>
<scalar_expression> ::=<scalar_value>[{operator}<scalar_value>…]
<scalar_value> ::= <numeric_literal> | <exact_numeric_literal>|
<unsigned_numeric_literal>
<operator>::=+|-|/|*
```

Specifies the comparison value. This can be based on either scalar literals or scalar variables.

```
<then_stmts1> ::= <proc>
<then_stmts2> ::= <proc_stmts>
<else_stmts3> ::= <proc_stmts>
```

```
<proc_stmts> ::= !! SQLScript procedural statements
```

Defines procedural statements to be executed dependent on the preceding conditional expression.

Examples:

Example 1

Use the `IF` statement to implement the SAP HANA database`s `UPSERT` statement functionality:

```
CREATE PROCEDURE upsert_proc (IN v_isbn VARCHAR(20))
LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE found INT = 1;
    SELECT count(*) INTO found FROM books WHERE isbn = :v_isbn;
    IF :found = 0
    THEN
        INSERT INTO books
        VALUES (:v_isbn, 'In-Memory Data Management', 1, 1,
                '2011', 42.75, 'EUR');
    ELSE
        UPDATE books SET price = 42.75 WHERE isbn =:v_isbn;
    END IF;
END;
```

Example 2

Use the `IF` statement to check if the `:found` variable is `NULL`:

```
SELECT count(*) INTO found FROM books WHERE isbn = :v_isbn;
IF :found IS NULL THEN
   CALL ins_msg_proc('result of count(*) cannot be NULL');
ELSE
   CALL ins_msg_proc('result of count(*) not NULL - as expected');
END IF;
```

Example 3

You can also use a scalar UDF in the condition, as shown in the following example:

```
CREATE PROCEDURE proc (in input1 INTEGER, out output1 TYPE1)
AS
BEGIN
    DECLARE i INTEGER DEFAULT :input1;
    IF SUDF(:i) = 1 THEN
         output1 = SELECT value FROM T1;
    ELSEIF SUDF(:i) = 2 THEN
         output1 = SELECT value FROM T2;
    ELSE
         output1 = SELECT value FROM T3;
    END IF;
END;
```

9.4.2 While Loop

Sets a condition for the repeated execution of a statement or statement block. The statements are executed repeatedly, as long as the specified condition is true.

Syntax:

```
WHILE <condition> DO
    <proc_stmts>
END WHILE
```

Syntax elements:

```
<null_check>    ::= <comp_val> IS [NOT] NULL
<comparator>    ::= < | > | = | <= | >= | !=
<comp_val>    ::= <scalar_expression>|<scalar_udf>
<scalar_expression> ::=<scalar_value>[{operator}<scalar_value>…]
<scalar_value> ::= <numeric_literal> | <exact_numeric_literal>|
<unsigned_numeric_literal>
<operator> ::= +|-|/|*
```

Defines a Boolean expression which evaluates to true or false.

```
<proc_stmts> ::= !! SQLScript procedural statements
```

Description:

The while loop executes the statements `<proc_stmts>` in the body of the loop as long as the Boolean expression at the beginning `<condition>` of the loop evaluates to true.

Example 1

Uses `WHILE` to increment the `:v_index1` and `:v_index2` variables using nested loops:

```
CREATE PROCEDURE proc WHILE (OUT v_index2 INTEGER)
AS
BEGIN
    DECLARE v_index1 INT = 0;
    WHILE :v_index1 < 5 DO
        v_index2 = 0;
        WHILE :v_index2 < 5 DO
            v_index2 = :v_index2 + 1;
        END WHILE;
        v_index1 = :v_index1 + 1;
    END WHILE;
END;
```

Example 2

Uses scalar UDF for the WHILE condition:

```
CREATE PROCEDURE proc (in input1 INTEGER, out output1 TYPE1)
AS
BEGIN
    DECLARE i INTEGER DEFAULT :input1;
    DECLARE cnt INTEGER DEFAULT 0;
    WHILE SUDF(:i) > 0 DO
        cnt = :cnt + 1;
        i = :i - 1;
    END WHILE;
    output1 = SELECT value FROM T1 where id = :cnt ;
```

```
END;
```

⚠ Caution

No specific checks are performed to avoid infinite loops.

9.4.3 For Loop

The FOR loop iterates a range of numeric values and binds the current value to a variable `<loop-var>` in ascending order. Iteration starts with the value of `<start_value>` and is incremented by one until the `<loop-var>` is greater than `<end_value>`.

If `<start_value>` is larger than `<end_value>`, `<proc_stmts>` in the loop will not be evaluated.

Syntax:

```
FOR <loop-var> IN [REVERSE] <start_value> .. <end_value> DO
    <proc_stmts>
END FOR
```

Syntax elements:

```
<loop-var> ::= <identifier>
```

Defines the variable that will contains the loop values.

```
REVERSE
```

When defined causes the loop sequence to occur in a descending order.

```
<start_value> ::= <signed_integer>
```

Defines the starting value of the loop.

```
<end_value> ::=  <signed_integer>
```

Defines the end value of the loop.

```
<proc_stmts> ::= !! SQLScript procedural statements
```

Defines the procedural statements that will be looped over.

Example 1

Uses nested `FOR` loops to call a procedure that traces the current values of the loop variables appending them to a table:

```
CREATE PROCEDURE proc (out output1 TYPE1)
AS
BEGIN
    DECLARE pos INTEGER DEFAULT 0;
    DECLARE i INTEGER;
    FOR i IN 1..10 DO
        pos = :pos + 1;
    END FOR;
    output1 = SELECT value FROM T1 where position = :i ;
END;
```

Example 2

Uses scalar UDF in the `FOR` loop:

```
CREATE PROCEDURE proc (out output1 TYPE1)
AS
BEGIN
    DECLARE pos INTEGER DEFAULT 0;
    DECLARE i INTEGER;
    FOR i IN 1..SUDF_ADD(1, 2) DO
        pos = :pos + 1;
    END FOR;
    output1 = SELECT value FROM T1 where position = :i ;
END;
```

9.4.4 Break and Continue

BREAK specifies that a loop should stop being processed. CONTINUE specifies that a loop should stop processing the current iteration, and should immediately start processing the next.

Syntax:

```
BREAK
CONTINUE
```

Description:

These statements provide internal control functionality for loops.

Example:

Defines a loop sequence. If the loop value `:x` is less than 3, the iterations are skipped. If `:x` is 5, then the loop terminates.

```
CREATE PROCEDURE proc ()
AS
BEGIN
    DECLARE x  integer;
    FOR x IN 0 .. 10 DO
        IF :x < 3 THEN
            CONTINUE;
        END IF;
        IF :x = 5 THEN
            BREAK;
        END IF;
    END FOR;
END;
```

9.5 Cursors

Cursors are used to fetch single rows from the result set returned by a query. When the cursor is declared it is bound to a query. It is possible to parameterize the cursor query.

9.5.1 Define Cursor

Cursors can be defined either after the signature of the procedure and before the procedure's body, or at the beginning of a block with the `DECLARE` token. The cursor is defined with a name, optionally a list of parameters, and an SQL `SELECT` statement. The cursor provides the functionality to iterate through a query result row by row. Updating cursors is not supported.

> **i Note**
>
> Avoid using cursors when you can express the same logic with SQL. You should do this as cursors cannot be optimized the same way SQL can.

Syntax:

```
CURSOR <cursor_name> [({<param_def>{,<param_def>} ...)]
           FOR <select_stmt>
```

Syntax elements:

Specifies the name of the cursor:

```
<cursor_name> ::= <identifier>
```

Defines an optional SELECT parameter:

```
<param_def> = <param_name> <param_type>
```

Defines the variable name of the parameter:

```
<param_name> ::= <identifier>
```

Defines the data type of the parameter.

```
<param_type> ::= DATE | TIME | SECONDDATE | TIMESTAMP | TINYINT
               | SMALLINT | INTEGER | BIGINT | SMALLDECIMAL | DECIMAL
               | REAL | DOUBLE | VARCHAR | NVARCHAR | ALPHANUM
               | VARBINARY | BLOB | CLOB | NCLOB
```

Defines an SQL select statement. See SELECT:

```
<select_stmt> !!= SQL SELECT statement.
```

Example:

Creates a cursor `c_cursor1` to iterate over results from a `SELECT` on the `books` table. The cursor passes one parameter `v_isbn` to the `SELECT` statement:

```
DECLARE CURSOR c_cursor1 (v_isbn VARCHAR(20)) FOR
             SELECT isbn, title, price, crcy FROM books
             WHERE isbn = :v_isbn ORDER BY isbn;
```

9.5.2 Looping over Result Sets

Opens a previously declared cursor and iterates over each row in the result set of the query bound to the cursor. For each row in the result set the statements in the body of the procedure are executed. After the last row from the cursor has been processed, the loop is exited and the cursor is closed.

→ Tip

Resource leaks can be avoided because this loop method takes care of opening and closing cursors. Consequently this loop is preferred to opening and closing a cursor explicitly and using other loop-variants.

Within the loop body, the attributes of the row that the cursor currently iterates over can be accessed like an attribute of the cursor. Assuming `<row_var>` is `a_row` and the iterated data contains a column `test`, then the value of this column can be accessed using `a_row.test`.

Syntax:

```
FOR <row_var> AS <cursor_name>[(<argument_list>)] DO
<proc_stmts> | {<row_var>.<column>}
END FOR
```

Syntax elements:

Defines an identifier to contain the row result:

```
<row_var> ::= <identifier>
```

Specifies the name of the cursor to be opened:

```
<cursor_name> ::= <identifier>
```

Specifies one or more arguments to be passed to the select statement of the cursor:

```
<argument_list> ::= <arg>[,{<arg>}...]
```

Specifies a scalar value to be passed to the cursor:

```
<arg> ::= <scalar_value>
```

Defines the procedural statements that will be looped over:

```
<proc_stmts> ::= !! SQLScript procedural statements
```

To access the row result attributes in the body of the loop you use the syntax shown:

```
<row_var>.<column> ::= !! Provides attribute access
```

Example:

Uses a `FOR`-loop to loop over the results from `c_cursor1`:

```
CREATE PROCEDURE foreach_proc() LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE v_isbn    VARCHAR(20) = '';
    DECLARE CURSOR c_cursor1 (v_isbn VARCHAR(20)) FOR
               SELECT isbn, title, price, crcy FROM books
               ORDER BY isbn;
    FOR cur_row AS c_cursor1(v_isbn)
    DO
        CALL ins_msg_proc('book title is: ' || cur_row.title);
    END FOR;
END;
```

9.5.3 Open Cursor

Evaluates the query bound to a cursor and opens the cursor so that the result can be retrieved. When the cursor definition contains parameters then the actual values for each of these parameters must be provided when the cursor is opened.

This statement prepares the cursor so the results can be fetched for the rows of a query.

Syntax:

```
OPEN <cursor_name>[(<argument_list>)]
```

Syntax elements:

Specifies the name of the cursor to be opened:

```
<cursor_name> ::= <identifier>
```

Specifies one or more arguments to be passed to the select statement of the cursor:

```
<argument_list> ::= <arg>[,{<arg>}...]
```

Specifies a scalar value to be passed to the cursor:

```
<arg> ::= <scalar_value>
```

Example:

Opens the cursor `c_cursor1` and passes the `'978-3-86894-012-1'` string as a parameter:

```
OPEN c_cursor1('978-3-86894-012-1');
```

9.5.4 Close Cursor

Closes a previously opened cursor and releases all associated state and resources. It is important to close all cursors that were previously opened.

Syntax:

```
CLOSE <cursor_name>
```

Syntax elements:

Specifies the name of the cursor to be closed:

```
<cursor_name> ::= <identifier>
```

Example:

Closes the cursor `c_cursor1`.

```
CLOSE c_cursor1;
```

9.5.5 Fetch Query Results of a Cursor

Fetches a single row in the result set of a query and advances the cursor to the next row. This assumes that the cursor was declared and opened before. One can use the cursor attributes to check if the cursor points to a valid row.

See Attributes of a Cursor [page 289].

Syntax:

```
FETCH <cursor_name> INTO <variable_list>
```

Syntax elements:

Specifies the name of the cursor where the result is obtained:

```
<cursor_name> ::= <identifier>
```

Specifies the variables where the row result from the cursor is stored:

```
<variable_list> ::= <var>[,{<var>}...]
```

Specifies the identifier of a variable:

```
<var> ::= <identifier>
```

Example:

Fetches a row from the cursor `c_cursor1` and stores the results in the variables shown:

```
FETCH c_cursor1 INTO v_isbn, v_title, v_price, v_crcy;
```

9.5.6 Attributes of a Cursor

A cursor provides a number of methods to examine its current state. This table summarizes the attributes for a cursor bound to variable `c_cursor1`:

Table 6: Cursor Attributes

Attribute	Description
`c_cursor1::ISCLOSED`	Is true if cursor `c_cursor1` is closed, otherwise false.
`c_cursor1::NOTFOUND`	Is true if the previous fetch operation returned no valid row, otherwise false. Before calling `OPEN` or after calling `CLOSE` on a cursor this will always return true.
`c_cursor1::ROWCOUNT`	Returns the number of rows that the cursor fetched so far. This value is available after the first `FETCH` operation. Before the first fetch operation, the number is 0.

Example

Shows a complete procedure using the attributes of the cursor `c_cursor1` to check if fetching a set of results is possible:

```
CREATE PROCEDURE cursor_proc LANGUAGE SQLSCRIPT AS
BEGIN
    DECLARE v_isbn    VARCHAR(20);
    DECLARE v_title VARCHAR(20);
    DECLARE v_price DOUBLE;
    DECLARE v_crcy VARCHAR(20);
    DECLARE CURSOR c_cursor1 (v_isbn VARCHAR(20)) FOR
       SELECT isbn, title, price, crcy FROM books
       WHERE isbn = :v_isbn ORDER BY isbn;
    OPEN c_cursor1('978-3-86894-012-1');
    IF c_cursor1::ISCLOSED THEN
       CALL ins_msg_proc('WRONG: cursor not open');
    ELSE
       CALL ins_msg_proc('OK: cursor open');
    END IF;
    FETCH c_cursor1 INTO v_isbn, v_title, v_price, v_crcy;
    IF c_cursor1::NOTFOUND THEN
       CALL ins_msg_proc('WRONG: cursor contains no valid data');
    ELSE
       CALL ins_msg_proc('OK: cursor contains valid data');
    END IF;
    CLOSE c_cursor1;
END
```

9.6 Autonomous Transaction

Syntax:

```
<proc_bloc> :: = BEGIN AUTONOMOUS TRANSACTION
        [<proc_decl_list>]
        [<proc_handler_list>]
        [<proc_stmt_list>]
```

```
END;
```

Description:

The autonomous transaction is independent from the main procedure. Changes made and committed by an autonomous transaction can be stored in persistency regardless of commit/rollback of the main procedure transaction. The end of the autonomous transaction block has an implicit commit.

```
BEGIN AUTONOMOUS TRANSACTION
    …(some updates) –(1)
    COMMIT;
    …(some updates) –(2)
    ROLLBACK;
    …(some updates) –(3)
END;
```

The examples show how commit and rollback work inside the autonomous transaction block. The first updates (1) are committed, whereby the updates made in step (2) are completely rolled back. And the last updates (3) are committed by the implicit commit at the end of the autonomous block.

```
CREATE PROCEDURE PROC1( IN p INT , OUT outtab TABLE (A INT)) LANGUAGE SQLSCRIPT
AS
BEGIN
          DECLARE errCode INT;
          DECLARE errMsg VARCHAR(5000);
          DECLARE EXIT HANDLER FOR SQLEXCEPTION
          BEGIN AUTONOMOUS TRANSACTION
              errCode= ::SQL_ERROR_CODE;
              errMsg=  ::SQL_ERROR_MESSAGE ;
              INSERT INTO ERR_TABLE (PARAMETER,SQL_ERROR_CODE, SQL_ERROR_MESSAGE)
                      VALUES ( :p, :errCode, :errMsg);
          END;
          outtab = SELECT 1/:p as A FROM DUMMY;    -- DIVIDE BY ZERO Error if p=0
END
```

In the example above, an autonomous transaction is used to keep the error code in the `ERR_TABLE` stored in persistency.

If the exception handler block were not an autonomous transaction, then every insert would be rolled back because they were all made in the main transaction. In this case the result of the `ERR_TABLE` is as shown in the following example.

```
 P |SQL_ERROR_CODE| SQL_ERROR_MESSAGE
--------------------------------------------
0  |      304     | division by zero undefined:  at function /()
```

It is also possible to have nested autonomous transactions.

```
CREATE PROCEDURE P2()
AS BEGIN
    BEGIN AUTONOMOUS TRANSACTION
           INSERT INTO LOG_TABLE VALUES ('MESSAGE');
           BEGIN AUTONOMOUS TRANSACTION
                  ROLLBACK;
           END;
    END;
END;
```

The `LOG_TABLE` table contains `'MESSAGE'`, even though the inner autonomous transaction rolled back.

Supported statements inside the block

- `SELECT`, `INSERT`, `DELETE`, `UPDATE`, `UPSERT`, `REPLACE`
- `IF`, `WHILE`, `FOR`, `BEGIN/END`
- `COMMIT`, `ROLLBACK`, `RESIGNAL`, `SIGNAL`
- Scalar variable assignment

Unsupported statements inside the block

- Calling other procedures
- DDL
- Cursor
- Table assignments

> **i Note**
>
> You have to be cautious if you access a table both before and inside an autonomous transaction started in a nested procedure (e.g. TRUNCATE, update the same row), because this can lead to a deadlock situation. One solution to avoid this is to commit the changes before entering the autonomous transaction in the nested procedure.

9.7 Dynamic SQL

Dynamic SQL allows you to construct an SQL statement during the execution time of a procedure. While dynamic SQL allows you to use variables where they might not be supported in SQLScript and also provides more flexibility in creating SQL statements, it does have the disadvantage of an additional cost at runtime:

- Opportunities for optimizations are limited.
- The statement is potentially recompiled every time the statement is executed.
- You cannot use SQLScript variables in the SQL statement.
- You cannot bind the result of a dynamic SQL statement to a SQLScript variable.
- You must be very careful to avoid SQL injection bugs that might harm the integrity or security of the database.

> **i Note**
>
> You should avoid dynamic SQL wherever possible as it can have a negative impact on security or performance.

9.7.1 EXEC

Syntax:

```
EXEC '<sql-statement>'
```

Description:

`EXEC` executes the SQL statement passed in a string argument.

Example:

You use dynamic SQL to insert a string into the `message_box` table.

```
v_sql1 = 'Third message from Dynamic SQL';
EXEC 'INSERT INTO message_box VALUES (''' || :v_sql1 || ''')';
```

9.7.2 EXECUTE IMMEDIATE

Syntax:

```
EXECUTE IMMEDIATE '<sql-statement>'
```

Description:

`EXECUTE IMMEDIATE` executes the SQL statement passed in a string argument. The results of queries executed with `EXECUTE IMMEDIATE` are appended to the procedures result iterator.

Example:

You use dynamic SQL to delete the contents of table `tab`, insert a value and finally to retrieve all results in the table.

```
CREATE TABLE tab (i int);
CREATE PROCEDURE proc_dynamic_result2(i int) AS
BEGIN
    EXEC 'DELETE from tab';
    EXEC 'INSERT INTO tab VALUES (' || :i || ')';
    EXECUTE IMMEDIATE 'SELECT * FROM tab ORDER BY i';
 END;
```

10 Exception Handling in SAP ASE SQLScript

Exception handling is a method for managing exception and completion conditions in a SQLScript procedure. SQLScript includes syntax for exception handling for the DECLARE EXIT HANDLER, DECLARE CONDITION, SIGNAL, and RESIGNAL commands.

Generally, exceptions can be managed with an exception handler declared at the beginning of statements that make an explicit signal exception.

i Note

Because error code numbers in SAP HANA and SAP ASE are different, you cannot specify system error code numbers in your exception handler to catch SAP ASE and SAP HANA errors.

In this example from an industry data collection program, there are a number of characteristic to be measured, including length, weight, outer diameter, and internal diameter. Each characteristic has its specification, including upper specification limitation (`USL`) and lower specification limitation (`LSL`). A worker may measure the data using meters. If data is outside the range of speciation, the program throws an exception and drops it.

```
CREATE TABLE TB_CHARA(ID INT, NAME VARCHAR(50), USL DOUBLE, LSL DOUBLE);
INSERT INTO TB_CHARA VALUES(1, 'length', 9.65, 9.39);

CREATE PROCEDURE MYPROC (spec_id INT, data DOUBLE) AS
BEGIN
    DECLARE usl, lsl DOUBLE;
    DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 20005
    BEGIN
        SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
        SELECT :usl, :lsl, :data FROM DUMMY;
    END;
    SELECT USL, LSL INTO usl, lsl FROM TB_CHARA WHERE ID = spec_id;
    IF data < lsl OR data > usl THEN
        SIGNAL SQL_ERROR_CODE 20005;  --raise error: 20005
        SELECT 'NeverReached_noContinueOnErrorSemantics' FROM DUMMY;
    END IF;
END;
CALL MYPROC(1, 9.25);
```

10.1 DECLARE EXIT HANDLER

The `DECLARE EXIT HANDLER` parameter allows you to define an exit handler to process exception conditions in your procedure or function.

The syntax is:

```
DECLARE EXIT HANDLER FOR <proc_condition_value>  {,<proc_condition_value>}...]
<proc_stmt>

<proc_condition_value> ::= SQLEXCEPTION
```

```
    | SQL_ERROR_CODE <error_code>
    | <condition_name>
```

For example, the following exit handler catches all `SQLEXCEPTION` occurrences and returns information that an exception was thrown:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'EXCEPTION was thrown' AS ERROR
FROM dummy;
```

Use the `::SQL_ERROR_CODE` and `::SQL_ERROR_MESSAGE` system variables to receive the error codes and messages. For example:

```
CREATE PROCEDURE MYPROC (IN in_var INTEGER, OUT outtab TABLE(I INTEGER) ) AS
BEGIN
   DECLARE grade int;
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
   SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;

   IF in_var < 60 THEN
      SIGNAL SQL_ERROR_CODE 20001 SET  MESSAGE_TEXT ='value is too small.';
   ELSEIF in_var <= 70 THEN
      grade = 4;
   ELSEIF in_var <= 80 THEN
      grade = 3;
   ELSEIF in_var <= 90 THEN
      grade = 2;
   ELSEIF in_var <= 100 THEN
      grade = 1;
   ELSE
      SIGNAL SQL_ERROR_CODE 20002 SET  MESSAGE_TEXT ='value is too big.';
   END IF;
   outtab = SELECT grade as I FROM dummy;
END;
```

In addition to defining an exit handler for an arbitrary `SQLEXCEPTION` occurrence, you can define it for a specific error code number with the keyword `SQL_ERROR_CODE`, followed by an SQL error code number.

For example:

```
DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 20003
BEGIN
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
END;
IF in_var <60 || in_var > 100 THEN
      SIGNAL SQL_ERROR_CODE 20003 SET  MESSAGE_TEXT ='invalid input.';
END IF;
tab = SELECT (CASE WHEN in_var <= 70 THEN 4 WHEN in_var <= 80 THEN 3 WHEN in_var
<= 90 THEN 2 ELSE 1 END) as I FROM dummy;
```

You can define the exit handler for a condition instead of using an error code.

Use the `BEGIN…END` to add complexity to the exit handler. This example prepares additional information and inserts the error into a table:

```
DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 20003
BEGIN
   DECLARE procedure_name NVARCHAR(500) = "my_proc"

   DECLARE parameters NVARCHAR(255) =
      'IN_VAR = '||:in_var;

   INSERT INTO LOG_TABLE VALUES ( ::SQL_ERROR_CODE,
      ::SQL_ERROR_MESSAGE,
      :procedure_name,
      :parameters );
END;
IF in_var <60 || in_var > 100 THEN
SIGNAL SQL_ERROR_CODE 20003 SET MESSAGE_TEXT ='invalid input.';
END IF;
tab = SELECT (CASE WHEN in_var <= 70 THEN 4 WHEN in_var <= 80 THEN 3 WHEN in_var
<= 90 THEN 2 ELSE 1 END) as I FROM dummy;
```

10.2 DECLARE CONDITION

Declaring a `CONDITION` variable allows you to specify individual SQL error codes or create a user-defined condition.

The syntax is:

```
DECLARE <condition name> CONDITION [ FOR SQL_ERROR_CODE <error_code> ];
```

> **i Note**
>
> User-defined error codes must be greater than or equal to 20000.

You can use variables in an `EXIT HANDLER` declaration as well as in `SIGNAL` and `RESIGNAL` statements. However, `SIGNAL` and `RESIGNAL` statements allow only user-defined conditions.

Use condition variables for specific SQL error codes to make the procedure or function code more readable. For example, instead of using user-defined error code 20005, which signals an error, issue this to declare a more meaningful condition:

```
DECLARE data_beyond_spec CONDITION FOR SQL_ERROR_CODE 20005;
```

This is the corresponding EXIT HANDLER:

```
ECLARE EXIT HANDLER FOR data_beyond_spec
                SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
```

You can declare a user-defined condition instead of declaring a condition (with or without a user-defined error code) for an existing SQL error code. This syntax specifies a user-defined condition for an invalid procedure:

```
DECLARE invalid_input CONDITION;
```

Optionally, you can associate a user-defined error code (in this case, 20000):

```
DECLARE invalid_input CONDITION FOR SQL_ERROR_CODE 20000;
```

10.3 SIGNAL and RESIGNAL

Use the `SIGNAL` statement to explicitly raise a user-defined exception from within your procedure or function.

The syntax is:

```
SIGNAL (<user_defined_condition> | SQL_ERROR_CODE <int_const> )[SET MESSAGE_TEXT
= '<message_string>']
```

The error value returned by the `SIGNAL` statement is either a `SQL_ERROR_CODE` or a user-defined condition. User-defined error codes must be at least 20000. This example signals a SQL_ERROR_CODE of 20000:

```
SIGNAL SQL_ERROR_CODE 20000;
```

To raise a user-defined condition named `invalid_input`, issue:

```
SIGNAL invalid_input;
```

Although these user-defined exceptions in this example do not include the error message text — which means the value of the system variable `::SQL_ERROR_MESSAGE` is empty — the value of `::SQL_ERROR_CODE` is 20000.

Both statements produce the following information when the user-defined exception is thrown:

```
Could not execute the procedure. user-defined error: [20000] user-defined error
exception: my error message
(return status = -6)
```

Use `SET MESSAGE_TEXT` to set a corresponding error message. For example:

```
SIGNAL invalid_input SET MESSAGE_TEXT = 'Invalid input arguments';
```

The resulting user-defined exception looks similar to:

```
Could not execute the procedure. user-defined error: [20000] user-defined error
exception: Invalid input arguments
(return status = -6)
```

In this example, the procedure signals an error if the `start_date` input argument is greater than the `end_date` input argument:

```
CREATE PROCEDURE GET_CUSTOMERS( IN start_date DATE,
                IN end_date DATE,
                OUT aCust TABLE (first_name NVARCHAR(255),
                last_name NVARCHAR(255))
                )
                AS
                BEGIN
                DECLARE invalid_input CONDITION FOR SQL_ERROR_CODE 20000;

                IF :start_date > :end_date THEN
                SIGNAL invalid_input SET MESSAGE_TEXT =
                'START_DATE = '||:start_date||' > END_DATE =
'
                ||:end_date;
                END IF;

                aCust = SELECT first_name, last_name
```

```
                 FROM CUSTOMER C
                 WHERE    c.bdate >= :start_date
                 AND c.bdate <= :end_date;
                 END;
```

Users receive this error message if they call the `GET_CUSTOMERS` procedure with invalid input arguments:

```
Could not execute the procedure. user-defined error: [20000] user-defined error
exception: START_DATE = 2011-03-03 > END_DATE = 2010-03-03
(return status = -6)
```

The `RESIGNAL` statement passes the exception that is handled in the exit handler. The syntax is:

```
RESIGNAL [<user_defined_condition> | SQL_ERROR_CODE <int_const> ] [SET
MESSAGE_TEXT = '<message_string>']
```

`RESIGNAL` allows you to pass the original exception and change some of the information before passing it on. The `RESIGNAL` statement can be used only in the exit handler.

Use `RESIGNAL` statement syntax similar to this to not change the related information of an exception:

```
CREATE PROCEDURE MYPROC (IN in_var INTEGER, OUT outtab TABLE(I INTEGER) ) AS
BEGIN
   DECLARE grade int;
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
      RESIGNAL;

   IF in_var < 60 THEN
      SIGNAL SQL_ERROR_CODE 20001 SET  MESSAGE_TEXT ='value is too small.';
   ELSEIF in_var <= 70 THEN
      grade = 4;
   ELSEIF in_var <= 80 THEN
      grade = 3;
   ELSEIF in_var <= 90 THEN
      grade = 2;
   ELSEIF in_var <= 100 THEN
      grade = 1;
   ELSE
      SIGNAL SQL_ERROR_CODE 20002 SET  MESSAGE_TEXT ='value is too big.';
   END IF;
   outtab = SELECT grade as I FROM dummy;
END;
```

A value of `<in_var>` = 0 raises an error consisting of the original SQL error code and message text.

Use `SET MESSAGE _TEXT` to change the error message of a SQL error:

```
CREATE PROCEDURE MYPROC (IN in_var INTEGER, OUT outtab TABLE(I INTEGER) ) AS
BEGIN
   DECLARE grade int;
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
                RESIGNAL SET MESSAGE_TEXT = 'for the input parameter in_var = '||
                :in_var || ' exception was raised ';

   IF in_var < 60 THEN
      SIGNAL SQL_ERROR_CODE 20001 SET  MESSAGE_TEXT ='value is too small.';
   ELSEIF in_var <= 70 THEN
      grade = 4;
   ELSEIF in_var <= 80 THEN
      grade = 3;
   ELSEIF in_var <= 90 THEN
      grade = 2;
   ELSEIF in_var <= 100 THEN
      grade = 1;
   ELSE
      SIGNAL SQL_ERROR_CODE 20002 SET  MESSAGE_TEXT ='value is too big.';
   END IF;
   outtab = SELECT grade as I FROM dummy;
END;
```

The stored procedure then replaces the original SQL error message with this text:

```
Could not execute the procedure. user-defined error: [20001] user-defined error
exception: for the input parameter in_var = 0 exception was raised
(return status = -6)
```

You can still see the original error message with the `::SQL_ERROR_MESSAGE` system variable, which can be useful for adding additional information to the original text:

```
CREATE PROCEDURE MYPROC (IN in_var INTEGER, OUT outtab TABLE(I INTEGER) ) AS
BEGIN
   DECLARE grade int;
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
                RESIGNAL SET MESSAGE_TEXT = 'for the input parameter in_var = '||
                :in_var || ' exception was raised '
                || ::SQL_ERROR_MESSAGE;

   IF in_var < 60 THEN
      SIGNAL SQL_ERROR_CODE 20001 SET  MESSAGE_TEXT ='value is too small.';
   ELSEIF in_var <= 70 THEN
      grade = 4;
   ELSEIF in_var <= 80 THEN
      grade = 3;
   ELSEIF in_var <= 90 THEN
      grade = 2;
   ELSEIF in_var <= 100 THEN
      grade = 1;
   ELSE
      SIGNAL SQL_ERROR_CODE 20002 SET  MESSAGE_TEXT ='value is too big.';
   END IF;
   outtab = SELECT grade as I FROM dummy;
END;
```

10.4 Exception Handling Examples

This section describes a number of exception handling situations.

General Exception Handling

General exceptions can be handled with an exception handler declared at the beginning of statements that make an explicit signal exception. For example:

```
CREATE PROCEDURE MYPROC AS BEGIN
    DECLARE VAR1 INT;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
    VAR1 := 1;
    SIGNAL SQL_ERROR_CODE 20000 ;  -- raise error: 20000
    -- will not be reached
END;
```

Error Code Exception Handling

Declare an exception handler that catches exceptions with a specific error code number. For example:

```
CREATE PROCEDURE MYPROC AS BEGIN
    DECLARE VAR1 INT;
    DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 20000
SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
    VAR1 := 1;
    SIGNAL SQL_ERROR_CODE 20000 ;  -- raise error: 20000
    -- will not be reached
END;
```

This example from an industry data collection includes many characteristics of data to be measured: length, weight, outer diameter, and internal diameter. Each characteristic has its specification, including upper specification limitation (USL) and lower specification limitation (LSL). A worker may measure the data using meters, if the data is beyond the range of specification. This example throws an exception and drops it:

```
CREATE TABLE TB_CHARA(ID INT, NAME VARCHAR(50), USL DOUBLE, LSL DOUBLE);
INSERT INTO TB_CHARA VALUES(1, 'length', 9.65, 9.39);

CREATE PROCEDURE MYPROC (spec_id INT, data DOUBLE) AS
    BEGIN
        DECLARE usl, lsl DOUBLE;
        DECLARE EXIT HANDLER FOR SQL_ERROR_CODE 20005
    BEGIN
        SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE FROM DUMMY;
        SELECT :usl, :lsl, :data FROM DUMMY;
    END;
        SELECT USL, LSL INTO usl, lsl FROM TB_CHARA WHERE ID = spec_id;
        IF data < lsl OR data > usl THEN
           SIGNAL SQL_ERROR_CODE 20005;  --raise error: 20005
        SELECT 'NeverReached_noContinueOnErrorSemantics' FROM DUMMY;
    END IF;
END;
CALL MYPROC(1, 9.25);
```

Conditional Exception Handling

Declare exceptions using a `CONDITION` variable, which can be specified with an error code number:

```
CREATE PROCEDURE MYPROC AS BEGIN
    DECLARE var1 INT;
    DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 20000;
    DECLARE EXIT HANDLER FOR MYCOND SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
FROM DUMMY;
    var1 := 1;
    SIGNAL MYCOND ;  -- raise error: 20000
    -- will not be reached
END;
```

Signal an exception

Use the `SIGNAL` statement to explicitly raise an exception from within your procedures.

i Note

Use-defined error codes must at least 20000.

For example:

```
CREATE PROCEDURE MYPROC AS BEGIN
    DECLARE var1 INT;
    DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 20000;
    DECLARE EXIT HANDLER FOR MYCOND SELECT ::SQL_ERROR_CODE, ::SQL_ERROR_MESSAGE
FROM DUMMY;
    var1 := 1;
    SIGNAL MYCOND SET MESSAGE_TEXT = 'my error';  -- raise error: 20000
    -- will not be reached
END;
```

Resignal an exception

The RESIGNAL statement raises an exception on the action statement in the exception handler. If you do nto specify the error code, RESIGNAL throws the caught exception. For example:

```
CREATE PROCEDURE MYPROC AS BEGIN
    DECLARE var1 INT;
    DECLARE MYCOND CONDITION FOR SQL_ERROR_CODE 20000;
    DECLARE EXIT HANDLER FOR MYCOND RESIGNAL;
    var1 := 1;
    SIGNAL MYCOND SET MESSAGE_TEXT = 'my error';  -- raise error: 20000
    -- will not be reached
END;
```

Nested block exceptions.

You can declare exception handlers for nested blocks. For example:

```
CREATE TABLE TB_CHARA(ID INT, NAME VARCHAR(50), USL DOUBLE, LSL DOUBLE);
INSERT INTO TB_CHARA VALUES(1, 'length', 9.65, 9.39);

CREATE PROCEDURE MYPROC (spec_id INT, data DOUBLE) AS
    BEGIN
        DECLARE usl, lsl DOUBLE;
        DECLARE EXIT HANDLER FOR SQLEXCEPTION RESIGNAL SET MESSAGE_TEXT = 'level
1';
    BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION RESIGNAL SET MESSAGE_TEXT = 'level
2';
        SELECT USL, LSL INTO usl, lsl FROM TB_CHARA WHERE ID = spec_id;
        IF data < lsl OR data > usl THEN
        SIGNAL SQL_ERROR_CODE 20005 SET MESSAGE_TEXT =' error';
        END IF;
    END;
END;
CALL MYPROC(1, 9.25);
```

11 Troubleshooting

Table 7: Troubleshooting Error Messages

Error Number	Error Message
Error 154	`%S_MSG is not allowed in %S_MSG.`
Error 174	`The function '%.*s' requires %d arguments.`
Error 3774	`Can't drop with RESTRICT specification: %.*s.`
Error 16371	`invalid %S_MSG: %S_MSG cannot be more than %S_MSG or less than %S_MSG`
Error 16372	`numeric overflow: '%.*s'.`
Error 16373	`invalid %S_MSG: absolute value of %S_MSG must be less than %S_MSG minus %S_MSG`
Error 16374	`invalid %S_MSG: %S_MSG should not be greater than '%d' and %S_MSG minus %S_MSG`
Error 16375	`invalid %S_MSG: %S_MSG must be more than %ld`
Error 16376	`invalid %S_MSG: %S_MSG must be less than %ld`
Error 16378	`Invalid number: not a valid number string '%.*s'.`
Error 16379	`invalid %S_MSG: %S_MSG must be a non-zero integer`
Error 16380	`invalid %S_MSG: %S_MSG should not be greater than '%d' and %S_MSG minus %S_MSG`
Error 16390	`TOP operator conflict with the LIMIT clause.`
Error 16392	`Invalid trimchar length specified in the second parameter. The valid length is 1.`
Error 16395	`Table UDF error: for table UDF in sqlscript, a subquery or a table variable must be specified in return statement.`
Error 16399	`invalid %S_MSG: %S_MSG should not be greater than '%d' and %S_MSG minus %S_MSG`
Error 16870	`sequence is exhausted`

Error Number	Error Message
Error 16871	`CURRVAL of given sequence is not yet defined.`
Error 16874	`Could not execute the procedure. user-defined error: [%d] user-defined error exception: %.*s`
Error 16879	`INSERT and UPDATE are disallowed on the generated field: cannot insert into or update generated identity column field %.*s.`
Error 16880	`Cannot modify generated identity column: '%.*s'.`
Error 16881	`Cannot create temporary table having identity column: '%.*s'.`
Error 16882	`Cannot alter temporary table to add identity column: '%.*s'.`
Error 16883	`invalid sequence: more than one column in the SELECT list of the RESET BY query.`
Error 16884	`invalid sequence: type of given column in the RESET BY query must be numeric.`
Error 16885	`invalid sequence: RESET BY query has no result.`
Error 16886	`invalid sequence: RESET BY query returns null value.`
Error 16887	`invalid sequence: RESET BY query returns more than one row.`
Error 16888	`invalid sequence: RESET BY query is invalid.`
Error 16891	`Invalid column type for identity column '%.*s', it must be a numeric type.`
Error 16893	`Failed to get sequence value.`
Error 16897	`Invalid name of function or procedure: there is no output parameter name %s in user defined function.`
Error 16898	`Invalid name of function or procedure: This user defined function has multiple outputs, but this usage requires a single output. Specify a single output.`

For more information, see the *Troubleshooting: Error Messages* guide.

Important Disclaimers and Legal Information

www.sap.com/contactsap

THE BEST RUN SAP