English

Fujitsu Software BS2000

SESAM/SQL-Server

Reference Manual Part 1

User Guide

Valid for:
SESAM/SQL-Server V9.1

Edition June 2019

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to: bs2000.info@fujitsu.com

Certified documentation according to DIN EN ISO 9001:2015

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2015.

Copyright and Trademarks

Copyright © 2025 Fujitsu

All rights reserved.
Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers.

Table of Contents

SQL Reference Manual Part 1 14
1 Preface 15
1.1 Objectives and target groups of this manual 16
1.2 Summary of contents 17
1.3 Notational conventions 18
2 Embedding of SQL in programs 20
2.1 Program structure 21
2.2 Host variables 22
2.2.1 Defining host variables 23
2.2.2 Using host variables 24
2.2.3 Indicator variables 25
2.2.3.1 Defining indicator variables 26
2.2.3.2 Using indicator variables 27
2.3 Monitoring success and error handling 28
2.3.1 Monitoring success 29
2.3.2 Error handling 30
2.4 Cursor 31
2.4.1 Read-only cursors 32
2.4.2 Updatable cursors 33
2.4.3 Defining a cursor 34
2.4.4 Opening a cursor 35
2.4.5 Position cursor and read row 36
2.4.6 Updating or deleting a row 37
2.4.7 Storing a cursor 38
2.4.8 Close a cursor 39
2.4.9 Restore a cursor 40
2.4.10 Cursor examples 41
2.5 Dynamic SQL 43
2.5.1 Dynamic statement 44
2.5.1.1 Prepare a dynamic statement 45
2.5.1.2 Querying the data types of the placeholders and values 46
2.5.1.3 Execute a dynamic statement 47
2.5.2 Dynamic cursor descriptions 48
2.5.2.1 Preparing dynamic cursor descriptions 49
2.5.2.2 Determining the SQL data types of the placeholders 50
2.5.2.3 Determining the SQL data types of the derived columns 51
2.5.2.4 Evaluating dynamic cursor descriptions 52

2.5.2.5 Storing results 53
2.5.3 Descriptor area 54
2.5.3.1 Creating a descriptor area 55
2.5.3.2 Structure of a descriptor area 56
2.5.3.3 Descriptor area fields 57
2.5.3.4 Assigning values to the descriptor area 62
2.5.3.5 Querying the descriptor area 63
2.5.3.6 Using values from the descriptor area 64
2.5.3.7 Releasing the descriptor area 65
2.6 SQL statements in CALL DML transactions 66
2.6.1 Step-by-step conversion of CALL DML statements 67
2.6.2 Using User-Close and release session resources 68
2.6.3 Setting the isolation level 69
3 Lexical elements and names 70
3.1 SESAM/SQL character repertoire 71
3.2 Lexical units 72
3.2.1 Strings 73
3.2.2 Numerics 74
3.2.3 Delimiter symbols 75
3.2.4 Separators 76
3.2.5 Comments 77
3.3 Pragmas and annotations 78
3.3.1 AUTONOMOUS TRANSACTION pragma 81
3.3.2 DATA TYPE pragma 82
3.3.3 DEBUG ROUTINE pragma 83
3.3.4 DEBUG VALUE pragma 84
3.3.5 EXPLAIN pragma 86
3.3.6 ISOLATION LEVEL pragma 87
3.3.7 LIMIT ABORT_EXECUTION pragma 88
3.3.8 LOCK MODE pragma 89
3.3.9 LOOP LIMIT pragma 90
3.3.10 PREFETCH pragma 91
3.3.11 UTILITY MODE pragma 93
3.4 Names 94
3.4.1 Unqualified names 95
3.4.2 Qualified names 99
3.4.3 Defining names 102
4 Data types and values 103
4.1 Overview of data types and the associated value ranges 104
4.1.1 Data type groups 105
4.1.2 Range of values 106

4.1.3 Column 107
4.1.4 Parameters of routines and local variables 108
4.2 Data types 109
4.2.1 Overview of SQL data types 110
4.2.2 Alphanumeric and national data types 111
4.2.3 CHARACTER - String with a fixed length 112
4.2.4 CHARACTER VARYING - String with a variable length 113
4.2.5 NATIONAL CHARACTER - Strings with a fixed length 114
4.2.6 NATIONAL CHARACTER VARYING - Strings with a variable length 115
4.2.7 Numeric data types 116
4.2.8 SMALLINT - Small integer 117
4.2.9 INTEGER - Integers 118
4.2.10 NUMERIC - Fixed-point numbers 119
4.2.11 DECIMAL - Fixed-point numbers 120
4.2.12 REAL- Single-precision floating-point numbers 121
4.2.13 DOUBLE PRECISION - Double-precision floating-point numbers 122
4.2.14 FLOAT - Floating-point numbers 123
4.2.15 Time data types 124
4.2.16 DATE 125
4.2.17 TIME 126
4.2.18 TIMESTAMP 127
4.2.19 Compatibility between data types 128
4.3 Values 129
4.3.1 Literals 130
4.3.2 Specifying values 131
4.3.3 Values for multiple columns 132
4.3.4 NULL value 133
4.3.4.1 Keyword for the NULL value 134
4.3.4.2 NULL value in table columns 135
4.3.4.3 NULL value in functions, expressions and predicates 136
4.3.4.4 NULL value in GROUP BY 137
4.3.4.5 NULL value in ORDER BY 138
4.3.5 Strings 139
4.3.5.1 Alphanumeric literals 140
4.3.5.2 National literals 142
4.3.5.3 Special literals 145
4.3.5.4 Using strings 147
4.3.6 Numeric values 149
4.3.6.1 Numeric literals 150
4.3.6.2 Using numeric values 151
4.3.7 Time values 152

4.3.7.1 Time literals . . . 153
4.3.7.2 Using time values . . . 155
4.4 Assignment rules . . . 156
4.4.1 Entering values in table columns . . . 157
4.4.2 Default values for table columns . . . 158
4.4.3 Values for placeholders . . . 159
4.4.4 Reading values into host variables or a descriptor area . . . 160
4.4.5 Transferring values between host variables and a descriptor area . . . 161
4.4.6 Modifying the target data type by means of the CAST operator . . . 163
4.4.7 Supplying input parameters for routines . . . 164
4.4.8 Entering values in a procedure parameter (output) or local variable . . . 165
5 Compound language constructs . . . 166
5.1 Expression . . . 167
5.2 Function . . . 172
5.2.1 Time functions . . . 174
5.2.2 String functions . . . 175
5.2.3 Numeric functions . . . 177
5.2.4 Aggregate functions . . . 178
5.2.5 Table functions . . . 181
5.2.6 Cryptographic functions . . . 182
5.2.7 User Defined Functions (UDFs) . . . 184
5.2.8 Alphabetical reference section: Functions . . . 185
5.2.9 ABS() - Absolute value . . . 186
5.2.10 AVG() - Arithmetic average . . . 187
5.2.11 CEILING() - Smallest integer greater than the value . . . 189
5.2.12 CHAR_LENGTH() - Determine string length . . . 190
5.2.13 COLLATE() - Determine collation element for national strings . . . 192
5.2.14 COUNT(*) - Count table rows . . . 194
5.2.15 COUNT() - Count elements . . . 195
5.2.16 CSV() - Reading a BS2000 file as a table . . . 197
5.2.17 CURRENT_DATE - Current date . . . 201
5.2.18 CURRENT_TIME(3) - Current time . . . 202
5.2.19 CURRENT_TIMESTAMP(3) - Current time stamp . . . 203
5.2.20 DATE_OF_JULIAN_DAY() - Convert Julian day number . . . 204
5.2.21 DECRYPT() - Decrypt data . . . 205
5.2.22 DEE() - Table without columns . . . 208
5.2.23 ENCRYPT() - Encrypt data . . . 209
5.2.24 EXTRACT() - Extract components of a time value . . . 211
5.2.25 FLOOR() - Largest integer less than the value . . . 213
5.2.26 HEX_OF_VALUE() - Present any value in hexadecimal format . . . 214
5.2.27 JULIAN_DAY_OF_DATE() - Convert date . . . 217

5.2.28 LOCALTIME(3) - Current local time . . . 219
5.2.29 LOCALTIMESTAMP(3) - Current local time stamp . . . 220
5.2.30 LOWER() - Convert uppercase characters . . . 221
5.2.31 MAX() - Determine largest value . . . 222
5.2.32 MIN() - Determine lowest value . . . 224
5.2.33 MOD() - Remainder of an integer division (modulo) . . . 226
5.2.34 NORMALIZE() - Convert national string to normal form . . . 227
5.2.35 OCTET_LENGTH() - Determine string length . . . 229
5.2.36 POSITION() - Determine string position . . . 230
5.2.37 REP_OF_VALUE() - Present any value as a string . . . 231
5.2.38 SIGN() - Determine sign . . . 233
5.2.39 SUBSTRING() - Extract substring . . . 234
5.2.40 SUM() - Calculate sum . . . 237
5.2.41 TRANSLATE() - Transliterate / transcode string . . . 239
5.2.42 TRIM() - Remove characters . . . 242
5.2.43 TRUNC() - Remove decimal places . . . 244
5.2.44 UPPER() - Convert lowercase characters . . . 245
5.2.45 VALUE_OF_HEX() - Present hexadecimal format as a value . . . 246
5.2.46 VALUE_OF_REP() - Present a string as a value . . . 248
5.3 Predicates . . . 250
5.3.1 Comparison of two rows . . . 252
5.3.1.1 Comparison rules . . . 254
5.3.2 Quantified comparison (comparison with the rows of a table) . . . 257
5.3.3 BETWEEN predicate (range query) . . . 259
5.3.4 CASTABLE predicate (convertibility check) . . . 261
5.3.5 IN predicate (elementary query) . . . 262
5.3.6 LIKE predicate (simple pattern comparison) . . . 265
5.3.7 LIKE_REGEX predicate (pattern comparison with regular expressions) . . . 268
5.3.8 NULL predicate (comparison with the NULL value) . . . 274
5.3.9 EXISTS predicate (existence query) . . . 276
5.4 Search conditions . . . 277
5.5 CASE expression . . . 280
5.5.1 CASE expression with search condition . . . 282
5.5.2 Simple CASE expression . . . 284
5.5.3 CASE expression with NULLIF . . . 286
5.5.4 CASE expression with COALESCE . . . 287
5.5.5 CASE expression with MIN / MAX . . . 289
5.6 CAST expression . . . 291
5.7 Integrity constraint . . . 296
5.7.1 Column constraints . . . 298
5.7.2 Table constraints . . . 301

5.8 Column definitions . . . 304
6 Query expression . . . 308
6.1 Table specifications . . . 310
6.2 SELECT expression . . . 313
6.2.1 SELECT list - Select derived columns . . . 314
6.2.2 SELECT...FROM - Specify table . . . 317
6.2.3 SELECT...WHERE - Select derived columns . . . 319
6.2.4 SELECT...GROUP BY - Group derived rows . . . 321
6.2.5 SELECT...HAVING - Select groups . . . 323
6.3 TABLE - Table query . . . 324
6.4 Joins . . . 325
6.4.1 Join expression . . . 326
6.4.2 Joins without join expression . . . 328
6.4.3 Join types . . . 329
6.4.3.1 Cross joins . . . 330
6.4.3.2 Inner joins . . . 332
6.4.3.3 Outer joins . . . 334
6.4.3.4 Union joins . . . 335
6.4.3.5 Compound joins . . . 336
6.5 Subquery . . . 339
6.5.1 Correlated subqueries . . . 340
6.6 Combining query expressions with UNION . . . 342
6.7 Combining query expressions with EXCEPT . . . 345
6.8 Updatability of query expressions . . . 347
6.8.1 Rules for updatable query expressions . . . 348
6.8.2 Updatable view . . . 349
6.8.3 Update via cursor . . . 350
7 Routines . . . 351
7.1 Procedures (Stored Procedures) . . . 353
7.1.1 Creating a procedure . . . 354
7.1.2 Execute a procedure . . . 356
7.1.3 Delete a procedure . . . 357
7.1.4 Examples of procedures . . . 358
7.2 User Defined Functions (UDFs) . . . 362
7.2.1 Creating a UDF . . . 363
7.2.2 Executing a UDF . . . 365
7.2.3 Deleting a UDF . . . 366
7.2.4 Uncorrelated function calls . . . 367
7.2.5 Examples of UDFs . . . 369
7.3 EXECUTE privilege for routines . . . 370
7.4 Information on routines . . . 371

7.5 Pragmas in routines 372
7.6 Control statements in routines 374
7.7 COMPOUND statement in routines 375
7.8 Diagnostic information in routines 376
8 SQL statements 382
8.1 Summary of contents 383
8.1.1 SQL statements for schema definition and administration 384
8.1.2 SQL statements for querying and updating data 386
8.1.3 SQL statements for transaction management 387
8.1.4 SQL statements for session control 388
8.1.5 SQL statements for dynamic SQL 389
8.1.6 WHENEVER statement for ESQL error handling 390
8.1.7 SQL statements for managing the storage structure 391
8.1.8 SQL statements for managing user entries 392
8.1.9 Utility statements 393
8.1.10 Control statements 394
8.1.11 Diagnostic statements 395
8.2 Descriptions in alphabetical order 396
8.2.1 Description format 397
8.2.2 SQL statements in routines 398
8.2.3 SQL statement descriptions 402
8.2.3.1 ALLOCATE DESCRIPTOR - Request SQL descriptor area 404
8.2.3.2 ALTER SPACE - Modify space parameters 406
8.2.3.3 ALTER STOGROUP - Modify storage group 408
8.2.3.4 ALTER TABLE - Modify base table 410
8.2.3.5 CALL - Execute procedure 424
8.2.3.6 CASE - Execute SQL statements conditionally 427
8.2.3.7 CLOSE - Close cursor 430
8.2.3.8 COMMIT WORK - Terminate transaction 431
8.2.3.9 COMPOUND - Execute SQL statements in a common context 433
8.2.3.10 CREATE FUNCTION - Create User Defined Function (UDF) 441
8.2.3.11 CREATE INDEX - Create index 444
8.2.3.12 CREATE PROCEDURE - Create procedure 447
8.2.3.13 CREATE SCHEMA - Create schema 450
8.2.3.14 CREATE SPACE - Create space 453
8.2.3.15 CREATE STOGROUP - Create storage group 456
8.2.3.16 CREATE SYSTEM_USER - Create system entry 458
8.2.3.17 CREATE TABLE - Create base table 461
8.2.3.18 CREATE USER - Create authorization identifier 469
8.2.3.19 CREATE VIEW - Create view 470
8.2.3.20 DEALLOCATE DESCRIPTOR - Release SQL descriptor area 474

8.2.3.21 DECLARE CURSOR - Declare cursor 475
8.2.3.22 DELETE - Delete rows 481
8.2.3.23 DESCRIBE - Query data type of input and output values 484
8.2.3.24 DROP FUNCTION - Delete User Defined Function (UDF) 486
8.2.3.25 DROP INDEX - Delete index 487
8.2.3.26 DROP PROCEDURE - Delete procedure 488
8.2.3.27 DROP SCHEMA - Delete schema 489
8.2.3.28 DROP SPACE - Delete space 490
8.2.3.29 DROP STOGROUP - Delete storage group 492
8.2.3.30 DROP SYSTEM_USER - Delete system entry 493
8.2.3.31 DROP TABLE - Delete base table 496
8.2.3.32 DROP USER - Delete authorization identifier 498
8.2.3.33 DROP VIEW - Delete view 499
8.2.3.34 EXECUTE - Execute prepared statement 500
8.2.3.35 EXECUTE IMMEDIATE - Execute dynamic statement 504
8.2.3.36 FETCH - Position cursor and read row 507
8.2.3.37 FOR - Execute SQL statements in a loop 512
8.2.3.38 GET DESCRIPTOR - Read SQL descriptor area 515
8.2.3.39 GET DIAGNOSTICS - Output diagnostic information 518
8.2.3.40 GRANT - Grant privileges 521
8.2.3.41 IF - Execute SQL statements conditionally 527
8.2.3.42 INCLUDE - Insert program text into ESQL programs 529
8.2.3.43 INSERT - Insert rows in table 530
8.2.3.44 ITERATE - Switch to the next loop pass 536
8.2.3.45 LEAVE - Terminate a loop or COMPOUND statement 537
8.2.3.46 LOOP - Execute SQL statements in a loop 538
8.2.3.47 MERGE - Insert rows in a table or update column values 540
8.2.3.48 OPEN - Open cursor 545
8.2.3.49 PERMIT - Specify user identification for SESAM/SQL V1.x 547
8.2.3.50 PREPARE - Prepare dynamic statement 548
8.2.3.51 REORG STATISTICS - Regenerate global statistics 557
8.2.3.52 REPEAT - Execute SQL statements in a loop 558
8.2.3.53 RESIGNAL - Report exception in local exception routine 560
8.2.3.54 RESTORE - Restore cursor 562
8.2.3.55 RETURN - Supply the return value of a User Defined Function (UDF) . . 563
8.2.3.56 REVOKE - Revoke privileges 564
8.2.3.57 ROLLBACK WORK - Roll back transaction 571
8.2.3.58 SELECT - Read individual rows 572
8.2.3.59 SET - Assign value 575
8.2.3.60 SET CATALOG - Set default database name 576
8.2.3.61 SET DESCRIPTOR - Update SQL descriptor area 577

8.2.3.62 SET SCHEMA - Specify default schema name 582
8.2.3.63 SET SESSION AUTHORIZATION - Set authorization identifier 584
8.2.3.64 SET TRANSACTION - Define transaction attributes 586
8.2.3.65 SIGNAL - Report exception in routine 590
8.2.3.66 STORE - Save cursor position 592
8.2.3.67 UPDATE - Update column values 593
8.2.3.68 WHENEVER - Define error handling 598
8.2.3.69 WHILE - Execute SQL statements in a loop 600
9 SESAM-CLI 602
9.1 Concept of the SESAM CLI 603
9.1.1 Structure of SESAM CLI calls 604
9.1.2 Statements that initiate transactions in CLI calls 608
9.2 SESAM CLI calls 609
9.2.1 Overview 610
9.2.2 Alphabetical reference section 612
9.2.3 SQL_BLOB_CLS_ISBTAB - SQLbcis 613
9.2.4 SQL_BLOB_CLS_REF - SQLbcre 615
9.2.5 SQL_BLOB_OBJ_CLONE - SQLbocl 616
9.2.6 SQL_BLOB_OBJ_CREATE - SQLbocr 617
9.2.7 SQL_BLOB_OBJ_CREAT2 - SQLboc2 618
9.2.8 SQL_BLOB_OBJ_DROP - SQLbodr 620
9.2.9 SQL_BLOB_TAG_GET - SQLbtge 621
9.2.10 SQL_BLOB_TAG_PUT - SQLbtpu 623
9.2.11 SQL_BLOB_VAL_CLOSE - SQLbvcl 625
9.2.12 SQL_BLOB_VAL_FETCH - SQLbvfe 626
9.2.13 SQL_BLOB_VAL_GET - SQLbvge 628
9.2.14 SQL_BLOB_VAL_LEN - SQLbvle 630
9.2.15 SQL_BLOB_VAL_OPEN - SQLbvop 631
9.2.16 SQL_BLOB_VAL_PUT - SQLbvpu 633
9.2.17 SQL_BLOB_VAL_STOW - SQLbvst 634
9.2.18 SQL_DIAG_SEQ_GET - SQLdsg 636
10 Information schemas 638
10.1 Views of the INFORMATION_SCHEMA 639
10.1.1 BASE_TABLES 641
10.1.2 BASE_TABLE_COLUMNS 642
10.1.3 CATALOG_PRIVILEGES 646
10.1.4 CHARACTER_SETS 647
10.1.5 CHECK_CONSTRAINTS 648
10.1.6 COLLATIONS 649
10.1.7 COLUMNS 650
10.1.8 COLUMN_PRIVILEGES 654

10.1.9 CONSTRAINT_COLUMN_USAGE 655
10.1.10 CONSTRAINT_TABLE_USAGE 656
10.1.11 DA_LOGS 657
10.1.12 INDEXES 658
10.1.13 INDEX_COLUMN_USAGE 659
10.1.14 KEY_COLUMN_USAGE 660
10.1.15 MEDIA_DESCRIPTIONS 661
10.1.16 MEDIA_RECORDS 662
10.1.17 PARAMETERS 663
10.1.18 PARTITIONS 666
10.1.19 RECOVERY_UNITS 667
10.1.20 REFERENTIAL_CONSTRAINTS 669
10.1.21 ROUTINES 670
10.1.22 ROUTINE_COLUMN_USAGE 674
10.1.23 ROUTINE_PRIVILEGES 675
10.1.24 ROUTINE_ROUTINE_USAGE 676
10.1.25 ROUTINE_TABLE_USAGE 677
10.1.26 SCHEMATA 678
10.1.27 SPACES 679
10.1.28 SQL_FEATURES 680
10.1.29 SQL_IMPL_INFO 681
10.1.30 SQL_LANGUAGES_S 682
10.1.31 SQL_SIZING 683
10.1.32 STOGROUPS 684
10.1.33 STOGROUP_VOLUME_USAGE 685
10.1.34 SYSTEM_ENTRIES 686
10.1.35 TABLES 687
10.1.36 TABLE_CONSTRAINTS 688
10.1.37 TABLE_PRIVILEGES 689
10.1.38 TRANSLATIONS 690
10.1.39 USAGE_PRIVILEGES 691
10.1.40 USERS 692
10.1.41 VIEWS 693
10.1.42 VIEW_COLUMN_USAGE 694
10.1.43 VIEW_ROUTINE_USAGE 695
10.1.44 VIEW_TABLE_USAGE 696
10.2 Views of the SYS_INFO_SCHEMA 697
10.2.1 SYS_CATALOGS 699
10.2.2 SYS_CHECK_CONSTRAINTS 700
10.2.3 SYS_CHECK_USAGE 701
10.2.4 SYS_COLUMNS 702

10.2.5 SYS_DA_LOGS 705
10.2.6 SYS_DBC_ENTRIES 706
10.2.7 SYS_DML_RESOURCES 708
10.2.8 SYS_ENVIRONMENT 709
10.2.9 SYS_INDEXES 710
10.2.10 SYS_LOCK_CONFLICTS 712
10.2.11 SYS_MEDIA_DESCRIPTIONS 716
10.2.12 SYS_PARAMETERS 717
10.2.13 SYS_PARTITIONS 719
10.2.14 SYS_PRIVILEGES 720
10.2.15 SYS_RECOVERY_UNITS 721
10.2.16 SYS_REFERENTIAL_CONSTRAINTS 724
10.2.17 SYS_ROUTINES 725
10.2.18 SYS_ROUTINE_ERRORS 727
10.2.19 SYS_ROUTINE_PRIVILEGES 729
10.2.20 SYS_ROUTINE_ROUTINE_USAGE 730
10.2.21 SYS_ROUTINE_USAGE 731
10.2.22 SYS_SCHEMATA 732
10.2.23 SYS_SPACES 733
10.2.24 SYS_SPACE_PROPERTIES 734
10.2.25 SYS_SPECIAL_PRIVILEGES 737
10.2.26 SYS_STOGROUPS 738
10.2.27 SYS_SYSTEM_ENTRIES 739
10.2.28 SYS_TABLES 740
10.2.29 SYS_TABLE_CONSTRAINTS 742
10.2.30 SYS_UNIQUE_CONSTRAINTS 743
10.2.31 SYS_USAGE_PRIVILEGES 744
10.2.32 SYS_USERS 745
10.2.33 SYS_VIEW_USAGE 746
10.2.34 SYS_VIEW_ROUTINE_USAGE 747
11 Appendix 748
11.1 Syntax elements of SESAM/SQL 749
11.2 Syntax overview of the CSV file 761
11.3 SQL keywords 763
12 Related publications 773

SQL Reference Manual Part 1

1 Preface

The functions and architectural features of the SESAM/SQL-Server database system meet all the demands placed on a powerful database server in today's world. These characteristics are reflected in its name: SESAM/SQL-Server.

SESAM/SQL-Server is available in a standard edition for single-task operation and in an enterprise edition for multitask operation.

For the sake of simplicity, we shall use the name SESAM/SQL throughout this manual to refer to SESAM/SQL-Server.

The following introductory descriptions are contained centrally in the " Core manual":

- Brief product description
- Structure of the SESAM/SQL server documentation
- Demonstration database
- Readme file
- Changes since the last editions of the manuals

1.1 Objectives and target groups of this manual

This manual is intended for all SESAM/SQL users working with SQL.

It is assumed that you are already familiar with the “ Core manual”, in particular with the SESAM/SQL objects and concepts upon which SQL statements are based. It is also assumed that you have a basic knowledge of relational databases.

If you want to call SQL statements interactively via the utility monitor, you must be familiar with the utility monitor (see the “ Utility Monitor” manual).

If you plan on embedding SQL statements in a program, you must be familiar with the COBOL programming language and the ESQL precompiler (see the “ ESQL-COBOL for SESAM/SQL-Server” manual.)

1.2 Summary of contents

This manual contains a complete description of the SQL database language as used in the database system SESAM/SQL. Specific reference is made to any differences to or extensions of the SQL standard.

The chapter “Embedding of SQL in programs” describes SQL-specific concepts for using SQL statements in a host language (COBOL). The remaining chapters describe SQL language constructs in logical sequence. In each chapter, it is assumed that you are familiar with the language constructs dealt with in the previous chapters and are not described again.

The chapter “SQL statements” includes an alphabetical reference section containing all the SQL statements.

The chapter “SESAM-CLI” describes the structure of the SESAM-CLI interface. This interface is used to create and edit BLOB objects. It also includes an alphabetical reference section which explains the individual CLI calls in detail.

The chapter “Information schemas” describes the views of the INFORMATION_SCHEMA and SYS_INFO_SCHEMA schemas.

The appendix is an alphabetical reference section for the syntaxes used and reserved keywords of SESAM/SQL.

A list of references and an index is provided at the end of the manual.

The manual contains a large number of examples. These refer in each case to the content of the preceding description. Some of the examples for SQL language constructs, particularly those for expressions and query expressions, run only in a superordinate statement and are not executable independently.

1.3 Notational conventions

The following notational conventions are used in this manual:

	Syntax definitions
UPPERCASE	SQL keywords
underscored	Default values
bold	Used for emphasis in running text
italics	Variables in syntax definitions and running text
`Fixed-space font`	Program text in syntax definitions and examples

<table>
<tr><td><code>::=</code></td><td>Definition character
The specification to the right of <code>::=</code> defines the syntax of the element on the left.</td></tr>
<tr><td>|</td><td>In unqualified syntax definitions this character separates the alternative specifications.</td></tr>
<tr><td>[ ]</td><td>May be omitted
The brackets are metacharacters and must not be entered in an SQL statement.</td></tr>
<tr><td>{ | }</td><td>Alternative specifications in syntax definitions (on a single line).
The braces are metacharacters and must not be entered in an SQL statement.</td></tr>
<tr><td>{ |
|
}</td><td>Alternative specifications in syntax definitions (over several lines). Each line contains one alternative.
The braces are metacharacters and must not be entered in an SQL statement.</td></tr>
<tr><td>,...</td><td>In syntax definitions, a comma followed by three dots means that you can repeat the preceding specification any number of times, separating each specification with a comma. If you do not repeat a specification, you must omit the comma.</td></tr>
<tr><td>...</td><td>In syntax definitions, an ellipsis means that you can repeat the preceding specification any number of times. In examples, the ellipsis means that the rest of the statement is of no significance to the example.
The ellipsis is a metacharacter and must not be entered in an SQL statement.</td></tr>
</table>

i Indicates notes that are of particular importance.

! Indicates warnings.

The strings `<date>`, `<time>` and `<ver>` in examples indicate the current displays for date, time and version when the examples are otherwise independent of date, time and version.

2 Embedding of SQL in programs

Programming language-specific interfaces that allow you to incorporate SQL statements in a program are available, thus allowing you to access a database from a program. SESAM/SQL provides an interface for the programming language COBOL.

The concepts involved in embedding SQL statements in a program are the same for all programming languages and are referred to as ESQL (Embedded SQL). Programs that include embedded SQL statements are called ESQL programs.

This chapter explains the concepts involved in embedding SQL statements in a program. It covers the following topics:

- Program structure
- Host variables
- Monitoring success and error handling
- cursor
- Dynamic SQL

You will find language-specific details in the " ESQL-COBOL for SESAM/SQL-Server" manual.

2.1 Program structure

An ESQL program consists of program text in the relevant programming language, also referred to as the host language, and SQL statements. SQL statements may be included wherever host language statements are permitted. The beginning and end of an SQL statement are marked so that they can be distinguished from the statements in the host language. The way in which the statements are marked depends on the programming language involved.

If host language variables (host variables) are used in the SQL statements, the program includes additional sections (DECLARE SECTION) in which these variables are defined. DECLARE SECTIONs may be included wherever variable definitions in the host language are allowed. The beginning and end of a DECLARE SECTION are marked by EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION respectively (the exact syntax is language-specific and is described in the “ ESQL-COBOL for SESAM/SQL-Server” manual. An ESQL program may include any number of DECLARE SECTIONs.

ESQL COBOL programs with executable examples of database statements can be found in the demonstration database of SESAM/SQL (see the “ Core manual”).

2.2 Host variables

A host variable is a host language variable that can be used in an embedded SQL statement. A host variable is used to transfer values from the database to the program in the host language for further processing or to transfer data to the database and provide values required for certain calculations.

2.2.1 Defining host variables

A host variable must be defined in the program in a DECLARE SECTION in accordance with programming language conventions. The location of the definition and use of a host variable must satisfy the following conditions:

- In the program text, a variable must be defined before it is used in an SQL statement.
- The definition must be valid, with regard to programming language conventions, for any use to which the variable may be put in an SQL or host language statement.
- The definition of a variable that is used in a DECLARE CURSOR statement defining a cursor must be valid for all OPEN statements of the defined cursor.

The data type of the host variable depends on the data type of the SESAM/SQL values for which this host variable is to be used. The ESQL language interface provides predefined data types that must be used for host variables. The assigned COBOL data type is specified for each SESAM/SQL data type in the “ ESQL-COBOL for SESAM /SQL-Server” manual.

2.2.2 Using host variables

In SQL statements that query data in the database, the values read can be stored in host variables.

In SQL statements that insert values into the database, update values in the database or in which calculations are performed (functions, expressions, predicates, search conditions), the values can be made available via host variables.

Other instances in which values in SQL statements can or must be provided via host variables are described in the chapter "SQL statements" as part of the description of the individual SQL statements.

A host variable is preceded in an SQL statement by a colon:
: *host-variable*

Host variables can also be vectors containing several values of the same data type. This allows you to assign aggregates to multiple columns or to transfer aggregates from multiple columns to a host variable. The syntax for vectors is language-specific and is described in the " ESQL-COBOL for SESAM/SQL-Server" manual.

2.2.3 Indicator variables

A host variable can be combined with another host variable known as an indicator variable. An indicator variable is used to express the NULL value, which does not exist in programming languages, and to monitor the transfer of alphanumeric and national values from the database.

2.2.3.1 Defining indicator variables

When you define a host variable that you want to use as an indicator variable, you must assign it the host language data type that corresponds to the SQL data type SMALLINT. The exact data type is specified in the “ ESQL-COBOL for SESAM/SQL-Server” manual.

2.2.3.2 Using indicator variables

A host variable can only be combined with an indicator variable for the purpose of querying data in the database, inserting values in the database, updating values in the database or for use in calculations (functions, expressions, predicates, search conditions).

You specify an indicator variable after the host variable. They may be separated by the keyword INDICATOR, although this is not necessary:
`:`*host-variable* `[INDICATOR] :`*indicator-variable*

If the host variable is a vector, the associated indicator variable must also be a vector with the same number of elements. Each element in the host variable is assigned the corresponding element in the indicator variable. The syntax for vectors is language-specific and is described in thel " ESQL-COBOL for SESAM/SQL-Server" manual.

Querying values

SESAM/SQL assigns one of the following values to the indicator variable when you query a value in the database and subsequently assign it to a host variable:

0	The host variable contains the value read. The assignment was error free.
-1	The value to be assigned is the NULL value.
> 0	For alphanumeric or national values: The host variable was assigned a truncated string. The value of the indicator variable indicates the original length in code units.

Inserting or updating values

If you specify values in SQL statements via host variables, you can use the indicator variable to specify a NULL value. To do this, you must assign the indicator variable a negative value before the SQL statement is called. When the SQL statement is executed, the NULL value is used instead of the value of the host variable.

2.3 Monitoring success and error handling

Once an SQL statement has been executed, the ESQL program should check whether execution was successful so that appropriate action can be taken in the event of an error.

2.3.1 Monitoring success

Use the host variable SQLSTATE, which SESAM/SQL supports in the ESQL interface, to check whether a statement was successful.

You must define SQLSTATE in your program in a DECLARE SECTION with the SQL data type CHAR(5). This definition must be located before the first SQL statement in the program text and must be valid, with regard to programming language conventions, for all the statements that use it.

After an SQL statement has been executed, SQLSTATE is assigned an SQL status code. The possible values for SQLSTATE are described in the “ Messages” manual.

For reasons of compatibility with SESAM/SQL V1.x, the host variable SQLCODE for monitoring the success of SQL statements is supported. You should not, however, use this host variable in new applications.

2.3.2 Error handling

There are two ways of taking appropriate action if an SQL statement was unsuccessful:

- Query SQLSTATE and branch according to the status code
- Use the WHENEVER statement

  You can use WHENEVER to specify that, after execution of an SQL statement with an SQLSTATE '00*xxx*' and '01*xxx*', the program is to continue executing or is to branch to a certain part of the program where error handling is performed. You can specify branching within the program for two error classes:

  - NOT FOUND: no data available, e.g. when the end of a table is reached
  - SQLERROR: other errors that result in abortion of SQL statements

  You can specify the WHENEVER statement more than once in a program. The specifications made in a WHENEVER statement are valid for all subsequent SQL statements in the program text up to the next WHENEVER statement for the same error class.

2.4 Cursor

Because many programming languages do not provide an equivalent of the type “table”, the concept of the cursor is used when SQL statements are embedded in programs. A cursor enables you to process the rows of a table individually one after the other.

A cursor is assigned to a table referred to as the cursor table. This table is the derived table of the query expression that defined the cursor.

There are a number of SQL statements that can be used with cursors:

DECLARE CURSOR	Declare a cursor
OPEN	Opening a cursor
CLOSE	Close a cursor
FETCH	Position cursor and read row
DELETE ... WHERE CURRENT OF ...	Delete current row
UPDATE ... WHERE CURRENT OF ...	Update current row
STORE	Save cursor position
RESTORE	Restore cursor position

A cursor must be defined, be opened before it is used, and be closed after it has been used. The SQL statements must be used in a predefined order.

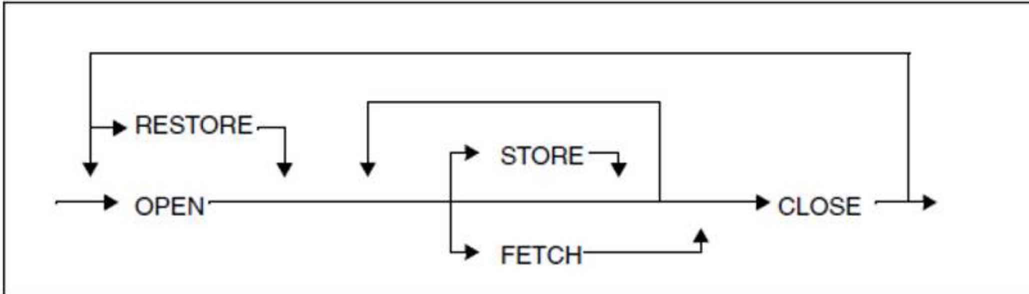
There are two types of cursors: cursors that can be updated (updatable cursor) and cursors that cannot be updated.

> In routines, local cursors which can **only** be addressed within the COMPOUND statement are defined with the DECLARE CURSOR statement, see section “Local cursors”.
>
> A local cursor differs from a normal cursor only in its limited area of validity.

2.4.1 Read-only cursors

A cursor that cannot be updated can only be used for reading rows from the cursor table and is therefore referred to as a read-only cursor.

The diagram below indicates the SQL statements that can be used for a non-dynamic readonly cursor and the order in which they are used:



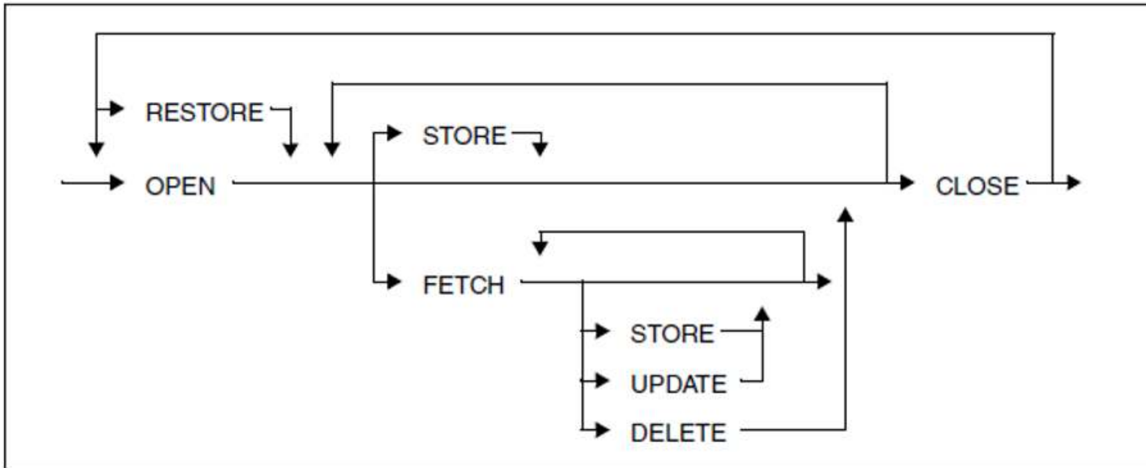
RESTORE can only be used to open a cursor after a STORE statement. If a cursor position has been stored, FETCH cannot be used.

Other statements that can be used with dynamic cursors are described in the section “Dynamic cursor descriptions”.

2.4.2 Updatable cursors

An updatable cursor can be used to delete or update rows in a table in addition to reading rows.

The diagram below indicates the SQL statements that can be used for a non-dynamic updatable cursor and the order in which they are used:



RESTORE can only be used to open a cursor after a STORE statement. If a cursor position has been stored, FETCH cannot be used.

Other statements that can be used with dynamic cursors are described in the section “Dynamic cursor descriptions”.

2.4.3 Defining a cursor

A cursor is defined with a DECLARE CURSOR statement. During definition, the cursor is assigned a cursor description. The cursor description is the query expression that defines the cursor table.

The query expression is specified directly in the DECLARE CURSOR statement for static cursors and local cursors (in routines). In the case of dynamic cursors, it is created when the program is executed (see section “Dynamic cursor descriptions”).

The following characteristics of the cursor can be specified in the definition:

Positioning

There are two kinds of cursors: scrollable cursors and sequential cursors.

A scrollable cursor can be positioned freely on any row in the cursor table. It is defined by specifying the keyword SCROLL.

A cursor defined with NO SCROLL can only be positioned on the next row in the cursor table.

Lifetime

If a cursor is to remain open after the end of a transaction, this can be specified using the WITH HOLD clause. The only prerequisite is that the cursor must be open prior to completion of the transaction. The WITH HOLD clause is not permitted for local cursors (in routines).

A cursor defined with WITHOUT HOLD is closed implicitly once the transaction has completed. WITHOUT HOLD is the default value.

Sorting

An ORDER BY clause can be specified in the cursor description indicating that the rows in the cursor table are to be sorted.

Number of hits

A FETCH FIRST *max* ROWS ONLY clause for limiting the number of hits supplied can only be specified in the cursor description.

Updatability

A cursor is updatable if the query expression used to define the cursor is updatable (see section “Updatability of query expressions”), and neither SCROLL nor ORDER BY nor the FOR READ ONLY clause was specified in the cursor declaration.

An updatable cursor references exactly one base table. Individual rows in this table can be deleted or updated using the cursor position to indicate the appropriate row. The FOR UPDATE clause in the cursor description can be used for updatable cursors to specify the columns whose values can be updated.

If a cursor is not updatable, it can only be used to read rows from the relevant cursor table. A cursor cannot be updated in the case of `FETCH FIRST` *max* `ROWS ONLY`, either.

2.4.4 Opening a cursor

A cursor must be opened before it can be used.

The OPEN statement is used to open a cursor. The values for host variables in the cursor description and for special literals (see "Special literals") and time functions (CURRENT_DATE, CURRENT_TIME(3), CURRENT_TIMESTAMP(3), etc.) are determined. After a cursor is opened, it is positioned before the first row of the corresponding cursor table (see section “OPEN - Open cursor”).

2.4.5 Position cursor and read row

If you want to read a row in the cursor table, you must position the cursor on this row with FETCH. The column values of the current row are fetched into host variables or into a descriptor area (see section “Descriptor area”).

In order to read the next row, the cursor must be repositioned. A cursor declared with SCROLL can be positioned freely. A cursor defined without SCROLL or with NO SCROLL can only be positioned on the next row.

2.4.6 Updating or deleting a row

If you are using an updatable cursor, you can update or delete a row in the base table upon which the cursor description is based after you have positioned the cursor. To do this, use the UPDATE...WHERE CURRENT OF or DELETE...WHERE CURRENT OF statement.

The update or delete operation refers to the row in the cursor table on which the cursor is currently positioned. The position of the cursor is not changed by an update operation. After a delete operation, the cursor is positioned on the next row in the cursor table (or after the last row, if the end of the table has been reached. You must reposition the cursor with FETCH before you can perform another update or delete operation.

2.4.7 Storing a cursor

If you want to retain the cursor table and the cursor position beyond the end of the current transaction, you can save the cursor with the STORE statement. Please note, however, that between STORE and the subsequent closure of the cursor, the cursor table can no longer be read with FETCH. STORE is not permitted for local cursors (in routines).

Another simpler option for keeping a cursor open across several transactions is to use the WITH HOLD clause in the cursor definition. The WITH HOLD clause is not permitted for local cursors (in routines).

2.4.8 Close a cursor

You close a cursor with the CLOSE statement.

In addition, a cursor is closed when the transaction in which the cursor was opened is terminated. However, this does not apply if the cursor was specified with WITH HOLD and the transaction is not reset.

2.4.9 Restore a cursor

A cursor saved with STORE can be restored with the RESTORE statement. The cursor is opened and the cursor table can again be accessed. RESTORE is not permitted for local cursors (in routines).

The information that has been stored can be lost under certain circumstances. These circumstances are described in the section “RESTORE - Restore cursor”.

2.4.10 Cursor examples

Example of a cursor with ORDER BY

The cursor CUR_CONTACTS defines a section of the CONTACTS table containing the last name, first name and department for all customers with customer numbers greater than 103. The rows are to be sorted in ascending sequence by department and, within the departments in descending sequence by last name.

```
DECLARE cur_contacts CURSOR FOR
SELECT lname, fname, department
FROM contacts WHERE cust_num > 103
ORDER BY department ASC, lname DESC
```

The cursor is opened with the OPEN statement

```
OPEN cur_contacts
```

At this point, the cursor table includes the following rows:

lname	fname	department
Buschmann	Anke	
Bauer	Xaver	
Heinlein	Robert	Purchasing
Davis	Mary	Purchasing

Null values are shown in the table above as empty fields. When rows are sorted using ORDER BY in SESAM/SQL, null values are regarded as being less than any non-null value.

In an ESQL program, the cursor table can be read row by row in a loop. The column values are passed to the host variables NAME, FIRSTNAME and DEPT.

```
FETCH cur_contacts INTO :LNAME,
:FIRSTNAME INDICATOR :IND_FIRSTNAME,
:DEPT INDICATOR :IND_DEPT
```

Example of SQL data manipulation using a cursor

Use the cursor CUR_VAT to select all services for which no VAT is calculated. It is specified with WITH HOLD so that it remains open even after a COMMIT WORK provided that it was open at the end of the transaction:

```
DECLARE CUR_VAT CURSOR WITH HOLD FOR
SELECT service_num, service_text, vat
FROM service WHERE vat=0.00


OPEN cur_vat
```

The following cursor table is produced when the cursor is opened:

service_num	service_text	vat
4	Systems analysis	0.00
5	Database design	0.00
10	Travel expenses	0.00

A VAT rate of 15% is to be charged for these services. A sequence of FETCH and UPDATE statements allows the rows of the SERVICE table to be updated. FETCH NEXT positions the cursor on the next row.

```
FETCH NEXT cur_vat INTO :SERVICE_NUM,
:SERVICE_TEXT INDICATOR :IND_SERVICE_TEXT
:VAT INDICATOR :IND_VAT

UPDATE service SET vat=0.15 WHERE CURRENT OF cur_vat
```

The cursor is then positioned on the second row of the cursor table:

```
FETCH NEXT cur_vat INTO :SERVICE_NUM,
:SERVICE_TEXT INDICATOR :IND_SERVICE_TEXT
:VAT INDICATOR :IND_VAT

UPDATE service SET vat=0.15 WHERE CURRENT OF cur_vat
```

The transaction is closed with COMMIT WORK. Because of the WITH HOLD clause, the cursor can be positioned on the third row of the cursor table by issuing a FETCH statement immediately after COMMIT WORK.

2.5 Dynamic SQL

SESAM/SQL allows you to generate SQL statements and cursor descriptions dynamically during execution of an ESQL program. The concepts and language resources involved in this are referred to by the term dynamic SQL and are described in this section.

A dynamic statement (or cursor description) does not have to be known when a program is compiled. Instead, it can be constructed dynamically when the program is executed and is made available in a host variable.

A routine (see chapter “Routines”) may not contain any dynamic SQL statements or cursor descriptions.

Placeholder

You cannot use host variables in a dynamic SQL statement (or cursor description). Instead, you use question marks as placeholders for unknown input values. The rules governing placeholders are described in the “PREPARE - Prepare dynamic statement”.

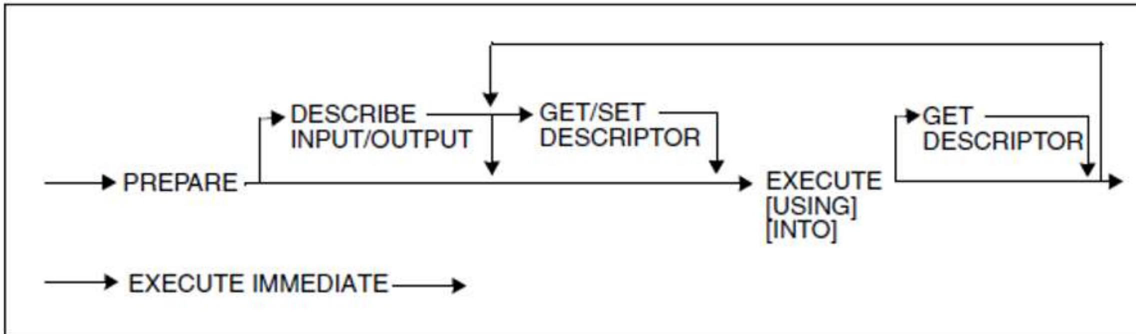
2.5.1 Dynamic statement

A dynamic statement can either be executed directly once, or it can be prepared. A prepared statement can be executed any number of times.

You cannot use any placeholders in a statement that is executed directly, and it must not return any values.

A prepared statement remains prepared for execution for at least the duration of the current transaction.

The diagram below provides you with an overview of the SQL statements that can be used in dynamic statements:



A descriptor area must be created with ALLOCATE DESCRIPTOR before it is used in DESCRIBE and GET/SET DESCRIPTOR (see section “Descriptor area”).

2.5.1.1 Prepare a dynamic statement

You prepare a dynamic statement with PREPARE. You define a name, or statement identifier, that is used to refer to the dynamic statement in subsequent statements and in the EXECUTE statement in particular. All SQL statements that can be prepared are listed in the section “Assignments for PREPARE”.

You specify an alphanumeric host variable for the as yet unknown SQL statement represented by the statement identifier. The length of the variable must not exceed 32000 characters. You cannot specify an indicator variable.

In the program, you assign the host variable the desired SQL statement as an alphanumeric string. You can, for example, read in the SQL statement via an interactive program and then use it to construct the string that is transferred to the host variable.

When the PREPARE statement is executed, the dynamic statement must be known with the exception of the values of the placeholders. If the statement is not correct, the PREPARE statement is aborted with errors.

2.5.1.2 Querying the data types of the placeholders and values

If a dynamic statement contains placeholders, you can query the number and SQL data types of the placeholders with DESCRIBE INPUT after you have prepared the statement with PREPARE. To do this, you must specify a descriptor area to which the description of the SQL data types is returned.

You can query the number and data types of the values returned by the prepared statement with DESCRIBE OUTPUT and store the information in a previously requested descriptor area. The number is 0 if the prepared statement is not a SELECT statement or cursor description.

You can read the item descriptors in the descriptor area with GET DESCRIPTOR (see section “Descriptor area”).

2.5.1.3 Execute a dynamic statement

You can prepare and execute a dynamic statement directly with EXECUTE IMMEDIATE. In this case, however, the statement cannot include any placeholders or return any values. All the SQL statements that can be executed with EXECUTE IMMEDIATE are listed in the description of the EXECUTE IMMEDIATE statement, "EXECUTE IMMEDIATE - Execute dynamic statement".

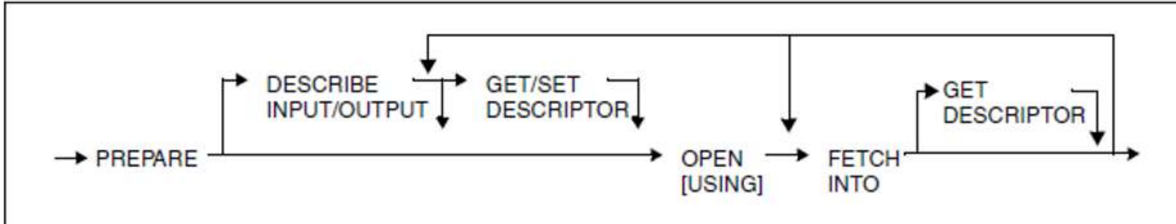
You execute a statement prepared with PREPARE with the EXECUTE statement. If the statement includes placeholders, the corresponding values can be made available via host variables or via a descriptor area that has already been supplied with values in the USING clause of the EXECUTE statement.

In a dynamic SELECT statement, the INTO clause can be used to store the results in host variables or in a previously created descriptor area.

2.5.2 Dynamic cursor descriptions

A cursor can also be assigned a dynamic cursor description in the DECLARE CURSOR statement. The cursor is then referred to as a dynamic cursor. A non-dynamic cursor is also referred to as a static cursor. A dynamic cursor description is prepared with the PREPARE statement.

The figure below provides you with an overview of the SQL statements for dynamic cursor descriptions:



The other SQL statements relevant to cursors are described in the sections “Read-only cursors” and “Updatable cursors”.

2.5.2.1 Preparing dynamic cursor descriptions

You prepare a dynamic cursor description with the PREPARE statement. You define a name, or statement identifier, for the cursor description. Each cursor declared with this statement identifier is assigned the corresponding cursor description.

You specify an alphanumeric host variable for the as yet unknown query expression. The length of the variable must not exceed 32000 characters. You cannot specify an indicator variable.

When the program is executed, you assign the host variable the desired query expression as an alphanumeric string.

Except for the values of the placeholders, the query expression must be known when the PREPARE statement is executed. If the query expression is not correct, the PREPARE statement is aborted with errors.

2.5.2.2 Determining the SQL data types of the placeholders

If a dynamic cursor description includes placeholders, you can query the number and SQL data types of the placeholders with DESCRIBE INPUT after the cursor description has been prepared with the PREPARE statement.

To do this, you must specify a descriptor area to which the description of the data types is returned. You can read the item descriptors in the descriptor area with GET DESCRIPTOR (see section “Descriptor area”).

2.5.2.3 Determining the SQL data types of the derived columns

You can query the number and SQL data types of the derived columns of a dynamic cursor description with DESCRIBE OUTPUT and store the information in a previously created descriptor area.

2.5.2.4 Evaluating dynamic cursor descriptions

A dynamic cursor description is evaluated when the cursor is opened with the OPEN statement.

If a dynamic cursor description includes placeholders, the associated values can be made available in the USING clause of the OPEN statement via host variables or a descriptor area that has already been supplied with values. Otherwise, the same rules apply to the evaluation of a dynamic cursor as apply to a static cursor.

2.5.2.5 Storing results

The rows of the cursor table are read with FETCH, just as they are for a static cursor. Unlike a static cursor, the column values of a row that are read can be stored not only in host variables but also in a previously created descriptor area.

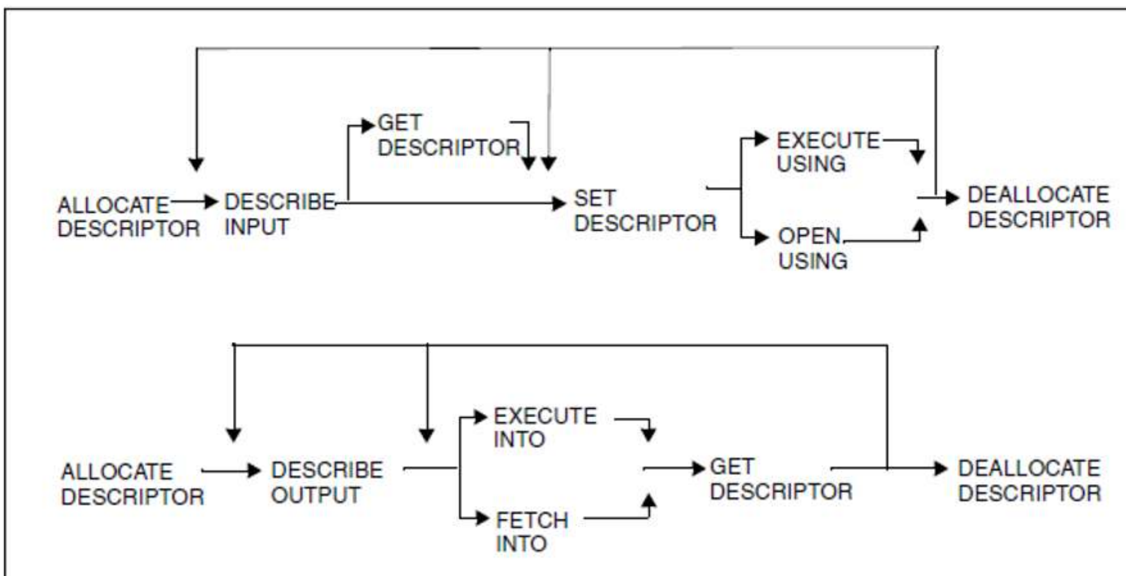
2.5.3 Descriptor area

A descriptor area is a storage area that you use to store values or information about the SQL data types for dynamic statements or cursor descriptions.

A descriptor area can be used in the following cases:

- The SQL data types of the placeholders in a prepared statement or cursor description can be queried and stored in a descriptor area (DESCRIBE INPUT).
- The SQL data types of the derived columns of a prepared SELECT statement or cursor description can be queried and stored in a descriptor area (DESCRIBE OUTPUT).
- The values for the placeholders in a dynamic statement or cursor description can be transferred from a descriptor area upon execution (USING clause of EXECUTE or OPEN).
- The values returned by a dynamic statement or cursor description can be stored in a descriptor area (INTO clause of EXECUTE or FETCH).

There are a number of SQL statements that use descriptor areas. These statements must be called in a predefined order.
The figure below provides you with an overview of these statements and indicates the order in which the statements can be called (GET/SET DESCRIPTOR can be a series of GET/SET DESCRIPTOR statements).



2.5.3.1 Creating a descriptor area

You create a descriptor area with ALLOCATE DESCRIPTOR. You must specify the maximum number of items that this descriptor area can hold.

The items themselves are still undefined after ALLOCATE DESCRIPTOR.

2.5.3.2 Structure of a descriptor area

A descriptor area consists of a COUNT field and a number of items (item descriptors).

Each item in the descriptor area consists of a number of fields that describe an SQL data type and which may contain a value of this type.

One item descriptor is used for an atomic column or value. In the case of a multiple column or aggregate, one item descriptor is used for each column element or occurrence.

2.5.3.3 Descriptor area fields

The descriptor area fields include the COUNT field, which exists once for each descriptor area, and the fields of the various items.

Each descriptor item consists of the following fields:

- REPETITIONS
- TYPE
- DATETIME_INTERVAL_CODE
- PRECISION
- SCALE
- LENGTH
- INDICATOR
- DATA
- OCTET_LENGTH
- NULLABLE
- NAME
- UNNAMED

You will find detailed descriptions of the various fields below.

COUNT

The COUNT descriptor area field contains a value for the number of item descriptors used or required.

If the number of item descriptors specified in a DESCRIBE statement is greater than the defined maximum number of items, only the COUNT field is set to the specified number. All other fields are not assigned a value.

SQL data type: SMALLINT

Item descriptor fields

Not all the fields are supplied with a value for each item descriptor. Fields that have not been supplied with a value have an undefined value.

The fields are described in alphabetical order below.

DATA

> Is only defined if the value in the INDICATOR field is greater than or equal to 0: Value of the item descriptor.
>
> SQL data type: determined by the fields TYPE, LENGTH, PRECISION, SCALE and DATETIME_INTERVAL_CODE

DATETIME_INTERVAL_CODE

> Only for date and time data types:
>
> Data type of the item descriptor.

DATETIME_INTERVAL_CODE	SQL data type
1	DATE
2	TIME
3	TIMESTAMP

Table 1: Descriptor area field DATETIME_INTERVAL_CODE

SQL data type: SMALLINT

INDICATOR

Information on the value of the item descriptor:

< 0	Value is the NULL value
> 0	Original length of an alphanumeric or national string that was truncated during transfer from the database
0	else

SQL data type: SMALLINT

LENGTH

Only for alphanumeric, national and time data types:

Length of the SQL data type in characters or code units for national data types.

LENGTH	For SQL data type
length	CHAR(*length*)
max	VARCHAR(*max*)
cu_length	NCHAR(*cu_length*)
cu_max	NVARCHAR(*cu_max*)
10	DATE
12	TIME(3)
23	TIMESTAMP(3)

Table 2: Descriptor area field LENGTH

SQL data type: SMALLINT

NAME

Column name if the item refers to a column, otherwise a column name that is used internally.

SQL data type: CHAR(*n*) or VARCHAR(*n*), where *n* >= 128

NULLABLE

Specification of whether the value of the item descriptor can be the NULL value.

1	Value can be the NULL value
0	else

SQL data type: SMALLINT

OCTET_LENGTH

Maximum memory requirements of the data type indicated by the fields TYPE, LENGTH, PRECISION, SCALE and DATETIME_INTERVAL_CODE in bytes. If these fields do not specify a correct SQL data type, the value of OCTET_LENGTH is undefined.

The value of OCTET_LENGTH is implementation-dependent for numeric and time data types and may change in future versions of SESAM/SQL.

OCTET_LENGTH	**For SQL data type**
length	CHAR(*length*)
max+2	VARCHAR(*max*)
*2*cu_length*	NCHAR(*cu_length*)
*2*cu_max*+2	NVARCHAR(*cu_max*)
precision+1	NUMERIC(*precision*,*scale*)
precision/2+1, if *precision* even (*precision*-1)/2+1, else	DECIMAL(*precision*,*scale*)
4	INTEGER
2	SMALLINT
4, if *precision*<22 8, else	FLOAT(*precision*)
4	REAL
8	DOUBLE PRECISION
6	DATE
8	TIME(3)
14	TIMESTAMP(3)

Table 3: Descriptor area field OCTET_LENGTH

SQL data type: SMALLINT

PRECISION

Only for numeric data types and TIME and TIMESTAMP:

number of decimal or binary digits of the SQL data type.

PRECISION	For SQL data type
precision	NUMERIC(*precision*,*scale*)
precision	DECIMAL(*precision*,*scale*)
31	INTEGER
15	SMALLINT
precision	FLOAT(*precision*)
21	REAL
53	DOUBLE PRECISION
3	TIME(3)
3	TIMESTAMP(3)

Table 4: Descriptor area field PRECISION

SQL data type: SMALLINT

REPETITIONS

Dimension of a multiple column or aggregate.

A separate item in the descriptor area is used for each occurrence of a multiple column or aggregate. The REPETITIONS field of the first item descriptor contains the number of occurrences or column elements. The REPETITIONS field of all subsequent item descriptors is set to 1.

REPETITIONS is set to 1 for atomic values.

SQL data type: SMALLINT

SCALE

Only for integer and fixed-point number data types:

number of places to the right of the decimal point for the SQL data type.

SCALE	For SQL data type
scale	NUMERIC(*precision*,*scale*)
scale	DECIMAL(*precision*,*scale*)

0	INTEGER
0	SMALLINT

Table 5: Descriptor area field SCALE

SQL data type: SMALLINT

TYPE

SQL data type of the item descriptor:

TYPE	SQL data type
-42	NVARCHAR
-31	NCHAR
1	CHAR
2	NUMERIC
3	DECIMAL
4	INTEGER
5	SMALLINT
6	FLOAT
7	REAL
8	DOUBLE PRECISION
9	DATE, TIME or TIMESTAMP
12	VARCHAR

Table 6: Descriptor area field TYPE

SQL data type: SMALLINT

UNNAMED

Specification of whether the NAME field contains a valid column name.

0	NAME contains a column name
1	else

SQL data type: SMALLINT

2.5.3.4 Assigning values to the descriptor area

Once you have created a descriptor area, you can assign values to this area in a number of ways:

- Data type descriptions:
  You can use DESCRIBE to place the description of the SQL data types of the placeholders or derived values of a prepared statement or cursor description in the descriptor area.
- Values:
  You can use EXECUTE ... INTO or FETCH ... INTO to place queried values in the descriptor area.
- Data type descriptions and values:
  You can use SET DESCRIPTOR to set the items in the descriptor area. The values assigned to the item descriptor fields are described in the section “SET DESCRIPTOR - Update SQL descriptor area”.

The fields NAME, UNNAMED and NULLABLE are only set for DESCRIBE.

The fields TYPE, DATETIME_INTERVAL_CODE, LENGTH, PRECISION, SCALE, REPETITIONS can be set with SET DESCRIPTOR and DESCRIBE.

The fields INDICATOR and DATA can be set with SET DESCRIPTOR or with EXECUTE INTO and FETCH INTO if an SQL descriptor area is used.

If a value is transferred from a host variable to a descriptor area field, the SQL data type of the host variable must satisfy the conditions described for SET DESCRIPTOR, "SET DESCRIPTOR - Update SQL descriptor area", and in the section “Transferring values between host variables and a descriptor area”.

2.5.3.5 Querying the descriptor area

You can query the value of the COUNT field and the fields of individual item descriptors with GET DESCRIPTOR.

To query an item, enter the number of the item descriptor and the fields whose values you wish to query. The item descriptor fields are described in section “Descriptor area fields”.

When transferring a value from an item descriptor field to a host variable, the SQL data type of the host variable must satisfy the conditions described for GET DESCRIPTOR on "GET DESCRIPTOR - Read SQL descriptor area" and in the section “Transferring values between host variables and a descriptor area”.

2.5.3.6 Using values from the descriptor area

The fields TYPE, DATETIME_INTERVAL_CODE, LENGTH, PRECISION, SCALE, REPETITIONS are read for EXECUTE, OPEN and FETCH if an SQL descriptor area is used for the input or output values.

The fields INDICATOR and DATA are read for EXECUTE USING and OPEN USING if an SQL descriptor area is used for the input values.

2.5.3.7 Releasing the descriptor area

If you no longer need a descriptor area, you release the memory used by the descriptor area with DEALLOCATE DESCRIPTOR.

2.6 SQL statements in CALL DML transactions

SESAM/SQL supports the SQL and CALL DML interfaces.
In mixed mode operation, both interfaces can be used together in an ESQL COBOL application (see the “ CALL-DM Applications” manual).
You can use SQL and CALL DML interfaces together within the same transaction: In order to simplify the step-by-step conversion to the SQL environment, it is possible to issue SQL statements within CALL DML transactions in existing CALL DML applications.

CALL DML transaction

A CALL DML transaction starts with the CALL DML statement BTA and ends with a roll forward or rollback of the transaction.
You use the CALL DML statement ETA to roll a CALL DML transaction forward. A transaction is rolled back either by means of the statement RTA or internally by SESAM/SQL DBH when, for example, a deadlock is resolved.
Under openUTM, a transaction is rolled forward by the PEND variable which ends the transaction and rolled back by rolling back the UTM transaction.

Permitted SQL statements in a CALL DML transaction

Within a CALL DML transaction you can execute all SQL statements which are used to query and change data, SQL statements for dynamic SQL, some SQL statements for session control, the CALL statement, and the WHENEVER statement (for the initiation of the SQL statements, see section “Summary of contents”).

The following SQL statements are not permitted within a CALL DML transaction:

- COMMIT WORK
- ROLLBACK WORK

Any statements which are not permitted in a SQL-DML transaction are also not permitted:

- SET TRANSACTION
- SET SESSION AUTHORIZATION
- SQL statements for schema definition and administration
- SQL statements for managing the storage structure
- SQL statements for managing user entries
- Utility statements

> **i** If the SET TRANSACTION statement is issued before a CALL DML transaction, the settings are only valid for existing SQL statements within the following (CALL DML) transaction. After the transaction is finished the defaults are valid again.

2.6.1 Step-by-step conversion of CALL DML statements

In order to convert existing CALL DML statements to work with the SQL interface, it is advisable to perform the steps in a given order. Below you can find a brief summary of the most important steps listed in accordance with the type of application or statement.

TIAM application

If you want to convert a CALL DML transaction into a TIAM statement for use with the SQL interface, proceed as follows:

1. One at a time, replace all CALL DML statements other than BTA, ETA and RTA with SQL statements
2. Then replace the BTA, ETA and RTA statements:
    - delete BTA without replacement
    - replace ETA with COMMIT WORK
    - replace RTA with ROLLBACK WORK

openUTM application

If you want to convert a CALL DML transaction into an openUTM application for use with the SQL interface, proceed as follows:

1. One at a time, replace all CALL DML statements other than BTA, ETA and RTA with SQL statements
2. Then replace the BTA, ETA and RTA statements:
    - delete BTA without replacement
    - delete ETA without replacement
    - replace RTA with RSET (RSET is a function at the openUTM KDCS interface)

CALL DML statements outside a CALL DML transaction

In order to convert CALL DML statements which are issued outside of CALL DML transactions for use at the SQL interface, you must replace them by the corresponding SQL statements. In this case, there are no restrictions concerning permitted SQL statements. Note that most SQL statements implicitly open a transaction. This must be closed before the next CALL DML statement.

2.6.2 Using User-Close and release session resources

The User-Close in a CALL DML application closes all the requesting user's logical files. After the successful execution of User-Close, all resources of the logical files of this user are released.
Within an SQL application it is not possible to terminate an SQL conversation explicitly. The resources of an SQL conversation are not released until the associated TIAM application has terminated. Under openUTM, the resources of an SQL conversation are released when the associated UTM conversation terminates.

There is no statement in SQL which is equivalent to a User-Close in a CALL DML application. If a CALL DML statement contains multiple User-Close statements you should therefore increase the DBH option USERS before you switch to the SQL interface. In this way, you can avoid resource bottlenecks.

2.6.3 Setting the isolation level

The locking concept which ensures data consistency is implemented in CALL DML applications in the following way: if a retrieval statement accesses the user data in a CALL DML table, SESAM/SQL DBH locks the relevant record against access by other transactions until the executing transaction is either terminated or rolled back. Depending on the Open mode a shared or exclusive lock is set. In addition, SESAM/SQL permits the following modifications of the locking concept for individual CALL DML statements:

- reading without a lock (Read No Lock)
- ignoring the lock (Read No Wait)
- reading without a lock and ignoring the lock

When a CALL DML transaction is converted it is advisable to change the locking behavior as little as possible. If a shared or exclusive lock is set for a CALL DML transaction, you should use the SQL statement SET TRANSACTION to set the isolation level REPEATABLE READ prior to the transaction.

If the locking behavior for individual CALL DML applications has been changed, it is advisable to use the pragma ISOLATION LEVEL. You can use this to define a specific isolation level for the corresponding SQL statement which is equivalent to the locking behavior of the associated CALL DML statement:

- replace “Read No Lock“ with READ COMMITED
- replace “Read No Lock and Read No Wait“ with READ UNCOMMITED

Only in the case of “Read No Lock“ SESAM/SQL ignorant of the corresponding isolation level. Here, you should decide on a case-by-case basis whether the isolation level READ COMMITED or READ UNCOMMITED is more suitable.

3 Lexical elements and names

This chapter describes the following:

- SESAM/SQL character repertoire
- Lexical units
- Pragmas and annotations
- Names

3.1 SESAM/SQL character repertoire

The SESAM/SQL character repertoire consists of letters, digits and special characters.

Letters are uppercase letters A-Z and lowercase letters a-z (without umlauts and ß).

Digits are the characters 0-9.

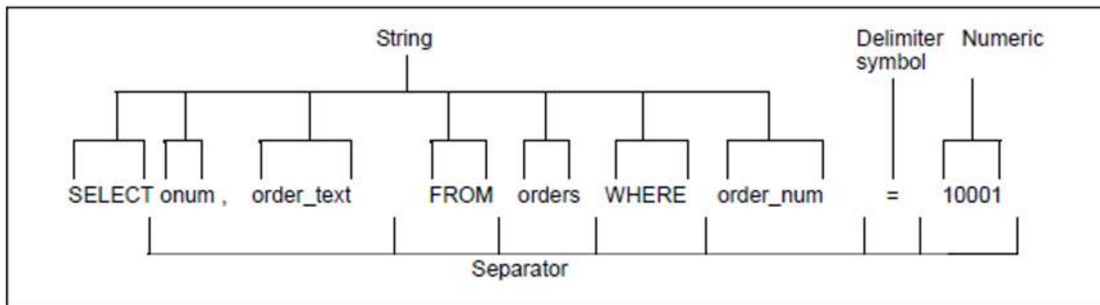
The following are special characters:

" ' : ; , . - & | ( ) = + * / < > ? % _ [ ] (Leerzeichen)

3.2 Lexical units

The text sequences formed from the SQL character repertoire are divided into lexical units. An SQL statement consists of the following lexical units:

- strings
- numerics
- delimiter symbols
- Separators
- Comments



3.2.1 Strings

Examples of character strings are the SQL keywords and names, as well as alphanumeric literals, national literals and time literals.

Strings for SQL keywords

An SQL keyword is a sequence of uppercase or lowercase letters. An SQL keyword is not enclosed in double or single quotes. You will find a list of all SQL keywords in the section “SQL keywords”.

Example: SELECT

In this manual, all SQL keywords appear in uppercase letters to distinguish them from the rest of the text.

Strings for names

The syntax for names is described in the section “Names”.

Strings for literals

Strings for alphanumeric literals, national literals and time literals are enclosed in single quotes (see section “Alphanumeric literals”, section “National literals” and section “Time literals”).

Example: 'Miller'

3.2.2 Numerics

A numeric is a sequence made up of the digits 0-9. Numeric literals are constructed from numerics and the characters + - . E.

Example: 314

The syntax for numeric literals is described in section “Numeric literals”.

3.2.3 Delimiter symbols

Examples of the delimiter symbols are the operators and the following special characters:

```
: ; , . ( ) [ ] ?
```

Operators

Operators are used to create expressions and predicates. The following table provides an overview of the operators defined in SESAM/SQL:

Operator	Meaning
*	Multiplication
/	Division
+	Addition
-	Minus sign
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
||	Concatenation

Table 7: Operators

The meaning of the operators is explained in detail in the chapter “Compound language constructs”.

3.2.4 Separators

You use separators to separate lexical units. Separators are blanks, newline markers and comments.

3.2.5 Comments

SQL allows you to add comments for the purpose of documenting SQL statements. Comments start with the character string -- and end with the end of the line. There are also parenthesized comments which start with `/*` and end with `*/` and which can also be nested.

Pragmas and annotations are also considered comments (see section "Pragmas and annotations").

3.3 Pragmas and annotations

Pragmas and annotations are special SQL comments which are interpreted by SESAM/SQL. You can use them to provide information for the execution of SQL or utility statements. Pragmas and annotations containing syntax errors are treated as comments and ignored by SESAM/SQL.

A **pragma** can only be contained at the start of an SQL or utility statement. They may be preceded only by comments (including further pragmas) and delimiters. Pragmas have an effect on the entire statement, including the views used. The PREFETCH pragma even has an effect on all operators with a cursor.

An **annotation** can only be contained at certain positions in the text of a statement. Irrespective of its position, it has an effect only on one particular operation in the statement. Only one annotation can ever be contained at each of these positions. However, a statement can contain multiple annotations and also the views used.

Pragmas and annotations have an effect only in the case of particular sets of statements, otherwise they are ignored. For information on using pragmas in routines, see section "Pragmas in routines".

Pragmas and annotations are used for different purposes. They are described in various SESAM/SQL manuals, see the tables on the following pages.

Format

pragma ::= `--%PRAGMA` *pragma_text* , ... *end_of_line*

annotation ::= `/*%` *annotation_text* `%*/`

pragma_text

A string of keywords, literals and names.
The string may contain blanks but no other delimiters.

The formats for *pragma_text* and its effect are described in the places specified in the table below:

***pragma_text* begins with**	**Meaning**	**For description see**
AUTONOMOUS TRANSACTION	Write data independently of the surrounding transaction	"AUTONOMOUS TRANSACTION pragma"
CHECK	Observe integrity constraints	" SQL Reference Manual Part 2: Utilities" manual
DATA TYPE	Use old CALL-DML types	"DATA TYPE pragma"
DEBUG ROUTINE	Receive error information for routines	"DEBUG ROUTINE pragma"
DEBUG VALUE	Receive information for assignments in routines	"DEBUG VALUE pragma"
EXPLAIN	Output access plan	"EXPLAIN pragma"
IGNORE	Ignore index	" Performance" manual

ISOLATION LEVEL	Define isolation level	"ISOLATION LEVEL pragma"
JOIN	Select join method	" Performance" manual
KEEP JOIN ORDER	Retain join order	" Performance" manual
LIMIT ABORT_EXECUTION	Limit resource utilization	"LIMIT ABORT_EXECUTION pragma"
LOCK MODE	Set lock mode	"LOCK MODE pragma"
LOOP LIMIT	Limit number of loop passes	"LOOP LIMIT pragma"
OPTIMIZATION	Restrict access planning	" Performance" manual
PREFETCH	Control block mode	"PREFETCH pragma"
SIMPLIFICATION	Control optimization techniques	" Performance" manual
USE	Use index	" Performance" manual
UTILITY MODE	Control transaction management	"UTILITY MODE pragma"

Table 8: pragmas

When you specify more than one pragma beginning with the same keyword in a statement, the last one specified is used. However, regardless of their order the IGNORE and USE pragmas are interpreted according to special rules.

end_of_line

New line in the SQL source text.
When the SQL text is specified as a string in a PREPARE or EXECUTE IMMEDIATE statement, the alphanumeric character X'15' in this string means new line.

annotation_text

A string of keywords.
The string may contain blanks and new lines, but no comments.

An annotation must follow a keyword. Only blanks and new lines may be contained between these, but no comments. The preceding keyword determines the permitted format of *annotation_text* and the effect of the annotation. An annotation which does not comply with these rules is regarded as a comment and ignored.

The formats for *annotation_text* and its effects are described in the place specified in the table below:

Annotation after keyword	**Meaning**	**For description see**
JOIN	Select join algorithm	" Performance" manual
CACHE	Cache CSV file in temporary file	" Performance" manual

VOLATILE	Always calculate function value anew	"Uncorrelated function calls"
IMMUTABLE	Do not calculate function value anew in uncorrelated function calls	"Uncorrelated function calls"

Table 9: Annotations

If a pragma and an annotation would have different effects on an operation in a statement (e.g. selection of different Join algorithm), the annotation normally has priority. The description of the annotation contains the details.

3.3.1 AUTONOMOUS TRANSACTION pragma

The pragma AUTONOMOUS TRANSACTION enables data to be written to a database irrespective of the surrounding transaction.
In particular, the data is written persistently to the database before the SQL statement ROLLBACK WORK has possibly executed the transaction.

The pragma may only be specified in SQL statements for modifying data, i.e. in INSERT, UPDATE (search condition satisfied), DELETE (search condition satisfied), MERGE, and CALL. If the pragma is specified in statements for querying data, the statement is rejected with SQLSTATE.

The pragma may not be used in routines.

```
AUTONOMOUS TRANSACTION
```

Notes

- The SQL statement after the pragma AUTONOMOUS TRANSACTION is executed in the user's current transaction, but in a separate runtime environment (own thread, own transaction context). The user's transaction-control statements have no effect.

  The internal user identification (`APPLICATION-NAME=AUTTRAN`) is used, see the " Database Operation" manual. It is visible in information outputs while the autonomous transaction is executing. However, an autonomous transaction cannot be administered.
- Lock conflicts
  The transaction context of the autonomous transaction is independent of the application's surrounding transaction and of other transactions.
  On the one hand, this can lead to a deadlock between the autonomous transaction and the surrounding transaction. This deadlock is resolved by resetting the autonomous transaction. The autonomous transaction is reported to the SQLSTATE 81SAT.On the other hand, this can lead to a deadlock between the autonomous transaction and other transactions. Such deadlocks are resolved by resetting the "least costly" transaction. When the autonomous transaction is affected by this, the SQLSTATE 81SAT is reported to it.
- Canceling the application
  When the application which triggered thr autonomous transaction aborts, first the autonomous transaction is canceled, and then the current transaction or the application.

3.3.2 DATA TYPE pragma

The DATA TYPE pragma indicates that a column can only be created in the attribute format for CALL DML tables.

This pragma only takes effect if it is specified in the ALTER TABLE ... ADD COLUMN statement and the table is a CALL DML table.

```
DATA TYPE OLDEST
```

3.3.3 DEBUG ROUTINE pragma

The DEBUG ROUTINE pragma provides additional information on an execution of a routine which is possibly errored. This information can be read using the SYS_ROUTINE_ERRORS view of the SYS_INFO_SCHEMA, see "SYS_ROUTINE_ERRORS".

The DEBUG ROUTINE pragma is effective only outside routines. It is only effective ahead of the SQL statement CALL and ahead of the DML statements DECLARE CURSOR, DELETE, INSERT, MERGE, SELECT, and UPDATE. When specified **ahead of** DML statements, the pragma has an effect on all User Defined Functions (UDFs) and the routines of the DML statement these contain.

> i The pragma has been renamed SESAM/SQL V9.0. For compatibility reasons, DEBUG PROCEDURE can also still be specified.

```
DEBUG ROUTINE [ALL | USER] [LEVEL unsigned_integer ]
```

unsigned_integer

When *unsigned_integer* > 0, additional information is collected for the executed SQL statements of the current routine.

unsigned_integer = 1 is the default value when the `LEVEL` clause is not specified.

When *unsigned_integer* = 0, the pragma is ignored.

The following approach makes sense:
The pragma is initially active in an application with a value > 0 in, and then later (without changing the text length) disabled by the value 0.

USER

Depending on the `LEVEL` set, information is collected for the SQL statements which are prefixed by the DEBUG VALUE pragma (see "DEBUG VALUE pragma").

ALL

In addition to the DEBUG information mentioned under USER, general DEBUG information is also created (irrespective of the `LEVEL` set).
For example, every SQLSTATE or SQLrowcount reported by an errored SQL statement is recorded. Internal calls of routines are also recorded. The position of an SQL statement within the text of a routine is normally also recorded.

3.3.4 DEBUG VALUE pragma

The DEBUG VALUE pragma provides additional information for the following SQL statements.

- SET in routines (procedures and User Defined Functions (UDFs))
- RETURN in User Defined Functions (UDFs)

This information can be read using the SYS_ROUTINE_ERRORS view of the SYS_INFO_SCHEMA, see "SYS_ROUTINE_ERRORS".

The DEBUG VALUE pragma is currently only effective before these SQL statements.

```
DEBUG VALUE [LEVEL unsigned_integer ]
```

unsigned_integer

When *unsigned_integer* > 0, additional information is collected for the aforementioned statements when the DEBUG ROUTINE pragma is positioned ahead of the SQL statement CALL or ahead of a DML statement (for routines contained in this). In addition, *unsigned_integer* for DEBUG ROUTINE must be greater than or equal to *unsigned_integer* for DEBUG VALUE.

The following information is then collected:

- In the case of SET, the assigned value and the name of the target field (parameter or local variable)
- In the case of RETURN, the value returned

In the case of strings, long values are, if required, truncated.

unsigned_integer = 1 is the default value when the `LEVEL` clause is not specified.

When *unsigned_integer* = 0, the pragma has no effect.

The following approach makes sense:

> The pragma is initially active in an application with a value > 0 in, and then later (without changing the text length) disabled by the value 0.

> i The DEBUG VALUE pragma can also remain in the text of a routine after the end of a test or debugging phase provided the calling SQL statements do not use the corresponding DEBUG ROUTINE pragma.

Example

The SET statements of a procedure can be prefixed with the DEBUG VALUE pragma with various values for *unsigned_integer* . Calling the routine with the DEBUG ROUTINE pragma and different values for *unsigned_integer* causes information to be collected in various scopes.

```
CREATE PROCEDURE P (OUT par1 INTEGER,OUT par2 INTEGER)
       MODIFIES SQL DATA
       BEGIN
       --%PRAGMA DEBUG VALUE LEVEL 3
          SET par1 = 42;
       --%PRAGMA DEBUG VALUE LEVEL 10
          SET par2 = 43;
       END
```

With the procedure call below, only the first assignment (`par1=42`) is recorded:

```
-- %PRAGMA DEBUG ROUTINE LEVEL 5

CALL P(mypar1, mypar2)
```

Both assignments are recorded in the case of the procedure call below:

```
-- %PRAGMA DEBUG ROUTINE LEVEL 20

CALL P(mypar1, mypar2)
```

The DEBUG VALUE pragmas can remain unchanged in the text of the routine. They only have an effect when there is a corresponding *unsigned_integer* in the DEBUG ROUTINE pragma.

3.3.5 EXPLAIN pragma

The EXPLAIN pragma is used to output the access plan selected by the optimizer. You can only use this pragma if the current authorization identifier has the special privilege UTILITY.

This pragma is only effective in the following SQL statements:

- CALL
- cursor description (for dynamic cursors)
- DECLARE CURSOR (for a static cursor)
- DELETE
- INSERT
- MERGE
- SELECT
- UPDATE

In routines, the pragma is ignored, see section “Pragmas in routines”.

This pragma is only effective in a static statement if you precompile the program while the database is online.

```
EXPLAIN INTO file
```

file

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

If *file* includes a BS2000 user ID, this user ID is used. If not, the ID of the Data Base Handler for the database referenced in the SQL statement is used. In both cases the DBH must have write permission for the file. You specify an alphanumeric literal for *file*. No lowercase letters should be contained in this.

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

The explanation comprises the SQL statement and an edited representation of the access plan. The representation of access plans is described in the “ Performance” manual.

You can display the contents of the file with SHOW-FILE. If you want to read the file with EDT, you must enter the following command:
`ADD-FILE-LINK LINK-NAME=EDTSAM,FILE-NAME=`*file*`,...,BUFFER-LENGTH=(STD,2),...`

In the EDT you can also enter: `@OPEN F=`*file*`,TYPE=CATALOG`

3.3.6 ISOLATION LEVEL pragma

The ISOLATION LEVEL pragma determines the isolation level for database accesses performed by an SQL or utility statement.

This pragma is only effective in the following SQL statements:

- CALL and in routines (see section “Pragmas in routines”)
- cursor description (for dynamic cursors)
- DECLARE CURSOR (for a static cursor)
- DELETE
- INSERT
- MERGE
- SELECT
- UPDATE

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

> **CAUTION!** The ISOLATION LEVEL READ NOWAIT can only be set by Pragma but not within the SET TRANSACTION Statement. If you have specified the ISOLATION LEVEL pragma, any database access performed in connection with this statement takes place under CONSISTENCY LEVEL 1, see "SET TRANSACTION - Define transaction attributes".
>
> If you specify a lower isolation level than specified for the transaction, the isolation level defined for the transaction is no longer guaranteed.

The isolation levels are described in the section “SET TRANSACTION - Define transaction attributes”.

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

3.3.7 LIMIT ABORT_EXECUTION pragma

The LIMIT ABORT_EXECUTION pragma controls the use of resources during the processing of an SQL statement. This pragma allows you to systematically provide statements with a local stop criterion. This local stop criterion is more restrictive than the global stop criterion ABORT-EXECUTION required for complex batch programs. ABORT-EXECUTION is set using RETRIEVAL-CONTROL or MODIFY-RETRIEVAL-CONTROL.

The local stop criterion set using LIMIT ABORT_EXECUTION

- is only valid for the current request.
- cannot be overridden by MODIFY-RETRIEVAL-CONTROL.
- has no effect if the pragma is not in a “searching“ statement.
- has no effect if the value has been specified as 0 or the specified value is greater than that of the global stop criterion. In this case the value of the global stop criterion applies.

If several LIMIT ABORT_EXECUTION pragmas are specified in one request, the last valid pragma value will apply. If no LIMIT ABORT_EXECUTION pragma is specified, the global stop criterion will apply.

In a sequence of DECLARE CURSOR, OPEN and FETCH statements, the pragma must be specified in the DECLARE CURSOR statement. Its effect depends on the search path selected, but only when the OPEN or FETCH statement is executed.

The pragma can also be used in CALL and in routines, see section “Pragmas in routines” .

`LIMIT ABORT_EXECUTION` *block_access*

---

block_access

This argument allows you to specify the number of logical block access instances. Once this number has been reached, no more hits will be detected and the statement will be terminated. The number of block access instances should be specified as an unsigned integer ranging from 0 to 2147483647.

3.3.8 LOCK MODE pragma

The LOCK MODE pragma sets the lock mode. It is only effective in SQL-DML statements.

The pragma can be used in CALL and in routines, see section “Pragmas in routines”.

```
LOCK MODE EXCLUSIVE
```

If LOCK MODE EXCLUSIVE is specified, every access to the database connected directly or indirectly with this SQL statement involves exclusive locks. Otherwise the lock mode is defined by the system.

3.3.9 LOOP LIMIT pragma

The LOOP LIMIT pragma enables you to limit the number of loop passes in a routine.

The LOOP LIMIT pragma is effective ahead of the SQL statement CALL and ahead of other DML statements. When specified ahead of DML statements, the pragma has an effect on all User Defined Functions (UDFs) and the routines of the DML statement these contain. When placed ahead of SQL statements, the pragma has no effect in a routine.

`LOOP LIMIT` *unsigned_integer*

---

unsigned_integer

Specifies the maximum number of passes for a loop.

When *unsigned_integer*=0, the number of loop passes is unlimited.
unsigned_integer=0 is also the default value when the pragma is not specified.

When this pragma is specified, the loop body is canceled after the specified number of passes has been executed for each called loop of the routine concerned, and an SQLSTATE is reported. This enables endless loops to be avoided.

3.3.10 PREFETCH pragma

The PREFETCH pragma controls the block mode of the SQL statement FETCH (for positioning the cursor). Block mode accelerates the execution of the FETCH statement. It is effective only when FETCH positions the cursor on the next record in the cursor table (FETCH NEXT...).

The PREFETCH pragma allows you to activate block mode and specify a blocking factor (n). When the first FETCH NEXT... statement is executed, the column values of the current record are read, and the next n -1 records of the associated cursor table are stored in a buffer. When the next n-1 FETCH NEXT... statements that specify the same cursor are executed, the next record can be accessed directly without involving the DBH.

The PREFETCH pragma is effective only in the following SQL statements:

- DECLARE CURSOR (for a static cursor)
- cursor description (for dynamic cursors)

If the cursor description of the DECLARE CURSOR statement or the cursor description for dynamic cursors contains a FOR UPDATE clause, the PREFETCH pragma is ignored and block mode is not activated.

When block mode is activated, it makes the cursor defined in the DECLARE CURSOR statement or the cursor description the prefetch cursor.

Block mode cursors are not supported in linked-in mode.

`PREFETCH` *blocking_factor*

---

blocking_factor

You must enter an integer without a preceding sign as the blocking factor (data type SMALLINT).

If the blocking factor (n) is greater than 0, up to n-1 records of the specified cursor table are stored in a buffer.

If the blocking factor is 0, the PREFETCH pragma has no effect.

You can enable/disable the pragma and thus activate/deactivate block mode by specifying either a value greater than 0 or the value 0 itself for n.

When block mode is activated, the following restrictions apply:

- Only the FETCH NEXT statement is permitted for the prefetch cursor *cursor* in the same compilation unit. The following SQL statements can no longer be executed:
  - UPDATE ... WHERE CURRENT of *cursor*
  - DELETE ... WHERE CURRENT of *cursor*
  - STORE *cursor*
  - FETCH *cursor* with a cursor position other than NEXT or with a different INTO clause to the first FETCH NEXT statement.
- After the execution of a FETCH NEXT statement whose INTO clause contains the name of an SQL descriptor area, this SQL descriptor area must not be modified by a SET DESCRIPTOR, DESCRIBE or DEALLOCATE DESCRIPTOR statement.

- The prefetch cursor must always be addressed by the same FETCH NEXT statement, i.e. by the same statement in a loop or subroutine.

3.3.11 UTILITY MODE pragma

The UTILITY MODE pragma determines whether transaction logging is effective in the SQL statement in which this pragma is specified. Transaction logging makes it possible to roll a transaction back to a consistent state.

The UTILITY MODE pragma is only effective in the SQL statement ALTER TABLE:

It only works if the ALTER TABLE statement adds, changes or deletes columns in a base table. In an ALTER TABLE statement which adds or deletes integrity constraints, the UTILITY MODE pragma has no effect.

```
UTILITY MODE [ON | OFF]
```

ON Transaction logging is deactivated during the execution of the SQL statement. The associated ALTER TABLE statement does not open a transaction.
No save data for the ALTER TABLE statement is stored. If an error occurs which results in an interruption of the statement, the transaction cannot be rolled back to a consistent state. When an error occurs, the space containing the base table is damaged and must be repaired using the RECOVER utility statement (see the “ SQL Reference Manual Part 2: Utilities”).

OFF The pragma has no effect.
The transaction logging remains active.

An ALTER TABLE statement, for which the UTILITY MODE pragma is switched ON and is effective, is aborted with an error message in the following cases:

- when a transaction is active
- when the ALTER TABLE statement deletes a column, i.e. using DROP COLUMN *column* CASCADE
- when the ALTER TABLE statement deletes a column and an index for this column is still defined
- when the ALTER TABLE statement adds a column with an index definition for this column

If no UTILITY MODE pragma is specified for an ALTER TABLE statement then the default setting, UTILITY MODE OFF, is effective.

! **CAUTION!** If you use the UTILITY MODE ON pragma then, after an error or consistency check, the space containing the base table to be changed is defective. To avoid data loss, you should save the space before issuing the ALTER TABLE statement. The save is necessary if you want to use the utility statement RECOVER to repair it.

3.4 Names

Names are strings used to identify objects.

In SESAM/SQL, there are names for the following SQL objects:

- database (catalog)
- Schema
- Space
- Storage group
- table (base table, view, correlation)
- Column
- Index
- Integrity constraint
- Authorization identifier
- cursor
- Routine (routine parameter, local variable, error)
- label
- dynamic statement:
  The name of a dynamic statement is referred to in this manual as the **statement identifier** to distinguish it from the actual name of the statement, such as SELECT, for example.
- symbolic attribute name of a CALL DML column:
  The syntax for the symbolic attribute name of a column is the same as the syntax for symbolic attribute names in SESAM/SQL Version 1.x.
- host variable:
  The name of a host variable must observe the conventions of the programming language involved. These conventions are described in the manuals for the relevant programming language and they are not explained here.

3.4.1 Unqualified names

Unqualified names are either regular names consisting of letters, digits and the underscore character that are not enclosed in double quotes, or special names, which must be enclosed in double quotes.

unqual_name `::=` { *regular_name* | *special_name* }

regular_name `::=` *letter* `[` { *letter* | *digit* | `_` } `] ...`

special_name `::=` `"` *character...* `"`

letter `::=` `a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|`

`A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z`

digit `::=` `0|1|2|3|4|5|6|7|8|9`

regular_name

Regular name, which is not enclosed in double quotes. A regular name cannot be a reserved SQL keyword (see section "SQL keywords").

letter

Lowercase letter between a and z or uppercase letter between A and Z of the SESAM/SQL character repertoire. Lowercase letters are automatically converted into uppercase letters. Umlauts cannot be used.

digit

Digit between 0 and 9.

_ Underscore character.

special_name

Special name, which must be enclosed in double quotes. A special name can be a reserved SQL keyword and can include special characters.

character

The first character cannot be the underscore character. Otherwise, you can use any printable character (i.e. >=X'40') in the SESAM/SQL character repertoire for *character*. A distinction is made between uppercase and lowercase letters. If *character* is the double quote character itself (X'7F'), it must be represented by two immediately adjacent double quotes. The pair of double quote characters is considered a single character.

Identical unqualified names

Two regular names are considered identical if, after the letters have been converted into uppercase letters, the characters at the corresponding positions in each name are identical.

A regular name and a special name are considered identical if, after the letters in the regular name have been converted into uppercase letters and the quotations have been removed from the special name, the characters at

the corresponding positions in each name are identical. If the strings have different lengths, the shorter one is padded with blanks.

Two special names are considered identical if, after the quotations have been removed, the characters at the corresponding positions in each name are identical. If the strings have different lengths, the shorter one is padded with blanks.

Example

The following unqualified names are considered identical:

```
ABc
abc
"ABC"
"ABC    "
```

The following unqualified names are different:

`Abc` and `"Abc"`
`"ABC"` and `"abc"`

Identical names can be used interchangeably any time they occur.

The following names of database objects are unqualified names:

{ *statement_id*
| *authorization_identifier*
| *catalog*
| *cursor*
| *unqual_base_table_name*
| *unqual_constraint_name*
| *unqual_index_name*
| *unqual_routine_name*
| *unqual_schema_name*
| *unqual_space_name*
| *unqual_stogroup_name*
| *unqual_view_name*
| *error_name*
| *correlation_name*
| *local_variable*
| *label*
| *routine_parameter*
| *column* }
::= *unqual_name*

statement_id

Name of a dynamic statement. The statement identifier must be unique within the compilation unit.
The statement identifier can be up to 18 characters long.

authorization_identifier

Name of an authorization identifier. The first 10 characters of the authorization identifier must be unique within the database.

If the name of the authorization identifier is specified without double quotes, it can include only letters and digits. If it is enclosed in double quotes, it must start with an uppercase letter and can only include uppercase letters, digits and the special characters "-" and ".". The special characters cannot occur at the end of the significant part of the authorization identifier (the first 10 characters).
The strings "..", ".-" and "-." are not permitted.
The string "--" is permitted.
The authorization identifier can be up to 18 characters long.

catalog

Name of a database. If the name of the database is specified without double quotes, it can include only letters and digits. If it is enclosed in double quotes, it must start with an uppercase letter and can only include uppercase letters, digits and the special characters "-" and ".". The special characters cannot occur at the end of the database name. The strings "..", ".-" and "-." are not permitted. The string "--" is permitted.

The database name may be up to 18 characters long.

cursor

Name of a cursor. A cursor name can only occur once in a DECLARE CURSOR statement within a compilation unit.
The cursor name may be up to 18 characters long.

unqual_base_table_name

Name of a base table. The unqualified name of a base table must be different from the other base table names and view names in the schema.
The unqualified base table name may be up to 31 characters long.

unqual_constraint_name

Name of an integrity constraint. The name must be different from the other integrity constraint names in the schema.
The unqualified name of an integrity constraint can be up to 31 characters long.

unqual_index_name

Name of an index. The unqualified index name must be unique within the index names of the schema.
The unqualified index name may be up to 18 characters long.

unqual_routine_name

Name of a routine. The unqualified routine name must be different from the other routine names in the schema.
The unqualified routine name may be up to 31 characters long.

unqual_schema_name

Name of a schema. The unqualified schema name must be unique within the schema names of a database.
The unqualified schema name may be up to 31 characters long.

unqual_space_name

Name of a space. The first 12 characters of the unqualified space name must be unique within the space names of a database. If the space name is specified without double quotes, it can include only letters and digits. If it is enclosed in double quotes, it must start with an uppercase letter and can only include uppercase letters, digits and the special characters "-" and ".". The special characters cannot occur at the end of the significant part of the space name (the first 12 characters).
The strings "..", ".-" and "-." are not permitted.
The string "--" is permitted.
The unqualified space name may be up to 18 characters long.

unqual_stogroup_name

Name of a storage group. The unqualified name of the storage group must be unique within the storage group of a database. The unqualified name of a storage group can be up to 18 characters long.

unqual_view_name

Name of a view. The unqualified name of a the view must be different from the other base table names and view names in the schema.
The unqualified view name may be up to 31 characters long.

exception_name

Name of an exception or SQLSTATE in a COMPOUND statement.
All exception names in the COMPOUND statement must differ from each other.The exception name may be up to 31 characters long.

correlation_name

Rename a table.
The correlation name may be up to 31 characters long.

local_variable

Name of a local variable in a COMPOUND statement. The variable name must be unique in the COMPOUND statement and differ from all parameter names in the routine.
The variable name may be up to 31 characters long.

label

Name of a label in a routine. The label may not be identical to another label in the body statement.
Reserved keywords and the following names are not permitted as label names: ATOMIC, DO, ELSEIF, ITERATE, IF, LEAVE, LOOP, REPEAT, RESIGNAL, SIGNAL, UNTIL, WHILE.
The label name may be up to 31 characters long.

routine_parameter

Name of a routine parameter. The parameter name must be unique within the routine. The parameter name may be up to 31 characters long.

column

Name of a column. The column name must be unique within the table.
The unqualified column name may be up to 31 characters long.

3.4.2 Qualified names

You can qualify the names of objects in an SQL statement in order to uniquely identify different objects that have the same name. The following qualifications are possible:

- qualification with the database name for:
  schema, space, storage group, table, index, integrity constraint and routine
- qualification with the schema name for:
  table, index, integrity constraint and routine
- qualification with the table name or the correlation name for:
  column (see "Table specifications")

The syntax overview below illustrates these possibilities:

---

qualified_name ::=

{ *index*

| *integrity_constraint_name*

| *routine*

| *schema*

| *space*

| *stogroup*

| *table* }

index ::= [[*catalog* .] *unqual_schema_name* .] *unqual_index_name*

integrity_constraint_name ::= [[*catalog* .] *unqual_schema_name* .] *unqual_constraint_name*

routine ::= [[*catalog* .] *unqual_schema_name* .] *unqual_routine_name*

schema ::= [*catalog* .] *unqual_schema_name*

space ::= [*catalog* .] *unqual_space_name*

stogroup ::= [*catalog* .] *unqual_stogroup_name*

table ::=

{ [[*catalog* .] *unqual_schema_name* .] *unqual_base_table_name* |

[[*catalog* .] *unqual_schema_name* .] *unqual_view_name* |
correlation_name }

Implicit qualification

The following implicit qualification is valid:

- If no schema qualification is specified, the name refers to the default schema.

- If no catalog qualification is specified, the name refers to the default database.

The default schema and database are set with the precompiler option SOURCE-PROPERTIES (see the “ ESQL-COBOL for SESAM/SQL-Server” manual). The default database and schema names can be redefined with SET CATALOG and SET SCHEMA respectively. The redefined default values are valid for all statements prepared with PREPARE or executed with EXECUTE IMMEDIATE from the time redefinition is performed up until the defaults are redefined again or until the end of the SQL session.

> **i** Other rules for implicit qualification apply to CREATE and GRANT statements within a CREATE SCHEMA statement (see section “CREATE SCHEMA - Create schema”).

Example

Qualifying a table name indicates the schema and database to which the table belongs:

`ordercust.orderproc.customers:`

`CUSTOMERS` table in the `ORDERPROC` schema of the `ORDERCUST` database

`orderproc.customers:`

`CUSTOMERS` table in the `ORDERPROC` schema of the default database.

`customers:`

CUSTOMERS table in the default schema

Overview

Name type	Examples	Meaning
Regular name	`Customers` `customers`	“Customers” and “customers” are equivalent
	`job_2`	Numerics and the underscore character are permitted
Special name	`"TAB-ELLE"` `";$&%!"`	Special characters are permitted
	`"with_2_quotes:"""""`	Quotes must be entered twice
Unqualified name	`orderproc`	Schema ORDERPROC
Qualified name	`ordercust.orderproc.View1`	Table VIEW1 in the schema ORDERPROC of the database ORDERCUST
	`"View"."SELECT(5)"`	Single column SELECT(5) in the table View
	`"VIEW"."SELECT"(5)`	Occurrence of the multiple column SELECT of the table VIEW

	A.order_num	Column name ORDER_NUM qualified by the correlation name A

Table 10: Names in SESAM/SQL

3.4.3 Defining names

The name of an object is usually defined when the object itself is defined using the appropriate SQL statement. The name has then been introduced and the object can be referenced using this name in any subsequent statements.

The table below illustrated how the various names can be defined or declared:

SQL object	SQL statement or part of statement
database (catalog)	CREATE CATALOG (utility statement)
Schema	CREATE SCHEMA
TABLE Base table View Correlation	 CREATE TABLE CREATE VIEW Table specification in query expression
Column	CREATE TABLE, ALTER TABLE CREATE VIEW, query expression
Integrity constraint	CONSTRAINT clause in CREATE TABLE, ALTER TABLE
Index	CREATE INDEX
Routine Procedure User Defined Function (UDF)	 CREATE PROCEDURE CREATE FUNCTION
Storage group	CREATE STOGROUP
Space	CREATE SPACE
Authorization identifier	CREATE USER
cursor	DECLARE CURSOR
statement identifier	PREPARE

Table 11: Defining names

4 Data types and values

This chapter is subdivided into the following sections:

- Overview
- Data types
- Values
- Assignment rules

It has two parts. After an overview of the SESAM/SQL data types and their corresponding range of values, the first part provides you with all the information you need to know about data types with regard to defining table columns:

- syntax
- range of values defined by the data type
- Compatibility between data types

> i In routines, the routine parameters and the local variables also have a data type.

The second part provides you with all the information you need for using the values of a data type:

- syntax of the literals
- rules for entering the values in table columns, routine parameters, and local variables
- rules for using values in expressions and search conditions
- rules governing data type compatibility and conversion during assignment

4.1 Overview of data types and the associated value ranges

The values, or data, that a table contains must lie within a specific range of values. The range of values is determined by the data type.

4.1.1 Data type groups

SESAM/SQL supports the following data types:

- Strings:
  - Alphanumeric data types:
    - CHARACTER
    - CHARACTER VARYING

> i In the SESAM/SQL suite of manuals the term “alphanumeric” expresses the affiliation to an EBCDIC character set, e.g. alphanumeric data type, alphanumeric value, alphanumeric literal. The short forms CHAR and VARCHAR are used in this manual for the alphanumeric data types.

  - National data types:
    - NATIONAL CHARACTER
    - NATIONAL CHARACTER VARYING

> i In the SESAM/SQL suite of manuals the term “national” expresses the affiliation to a Unicode character set, e.g. national data type, national value, national literal. The short forms NCHAR and NVARCHAR are used in this manual for the national data types.

- Numeric data types
  - Integer data types:
    - SMALLINT
    - INTEGER
  - Fixed-point number data types:
    - NUMERIC
    - DECIMAL
  - Floating-point number data types:
    - REAL
    - DOUBLE PRECISION
    - FLOAT
- Time data types:
  - DATE
  - TIME
  - TIMESTAMP

4.1.2 Range of values

Each data type defines a corresponding range of values. Like the data type groups, there are alphanumeric values, national values, numeric values and time values. There are also NULL values (see section “NULL value”).

Appropriate literals and rules on how the values can be used exist for these values. These are described in the section “Values”.

4.1.3 Column

The rows in a table are divided into columns. Each column has a name and data type.

SESAM/SQL distinguishes between atomic and multiple columns.

In an atomic column, exactly one value can be stored in each row.

In a multiple column, several values of the same type can be stored in each row. A multiple column is made up of a number of column elements. In the case of a single column, a single value is stored for each row. The value of a column element is called an **occurrence**. The value of a multiple column is called an **aggregate**. An aggregate is made up of the occurrences of the individual column elements.

A column element is referenced within the multiple column using its position number. Contiguous subareas of a multiple column are specified using the position numbers of the first and last column elements in the subarea.

Example

X[2] or X(2)

Second column element of the multiple column X

X[4..7] or X(4..7)

Subarea consisting of column elements 4, 5, 6, and 7 of the multiple column X

4.1.4 Parameters of routines and local variables

In routines, parameters and local variables can be used. Parameters and local variables have a name and a data type. In contrast to columns, they cannot be multiple.

4.2 Data types

You must specify a data type for each column in a table when you define the columns with CREATE TABLE or ALTER TABLE. The data type defines the type of values that you can enter in the column. After you have defined a table, you can use ALTER TABLE to a certain extent to change the existing data type.

BLOBs (Binary Large Objects) are based on existing data types in SESAM/SQL and are therefore not a new data type in themselves. Information on their structure and how to use them can be found in the chapter “SESAM-CLI” and in the “ Core manual”.

Excluding the NULL value

If you want to exclude the NULL value for a column, you must specify this when you define the table with CREATE TABLE or ALTER TABLE by including a NOT NULL constraint (see section “Column constraints”).

Multiple columns

All the elements in a multiple column have the same data type. You can use any data type except VARCHAR and NVARCHAR for a multiple column. The dimension of a multiple column indicates the number of elements; it is specified when the data type is assigned and must be between 1 and 255.

4.2.1 Overview of SQL data types

The following overview indicates the syntax for all SQL data types used in column definitions:

```
datentyp ::=
{
    [{[ dimension ] | ( dimension )}] CHAR[ACTER][( length )] |
    { CHAR[ACTER] VARYING | VARCHAR } ( max ) |
    [{[ dimension ] | ( dimension ) }] { NATIONAL CHAR[ACTER] | NCHAR }[( cu_length
    [CODE_UNITS])] |
    { NATIONAL CHAR[ACTER] VARYING | NCHAR VARYING | NVARCHAR } ( cu_max [CODE_UNITS]
    ) |
    [{[ dimension ] | ( dimension )}]
    {
        SMALLINT |
        INT[EGER] |
        NUMERIC [( precision [, scale ])] |
        DEC[IMAL][( precision [, scale ])] |
        REAL |
        DOUBLE PRECISION |
        FLOAT [( precision )] |
        DATE |
        TIME(3) |
        TIMESTAMP(3)
    }
}
```

i Any square brackets shown here in italics are special characters, and must be specified in the statement.

The data types are described in the order in which they are listed in the overview.

4.2.2 Alphanumeric and national data types

The alphanumeric and national data types are described in the following sections.

4.2.3 CHARACTER - String with a fixed length

You use the data type CHARACTER or CHAR for columns that can store alphanumeric values of a fixed length (see section “Alphanumeric literals”).

```
[{[ dimension ] | ( dimension )}] CHAR[ACTER][( length )]
```

dimension

Unsigned integer between 1 and 255. The column is a multiple column; *dimension* indicates the number of column elements. *dimension* can be enclosed in square brackets or parentheses.

> **i** Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

length

Unsigned integer between 1 and 256 that indicates the length of the CHAR column.

length omitted:
length=1.

Range of values for CHAR columns

A CHAR column can contain alpnanumeric values of the length specified for the column.

Example

The CUSTOMERS table contains 6 CHAR columns of varying lengths.
The values that the columns can store are alphanumeric strings with a length of 3, 25, 40 and 50 respectively:

```
company      CHAR(40) NOT NULL
street       CHAR(40)
city         CHAR(40)
country      CHAR(3)
cust_tel     CHAR(25)
cust_info    CHAR(50)
```

4.2.4 CHARACTER VARYING - String with a variable length

You use the data type CHARACTER VARYING or VARCHAR for columns that can store alphanumeric values of a variable length (see section "Alphanumeric literals").

```
{ CHAR[ACTER] VARYING( max ) | VARCHAR( max ) }
```

max

Unsigned integer between 1 and 32 000 that defines the maximum length of the VARCHAR column.

Range of values for VARCHAR columns

A VARCHAR column can contain alphanumeric values of any length that are less than or equal to the specified maximum length.

Example

You define a VARCHAR column description that can store alphanumeric values with a maximum length of 1000 characters as follows:

```
description VARCHAR(1000)
```

4.2.5 NATIONAL CHARACTER - Strings with a fixed length

The data type NATIONAL CHARACTER or NCHAR is used for columns which can contain fixed-length national values (see the section “National literals”).

```
[ {[ dimension ] | ( dimension )] } { NATIONAL CHAR[ACTER] | NCHAR } [ ( cu_length
[CODE_UNITS])]
```

dimension

Unsigned integer between 1 and 255. The column is a multiple column; *dimension* indicates the number of column elements. *dimension* can be enclosed in square brackets or parentheses.

> i Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

cu_length

Unsigned integer between 1 and 128 that defines the length of the NCHAR column in code units.

cu_length omitted:
cu_length=1.

> i In SESAM/SQL the encoding form UTF-16 in which each code unit consists of 2 bytes is used for Unicode strings.

Range of values for NCHAR columns

An NCHAR column can contain national values of the length specified for the column.

Example

The MANUALS table contains one INTEGER and two NCHAR columns of fixed length. The values which the NCHAR columns can contain are national strings of the length 20 or 30:

```
ord_num        INTEGER
language       NCHAR(20)
title          NCHAR(30)
```

4.2.6 NATIONAL CHARACTER VARYING - Strings with a variable length

The data type NATIONAL CHARACTER VARYING or NVARCHAR is used for columns which can contain national values (see the section “National literals”) with a variable length.

```
{ NATIONAL CHAR[ACTER] VARYING | NCHAR VARYING | NVARCHAR }( cu_max [CODE_UNITS])
```

cu_max

Unsigned integer between 1 and 16000 that defines the maximum length of the NVARCHAR columns in code units.

> i In SESAM/SQL the encoding form UTF-16 in which each code unit consists of 2 bytes is used for Unicode strings.

Range of values for NVARCHAR columns

An NVARCHAR column can contain national values of any length which are less than or equal to the specified maximum length.

Example

You define an NVARCHAR column description_in_Greek which can contain national values with a maximum length of 1000 characters as follows:

```
description_in_Greek NVARCHAR(1000)
```

4.2.7 Numeric data types

The numeric data types are described in the following sections.

4.2.8 SMALLINT - Small integer

You use the data type SMALLINT for columns that can store small integers (see section “Numeric values”).

```
[{[ dimension ] | ( dimension )}] SMALLINT
```

dimension

Unsigned integer between 1 and 255. The column is a multiple column;
dimension indicates the number of column elements.

> Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

Range of values for SMALLINT columns

The range of values for a SMALLINT column is -2^{15} to $2^{15}-1$.

Example

You define a SMALLINT columns quantity as follows:

```
quantity SMALLINT
```

4.2.9 INTEGER - Integers

You use the data type INTEGER for columns that can store large integers (see section “Numeric values”).

```
[{[ dimension ] | ( dimension )}] INT[EGER]
```

dimension

Unsigned integer between 1 and 255. The column is a multiple column;
dimension indicates the number of column elements.

i Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

Range of values for INTEGER columns

The range of values for an INTEGER column is -2^{31} to $2^{31}-1$.

Example

The SERVICE table has three INTEGER columns:

```
service_num            INTEGER
order_num              INTEGER NOT NULL
service_total          INTEGER CHECK (service_total > 0)
```

4.2.10 NUMERIC - Fixed-point numbers

You use the data type NUMERIC for columns that can store fixed-point numbers (see section “Numeric values”). Unlike DECIMAL, the internal representation of NUMERIC is more efficient with regard to output to the screen.

[{[*dimension*] | (*dimension*)}] NUMERIC[(*precision* [, *scale*])]

dimension

Unsigned integer between 1 and 255. The column is a multiple column;
dimension indicates the number of column elements.

> **i** Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

precision

Unsigned integer between 1 and 31 that indicates the total number of significant digits.

precision omitted:
precision=1.

scale

Unsigned integer between 0 and *precision* that indicates the number of digits to the right of the decimal point.

scale omitted:
scale=0.

Range of values for NUMERIC fixed-point columns

A NUMERIC fixed-point column can store fixed-point numbers whose value is 0 or ranges

from $10\ -^{scale}$ to 10^{scale} to $10^{precision-scale}-10^{-scale}$.

Example

The SERVICE table has three NUMERIC fixed-point columns:

```
service_price       NUMERIC(5,0)
vat                 NUMERIC(2,2)
inv_num             NUMERIC(4,0)
```

The vat column contains fixed-point numbers with two digits to the right of the
decimal point and no digits (that are not equal to null) to the left of the decimal point.

4.2.11 DECIMAL - Fixed-point numbers

You use the data type DECIMAL for columns that can store fixed-point numbers (see section “Numeric values”).

Unlike NUMERIC, the internal representation of DECIMAL is shorter and more efficient for calculation purposes.

```
[{[ dimension ] | ( dimension )}] DEC[IMAL][( precision [ , scale ])]
```

dimension

Unsigned integer between 1 and 255. The column is a multiple column;
dimension indicates the number of column elements.

> Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

precision

Unsigned integer between 1 and 31 that indicates the total number of significant digits.

precision omitted:
precision=1.

scale

Unsigned integer between 0 and *precision* that indicates the number of digits to the right of the decimal point.

scale omitted:
scale=0.

Range of values for DECIMAL fixed-point columns

A DECIMAL fixed-point column can contain fixed-point numbers whose value is 0 or ranges

from 10^{-scale} to $10^{precision-scale}-10^{-scale}$.

Example

You define a DECIMAL column weight with six digits to the left of the decimal point and two digits to the right of the decimal point as follows:

```
weight DECIMAL(8,2)
```

4.2.12 REAL- Single-precision floating-point numbers

You use the data type REAL for columns that can store single-precision floating-point numbers (see section “Numeric values”).

```
[{[ dimension ] | ( dimension )}] REAL
```

dimension

Unsigned integer between 1 and 255. The column is a multiple column; *dimension* indicates the number of column elements.

> i Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

Range of values for REAL columns

A REAL column can contain floating-point numbers whose value is 0 or ranges from $5.4E^{-79}$ to $7.2E^{+75}$.

The precision of REAL floating-point numbers is 21 binary digits, which is approximately 6 decimal digits.

Example

You define a REAL column weight as follows:

```
weight REAL
```

4.2.13 DOUBLE PRECISION - Double-precision floating-point numbers

You use the data type DOUBLE PRECISION for columns that can store double-precision floating-point numbers (see section “Numeric values”).

```
[{[ dimension ] | ( dimension )}] DOUBLE PRECISION
```

dimension

Unsigned integer between 1 and 255. The column is a multiple column;
dimension indicates the number of column elements.

> Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

Range of values for DOUBLE PRECISION columns

A DOUBLE PRECISION column can contain floating-point numbers whose value is 0

or ranges from 5.4E $^{-79}$ to 7.2E $^{+75}$.

The precision of DOUBLE PRECISION floating-point numbers is 53 binary digits or approximately 16 decimal digits.

Example

You define a DOUBLE PRECISION column weight as follows:

```
weight DOUBLE PRECISION
```

4.2.14 FLOAT - Floating-point numbers

You use the data type FLOAT for columns that can store floating-point numbers (see section “Numeric values”). The precision can be specified.

```
[{[ dimension ] | ( dimension )}] FLOAT[( precision )]
```

dimension

Unsigned integer between 1 and 255. The column is a multiple column; *dimension* indicates the number of column elements.

> i Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

precision

Unsigned integer between 1 and 53 that indicates the minimum number of binary digits for the mantissa.

precision omitted:
precision=1.

Range of values for FLOAT columns

A FLOAT column can contain a floating-point number whose value is 0 or ranges from

$5.4E^{-79}$ to $7.2E^{+75.}$

In SESAM/SQL, the precision of FLOAT floating-point numbers is 53 binary digits if *precision* is greater than 21, otherwise it is 21 binary digits.

Example

You define a FLOAT column test_value with a precision of at least 30 binary digits as follows:

```
test_value FLOAT(30)
```

4.2.15 Time data types

The date and time data types are described in the following sections.

4.2.16 DATE

You use the data type DATE for columns that can store a date (see section “Time values”).

[{[*dimension*] | (*dimension*)}] DATE

dimension

Unsigned integer between 1 and 255. The column is a multiple column; *dimension* indicates the number of column elements.

i Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

Range of values for DATE columns

A DATE column can contain date specifications lying in the range 0001-01-01 to 9999-12-31. The date specification must observe the rules of the Gregorian calendar even if the date involved is before the introduction of the Gregorian calendar.

Example

The ORDERS table contains three DATE columns:

```
order_date     DATE DEFAULT CURRENT_DATE
actual         DATE
target         DATE
```

4.2.17 TIME

You use the data type TIME for columns that can store a time (see section “Time values”).

[{[*dimension*] | (*dimension*)}] TIME(3)

dimension

Unsigned integer between 1 and 255. The column is a multiple column;
dimension indicates the number of column elements.

> Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

Range of values for TIME columns

A TIME column can contain times that lie within the range 00:00:00.000 to 23:59:61.999. The range for seconds (00.000 to 61.999) allows you to specify up to two leap seconds.

Example

You define a TIME column wakeup_time as follows:

```
wakeup_time TIME(3)
```

4.2.18 TIMESTAMP

You use the data type TIMESTAMP for columns that can store a time stamp (see section “Time values”).

```
[{[ dimension ] | ( dimension )}] TIMESTAMP(3)
```

dimension

Unsigned integer between 1 and 255. The column is a multiple column;
dimension indicates the number of column elements.

> i Any square brackets shown here in italics are special characters, and must be specified in the statement.

dimension omitted:
The column is an atomic column.

Range of values for TIMESTAMP columns

A TIMESTAMP column can contain dates that lie within the range 0001-01-01 to 9999-12-31 and times that lie within in the range 00:00:00.000 to 23:59:61.999.

The range for seconds (00.000 to 61.999) allows you to specify up to two leap seconds. The date specification must observe the rules of the Gregorian calendar even if the date involved is before the introduction of the Gregorian calendar.

Example

You define a TIMESTAMP column appointment as follows:

```
appointment TIMESTAMP(3)
```

4.2.19 Compatibility between data types

If values are used in calculations, predicates and assignments, the data types of the operands involved must be compatible.

Two data types are compatible if they fulfill the following conditions:

- Both data types are CHAR or VARCHAR.
- Both data types are NCHAR or NVARCHAR.
- Both data types are numeric (SMALLINT, INTEGER, NUMERIC, DECIMAL, REAL, DOUBLE PRECISION or FLOAT).
- Both data types are DATE.
- Both data types are TIME.
- Both data types are TIMESTAMP.

Values from various character sets are not converted implicitly in SESAM/SQL to make them compatible. Transliteration of strings is possible with the TRANSLATE function, see the section “TRANSLATE() - Transliterate / transcode string”.

4.3 Values

Values are specified in SESAM/SQL statements for the following purpose:

- insert or update column values (INSERT, MERGE, UPDATE)
- perform calculations and comparisons (e.g. SELECT column selection, HAVING, ON and WHERE search conditions)

SESAM/SQL makes a distinction between NULL values and non-NULL values. Non-NULL values are grouped according to data type.

Therefore, there are the following groups of values:

- NULL values (see section “NULL value”)
- alphanumeric values (see section “Strings”)
- national values (see section “Strings”)
- numeric values (see section “Numeric values”)
- time values (see section “Time values”)

REF values, which occur in conjunction with BLOBs (Binary Large Objects), are special alphanumeric values used to reference BLOBs in base tables. Information on defining REF values in base tables can be found in the section “Column definitions”. Information on their structure and how to use them can be found in the chapter “SESAM-CLI” and in the “ Core manual”.

4.3.1 Literals

With the exception of NULL values, there are corresponding literals for each group of values:

literal ::= { *alphanumeric_literal* | *national_literal* | *special_literal* | *numeric_literal* | *time_literal* }

alphanumeric_literal

Alphanumeric literal (see section “Alphanumeric literals”).

national_literal

National literal (see section “National literals”).

special_literal

Special literal (see section “Special literals”).

numeric_literal

Numeric literal (see section “Numeric literals”).

time_literal

Time value (see section “Time literals”).

4.3.2 Specifying values

A value can be specified in the following ways:

- as a literal
- with a user variable when the statement is **not** part of a routine (see section "Host variables")
- with a parameter (see "CREATE PROCEDURE - Create procedure") or a local variable (see "COMPOUND - Execute SQL statements in a common context") when the statement is part of a routine
- with a placeholder "?" for values which are not yet known
  (in a dynamic statement or cursor description, see section "Dynamic SQL")

---

value ::=

{

literal |

: *host_variable* [[INDICATOR] : *indicator_variable*] |

routine_parameter |

local_variable |

?

}

---

literal

Alphanumeric literal, national literal, special literal, numeric literal or time literal.

host_variable

Name of the host variable that contains the value.

If you have specified an indicator variable and the value of the indicator variable is negative, the NULL value is used instead of the value of the host variable.

indicator_variable

Name of an indicator variable for the preceding host variable. The data type of *indicator_variable* is SMALLINT.

routine_parameter

Name of a routine's parameter which contains the value.

local_variable

Name of a routine's local variable which contains the value.

? Placeholder in a dynamic SQL statement.

4.3.3 Values for multiple columns

The value for a multiple column is an aggregate. An aggregate consists of one or more elements called occurrences. The number of occurrences must be between 1 and 255 and must correspond to the dimension of the multiple column. Values in multiple columns are referred to as multiple values; values in atomic columns are referred to as atomic values (or simply as values).

aggregate `::= <{` *value* `|` `NULL },...>`

value

Value of the occurrence.

NULL

NULL value for the occurrence.

If you set elements of the multiple column to the NULL value with INSERT or UPDATE and the subsequent elements are not null, the non-NULL values in the multiple columns are moved to smaller position numbers and the NULL values are entered after all the non-NULL values.

Example

You can use INSERT to assign values to the numeric multiple column COLOR_TAB with three elements:

```
INSERT INTO color_tab (rgb(1..3)) VALUES (<0.88,NULL,0.77>)
```

The multiple column then contains the multiple value:

```
<0.88,0.77,NULL>
```

4.3.4 NULL value

NULL values are a special feature of relational databases. A NULL value means a value is undefined or unknown.

The NULL value is different to all other values. Do not confuse it with a string with the length 0, the blank or numeric 0.

4.3.4.1 Keyword for the NULL value

The keyword for the NULL value is NULL. NULL can only be specified during INSERT, MERGE and UPDATE operations, in a CAST expression, in a CASE expression and as the DEFAULT in column definitions to set a column value to the NULL value.

Example

You enter an item whose color is unknown into the ITEMS table as follows:

```
INSERT INTO items VALUES (5, 'Valve', NULL, 1.00, 350, 100)
```

NULL can also be specified in predicates (search queries, IF statement), as the default value of local variables (in routines), and in SET and RETURN statements.

4.3.4.2 NULL value in table columns

You can prohibit use of the NULL value in a column in a base table by specifying one of the following column constraints in the column definition:

- NOT NULL constraint
- PRIMARY KEY constraint
- check constraint that prohibits use of the NULL value

If use of the NULL value is not prohibited, a column can contain the NULL value.

4.3.4.3 NULL value in functions, expressions and predicates

The keyword NULL cannot be specified for values in expressions (except in CASE and CAST expressions), functions and predicates. You can, however, specify subexpressions (for example, a column name) whose result is the NULL value.

If the NULL value occurs in an expression, the result of the expression is also the NULL value.

If the NULL value occurs in a predicate, the result is usually the truth value unknown. There are, however, exceptions such as the predicate IS [NOT] NULL, for example. The result of each function, operator and predicate if an operand is the NULL value can be found in the chapter “Compound language constructs”.

4.3.4.4 NULL value in GROUP BY

If you specify the GROUP BY clause in a SELECT statement, all the rows that contain the NULL value in the same grouping columns and identical values in the rest of the grouping columns are grouped together.

4.3.4.5 NULL value in ORDER BY

If you specify the ORDER BY clause in a cursor description, indicating that a cursor table is to be sorted, NULL values are smaller than all non-NULL values.

4.3.5 Strings

Strings are sequences of any characters in EBCDIC or Unicode. EBCDIC strings are termed “alphanumeric values”, Unicode strings are termed “national values”.

In SESAM/SQL, alphanumeric literals, national literals and special literals are used to represent strings.

4.3.5.1 Alphanumeric literals

The syntax for an alphanumeric literal is defined as follows:

alphanumeric_literal ::=

{

  '[*character* ...]'[*separator* ...'[*character* ...]']... |

  X'[*hex hex*]...'[*separator* ...'[*hex hex*]...']...

}

hex ::= 0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f|A|B|C|D|E|F

character

Any EBCDIC character. If a string contains a single quote ('), you must duplicate this single quote. The pair of single quote characters is considered a single character (e.g. `'Variable length strings are of the type ''CHARACTER VARYING'''`).

hex

A hexadecimal character from the range 0-9, A-F or a-f

The data type of an alphanumeric literal is CHAR(*length*). *length* is the number of characters or pairs of hexadecimal numbers. Alphanumeric literals can be up to 256 characters long. Strings with the length 0 are permitted as literals although it is not possible to define a data type CHAR(0) (see section “Alphanumeric and national data types”). The data type is then VARCHAR(0).

The two forms of alphanumeric literal, *character* and *hex*, may be concatenated, as for instance in the German number “fünfzig” (50, `'f'||x'FD'||'nfzig'`) or in a concatenation with a special literal (`'User:'||CURRENT_USER`).

“||” must be used as the operator for the concatenation.

> i When strings are concatenated, either both operands must be alphanumeric (CHAR or VARCHAR) or both must be of the national type (NCHAR or NVARCHAR), see the section “Compatibility between data types”.

separator

Separator that separates two substrings from each other (see section “Separators”). If an alphanumeric literal consists of two or more substrings, adjacent substrings must be separated by one or more separators. At least one of the separators must be a transition point to the next row.

The result of an alphanumeric literal comprising substrings is the concatenation of the substrings involved without the operator for concatenation having to be written for this purpose.

Example

The following alphanumeric literal consists of three substrings:

```
'Separated '      -- First substring
'by table '       -- Second substring
'and bed'         -- Third substring
```

The result is the string 'Separated by table and bed'.

4.3.5.2 National literals

The syntax for a national literal is defined as follows:

---

national_literal ::=

{

N'[*character* ...]'[*separator* ...'[*character* ...]'] ... |
NX'[*4hex* ...]'[*separator* ...'[*4hex* ...]'] ...|
U&'[*uc-character* ...]'[*separator...* '[*uc-character* '...] ... [UESCAPE' *esc* ']

}

uc-character ::= { *character* | *esc 4hex* | *esc+ 6hex* | *esc esc* }

---

character

A Unicode character which is also contained in the EDF03IRV character set. If a string contains a single quote ('), you must duplicate this single quote. The duplicated single quote counts as **one** character.

4hex

4hex is a group of 4 consecutive hexadecimal characters and constitutes a UTF-16 code unit which must be in the range 0000 through FFFF. (However, the UTF-16 code units FFFE and FFFF and the code units in the range FDD0 - FDEF are so-called noncharacters and may not be used in literals in SESAM/SQL, see the Unicode concept in SESAM/SQL in the “Core Manual”.) When *4hex* is specified, lower case is permitted for the hexadecimal characters A through F.

Example

`NX'004100420043'` for the string 'ABC'.

esc 4hex

Hexadecimal representation of a code point through the escape character *esc* and (without any intervening blank) a 4-digit hexadecimal value *4hex* which must be in the range 0000 through FFFD. The specification *esc* must be written exactly as specified in the UESCAPE clause. When *esc4hex* is specified, lower case is permitted for the hexadecimal characters A through F.

Example

`U&'\00DF'` for the character 'ß’

`U&'\0395\03BB\03BB\03B7\03BD\03B9\03BA\03AC means Greek’`

returns the string “ means Greek”

esc+ 6hex

Hexadecimal representation of a code point through the escape character *esc* followed by „+“and (without any intervening blank) a 6-digit hexadecimal value *6hex* which must be in the range 000000 through 10FFFD. (The code points 10FFFE and 10FFFF and also the code points from the ranges 0xFFFE and 0xFFFF (where x is a hexadecimal number) are so-called noncharacters and may not be used in literals in SESAM/SQL, see the Unicode concept in SESAM/SQL in the “ Core manual”). The specification *esc* must be written exactly as specified in the UESCAPE clause. When *esc+ 6hex* is specified, lower case is permitted for the hexadecimal characters A through F.

Example

`U&'\+0000DF'` for the character 'ß'.

esc esc

With *esc esc* (without any intervening blank) you can invalidate the *esc* character, as a result of which this string represents an *esc* character.

Example

`U&'\\'` for the character '\'

UESCAPE `'` *esc* `'`

Specification of an escape character. *esc* can be any alphanumeric character with the exception of the plus character, double quotes ("), single quote (') and blank.

If UESCAPE `'` *esc* `'` is not specified, the backslash (\) is used as the default.

The data type of a national literal is NCHAR(*cu_length*). *cu_length* is the number of code units (1 code unit in UTF-16 = 2 bytes). The strings may be up to 128 code units long. Strings of which are 0 characters long are permitted as literals, although it is not possible to define a data type NCHAR(0) (see the section “Strings”). The data type is then NVARCHAR(0).

1 code unit is required to represent a code point in UTF-16, except in the case of code points which are contained in the range 010000 through 10FFFD. These code points require two code units.

The various forms of national data type can be concatenated as, for example, in “Price in €”:

```
N'Price in '||NX'20AC'
```

```
N'Price in '||U&'\20AC'
```

“||” must be used as the operator for the concatenation.

i When strings are concatenated, either both operands must be alphanumeric (CHAR or VARCHAR) or both must be of the national type (NCHAR or NVARCHAR), see the section “Compatibility between data types”.

separator

Separator that separates two substrings from each other (see section "Separators"). If a national literal consists of two or more substrings, adjacent substrings must be separated by one or more separators. At least one of the separators must be a transition point to the next row.

The result of a string literal consisting of substrings is the concatenation of the substrings involved without the operator for concatenation having to be written for this purpose.

4.3.5.3 Special literals

The syntax for special literals is as follows:

```
special_literal ::=

{
   CURRENT_CATALOG |
   CURRENT_ISOLATION_LEVEL |
   CURRENT_REFERENCED_CATALOG |
   CURRENT_SCHEMA |
   [CURRENT_]USER |
   SYSTEM_USER
}
```

CURRENT_CATALOG

Name of the database preset with the SQL statement SET CATALOG or SET SCHEMA or the `*IMPLICIT` string if no database is preset.
The result is a string of the type CHAR(18).

CURRENT_ISOLATION_LEVEL

Isolation level of the current transaction (defined implicitly by the user configuration or explicitly by the SQL statement SET TRANSACTION *level* at the beginning of a transaction). It does not specify the isolation level which is defined on a statementspecific basis with the pragma ISOLATION LEVEL.
The result is a value of the type INTEGER in accordance with the table below: .

Result	Isolation level	Consistency levels
8	SERIALIZABLE	4
4	REPEATABLE READ	3
5	READ NO WAIT	1
2	READ COMMITTED	2
1	READ UNCOMMITTED	0

CURRENT_REFERENCED_CATALOG

Name of the database to which the current statement refers.
The result is a string of the type CHAR(18).

CURRENT_SCHEMA

Name of the schema preset with the SQL statement SET SCHEMA or the `*IMPLICIT` string if no schema is preset.
The result is an alphanumeric string of the type VARCHAR(31).

[CURRENT_]USER

Name of the current authorization identifier.
The result is a string of the type CHAR(18).

SYSTEM_USER

Name of the current system user. The name is made up of the host name, the UTM application name (or blanks) and the UTM or BS2000 user ID.
The result is a string of the type CHAR(24).

4.3.5.4 Using strings

An alphanumeric or a national value can be used in:

- Assignments:
  (see section “Assignment rules”)
- Functions:
  An alphanumeric or a national value can be used in the aggregate functions COUNT(), MIN() and MAX(), in numeric functions and in string functions.
- Concatenation:
  Two alphanumeric values can be concatenated to create a single alphanumeric value; two national values can be concatenated to create a single national value. See section “Compatibility between data types”.
- Predicates:
  An alphanumeric or a national value can be used in comparisons with another value or with a derived column, in range queries, in element queries and in pattern comparisons. All the values concerned must be either alphanumeric values or national values, see the section “Compatibility between data types”. The rules governing comparisons are described in the section “Comparison of two rows”.

Functions, expressions and predicates are described in detail in the chapter “Compound language constructs”.

Alphanumeric literals in the form X'...' must not be used in SET CATALOG, SET SCHEMA, SET SESSION AUTHORIZATION statements or in the GLOBAL *descriptor*.

Examples

Enter first and last name in the CUSTOMERS table:

```
INSERT INTO customers (cust_num, company, street, zip)

VALUES (100,'Siemens AG','Otto-Hahn-Ring 6',81739)
```

```
INSERT INTO customers (cust_num, company, street, zip)

VALUES (100,Siemens AG,"Otto-Hahn-Ring 6",81739)
```

This is an error: strings must be enclosed in single quotes.

Search for the names of the tables, the authorization identifiers and the privileges for which the current authorization identifier has a table privilege:

```
CREATE VIEW privileged AS SELECT TABLE_NAME, GRANTEE, PRIVILEGE_TYPE

FROM INFORMATION_SCHEMA.TABLE_PRIVILEGES WHERE GRANTOR = UTIUNIV
```

Define the table BOOKS with the VARCHAR column TITLE and enter values:

```
CREATE TABLE books (order_number INTEGER, title VARCHAR(50))
COMMIT WORK
INSERT INTO books VALUES (3456, 'Not Now Bernard')
INSERT INTO books VALUES (5777, 'Lullabies')
```

```
INSERT INTO books VALUES (7888,
'This is a very long title with more than fifty characters')
```

The last title is not entered. An error message is issued.

Enter additional information on the contact person Mary Davis in the CONTACTS table:

```
UPDATE contacts set contact_info=('Ms. Davis is '
'on leave from ’
'1.8 to 31.10') where contact_num=40
```

The following is incorrect:

```
UPDATE contacts set contact_info=
('Ms. Davis is ' 'on leave ' 'from 1.8 to 31.10')
where contact_num=40
```

At least one of the separators between the substrings must be a transition to the next line.

Comparing strings

`' Mai' < ' Maier'` is true

`' Majer' < ' Maier'` is falsch

Define the MANUALS table with the NCHAR columns LANGUAGE and TITLE and enter

```
CREATE TABLE manuals
(ord_num INTEGER, language NCHAR(20), title NCHAR(30))
COMMIT WORK
INSERT INTO manuals
VALUES (1001, N'Deutsch', N'Betriebsanleitung'),
       (1002, N'English', N'Operating Manual'),
       (1003, U&'Fran\00E7ais', N'Manuel d''utilisation'),
       (1004, U&'Espa\00F1ol', N'Manual de instrucciones'),
       (1005, N'Italiano', N'Istruzioni per l''uso'),
       (1006, NX'039503BB03BB03B703BD03B903BA03AC',
              NX'039F03B403B703B303AF03B503C2002003BB'
                '03B503B903C403BF03C503C103B303AF03B103C2')
```

The LANGUAGE and TITLE titles then contain the following national values:

```
LANGUAGE        TITLE
Deutsch         Betriebsanleitung
English         Operating Manual
Français        Manuel d'utilisation
Español         Manual de instrucciones
Italiano        Istruzioni per l'uso
```

4.3.6 Numeric values

Numeric values are integers, fixed-point numbers and floating-point numbers.

4.3.6.1 Numeric literals

The syntax for numeric literals is defined as follows:

```
numeric_literal ::= { integer | fixed_pt_number | floating_pt_number }

integer ::= [{+|-}] unsigned_integer [.]

fixed_pt_number ::= [{+|-}]

{
unsigned_integer [.unsigned_integer] |
unsigned_integer. |
.unsigned_integer

}

floating_pt_number ::= fixed_pt_number E[{+|-}] unsigned_integer

unsigned_integer ::= digit...
```

digit

Decimal digit 0 to 9.
Integers and fixed-point literals can have up to 31 digits.

The data type of the literal is integer, fixed-point number or floating-point number with the specified number of digits to the right and left of the decimal point.

4.3.6.2 Using numeric values

A numeric value can be used in:

- Assignments:
  (see section “Assignment rules”)
- Aggregate functions:
  A numeric value can be used in the aggregate functions AVG(), COUNT(), MIN(), MAX() and SUM().
- Time functions:
  A numeric value can be used in the time function DATE_OF_JULIAN_DAY()
- Expressions:
  A numeric value can be used in calculations with the operators +, -, * and /. All the values in the expression must be numeric.
- Predicates:
  A numeric value can be used in comparisons with another value or with a derived column, in range queries and in element queries.
  All the values in the expression must be numeric. The rules governing comparisons are described in the section “Comparison of two rows”.

Functions, expressions and predicates are described in detail in the chapter “Compound language constructs”.

Examples

The following examples refer to the SERVICE table.

Enter an order number as follows:

```
INSERT INTO service (service_num, order_num, service_total, service_price)
VALUES (5000, 250, 1, NULL)
```

Update the order quantity:

```
UPDATE service SET service_total=34.75 WHERE service_num=5000
```

The specified value is converted into an integer.

```
UPDATE service SET service_total='lots' WHERE service_num=5000
```

This is an error: The specified value is not numeric.

4.3.7 Time values

SESAM/SQL makes a distinction between the following types of time values:

- Date A date consists of the specifications: year, month and day.
- Time A time consists of the specifications: hours, minutes, seconds and fractions of a second.
- Timestamp A time stamp contains a date and time.

4.3.7.1 Time literals

The syntax for time literals is defined as follows:

time_literal ::=

{

DATE ' *year-month-day* ' |

TIME ' *hour:minute:second* ' |

TIMESTAMP' *year-month-day hour:minute:second* '

}

DATE

Date. The data type of the time literal is DATE.

TIME

Time. The data type of the time literal is TIME(3).

TIMESTAMP

Timestamp. The data type of the time literal is TIMESTAMP(3).

year

Four-digit unsigned integer between 0001 and 9999 indicating the year.

month

Two-digit unsigned integer between 01 and 12 indicating the month.

day

Two-digit unsigned integer between 01 and 31 (corresponding to the month and year) indicating the day.

hour

Two-digit unsigned integer between 00 and 23 indicating the hour.

minute

Two-digit unsigned integer between 00 and 59 indicating the minute.

second

Unsigned fixed-point number between 00.000 and 60.999 that indicates the seconds and fractions of a second. A two-digit specification must be made for the seconds and a three-digit specification for the fractions of a second.

The range of values allows specification of one leap second.

A date specification must observe the rules of the Gregorian calendar even if the date involved is before the introduction of the Gregorian calendar.

In SESAM/SQL, you can use an abbreviated notation without an introductory time keyword if it is clear from the context that you are dealing with a time literal and not an alphanumeric literal.

Examples

To output, from the ORDERS table, all orders which were completed before the specified date.

```
SELECT * FROM orders WHERE actual < '2013-01-01'
```

The actual column was defined with the DATE data type during table creation. It is therefore immediately obvious from the left-hand comparison operand that the specified literal is a time literal. The keyword DATE can therefore be omitted on the right-hand side.

Literal in the SELECT list.

```
SELECT COUNT(*) AS number, '2013-05-01' AS date FROM orders
```

The derived table contains a row with the number of orders and with the DATE column. The data type results from the specified expression. The data type for the DATE column is therefore CHAR(10).

To avoid possible sources of error, you are recommended to always specify time literals with an introductory time keyword (DATE, TIME, TIMESTAMP).

! **CAUTION!**The separators between the component values must be specified exactly as stated below: hyphen “-” between year, month and dayblank “ ” between day and hourcolon “:” between hour, minutes and secondsperiod “.” between seconds and fractions of a second.

4.3.7.2 Using time values

A time value can be used in:

- Assignments:
  (see section “Assignment rules”)
- Aggregate functions:
  A time value can be used in the aggregate functions COUNT(), MIN() and MAX().
- Numeric functions:
  A time value can be used in the numeric function JULIAN_DAY_OF_DATE().
- Predicates:
  A time value can be used in comparisons with another value or with a derived column, in range queries and in element queries. All the values involved must be of the same time data type. The rules governing comparisons are described in the section “Comparison of two rows”.
- CAST expressions:
  A time value can be converted to a value of a different data type.

Functions and predicates are described in detail in the chapter “Compound language constructs”.

Examples

The following examples refer to the ORDERS table and the fictitious table EXAMPLE.

Update the delivery date for order 300:

```
UPDATE orders SET order_date=DATE'2013-10-06' WHERE order_num=300
```

```
UPDATE orders SET order_date=DATE'2013-10-06' WHERE order_num=300
```

The last one is incorrect: Since the single-digit value 6 for a day is not permitted. The correct specification would be 06.

In the column wakeup_time, the time 7:51 hours and 19.77 seconds is entered:

```
CREATE TABLE example (wakeup_time TIME (3), appointment TIMESTAMP (3))
```

```
INSERT INTO example (wakeup_time) VALUES (TIME'07:51:19.770')
```

In the column appointment, the time stamp 16:00 hours on November 24th, 2010 is entered:

```
INSERT INTO example (appointment) VALUES (TIMESTAMP'2013-10-06 16:00:00.000')
INSERT INTO example (appointment) VALUES (TIMESTAMP'2013-10-06 16:00')
```

The last one is incorrect as seconds have not been specified.

4.4 Assignment rules

When values are assigned or transferred, the source data type and the target data type must be compatible (see section "Compatibility between data types").

Other rules depend on where the values are being transferred to or from.

A distinction is made between the following:

- Entering values in table columns
- Default values for table columns
- Values for placeholders
- Storing values in host variables or a descriptor area
- Transferring values between host variables and a descriptor area
- Modifying the target data type by means of the CAST operator
- Supplying input parameters for routines
- Entering values in a procedure parameter (output) or local variable

The following sections provide you with an overview of the assignment rules for the abovementioned cases.

4.4.1 Entering values in table columns

The following rules apply when inserting or updating values into table columns with INSERT, MERGE or UPDATE:

- Atomic values and multiple values with the dimension 1 can be entered in atomic columns and in multiple columns (or subareas) with the dimension 1.
- Multiple values with a dimension greater than 1 can be entered in multiple columns (or subareas) with the same dimension.
- Additional data-type-specific rules, which depend on the data type involved, also apply. These are described below.

Strings

You can enter an alphanumeric value in a column with an alphanumeric data type or a national value in a column with a national data type. The following rules apply:

- If the target data type is CHAR or NCHAR and the length of the value is smaller than the length of the target data type, the value is padded on the right with blanks.
- If the target data type is CHAR or NCHAR and the length of the value is greater than the length of the target data type, the value is truncated from the right to the length of the target data type. If characters are removed that are not blanks, the value is not entered and an error message is issued.
- If the target data type is VARCHAR or NVARCHAR and the length of the value is greater than the maximum length of the target data type, the value is truncated from the right to the maximum length of the target data type. If characters are removed that are not blanks, the value is not entered and an error message is issued.

Numeric values

You can enter a numeric value in a column with a numeric data type. If the numeric data types are not the same, the value is converted to the data type of the column. The following rules apply:

- If the number of digits to the right of the decimal point of the value is too large for the data type of the column, the value is rounded.
- If the value is too large for the data type of the column, the value is not entered and an error message is issued.

Time values

You can only enter a time value in a column with the same data type:

- a date in a DATE column
- a time in a TIME column
- a time stamp in a TIMESTAMP column

4.4.2 Default values for table columns

The rules that apply to the default value for a column that you can specify with the DEFAULT clause of the CREATE TABLE or ALTER TABLE statement are more strict than those for entering values in table columns. The rules also apply for the definition of local variables (in routines). They are contained in the table below:

SQL data type of the column	Possible SQL default value
CHAR(*length*) VARCHAR(*max*)	• Alphanumeric literal with length <= *length* or *max* • Special literal ([CURRENT_]USER and SYSTEM_USER only (only recommended for *length* or *max* <= 128)) • NULL
NCHAR(*cu_length*) NVARCHAR(*cu_max*	• National literal with length >= *cu_length* or *cu_max* • NULL
REF(*table*)	• As for CHAR(237)
DECIMAL(*precision,scale*) NUMERIC(*precision,scale*) INTEGER SMALLINT	• Fixed-point or floating-point number belonging to the range of values for the column • NULL
REAL, DOUBLE PRECISION FLOAT(*precision*)	• Numeric literal (the number is rounded off if necessary) • NULL
DATE	• Literal of the type DATE • CURRENT_DATE • NULL
TIME(3)	• Literal of the type TIME(3) • CURRENT_TIME • NULL
TIMESTAMP(3)	• Literal of the type TIMESTAMP(3) • CURRENT_TIMESTAMP • NULL

Table 12: Default values for table columns

4.4.3 Values for placeholders

The following rules apply if values are made available for placeholders in host variables or in a descriptor area (EXECUTE...USING, OPEN...USING):

- The data type of the input value must be compatible with the data type of the placeholder, which is indicated by the position of the placeholder (see “Rules for placeholders”).
- Values for atomic placeholders and multiple placeholders with the dimension 1 can be made available via an atomic host variable, a vector with one element, or via an item descriptor.
- Placeholders for aggregates with a dimension *d* > 1 can be made available via a vector with *d* elements or via *d* sequential item descriptors.
- Additional data-type-specific rules, which depend on the data type involved, also apply. These are described below.

Strings

You can use the value of a host variable or item descriptor with an alphanumeric data type for an alphanumeric placeholder. For a placeholder with a national data type you can use the value from a user variable or a descriptor area entry with a national data type. The following rules apply:

- If the target data type is CHAR or NCHAR and the length of the value is smaller than the length of the target data type, the value is padded on the right with blanks.
- If the target data type is CHAR or NCHAR and the length of the value is greater than the length of the target data type, the value is truncated from the right to the length of the target data type. If characters are removed that are not blanks, the value is not entered and a warning is issued.
- If the target data type is VARCHAR or NVARCHAR and the length of the value is greater than the maximum length of the target data type, the value is truncated from the right to the maximum length of the target data type. If characters are removed that are not blanks, the value is not entered and a warning is issued.

Numeric values

You can use a value from a host variable or an item descriptor with a numeric data type for a numeric placeholder. If the numeric data types are not the same, the value is converted to the target data type. The following rules apply:

- If the number of digits to the right of the decimal point of the value is too large for the target data type, the value is rounded.
- If the value is too large for the target data type, the value is not entered and an error message is issued.

Time values

In the case of a placeholder with a date or time data type, you can only use a value from a host variable or item descriptor of the same data type:

- a date for a DATE placeholder
- a time for a TIME placeholder
- a time stamp for a TIMESTAMP placeholder

4.4.4 Reading values into host variables or a descriptor area

The following rules apply if values from table columns or output parameters of a routine are stored in a host variable or in a descriptor area (SELECT...INTO, EXECUTE...INTO, FETCH...INTO, INSERT...RETURN INTO, CALL):

- Values from atomic columns, multiple columns with the dimension 1 or output parameters of a procedure can be stored in an atomic host variable, a vector with one element, or in an item descriptor.
- Aggregates from multiple columns with a dimension *d* > 1 can be stored in a vector with *d* elements or in *d* sequential item descriptors.
- If the value to be transferred is a NULL value, the indicator variable or item descriptor field INDICATOR, as appropriate, is set to -1. If no indicator variable has been specified for a host variable, an error message is issued.
- Depending on the data type, data-type-specific rules which are contained below also apply.

Strings

You can read an alphanumeric column value or an alphanumeric output parameter of a procedure into an alphanumeric host variable or item descriptor. You can read a national column value or a national output parameter of a procedure into a national host variable or item descriptor with a national data type. The following rules apply:

- If the target data type is CHAR or NCHAR and the length of the value is smaller than the length of the target data type, the value is padded on the right with blanks.
- If the target data type is CHAR or NCHAR and the length of the value is greater than the length of the target data type, the value is truncated from the right to the length of the target data type and a warning is issued. The indicator variable (if specified) or item descriptor field INDICATOR, as appropriate, is set to the original length of the column value.
- If the target data type is VARCHAR or NVARCHAR and the length of the value is greater than the maximum length of the target data type, the value is truncated from the right to the maximum length of the target data type and a warning is issued. The indicator variable (if specified) or item descriptor field INDICATOR, as appropriate, is set to the original length of the column value.

Numeric values

You can read a numeric column value or a numeric output parameter of a procedure into a numeric host variable or item descriptor. If the numeric data types are not the same, the value is converted to the target data type. The following rules apply:

- If the number of digits to the right of the decimal point of the value is too large for the target data type, the value is rounded.
- If the value is too large for the target data type, the value is not entered and an error message is issued.

Time values

You can only read a column value with a time data type or an output parameter of a procedure with a time data type into a host variable or item descriptor of the same data type:

- a date into a DATE host variable or item descriptor
- a time into a TIME host variable or item descriptor
- a time stamp into a TIMESTAMP host variable or item descriptor

4.4.5 Transferring values between host variables and a descriptor area

The rules governing the transfer of values between host variables an a descriptor area are more strict than those for transferring values between host variables (or descriptor area) and table columns:

- The following applies to all fields except NAME and DATA: The SQL data type of the host variable in which the value of a field is stored or from which a value is read must be SMALLINT.
- If the value of the NAME field is read, the host variable must be of the type CHAR(*n*) or VARCHAR(*n*) where *n* >= 128.
- If the value of the DATA field is stored in a host variable or read from a host variable, the SQL data type of the host variable must match the data type described by the fields TYPE, DATETIME_INTERVAL_CODE, LENGTH, PRECISION and SCALE of the same item descriptor. The rules are contained below in accordance with the data type.

Strings

The length of the host variable must be the same as the value in the item descriptor field LENGTH for the SQL data types CHAR and NCHAR.

In the case of the SQL data types VARCHAR and NVARCHAR, the maximum length of the host variable must be the same as the value of the item descriptor field LENGTH if the value is to be transferred from the host variable to the descriptor area. If the value is transferred from the descriptor area to the host variable, the maximum length of the host variable must be at least as big as the value of the item descriptor field LENGTH.

Numeric values

For the SQL data type NUMERIC or DECIMAL, the total number of significant digits of the host variable must be the same as the value of the item descriptor field PRECISION and the number of digits to the right of the decimal point the same as the value of the item descriptor field SCALE.

Time values

The SQL data type of the host variable must correspond to the data type of the item descriptor field DATETIME_INTERVAL_CODE.
In the case of the SQL data types TIME and TIMESTAMP, the item descriptor field PRECISION must contain the value 3.

Recommended procedure

The following procedure is recommended if you do not want to have to define host variables for every possible data type:

1. Use DESCRIBE to store the data type description for the value in the DATA field of the item descriptor.
2. Query the data type of the item descriptor with GET DESCRIPTOR.
3. Change the data type of the item descriptor to match the data type of the host variable with SET DESCRIPTOR.
4. Transfer the value from DATA to or from the host variable.

Example

You want to prepare the following dynamic statement:

```
SELECT street, country, zip, city FROM customers WHERE company='Siemens'
```

After executing DESCRIBE OUTPUT, GET DESCRIPTOR will provide you with the following data type descriptions:

VALUE	REPETITIONS	TYPE	LENGTH	PRECISION	SCALE	Corresponding data type
1	1	1	40	0	0	CHAR(40)
2	1	1	3	0	0	CHAR(3)
3	1	2		5	0	NUMERIC(5,0)
4	1	1	40	0	0	CHAR(40)

If you want to use host variables of the type CHAR(100) and NUMERIC(15,5) for storing values, use SET DESCRIPTOR to set the item descriptor fields to the following values:

VALUE	REPETITIONS	TYPE	LENGTH	PRECISION	SCALE
1	1	1	100		
2	1	1	100		
3	1	2		15	5
4	1	1	100		

You can now execute the prepared statement with EXECUTE. The values are stored in the descriptor area. STREET, COUNTRY and CITY are padded on the right with blanks until their length is 100. Five leading zeros and five zeros after the decimal point are added to ZIP.

You can use GET DESCRIPTOR to transfer the values to the appropriate host variables and process them.

4.4.6 Modifying the target data type by means of the CAST operator

In some cases, you can use the CAST operator (see section “CAST expression”) to specify an appropriate target data type, even if SESAM/SQL determines a different data type internally.

Example

The following dynamic statement contains a two-digit operator with a placeholder (?).

```
UPDATE service SET vat=0.15+?
```

SESAM/SQL determines the data type of the placeholder for this two-digit operator from the data type of the other operator with NUMERIC(3,2). If the user wants a different data type, such as NUMERIC(4,2), he or she can use the CAST operator to specify this:

```
UPDATE service SET vat=CAST(? AS NUMERIC(4,2))
```

4.4.7 Supplying input parameters for routines

When you assign values to the input parameters for the routine in a CALL statement (procedure call) or when a User Defined Function (UDF) is called, data-type-specific rules apply. These are described below.

Strings

You can assign an alphanumeric value to an input parameter with the alphanumeric data type or a national value to an input parameter with a national data type. The following rules apply:

- If the target data type is CHAR or NCHAR and the length of the value is smaller than the length of the target data type, the value is padded on the right with blanks.
- If the target data type is CHAR or NCHAR and the length of the value is greater than the length of the target data type, the value is truncated from the right to the length of the target data type. If characters are removed that are not blanks, the value is not entered and an error message is issued.
- If the target data type is VARCHAR or NVARCHAR and the length of the value is greater than the maximum length of the target data type, the value is truncated from the right to the maximum length of the target data type. If characters are removed that are not blanks, the value is not entered and an error message is issued.

Numeric values

You can assign a numeric value to an input parameter with a numeric data type. If the numeric data types are not the same, the value is converted to the data type of the input parameter. The following rules apply:

- If the number of digits to the right of the decimal point of the value is too large for the data type of the input parameter, the value is rounded.
- If the value is too large for the data type of the input parameter, the value is not entered and an error message is issued.

Time values

You can only assign a time value to an input parameter with the same data type:

- a date in an input parameter with the data type DATE
- a time for an input parameter with the data type TIME
- a time stamp in an input parameter with the data type TIMESTAMP

4.4.8 Entering values in a procedure parameter (output) or local variable

When you assign values to the output parameters in a procedure or to the local variables or the function value of a UDF in a routine (SET, RETURN, SELECT...INTO, FETCH...INTO, INSERT...RETURN INTO), data-type-specific rules apply. These are described below.

Strings

You can enter an alphanumeric value in an output parameter or a local variable with an alphanumeric data type. You can enter a national value in an output parameter or local variable with a national data type. The following rules apply:

- If the target data type is CHAR or NCHAR and the length of the value is smaller than the length of the target data type, the value is padded on the right with blanks.
- If the target data type is CHAR or NCHAR and the length of the value is greater than the length of the target data type, the value is truncated from the right to the length of the target data type and a warning is issued.
- If the target data type is VARCHAR or NVARCHAR and the length of the value is greater than the maximum length of the target data type, the value is truncated from the right to the maximum length of the target data type and a warning is issued.

Numeric values

You can enter a numeric value in an output parameter or local variable with a numeric data type. If the numeric data types are not the same, the value is converted to the target data type. The following rules apply:

- If the number of digits to the right of the decimal point of the value is too large for the target data type, the value is rounded.
- If the value is too large for the target data type, the value is not entered and an error message is issued.

Time values

You can only enter a value with time data type in an output parameter or local variable with the same data type:

- A date in an output parameter or local variable with the data type DATE
- A time in an output parameter or local variable with the data type TIME
- A time stamp in an output parameter or local variable with the data type TIMESTAMP

5 Compound language constructs

This chapter describes the compound language constructs that can occur in SESAM/SQL statements. It is subdivided into the following sections:

- Expression
- Function
- Predicates
- Search condition
- CASE expression
- CAST expression
- Integrity constraint
- Column definitions

These language constructs are made up of basic elements, such as names, literals and other language constructs. They are described in logical sequence.

5.1 Expression

The evaluation of an expression returns a value or supplies a table (table functions).

Expressions can occur in:

- Column selection (SELECT expression, SELECT expression)
- predicates in search conditions (e.g. WHERE clause, HAVING clause)
- assignments (INSERT, MERGE or UPDATE statement)
- SQL statements which are used in routines (e.g. CASE statement)

An expression consists of operands and can include operators. The operators are used on the results of the operands.

The result of the evaluation is an alphanumeric, national, numeric or time value.

A table function returns a table as a result.

The operands are not evaluated in a predefined order. In certain cases, a partial expression is not calculated if it is not required for calculating the total result.

When an operand is evaluated with a function call, the function is first performed and then the function call replaced by the resulting value or the table which is returned.

Syntax diagram of an expression:

```
expression ::=

{
   value |
   [ table . ] {
                  column |
                   { column ( posno ) | column [ posno ] } |
                   { column ( min..max ) | column [ min..max ] }
               } |
   function |
   subquery |
   monadic_op expression |
   expression dyadic_op expression |
   case_expression |
   cast_expression |
   ( expression )
}


column ::= unqual_name

posno ::= unsigned_integer
min ::= unsigned_integer
max ::= unsigned_integer
```

monadic_op ::= { + | - }

dyadic_op ::= { * | / | + | - | || }

value

Alphanumeric value, national value, numeric value or time value (see section “Values”).

table

Name of the table containing *column*. If a correlation name has been defined for the table, specify the correlation name instead of the table name.

column

Name of the column from which the values are to be taken.

Any square brackets shown here in italics are special characters, and must be specified in the statement.

pos_no

Unsigned integer

The value is taken from the (*pos_no*-col_{min}+1)th column element of the multiple column *column* and can be used as an atomic value.

If *column* is not a multiple column, *pos_no* is smaller than col_{min} or *pos_no* is greater than col_{max}, an error message is issued.

col_{min} and col_{max} are the smallest and largest position numbers of the multiple column.

min .. *max*

Unsigned integers

The value is the aggregate from the column elements (*min*-col_{min}+1) to (*max*-col_{min}+1) of the multiple column *column*.

If *column* is not a multiple column, *min* is not smaller than *max*, *min* is smaller than col_{min} or *max* is greater than col_{max}, an error message is issued.

col_{min} and col_{max} are the smallest and largest position numbers of the multiple column.

pos_no or *min* .. *max* omitted:

column cannot be a multiple column.

function

Function (see section “Function”).

subquery

Subquery (see section "Subquery") that returns exactly one value.

monadic_op

Monadic operator that sets the sign. *expression* must be numeric and cannot be a multiple value with a dimension > 1.

+ The value remains as it is.

- The value is negated.

dyadic_op

Dyadic operator. Neither of the operand expressions can be a multiple value with a dimension > 1.

a * *b*

Multiply *a* with *b*.

The expressions *a* and *b* must be numeric.

If *a* and *b* are integers or fixed-point numbers, the result is an integer or fixed-point number with t_a+t_b significant digits with a maximum number of 31 digits.

The number of digits to the right of the decimal point is r_a+r_b, with a maximum number of 31 digits.

t_a and t_b are the total number of significant digits for *a* and *b*.

r_a and r_b are the number of digits to the right of the decimal point for *a* and *b* respectively.

If *a* or *b* is a floating-point numbers, the result is a floating-point number with a total number of significant digits of 24 bits for REAL numbers and 56 bits for DOUBLE PRECISION numbers.

If the result value is too big for the resulting data type, an error message is issued. If the total number of significant digits is too big, the number is rounded.

a **/** *b*

Divide *a* by *b*.

The expressions *a* and *b* must be numeric.

If *a* and *b* are integers or fixed-point numbers, the result is an integer or fixed-point number with 31 significant digits.

The number of digits to the right of the decimal point is $31-l_a-r_b$, at least however 0.

l_a is the number of digits to the left of the decimal point for *a*.

r_b is the number of digits to the right of the decimal point for *b*.

If *a* or *b* is a floating-point numbers, the result is a floating-point number with a total number of significant digits of 24 bits for REAL numbers and 56 bits for DOUBLE PRECISION numbers.

If the result value is too big for the resulting data type or the value of *b* is 0, an error message is issued. If the total number of significant digits is too big, the number is rounded.

a **+** *b*

Add *a* and *b*.

The expressions *a* and *b* must be numeric.

If *a* and *b* are integers or fixed-point numbers, the result is an integer or fixed-point number with $l_{max}+r_{max}+1$ significant digits with a maximum number of 31 digits.

The number of digits to the right of the decimal point is r_{max}.

l_{max} is the larger of the two numbers of digits to the left of the decimal point for *a* and *b*.

r_{max} is the larger of the two numbers of digits to the right of the decimal point for *a* and *b*.

If *a* or *b* is a floating-point numbers, the result is a floating-point number with a total number of significant digits of 24 bits for REAL numbers and 56 bits for DOUBLE PRECISION numbers.

If the result value is too big for the resulting data type, an error message is issued. If the total number of significant digits is too big, the number is rounded.

a **-** *b*

Subtract *b* from *a*.

The expressions *a* and *b* must be numeric.

If *a* and *b* are integers or fixed-point numbers, the result is an integer or fixed-point number with $l_{max}+r_{max}+1$ significant digits with a maximum number of 31 digits.

The number of digits to the right of the decimal point is r_{max}.

l_{max} is the larger of the two numbers of digits to the left of the decimal point for *a* and *b*.

r_{max} is the larger of the two numbers of digits to the right of the decimal point for *a* and *b*.

If *a* or *b* is a floating-point numbers, the result is a floating-point number with a total number of significant digits of 24 bits for REAL numbers and 56 bits for DOUBLE PRECISION numbers.

If the result value is too big for the resulting data type, an error message is issued. If the total number of significant digits is too big, the number is rounded.

a **||** *b*

Concatenate *a* and *b*.

The expressions *a* and *b* must result in alphanumeric or national values.

If *a* and *b* are of the data type CHAR, the result is of the data type CHAR with a length of l_a+l_b (in characters), and this sum may not be greater than 256.

If *a* and *b* are of the data type NCHAR, the result is of the data type NCHAR with a length of $l_a+ l_b$ (in code units), and this sum may not be greater than 128.

If *a* or *b* is of the data type VARCHAR, the result is of the data type VARCHAR with a length of l_a+l_b (in characters), but at most 32 000.

If *a* or *b* is of the data type NVARCHAR, the result is of the data type NVARCHAR with a length of $l_a+ l_b$ (in code units), but at most 16 000. l_a and l_b are the lengths of *a* and *b*.

If a result of the type CHAR is longer than 256 characters or the result of the type NCHAR is longer than 128 characters, an error message is issued.

If a result of the type VARCHAR is longer than 32 000 characters, the string is truncated from the right to a length of 32 000 characters and if a result of the type NVARCHAR is longer than 16 000 characters, the string is truncated from the right to a length of 16 000 characters. If characters are removed that are not blanks, an error message is issued.

case_expression

CASE expression (see section “CASE expression”).

cast_expression

CAST expression (see section “CAST expression”).

Precedence

- Expressions enclosed in parentheses have highest precedence.
- Monadic operators take precedence over dyadic operators.
- The operators for multiplication (*) and division (/) take precedence over the operators for addition (+) and subtraction (-).
- Operators for multiplication all have the same precedence level.
- Operators for addition all have the same precedence level.
- Operators with the same precedence level are applied from left to right.
- When *expression* is an unqualified name *unqual_name* for which there is both a column and a routine parameter or a local variable with this name in the area of validity, the routine parameter or the local variable is used.

> i *Recommendation* The names of routine parameters and local variables should differ from column names (e.g. by assigning a prefix such as `par_` or `var_`).

5.2 Function

A function calculates a value or returns a table (table function). Functions can be called from within expressions. When an operand is evaluated with a function call, the function is first performed and then the function call replaced by the resulting value or the table which is returned. SESAM/SQL functions fall into two groups:

- Time functions
- String functions
- Numeric functions
- Aggregate functions
- Table functions
- Cryptographic functions
- User Defined Functions (UDFs)

---

function ::=

{

time_function |

string_function |

numeric_function |

aggregate_function |

table_function |

crypto_function |

user_defined_function

}

---

time_function

Time function (see section “Time functions”).

string_function

String function (see section “String functions”).

numeric_function

Numeric function (see section “Numeric functions”).

aggregate_function

Aggregate function (see section “Aggregate functions”).

table_function

Table function (see section “Table functions”).

crypto_function

Cryptographic function (see section “Cryptographic functions”).

user_defined_function

User Defined Function (see section “User Defined Functions (UDFs)”).

5.2.1 Time functions

Time functions determine following data

- current date (CURRENT_DATE)
- current time (CURRENT_TIME(3) or LOCALTIME(3))
- time stamp with the current date and current time (CURRENT_TIMESTAMP(3) or LOCALTIMESTAMP(3))
- date corresponding to an integer value (DATE_OF_JULIAN_DAY) (see also the inverse function JULIAN_DAY_OF_DATE on "JULIAN_DAY_OF_DATE() - Convert date").

LOCALTIMESTAMP(3) and CURRENT_TIMESTAMP(3) are equivalent in SESAM/SQL, as are LOCALTIME(3) and CURRENT_TIME(3).

---

time_function ::=

```
{
   CURRENT_DATE |
   CURRENT_TIME(3) |
   LOCALTIME(3) |
   CURRENT_TIMESTAMP(3) |
   LOCALTIMESTAMP(3) |
   DATE_OF_JULIAN_DAY( expression )
}
```

---

expression

Numeric integer value which SESAM/SQL interprets as a Julian day number. *expression* may not be a multiple value with dimension > 1.

If the time functions CURRENT_DATE, CURRENT_TIME(3), LOCALTIME(3), CURRENT_TIMESTAMP(3) and LOCALTIMESTAMP(3) are included in a statement multiple times, they are executed simultaneously. This also applies for all time functions that are evaluated as the result of the statement:

- time functions in the DEFAULT clause of the column definition if the default value is used
- time functions that occur in the SELECT expression of a view or temporary view if the view or temporary view is referenced

All the values that are returned have the same data and/or time. Therefore, you cannot use time functions to determine execution times within a statement.

Time functions in dynamic statements and in cursor descriptions are evaluated when the EXECUTE, EXECUTE IMMEDIATE or OPEN statement is performed.

5.2.2 String functions

String functions perform the following tasks:

- extract substrings (SUBSTRING)
- transliterate alphanumeric strings to national strings or vice versa (TRANSLATE)
- transcode national strings from UTFE to UTF-16 or vice versa (TRANSLATE)
- remove leading or trailing characters of strings (TRIM)
- convert uppercase letters to lowercase letters or lowercase letters to uppercase letters (LOWER, UPPER)
- convert a value of any data type to the internal presentation (as an alphanumeric string or in hexadecimal format) and vice versa (HEX_OF_VALUE, VALUE_OF_HEX, REP_OF_VALUE, VALUE_OF_REP)
- for national strings, supply the collation element in accordance with the Default Unicode Translation Table (COLLATE)
- convert national strings to normal form (NORMALIZE)

```
string_function ::=

{

    SUBSTRING ( expression FROM startposition [FOR substring_length ][USING CODE_UNITS])|
    TRANSLATE ( expression USING [[ catalog .]INFORMATION_SCHEMA.] transname

                        [DEFAULT character ] [ ,length ]) |

    TRIM ([[LEADING |TRAILING | BOTH] [ character ] FROM] expression ) |
    LOWER ( expression ) |
    UPPER ( expression ) |
    HEX_OF_VALUE ( expression2 ) |
    VALUE_OF_HEX ( expression3 , data_type ) |
    REP_OF_VALUE ( expression2 ) |
    VALUE_OF_REP ( expression3 , data_type ) |
    COLLATE ( expression USING { DUCET_WITH_VARS | DUCET_NO_VARS } [ ,length ]) |
    NORMALIZE ( expression [ ,NFC | NFD [ , length ]])

}

character ::= expression
length ::= unsigned_integer
```

---

expression

Alphanumeric expression or national expression. Its evaluation must return either an alphanumeric string (data type CHAR or VARCHAR) or a national string (data type NCHAR or NVARCHAR). *expression* may not be a multiple value with dimension > 1. See also section “Compatibility between data types”.

Restrictions that apply to a function are described in the description of the relevant function.

expression2

Expression of any data type. The internal presentation of this value is returned as an alphanumeric string or in hexadecimal format.
expression2 may not be a multiple value with dimension > 1.

expression3

Alphanumeric expression which is the internal presentation of a value of the type *data_type*. This value is the result.
expression3 may not be a multiple value with dimension > 1.

startposition

Integral numeric expression for the position of the start of the substring.

substring_length

Integral numeric expression for the length of the substring.

data_type

Data type of the result.

length

Maximum length of the result string.

5.2.3 Numeric functions

Numeric functions achieve various purposes:

- ABS(), CEILING(), FLOOR(), MOD(), SIGN() and TRUNC() execute the corresponding mathematical functions on the specified numeric expressions.
- CHARACTER_LENGTH( ), OCTET_LENGTH( ) and POSITION( ) calculate the number of bytes or code units in a string or the position of a string in another string.
- JULIAN_DAY_OF_DATE( ) converts a date into an integer value.
- EXTRACT() extracts specific components of a time value.

When a numeric function is evaluated, a numeric value is returned.

---

numeric_function ::=

```
{
    ABS ( expression ) |
    CEIL[ING] ( expression ) |

    FLOOR ( expression ) |
    MOD ( dividend,divisor ) |
    SIGN ( expression ) |
    TRUNC ( expression ) |

    { CHAR_LENGTH | CHARACTER_LENGTH } ( expression [USING { CODE_UNITS | OCTETS }]) |

    OCTET_LENGTH ( expression ) |
    POSITION ( expression IN expression [USING CODE_UNITS]) |
    JULIAN_DAY_OF_DATE ( expression ) |
    EXTRACT ( part FROM expression )
}
```

---

expression

In ABS(), CEILING(), FLOOR(), MOD(), SIGN() and TRUNC(): numeric expression.

In EXTRACT() and JULIAN_DAY_OF_DATE( ): time value expression.

Otherwise: alphanumeric expression or national expression.

expression may not be a multiple value with dimension > 1.

5.2.4 Aggregate functions

Aggregate functions return the average, count, maximum value, minimum value or sum of a set of values or the number of rows in a derived table.

aggregate_function `::= { ` *operator* `([ ALL | DISTINCT ]` *expression* `) | COUNT(*) }`

operator `::= {AVG | COUNT | MAX | MIN | SUM }`

expression

Expression determining the values in the set (see section “Expression”).

The *expression* for each aggregate function except for COUNT(*) can have a certain data type. The permitted data type(s) for each function is specified in the function description.

The following restrictions apply to *expression*:

- *expression* cannot include any multiple columns.
- *expression* cannot include any aggregate functions.
- *expression* cannot include any subqueries.
- If a column name in *expression* specifies a column of a higher-level query expression (external reference), *expression* may only include this column name.

  In this case, the aggregate function must satisfy one of the following conditions:

  - The aggregate function is included in a SELECT list.
  - The aggregate function is included in a subquery of a HAVING clause. The column name must indicate a column of the SELECT expression that contains a HAVING clause.

> i The aggregate functions MIN() and MAX() reference the set of all values in a column in a table. They differ in this way from a CASE expression with MIN / MAX (see "CASE expression with MIN / MAX"), which references different expressions.

Calculating aggregate functions

In all the aggregate functions except COUNT(*), the expression specified as the function argument determines the set of values used in the aggregate function.

If the SELECT expression or SELECT statement in which the aggregate function occurs does not include a GROUP BY clause, the argument expression is used on all the rows in the table (or the rows that satisfy the WHERE clause) referenced by the column specifications in the argument expression. If the argument expression does not contain a column specification, the argument expression is used on all the rows in the table of the SELECT expression. The result is a single-column table.

If this table contains NULL values, these are removed before the aggregate function is performed. A warning is issued.
If DISTINCT is specified in the aggregate function, only unique values are taken into account, i.e. if a value occurs more than once in a table, the duplicates are removed before the aggregate function is performed.
The aggregate function is then used on the remaining values of the single-column table and returns exactly one value.

If the corresponding SELECT expression (or SELECT statement) includes a GROUP BY clause, the aggregate function is calculated as described for each group separately and returns exactly one value per group.

Examples

Without GROUP BY: The following expression calculates the sum of the trebled price of the items from the ITEMS table:

```
SELECT SUM (3*price) FROM items
```

In order to calculate the expression, the argument expression `3*price` is used on all the rows of the ITEMS table. This returns the following derived column:

```
2101.50
 690,00
 450.00
 450.00
 120.00
 120.00
 180.00
  15.00
  15.00
  30.00
   3.00
   3.30
   2.25
```

The sum of the values is 41880.05.

With GROUP BY: The following expression calculates the total stock per location from the WAREHOUSE table.

```
SELECT location, SUM (stock) FROM warehouse GROUP BY location
```

In order to calculate the expression the stock per location is grouped together first:

```
location            stock
Main warehouse         2
                       1
                      10
                      10
                       3
                       3
                       1
                      15
                       8
                       6
                      11
                     120
                     248
Parts warehouse        9
```

```
                      6
                      3
                    200
                    180
                     47
```

Subsequently, the stock is added together for each warehouse.

```
location
Main warehouse           438
Parts warehouse          445
```

5.2.5 Table functions

Table functions generate tables whose content depends on the call parameters or is derived from external data sources, e.g. files.

table_function ::= {

CSV ([FILE] *file* DELIMITER *delimiter* [QUOTE *quote*] [ESCAPE *escape*], *data_type* ,...) |

DEE [()]

}

The table functions are described on "CSV() - Reading a BS2000 file as a table" and "DEE() - Table without columns".

5.2.6 Cryptographic functions

The ENCRYPT() and DECRYPT() functions are used to encrypt and decrypt individual values. Sensitive data is protected against unauthorized access by encryption. Only the users who know the “key” can decrypt the data.

The REP_OF_VALUE() and VALUE_OF_REP() functions can be used to jointly encrypt multiple values and to decrypt them again.

Introductory information on access control by means of data encryption in SESAM/SQL is provided in the “ Core manual”.

---

crypto_function ::= { `ENCRYPT` (*expression* , *key*) | `DECRYPT` (*expression2* , *key* , *data_type*) }

key ::= *expression*

---

expression

Expression whose value is to be encrypted. *expression* must not be a multiple value with dimension > 1.

expression2

Alphanumeric expression whose value is to be encrypted. e*xpression2* must not be a multiple value with dimension > 1.

key

Key for encryption and decryption.

data_type

Data type of the decrypted value.*data_type* must not be an aggregate (see "Values for multiple columns").

Application information

Since the encryption algorithm AES (see the “ Core manual”) - as it is used in SESAM/SQL - processes blocks of 16 characters, the length of the output value is always a multiple of 16 characters. If two input values differ in only one bit, all the characters in their encrypted values will differ.

Encrypted values can be compared to see whether they are identical or not identical. They are identical or not identical precisely when the unencrypted values are identical or not identical. The unencrypted values must have the same data type here. In the case of strings the unencrypted values must also have the same length.

> i However, the comparisons 01 = 1.0 and 'abc' = 'abc ' each returns the truth value TRUE although the encryptions of these four values are all different.

Other comparisons (e.g. with < or <=) of encrypted values return results which have nothing to do with the corresponding comparisons of the unencrypted values. The predicates BETWEEN and LIKE do not make sense for encrypted data, either. The same applies for sorting by means of ORDER BY.

The encryption of a NULL value returns the NULL value of the corresponding data type. Whether or not a value is a NULL value is therefore not confidential information when encryption takes place. The encryption of a string with the length 0, on the other hand, returns a string with the length 16. Without knowing the key no distinction can be made from the encryptions of strings with 1 to 14 alphanumeric characters.

> i **CAUTION!**Encrypted values can normally not be encrypted if they are truncated or extended (even if the new length is a multiple of 16). A column with encrypted values should therefore, for example, not have

the data type CHAR(20) because then 4 blanks would be added to each encrypted value. These blanks would have to be removed again before encryption could take place.

5.2.7 User Defined Functions (UDFs)

UDFs have an almost identical function scope to procedures. They are described in detail in the chapter “Routines”.

The current authorization identifier must have the EXECUTE privilege for the UDF.

CHECK constraints may not contain a UDF.

---

user_defined_function ::= *unqual_routine_name arguments*

arguments ::= ([*expression* [{ , *expression* } ...]])

---

unqual_routine_name

Name of the UDF to be executed. You can qualify the unqualified UDF name with a database and schema name.

([*expression* [{ , *expression*} ...]])

List of arguments. The number of arguments must be the same as the number of UDF parameters in the UDF definition. The order of the arguments must correspond to that of the parameters. If no parameter is defined for the UDF, the list consists only of the parentheses.

The nth parameter is assigned the value of the nth argument before the UDF is executed.

The data type of the nth argument must be compatible with the data type of the nth parameter. For input parameters, see the information in section “Supplying input parameters for routines”.

5.2.8 Alphabetical reference section: Functions

The functions are described in alphabetical order in the following sections.

5.2.9 ABS() - Absolute value

Function group: numeric function

ABS() determines the absolute value of a numeric value.

```
ABS ( expression )
```

expression

Numeric expression.
expression may not be a multiple value with dimension > 1.

Result

When *expression* returns the NULL value, the result is the NULL value.

Otherwise: the absolute value of *expression*. In other words the value of *expression* when *expression* is positive, otherwise the value of -(*expression*).

Data type: like *expression*

Examples

`ABS (3,14)` returns the value 3,14.

`ABS (-3,14)` returns the value 3,14.

5.2.10 AVG() - Arithmetic average

Function group: aggregate function

AVG() calculates the average of a set of numeric values. NULL values are ignored.

AVG ([ALL | DISTINCT] *expression*)

ALL

All values are taken into account, including duplicate values.

DISTINCT

Only unique values are taken into account. Duplicate values are ignored.

expression

Numeric expression (see section “Aggregate functions” for information on restrictions).

Result

If the set of values returned by *expression* is empty, the result or the result for this group is the NULL value.

Otherwise:

Without GROUP BY clause:

Returns the arithmetic average of all the values in the specified *expression* (see “Calculating aggregate functions”).

With GROUP BY clause:

Returns the arithmetic average per group of all the values in the derived column for this group.

Data type: like *expression* with the following number of digits:

- Integer or fixed-point number:

  The total number of significant digits is 31, the number of digits to the right of the decimal point is 31-*t*+*r*. *t* and *r* are the total number of significant digits and the number of digits after the decimal point, respectively, in *expression*.

- Floating-point number:

  The total number of significant digits corresponds to 21 binary digits for REAL numbers and 53 for DOUBLE PRECISION.

Examples

SELECT without GROUP BY:

Calculate the average price of the services in the SERVICE table of the demonstration database (result: 783.33):

```
SELECT AVG(service_price) FROM service
```

If you enter a row in the table that contains the NULL value in the column service_price, the result does not change.

SELECT with GROUP BY:

The average price is calculated for each order number:

```
SELECT order_num, AVG(service_price) FROM service GROUP BY order_num
   order_num
   200      1025
   211      662.5
   250      662.5
```

5.2.11 CEILING() - Smallest integer greater than the value

Function group: numeric function

CEILING() ("round up to the ceiling") determines the smallest integer which is greater than or equal to the specified numeric value. In the case of non-integer numeric values, CEILING() always rounds up.

```
CEIL[ING] ( expression )
```

expression

Fixed-point value of the type NUMERIC(p,s) or DECIMAL(p,s) if the number of decimal places s is greater than 0, otherwise a numeric expression.
expression may not be a multiple value with dimension > 1.

Result

When *expression* returns the NULL value, the result is the NULL value.

Otherwise:

The smallest integer which is greater than the specified numeric value.

Data type: NUMERIC(q+1,0) or DECIMAL(q+1,0) where q=MIN(31,p+1) if the number of decimal places s is greater than 0, otherwise like *expression*.

Examples

`CEILING (3,14)` returns the value 4.

`CEILING (-3,14)` returns the value -3.

`CEILING (10,14)` returns the value 11.

5.2.12 CHAR_LENGTH() - Determine string length

Function group: numeric function

CHAR_LENGTH() or CHARACTER_LENGTH() determines the number of bytes or code units in a string.

```
{ CHAR_LENGTH | CHARACTER_LENGTH }( expression [USING [CODE_UNITS | OCTETS]])
```

expression

Alphanumeric expression or national expression. Its evaluation must return either an alphanumeric string (data type CHAR or VARCHAR) or a national string (data type NCHAR or NVARCHAR).

In the case of the alphanumeric data types CHAR and VARCHAR, CHAR_LENGTH( ) and OCTET_LENGTH( ) (see section “OCTET_LENGTH() - Determine string length”) return the same values because each character is represented in precisely one byte (octet).

In the case of the national data types NCHAR and NVARCHAR the length can be determined either in bytes (OCTET_LENGTH and CHAR_LENGTH ... USING OCTETS functions) or in UTF-16 code units (CHAR_LENGTH ... USING CODE_UNITS function). A code unit in UTF-16 = 2 bytes. The number of Unicode characters in a national string can be less than the number of code units in UTF-16 as some Unicode characters are represented by two consecutive code units in UTF-16 (surrogate pairs).

expression may not be a multiple value with dimension > 1. See also section “Compatibility between data types”.

USING CODE_UNITS

The length is to be output in code units.
In the data types CHAR and VARCHAR, 1 code unit = 1 byte.
In the data types NCHAR and NVARCHAR, 1 code unit = 2 bytes.

USING OCTETS

The length is to be output in bytes.
In the data types CHAR and VARCHAR, 1 character = 1 byte.
In the data types NCHAR and NVARCHAR, 1 character = 1 or 2 code units = 2 or 4 bytes respectively.

Result

If the string contains the NULL value, the result is the NULL value.

Otherwise:

The result is the number of bytes or code units in the string.

Data type: INTEGER

Examples

Determine the number of bytes (characters) contained in the alphanumeric string 'only' (result: 4).

```
CHAR_LENGTH ('only') USING OCTETS
```

Determine the number of bytes contained in the national string 'for' (result: 6).

```
CHAR_LENGTH (N'for') USING OCTETS
```

Determine the number of code units contained in the national string 'for' (result: 3).

```
CHAR_LENGTH (N'for') USING CODE_UNITS
```

Determine the number of code units contained in the national string 'München' (result: 7).

```
CHAR_LENGTH (U&'M\00FCnchen')
```

5.2.13 COLLATE() - Determine collation element for national strings

Function group: string function

COLLATE() supplies, for national strings, the collation element in accordance with the Default Unicode Collation Table (DUCET), see the “ Core manual”.

Code points which are not assigned and code points > `U+2FFF` are ignored. Collation elements extend to comparison level 3; level 4 is ignored.

`COLLATE (` *expression* `USING [[` *catalog* `.]INFORMATION_SCHEMA.] [` *collation* `,` *length* `])`

collation ::= `{ DUCET_WITH_VARS | DUCET_NO_VARS }`

length `::=` *unsigned_integer*

---

expression

National expression.

DUCET_WITH_VARS| DUCET_NO_VARS

Name of the collation (sort sequence) to be used.
In SESAM/SQL all collation names are predefined. These are the names which are also defined in the BS2000 software product SORT for sorting strings.

In the case of DUCET_NO_VARS, the variable collation elements, e.g. blanks, punctuation marks and continuation characters, are ignored.

In the case of DUCET_WITH_VARS, they are taken into account.
The strings U&'cannot' and U&'can not' are sorted in this order with
DUCET_NO_VARS, and in the opposite order with DUCET_WITH_VARS.

The collation can be qualified by a database name and the schema name INFORMATION_SCHEMA, otherwise the INFORMATION_SCHEMA is taken as the predefined database.

length

Maximum length of the collation element where 1 <= *length* <= 32000.

Length not specified:
The result can have a length length of 32000 bytes, depending on *expression*.

Result

When *expression* returns the NULL value, the result is the NULL value.

Otherwise:

The result is the collation element for *expression* in accordance with the Default Unicode Collation Table (DUCET) with the length *n* = 4 + 6 * (length of *expression* in code units), where *n* <= 32000.

If the length of the collation clement is greater than the specified or maximum length, the function is aborted with SQLSTATE.

Data type: VARCHAR(*n*)

Examples

Output of a list of customer contacts sorted according to the Default Unicode Collation Table taking into account the variable collation elements:

```
UNLOAD ONLINE DATA CONTACTS (LNAME,FNAME,TITLE,CONTACT_TEL,POSITION) -
    INTO FILE 'DAT.070.C.DUCETWITHVARS' -
    CSV_FORMAT DELIMITER ';' QUOTE '"' ESCAPE '\' EBCDIC -
    ORDER BY COLLATE(TRANSLATE(LNAME USING EDF041 DEFAULT N'?') -
                     USING DUCET_WITH_VARS,200) -
             ASC, -
             COLLATE(TRANSLATE(FNAME USING EDF041 DEFAULT N'?') -
                     USING DUCET_WITH_VARS,200) -
             ASC
```

Output of the collation element for a letter:

```
HEX_OF_VALUE(COLLATE(TRANSLATE ('A' USING EDF041) USING DUCET_NO_VARS))

0E33000020000800
```

5.2.14 COUNT(*) - Count table rows

Function group: aggregate function

COUNT(*) counts the rows in a table. Rows containing NULL values are included in the count.

```
COUNT (*)
```

Result

Without GROUP BY clause:

Returns the number of rows in the derived table of the corresponding SELECT expression (or corresponding SELECT statement). Duplicate rows and rows containing only NULL values are included.

With GROUP BY clause:

Returns the number of rows per group for each group in the derived table.

Data type: DECIMAL(31,0)

Examples

SELECT without GROUP BY:

Query the number of customers living in Munich in the CUSTOMERS table (result: 3):

```
SELECT COUNT(*) FROM customers WHERE city='Munich'
```

SELECT with GROUP BY:

Count the customers for each city:

```
SELECT city, COUNT(*) FROM customers GROUP BY city
city
Berlin                    1
Bern 33                   1
Hanover                   1
Moenchengladbach          1
Munich                    3
New York, NY              1
```

5.2.15 COUNT() - Count elements

Function group: aggregate function

COUNT() counts the elements in a set of values. NULL values are not included in the count.

```
COUNT ([ALL | DISTINCT ] expression )
```

ALL

All values are taken into account, including duplicate value.

DISTINCT

Only unique values are taken into account. Duplicate values are ignored.

expression

Numeric expression, alphanumeric expression, national expression or time value expression (see section "Aggregate functions" for information on restrictions).

Result

Without GROUP BY clause:

Number of values in the set returned by *expression* (see "Calculating aggregate functions").

With GROUP BY clause:

Returns the number of values in each group.

Data type: DECIMAL (31, 0)

Examples

SELECT without GROUP BY:

Determine the number of different service descriptions in the SERVICE table (result: 7):

```
SELECT COUNT(DISTINCT service_text) FROM service
```

SELECT with GROUP BY:

Count the number of different services for each order number:

```
SELECT order_num, COUNT(DISTINCT service_text) FROM service
   GROUP BY order_num
   order_num
   200                 2
   211                 4
   260                 2
```

5.2.16 CSV() - Reading a BS2000 file as a table

Function group: table function

The table function CSV() enables you to use the content of a BS2000 file as a “read-only” table in any SQL statements.

CSV format (CSV: Comma Separated Values) is used to display SQL tables in files here. This is a standardized format for the platform-independent exchange of table data, see “Format of CSV files”. The file contains the sequence of table rows, each row containing its column values sequentially as a string. Such files can be generated with a large number of software products (e.g. with Microsoft EXCEL).

`CSV ([FILE]` *file* `DELIMITER` *delimiter* `[QUOTE` *quote* `] [ESCAPE` *escape* `],` *data_type* `,...)`

FILE *file*

Name of the input file. You must specify *file* as an alphanumeric literal.

The input file must be a SAM file.
If the input file is not located in the ID of the DBH, the DBH ID must have read authorization for this file. Otherwise the DBH cannot access the input file.

If a read password is required for the file, this must be added to the BS2000 file in the form `?PASSWORD=<`*password*`>`, e.g. `':8OSH:$ABC.MYFILE?PASSWORD=C''ABCD'''`.

password can be specified in several different ways:

- C`''`*string*`''`
  string contains four printable characters.
- X`''`*hex_string*`''`
  hex_string contains eight hexadecimal characters.
- *n*
  n identifies an integer from - 2147483648 to + 2147483647.

DELIMITER *delimiter*

Delimiters (DELIMITER characters) between the column values of the CSV file. A DELIMITER character can also be part of a value, see the descriptions of *quote* and *escape* below.
delimiter must be specified as an alphanumeric literal with the length 1.

QUOTE *quote*

QUOTE characters in which the column values in the CSV file can be enclosed. These QUOTE characters are not part of the column value. A QUOTE character in the column value must be entered twice in the CSV file. When a value is enclosed in QUOTE characters, it can also contain NEWLINE characters (which are not interpreted as a line break) or DELIMITER characters. A value consisting only of an opening and a closing QUOTE character is interpreted as a value with the length 0.
quote must be specified as an alphanumeric literal with the length 1.
When QUOTE is not specified, the column values in the CSV file cannot be enclosed in QUOTE characters.

ESCAPE *escape*

ESCAPE character with which ESCAPE sequences consisting of two characters in the input file begin. ESCAPE sequences enable DELIMITER characters, QUOTE characters and ESCAPE characters to be written as part of a column value and NEWLINE characters to be ignored as a delimiter between two input lines.
escape must be specified as an alphanumeric literal with the length 1.
When ESCAPE is not specified, no ESCAPE sequences can be used in the CSV file.

> i The characters specified for DELIMITER, QUOTE and ESCAPE must all be different.

data_type,...

Data types of the various columns in the table which is read from the CSV file.Every *data_type* must be data type CHARACTER(n) (where 1 <= n <= 256) or CHARACTER VARYING(n) (where 1 <= n <= 32000).

Result

A table with as many columns as data types which are specified, each with the specified data type.

Example

A new SERVICE_ENCR base table is set up. Its contents are taken from a CSV file.

```
INSERT INTO service_encr (setext, seprice_encr)
SELECT a,b
   FROM TABLE(CSV(FILE 'out.service.070' DELIMITER ':',
           CHAR(25),VARCHAR(16)))
   AS t(a,b)
```

Format of CSV files

The CSV format (CSV: Comma Separated Values) is a standardized format for the platformindependent exchange of table data. Such files can be generated and edited with a large number of software products (e.g. with Microsoft EXCEL).

Tables are presented in CSV files as a sequence of lines, the lines in a file being separated by (one or more) NEWLINE characters (line breaks). The transition to the next record in a SAM file is also such a new line, although this is not an EBCDIC character. A record in a SAM file can contain multiple lines, separated by a NEWLINE character. New line characters may also occur before the first and after the last line.

The various column values in a line are separated by a single DELIMITER character. A DELIMITER character may also occur after the last column value of a line.

There are two ways of presenting the various column values in each line: The individual characters in a column can be enclosed in QUOTE characters or not. In the first case the column values can also contain the NEWLINE and the DELIMITER characters. However, a QUOTE character in the column value must be entered twice (otherwise it terminates the column value). Column values in QUOTE characters can only be used if the QUOTE operand is specified in the CSV function.

If a column value does not begin with the QUOTE character (or if the QUOTE operand is not specified in the CSV function), the column value will end before the next DELIMITER or NEWLINE character.

In SESAM/SQL you can also define an ESCAPE character. The ESCAPE character enables you to use ESCAPE sequences in the column value, which are interpreted as follows:

Escape sequence	Interpreted as
escape newline	“no character”
escape delimiter	a DELIMITER character
escape quote	a QUOTE character
escape escape	an ESCAPE character

ESCAPE sequences are also permitted in column values which are enclosed in QUOTE characters. ESCAPE NEWLINE in particular is useful, because when an ESCAPE character is contained at the end of a SAM record, the line is regarded as not yet completed and is continued with the following SAM record. The lines in a CSV file can thus be longer than one record in a SAM file of BS2000.

If errors occur when the CSV file is read or an infringement of the CSV format is detected (e.g. in the case of end of file in a column value which begins with a QUOTE character but does not end with one), this is indicated with an error code.

Note on NEWLINE characters

In CSV format four EBCDIC control characters are interpreted as a NEWLINE characters:

X'04' is the NEXT LINE character

X'0D' is the CARRIAGE RETURN character. Its ASCII equivalent is used as the newline character in some Macintosh systems.

X'15' is the LINE FEED character. Its ASCII equivalent is used as the newline character in POSIX and LINUX systems. In EBCDIC systems from IBM it is used as NEXT LINE or LINE FEED. The ASCII equivalent of X'0D15' is used as a string for (one) newline character in Windows systems.

X'25' is the PRIVATE USE TWO character. However, in EBCDIC systems from IBM it is used as LINE FEED or NEXT LINE, and in the IBM z/OS Unix System Services as a newline character.

The CSV format accepts all these control characters (like the transition to the next record of a SAM file) as newline characters.

Syntax of a CSV file

A syntactical presentation of the format of a CSV file is provided on "Syntax overview of the CSV file".

Interpreting CSV files as an SQL table

In the CSV function the number of columns to be read and their data types are specified. These columns correspond to the column values in the CSV file in the same order. If a line in the CSV file contains fewer column values, NULL values are added. If a line in the CSV file contains more column values, the surplus column values are ignored.

A line in a CSV file must contain at least one character. Multiple consecutive newline characters are treated as one newline character.

An empty column value (e.g. between two consecutive DELIMITER characters) is interpreted as a NULL value.

A column value which is longer than the (maximum) length of the column's data type is truncated. A warning is issued.

If the data type of the column is CHARACTER(n) but the column value is shorter than n, the column value is padded at the end with blanks (X'40').

A column value with the length 0 can be written with QUOTE characters, e.g. as "" if DELIMITER ';' QUOTE '"' is specified in the CSV function.

Restrictions in the use of CSV files

The BS2000 file is opened exclusively. It can therefore not be used simultaneously by the same or another SQL transaction in another CSV function. A remedy is offered by the CACHE annotation, in which the CSV is cached temporarily, see the " Performance" manual.

If the file cannot be opened, an error message is issued and processing is terminated.

The file is closed only when the query containing it has been analyzed fully or when the query is no longer required (e.g. because the cursor which used the file is closed) or when the CSV file is cached.

In addition, there is a maximum number of CSV files (currently 4) which may be opened simultaneously. If this maximum number is exceeded, a corresponding error message is issued.

When one coded character set (`CODE_TABLE` not equal to `_NONE_`or `CODED-CHARACTER-SET` not equal to `*NONE`) each is defined for the database used and for the CSV file, the two names specified must be the same.

5.2.17 CURRENT_DATE - Current date

Function group: time function

CURRENT_DATE returns the current date.

```
CURRENT_DATE
```

Result

Current date

If several time functions are included in a statement, they are all executed simultaneously (see section “Time functions”).

Data type: DATE

5.2.18 CURRENT_TIME(3) - Current time

Function group: time function

CURRENT_TIME(3) returns the current time.

```
CURRENT_TIME(3)
```

Result

Current time

If several time functions are included in a statement, they are all executed simultaneously (see section “Time functions”).

Data type: TIME

5.2.19 CURRENT_TIMESTAMP(3) - Current time stamp

Function group: time function

CURRENT_TIMESTAMP(3) returns the current time stamp.

```
CURRENT_TIMESTAMP(3)
```

Result

Current time stamp

If several time functions are included in a statement, they are all executed simultaneously (see section “Time functions”).

Data type: TIMESTAMP

5.2.20 DATE_OF_JULIAN_DAY() - Convert Julian day number

Function group: time function

DATE_OF_JULIAN_DAY() returns the corresponding date in the Gregorian calendar for a given Julian day number (see also the inverse function JULIAN_DAY_OF_DATE() on "JULIAN_DAY_OF_DATE() - Convert date").

The Julian day number of a date is the number of days which have passed since the 24th November, 4714 BC (in accordance with the Gregorian calendar).

> i DATE_OF_JULIAN_DAY() and JULIAN_DAY_OF_DATE() are inverse functions. When, for example, a constraint exists in the form JULIAN_DAY_OF_DATE(*column*) < *:user_variable*, the SQL Optimizer can then convert this constraint internally to the constraint *column* < DATE_OF_JULIAN_DAY(*:user_variable*) in order to permit the use of indexes on *column*. Consequently *:user_variable* may only contain values which are permitted as an argument of DATE_OF_JULIAN_DAY(). This also applies for any constant expressions in place of *:user_variable*.

```
DATE_OF_JULIAN_DAY ( expression )
```

expression

Numeric integer expression. Its value represents the number of days which have passed since the 24th November 4714 B.C. Its value must lie between 1721426 and 5373484.
expression may not be a multiple value with dimension > 1.

Result

When *expression* returns the NULL value, the result is the NULL value.

Otherwise:

SESAM/SQL interprets the value of *expression* as a Julian day number. The result of the function is the date which corresponds to this Julian day number.

Data type: DATE

Example

```
DATE_OF_JULIAN_DAY (2451545)
```

```
2000-01-01
```

5.2.21 DECRYPT() - Decrypt data

Functon group: cryptographic function

DECRYPT() decrypts strings in accordance with the AES algorithm and using a key of 128 bits (16 bytes) in Electronic Codebook Mode (ECM) to the corresponding value of a specified data type.

```
DECRYPT ( expression , key , data_type )
```

expression

Specifies the value which is to be decrypted.
The value must be of the alphanumeric data type CHARACTER or CHARACTER VARYING.
expression may not be a multiple value with dimension > 1.
The length of *expression* must be an integral multiple of 16 and greater than 0. A NULL value is also permitted.

key

Key with which the value of *expression* is to be decrypted.
Alphanumeric string with a length of 16 characters, i.e. of the data type CHARACTER(16) or CHARACTER VARYING(n) where n >=16.
A NULL value of one of these data types is also permissible.
To obtain a correct result, the key must be the same as that which was used for encryption with ENCRYPT().

data_type

Data type of the decrypted value (without *dimension* specification). The data types permitted depend on the (maximum) length of the data type of *expression*, see the table on the next page.

Result

If the value of *expression* or *key* is the NULL value, the result is the NULL value.

Otherwise:

For the decrypted value of *expression* in the specified data type, see the table on the next page. For possible errors, see "Error cases".

Data type: the specified *data_type*

Data type of *expression*	***data_type* and data type of the result**
CHAR(m), VARCHAR(>= m) [1]	CHAR(n) if n <= 256 [2]
CHAR(m), VARCHAR(>= m) [1]	VARCHAR(n) [2]
CHAR(m), VARCHAR(>= m) [1]	NCHAR(n) [3]

CHAR(m), VARCHAR(>= m) [1]	NVARCHAR(n) [3]
CHAR(16), VARCHAR(>= 16)	SMALLINT, INTEGER
CHAR(16), VARCHAR(>= 16)	NUMERIC (up to 14 characters)
CHAR(32), VARCHAR(>= 32)	NUMERIC (15 to 30 characters)
CHAR(48), VARCHAR(>= 48)	NUMERIC (31 characters)
CHAR(16), VARCHAR(>= 16)	DECIMAL (up to 27 characters)
CHAR(32), VARCHAR(>= 32)	DECIMAL (28 to 31 characters)
CHAR(16), VARCHAR(>= 16)	FLOAT, REAL, DOUBLE PRECISION
CHAR(16), VARCHAR(>= 16)	DATE, TIME(3), TIMESTAMP(3)

Table 13: Permitted combinations in the case of DECRYPT()

[1]m must be>= 16 and an integral multiple of 16
[2]Length n must be >= 1 and between (m - 17) and (m -2) (inclusive)
[3]Length n must be >= 1 and between (m/2 - 1) and (m/2 - 8) (inclusive)

Examples

Decryption in a SELECT expression:

```
SELECT DECRYPT(sprice_encr,'0123456789ABCDEF',NUMERIC(5,0))

AS test_decr FROM service
```

The VALUE_OF_REP function also enables individual values of a jointly encoded string to be decrypted (see also "ENCRYPT() - Encrypt data"):

```
VALUE_OF_REP (SUBSTRING (DECRYPT (wagesandbonus, :key, CHAR(12))
   FROM 7 FOR 6), NUMERIC(6))
   AS bonus
```

Error cases

The following errors can occur when the DECRYPT function is executed:

- The length of the encrypted string is 0 or not an integral multiple of 16.
- The key *key* is a string with a length which is not 16 or it is not the key that was used for encryption.
- The decrypted value does not match the data type specified in the result (when, for example, a SMALLINT value is encrypted, but INTEGER was specified as the result type in the DECRYPT function (or vice versa)).

However, when the DECRYPT function is executed no check is made to see whether the decrypted result is assigned precisely the same data type as the encrypted value. Only the internal presentation of values is encrypted and decrypted, but no additional information.

Thus, for example, in SESAM/SQL the values of the data types INTEGER, CHARACTER(4), NUMERIC(4,0), DECIMAL(7,2) and REAL which are not equal to NULL all have an internal presentation with precisely 4 bytes.

Consequently a value of the data type INTEGER can be encrypted and decrypted to a value of the type CHAR(4) or REAL. The DECRYPT function does not return an error even if decryption is to the type NUMERIC(4,0). Depending on the decrypted value, however, an error can occur in a subsequent arithmetic operation.

5.2.22 DEE() - Table without columns

Function group: table function

The table function DEE() returns a table without columns with one row.

In SESAM/SQL there are no other tables of this kind. They can, for example, be used to analyze an expression without reference to a base table. No SQL privilege is required for reading with DEE().

```
DEE [()]
```

Result

The table without columns with one row.

Examples

This query returns details of SQL mode:

```
SELECT CURRENT_USER AS "Who am I",
       LOCALTIMESTAMP(3) AS "and what time is it, anyway"
   FROM TABLE(DEE())
```

The following query is executed for database k9 and could return a different time:

```
SELECT LOCALTIMESTAMP(3) AS "local time on catalog K9"

FROM TABLE(K9.DEE())
```

The following query expands table T by one row with NULL values:

```
SELECT * FROM T UNION JOIN TABLE(DEE())
```

5.2.23 ENCRYPT() - Encrypt data

Functon group: cryptographic function

ENCRYPT() encrypts values of any data type using the AES algorithm and a key of 128 bits (16 bytes) in Electronic Codebook Mode (ECM).

```
ENCRYPT ( expression , key )
```

expression

Expression whose value is to be encrypted.
The value may be of any data type, but not CHARACTER VARYING with length >= 31998 and not NATIONAL CHARACTER VARYING (16000).
expression may not be a multiple value with dimension > 1.

key

Key with which the value of *expression* is to be encrypted.
Alphanumeric string with a length of 16 characters, i.e. of the data type CHARACTER(16) or CHARACTER VARYING(n) where n >=16.
A NULL value of one of these data types is also permissible.

Result

If the value of *expression* or *key* is the NULL value, the result is the NULL value.

Otherwise:

The encrypted value of *expression*.

Data type: CHARACTER VARYING with a maximum length in accordance with the table on the next page.

Data type of *expression*	Data type of the result
CHAR(m)	VARCHAR(n) [1]
VARCHAR(m) where m <= 31998	VARCHAR(n) [1]
NCHAR(m)	VARCHAR(n) [2]
NVARCHAR(m) where m <= 15999	VARCHAR(n) [2]
SMALLINT, INTEGER	VARCHAR(16)
NUMERIC (up to 14 characters)	VARCHAR(16)
NUMERIC (15 to 30 characters)	VARCHAR(32)
NUMERIC (31 characters)	VARCHAR(48)

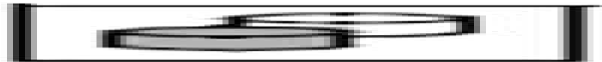
DECIMAL (up to 27 characters)	VARCHAR(16)
DECIMAL (28 to 31 characters)	VARCHAR(32)
FLOAT, REAL, DOUBLE PRECISION	VARCHAR(16)
DATE, TIME(3), TIMESTAMP(3)	VARCHAR(16)

Table 14: Data type of the result of ENCRYPT()

[1]Where n is the lowest integral multiple of 16 which is>= m + 2
[2]Where n is the lowest integral multiple of 16 which is >= 2*m + 2

> **i** If *expression* has a data type whose values can have different lengths (i.e. (NATIONAL) CHARACTER VARYING), the encrypted values can also have different lengths. However, the length of the encrypted value is always a multiple of 16 characters, see the table above.If, for example, *expression* has the data type VARCHAR(20), the result ENCRYPT() will have the data type VARCHAR(32); strings with 0 to 14 characters are encrypted in strings with 16 characters, strings with 15 to 20 characters in strings with 32 characters. The precise length of the unencrypted value cannot be determined from the encrypted value without knowledge of the key (it is encrypted together with the value).

Examples



The values of the SERVICE_PRICE column are encrypted in the SREC_ENCR column; the unencrypted values of the SERVICE_PRICE column are converted to NULL:

```
UPDATE service SET
   srec_encr=ENCRYPT
(service_price,'0123456789ABCDEF'),
   service_price = NULL WHERE
service_price IS NOT NULL
```

The REP_OF_VALUE function also enables multiple values to be encrypted in a string (see also "DECRYPT() - Decrypt data"):

```
ENCRYPT (REP_OF_VALUE(wages) ||
REP_OF_VALUE(bonus), :key)
```

5.2.24 EXTRACT() - Extract components of a time value

Function group: numeric function

EXTRACT() selects the specified component from a time value.
EXTRACT() uses the Gregorian calendar to do this, including the dates before its introduction on 10/15/1582.

```
EXTRACT ( component FROM expression )

component ::= { YEAR | MONTH | DAY | HOUR | MINUTE | SECOND |
                YEAR_OF_WEEK | WEEK_OF_YEAR | DAY_OF_WEEK | DAY_OF_YEAR }
```

component

Specification of the component. Permissible entries:

YEAR	selects the year of timestamp or date, e.g. 2013
MONTH	selects the month of the year of a timestamp or date, e.g. 2 for February
DAY	selects the day of the month of a timestamp or date, e.g. 25
HOUR	selects the hour of the day of a timestamp or of a time, e.g. 23
MINUTE	selects the minute of the hour of a timestamp or of a time, e.g. 58
SECOND	selects the second of the minute of a timestamp or of a time, e.g. 35.765
YEAR_OF_WEEK	determines the year in which the week of a timestamp or day lies, e.g. 2013
WEEK_OF_YEAR	determines the week of the year of a timestamp or date, e.g. 52
DAY_OF_WEEK	determines the day of the week of a timestamp or date, e.g. 3 for Wednesday
DAY_OF_YEAR	determines the day of the year of a timestamp or date, e.g. 365

expression

Time value expression. Permissible types are:

- TIMESTAMP is permissible for every *component*
- TIME with *component* HOUR, MINUTE or SECOND
- DATE with *component* YEAR, MONTH, DAY, YEAR_OF_WEEK, WEEK_OF_YEAR, DAY_OF_WEEK or DAY_OF_YEAR

expression may not be a multiple value with dimension > 1.

Result

When *expression* returns the NULL value, the result is the NULL value.

Otherwise:

The corresponding numeric value.

Datentyp:

DECIMAL(1,0)	with *component* `DAY_OF_WEEK`
DECIMAL(2,0)	with *component* `MONTH`, `DAY`, `HOUR`, `MINUTE`, `WEEK_OF_YEAR`
DECIMAL(3,0)	with *component* `DAY_OF_YEAR`
DECIMAL(4,0)	with *component* `YEAR` und `YEAR_OF_WEEK`
DECIMAL(5,3)	with *component* `SECOND`

Examples

Determining the current year number.

```
EXTRACT (YEAR FROM CURRENT_DATE)
```

Determining the day in the year.

```
EXTRACT (DAY_OF_YEAR FROM DATE '<date>')
```

Determining the current second.

```
EXTRACT (SECOND FROM CURRENT_TIME(3))
```

5.2.25 FLOOR() - Largest integer less than the value

Function group: numeric function

FLOOR() ("round down to the floor") determines the largest integer which is less than or equal to the specified numeric value. In the case of non-integer numeric values, FLOOR() always rounds down.

```
FLOOR ( expression )
```

expression

Fixed-point value of the type NUMERIC(p,s) or DECIMAL(p,s) if the number of decimal places s is greater than 0, otherwise a numeric expression.
expression may not be a multiple value with dimension > 1.

Result

When *expression* returns the NULL value, the result is the NULL value.

Otherwise:

The largest integer which is less than the specified numeric value.

Data type: NUMERIC(q+1,0) or DECIMAL(q+1,0) where q=MIN(31,p+1) if the number of decimal places s is greater than 0, otherwise like *expression*.

Examples

`FLOOR (3,14)` returns the value 3.

`FLOOR (-3,14)` returns the value -4.

`FLOOR (10,54)` returns the value 10.

5.2.26 HEX_OF_VALUE() - Present any value in hexadecimal format

Function group: string function

HEX_OF_VALUE() presents a value of any data type in hexadecimal format, i.e. in a string consisting of the hexadecimal characters 0,1,2,...,9,a,b,...,f.

This enables any bit patterns to be output in readable format.

```
HEX_OF_VALUE ( expression )
```

expression

Expression whose value is to be presented in hexadecimal format.
The data type may not be CHARACTER VARYING(n) with a maximum length of n > 16000 and not NATIONAL CHARACTER VARYING(n) with a maximum length of n > 8000.
expression may not be a multiple value with dimension > 1.

Result

If the value of *expression* is the NULL value, the result is the NULL value.

Otherwise:

The internal presentation of the value of *expression* in hexadecimal format as an alphanumeric string. Its length is specified in the table on the next page.

Data type: CHARACTER VARYING with a maximum length in accordance with the table on the next page.

Data type of *expression*	Data type of the result	Length of the result if not NULL
CHAR(n)	VARCHAR(2*n)	2*n
VARCHAR(n) where n <= 16000	VARCHAR(2*n)	0 to 2*n, even
NCHAR(n)	VARCHAR(4*n)	4*n
NVARCHAR(n) where n <= 8000	VARCHAR(4*n)	0 to 4*n, divisible by 4
SMALLINT	VARCHAR(4)	4
INTEGER	VARCHAR(8)	8
NUMERIC(p,s)	VARCHAR(2*p)	2*p
DECIMAL(p,s)	VARCHAR(q [1])	q [1]
REAL, FLOAT (<= 21 characters)	VARCHAR(8)	8
DOUBLE PRECISION, FLOAT (>= 22 characters)	VARCHAR(16)	16

DATE	VARCHAR(12)	12
TIME(3)	VARCHAR(16)	16
TIMESTAMP(3)	VARCHAR(28)	28

Table 15: Data types and lengths in the case of HEX_OF_VALUE()

[1]q=p+2 if p is even; q=p+1 if p is odd.

Examples

```
HEX_OF_VALUE (CAST (254 AS SMALLINT))
   00fe
HEX_OF_VALUE ('ABC')
   c1c2c3
```

Internal presentation of values in SESAM/SQL

The internal presentation of values which are not equal to NULL in SESAM/SQL as returned by the REP_OF_VALUE() and HEX_OF_VALUE() functions is similar to the internal presentation of corresponding values in other programming languages (e.g. COBOL, C).

SQL data_type	Sample value	internal presentation (hexadecimal format)
CHAR, VARCHAR EBCDIC string	'ABC'	`c1c2c3`
NCHAR, NVARCHAR UTF16 string	N'ABC'	`004100420043`
SMALLINT 2 bytes with binary presentation of value (2 Excess Code)	+300 -300	`012C` `fed4`
INTEGER 4 bytes with binary presentation of value (2 Excess Code)	+300 -300	`0000012c` `fffffed4`
NUMERIC(p,s) p bytes with EBCDIC characters for digits, sign in the last byte	+123.5 -123.5	`f1f2f3f5` `f1f2f3d5`
DECIMAL(p,s) FLOOR(p/2)[1] bytes with 2 digits each, last byte with 1 digit and sign	+123.5 -123.5	`01235c` `01235d`
REAL, FLOAT (<= 21 characters) 1 byte for sign and exponent, 3 bytes mantissa	+2.550625e+2 (=255 + 1/16)	`45ff1000`
DOUBLE PRECISION,	+2.5506250000e+2	`c5ff100000000000`

FLOAT (>=22 characters) 1 byte for sign and exponent for base 16, 7 bytes mantissa		
DATE 2 bytes each with year, month, day in binary format	DATE'2000-08-11'	`07d800008000b`
TIME(3) 2 bytes each with hours, minutes, seconds and milliseconds in binary format	TIME'12:34:56.123'	`000c00220038007b`
TIMESTAMP(3) Like DATE and TIME(3)	TIMESTAMP '2000-08-11 12:34:56.123'	`07d800008000b000c00` `220038007b`

Table 16: Overview of the internal presentation of values in SESAM/SQL

[1]FLOOR(p/2) is the largest whole number<= p/2

5.2.27 JULIAN_DAY_OF_DATE() - Convert date

Function group: numeric function

JULIAN_DAY_OF_DATE() returns the Julian day number which corresponds to a given date time value (see also the inverse function “DATE_OF_JULIAN_DAY()” on "DATE_OF_JULIAN_DAY() - Convert Julian day number").

The Julian day number for the 24th November 4714 B.C. (in accordance with the Gregorian calendar) is “0”.

The Julian day number for a later date is the number of days which have passed between the 24th November 4714 B.C. and the later date. For example, the DATE '0001-01-01' corresponds to the Julian day number “1721426”, the DATE '9999-12-31' corresponds to the Julian day number “5373484”.

> DATE_OF_JULIAN_DAY() and JULIAN_DAY_OF_DATE() are inverse functions. When, for example, a constraint exists in the form JULIAN_DAY_OF_DATE(*column*) < *:user_variable*, the SQL Optimizer can then convert this constraint internally to the constraint *column* < DATE_OF_JULIAN_DAY(*:user_variable*) in order to permit the use of indexes on *column*. Consequently *:user_variable* may only contain values which are permitted as an argument of DATE_OF_JULIAN_DAY(). This also applies for any constant expressions in place of *:user_variable*.

```
JULIAN_DAY_OF_DATE ( expression )
```

expression

Time value expression whose evaluation gives a value of the DATE data type; value is between 0001-01-01 and 9999-12-31.
expression may not be a multiple value with dimension > 1.

Result

When *expression* returns the NULL value, the result is the NULL value.

Otherwise:

the result is the Julian day number which represents the date which results from *expression*.

Data type: INTEGER

Examples

```
JULIAN_DAY_OF_DATE(DATE'2000-01-01')
```

```
2451545
```

To create a view which outputs the orders for the last two weeks:

```
CREATE VIEW orders AS SELECT * FROM job

WHERE todate >= DATE_OF_JULIAN_DAY(JULIAN_DAY_OF_DATE(CURRENT_DATE)-14)
```

5.2.28 LOCALTIME(3) - Current local time

Function group: time function

LOCALTIME(3) returns the current local time.

```
LOCALTIME(3)
```

Result

Current local time

If several time functions are included in a statement, they are all executed simultaneously (see section “Time functions”).

Data type: TIME

5.2.29 LOCALTIMESTAMP(3) - Current local time stamp

Function group: time function

LOCALTIMESTAMP(3) returns the current local time stamp.

```
LOCALTIMESTAMP(3)
```

Result

Current local time stamp

If several time functions are included in a statement, they are all executed simultaneously (see section "Time functions").

Data type: TIMESTAMP

5.2.30 LOWER() - Convert uppercase characters

Function group: string function

LOWER() converts uppercase characters in a string to lowercase characters.

```
LOWER ( expression )
```

expression

Alphanumeric expression or national expression.

Result

When *expression* returns the NULL value, the result is the NULL value.

Otherwise:

- If *expression* is an alphanumeric expression, the result is a copy of the string which results from the evaluation of *expression*, uppercase letters of the SESAM/SQL character repertoire (see "SESAM/SQL character repertoire") being replaced by equivalent lowercase letters (A-Z without umlauts and ß).
- If *expression* is a national expression, uppercase letters are replaced by equivalent lowercase letters in accordance with the Unicode rules (as with the XHCS function `tolower`).

Data type: like *expression*

Examples

```
SELECT LOWER(strasse) FROM kunde WHERE knr=100
```

```
otto-hahn-ring 6
```

`LOWER('Ä')` returns the value 'Ä'.

`LOWER(NX'00C4')` returns the value NX'00E4' (which corresponds to 'ä') because the Unicode rules are used.

5.2.31 MAX() - Determine largest value

Function group: aggregate function

MAX() determines the largest value in a set of values. NULL values are ignored. Comparing alphanumeric values, national values, numeric values and time values is described in section “Comparison of two rows”.

```
MAX ([ALL | DISTINCT ] expression )
```

ALL / DISTINCT

ALL or DISTINCT can be specified but has no effect on the result.

expression

Numeric expression, alphanumeric expression, national expression or time value expression (see section “Aggregate functions” for information on restrictions).

Result

If the set of values returned by *expression* is empty, the result or the result for this group is the NULL value.

Otherwise:

Without GROUP BY clause:

Determines the largest value in the set of values returned by *expression* (see “Calculating aggregate functions”).

With GROUP BY clause:

Returns the largest value of each group.

Data type: like *expression*

Examples

SELECT without GROUP BY:

Query the highest service price for order 211 in the SERVICE table (result: 1200):

```
SELECT MAX(service_price) FROM service WHERE order_num=211
```

SELECT with GROUP BY:

Determine the highest service price for each order number:

```
SELECT order_num, MAX(service_price) FROM service GROUP BY order_num
   order_num
```

```
200          1500
211          1200
250          1200
```

5.2.32 MIN() - Determine lowest value

Function group: aggregate function

MIN() determines the smallest element in a set of values. NULL values are ignored. Comparing alphanumeric values, national values, numeric values and time values is described in section “Comparison of two rows”.

MIN ([ALL | DISTINCT] *expression*)

ALL / DISTINCT

ALL or DISTINCT can be specified but has no effect on the result.

expression

Numeric expression, alphanumeric expression, national expression or time value expression (see section “Aggregate functions” for information on restrictions).

Result

If the set of values returned by *expression* is empty, the result or the result for this group is the NULL value.

Otherwise:

Without GROUP BY clause:

Determines the lowest value in the set of values returned by *expression* (see “Calculating aggregate functions”).

With GROUP BY clause:

Returns the lowest value of each group.

Data type: like *expression*

Examples

SELECT without GROUP BY:

Query the lowest service price for order 211 in the SERVICE table (result: 50):

```
SELECT MIN(service_price) FROM service WHERE order_num=211
```

SELECT with GROUP BY:

Determine the lowest service price for each order number:

```
SELECT order_num, MIN(service_price) FROM service GROUP BY order_num
   order_num
```

```
200        75
211        50
250       125
```

5.2.33 MOD() - Remainder of an integer division (modulo)

Function group: numeric function

MOD() determines the remainder of a division of two integers.

`MOD (` *dividend, divisor* `)`

dividend `::=` *expression*
divisor `::=` *expression*

dividend

Integer numeric expression (SMALLINT, INTEGER, NUMERIC(p,0), DECIMAL(p,0)) for the dividend of the division.

divisor

Integer numeric expression (SMALLINT, INTEGER, NUMERIC(q,0), DECIMAL(q,0)) for the divisor of the division. *divisor* may not be 0.

dividend and *divisor* may not be multiple values with a dimension > 1.

Result

When *dividend* or *divisor* returns the NULL value, the result is the NULL value.

When *dividend* returns the value 0, the result is 0.

Otherwise:

The result is the integer remainder of the division *dividend* / *divisor* with the same sign as *dividend*.

Data type: like *divisor*.

Examples

`MOD (3,2)` returns the value 1.

`MOD (-3,-2)` returns the value -1.

5.2.34 NORMALIZE() - Convert national string to normal form

Function group: string function

The encoding of a character in Unicode is not unambiguous, i.e. more than one coding can exist for a character, see the " Core manual".

A typical example of this is provided by the German umlauts. For example, the character Ä has both the code point `U+00C4` (composed form) and the code point combination `U+0041` and `U+0308` (decomposed form). In normalized presentation forms these differences do not occur. If two normalized strings differ, it is in their different code point presentations.

NORMALIZE() converts a national string with national characters which have code points in the range `U+0000` through `U+2FFF` to a normalized form. Other characters, e.g. surrogates, remain unchanged.

`NORMALIZE (` *expression* `[ ,{ NFC | NFD } [ ,` *length* `]])`

length `::=` *unsigned_integer*

---

expression

National-expression. Its evaluation returns a national string (data type NCHAR or NVARCHAR) in normalized form.
expression may not be a multiple value with dimension > 1.

NFC / NFD

Normalization forms C ("Canonical Decomposition followed by Canonical Composition") and D ("Canonical Decomposition") of the Unicode standard.

NFC maps all code points which together result in a character to the corresponding code point. NFD breaks down each "compound" character into its component parts, to the basic characters and the diacritical characters linked to these. The order of the linked diacritical characters is strictly defined here.

length

Maximum length of the normalized presentation in code units.

Length not specified:
The result can have a length of up to 16000 code units, depending on *expression*.

Result

If the value of *expression* is the NULL value, the result is the NULL value.

Otherwise:

The normalized presentation of the value of *expression*.
The following applies: length of the normalized presentation (NFC) <= length of the nonnormalized presentation <= length of the normalized presentation (NFD).

If the length of the normalized presentation is greater than the specified *length*, the function is aborted with SQLSTATE.

Data type: NVARCHAR(MIN(2*n,16000)),
where n is the length of the argument data type NCHAR(n) or NVARCHAR(n). For an argument of type NCHAR the data type is NVARCHAR too.

Example

The following search condition normalizes a user name in order to detect unwanted users who can log in various presentation forms.

```
... WHERE NORMALIZE(:customer,NFC)

          NOT IN (SELECT name FROM unwanted_customers)
```

5.2.35 OCTET_LENGTH() - Determine string length

Function group: numeric function

OCTET_LENGTH() determines the number of bytes in a string.

```
OCTET_LENGTH ( expression )
```

expression

Alphanumeric expression or national expression. Its evaluation must return either an alphanumeric string (data type CHAR or VARCHAR) or a national string (data type NCHAR or NVARCHAR). *expression* may not be a multiple value with dimension > 1. See also section “Compatibility between data types”.

Result

If the string contains the NULL value, the result is the NULL value.

Otherwise:

The result is the number of bytes in the string.

Data type: INTEGER

Examples

Determine the number of bytes in the alphanumeric string 'only' (result: 4).

```
OCTET_LENGTH ('only')
```

Determine the number of bytes in the national string 'An evening in old München' (result: 16).

```
OCTET_LENGTH (U&'An evening in old M\00FCnchen')
```

5.2.36 POSITION() - Determine string position

Function group: numeric function

POSITION() determines the position of a string in another string.

```
POSITION ( expression IN expression [USING CODE_UNITS])
```

expression

Alphanumeric expression or national expression. Its evaluation must return either an alphanumeric string (data type CHAR or VARCHAR) or a national string (data type NCHAR or NVARCHAR). *expression* may not be a multiple value with dimension > 1. See also section “Compatibility between data types”.

Result

In the following description of the possible results, *string1* is the string whose position is to be determined, and *string2* is the other string.

string1 and/or *string2* contains the NULL value:

The result is the NULL value.

string1 has the length 0:

The result is 1.

string1 is in *string2*:

The result is 1 greater than the number of characters (for CHAR/VARCHAR) or code units (for NCHAR /NVARCHAR) of *string2* which precede the first character or the first code unit of *string1*.

Otherwise*:* The result is 0.

Data type: INTEGER

Examples

Determine the position of the string 'nett' in the string 'annette' (result: 3):

```
POSITION ('nett' IN 'annette')
```

Determine the position of the string 'Vogue' (result: 26):

```
POSITION('Vogue' IN 'If it''s in vogue it''s in Vogue.')
```

Determine the position of the string 'Puss' in the string 'boots' (result: 0):

```
POSITION ('Puss' IN 'boots')
```

5.2.37 REP_OF_VALUE() - Present any value as a string

Function group: string function

REP_OF_VALUE() presents a value of any data type as a alphanumeric string (sequence of bytes).

```
REP_OF_VALUE ( expression )
```

expression

Expression whose value is to be presented as a string.
expression may not be a multiple value with dimension > 1.

Result

If the value of *expression* is the NULL value, the result is the NULL value.

Otherwise:

The internal presentation of the value of *expression* as a sequence of bytes in an alphanumeric string. For the internal presentation of the various data types, see table 16.

Data type: CHARACTER VARYING(n), where the maximum length n of the data type *expression* is dependent on the values shown in the table on the next page.

Data type of *expression*	Data type of the result	Length of the result if not NULL
CHAR(n)	VARCHAR(n)	n
VARCHAR(n)	VARCHAR(n)	0 to n
NCHAR(n)	VARCHAR(2*n)	2*n
NVARCHAR(n)	VARCHAR(2*n)	0 to 2*n, even
SMALLINT	VARCHAR(2)	2
INTEGER	VARCHAR(4)	4
NUMERIC(p,s)	VARCHAR(n)	p
DECIMAL(p,s)	VARCHAR(q [1])	q [1]
REAL, FLOAT (<= 21 characters)	VARCHAR(4)	4
DOUBLE PRECISION, FLOAT (>= 22 characters)	VARCHAR(8)	8
DATE	VARCHAR(6)	6
TIME(3)	VARCHAR(8)	8

TIMESTAMP(3)	VARCHAR(14)	14

Table 17: Data types and lengths in the case of REP_OF_VALUE

[1]q=(p + 2)/2 if p is even; q=(p + 1)/2 is p is odd.

Examples

```
REP_OF_VALUE (CAST (254 AS SMALLINT))
```

254 is presented in binary format as X'00fe' (2 bytes).
These 2 bytes (not printable) are also the result of the expression.

```
REP_OF_VALUE ('ABC')
```

The result is the string 'ABC'.

5.2.38 SIGN() - Determine sign

Function group: numeric function

SIGN() determines the sign of a numeric value.

```
SIGN ( expression )
```

expression

Numeric expression.
expression may not be a multiple value with dimension > 1.

Result

When *expression* returns the NULL value, the result is the NULL value.

When *expression* returns the value 0, the result is 0.

When *expression* is > 0, the result is 1.

When *expression* is < 0, the result is -1.

Data type: DECIMAL(1,0)

Examples

`SIGN (3,14)` returns the value 1.

`SIGN (-3,14)` returns the value -1.

5.2.39 SUBSTRING() - Extract substring

Function group: string function

SUBSTRING() extracts a substring from a string.

```
SUBSTRING ( expression FROM startposition [FOR substring_length ][USING CODE_UNITS])
```

expression

Alphanumeric expression or national expression. Its evaluation must return either an alphanumeric string (data type CHAR or VARCHAR) or a national string (data type NCHAR or NVARCHAR). See also section "Compatibility between data types".

startposition

Numeric expression whose data type is DECIMAL or NUMERIC without decimal places (SCALE 0), SMALLINT or INTEGER. The evaluation of *startposition* returns an integer or a fixed-point number without decimal places. It cannot be a multiple value with a dimension greater than 1.

startposition specifies the position of a character in or outside the string returned when *expression* is evaluated. *startposition* specifies the character as of which the substring is to be extracted.

substring_length

Numeric expression whose data type is DECIMAL or NUMERIC without decimal places (SCALE 0), SMALLINT or INTEGER. The evaluation of *substring_length* returns an integer or a fixed-point number without decimal places. The value of *substring_length* cannot be less than 0. It cannot be a multiple value with a dimension greater than 1.

substring_length specifies the maximum length of the substring.

Result

In the following description of the possible results, *string* is the string returned when *expression* is evaluated.

The result is the NULL value when *expression, startposition* and/or *substring* have the NULL value.

The result is a string with a length of 0 when any of the following conditions are fulfilled:

- *startposition* is greater than the number of characters in *string.*
- *string* has the length 0.
- *substring_length* is 0.
- The sum of *startposition* and *substring_length* is <= 1.

Otherwise:

The result is a substring of *string*. The order in which the characters occur corresponds to the order of the characters in *string*. The substring contains the number of characters specified by *startposition* and *substring_length*.

substring_length is specified and *startposition* >=1:

The substring contains *substring_length* characters (but not beyond the last character of *string)*, beginning with the character of *string* specified by *startposition*.

substring_length is specified and *startposition* < 1:

The substring contains (*startposition* + *substring_length-1)* characters (but not beyond the last character of *string)*, beginning with the first character of *string*.

substring_length is not specified and *startposition* >= 1

The substring contains, as of *startposition*, all the characters in the string up to the last character.

substring_length is not specified and *startposition* < 1

The whole string is extracted.

Data type: If *expression* has the alphanumeric data type CHAR(*n*) or VARCHAR(*n*), the result has the alphanumeric data type VARCHAR(*n*).

If *expression* has the national data type NCHAR(*n*) or NVARCHAR(*n*), the result has the national data type NVARCHAR(*n*).

Examples

A substring is to be extracted from the string 'The Poodle Parlor'. 'The Poodle Parlor' is the company name of a customer in the CUSTOMERS table.

startposition is > 1*, substring_length* is specified:

```
SELECT SUBSTRING (company FROM 6 FOR 4) FROM customers WHERE cust_num=105
```

The result is the string 'Poodle'.

startposition is 0, *substring_length* is specified:

```
SELECT SUBSTRING (company FROM 0 FOR 5) FROM customers WHERE cust_num=105
```

The result is the string 'The' with a length of (0+4-1) = 3.

startposition is <0 and (*startposition* + *substring_length -1)* is greater than the length of *string*:

```
SELECT SUBSTRING (company FROM -2 FOR 20) FROM customers WHERE

cust_num=105
```

The result is the string 'The Poodle Parlor'.

startposition is > 1, *substring_length* is not specified:

```
SELECT SUBSTRING (company FROM 6) FROM customers WHERE cust_num=105
```

The result is the string 'Poodle Parlor'.

startposition is greater than the number of characters in *string*.

```
SELECT SUBSTRING (company FROM 15 FOR 5) FROM customers WHERE cust_num=105
```

The result is a string with a length of 0.

5.2.40 SUM() - Calculate sum

Function group: aggregate function

SUM() calculates the sum of all the values in a set. NULL values are ignored.

```
SUM ([ALL | DISTINCT ] expression )
```

ALL

All values are taken into account, including duplicate value.

DISTINCT

Only unique values are taken into account. Duplicate values are ignored.

expression

Numeric expression (see section “Aggregate functions” for information on restrictions).

Result

If the set of values returned by *expression* is empty, the result or the result for this group is the NULL value.

Otherwise:

Without GROUP BY clause:

Calculates the sum of the values returned by *expression*(see “Calculating aggregate functions”).

With GROUP BY clause:

Returns the sum of the values in the derived column of each group.

Data type: like *expression* with the following number of digits:

Integer or fixed-point number:

The total number of significant digits is 31, the number of digits to the right of the decimal point remains the same.

Floating-point number:

The total number of significant digits corresponds to 21 binary digits for REAL numbers and 53 for DOUBLE PRECISION.

If the sum of the values is too large for this data type, an error message is issued.

Example

Calculate the sum of the parts for each item number in the PURPOSE table:

```
SELECT item_num, SUM(number) FROM purpose GROUP BY item_num
   item_num
     1         4
   120        27
   200        20
```

5.2.41 TRANSLATE() - Transliterate / transcode string

Function group: string function

TRANSLATE() transliterates, i.e. converts, an alphanumeric string into a national string or vice versa, see the “ Core manual”.

TRANSLATE() transcodes, i.e. converts, a string in the character set UTFE to a national string in the character set UTF-16 or vice versa, see the “ Core manual”.

```
TRANSLATE ( expression
        USING [[ catalog. ]INFORMATION_SCHEMA.] transname [DEFAULT character ] [ ,length ])


character::= expression
length::= unsigned_integer
```

expression

Alphanumeric expression or national expression.
Its evaluation returns either an alphanumeric string or a national string. See also section “Compatibility between data types”.
expression may not be a multiple value with dimension > 1.

transname

Unqualified Name for a transliteration of EBCDIC to Unicode (character set UTF-16) and vice versa or for a transcoding of UTF-EBCDIC to UTF-16 and vice versa.

In SESAM/SQL all transliteration names are predefined. They are either the CCS names which are defined in the BS2000 subsystem XHCS for transliteration between EBCDIC and UTF-16 or CATALOG_DEFAULT for transliteration in the preselected database if CODE_TABLE is not set to `_NONE_` for the latter (see CREATE /ALTER CATALOG statements in the “ SQL Reference Manual Part 2: Utilities”). The CCS name can be up to 8 characters long.

When *expression* is an alphanumeric expression and the transliteration name `UTFE` (!) is specified, *expression* is transcoded from UTF-EBCDIC (character set `UTFE`) to the character set UTF-16.

When *expression* is a national expression (i.e. the character set is `UTF-16`) and the transliteration name `UTFE` is specified, *expression* is transcoded from UTF-16 to the character set UTFE.

Transliteration and transcoding can be qualified by a database name and the schema name INFORMATION_SCHEMA, otherwise the INFORMATION_SCHEMA of the predefined database is assumed.

character

With *character* you can define a substitute character which is to be output in place of characters which cannot be processed with the specified *transname*. If you have not specified `DEFAULT` *character* and *expression* contains a character that cannot be processed with the specified *transname*, the containing SQL statement is

aborted with SQLSTATE.If *expression* has the alphanumeric data type CHAR or VARCHAR, the substitute character must have the national data type NCHAR(1) or NVARCHAR(*n*) with *n*>=1.If *expression* has the national data type NCHAR or NVARCHAR, the substitute character must have the alphanumeric data type CHAR(1) or VARCHAR(*n*) with *n*>=1.

length

Maximum length of the transliterated or transcoded string in code units.

1 <= *length* <= 16000 when *expression* is an alphanumeric string
(transliteration name is an EBCDIC character set or `UTFE`).

1 <= *length* <= 32000 when *expression* is a national string
(transliteration name is an EBCDIC character set).

Length not specified:
The result has the maximum possible length (see above).

Result

If *expression* and/or *character* return NULL, the result is NULL.

Otherwise:

The result is the string with the specified or maximum length which results from the transliteration or transcoding of *expression*.

If the substitute character had to be used in the transliteration, the warning `SQLSTATE '01SBB'` is issued.

When the length of the transliterated or transcoded string is greater than the specified or maximum *length*, the function is aborted with SQLSTATE.

Data type:

If *expression* has the alphanumeric data type CHAR(*n*) or VARCHAR(*n*), the result has the national data type NVARCHAR(*n*).

If *expression* has the national data type NCHAR or NVARCHAR, the result of the transliteration has the alphanumeric data type VARCHAR(*n*) and, in the case of transcoding, the national data type NVARCHAR(*n*) .

Examples

The specified national string is to be transliterated by transliterating EDF03IRV to the standard BS2000 character set. Non-displayable characters are represented as question marks.

```
TRANSLATE (NX'0041004200430308' USING

WORLD_CUST.INFORMATION_SCHEMA.EDF03IRV DEFAULT '?')
```

The result ist the string 'ABC?'.

The specified alphanumeric string is to be interpreted as a string with the character set UTF-EBCDIC and to be transcoded to the Unicode character set UTF-16.

```
TRANSLATE ('ABC' USING UTFE)
```

```
004100420043
```

Interprets a file `NAMETITEL.TXT` in the character set UTFE (created, e.g., with UNLOAD) as a CSV file.

```
CREATE VIEW MYVIEW(x,y) AS
SELECT TRANSLATE(name USING UTFE), TRANSLATE(titel USING UTFE)
FROM TABLE(CSV(FILE 'NAMETITEL.TXT' DELIMITER ';',CHAR(25),VARCHAR(16)))
AS T(name,titel)
```

5.2.42 TRIM() - Remove characters

Function group: string function

TRIM() removes leading and/or trailing characters of a string.

```
TRIM ([[LEADING | TRAILING | BOTH][ character ] FROM] expression)

character ::= expression
```

character / *expression*

character and *expression* are either both alphanumeric expressions (data type CHAR or VARCHAR) or both national expressions (data type NCHAR or NVARCHAR). Neither of the operands may be a multiple value with a dimension greater than 1. The value of *character* has the length 1. If you do not specify *character*, the default is a blank ().

FROM

FROM operator; you can only specify FROM is you also specify LEADING, TRAILING or BOTH and/or *character*.

Result

If *character* and/or *expression* returns the NULL value, the result is the NULL value.

Otherwise:

The result is a copy of the string returned when *expression* is evaluated, except that leading and/or trailing characters that correspond to the value of *character* are removed. Whether leading or trailing characters are removed depends on whether you specify LEADING, TRAILING or BOTH:

LEADING: Leading characters are removed.

TRAILING: Trailing characters are removed.

BOTH: Leading and trailing characters are removed. BOTH is the default.

Data type:

If *expression* has the alphanumeric data type CHAR(*n*) or VARCHAR(*n*), the result has the alphanumeric data type VARCHAR(*n*).

If *expression* has the national data type NCHAR(*n*) or NVARCHAR(*n*), the result has the national data type NVARCHAR(*n*).

Examples

The following examples are equivalent and return 'ABC'.

```
TRIM(' ABC ')

TRIM (BOTH ' ' FROM ' ABC ')
```

The following example returns 'BLE WAS I ERE I SAW ELB'.

```
TRIM (BOTH N'N' FROM N'NURDUGUDRUN')
```

A record is inserted in the table PROFESSORS. The form_of_address column in the table has the data type VARCHAR(50). It is to receive the value 'Professor'.
The corresponding COBOL user variable has the data type PIC X(50). To ensure that only the value 'Professor' rather than the value 'Professor...' with 36 trailing characters is transferred, you use the TRIM string function:

```
INSERT INTO professors (..., form_of_address, ...)

VALUES (..., TRIM (TRAILING FROM :FORM_OF_ADDRESS), ...)
```

5.2.43 TRUNC() - Remove decimal places

Function group: numeric function

TRUNC() determines the integer share of a numeric value.

TRUNC() performs no rounding in the case of non-integer values.

```
TRUNC ( expression )
```

expression

Numeric expression.
expression may not be a multiple value with dimension > 1.

Result

When *expression* returns the NULL value, the result is the NULL value.
Otherwise:

expression >= 0: the largest integer which is less than or equal to the specified numeric FLOOR(*ausdruck)*.

expression < 0: the smallest integer which is greater than or equal to the specified numeric value, i.e. CEILING(*expression*).

Data type:	NUMERIC(p-s,0) DECIMAL(q-s,0)	for data type of *expression* NUMERIC(p,s) or DECIMAL(q,s) where p,q > s
	like *expression*	for data type of *expression* integer numeric (SMALLINT, INTEGER, NUMERIC(p,0), DECIMAL(q,0) or REAL, DOUBLE PRECISION, FLOAT

Examples

`TRUNC (3,14)` returns the value 3.

`TRUNC (-3,14)` returns the value -3.

5.2.44 UPPER() - Convert lowercase characters

Function group: string function

UPPER() converts the lowercase characters in a string to uppercase characters.

```
UPPER ( expression )
```

expression

Alphanumeric expression or national expression.

Result

When *expression* returns the NULL value, the result is the NULL value.

Otherwise:

- If *expression* is an alphanumeric expression, the result is a copy of the string which results from the evaluation of *expression*, lowercase letters of the SESAM/SQL character repertoire (see "SESAM/SQL character repertoire") being replaced by equivalent uppercase letters (a-z without umlauts and ß).
- If *expression* is a national expression, lowercase letters are replaced by equivalent uppercase letters in accordance with the Unicode rules (as with the XHCS function `toupper`).

Data type: like *expression*

Examples

```
SELECT UPPER(city) FROM customers WHERE cust_num=100
```

Returns the string 'MUNICH'.

```
UPPER('ä')
```

Returns the value 'ä'.

```
UPPER(NX'00E4')
```

Returns the value NX'00C4' (which corresponds to 'Ä') because the Unicode rules are used.

5.2.45 VALUE_OF_HEX() - Present hexadecimal format as a value

Function group: string function

The VALUE_OF_HEX() function returns a value of the specified data type from the internal presentation provided in hexadecimal format.
It is the inverse function of HEX_OF_VALUE().

VALUE_OF_HEX (*expression* , *data_type*)

expression

The internal presentation of the result value in hexadecimal format.
The value of *expression* may only contain the characters '0' through '9', 'a' through 'f' and 'A' through 'F'.
expression must have the data type CHARACTER(n) (n even) or CHARACTER VARYING(n).
Its value must either be the NULL value or have a length which suits the data type *data_type* (see the table on the next page). The data type of *expression* must permit values of this length or of the maximum length.
expression may not be a multiple value with dimension > 1.

data_type

Data type of the value (without *dimension* specification), *expression* being the presentation in hexadecimal format.
The data type may not be CHARACTER VARYING(n) with a maximum length of n > 16000 and not NATIONAL CHARACTER VARYING(n) with a maximum length of n > 8000.

Result

If the value of *expression* is the NULL value, the result is the NULL value.

Otherwise:

The value of the specified *data_type* whose internal presentation in hexadecimal format is the value of *expression*.
For the internal presentation of the various data types, see table 16.

Data type: the specified *data_type*

i When this function is executed, no check is made to see whether *data_type* is the same data type which was used beforehand for the corresponding presentation in internal formal using HEX_OF_VALUE().

Length of *expression* in characters	*data_type*
2*n	CHAR(n)
0 to 2*n	VARCHAR(n)
4*n	NCHAR(n)

0 to 4*n, divisible by 4	NVARCHAR(n)
4	SMALLINT
8	INTEGER
2*p	NUMERIC(p,s)
q [1]	DECIMAL(p,s)
8	REAL, FLOAT (<= 21 characters)
16	DOUBLE PRECISION, FLOAT (>= 22 characters)
12	DATE
16	TIME(3)
28	TIMESTAMP(3)

Table 18: Data types and lengths in the case of VALUE_OF_HEX

[1]q=p+2 if p is even; q=p+1 if p is odd.

Examples

```
VALUE_OF_HEX ('00fe', SMALLINT)
   254
VALUE_OF_HEX ('c1c2c3', CHAR(3))
   ABC
```

5.2.46 VALUE_OF_REP() - Present a string as a value

Function group: string function

The VALUE_OF_REP() function returns a value of the specified data type from the internal presentation provided (sequence of bytes).
It is the inverse function of REP_OF_VALUE().

VALUE_OF_REP (*expression* , *data_type*)

expression

The internal presentation of the result value. For the internal presentation of the various data types, see table 16.
expression must have the data type CHARACTER(n) (n even) or CHARACTER VARYING(n).
Its value must either be the NULL value or have a length which suits the data type *data_type* (see the table on the next page). The data type of *expression* must permit values of this length or of the maximum length.
expression may not be a multiple value with dimension > 1.

data_type

Data type of the value (without *dimension* specification), *expression* being the internal presentation.

Result

If the value of *expression* is the NULL value, the result is the NULL value.

Otherwise:

The value of the specified *data_type* whose internal presentation is the value of *expression*.

Data type: the specified *data_type*

When this function is executed, no check is made to see whether *data_type* is the same data type which was used beforehand for the corresponding presentation in internal formal using REP_OF_VALUE().

Length of *expression* in characters	*data_type*
n	CHAR(n)
0 to n	VARCHAR(n)
2*n	NCHAR(n)
0 to 2*n, even	NVARCHAR(n)
2	SMALLINT
4	INTEGER

p	NUMERIC(p,s)
q [1]	DECIMAL(p,s)
4	REAL, FLOAT (<= 21 characters)
8	DOUBLE PRECISION, FLOAT (>= 22 characters)
6	DATE
8	TIME(3)
14	TIMESTAMP(3)

Table 19: Data types and lengths in the case of VALUE_OF_REP

[1]q=(p + 2)/2 if p is even; q=(p + 1)/2 if p is odd

Examples

```
VALUE_OF_REP (X'00fe', SMALLINT)
   254
VALUE_OF_REP ('ABC', CHAR(3))
   ABC
```

5.3 Predicates

Predicates are components of search conditions (see section “Search conditions”).

A predicate consists of operands and operators. Predicates can be grouped together as follows according to the operator involved:

- Comparison of two rows
- Quantified comparison (comparison with the rows of a table)
- BETWEEN predicate (range query)
- CASTABLE predicate (convertibility check)
- IN predicate (elementary query)
- LIKE predicate (simple pattern comparison)
- LIKE_REGEX predicate (pattern comparison with regular expressions)
- NULL predicate (comparison with the NULL value)
- EXISTS predicate (existence query)

The individual groups are described below in the above order.

A predicate returns the truth value true, false or unknown. The value of a predicate is calculated by calculating the values of the operands and applying the appropriate operators to the calculated values. In certain cases an operand is not calculated at all, or is only partially calculated, if this is enough to determine the result.

The diagram below provides a simplified overview of the syntax of all predicates:

```
praedicate ::=

{
  row comparison_op row |

  vector_column comparison_op expression |

  row comparison_op { ALL | SOME | ANY } subquery |

  row [NOT] BETWEEN row AND row |

  vector_column [NOT] BETWEEN expression AND expression |

  expression IS [NOT] CASTABLE AS data_type |

  row [NOT] IN { subquery | (row ,...) } |

  vector_column [NOT] IN (expression , expression ,...) |

  operand [NOT] LIKE pattern [ESCAPE character...] |

  operand [NOT] LIKE_REGEX regular_expression [FLAG flag] |

  expression IS [NOT] NULL |

 EXISTS subquery

}
```

row ::= { (*expression* , ...) | *expression* | *subquery* }
vector_column ::= [*table*.]{ *column[min..max]* | *column*(*min..max*) }
comparison_op ::= { = | < | > | <= | >= | <> }
operand ::= *expression*
pattern ::= *expression*
character ::= *expression*
regular_expression ::= *expression*
flag ::= *expression*

5.3.1 Comparison of two rows

Two rows are compared lexicographically according to a comparison operator. If both rows only have one column, you will obtain the normal comparison of two values.

{ *row comparison_op row* | *vector_column comparison_op expression* }

row ::= { (*expression* ,...) | *expression* | *subquery* }

vector_column ::= [*table* .]{ *column[min..max]* | *column* (*min..max* }

comparison_op ::= { = | < | > | <= | >=| <> }

row

Operands for comparison.

Each *expression* in *row* must be atomic. The row consists of the *expression* values in the order specified. A single *expression* therefore returns a row with one column.

subquery must return a table without multiple columns, and with at most one row. This row is the comparison operand. If the table returned is empty, the comparison operand is a row with the NULL value in each column.

The rows to be compared must have the same number of columns and the corresponding columns of the left and right rows must have compatible data types (see section “Compatibility between data types”).

vector_column

A multiple column, which is compared according to special rules. The column specification may not be an external reference.

> Any square brackets shown here in italics are special characters, and must be specified in the statement.

expression

The *expression* value must be atomic and its data type must be compatible with the data type of the *vector_column* occurrences (see section “Compatibility between data types”).

comparison_op

Comparison operator.

=	Compare whether two values are the same
<	Compare whether one value is smaller than the other
>	Compare whether one value is greater than the other
<=	Compare whether one value is smaller than or equal to the other

>=	Compare whether one value is greater than or equal to the other
<>	Compare whether two values are not equal

Result

row comparison_op row

If rows with more than one column are compared , the lexicographical comparison rules for rows will apply (see section “Comparison rules”.

If single-column rows are compared, the comparison rules will depend on the data type of the columns (see section “Comparison rules”).

vector_column comparison_op expression

Each occurrence of *vector_column* is compared with *expression* according to the comparision rules for the data type (see section “Comparison rules” below). The comparison results are combined with OR.

Example

If X is a multiple column with 3 elements, the comparison

```
X[1..3] >= 13
```

is equivalent to the following comparisons:

```
X[1] >= 13 OR X[2] >= 13 OR X[3] >= 13
```

5.3.1.1 Comparison rules

The way in which a comparison operation is performed depends on the operands. Lexicographical comparison rules apply to the comparison of rows with more than one column; in the case of comparisons of single-column rows and values, the comparison rules are based on the data types. These rules are collected in the following paragraphs.

Lexicographical comparison

The result of the comparison is derived from the comparison of the values in corresponding columns of the two rows. The values in columns situated further to the right are only significant if the values in all the previous columns are the same for both operands (sorting in the lexicon also occurs according to these comparison rules).

In formal terms this means:
For a comparison of two rows with the comparison operator `OP` that is either “<” or “>”, with column values `L1`, `L2,...,Ln` in the left-hand operand and with column values `R1,R2,...,Rn` in the right-hand operand, the result is the truth value true or false or unknown respectively, if there is an `i` index between `1` and `n`, so that all the comparisons

```
L1     =   R1
L2     =   R2
 .     .    .
 .     .    .
 .     .    .
L(i-1) = R(i-1)
```

return the truth value true, and the comparison

`Li OP Ri`

returns the truth value true, or false, or unknown, respectively.

The individual comparisons are carried out as described below, depending on the data type.

Please note the following:

- The value in one of the columns may well be NULL without the result of the whole comparison being unknown.

  For example the comparison `(1,CAST(NULL AS INT)) < (2,0)` the truth value true as a result. The second column is ignored in the comparison because the values of the first columns are already different.
- Not all columns need to be relevant for the comparison result. You should not, therefore, rely on all of the columns in both rows always being evaluated.
- The comparison `(L1, L2, ..., Ln) = (R1, R2, ..., Rn)` is equivalent to the comparison `L1 = R1 AND L2 = R2 ... AND Ln = Rn`.

  In the case of the comparison operators “<”, “<=”, “>=”, and “>”, however, there is no straightforward correspondence.

Comparing two values

If one or both of the operands are the NULL value, all comparison operators return the truth value unknown (see also section “NULL value”).

Alphanumeric values

Two alphanumeric values are compared from left to right character by character. If the two values have different lengths, the shorter string is padded on the right with blanks (X'40') so that both values have the same length.

Two strings are identical if each has the same character at the same position.

If two strings are not identical, the EBCDIC code of the first two differing characters determines which string is greater or smaller.

National values

Two national values are compared from left to right code unit by code unit. If the two values have different lengths, the shorter string is padded on the right with blanks (NX'0020') so that both values have the same length.

Two strings are identical if each has the same code unit at the same position.

If two strings are not identical, the binary value of the first two differing UTF-16 code units determines which string is greater or smaller.

Numeric values

Values of numeric data types are compared in accordance with their arithmetic value. Two numeric values are the same if they are both 0, or if they have the same sign and the same amount.

Time values

Dates, times and time stamps can be compared. The data type of both operands must be the same.

- One date is greater than another if it is a later date.
- One time is greater than another if it is a later point in time.
- One time stamp is greater than another if either the date is later or, if the date is the same, the time is later.

Examples

1. `1 <= 1` is always true.
2. Comparing alphanumeric values:
   Select the customers from the CUSTOMERS table that come from Munich, and include the customer information:

```
SELECT company, cust_info, city FROM customers WHERE city = 'Munich'
   company            cust_info           city
   Siemens AG         Electrical           Munich
   Login GmbH         PC networks          Munich
   Plenzer Trading    Fruit market         Munich
```

3. Comparing with a subquery that returns an atomic value:
   Select the items that need the greatest number of part 501 from the PURPOSE table:

```
SELECT item_num FROM purpose
   WHERE part = 501 AND number = (SELECT MAX(number)
   FROM purpose WHERE part = 501)
```

The subquery returns one row exactly, as the maximum is determined for a single group.

```
item_num
```

```
200
```

You can also write the example with the comparison of two rows each with two columns:

```
SELECT item_num FROM purpose
   WHERE (part, number) = (SELECT 501, MAX(number)
   FROM purpose WHERE part = 501)
```

4. In this example a cursor table is defined with ORDER BY.

   The WHERE clause selects those rows that come after the rows with *cust_num* `012` and *target* `DATE'<date>'` ' in the order stipulated by ORDER BY:

```
DECLARE cur_order CURSOR FOR
                    SELECT order_num, cust_num, atext, target FROM orders
                    WHERE (cust_num, target) > (012, DATE'<date>')
   ORDER BY cust_num, target
```

   You will only receive orders which are to be finished after the specified date from a customer with customer number 012, and all orders from customers with a greater customer number.

   The lexicographical comparison rules differ from the comparison rules for ORDER BY only in the case of NULL values.

5. Lexicographical comparison of rows

```
DECLARE rest_purpose CURSOR FOR
   SELECT item_num, part, SUM(number)
   FROM purpose
   WHERE (item_num, part) > (:last_item_num, :last_part)
   GROUP BY item_num, part
   HAVING SUM(number) > 0
   ORDER BY item_num, part
```

   This cursor reads how many exemplars of each part are contained in the various items. Items are read in ascending order by their item number; items with identical item numbers are read in ascending order by part number.

   The WHERE clause allows for reading the cursor table piecemeal (FETCH). For example, if you have read up to item 120 and up to part 230 and if you have opened the cursor again with the user variables `:last_item_num = 120` and `:last_part = 230`, the cursor table will only contain entries for item 120 and parts with numbers > 230 and entries for items with numbers > 120 (and any parts).

5.3.2 Quantified comparison (comparison with the rows of a table)

The value of a row is compared with the rows of a table. It is determined whether the comparison holds true either for all the rows of the table, or else for at least one row.

row comparison_op { `ALL` | `SOME` | `ANY` } *subquery_1*

row `::=` { `(`*expression* `,...)` | *expression* | *subquery_2* }

comparison_op `::=` { `=` | `<` | `>` | `<=` | `>=`| `<>` }

row

Left operand for the comparison.

Each *expression* in *row* must be an atomic value. The row consists of the *expression* values in the order specified. A single *expression* therefore returns a row with one column.

subquery_2

must return a table without multiple columns and with at most one row. This row is the left comparison operand. If the table returned is empty, the comparison operand is a row with the NULL value in each column.

comparison_op

Comparison operator.

=	Compare whether two values are the same
<	Compare whether one value is smaller than the other
>	Compare whether one value is greater than the other
<=	Compare whether one value is smaller than or equal to the other
>=	Compare whether one value is greater than or equal to the other
<>	Compare whether two values are not equal

subquery_1

The number of columns must equal the number of columns of *row*, corresponding columns of *row* and *subquery_1* must have compatible data types (see section “Compatibility between data types”).

Result

ALL

True if the right-hand operand is an empty table or if the results of the comparisons of the left-hand operand with each row of the right-hand operand are all true.

False if the result of the comparison of the left-hand operand with at least one row of the right-hand operand is false.

Unknown in all other cases.

SOME / ANY

True if the result of the comparison of the left-hand operand with at least one row of the right-hand operand is true.

False if the right-hand operand is an empty table or if the results of the comparisons of the left-hand operand with each row of the right-hand operand are all false.

Unknown in all other cases.

All comparisons are carried out according to the comparison rules in section “Comparison rules” on "Comparison rules".

Examples

This returns true if the current date is later than all the dates in the derived column and all of these dates are non-null. It returns false if the current date is earlier than at least one date or is the same as at least one date other than NULL in the derived column. In all other cases, the comparison returns unknown.

```
CURRENT_DATE > ALL (SELECT target FROM orders)
```

From the PURPOSE table, select the items that have a part the total number of which is greater than the total number of all the parts of the item with the item number 1.

```
SELECT item_num FROM purpose

WHERE number > ALL (SELECT number FROM purpose WHERE item_num = 1)
```

5.3.3 BETWEEN predicate (range query)

It is determined whether the row lies within a range specified its lower and upper limits.

{ *row_1* [NOT] BETWEEN *row_2* AND *row_3* |

vector_column [NOT] BETWEEN *expression* AND *expression* }

row ::= { (*expression* , ...) | *expression* | *subquery_2* }

vector_column ::= [*table*.] { *column[min..max]* | *column*(*min..max*) }

row

Each *expression* in *row* must be atomic. The row consists of the *expression* values in the order specified. A single *expression* therefore returns a row with one column.

subquery must return a table without multiple columns and with at most one row. This row is the operand. If the table returned is empty, the operand is a row with the NULL value in each column.

All three rows must have the same number of columns; corresponding columns must have compatible data types (see section "Compatibility between data types").

vector_column

A multiple column with special rules for the result. The column specification may not be an external reference.

expression

The values must be atomic and their data types must be compatible with the data type of the *vector_column* occurrences (section "Compatibility between data types").

Result

row_1 BETWEEN *row_2* AND *row_3* is identical to:
(*row_1* >= *row_2*) AND (*row_1* <= *row_3*)

row_1 NOT BETWEEN *row_2* AND *row_3* is identical to:
NOT (*row_1* BETWEEN *row_2* AND *row_3*)

vector_column [NOT] BETWEEN *expression* AND *expression*

- The range query is performed for each occurrence of *vector_column*.
- The individual results are combined with OR.

Example

If X is a multiple column with 3 elements, the range query `X[1..3] BETWEEN 13 AND 20` is equivalent to the following range queries:

```
X[1] BETWEEN 13 AND 20 OR X[2] BETWEEN 13 AND 20 OR X[3] BETWEEN 13 AND 20
```

Examples

BETWEEN predicate with numeric range:

Select all the items from the ITEMS table whose price is between 0 and 10 Euros, which include the item name in the output.

```
SELECT item_num, item_name, price FROM items
   WHERE price BETWEEN 0.00 AND 10.00
item_num   item_name       price
  210      Front hub       5.00
  220      Back hub        5.00
  230      Rim            10.00
  240      Spoke           1.00
  500      Screw M5        1.10
  501      Nut M5          0.75
```

BETWEEN predicate with range of dates:

Select all the orders placed in December 2013 from the ORDERS table, which include the order number, customer number, order date and order text in the output:

```
SELECT order_num, cust_num, order_text, order_date FROM orders
   WHERE order_date BETWEEN DATE'2013-12-01' AND DATE'2013-12-31'
       order_ cust_  order_text                   order_date
       num    num
       210    106    Customer administration      2013-12-13
       211    106    Database design customers    2013-12-30
```

BETWEEN predicate with a host variable:

MINIMUM is a host variable. The comparison returns true if the product of SERVICE_PRICE*SERVICE_TOTAL (price per service unit times number of service units) is outside the specified range. It returns false if the product is within the range. The comparison returns unknown if the value of SERVICE_PRICE or
SERVICE_TOTAL is unknown.

```
service_price*service_total NOT BETWEEN :MINIMUM AND 2000
```

5.3.4 CASTABLE predicate (convertibility check)

This checks whether an expression can be converted to a particular data type.

The CASTABLE predicate enables you to check whether a corresponding CAST expression (see section “CAST expression”) can be executed before it is executed and to react appropriately.

expression `IS [NOT] CASTABLE AS` *data_type*

expression

CAST operand. The value of *expression* may not be a multiple value with a dimension > 1.

data_type

Target data type for the result of the corresponding CAST expression.
data_type may not contain a dimension for a multiple column.

> **i** It must be possible to combine the data type of *expression* with *data_type*, see the table 23.

Result

Without NOT:

True if *expression* can be converted to the specified data type.
False if *expression* cannot be converted to the specified data type.

With NOT:

True if *expression* cannot be converted to the specified data type.
False if *expression* can be converted to the specified data type.

Example

Check whether an entry can be converted to a numeric data type with a particular length.

```
CASE WHEN :input IS CASTABLE AS NUMERIC(7,2)
     THEN CAST :input AS NUMERIC(7,2)
     ELSE -1
END
```

5.3.5 IN predicate (elementary query)

This determines whether a row occurs in a table.

```
{ row_1 [NOT] IN { subquery_2 | ( row_2 , ...) } |
  vector_column [NOT] IN ( expression , ...) }
row_1 ::= { ( expression ,...) | expression | subquery_1 }
row_2 ::= { ( expression ,...) | expression }
vector_column ::= [ table .] { column[min..max] | column ( min..max ) }
```

row_1

returns one row.

Each *expression* in *row_1* must be atomic. The row consists of the *expression* values in the order specified. A single *expression* therefore returns a row with one column.

subquery_1

must return a table without multiple columns and with at most one row. This row is the left-hand operand. If the table returned is empty, the operand is a row with the NULL value in each column.

subquery_2

this table is the right-hand operand.

row_2

The right-hand operand is the table whose individual row(s) are specified with *row_2*. If *row_2* is specified several times then the data type of each column of the table is determined by the rules described under “Data type of the derived column for UNION” .

row_1, *row_2*, *subquery_1* and *subquery_2* must all have the same number of columns; the data types of the corresponding columns must be compatible (see section “Compatibility between data types”).

vector_column

A multiple column with special rules for the result. The column specification may not be an external reference.

expression

The values must be atomic and their data types must be compatible with the data type of the *vector_column* occurrences (section “Compatibility between data types”).

Result

row_1 IN *subquery_2* or *row_1* IN (*row_2* ,...):

True if the comparison for equality of *row_1* with at least one row of the right-hand operand yields true.

False if all the comparisons for equality of *row_1* with some row of the right-hand operand yield false, or if the right-hand operand is a subquery which returns an empty table.

Unknown in all other cases.

row_1 NOT IN *subquery_2* or *row_1* NOT IN (*row_2* ,...):

is identical to:

NOT (*row_1* IN *subquery_2*) or. NOT (*row_1* IN **(** *row_2* ,...**)**)

The comparison rules for “=” apply (see also section “Comparison rules”).

vector_column [NOT] IN (*expression*, ,...)

The IN predicate is evaluated for each occurrence of *vector_column*.
The individual results are combined with OR.

Example

If X is a multiple column with 3 elements, the range query `X[1..3] BETWEEN 13 AND 30` is equivalent to the following element queries:

```
X[1] IN (13, 20, 30) OR X[2] IN (13, 20, 30) OR X[3] IN (13, 20, 30)
```

Examples

IN predicate with single rows as right-hand operand:

Select the customers from Munich or Berlin from the CUSTOMERS table.

```
SELECT company, cust_info, city FROM customer
   WHERE city IN ('Munich','Berlin')
   company          cust_info       city
   Siemens AG       Electrical      Munich
   Login GmbH       PC networks     Munich
   Plenzer Trading  Fruit market    Munich
   Freddys Fishery  Unit retail     Berlin
```

IN predicate with subquery as right-hand operand:

Select the orders for which no training was performed from the ORDERS and SERVICE tables.

```
SELECT cust_num FROM orders

WHERE order_num NOT IN (SELECT order_num FROM service WHERE service_text =
'Training')
```

5.3.6 LIKE predicate (simple pattern comparison)

A LIKE predicate determines whether an alphanumeric or a national value matches a specified pattern. A pattern is a string that, in addition to normal characters, can also include placeholders and escape characters.

A placeholder represents either one character or else any number of characters. A placeholder can also be used as a normal character in a pattern if its special meaning is canceled with the escape character. You can define the escape character with the ESCAPE clause.

operand [NOT] LIKE *pattern* [ESCAPE *character*]

operand ::= *expression*
pattern::= *expression*
character::= *expression*

operand

Alphanumeric or national expression representing the operand for the pattern comparison.

The value of *operand* must either be atomic or the name of a multiple column. If the operand is a multiple column, the entry for the column cannot be an external reference (i.e. the column of a superordinate query expression).

pattern

Alphanumeric or national expression to which the value from *operand* is to be matched. *pattern* can include the following:

- normal characters (i.e. all except placeholders and escape characters)
- Placeholder

Placeholder	**Meaning**
_ (underscore) %	one arbitrary character arbitrary (possibly empty) character string

- escape characters (each followed by a placeholder or another escape character)

Blanks in *pattern*, even at the beginning or end, are part of the pattern.

ESCAPE clause

You use the ESCAPE clause to define an escape character. If you place an escape character in front of a placeholder, the placeholder loses its function as a placeholder and is interpreted instead as a normal character. You can also use the escape character to cancel the special meaning of the escape character and use it as a normal character.

character

Alphanumeric or national expression whose value has a length of 1. In this comparison, *character* acts as an escape character.

ESCAPE omitted:

No escape character is defined.

> **i** The data types of *operand*, *pattern* and *character* must be comparable, i.e. they all have either one of the data types CHAR and VARCHAR or all have one of the data types NCHAR and NVARCHAR, see also the section “Compatibility between data types”.

Result

operand is an atomic value:

Unknown if the value of *operand, pattern* or *character* is the NULL value, otherwise

Without NOT:

True if the placeholders for characters and strings in *pattern* can be replaced by characters and strings, respectively, so that the result is equal to the value of
operand, and has the same length.

False in all other cases.

With NOT:

True if the placeholders for characters and strings in *pattern* cannot be replaced by characters and strings, respectively, so that the result is equal to the value of
operand, and has the same length.

False in all other cases.

operand is a multiple column:

The pattern comparison is performed for every occurrence in operand.
The individual results are combined with OR.

Examples

Select all the contact people from the CONTACTS table whose first name starts with Ro:

```
SELECT fname, lname FROM contacts WHERE fname LIKE 'Ro%'
   fname      lname
   Roland     Loetzerich
   Robert     Heinlein
```

The following statement selects all the rows from table TAB whose column COL starts with the underscore character and ends with at least one space:

```
SELECT * FROM tab WHERE col LIKE '@_% ' ESCAPE '@'
```

The following predicate returns true for all three-character values for TITLE whose first character is “M” and whose third character is “.”, i.e. for titles such as “Mr.” or “Ms.”. “_” is a placeholder which stands for any single character. Since the data type for the Title column is TITLE CHAR(20), the string must be padded with blanks to a length of exactly 20 characters.

```
title LIKE 'M_. ’
```

The escape character “!” cancels the placeholder “%” with the result that the comparison only returns true for 'Travel expenses%Discount'.

```
service_text LIKE 'Travel expenses!%Discount ' ESCAPE '!'
```

5.3.7 LIKE_REGEX predicate (pattern comparison with regular expressions)

A check is made to see whether an alphanumeric value matches a specified regular expression. Regular expressions are precisely defined search patterns which go far beyond the options of the search patterns in the LIKE predicate. Regular expressions are a powerful means of searching large data sets for complex search conditions. They have long been used, for example, in the Perl programming language.

---

operand `[NOT] LIKE_REGEX` *regular_expression* `[FLAG` *modifiers* `]`

operand `::=` *expression*
regular_expression `::=` *expression*
modifiers `::=` *expression*

---

operand

Alphanumeric expression which presents the operand for the comparison with the regular expression.
The value of *operand* may not be a multiple value with a dimension > 1.

regular_expression

Alphanumeric expression whose value is a regular expression which the value of *operand* should match. For information on the structure of regular expressions, see "LIKE_REGEX predicate (pattern comparison with regular expressions)".
You specify modifiers for *regular_expression* in the FLAG clause.
The value of *regular_expression* may not be a multiple value with a dimension > 1.

FLAG clause

Alphanumeric expression of the modifiers for *regular_expression*. You can specify the following modifiers:

flag	**Meaning**
i (caseless)	If this modifier is set, letters in the pattern match both upper and lower case letters.
m (multiline)	By default, SESAM/SQL treats the subject string as consisting of a single “line” of characters, even if it actually contains several NEWLINE characters (see "CSV() - Reading a BS2000 file as a table"). The “start of line” metacharacter (^) matches only at the start of the string, while the “end of line” metacharacter ($) matches only at the end of the string. When this modifier is set, the “start of line” and “end of line” constructs match immediately following or immediately before any newline in the subject string, respectively, as well as at the very start and end. If there are no NEWLINE characters in a subject string, or no occurrences of ^ or $ in a pattern, setting this modifier has no effect.
s (dotall)	If this modifier is set, a dot metacharacter in the pattern matches all characters including NEWLINE characters (see "CSV() - Reading a BS2000 file as a table"). Without it, newlines are

	excluded. A negative class such as [^a] always matches a newline character, independent of the setting of this modifier.
x (extended)	If this modifier is set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class; and characters between an unescaped # outside a character class and the next newline character, inclusive, are also ignored. This makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (?(which introduces a conditional subpattern.

flag must consist of lowercase letters. Each character can be specified multiple times. No blanks may be specified.

FLAG clause not specified:
No modifiers are defined for *regular_expression*.

Result

Unknown if the value of *operand, regular_expression* or *flag* is the NULL value, otherwise

Without NOT:

True if the placeholders for characters and strings in *regular_expression* can be replaced by characters and strings, respectively, so that the result is equal to the value of *operand*, and has the same length.

False in all other cases.

With NOT:
True if the placeholders for characters and strings in *regular_expression* cannot be replaced by characters and strings, respectively, so that the result is equal to the value of *operand*, and has the same length.

False in all other cases.

Examples

Select all the contact people from the CONTACTS table whose last name contains the string with meier “or something similar”:

```
SELECT fname, lname FROM contacts
   WHERE lname LIKE_REGEX '[a-z]* M [ae]? [iy] [a-z]* r' FLAG 'ix'
   fname      lname
   Albert     Gansmeier
   Berta      Hintermayr
   Thea       Mayerer
   Herbert    Meier
   Anton      Kusmir
```

In the CONTACTS table find the incorrect ZIP codes in the ZIP column:

```
SELECT * FROM contacts WHERE zip NOT LIKE_REGEX '\d{5}'
```

In the CONTACTS table find all the email contacts for Fujitsu:

```
SELECT address FROM contacts
   WHERE address LIKE_REGEX '([A-Za-z])+\.([A-Za-z]+)@fujitsu\.com'
   address
   Albert.Gansmeier@fujitsu.com
   Berta.Hintermayr@fujitsu.com
   Thea.Mayerer@fujitsu.com
```

Reguläre Ausdrücke in SESAM/SQL

The regular expressions in the LIKE_REGEX predicate correspond to the regular expressions in the Perl programming language with the following exceptions:

- They are not enclosed in delimiters
- There is no “replace” function
- The modifiers are specified in the FLAG clause

Special characters

Special characters in regular expressions have special functions:

Character	Meaning	Example
.	The period stands for any character other than a period.	`en.e` Hits e.g.: entire, entice, fence
+	The plus sign stands for single or multiple occurrence of the character preceding it.	`e+` Hits e.g. speaker, feeling, veeery good
*****	The asterisk stands for no, single or multiple occurrence of the character preceding it.	`se*` Hits e.g. storm, very good, feeling
?	The question mark stands for no or single occurrence of the character preceding it.	`se?` Hits e.g. storm, seldom but not: seesaw

^	The circumflex can negate a sign class or, in the case of strings, specify that the following search pattern must occur at the start of the search area.	`^Hans`
		
		Hits e.g. Hans Master, Hans Müller
		but not: Master Hans
		`^[^äöüÄÖÜ]*$`
		
		Hits e.g. Master
		but not: Müller

$		`Hans$`

<table>
<tr><td></td><td>In the case of strings the dollar sign specifies that the preceding search pattern must occur at the end of the search area.</td><td>Hits e.g. Master Hans
but not: Hans Master</td></tr>
<tr><td>|</td><td>The vertical slash separates alternative expressions.</td><td><code>[M|m]aster</code>
Hits e.g. Master, master
but not: Naster, aster</td></tr>
<tr><td>\</td><td>The backslash masks the subsequent (special) character.</td><td><code>clif\?</code>
Hit with clif?
but not: cliff</td></tr>
<tr><td>[ ]</td><td>Square brackets limit a character class.</td><td><code>Ma[lns]ter</code>
Hits e.g. Malter, Manter, Master
but not: Marter</td></tr>
<tr><td>-</td><td>The hyphen separates the limits of a character class.</td><td><code>Ma[a-z]ter</code>
Hits e.g. Malter, Manter, Master
but not: Mastner</td></tr>
<tr><td>( )</td><td>Parentheses group partial expressions.</td><td><code>(Mr.|Ms.) M[a-z]+</code>
Hit with Mr. Master, Ms. Müller
but not: Baroness Master</td></tr>
<tr><td>{ }</td><td>Braces are a repetition specification for preceding characters.</td><td><code>clif{2,5}</code>
Hit with cliff, cliffffhanger
but not: clif</td></tr>
</table>

Character repetitions

You check single character repetitions with the special characters `+`, `*` or `?`, see the table above.

You can also use braces to check multiple character repetitions: **{m,n}**. Here **m** specifies the minimum number and **n** the maximum number of repetitions.

The following specifications are permitted:

{m}	Repetition exactly m times
{m,}	Repetition at least m times
{m,n}	Repetition at least m times, but not more than n times

`f{1,3}` returns, for example, hits with life, cliff and cliffhanger.

Groupings

Groupings are formed using parentheses. The subsequent repetition character the refers to the entire expression enclosed in parentheses.

`h(el)+lo` returns, for example, hits with hello, helello, helelello.

Selection of characters

A list of characters if square brackets offers a selection of characters which the regular expression can match. The expression in square brackets stands only for one character from the list.

`Ma[lns]ter` returns, for example, hits with Malter, Manter and Master, but not with Maltner.

In order to specify a selection from a digit range or a section of the alphabet, use the hyphen "-".

`[A-Z][a-z]+[0-9]{2}` returns hits with words which begin with an uppercase letter followed by one or more lowercase letters and are concluded with precisely two digits, e.g. Masterson15, Smith01, but not masterson15, Smith1.

Alternatives

You can use the vertical slash "|" to specify multiple alternative strings in a regular expression which are to be searched for a string.

`([M|m]r|[M|m]s] M[a-z]*` returns hits with titles of persons whose names begin with M, e.g. Mr Master, Ms Miller.

Masking special characters

You must mask special characters when you do not intend the special meaning of the character, but mean its literal, normal meaning, in other words a vertical slash as a vertical slash or a period as a period. The mask character is in all cases the backslash "\".

`([A-Z]|[a-z])+\.([A-Z]|[a-z])+@fujitsu\.com` returns hits with all email addresses in the format: `first_name.last_name@fujitsu.com`.

`[A-Z]+\.[a-z]+@fujitsu\.com` returns the same result if you specify `'i'` in the flag clause, in other words wish to ignore uppercase/lowercase.

Operators

Letters which are preceded by a backslash "\" indicate special characters or particular character classes:

\n	One of the NEWLINE characters, see "CSV() - Reading a BS2000 file as a table"Tabulator character
\t	FORM FEED character
\f	CARRIAGE RETURN character
\r	Blanks, tabulator characters, NEWLINE characters, CARRIAGE RETURN
\s	characters, FORM FEED characters
\S	All characters except blanks, tabulator characters, NEWLINE characters, CARRIAGE RETURN characters, FORM FEED characters
\d	A digit
\D	Any character which is not a digit
\w	A logographic character, i.e. A through Z, a through z, and the underscore "_"

\W Any character which is not a logographic character
\A Start of a string
\Z End of a string
\b Word boundary, i.e. when \b... or ...\b is specified, a pattern returns a hit only if it is at the start or end of the word.

\B Negative word boundary, i.e. when \b... or ...\b is specified, a pattern returns a hit only if it is not at the start or end of the word.

For example, `\d{3,4}` returns hits with all 3- or 4- digit numbers and `\w{5}` returns hits with all 5-character words

Priority in regular expressions

The special characters in regular expressions are evaluated according to a particular priority.

1st priority: `( )` (bracketing)

2nd priority: `+ * ? {m,n}` (repeat operators)

3rd priority: `abc ^ $ \b \B` (characters/strings, start/end of line, start/end of word)

4th priority: `|` (alternatives)

This enables every regular expression to be evaluated unambiguously. However, if you want the evaluation to be different in the expression from the priority, you can insert parentheses in the expression to force a different evaluation.

For example `a|bc|d` returns hits with 'a' or 'bc' or 'd'.
`(a|b)(c|d)` returns hits with 'ac' or 'ad' or 'bc' or 'bd'.

Notes

- Leading or trailing blanks may need to be dealt with using `\s*` in the pattern. In particular when `$` (end of the search area) is specified) hits that would otherwise be possible are not detected.

  Example

  With the data type CHAR(n), for instance, the string Berta*bbbb* (*b* represents a blank) with the pattern `B.*ta$` is not recognized as blanks follow it.
- With the LIKE predicate a `Ber%` pattern means that a hit value also really begins with `Ber`, while the same pattern in the LIKE REGEX predicate may also begin at any position in the record. The `^Ber.*` pattern means that the pattern is contained at the start of the record.

5.3.8 NULL predicate (comparison with the NULL value)

A comparison is performed to check whether an expression contains the NULL value.

operand `IS [NOT] NULL`

operand `::=` *expression*

operand

Operand for the comparison. The value of *operand* must either be atomic or the name of a multiple column. If the operand is a multiple column, the entry for the column cannot be an external reference (i.e. the column of a superordinate query expression).

Result

operand is an atomic value:

Without NOT:

True if the value of *operand* is the NULL value.

False in all other cases.

With NOT:

True if the value in *operand* is not the NULL value.

False in all other cases.

operand is a multiple column:

Without NOT:

True if at least one occurrence the multiple column is the NULL value.

False in all other cases.

With NOT:

True if at least one occurrence of the multiple column is not the NULL value.

False in all other cases.

Examples

```
language1 IS NOT NULL
```

In the example, `LANGUAGE1` is a single column. If `LANGUAGE1` does not contain the null value, the comparison is true. The comparison `NOT language1 IS NULL` would also return the same truth value.

`LANGUAGE2(1..5)` is a multiple column containing the null value in some, but not all of the columns. The comparison `language2(1..5) IS NOT NULL` returns true in this case and `NOT (language(1..5) IS NULL)` returns the truth value false.

column IS NOT NULL and NOT (*column* IS NULL) are thus not equivalent if *column* is a multiple column. This becomes clear if `language2(1..5) IS NOT NULL` is represented as:

`language2(1) IS NOT NULL OR language2(2) IS NOT NULL OR ...`

`language2(5) IS NOT NULL`

The comparison returns true if at least one occurrence of `LANGUAGE2` is non-null.

`NOT (language(1..5) IS NULL)` on the other hand, can be represented as:

`NOT (language(1) IS NULL OR language(2) IS NULL ... OR language(5) IS`

`NULL)`

This comparison returns true if the comparisons with the null value in the parentheses following NOT return false. i.e. if all the occurrences of LANGUAGE2 are non-NULL.

Select the orders from the ORDERS table that have not yet been dealt with completely, i.e. for which the actual date is the NULL value.

```
SELECT order_num, order_text, target FROM orders WHERE actual IS NULL
   order_num     order_text                   target
   250           Mailmerge intro              <date>
   251           Customer administration      <date>
   300           Network test/comparison
   305           Staff training               <date>
```

5.3.9 EXISTS predicate (existence query)

An existence query checks whether a derived table is empty.

```
EXISTS subquery
```

subquery

Subquery that returns a derived table.

Result

True if the derived table is not empty.

False if the derived table is empty.

Example

Select the customers that have not placed an order from the CUSTOMERS table:

```
SELECT company FROM customers
   WHERE NOT EXISTS (SELECT order_num FROM orders
                    WHERE orders.cust_num = customers.cust_num)
   company
   Siemens AG
   Plenzer Trading
   Freddys Fishery
   Externa & Co Kg
```

5.4 Search conditions

Search conditions are used to restrict the number of rows affected by a table operation or SQL statement of a routine. Only the rows that satisfy the specified search condition are taken into account. You may specify search conditions for DELETE, MERGE, UPDATE and SELECT, when joining tables (join expression) and in a conditional expression (CASE expression). You can specify search conditions in table and column constraints in order to formulate integrity constraints. Search conditions also occur in the case of statements in routines.

You define a search condition in a WHERE, HAVING, ON, CHECK or WHEN clause or in a control statement of a routine, and it may be used in the following statements and expressions or query expressions:

- WHERE clause
  - DELETE statement
  - SELECT statement
  - SELECT expression for CREATE VIEW, DECLARE, INSERT
  - UPDATE statement
- HAVING clause
  - SELECT statement
  - SELECT expression for CREATE VIEW, DECLARE, INSERT
- ON clause
  - MERGE statement
  - Join expression
- CHECK condition in the CREATE TABLE or ALTER TABLE statement
- WHEN clause in a CASE-expression with search condition
- IF, CASE, REPEAT, or WHILE statement in a routine

A search condition consists of predicates and can include logical operators. The predicates are the operands of the logical operators.

A search condition is evaluated by applying the operators to the results of the operands. The result is one of the truth values true, false or unknown.

The operands are not evaluated in a predefined order. In certain cases, an operand is not calculated if it is not required for calculating the total result.

search_condition `::=` {

praedicate |

search_condition { `AND` | `OR` } *search_condition* /

`NOT` *search_condition* |

(*search_condition* **)**

}

predicate

Predicate

AND

Logical AND

Result for Op1 AND Op2

	Op1 true	**Op1 false**	**Op1 unknown**
Op2 true	true	false	unknown
Op2 false	false	false	false
Op2 unknown	unknown	false	unknown

Table 20: Logical operator AND

OR

Logical OR

Result Op1 OR Op2

	Op1 true	**Op1 false**	**Op1 unknown**
Op2 true	true	true	true
Op2 false	true	false	unknown
Op2 unknown	true	unknown	unknown

Table 21: Logical operator OR

NOT

Negation

Result for NOT Op

	NOT Op
Op true	false
Op false	true
Op unknown	unknown

Table 22: Logical operator NOT

Precedence

- Expressions enclosed in parentheses have highest precedence.
- NOT takes precedence over AND and OR.
- AND takes precedence over OR.

- Operators with the same precedence level are applied from left to right.

Examples

Select all orders with company placed after the specified date in the tables ORDERS and CUSTOMERS.

```
SELECT o.order_num, c.company, o.order_text, o.order_date
   FROM orders o, customers c
   WHERE o.order_date > DATE '<date>' AND o.cust_num = c.cust_num
   order_ company           order_text              order_date
   num
   250    The Poodle Parlor  Mailmerge intro          20010-03-03
   251    The Poodle Parlor  Customer administration 2010-05-02
   300    Login GmbH         Network test/comparison 2010-02-14
   305    The Poodle Parlor  Staff training          2010-05-02
```

Delete all the items from the ITEMS table whose price is less than 500.00 and whose item name starts with the letter H:

DELETE FROM items WHERE price < 500.00 AND item_name LIKE 'H%'

Select all the orders from the `SERVICE` table that were filled in the specified period or for which no training was given or no training documentation or manual created.

```
SELECT order_num, service_date, service_text FROM service
   WHERE service_date BETWEEN DATE '2013-04-01' AND DATE '2013-04-30'
   OR service_text NOT IN('Training','Training documentation','Manual')
   service_  order_   service_date   service_text
   num       num
    1        200      2013-04-19    Training documentation
    2        200      2013-04-22    Training
    3        200      2013-04-23    Training
    4        211      2013-01-20    Systems analysis
    5        211      2013-01-28    Database design
    6        211      2013-02-15    Copies/transparencies
   10        250      2013-02-21    Travel expenses
```

5.5 CASE expression

A CASE expression is a conditional expression, i.e. an expression that contains conditions. Each condition is assigned an expression or the NULL value.
When the CASE expression is evaluated, the assigned expression value or NULL value is returned to whichever condition is true.

There are different types of CASE expression:

- CASE expression with search condition
- Simple CASE expression
- CASE expression with NULLIF
- CASE expression with COALESCE
- CASE expression with MIN or MAX

The syntax of the various types of expression is shown in the following overview:

case_expression ::=

{

CASE

WHEN *search_condition* THEN

...

[ELSE { *expression* | NULL }]

END |

CASE *expressionx*
WHEN *expression1* [, *expression2*] ... THEN { *expression* | NULL }

...

[ELSE { *expression* | NULL }]

END |

NULLIF (*expression1* , *expression2*) |

COALESCE (*expression1* , *expression2*, ... *expressionn*) |

{ MIN | MAX }(*expression1*,*expression2*, ..., *expressionn*)

}

The types of CASE expression are described below.

The SQL statement CASE also exists in routines, see section “CASE - Execute SQL statements conditionally”.

5.5.1 CASE expression with search condition

A CASE expression with a search condition has the following syntax:

```
case_expression ::=

CASE

WHEN search_condition THEN { expression | NULL }

...

[ELSE { expression | NULL }]

END
```

search_condition

Search condition that returns a truth value when evaluated

expression

Expression that returns an alphanumeric, national, numeric or time value when evaluated. It cannot be a multiple value with a dimension greater than 1.

expression must be contained in the THEN clause, the ELSE clause or in both.

The data types of the values of *expression* in the THEN clauses and in the ELSE clause must be compatible (see section “Compatibility between data types”).

Result

The result of the CASE expression is contained in the THEN clause whose associated *search_condition* is the first to return the truth value. The THEN clause contains the value of the *expression* assigned to the THEN clause or the NULL value. The WHEN clauses are processed from left to right.

If no *search_condition* returns the truth value true, the result is the contents of the ELSE clause, i.e. the value of the *expression* assigned to the ELSE clause or the NULL value. If you do not specify the ELSE clause, the default applies (NULL).

The data type of a CASE expression with a search condition is derived from the data types of the values of *expression* contained in the THEN clauses and the ELSE clause, as follows:

- Each *expression* has the data type CHAR or NCHAR respectively:The value of the CASE expression is that with the data type CHAR or NCHAR respectively and the greatest length.
- At least one value of *expression* has the data type VARCHAR or NVARCHAR respectively:The value of the CASE expression is that with the data type VARCHAR or NVARCHAR respectively and the greatest or greatest maximum length.
- Each *expression* is of the type integer or fixed-point number (INT, SMALLINT, NUMERIC, DEC):The value of the CASE expression has the data type integer or fixed-point number.

- The number of decimal places is the greatest number of decimal places among the various values of *expression*.
- The total number of places is the greatest number of places before the decimal point plus the greatest number of decimal places among the different values of *expression*, but not more than 31.

- At least one value of *expression* is of the type floating-point number (REAL, DOUBLE PRECISION, FLOAT); the others have any other numeric data type:The value of the CASE expression has the data type DOUBLE PRECISION.
- Each *expression* has the time data type:All values must have the same time data type, and the value of the CASE expression also has this data type.

Example

Sort the items in the ITEMS table in accordance with the urgency with which they need to be ordered.

```
SELECT item_num, item_name,
   CASE
       WHEN stock > min_stock THEN 'O.K.'
       WHEN stock = min_stock THEN 'order soon'
       WHEN stock > min_stock * 0.5 THEN 'order now'
       ELSE 'order urgently'
   END
FROM items
```

5.5.2 Simple CASE expression

A simple CASE expression has the following syntax:

```
case_expression ::=

CASE expressionx

      WHEN expression1 [, expression2 ] ... THEN { expression | NULL }

      ...

      [ELSE { expression | NULL }]

END
```

expression

Expression that returns an alphanumeric, national, numeric or time value when evaluated.
It cannot be a multiple value with a dimension greater than 1.

The values of *expressionx* and *expression1... expressionn* must have compatible data types (see section “Compatibility between data types”).

expression must be contained in the THEN clause, the ELSE clause or both clauses.

The data types of the values of *expression* in the THEN clauses and in the ELSE clause must be compatible (see section “Compatibility between data types”).

Result

The value of *expressionx* after CASE is compared (from left to right) with the values of the expressions *expression1*, *expression2*, ... contained in the WHEN clause. The first time a match is found, the result of the CASE expression if the contents of the associated THEN clause, i.e. the value of the associated *expression* or the NULL value. If the CASE expression contains several WHEN clauses, the result is the contents of the first THEN clause in whose associated WHEN clause an expression was found found which was identical to *expressionx*. The WHEN clauses are processed from top to bottom.

If none of the expressions (*expression1... expressionn*) in the WHEN clauses are identical to *expressionx*, the result is the contents of the ELSE clause, i.e. the value of the expression assigned to the ELSE clause or the NULL value. If you do not specify the ELSE clause, the default applies (NULL).

The data type of a simple CASE expression is derived from the data types of the values of *expression* that are contained in the THEN clauses and the ELSE clause. The same rules apply that apply to the data type of a CASE expression with a search condition (see "CASE expression with search condition").

A simple CASE expression corresponds to a CASE expression with a search condition of the following form:

```
CASE
   WHEN expressionx=expression1 THEN {expression|NULL}
   WHEN expressionx=expression2 THEN {expression|NULL}
   ...
```

```
   WHEN expressionx=expressionn THEN {expression|NULL}
   ELSE {expression|NULL}
END
```

Examples

Sort the companies in the CUSTOMERS table in accordance with their location. Here the country codes should be replaced by the names of the countries.

```
SELECT company,
   CASE country
       WHEN '  D' THEN 'Germany'
       WHEN 'USA' THEN 'America'
       WHEN ' CH' THEN 'Switzerland'
   END
FROM customers
```

For payroll accounting, a distinction is to be made according to workday and weekend.

```
CASE EXTRACT(DAY_OF_WEEK FROM CURRENT_DATE)
     WHEN 1,2,3,4,5 THEN 'workday'
     WHEN 6.7 THEN 'weekend'
              ELSE '?????'
END
```

5.5.3 CASE expression with NULLIF

A CASE expression with NULLIF has the following syntax:

case_expression ::= NULLIF (*expression1* , *expression2*)

expression

Expression that returns an alphanumeric, national, numeric or time value when evaluated.
It cannot be a multiple value with a dimension greater than 1.

Result

The result of the CASE expression is NULL when *expression1* and *expression2* are identical. If they are different, the result is *expression1*.

A CASE expression with NULLIF corresponds to a CASE expression with a search condition of the following form:

```
CASE
   WHEN expression1=expression2 THEN NULL
   ELSE expression1
END
```

Example

Using the SERVICE table, determine the VAT calculated at rates other than 0.07.

```
SELECT service_price * NULLIF (vat,0.07) AS tax FROM service
```

5.5.4 CASE expression with COALESCE

A CASE expression with COALESCE has the following syntax:

case_expression `::= COALESCE (` *expression1* `,` *expression2* `, ...,` *expressionn* `)`

expression

Expression that returns an alphanumeric, national, numeric or time value when evaluated.
It cannot be a multiple value with a dimension greater than 1.

Result

The result of the CASE expression is NULL if all the expressions contained in the parentheses (*expression1... expressionn*) return NULL. If at least one *expression* returns a value other than the NULL value, the result of the CASE expression is the value of the first *expression* that does not return the NULL value.

The CASE expression `COALESCE (expression1,expression2)` corresponds to a CASE expression with a search condition of the following form:

```
CASE
   WHEN expression1 IS NOT NULL THEN expression1
   ELSE expression2
END
```

The CASE expression `COALESCE (expression1,expression2,...,expressionn)` corresponds to the following CASE expression with a search condition:

```
CASE
   WHEN expression1 IS NOT NULL THEN expression1
   ELSE COALESCE (expression2 ...,expressionn)
END
```

Examples

A list of contacts is to be created for specific customer contacts. In addition to the title, last name, telephone number and position, either the department or, if this is not known, the reason for the previous contact is to be determined.

```
SELECT title, lname, contact_tel, position,

COALESCE(department, contact_info) AS info FROM contacts WHERE contact_num < 30
```

Derived table

title	lname	contact_tel	position	info
Dr.	Kuehne	089/6361896	CEO	Personnel
Mr.	Walkers	089/63640182	Secretary	Sales
Mr.	Loetzerich	089/4488870	Manager	Networks
Mr.	Schmidt	0551/123873	Training	
Ms.	Kredler	089/923764	Organization	SQL course

After the title, last name, telephone number and function, the department of the customer is determined. If this information is missing (NULL), the column value for the CONTACT_INFO column is determined for INFO. If both the DEPARTMENT and CONTACT_INFO columns contain NULL, INFO will also contain NULL.

A list of order completion dates is to be generated from the ORDERS table. The list is to contain the date when the order was made, the order description and its completion date. If the actual completion date is not known, the target completion date is to be entered.

```
SELECT order_date, order_text,
   COALESCE (actual, target) AS completion_date FROM orders
   order_date     order_text                completion_date
   <date>         Staff training             <date>
   <date>         Customer administration    <date>
   <date>         Database design customers  <date>
   <date>         Mailmerge intro            <date>
   <date>         Customer administration    <date>
   <date>         Network test/ comparison
   <date>         Staff training             <date>
```

To determine the values for COMPLETION_DATE, the ACTUAL column is evaluated. If there is a date in the column, this is accepted. If ACTUAL contains the NULL value, the corresponding column value in the TARGET column is determined and entered in the COMPLETION_DATE column. If both ACTUAL and TARGET contain the NULL value, the NULL value is entered in the COMPLETION_DATE column.

5.5.5 CASE expression with MIN / MAX

A CASE expression with MIN / MAX has the following syntax:

case_expression `::= { MIN | MAX }(` *expression1,expression2,* `. . . ,` *expressionn* `)`

expression

Expression that returns an alphanumeric, national, numeric or time value when evaluated.
It cannot be a multiple value with a dimension greater than 1.

The values of *expression1,expression2,...,expressionn* must have compatible data types (see section "Compatibility between data types").

A CASE expression with MIN or MAX references different expressions. In this way it differs from the aggregate functions MIN() and MAX() (see "Aggregate functions") which reference the set of all values in a column in a table.

Result

The result of the CASE expression is NULL if at least one of the expressions contained in the parentheses (*expression1,expression2*,...,*expressionn*) returns NULL.

If no *expression* returns NULL, the result of the CASE expression is the value of the smallest *expression* when MIN is specified, the value of the largest *expression* when MAX is specified.

The CASE expression `MIN(expression1,expression2)` corresponds to a CASE expression with a search condition in the following form:

```
CASE
   WHEN expression1 <= expression2 THEN expression1
   ELSE expression2
END
```

The CASE expression `MIN(expression1,expression2,...,expressionn)` corresponds to the CASE expression

`MIN(MIN(expression1,expression2,...),expressionn).`

The CASE expression `MAX(expression1,expression2)` corresponds to a CASE expression with a search condition in the following form:

```
CASE
   WHEN expression1 >= expression2 THEN expression1
```

```
   ELSE expression2
END
```

The CASE expression `MAX(expression1,expression2,...,expressionn)` corresponds to the CASE expression

`MAX(MIN(expression1,expression2,...),expressionn).`

Example

The example below selects all entries in the `turnover` table since the date entered with the user variable `input_date`, but at most for the last 90 says.

```
SELECT * FROM turnover WHERE turnover.date >= MAX(:input_date,

DATE_OF_JULIAN_DAY(JULIAN_DAY_OF_DATE(CURRENT_DATE) - 90))
```

5.6 CAST expression

The CAST expression converts a value of a data type to a value of a different data type.

cast_expression ::= CAST ({ *expression* | NULL } AS *data_type*)

expression / NULL

CAST operand. It contains the keyword NULL or an expression *expression*. The value of *expression* may not be a multiple value with a dimension > 1.

data_type

Target data type for the result of the CAST expression.
The target data type *data_type* cannot contain a dimension for a multiple column.

Result

The result of the CAST expression is an atomic value of the target data type *data_type*. Which value is returned depends, on the one hand, on the value of the CAST operand and, on the other, on its data type.

If *expression* returns the NULL value or if the CAST operand contains the keyword NULL, the result of the CAST expression is the NULL value.

Apart from that, the rules for the conversion of a value to a different data type described as of "CAST expression" apply.

Combinations of initial and target data types

The data type of *expression*, referred to here as the initial data type, can only be combined with certain target data types. The table 23 shows which initial data types you can combine with which target data types, and which combinations are impermissible

		Target data type	**Target data type**	**Target data type**	**Target data type**	**Target data type**	**Target data type**	**Target data type**
		INTEGER SMALLINT DECIMAL NUMERIC	REAL DOUBLE PRECISION FLOAT	CHAR VARCHAR	NCHAR NVARCHAR	DATE	TIME (3)	TIMESTAMP (3)
Initial data type	INTEGER SMALLINT DECIMAL NUMERIC	yes	yes	yes	yes	no	no	no
		yes	yes	yes	yes	no	no	no

Initial data type	REAL DOUBLE PRECISION FLOAT							
Initial data type	CHAR VARCHAR	yes	yes	yes	no	yes	yes	yes
Initial data type	NCHAR NVARCHAR	yes	yes	no	yes	yes	yes	yes
Initial data type	DATE	no	no	yes	yes	yes	no	yes
Initial data type	TIME(3)	no	no	yes	yes	no	yes	yes
Initial data type	TIMESTAMP (3)	no	no	yes	yes	yes	yes	yes

Table 23: Permissible and impermissible combinations of initial and target data types for the CAST expression

Rules for converting a value to a different data type

In addition to the permitted combinations of initial and target data type (see table 23), the rules described below also apply to the conversion of a value to a different data type. The description is subdivided into three groups, depending on the target data type:

- The target data type is a data type for integers, fixed-point numbers or floating-point numbers
- The target data type is a data type for strings of fixed or variable length
- The target data type is a time data type.

The target data type is a data type for integers, fixed-point numbers or floating-point numbers

- Numeric values are rounded up or down when they have too many decimal places for the target data type. If the numeric value is too high for the target data type, you receive an error message.

  Examples

  ```
  CAST (4502.9267 AS DECIMAL(6,2))
  ```

  The value 4502.9267 is rounded down to 4502.93.

  ```
  CAST (-115.05 AS DECIMAL(2,0))
  ```

  The value -115.05 is rounded down to -115. However, since the value is too high for the target data type, an error message appears.

  ```
  CAST (2450.43 AS REAL)
  ```

  The value 2450.43 is represented as the floating-point number of the value 2.45043E3.

- It must be possible to represent alphanumeric and national values without any loss of value as a value of the assigned target data type. Leading or trailing blanks are removed.

  Examples

  `CAST ('512 ' AS SMALLINT) / CAST (N'512 ' AS SMALLINT)`

  The blank at the end of the string is removed. The string '512' is represented as the small integer 512.

  `CAST ('sum' AS NUMERIC)`

  This is an error: The string 'sum' cannot be represented as a numeric value, because numeric literals can only contain digits.

  `CAST ('255' AS REAL) / CAST (N'255' AS REAL)`

  The blanks at the end of the string are removed, and the string '255' is represented as the floating-point number 2.55000E2.

The target data type is a data type for strings of fixed or variable length

- It must be possible to represent numeric values of the data type integer, fixed-point number or floating-point number without any loss as a string of fixed or variable length. In addition, it must be possible to represent values of the data type floating-point number that are not equal to 0 in the standard form, and otherwise in the form $0E^{0}$. The following applies to all numeric values: if the length of the value is less than the fixed length of the target data type CHAR or NCHAR, blanks are added to the end of the value; if the length of the value is less than the maximum length of the target data type VARCHAR or NVARCHAR, it is retained. If the length of the value is greater than the fixed or maximum length of the target data type, you receive an error message.

  Examples

  `CAST (1234 AS CHAR(5)) / CAST (1234 AS NCHAR(5))`

  The value of the integer 1234 returns the alphanumeric string '1234 ' or the national string N'1234 ' respectively.'

  `CAST (25.95 AS VARCHAR(5)) / CAST (25.95 AS NVARCHAR(5))`

  The value of the fixed-point number 25.95 returns the alphanumeric string '25.95' or the national string N'25.95' respectively.

  `CAST (45.5E2 AS CHAR(7)) / CAST (45.5E2 AS NCHAR(7))`

  The value of the floating-point number 45.5E2 returns the alphanumeric string '4.55E3 ' or the national string N'4.55E3 ' respectively.

- Blanks are added to the end of alphanumeric and national values whose length is less than the fixed length of the target data type CHAR or NCHAR. If the length of the value is less than the maximum length of the target data type VARCHAR or NVARCHAR, it is retained. If the length of the value is greater than the fixed or maximum length of the target data type, the value is truncated to the length of the target data type. If characters other than blanks are removed, you receive a warning.

  Examples

  `CAST ( 'Weekend' AS VARCHAR(5)) / CAST (N'Weekend' AS NCHAR(5))`

  The string 'Weekend' is too long for the data type CHAR(5) or NCHAR(5) respectively. It is truncated to the length of the string 'Weeke', and SESAM/SQL issues a warning.

  `CAST ( 'Week' AS VARCHAR(15)) / CAST (N'Week' AS NVARCHAR(15))`

  The result is the alphanumeric string 'Week' or the national string N'Week' respectively. The string is not padded with blanks to the maximum length of 15 characters.

- It must be possible to represent time values as a string. If the length of the time value is less than the fixed length of the target data type CHAR or NCHAR, blanks are added at the end of the value. If the length of the time value is less than the maximum length of the target data type VARCHAR or NVARCHAR, it is retained. If it is greater than the fixed or variable length of the target data type, you receive an error message.

  Examples

  `CAST (DATE'2013-08-11' AS VARCHAR(20))`

  `CAST (DATE'2013-08-11' AS NVARCHAR(20))`

  The result is the alphanumeric string '2013-08-11' or the national string N'2013-0811' respectively.

  `CAST (DATE'2013-08-11' AS VARCHAR(5))`

  The time value is too long for a string with a maximum variable length of 5. The time value is not converted and an error message appears.

The target data type is a time data type.

- It must be possible to represent alphanumeric and national values without any loss of value as a value of the assigned target data type. Leading or trailing blanks are removed.

  Examples

  `CAST (' 2013-08-11' AS DATE)`

  `CAST (N' 2013-08-11' AS DATE)`

  The leading blank of the string is removed, and the string is converted to the data type DATE.

  `CAST ('2013-08-11 17:57:35:000' AS TIMESTAMP(3))`

  This is an error: The string cannot be represented as a time stamp. The separator between the components seconds and fractions of a second must be a period (.) in time stamp values.
- The following rules apply to the conversion of time values:
  - If the target data type is DATE and the initial data type TIMESTAMP, the result value contains the date (year-month-day) of the initial value.
  - If the target data type is DATE and the initial data type TIME, you receive an error message.
  - If the target data type is TIME and the initial data type TIMESTAMP, the result value contains the time (hour: minute:second) of the initial value.
  - If the target data type is TIME and the initial data type DATE, you receive an error message.
  - If the target data type is TIMESTAMP and the initial data type DATE, the result value contains the date entry (year-month-day) of the initial value and the fields hour:minute:second set to 0 for the time.
  - If the target data type is TIMESTAMP and the initial data type TIME, the result value contains the date (year-month-day) of the current date (CURRENT_DATE) and the time (hour:minute:second) of the initial value.

Examples

`CAST (TIMESTAMP '2013-08-11 17:57:35.000' AS DATE)`

The result value is the date '8/11/2013'.

```
SELECT order_text, CAST (actual AS TIMESTAMP(3))
   FROM orders WHERE cust_num=106
```

```
order_text                    actual
Customer administration       2010-04-17 00:00:00.000
Database design customers     2010-04-10 00:00:00.000
```

The derived table contains the column actual with the data type TIMESTAMP. The time stamp fields for the time are set to 0.

5.7 Integrity constraint

An integrity constraint is a rule governing the permitted contents of the rows in a table. A row can only be inserted into a table (INSERT, MERGE) or deleted from a table (DELETE) and a column value can only be updated (MERGE, UPDATE) if, afterwards, all integrity constraints are satisfied.

Integrity constraints cannot be defined for multiple columns.

Integrity constraints can be defined for individual columns or for a table. A column constraint is an integrity constraint on a single column. A table constraint is an integrity constraint which can refer to more than one column in the base table.

NOT NULL constraint

The NOT NULL constraint requires that a column contain no NULL values. The NOT NULL constraint can only be specified as a column constraint.

UNIQUE constraint

The UNIQUE constraint requires that the specified column or set of columns accept only unique values or sets of values.

PRIMARY KEY constraint

The PRIMARY KEY constraint defines a column or set of columns as the primary key of a table. The PRIMARY KEY constraint requires that the column or set of columns satisfy the UNIQUE and NOT NULL constraints. A table can have a maximum of one primary key.

Check constraint

A check constraint requires that every row in a table, the search condition entered accepts the truth value true or unknown, but not, however, the truth value false.

The search condition can only reference the table for which the check constraint was defined.

Referential constraint

A referential constraint ([FOREIGN KEY]..REFERENCES) defines a column or a combination of columns as a foreign key for a table. The columns for the foreign key are assigned to one or more columns in a single table or in two tables. These columns are called the referenced columns. The UNIQUE constraint must be valid for the referenced columns. The table containing the foreign key is called the referencing table. The table to which the referenced columns belong is called the referenced table. If no columns are specified for the referenced table, the primary key of the referenced table is used.

SESAM/SQL rejects a table operation after checking the referential constraint

- if, when a row is inserted or column values are updated in the referencing table, no appropriate values would exist in the referenced columns.
- if, when deleting or updating rows or columns in the referenced tables, foreign key values would remain in the referencing tables for which appropriate values in the referenced columns or the corresponding column would no longer exist.

In the case of single-column foreign keys, the referential constraint requires that every nonNULL value of the foreign key for a table match a value in the referenced column.

In the case of multiple-column foreign keys, each set of values that does not include a NULL value must occur in the referenced columns. This means that in SESAM/SQL, a row satisfies the referential constraint if a NULL value occurs in at least one column of a multiple-column foreign key.

5.7.1 Column constraints

When a base table is created or updated (CREATE TABLE, ALTER TABLE), column constraints can be specified in the column definitions for the individual columns. The column cannot be a multiple column.

A column constraint is an integrity constraint on a single column. All the values in the column must satisfy the integrity constraint.

For CALL DML tables, only the integrity constraint PRIMARY KEY can be defined.

```
col_constraint ::=

{
   NOT NULL |
   UNIQUE |
   PRIMARY KEY |
   REFERENCES  table [ ( column ) ] |
   CHECK ( search_condition )
}
```

NOT NULL

NOT NULL constraint.
The column cannot contain any NULL values.

The NOT NULL constraint is stored as a check constraint (*column* IS NOT NULL).

UNIQUE

UNIQUE constraint.
Non-null column values must be unique.

The column length must observe the restrictions that apply to an index (see CREATE INDEX statement, "CREATE INDEX - Create index").

PRIMARY KEY

PRIMARY KEY constraint.
The column is the primary key of the table. The values in the column must be unique. Only one primary key can be defined for each table.

The column cannot have the data type VARCHAR or NVARCHAR. In a CALL DML table, the column length must be between 4 and 256 characters. In an SQL table, there is no minimum column length.

The NOT NULL constraint applies implicitly to a primary key column.

REFERENCES

Referential constraint.
The column of the referencing table can only contain a non-NULL value if the same value is included in the referenced column of the referenced table.

The current authorization identifier must have the REFERENCES privilege for the referenced column.

table

Name of the referenced base table.The referenced base table must be an SQL table. The name of the referenced base table can be qualified by a database or schema name. The database name must be the same as the database name of the referencing table.

(*column*)

Name of the referenced column.
The referenced column must be defined with UNIQUE or PRIMARY KEY. The referenced column cannot be a multiple column. The referencing column and referenced column must have exactly the same data type.

(*column*) omitted:
The primary key of the referenced table is used as the referenced column. The referencing column and referenced column must have exactly the same data type.

CHECK (*search_condition*)

Check constraint.
Each value in the column must accept the truth value true or unknown, but not, however, the truth value false for the search condition *search_condition*.

The following restrictions apply to *search_condition:*

- *search_condition* cannot contain any host variables.
- *search_condition* cannot contain any aggregate functions.
- *search_condition* cannot contain any subqueries, i.e. it can only reference the column of the table to which the column constraint belongs.
- *search_condition* cannot contain a time function.
- *search_condition* cannot contain special variables.
- *search_condition* cannot contain any transliteration between EBCDIC and Unicode.
- *search_condition* cannot contain any conversion of uppercase letters to lowercase letters or of lowercase letters to uppercase letters if the string to be converted is a Unicode string.
- *search_condition* cannot be a multiple column.
- *search_condition* may not contain a User Defined Function (UDF).

Special considerations for CALL DML tables

The following restrictions must be taken into account for column constraints in CALL DML tables:

- A CALL DML table must contain exactly one primary key as a column or table constraint.
- Only PRIMARY KEY is permitted as a column constraint.

- The data type of the column with PRIMARY KEY must be CHAR with a length of at least 4 characters.

Column constraints and indexes

If you define a UNIQUE constraint, an index with the column specified for UNIQUE is used:

- If you have already defined an index with CREATE INDEX that contains this column, this index is also used for the UNIQUE constraint.
- Otherwise, the required index is generated implicitly. The name of the implicitly generated index starts with UI and is followed by a 16-digit number.
  The index is stored in the space for the base table. In the case of a partitioned table the index is stored in the space of the table's first partition.

Examples of column constraints

The example shows part of the CREATE TABLE statement used to create the SERVICE table in the ORDERCUST database. A check constraint is defined for the column service_total.

```
CREATE TABLE service (...,

service_total INTEGER CONSTRAINT service_total_pos CHECK (service_total >
0))
```

A Non-NULL constraint with an explicitly specified name is defined for the COMPANY column. CUST_NUM is defined as the primary key in the column constraint CUST_NUM_PRIMARY.

```
CREATE TABLE customers

(cust_num INTEGER CONSTRAINT cust_num_primary PRIMARY KEY,

company CHAR(40) CONSTRAINT company_notnull NOT NULL)
```

A referential constraint FOREIGN1 is defined for the ORDERS table. The foreign key ORDERS.CUST_NUM references the column CUSTOMERS.CUST.NUM.

```
ALTER TABLE orders

ADD CONSTRAINT foreign1 FOREIGN KEY(cust_num)

REFERENCES customers(cust_num)
```

5.7.2 Table constraints

When a base table is created or updated (CREATE TABLE, ALTER TABLE), table constraints can be specified. A table constraint is an integrity constraint which can refer to more than one column in the base table. None of the columns can be a multiple column.

For CALL DML tables, only the integrity constraint PRIMARY KEY can be defined.

---

table_constraint ::=

{

  UNIQUE (*column* ,...) |

  PRIMARY KEY (*column* ,...) |

  FOREIGN KEY (*column* ,...) REFERENCES *table* [(*column* ,...)] |

  CHECK (*search_condition*)

}

UNIQUE (*column*,...)

> UNIQUE constraint.
> The combination of values for the columns specified must be unique within the table in the case that none of the values is equal to the NULL value.
>
> The length of the columns must observe the restrictions that apply to an index (see the CREATE INDEX statement, "CREATE INDEX - Create index").
>
> A column cannot be specified more than once in the column list.
>
> The sequence of columns specified with the column list must differ from the sequence of columns specified with the column list of another UNIQUE constraint or of a PRIMARY KEY constraint for the same table.

PRIMARY KEY (*column*,...)

> PRIMARY KEY constraint.
> The specified columns together constitute the primary key of the table.
> The set of column values must be unique. Only one primary key can be defined for each table.
>
> None of the columns can be VARCHAR or NVARCHAR columns. The sum of the column lengths must not exceed 256 characters.
>
> A column cannot be specified more than once in the column list.
>
> The sequence of columns specified with the column list must differ from the sequence of columns specified with the column list of any UNIQUE constraint for the same table.
>
> The NOT NULL constraint applies implicitly to the primary key columns.

FOREIGN KEY ... REFERENCES

Referential constraint.The referencing columns can only contain a set of values that does not include any NULL values if the set of values also occurs in the referenced columns.You must specify the same number of columns in the referencing and referenced table. The data types of the corresponding columns must be exactly the same.

The current authorization identifier must have the REFERENCES privilege for the referenced column.

FOREIGN KEY (*column*,...)

Columns of the referencing table whose sets of values should be contained in the referenced base table.
A column cannot be specified more than once in the column list.

REFERENCES *table*

Name of the referenced base table.
The referenced base table must be an SQL table. The name of the referenced base table can be qualified by a database or schema name. The catalog name must be the same as the catalog name of the referencing table.

(*column*,...)

Names of the referenced columns.
A UNIQUE or primary key constraint that uses the same columns and the same order must be defined for these columns. None of the columns can be a multiple column.
A column cannot be specified more than once in the column list.

(*column*,...) omitted:
The primary key of the referenced table is used as the referenced column.

CHECK (*search_condition*)

Check constraint.
The search condition *search_condition* must return the truth value true or undefined (but not the truth value false) for each row in the table.
The following restrictions apply to *search_condition:*

- *search_condition* cannot contain any host variables.
- *search_condition* cannot contain any aggregate functions.
- *search_condition* cannot include any subqueries, i.e. *search_condition* can only reference columns of the table to which the column constraint belongs.
- *search_condition* cannot contain a time function.
- *search_condition* cannot contain special variables.
- *search_condition* cannot contain any transliteration between EBCDIC and Unicode.
- *search_condition* cannot contain any conversion of uppercase letters to lowercase letters or of lowercase letters to uppercase letters if the string to be converted is a Unicode string.
- *search_condition* may not contain a User Defined Function (UDF).

Special considerations for CALL DML tables

The following restrictions must be taken into account for table constraints in CALL DML tables:

- A CALL DML table must contain exactly one primary key as a column or table constraint.

- Only PRIMARY KEY is permitted as the table constraint.
- The data type of the column with PRIMARY KEY must be CHAR, NUMERIC, INTEGER or SMALLINT. In the case of NUMERIC, decimal places are not permitted.
- The sum of the column lengths must be between 4 and 256 characters.
- The table constraint defines a compound primary key. The name corresponds to the verbal attribute name of the compound primary key in SESAM/SQL V1.x.

Table constraints and indexes

If you define a UNIQUE constraint, an index with the columns specified for UNIQUE is used:

- If you have already defined an index with CREATE INDEX that contains these columns, this index is also used for the UNIQUE constraint.
- Otherwise, the required index is generated implicitly. The name of the implicitly generated index starts with UI and is followed by a 16-digit number.
  The index is stored in the space for the base table. In the case of a partitioned table the index is stored in the space of the table's first partition.

Example of a table constraint

The example shows part of the CREATE TABLE statement used to create the CUSTOMERS table of the ORDERCUST database.

```
CREATE TABLE customers
...
CONSTRAINT PlausZip
   CHECK ((country = 'D' AND zip >= 00000) OR (country <> 'D'))
...
```

5.8 Column definitions

When a base table is created or modified (CREATE TABLE, ALTER TABLE), the column definition defines the name and the attributes of a column.

SESAM/SQL distinguishes between atomic and multiple columns. In an atomic column, exactly one value can be stored in each row. In a multiple column, several values of the same type can be stored in each row. A multiple column is made up of a number of column elements. In the case of a single column, a single value is stored for each row.

To incorporate BLOBs in base tables, you will need REF columns. These are defined using the FOR REF clause.

A base table can contain a maximum of 26134 columns of any data type except VARCHAR and NVARCHAR. It can contain up to 1000 VARCHAR and/or NVARCHAR columns. The restrictions that apply to CALL DML tables are described on "Column definitions".

---

column_definition ::= *column* { *data_type* [*default*] | FOR REF(*table*) }

[[CONSTRAINT *integrity_constraint_name*] *col_constraint*] ...

[*call_dml_clause*]

default ::= DEFAULT

{

alphanumeric_literal |
national_literal |
numeric_literal |
time_literal |
CURRENT_TIME(3) |
LOCALTIME(3) |
CURRENT_TIMESTAMP(3) |
LOCALTIMESTAMP(3) |
USER |
CURRENT_USER |
SYSTEM_USER |
NULL |
REF(*tabelle*)

}

call_dml_clause ::= CALL DML *call_dml_default* [*call_dml_symb_name*]

---

column

Name of the column. The column name must be unique within the base table.

data_type

Data type of the column.

FOR REF(*table*)

Defines a column containing references to BLOB values. This clause allows you to incorporate BLOBs in "normal" base tables. BLOB values are stored in BLOB tables. Information on defining a BLOB table can be found in the section "CREATE TABLE - Create base table". BLOB objects, tables and REF values are explained briefly in the section "Concept of the SESAM CLI". Detailed information on their structure can be found in the " Core manual".

- The column is assigned the data type CHAR(237).
- Its default value is the class REF value. The structure of REF values is described below.
- *table* must not contain the database name (*catalog*).

REF(*table*)

Class REF value which identifies the overall class of the BLOB values of a BLOB table. When a REF column is created, it is assigned this value as the default. This is determined by specifying the name of the BLOB table. Due to the syntax of the column definition, therefore, it is neither practical nor possible to specify a default value for the REF column at this point.
A REF value essentially has the following structure:

ss/*tt*?UID=*uuuu*&OID=*nn*

- *ss* is the unqualified name of the BLOB table's schema, excluding the database name.
- *tt* is the unqualified name of the BLOB table, excluding the schema and database name.
- *uuu* is the unique BLOB ID consisting of 32 hexadecimal digits. In the case of the class REF value, all the digits are 0.
- *nn* is the number of the BLOB in the BLOB table. In the case of the class REF value, this number is 0.

default

Defines an SQL default value that is entered in the column if a row is inserted or updated and no value or the default value is specified for the column.

- *column* cannot be a multiple column.
- *column* cannot be a CALL DML column.
- *default* must conform to the assignment rules for default values (see section "Default values for table columns").

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

default omitted:
There is no SQL default value.
The NULL value is entered in columns without a NOT NULL constraint.

[CONSTRAINT *integrity_constraint_name*] *column_constraint*

Defines an integrity constraint for the column. Integrity constraints cannot be specified for multiple columns.

[CONSTRAINT *integrity_constraint_name*] *column_constraint* omitted:
No column constraint defined.

CONSTRAINT *integrity_constraint_name*

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

CONSTRAINT *integrity_constraint_name* omitted:
The integrity constraint is assigned a name according to the following pattern:

UN *integrity_constraint_number*
PK *integrity_constraint_number*
FK *integrity_constraint_number*
CH *integrity_constraint_number*
where UN stands for UNIQUE, PK for PRIMARY KEY, FK for FOREIGN KEY and CH for CHECK.
integrity_constraint_number is a 16-digit number. The NOT NULL constraint is stored as a check constraint.

column_constraint

Indicates an integrity constraint that the column must satisfy.

call_dml_clause

The CALL DML clause ensures compatibility with SESAM/SQL V1.x. The CALL DML clause can only be specified for CALL DML tables, but not for columns used for the primary key. In this case, SESAM/SQL assigns both the *call_dml_default* and the *call_dml_symb_name*

call_dml_clause omitted:
The column definition is valid for either an SQL table or for the primary key of a CALL DML table. In the case of an SQL table, the CREATE TABLE or ALTER TABLE statement in which the column definition occurs cannot include a CALL DML clause.

call_dml_default

Indicates the non-significant value of a column as an alphanumeric literal.

call_dml_default corresponds to the non-significant value in SESAM/SQL Version 1.x.

call_dml_symb_name

Symbolic name of the column.

call_dml_symb_name corresponds to the symbolic attribute name in
SESAM/SQL Version 1.x.

call_dml_symb_name omitted:
call_dml_symb_name is assigned by the system.

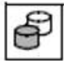
Special considerations for CALL DML tables

The following restrictions must be observed when creating column definitions for CALL DML tables:

- Only the data types CHAR, NUMERIC, DECIMAL, INTEGER and SMALLINT are permitted.
- No default value can be defined for the column with DEFAULT. The default value FOR REF is not permitted either.
- The table must contain exactly one primary key restraint as the column or table constraint.
- The table constraint defines a compound primary key and must be given a name that corresponds to the name of the compound primary key in SESAM/SQL V1.x.
- The column name must be different to the integrity constraint name of the table constraint since this name is used as the name of the compound primary key.
- A column that is not a primary key must have a CALL DML clause.

Examples of column definitions

This example shows part of the CREATE TABLE statement used to create the ORDERS table of the ORDERCUST database.



```
CREATE TABLE orders
(order_num                       INTEGER,
cust_num                         INTEGER NOT NULL,
contact_num                      INTEGER,
order_date                       DATE DEFAULT CURRENT_DATE,
order_text                       CHARACTER (30),
actual                           DATE,
target                           DATE,
order_stat                       INTEGER DEFAULT 1 NOT NULL,
...)
```

This example shows the CREATE TABLE statement used to create the ITEM_CAT table of the ORDERCUST database. This table contains two REF columns.

```
CREATE TABLE item_cat
(item_num                        INTEGER NOT NULL,
image                            FOR REF(addons.images),
desc                             FOR REF(addons.descriptions))
```

6 Query expression

In SESAM/SQL, query expressions are the most important means of querying data.

This chapter describes the syntax of query expressions and provides you with an explanation of the various joins. It is subdivided into the following sections:

- Table specifications
- SELECT expression
- Table queries
- Joins
- Subquery
- Combining query expressions with UNION
- Combining query expressions with EXCEPT DISTINCT
- Updatability of query expressions

Overview

You use query expressions to select rows and columns from base tables and views. The rows found constitute the derived table.

A query expression is part of an SQL statement. A query expression can occur in subqueries or in any of the following SQL statements:

CREATE VIEW	Define a view
DECLARE CURSOR	Declare a cursor
INSERT	Insert rows in table

The examples in this chapter only show the relevant query expression. Without the associated subquery or SQL statement, the examples are of course not executable.

If you want to use a subquery in an SQL statement, you must own the table referenced in the subquery or have SELECT privilege for the table involved.

---

query_expression ::= [*query_expression* { UNION [ALL | DISTINCT] | EXCEPT [DISTINCT] }]
{ *select_expression* | TABLE *table* | *join_expression* | (*query_expression*) }

---

select_expression

SELECT expression (see section “SELECT expression”)

TABLE *table*

Table query, see section “TABLE - Table query”.

join_expression

Join expression (see section “Join expression”)

(*query_expression*)

Subquery, see section “Subquery”.

UNION

Combine two query expressions with UNION, see section “Combining query expressions with UNION”.

EXCEPT DISTINCT

Combine two query expressions with EXCEPT, see section “Combining query expressions with EXCEPT”.

6.1 Table specifications

```
table_specification ::= { table [[AS] correlation_name [( column , ...)]] |
                          subquery [AS] correlation_name [( column , ...)] |
                            TABLE([ catalog .] table_function ) [WITH ORDINALITY]
                                   [[AS] correlation_name [( column , ...)]] |
                          join_expression }
```

table

Name of a base table or view.

The same table can occur several times in a table specification in the query expression. Correlation names are used to distinguish between different instances of the same table.

subquery

The table is the derived table that results from evaluating *subquery*.

[*catalog*.] *table_function*

The ("read-only") table (see the " Core manual") is the result of the table function *table_function*.

If table function DEE() is specified, no column names may be specified.

The database name *catalog* must be specified if the containing statement is not to be executed on the database set implicitly (see "Qualified names") (and consequently possibly with another SQL server).

WITH ORDINALITY

Definition of a counting column in the derived table. This specification may only be entered for the table function CSV(), but not for DEE().

The derived table must "at the end" contain one column more than the column specification in each line of the CSV file. The data type of the last column of the derived table must be DECIMAL(31,0). This column is used as the counting column. Beginning with 1 and in ascending order, it is assigned the ordinal number of the line which was read in from the CSV file. The WHERE clause also enables derived rows of particular ordinal numbers to be ignored in a SELECT expression, see the example on the next page.

The data types of each column of the derived table (with the exception of the last column) must match the data types of the column specifications in the CSV file.

WITH ORDINALITY not specified:
The number of columns in the derived table must be the same as the number of column specifications in the CSV file, and the data types of each column must match.

Example

`with.3.headers` is a CSV file with exactly 3 headers which are not evaluated or are skipped:

```
SELECT c1, c2,...,cn
       FROM TABLE(CSV('with.3.headers' DELIMITER ',' QUOTE '?'
                  ESCAPE '-', CHAR(20), CHAR(20),..., CHAR(20)))
       WITH ORDINALITY
       AS T(c1, c2,....,cn, counter)
WHERE counter > 3
```

correlation_name

Table name used in the query expression as a new name for the table.

The *correlation_name* must be used to qualify the column name in every column specification that references this instance of the table if the column name is not unambiguous.

The new name must be unique, i.e. *correlation_name* can only occur once in a table specification of this query expression.

You must give a table a new name if the columns in the table cannot otherwise be identified uniquely in the query expression.
correlation_name must be specified in the case of *table_function* (exception: DEE()).

In addition, you may give a table a new name in order to formulate the query expression so that it is more easily understood or to abbreviate long names.

Example

Join a table with itself:

`SELECT a.company, b.company`	`-- Query customer`
`FROM customers AS a,` `customers AS b`	
`WHERE a.city = b.city`	`-- who lives in the same city`
`AND a.cust_num < b.cust_num`	`-- but avoid duplicates`

column

Column name that is used within the query expression as the new name for the column of the corresponding table.

If you rename a column, you must give all the columns in the table a new name.

column is the new name of the column and must be unique within the table specified by *correlation name.* In this query expression the column may only be addressed with the new name.

The columns of a derived table must be renamed if the column names of the table upon which it is based are not unique, or if the derived columns are to be referenced using names that have been assigned internally.

Example

Give the columns in the WAREHOUSE table new, more informative names:

```
SELECT * FROM warehouse w (item_number, current_stock, location)

WHERE location = 'Parts warehouse'
```

column,... omitted:
The column names of the associated table are valid. These could be names that are assigned internally, which cannot be referenced in the query expression.

join_expression

Join expression that determines the tables from which the data is to be selected. Join expressions are described in the section “Join expression”.

Underlying base tables

Depending on the specification made in the table specification, the underlying base tables are defined as follows:

Specification in table specification	Underlying base table
Base table	the base table
View	all the base tables which the view references directly or indirectly
Subquery	Base table upon which the subquery is based
`TABLE([`*catalog*`.]`*table_function*`)`	no base table

Table 24: Underlying base tables

6.2 SELECT expression

select_expression ::=

SELECT [ALL | DISTINCT] *select_list*
FROM *table_specification*,...
[WHERE *search_condition*]
[GROUP BY *column*,...]
[HAVING *search_condition*]

select_list ::= { * | { *table*.* | *expression* [[AS] *column*] } }

The following applies to all clauses:

- The clauses must be specified in the given order.
- Column names must be unique. If a column name occurs in several tables, you must qualify the column name with the table name. If you rename a table using a correlation name for the duration of the SELECT statement (see section "Table specifications"), you must use only the correlation name.

  Example

  ```
  SELECT o.cust_num, s.service_price
  FROM orders o, service s WHERE o.order_num=s.order_num
  ```

Evaluation of SELECT expressions

SELECT expressions are evaluated in the following order:

1. The Cartesian product from all the table specifications in the FROM clause is created.
2. If a WHERE clause is specified, the WHERE search condition is applied to all the rows of the Cartesian product. The rows for which the search condition returns the value true are selected.
3. If a GROUP BY clause is specified, the rows determined in point 2 are combined into groups.
4. If a HAVING clause is specified, the HAVING search condition is applied to all the groups. The groups that satisfy the search condition are selected.
5. If the SELECT list includes an aggregate function and the derived table has not yet been divided into groups, all the rows in the derived table are combined to form a group.
6. If the derived table has been divided into (one or more) groups, the SELECT list is evaluated for each group.

   If the derived table has not been divided into groups, the SELECT list is evaluated for each derived row.

   The resulting rows then form the derived table of the SELECT expression.

6.2.1 SELECT list - Select derived columns

You determine the columns in the derived table with the SELECT list.

SELECT [ALL | DISTINCT] *select_list* ...

select_list ::= { * | { *table* .* | *expression* [[AS] *column*] } } ...

ALL

Duplicate rows in the derived table are retained.

DISTINCT

Duplicate rows are removed.

* Select all columns. The order and the names of the columns in the table specified in the FROM clause are used. If several tables are involved, the order of the tables in the FROM clause is used. At least one column must exist.

table.*

All the columns in *table* are selected. *table* must be included in the FROM clause. The order and the names of the columns in *table* are used. *table* may not be the correlation name for a DEE() table function.

expression

Expression denoting a derived column. If *expression* contains a column specification, the table to which the column belongs must be included in the FROM clause of this SELECT expression.

The names of the columns in the SELECT list must be unique. If you join tables and these base tables have columns with identical names, you must insert the appropriate table or correlation name in front of the column names in order to ensure unique identification.

If SELECT DISTINCT is specified, *expression* cannot consist of a multiple column specification.

If an aggregate function (AVG, COUNT, MAX, MIN, SUM) occurs in a column selection, the following restrictions apply:

- Only column names that are specified in the GROUP BY clause or which are arguments in the aggregate function can be included in the SELECT list.
- Only one aggregate function can be used with DISTINCT on the **same** level of a SELECT query. For example, you must not enter:

  SELECT COUNT(DISTINCT ...) ...SUM(DISTINCT ...) ...

[AS] *column*

Name of the derived column specified with *expression.*

Example

```
SELECT order_num AS order_no, COUNT(*) AS total FROM orders GROUP BY order_num


          order_no          total

          ...               ...
```

column omitted:

If *expression* is a column name, the derived column is assigned this name, otherwise, the column name is not defined.

Example

```
SELECT order_num, COUNT(*) FROM orders GROUP BY order_num
          order_num          ...

          ...                ...
```

Columns in the derived table

The order of the columns in the derived table corresponds to the order of the columns in the SELECT list.

The attributes of a derived column (data type, length, precision, digits to the right of the decimal point) are either taken from the underlying column or result from the specified expression.

A result column can return the NULL value if one of the following conditions is satisfied:

- One of the columns used can return the NULL value.
  This is always the case for columns of table functions. This is only the case for columns of base tables if a NOT NULL condition applies for the column.
- The expression that describes the result column contains at least one of the following operands or elements:
  - an indicator variable
  - a subquery
  - the aggregate function AVG, MAX, MIN or SUM
  - a CAST expression of the form CAST (NULL AS *data_type*)
  - a CASE containing the NULL value in at least one THEN or ELSE clause
  - a CASE expression with NULLIF
  - a CASE expression with COALESCE, where at least one operand of COALESCE (*expression1 ... expressionn*) contains one of the operands or elements listed above

Examples

“*” selects all the columns of the tables specified in the FROM clause. The sequence of the columns in the derived table is determined by the sequence of the tables in the from clause and by the defined sequence of columns within the tables.

```
SELECT * FROM orders, customers
```

CUSTMERS.* selects all columns from the CUSTOMERS table. DISTINCT specifies that duplicate rows are not to be included in the derived table.

```
SELECT DISTINCT order_num, customers.* FROM orders, customers
```

This selects the order numbers from the SERVICE and ORDERS tables. The column names must be unique. If tables with identical column names are linked, the column names must be qualified by the table name or correlation name. If you specify ALL (default), duplicate rows are included in the derived table.

```
SELECT ALL S.order_num, O.order_num FROM service S, orders O
```

This selects the name of the service and the price per service unit including VAT. If *expression* without the [AS] *column* specification is a column name, the column in the derived table is assigned this name (SERVICE_TEXT in the example).
[AS] *column* can be used to assign a name for the derived column, which is then referenced by *expression* (in the example this is GROSS_PRICE). The properties of a column in the derived table (data type, length, precision and scale) are either taken from the underlying column (SERVICE_TEXT) or are derived from the specified *expression* `(service_price*(1.0+vat))`.

```
SELECT service_text, service_price*(1.0+vat) AS gross_price
```

The derived table contains a single row. There is one column only in this row, which contains the sum of all the non-NULL values in SERVICE.SERVICE_PRICE, or NULL if there is no row matching this criterion. If the SELECT list includes an aggregate function, the list may only contain column names which occur within the argument of an aggregate function.

```
SELECT SUM(service_price) FROM service
```

The derived table contains a row with a single column containing the number of rows in CONTACTS. If *expression* without the AS clause does not identify a column, the column name is not defined.

```
SELECT COUNT(*) FROM contacts
```

6.2.2 SELECT...FROM - Specify table

You use the FROM clause to specify the tables from which data is to be selected.

In order to read rows in the specified tables, you must either own these tables or have SELECT permission.

```
SELECT ...

FROM table_specification, ...
```

table_specification

Specification of a table from which data is to be read. You can only specify tables located in the same database.

Examples

The columns CUST_NUM from the CUSTOMERS table and ORDER_NUM from the ORDERS table are selected on the basis of the Cartesian product of the CUSTOMERS and ORDERS tables. The CUSTOMERS and ORDERS tables are renamed within the SELECT expression by assigning correlation names. Every column specification within the SELECT expression which references the CUSTOMERS or ORDERS table must then be qualified with the correlation name. Correlation names can be used to qualify columns uniquely, to abbreviate long table names or to specify the appropriate table name in SELECT expressions. The columns A.CUST_NUM and B.ORDER_NUM are selected from the Cartesian product of the CUSTOMERS and ORDERS tables.

```
SELECT A.cust_num, B.order_num FROM customers A, orders B
```

Derived table

cust_num	order_num
100	200
100	210
100	211
etc.	etc.
107	300
107	305

The table ORDSTAT is renamed as ORDERSTATUS and the columns ORD_STAT_NUM and ORD_STAT_TEXT are selected using the new names ORDERSTATUSNUMBER and ORDERSTATUSTEXT. If all columns are selected by specifying “*” in the SELECT list, it is possible to assign new column names using “(*column*, ...)” in *table_specification*. Unlike the AS clause in the SELECT list, it is not possible to rename individual columns. It is only possible to rename all columns. The new names must be used in place of the old names in the WHERE, GROUP BY and HAVING clauses in the SELECT list.

```
SELECT * FROM ordstat
```

```
AS orderstatus (orderstatusnumber, orderstatustext)
```

If a table is specified more than once in the FROM clause, as is the case when a table is joined to itself, correlation names must be defined to allow unique identification of columns. References in the SELECT list and in the WHERE, GROUP BY and HAVING clauses must use these correlation names instead of the original table names.

```
SELECT A.cust_num, B.cust_num FROM customers A, customers B
```

6.2.3 SELECT...WHERE - Select derived columns

You use the WHERE clause to specify a search condition for selecting the rows for the derived table. The derived table contains only the rows that satisfy the search condition (i.e. the search condition is true). Rows for which the search condition returns the value false or unknown are not included in the derived table.

```
SELECT ...
WHERE search_condition
```

search_condition

Condition that the selected rows must satisfy.

Examples

The predicates are described in detail in chapter "Compound language constructs". Here, the most important types of search condition are illustrated using simple examples.

Comparison with constants: =, <, <=, >, > =, <>

```
SELECT cust_num, company FROM customers WHERE zip = 81739
```

Comparison with string pattern: [NOT] LIKE

```
SELECT * FROM customers WHERE company LIKE 'Sie%'
```

Range query: [NOT] BETWEEN

```
SELECT cust_num, company FROM customers WHERE zip BETWEEN 80000 AND 89999
```

Comparison with NULL value: IS [NOT] NULL

```
SELECT service_num, order_num, service_text FROM service WHERE inv_num IS NULL
```

Comparison with several values: [NOT] IN

```
SELECT cust_num, company FROM customers WHERE zip IN (81739, 80469)
```

Inner SELECT statement: [NOT] EXISTS

```
SELECT company FROM customers
WHERE NOT EXISTS (SELECT * FROM orders WHERE customers.cust_num = orders.cust_num)
```

Subquery (see section "Subquery"):

Subquery that returns a derived column: ALL, ANY, SOME, [NOT] IN

```
SELECT company FROM customers WHERE customers.cust_num =
SOME (SELECT cust_num FROM orders WHERE order_date = DATE '<date>')
```

Correlated subquery:

Select for each order, the service that is at least double the average service price for this order:

```
SELECT s1.service_num, s1.order_num, s1.service_text FROM service s1
   WHERE s1.service_total * s1.service_price > 2 *
   (SELECT AVG (s2.service_total*s2.service_price)
    FROM service s2 WHERE s2.order_num = s1.order_num)
```

Condition: AND, OR, NOT

```
SELECT service_num, order_num, service_date, service_text FROM service

WHERE service_text = 'Training' AND service_date > = DATE '<date>'
```

6.2.4 SELECT...GROUP BY - Group derived rows

You use the GROUP BY clause to combine table rows into groups. Two rows belong to the same group if, for each grouping column, the values in both rows are the same with regard to the comparison rules (see section “Comparison of two rows”), or both values are the NULL value.

The derived table contains a row for each group.

```
SELECT ...

GROUP BY column,...
```

column

Grouping column. *column* must be part of a table that was specified in the FROM clause. Ambiguous column names must be qualified with the table name. If you declared a correlation name for the table involved in the FROM clause, you must use this name to qualify the column names.

Multiple columns cannot be used as the grouping column.

Effect of the GROUP BY clause

If you specify the GROUP BY clause, only columns listed in GROUP BY or which are arguments in an aggregate function can be included in the SELECT list.

Aggregate functions for columns of a grouped table are evaluated for each group.

How are groups created?

- A group is a set of rows that all have the same values in each specified grouping column according to the comparison rules.
- Rows that have the NULL value in the same column and the same values in the other columns also constitute a group.

Examples

List the average amount of VAT for each order number:

```
SELECT order_num, AVG(vat) FROM service GROUP BY order_num
   order_num
   200          0.14
   211          0.06
   250          0.07
```

The number of contacts is determined for all customers outside the USA and grouped by customer number. If the GROUP BY clause is specified, only those columns may occur in the select list which are specified in the GROUP BY clause or which are arguments of an aggregate function. The derived table for the SELECT expression contains one row for each group.

```
SELECT contacts.cust_num, COUNT(*) AS total FROM contacts, customers
   WHERE contacts.cust_num = customers.cust_num AND customers.country <>'USA'
   GROUP BY contacts.cust_num
```

Derived table

cust_num	number
100	2
101	1
102	1
103	1
104	1
105	1

When the SELECT expression is supplemented by the HAVING clause below (see the next section), the derived table only contains the first row.

```
HAVING COUNT(*) > 1
```

6.2.5 SELECT...HAVING - Select groups

You use the HAVING clause to specify search conditions for selecting groups. If a group satisfies the specified search condition, the row for that group is included in the derived table. If no GROUP BY clause is specified, all the rows are considered one group.

```
SELECT ...
HAVING search_condition
```

search_condition

Search condition to be satisfied by a group.

Unlike a WHERE search condition, which is evaluated for each row in a table, the HAVING search condition is evaluated once for each group.

A column name in *search_condition* must satisfy one of the following conditions:

- The column is included in the GROUP BY clause.
- The column name is an argument of an aggregate function (AVG(), SUM(), ...). If the column name also appears in the SELECT list, it may also only appear there as the as an argument of an aggregate function.
- The column occurs in a subquery. If the column name references the table in the FROM clause, it must be included in the GROUP BY clause or be the argument in an aggregate function.
- The column is part of a table from a higher-level SELECT expression.

Example

Display the latest service provided for each order, but only if it was provided after the specified date:

```
SELECT order_num, MAX(service_date) FROM service GROUP BY order_num
HAVING MAX(service_date) > DATE'<date>'
```

6.3 TABLE - Table query

You use a table query to select all the columns of a table.

In order to read rows in the specified tables, you must either own these tables or have SELECT permission.

```
TABLE table
```

table

Name of the table (base table or view) all of whose columns are selected. The sequence, names and attributes (data type, length, precision, decimal places) of the columns of *table* are accepted.

The query expression TABLE *table* corresponds to the SELECT expression (SELECT * FROM *table*) (see section “SELECT expression”).

Example

Display all columns in the SERVICE table:

TABLE service

6.4 Joins

A join links the data from two or more tables. A table can also be joined to itself.

Which records of the tables involved are included in the derived table depends on the join type and any join conditions that exist.

There are two ways of creating a join:

- with a join expression
- without a join expression: in a SELECT expression or SELECT statement using the FROM clause and, if necessary, the WHERE clause.

6.4.1 Join expression

A join expression consists of the tables to be joined, the desired join operation and possibly a join condition.

A join expression can be specified

- as a query expression in an SQL statement
- in the FROM clause of a SELECT expression or SELECT statement
- in a subquery in the SELECT list and HAVING clause

The derived table of a join expression cannot be updated.

```
join_expression ::={ table_specification CROSS JOIN table_specification |
                     table_specification [ INNER | { LEFT | RIGHT | FULL } [OUTER] ]JOIN
                                   table_specification ON search_condition |
                     table_specification UNION JOIN table_specification |
                      ( join_expression ) }
```

table_specification

Specification of a table from which data is to be read (see section "Table specifications").

CROSS

CROSS operator for forming a cross join. A cross join corresponds to the Cartesian product of the tables involved (see section "Cross joins").

INNER

INNER operator for creating an inner join. In an inner join, the derived table only contains the rows that satisfy the join condition (see section "Inner joins").

LEFT, RIGHT, FULL

Operators for creating an outer join. A table that is part of an outer join cannot include multiple columns.

In an outer join, the type of outer join defines the dominant table(s) (see section "Outer joins").

If a row in the dominant table does not satisfy the join condition, the row is nevertheless included in the derived table. The derived column that references the other table is set to NULL values.

LEFT	The table to the left of the LEFT operator is the dominant table.
RIGHT	The table to the right of the RIGHT operator is the dominant table.
FULL	

The table to the left and the right of the FULL operator are both dominant tables. FULL joins the tables created with LEFT and RIGHT.

search_condition

Search condition to be used as the join condition for joining the specified tables.

The following applies to any column specified in *search_condition*:

The column must either be part of one of the tables to be joined or, in the case of subqueries, part of one of the tables from a higher-level SELECT expression.

If an aggregate function occurs in *search_condition*, one of the following conditions must be satisfied:

- The aggregate function is part of a subquery.
- The join expression is in a SELECT list or HAVING clause, and the column specified in the argument of the aggregate function is an external reference.

UNION

UNION operator for forming a union join. A table that is part of a union join cannot contain any multiple columns.

The derived table of a union join contains both the records of the table to the left of the UNION operator and the records of the table to the right of the UNION operator, including in each case the columns of the other table set to NULL values (see section “Union joins”).

join_expression

Nested join expression for creating a join from more than two tables.

6.4.2 Joins without join expression

In SESAM/SQL, an inner join or a cross join can also be created without a join expression. The tables to be joined are listed in the FROM clause of a SELECT expression, and the join search condition is formulated in the corresponding WHERE clause.

`SELECT ... FROM` *table_specification* `,` *table_specification* `[,...] WHERE` *search_condition_with_join_column*

Example

Select customer names and the associated order numbers from the CUSTOMERS and ORDERS tables:

```
SELECT company, order_num FROM customers, orders

WHERE customers.cust_num= orders.cust_num
```

6.4.3 Join types

SESAM/SQL supports cross joins, inner joins, outer joins and union joins. These are explained below and illustrated using examples.

6.4.3.1 Cross joins

The derived table of a cross join is the Cartesian product of the tables involved. Each record in the table to the left of the CROSS operator is linked to each record in the table to the right of the CROSS operator.

Example

Form the Cartesian product of the CUSTOMERS and ORDERS tables:

SELECT *	or	SELECT *
FROM customers, orders		FROM customers CROSS JOIN orders

Table CUSTOMERS

cust_ num	company	...
100	Siemens AG	
101	Login GmbH	
102	JIKO GmbH	
...	...	
106	Foreign Ltd.	
107	Externa & Co KG	

Table ORDER

order_ num	cust_ num	...
200	102	
210	106	
211	106	
...	...	
300	101	
305	105	

Derived table

	company	...			...

cust_ num			order_ num	cust_ num	
100	Siemens AG		200	102	
101	Login GmbH		200	102	
102	JIKO GmbH		200	102	
...	...		...	...	
106	Foreign Ltd.		200	102	
107	Externa & Co KG		200	102	
100	Siemens AG		210	106	
101	Login GmbH		210	106	
102	JIKO GmbH		210	106	
...	...		...	...	
106	Foreign Ltd.		210	106	
107	Externa & Co KG		210	106	
...	...		...	...	
100	Siemens AG		305	105	
101	Login GmbH		305	105	
102	JIKO GmbH		305	105	
...	...		...	...	
106	Foreign Ltd.		305	105	
107	Externa & Co KG		305	105	

Cartesian product of the CUSTOMERS and ORDERS tables

6.4.3.2 Inner joins

In an inner join, the derived tables contain only rows that satisfy the join condition.

Simple inner joins

A simple inner join selects rows from the Cartesian product of two tables.

Example

Select customer names and the associated order numbers from the CUSTOMERS and ORDERS tables:

SELECT company, order_num	or	SELECT company, order_num
FROM customers, orders		FROM customers JOIN orders
WHERE customers.cust_num = orders. cust_num		ON customers.cust_num = orders. cust_num

Customers who have not placed an order, e.g. Freddy's Fishery with the customer number 104, are not included in the derived table.

```
company             order_num
Login GmbH          300
JIKO GmbH           200
The Poodle Parlor   250
The Poodle Parlor   251
The Poodle Parlor   305
Foreign Ltd.        210
Foreign Ltd.        211
```

Example

Select the service associated with each order.

```
SELECT o.order_num, o.order_text, o.order_stat, s.service_num, s.service_text
FROM orders o INNER JOIN service s ON o.order_num = s.order_num

order_ order_                    order_ service_ service_
num    text                      stat   num      text

200    Staff training             5      1       Training
                                                 documentation
200    Staff training             5      2       Training
200    Staff training             5      3       Training
211    Database design customers  4      4       Systems analysis
211    Database design customers  4      5       Database design
```

```
211    Database design customers  4      6       Copies/transparencies
211    Database design customers  4      7       Manual
250    Mailmerge intro            2     10       Travel expenses
250    Mailmerge intro            2     11       Training
```

Multiple inner joins

A multiple inner join selects columns from the Cartesian product of more than two tables.

Example

Select the service provided for each customer who has placed an order from the CUSTOMERS, ORDERS and SERVICE:

SELECT c.company, o.order_num, s.service_num	or	SELECT c.company, o.order_num, s.service_num
FROM customers c, orders o, service s		FROM customers c JOIN orders o
WHERE c.cust_num=o.cust_num		ON c.cust_num=o.cust_num
AND o.order_num=s.order_num		JOIN service s ON o.order_num=s.order_num

```
company           order_num  service_num
JIKO GmbH         200          1
JIKO GmbH         200          2
JIKO GmbH         200          3
Foreign Ltd.      211          4
Foreign Ltd.      211          5
Foreign Ltd.      211          6
Foreign Ltd.      211          7
The Poodle Parlor 250         10
The Poodle Parlor 250         11
```

6.4.3.3 Outer joins

Another type of join is the outer join. It is created by using the keyword LEFT, RIGHT or FULL in the join expression. Unlike an inner join, the following applies to an outer join:

There are one (LEFT, RIGHT) or two (FULL) **dominant** tables. If a row in a dominant table does not satisfy the join condition, the row is nevertheless included in the derived table. The derived column that references the other table is set to NULL values.

Example

As in the first join example, select customer names and the associated order numbers from the CUSTOMERS and ORDERS tables. In this case, however, list all customers, even those who have not yet placed an order. To do this, you create the following outer join:

```
SELECT company, order_num FROM customers

LEFT OUTER JOIN orders ON customers.cust_num=orders.cust_num
```

Customers who have not placed an order, like Freddy's Fishery with the customer number 104, are now included in the derived table. The NULL value is entered for the missing order number.

```
company             order_num

Siemens AG
Login GmbH          300
JIKO GmbH           200
Plenzer Trading
Freddy's Fishery
The Poodle Parlor   250
The Poodle Parlor   251
The Poodle Parlor   305
Foreign Ltd.        210
Foreign Ltd.        211
Externa & Co KG
```

6.4.3.4 Union joins

Another type of join is the union join. The derived table of a union join is formed as follows:

- The table to the left of the UNION operator is extended on the right by having the columns of the other table added to it. The added columns are set to the NULL value.
- The table to the right of the UNION operator is extended on the left by having the columns of the other table added to it. The added columns are set to the NULL value.
- The derived table represents the set union of the two extended tables.

Example

Link the ITEMS and PURPOSE tables by means of a union join.

```
SELECT items.item_num, items.item_name, purpose. *
   FROM items UNION JOIN purpose

   item_num     item_name    item_num part          nmuber

   1            Bicycle
   2            Bicycle
   10           Frame
   11           Frame
   120          Front wheel
   130          Back wheel
   200          Handlebars
   ...
   501          Nut M5
                               1     10              1
                               1    120              1
                               1    130              1
                               1    200              1
                             120    210              1
                             ...
                             200    501              10
```

6.4.3.5 Compound joins

If you join more than two tables, you can nest several join expressions.

This allows you to combine inner and outer joins in a single SQL statement.

Examples

The following examples select the customer number, order number and service number from the CUSTOMERS, ORDERS and SERVICE tables. The results depend on the joins used.

- Take into account only those customers for whom orders with associated services exist.

```
SELECT c.cust_num, o.order_num, s.service_num
   FROM (customers c INNER JOIN orders o ON c.cust_num = o.cust_num)
                     INNER JOIN service s ON o.order_num = s.order_num
   WHERE c.cust_num BETWEEN 100 AND 107
   cust_num      order_num      service_num
   102           200             1
   102           200             2
   102           200             3
   105           250            10
   105           250            11
   106           211             4
   106           211             5
   106           211             6
   106           211             7
```

- Take into account all the customers from the CUSTOMERS table for whom orders exist,regardless of whether these orders have services associated with them. The join expression enclosed in parentheses is the dominant table for the outer join. The NULL value is entered for missing service numbers.

```
SELECT c.cust_num, o.order_num, s.service_num
   FROM (customers c INNER JOIN orders o ON c.cust_num = o.cust_num)
                     LEFT OUTER JOIN service s ON o.order_num = s.order_num
   WHERE c.cust_num BETWEEN 100 AND 107
   cust_num      order_num    service_num
   101           300
   102           200             1
   102           200             2
   102           200             3
   105           250            10
   105           250            11
   105           251
   105           305
   106           210
   106           211             4
   106           211             5
   106           211             6
   106           211             7
```

- Take into account all the customers in the CUSTOMERS table, regardless of whetherthey have placed orders or not. Orders are included in the derived table even if they are not yet associated with a service.

```
SELECT c.cust_num, o.order_num, s.service_num
   FROM (customers c LEFT OUTER JOIN orders o ON c.cust_num = o.cust_num)
                     LEFT OUTER JOIN service s ON o.order_num = s.order_num
   WHERE c.cust_num BETWEEN 100 AND 107
```

CUSTOMERS is the dominant table in the outer join that is enclosed in parentheses. The expression in parentheses is the dominant table of the outermost outer join. The NULL value is entered for missing item and service numbers.

```
cust_num     order_num    service_num
100
101          300
102          200            1
102          200            2
102          200            3
103
104
105          250           10
105          250           11
105          251
105          305
106          211            4
106          211            5
106          211            6
106          211            7
106          210
107
```

The following examples refer to the CUSTOMERS and ORDERS tables. In order to better illustrate the possibilities of an outer join, orders without customers are also permitted. This means that the foreign key definition for the ORDERS table is ignored here. We shall assume that an order with the number 400 is in the ORDERS table and is not yet associated with a customer.

- Select customer names and the associated order numbers from the CUSTOMERS and ORDERS tables and include customers who have not currently placed an order.

```
SELECT customers.company, orders.order_num FROM customers
        LEFT OUTER JOIN orders ON customers.cust_num=orders.cust_num
```

Customers who have not placed an order, like Freddy's Fishery with the customer number 104, are included in the derived table. The NULL value is entered for the missing order number.

```
company                   order_num
Siemens AG
Login GmbH                300
JIKO Gmbh                 200
Plenzer Trading
Freddy's Fishery
The Poodle Parlor         250
The Poodle Parlor         251
The Poodle Parlor         305
Foreign Ltd.              210
```

```
Foreign Ltd.              211
Externa & Co KG
```

- Select customer names and order numbers from the CUSTOMERS and ORDERS tables and include orders that are not associated with a customer.

```
SELECT customers.company, orders.order_num FROM customers
       RIGHT OUTER JOIN orders ON customers.cust_num=orders.cust_num
```

The order number 400 is also included in the derived table. The NULL value is entered for the missing customer name.

```
company                order_num
 JIKO Gmbh               200
 Foreign Ltd.            210
 Foreign Ltd.            211
 The Poodle Parlor       250
 The Poodle Parlor       251
 Login GmbH              300
 The Poodle Parlor       305
                         400
```

- Select customer names and the associated order numbers from the CUSTOMERS and ORDERS tables while taking customers without orders and orders without customers into account.

```
SELECT customers.company, orders.order_num FROM customers
       FULL OUTER JOIN orders ON customers.cust_num=orders.cust_num
```

A fictitious order with the order number 400, which is not yet associated with a customer, is also included in the derived tables, as is the customer Freddy’s Fishery who has not currently placed an order. NULL values are entered in place of the missing column values.

```
company                   order_num
Siemens AG
Login GmbH                300
JIKO Gmbh                 200
Plenzer Trading
Freddy’s Fishery
The Poodle Parlor         250
The Poodle Parlor         251
The Poodle Parlor         305
Foreign Ltd.              210
Foreign Ltd.              211
Externa & Co KG
                          400
```

6.5 Subquery

A subquery is a query expression that can be used in

- As an expression:
  The subquery must return a single-column derived table with a maximum of one row. The value of the subquery is then the value in the derived table or the NULL value if the derived table is empty.
- predicates:
  In the predicates ANY, SOME, ALL, IN and EXISTS the subquery returns a derived table.
- In the FROM clause of SELECT expressions:
  The subquery returns a derived table.
- In join expressions:
  The subquery returns a derived table.

A subquery is always enclosed in parentheses.

---

subquery ::= (*query_expression*)

---

query_expression

Query expression that returns the derived table.

In subqueries that are not specified in the predicate EXISTS or in a FROM clause, the derived table can only contain an atomic column or multiple columns with the dimension 1.

6.5.1 Correlated subqueries

In a nested query expression, an inner subquery is called a **correlated subquery** if it references columns of an outer table, i.e. a table that is used in one of the outer query expressions.

You can use correlated subqueries to determine the relationships between the values in a column.

Example

In a personnel table with a column for the age of each person, you can determine which people are exactly the average age (see example below).

Uncorrelated subqueries only need be evaluated once. Correlated subqueries must be evaluated several times for the various rows of the outer table. If the subquery is nested, the innermost subquery is evaluated first, etc.

Examples

The following query is a correlated subquery:

```
SELECT DISTINCT order_text FROM orders WHERE EXISTS

(SELECT * FROM service WHERE service.order_num = orders.order_num)
```

The inner subquery in the WHERE clause references the column ORDER_NUM in the ORDERS table of the outer query. ORDERS.ORDER_NUM is also known as an outer reference, since the column references a table in the outer query. The query is evaluated by determining the value of ORDERS.ORDER_NUM in the first row of the ORDERS table, evaluating the subquery on the basis of this value and using this result in the outer query. This is then repeated for the second value of ORDERS.ORDER_NUM and so on. The query returns a derived table:

order_text
Staff training
Database draft customers
Instruction concerning mail merge

For each order in the SERVICE table, you want to select the services whose price is above the average service price for this order:

```
SELECT s1.service_num, s1.order_num, s1.service_total*s1.service_price
   FROM service s1
   WHERE s1.service_total*s1.service_price >
   (SELECT AVG (s2.service_total*s2.service_price) FROM service s2 WHERE
   s1.order_num=s2.order_num)
```

Query expressions can be nested to any depth:

```
SELECT company, cust_num FROM customers WHERE cust_num IN
   (SELECT cust_num FROM orders WHERE order_num IN
       (SELECT order_num FROM service WHERE (service_price*service_total) IN
           (SELECT MAX(service_price*service_total) FROM service)))
```

Since these are not correlated subqueries, each subquery is evaluated once and the result is then used in the outer query.

Derived table

company	cust_num
Foreign Ltd.	106

6.6 Combining query expressions with UNION

select_expression ::= { *query_expression* | TABLE *table* | *join_expression* | *(query_expression)* }

[UNION [ALL | DISTINCT] *query_expression*]

select_expression

SELECT expression (see section “SELECT expression”)

TABLE *table*

Table query, see section “TABLE - Table query”.

join_expression

Join expression (see section “Join expression”)

(*query_expression*)

Subquery, see section “Subquery”.

UNION

The UNION clause combines two query expressions. The derived table contains all the rows that occur in the first or second derived table. You can combine more than two derived tables if you use the UNION clause several times.

If you want to combine query expressions with UNION, the following conditions must be met:

- The derived tables of both UNION operands must have the same number of columns and the data types of corresponding columns must be compatible (see section “Compatibility between data types”). The data type of a derived column is determined by applying the rules described in the “Data type of the derived column for UNION”.
- If the corresponding columns in both source tables have the same names, the derived column is given this name. Otherwise, the name of the derived column is undefined.
- Only atomic columns may be selected.

Query expressions combined with the UNION clause cannot be updated.

ALL

Duplicate rows in the derived table are retained.

DISTINCT

Duplicate rows are removed. If you do not specify ALL or DISTINCT, the default value is DISTINCT.

> **i** In contrast to the SELECT expression, the default value for UNION is DISTINCT. As it can be complicated to remove duplicate rows, the setting ALL is recommended for UNION if the application can dispense with removing duplicate rows.

Data type of the derived column for UNION

If two query expressions are combined with UNION, the data type of the derived column is determined by applying the following rules:

- Both source columns are of the type NCHAR:
  The derived column is of the type NCHAR with the longer of the two lengths.
- One source column is of the type VARCHAR and the other source column is of the type CHAR or VARCHAR:
  The derived column is of the type NVARCHAR with the greater length or greater maximum length.
- Both source columns are of the type NCHAR:
  The derived column is of the type NCHAR with the longer of the two lengths.
- One source column is of the type NVARCHAR and the other source column is of the type NCHAR or NVARCHAR:
  The derived column is of the type NVARCHAR with the greater length or greater maximum length.
- Both source columns are an integer or fixed-point type (INT, SMALLINT, NUMERIC, DEC):
  The derived column is of type integer or fixed-point.
  - The number of digits to the right of the decimal point is the greater of the two values of the source columns.
  - The total number of significant digits is the greater of the two values plus the greater of the two values for the number of digits after the decimal point of the source column with a maximum number of 31 digits.
- One source column is of a floating-point type (REAL, DOUBLE, FLOAT), the other is of any numeric data type:
  The derived column is of the type DOUBLE PRECISION.
- Both source columns have a date and time data type:
  Both columns must have the same date and time data type and the derived column also has this data type.

Examples

Determine all order numbers whose associated order value is at least 10,000 euros or whose target date is before the specified date.

```
SELECT order_num FROM service GROUP BY order_num
   HAVING SUM(service_total * service_price * (1 + vat)) > = 10000.00
   UNION DISTINCT
   SELECT order_num FROM orders WHERE target <= DATE '<date>'
```

The names of those companies are to be determined for which order documentation has already been archived or services have already been provided prior to the specified date:

```
SELECT c.company FROM customers c, orders o WHERE c.cust_num = o.cust_num
   AND o.order_num IN
   (SELECT o.order_num FROM orders o WHERE o.order_status > 4
                 UNION
                 SELECT DISTINCT s.order_num FROM service s
                        WHERE s.service_date < DATE'<date>')
```

The UNION expression in the subquery produces a derived table containing the order numbers 200 and 211.

The derived table is thus:

company
JIKO GmbH
Foreign Ltd

6.7 Combining query expressions with EXCEPT

query_expression `::=` { *select_expression* | `TABLE` *table* | *join_expression* | (*query_expression*) }

[`EXCEPT` [DISTINCT]] *query_expression*]

select_expression

SELECT expression (see section “SELECT expression”)

TABLE *table*

Table query, see section “TABLE - Table query”.

join_expression

Join expression (see section “Join expression”)

(*query_expression*)

Subquery, see section “Subquery”.

EXCEPT

The EXCEPT operation is similar to the difference between two sets in set theory. The derived table contains all rows from the first table which do not exist in the second table.

If you want to combine query expressions with EXCEPT, the following conditions must be met:

- The derived tables of both EXCEPT operands must have the same number of columns.
- The data types of the corresponding columns must be compatible (see section “Compatibility between data types”).
  The data type of a derived column is determined by applying the rules described in the “Data type of the derived column for EXCEPT”.

DISTINCT

Duplicate rows are removed from the derived table. DISTINCT is the default value.

Data type of the derived column for EXCEPT

If two query expressions are combined with EXCEPT, the data type of the derived column is determined by applying the following rules (as with UNION).

- Both source columns are of the type NCHAR:
  The derived column is of the type NCHAR with the longer of the two lengths.

- One source column is of the type VARCHAR and the other source column is of the type CHAR or VARCHAR:
  The derived column is of the type NVARCHAR with the greater length or greater maximum length.
- Both source columns are of the type NCHAR:
  The derived column is of the type NCHAR with the longer of the two lengths.
- One source column is of the type NVARCHAR and the other source column is of the type NCHAR or NVARCHAR:
  The derived column is of the type NVARCHAR with the greater length or greater maximum length.
- Both source columns are an integer or fixed-point type (INT, SMALLINT, NUMERIC, DEC):
  The derived column is of type integer or fixed-point.
  - The number of digits to the right of the decimal point is the greater of the two values of the source columns.
  - The total number of significant digits is the greater of the two values plus the greater of the two values for the number of digits after the decimal point of the source column with a maximum number of 31 digits.
- One source column is of a floating-point type (REAL, DOUBLE, FLOAT), the other is of any numeric data type:
  The derived column is of the type DOUBLE PRECISION.
- Both source columns have a date and time data type:
  Both columns must have the same date and time data type and the derived column also has this data type.

Example

Determine all customer numbers from which orders are currently planned or agreed contractually.

```
SELECT cust_num FROM customers
   EXCEPT DISTINCT
   SELECT cust_num FROM orders WHERE order_stat < 3
```

6.8 Updatability of query expressions

The following is defined regarding the updatability of query expressions:

- Whether a view can be updated
- Whether a base table or updatable view can be updated via a cursor

A base table is updatable.
A table function returns an unchangeable (“read-only”) table.

6.8.1 Rules for updatable query expressions

A query expression is updatable if the following conditions are fulfilled:

- The query expression does not contain a join expression.
- The query expression does not contain a UNION or EXCEPT operation.
- Only column names can be specified in the SELECT list. Other elements of an expression, e.g. subqueries, function calls or literals, are not permitted. Atomic columns cannot be specified more than once. Subareas from multiple columns cannot overlap.
- Only a table or updatable subquery can be specified in the FROM clause. If a table is specified, it must be a base table or an updatable view.
- No subquery can occur in the WHERE clause.
- The keyword DISTINCT cannot be specified.
- The SELECT expression cannot include a GROUP BY or HAVING clause.

6.8.2 Updatable view

A view is updatable if the query expression with which the view was defined is updatable. An updatable view can be specified in INSERT, MERGE, UPDATE and DELETE.

6.8.3 Update via cursor

A table can be updated via a cursor if the cursor description is updatable, i.e. the underlying query expression is updatable and no ORDER BY clause is specified. In addition, no SCROLL clause or FOR READ ONLY clause can be specified in the cursor declaration.

Use DELETE...WHERE CURRENT OF to delete rows in the updatable table via the cursor.

Use UPDATE...WHERE CURRENT OF to update rows in the updatable table via the cursor.

7 Routines

SESAM/SQL distinguishes between the following routines:

- **Procedures (Stored Procedure)**
- **User Defined Functions (UDFs)**.

> In SESAM/SQL, the generic term **routine** is used for procedures and User Defined Functions (UDFs) if the information applies both for procedures and for UDFs.
>
> The generic term “SQL-invoked routine” from the SQL standard is not used in SESAM/SQL.

This chapter first describes common features and differences between procedures and UDFs.

It then includes a number of sections providing detailed descriptions of Procedures (Stored Procedures) and User Defined Functions (UDFs).

These are followed by information on the topics in which procedures and UDFs do not differ or differ only slightly:

- EXECUTE privilege for routines
- Information on routines
- Pragmas in routines
- Control statements in routines
- COMPOUND statement in routines
- Diagnostic information in routines

Common features of routines

A routine is used to store and manage sequences of SQL statements in the database which can be executed later with a single call. A routine is comparable to a subroutine which runs entirely in the DBH, in other words without exchanging data with the application program.

In contrast to a subroutine (in ESQL-COBOL), a routine can be used on different clients with different programming languages (e.g. via JDBC).

All database accesses can be centralized and controlled using routines. Individual SQL statements can also be activated in this way. They can then also be integrated into other routines and SQL statements according to the “modular design principle”.

Routines can also be used to facilitate writing.

The application programmer needs no knowledge of the structure of the database. The routine can be created by a database specialist, who (except for SQL) requires no programming knowledge.

Changes to the database structure do not necessarily affect the application programs. It may be sufficient to modify routines. Recompiling and relinking programs is unnecessary in such cases.

For safety's sake, only the EXECUTE privilege is required to execute the routine concerned. Global table and column privileges are no longer required.

Routines are stored directly in the database (with a complete audit trail). Separate management to manage routines outside the database is not required.

Differences between procedures and User Defined Functions

Procedures and UDFs have an identical range of functions. However, in UDFs of SESAM/SQL, SQL statements are not permitted for modifying data.

Procedures and UDFs also differ in how they are called and in their return information:

- Procedures are called using the SQL statement CALL.
  They have any number of output parameters but no return value.
- UDFs are called by means of their function call in an expression.
  They have precisely one return value.

UDFs can be called in views. Procedures cannot.

7.1 Procedures (Stored Procedures)

In SESAM/SQL the term **procedure** is used to refer to a "Stored Procedure".

7.1.1 Creating a procedure

A procedure is created using the SQL statement CREATE PROCEDURE, see "CREATE PROCEDURE - Create procedure". A procedure can also be created using the SQL statement CREATE SCHEMA, see "CREATE SCHEMA - Create schema".

Procedures can be defined with input, input/output, and output parameters.

> *Recommendation* Parameter names should differ from column names (e.g. by assigning a prefix such as `par_`).

When a procedure is created, the current authorization identifier must have the EXECUTE privilege for the routines called directly in the procedure. It must also, for all tables and columns which are addressed in the procedure, have the privileges which are required to execute the DML statements contained in the procedure.

The procedure text in SESAM/SQL is written entirely in the SQL programming language. The following SQL statements for data searching and data manipulation are permitted in procedures, see section "CREATE PROCEDURE - Create procedure":

SQL statement without a cursor	**Function in the procedure**	see
SELECT	Reads a single row	"SELECT - Read individual rows"
INSERT	Insert rows in a table	"INSERT - Insert rows in table"
UPDATE	Changes the columns of the rows in a table which satisfy a particular search condition	"UPDATE - Update column values"
DELETE	Deletes the rows in a table which satisfy a particular search condition	"DELETE - Delete rows"
MERGE	Depending on a particular condition, changes rows in a table or enters rows in a table	"MERGE - Insert rows in a table or update column values"
SQL statement with a cursor	**Function in the procedure**	see
OPEN	Opens a local cursor	"OPEN - Open cursor"
FETCH	Positions a local cursor and, if necessary, reads the current row	"FETCH - Position cursor and read row"
UPDATE	Changes the columns of the row in a table to which the cursor is positioned	"UPDATE - Update column values"
DELETE	Deletes the row in a table to which the cursor is positioned	"DELETE - Delete rows"
CLOSE	Closes a local cursor	"CLOSE - Close cursor"

Table 25: SQL statements for data manipulation in procedures (section 2 of 2)

In addition to the SQL statements mentioned above, a procedure can also contain control statements (see section “Control statements in routines”) and diagnostic statements (see section “Diagnostic information in routines”).

A procedure may not contain any dynamic SQL statements or cursor descriptions, see section “Dynamic SQL”.

The current authorization identifier automatically obtains the EXECUTE privilege for the procedure created. If it even has authorization to pass on the relevant privileges, it may also pass on the EXECUTE privilege to other authorization identifiers.

An SQL statement in a procedure may access the parameters of the procedure and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

comments

Descriptive comments (see "Comments") can be inserted in a procedure as required.

7.1.2 Execute a procedure

A procedure is executed using the SQL statement CALL, see "CALL - Execute procedure". A procedure can also be called using a dynamic CALL statement.

When a procedure expects input parameters, the corresponding values (arguments) must be transferred to the procedure in the CALL statement.

Output values of procedures which are called outside a routine care stored in corresponding host variables or in the SQL descriptor area. Output values of procedures which are called in a higher-level routine are entered in output parameters or in local variables of the higher-ranking procedure.

In order to execute a procedure, the current authorization identifier requires the EXECUTE privilege for the procedure to be executed, but not the privileges which are required to execute the DML statements contained in the procedure. In addition, the SELECT privileges for the tables which are addressed in the routine’s call parameters by means of subqueries are required.

7.1.3 Delete a procedure

A procedure is deleted using the SQL statement DROP PROCEDURE, see "DROP PROCEDURE - Delete procedure".

7.1.4 Examples of procedures

Example 1: Access check

The `CUSTOMERS_LOGIN` procedure below implements a simple form of access check for customers. It belongs to the sample procedures in the demonstration database of SESAM/SQL (see the " Core manual").

> i In the demonstration database you will find further, detailed examples of sample procedures embedded in an order system.

The `CUSTOMERS_LOGIN` procedure uses only the `CONTACTS` table from the demonstration database. A check is made to see whether the customer is already stored in the table.

```
************************************************************************
* Define CUSTOMERS_LOGIN procedure
************************************************************************
SQL CREATE PROCEDURE CUSTOMERS_LOGIN                                    (1)
( -
   IN  PAR_CUST_NUM     INTEGER,                                        (2)
   IN  PAR_CONTACT_NUM  INTEGER,
   OUT PAR_STATUS       CHAR(40),
   OUT PAR_TITLE        CHAR(20),
   OUT PAR_LNAME        CHAR(25)
)
READS SQL DATA                                                          (3)
BEGIN                                                                   (4)
   /* Variables definition     */                                       (5)
   DECLARE VAR_EOD SMALLINT DEFAULT 0;
   /* Handler definition       */                                       (6)
   DECLARE CONTINUE HANDLER FOR NOT FOUND
      SET VAR_EOD = 1;                                                  (7)
   /* Statements              */                                        (8)
   SET PAR_TITLE   = ' ';
   SET PAR_LNAME = ' ';
   /* Check whether customer is already known      */
   SELECT TITLE, LNAME INTO PAR_TITLE, PAR_LNAME
      FROM CONTACTS
      WHERE CONTACT_NUM = PAR_CONTACT_NUM
      AND   CUST_NUM   = PAR_CUST_NUM;
   IF VAR_EOD = 1 THEN                                                 (9)
      SET PAR_STATUS = 'Customer unknown';
   ELSE
      SET PAR_STATUS = 'Login successful';
   END IF;
END                                                                    (10)
```

1. Procedure header with details of the procedure name (the database and schema names are predefined).
2. List of the procedure parameters.
3. The procedure can contain SQL statements for reading data, but no SQL statements for updating data.
4. The (only) procedure statement is a (non-atomic) COMPOUND statement. This executes further procedure statements in a common context.
5. Definition of local procedure variables.

6. Definition of exception handling in accordance with the SQLSTATE. In this case the procedure is continued if an SQLSTATE of class 02xxx (no data) occurs.
7. In the event of an exception, the local variable `VAR_EOD` is set.
8. The procedure statements will follow.
9. The procedure's output fields are supplied with values in accordance with the result of the query statement.
10. End of the COMPOUND statement and procedure.

Example 2: Complex COMPOUND statement

The `MyTables` procedure below consists of a complex COMPOUND statement and shows the various methods of exception handling. In the central base table `mySchema.myTabs` it stores the names of the tables which the current authorization identifier may access.

The input parameter `par_type` specifies whether base tables or views must taken into account. In the case of `par_type='B'` the names of the base tables are stored, and in the case of `par_type='V'` the names of the views. The following output parameters are returned:

`par_nbr_tables`

Total number of table names of the table type concerned (base table or view) which is stored for the current user

`par_nbr_new_tables`

Number of table names stored in addition for the current user by the procedure call

`par_message`

Message text (OK or error message)

```
-- Procedure header
CREATE PROCEDURE ProcSchema.MyTables
   ( IN par_type CHAR(1), OUT par_message CHAR(80),
     OUT par_nbr_tables INTEGER, OUT par_nbr_new_tables INTEGER )
   MODIFIES SQL DATA
-- Procedure body, COMPOUND statement, declaration section
myTab: BEGIN ATOMIC
   DECLARE var_table_type CHAR(18);
   DECLARE var_schema_name,var_table_name CHAR(31);
   DECLARE var_eot SMALLINT DEFAULT 0;
   DECLARE var_nbr_old_tables INTEGER DEFAULT 0;
   DECLARE myCursor CURSOR FOR
           SELECT table_schema, table_name
           FROM information_schema.tables
           WHERE table_type = var_table_type;
-- Error routines
   DECLARE EXIT HANDLER FOR SQLSTATE '42SND'
           SET par_message = 'catalog ' || CURRENT_REFERENCED_CATALOG
                                        || ' not accessible';
   DECLARE CONTINUE HANDLER FOR SQLSTATE '23SA5'
   -- Primary key not unique
           SET var_nbr_old_tables = var_nbr_old_tables + 1;
   DECLARE EXIT HANDLER FOR SQLSTATE '42SQK'
           SET par_message = 'table MyTabs not accessible';
   DECLARE UNDO HANDLER FOR SQLEXCEPTION
           BEGIN -- COMPOUND statement
```

```
            SET par_message = 'unexpected error';
            SET par_nbr_tables = 0;
            SET par_nbr_new_tables = 0;
            END;
   DECLARE CONTINUE HANDLER FOR SQLWARNING
            SET par_message = 'warning ignored';
   DECLARE CONTINUE HANDLER FOR NOT FOUND
            SET var_eot = 1;
-- Set initial values
   SET par_message = 'OK';
   SET par_nbr_tables = 0;
   SET par_nbr_new_tables = 0;
   IF par_type = 'V' THEN SET var_table_type = 'VIEW';
      ELSEIF par_type = 'B' THEN SET var_table_type = 'BASE TABLE';
      ELSE SET par_message = 'wrong input parameter par_type';
           LEAVE myTab;
   END IF;
-- Procedure statements
   OPEN myCursor;
   loop1: LOOP
          FETCH myCursor INTO var_schema_name, var_table_name;
          IF var_eot = 1 -- Set by error handler for error class 'not found'
             THEN LEAVE loop1; -- End of tables reached
          END IF;
          INSERT INTO mySchema.myTabs VALUES
                 (var_schema_name, var_table_name, var_table_type,
                  current_user, current_date);
          SET par_nbr_tables = par_nbr_tables + 1;
   END LOOP loop1;
   CLOSE myCursor;
   SET par_nbr_new_tables = par_nbr_tables - var_nbr_old_tables;
   -- var_nbr_old_tables set by error handler for SQLSTATE '23SA5'
END myTab
```

Example 3: Different CALLs

The `min_service_price` procedure returns the lowest service record for this order on the basis of the order number transferred.
If the NULL value was transferred as the order number, the value `-999` is returned as the service record.
If the order number exists but the service record is not significant in any of the rows concerned, the NULL value is returned.
If the order number does not exist, the CALL statement is terminated with SQLSTATE ("no data").

```
-- Procedure header
   CREATE PROCEDURE min_service_price
   ( IN in_anr CHAR(8), OUT out_service_price NUMERIC(6) )
   READS SQL DATA
-- Procedure body
   IF in_anr IS NULL THEN out_service_price = -999;
      ELSE SELECT MIN(service_price) INTO out_service_price FROM service
      WHERE anr = in_anr;
   END IF
```

The reactions to various CALLs of the procedure are illustrated using this procedure.

It must be noted that the in_anr and out_service_price parameters have no indicators (not permitted). The significance of in_anr is checked directly via IS NULL. Output parameter out_service_price can be assigned the NULL value directly in the INTO clause.

Various static CALL statements will now be examined. The argument for the input value can be presented in very different ways. On the other hand a host variable must always be specified as an argument for the output value. It must have a numeric data type (compatible with NUMERIC(6)). It also makes sense to use an indicator variable which must be initialized with -1 before the CALL. Otherwise the host variable itself must have been initialized with a correct value (according to its data type).

```
CALL min_service_record(:anr, :service_price INDICATOR :ind-service_price)
```

The input value is transferred as a host variable. As the NULL value can be returned, it makes sense to specify an indicator variable for the output value.

```
CALL min_service_record(:anr :ind-anr, :service_price :ind-service_price)
```

As above, but setting :ind-anr to -1 means that the NULL value can also be transferred.

```
CALL min_service_record('A#123456', :service_price)
```

The specific input value is A#123456. If the NULL value is to be returned for this, the specification of an indicator variable is missing, which results in an SQLSTATE SEW2202.

```
CALL min_service_record(CAST(NULL AS CHAR (8)), :service_price)
```

As the input value is NULL, the value -999 is returned. As the host variable :service_price has no indicator, it must have been initialized with the correct value (according to its data type) before the call.

```
CALL min_service_record((SELECT MAX(anr) FROM leistung),:service_price :
indservice_price)
```

The input value is the highest order number. As the NULL value can be returned, it makes sense to specify an indicator variable for the output value. If the service table is empty, the NULL value is then returned.

7.2 User Defined Functions (UDFs)

In SESAM/SQL, the abbreviation **UDF** is used for “User Defined Function”.

> **i** UDFs can be used in almost all expressions by means of their function call. They can occur in the DML statements and in the utility statements EXPORT ... WHERE and UNLOAD ONLINE.

7.2.1 Creating a UDF

A UDF is created using the SQL statement CREATE FUNCTION, see "CREATE FUNCTION - Create User Defined Function (UDF)". A UDF can also be created using the SQL statement CREATE SCHEMA, see "CREATE SCHEMA - Create schema".

UDFs can be defined with input parameters.

> *Recommendation* Parameter names should differ from column names (e.g. by assigning a prefix such as `par_`).

When a UDF is created, the current authorization identifier must have the EXECUTE privilege for the routines called directly in the UDF. It must also, for all tables and columns which are addressed in the UDF, have the (SELECT) privileges which are required to execute the DML statements contained in the routine.

The text of the UDF in SESAM/SQL is written entirely in the SQL programming language. The following SQL statements for data searching are permitted in UDFs, see section “CREATE FUNCTION - Create User Defined Function (UDF)”:

SQL statement **without a cursor**	**Function in the UDF**	**see**
SELECT	Reads a single row	"SELECT - Read individual rows"
SQL statement **with a cursor**		
OPEN	Opens a local cursor	"OPEN - Open cursor"
FETCH	Positions a local cursor and, if necessary, reads the current row	"FETCH - Position cursor and read row"
CLOSE	Closes a local cursor	"CLOSE - Close cursor"

Table 26: SQL statements for data manipulation in UDFs

SQL statements for modifying data (INSERT, UPDATE, DELETE, MERGE) are **not** permitted in the UDFs of SESAM/SQL.

In addition to the SQL statements mentioned above, a procedure can also contain control statements (see section “Control statements in routines”) and diagnostic statements (see section “Diagnostic information in routines”).

A UDF may not contain any dynamic SQL statements or cursor descriptions, see section “Dynamic SQL”.

The current authorization identifier automatically obtains the EXECUTE privilege for the UDF created. If it even has authorization to pass on the relevant privileges, it may also pass on the EXECUTE privilege to other authorization identifiers.

An SQL statement in a UDF may access the parameters of the UDF and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

comments

Descriptive comments (see "Comments") can be inserted in a UDF as required.

7.2.2 Executing a UDF

A UDF is called by means of its function call in an expression, see "User Defined Functions (UDFs)".

When a UDF expects input parameters, the corresponding values (arguments) must be transferred to the UDF in the function call.

The (only) return value of a UDF is determined by the RETURN statement, see "RETURN - Supply the return value of a User Defined Function (UDF)".

The EXECUTE privilege for the UDF to be executed is required to execute a UDF, but not the privileges which are required to execute the DML statements contained in the UDF. In addition, the SELECT privileges for the tables which are addressed in the routine’s call parameters by means of subqueries are required.

When an expression is evaluated, the function contained in it is performed and then the replaced by the calculated return value.

UDFs can be called in views.

7.2.3 Deleting a UDF

A UDF is deleted using the SQL statement DROP FUNCTION, see "DROP FUNCTION - Delete User Defined Function (UDF)".

7.2.4 Uncorrelated function calls

Function calls of a UDF with constant input values are referred to as **uncorrelated function calls**. Constant input values do not refer to the SQL statement which contains the function call,

Uncorrelated function calls are handled by SESAM/SQL as follows when the statement is executed:

- Function values of uncorrelated function calls are calculated once only to evaluate conditions.
- However, they are recalculated every time for the following output values:
  - in SELECT lists
  - for ORDER BY values
  - for values in INSERT rows
  - for UPDATE... SET ... values
  - for the INSERT- / UPDATE values in a MERGE statement

Example

```
SELECT f(1,2) FROM t WHERE col < g(5+4,8,9)
```

The function calls `f(1,2)` and `g(5+4,8,9)` of this SQL statement are uncorrelated.

The function `g` is calculated once only in order to evaluate the records of `t`. The set of hits of the query is then determined with this constant result. This also enable indexes to be used in the condition evaluation.

In the SELECT list, on the other hand, the `f` function is recalculated for each set of hits.

VOLATILE / IMMUTABLE annotations

The `/*% VOLATILE %*/` and `/*% IMMUTABLE %*/` annotations control the execution of uncorrelated function calls. In a function call, they are accepted only between the name of the function and the opening parenthesis for the function parameters. In any other position these annotations lead to a syntax error for the statement.

When `/*% VOLATILE %*/` is specified, the function value is always recalculated.

When `/*% IMMUTABLE %*/` is specified in an uncorrelated function call, the function value is **not** calculated again. The function value calculated beforehand is used. The function value is recalculated when the first function call takes place.

When these annotations are not specified, the SESAM/SQL procedure described above is used.

Example

```
SELECT f /*% VOLATILE %*/ (1,2)

FROM t WHERE col < g /*% IMMUTABLE %*/ (5+4,8,9)
```

These function calls map the existing SESAM/SQL procedure with annotations.

```
SELECT f /*% IMMUTABLE %*/ (1,2)
```

```
FROM t WHERE col < g /*% VOLATILE %*/ (5+4,8,9)
```

Specifying the annotations always causes function g to be recalculated. Function f is only calculated once.

7.2.5 Examples of UDFs

Example 1: Determining the year number

The `GetCurrentYear` UDF below returns the current year as a number. It contains no SQL statements for reading or updating data.

```
CREATE FUNCTION GetCurrentYear (IN time TIMESTAMP(3))
   RETURNS DECIMAL(4)
   CONTAINS SQL
   RETURN EXTRACT (YEAR FROM time)
```

The `GetCurrentYear` UDF in the schema `FuncSchema` is used:

- Determining all orders of the year 2014:

  ```
  DECLARE cursor_1 CURSOR FOR
  SELECT order_number, customer_name FROM orders
  WHERE FuncSchema.GetCurrentYear(order_completion_date) = 2014
  ```

- Set expiration year to the year after next (schema `FuncSchema` is preset):

  ```
  UPDATE model.exemplar
  SET expiration_year = GetCurrentYear(CURRENT_TIMESTAMP(3)) + 2
  ```

Example 2: Determining the price of an item

```
CREATE FUNCTION ITEM_PRICE (IN P_ITEMNUM INTEGER)
   RETURNS NUMERIC(8,2)
   READS SQL DATA
   BEGIN
      RETURN (SELECT PRICE FROM PARTS.ITEM WHERE ITEMNUM= P_ITEMNUM);
   END
```

Example 3: Anonymizing a credit card number

The UDF `mask_credit_card_number` below anonymizes a credit card number by masking the last four digits:

```
CREATE FUNCTION mask_credit_card_number(IN card_no CHAR(16))
   RETURNS CHAR(16)
   CONTAINS SQL
   RETURN SUBSTRING(card_no FROM 1 FOR 12) || '****'
```

A notification could thus be structured as follows:

```
Select surname, first_name, mask_credit_card_number(credit_card_number)
from ...
```

7.3 EXECUTE privilege for routines

SESAM/SQL provides the EXECUTE privilege for routines. It is assigned using the SQL statement GRANT and revoked using the SQL statement REVOKE.

When a routine is created, the current authorization identifier must have the EXECUTE privilege for the routines called directly in the routine. It must also, for all tables and columns which are addressed in the routine, have the privileges which are required to execute the DML statements contained in the routine.

When a view is created, the current authorization identifier must have the EXECUTE privilege for the UDFs called directly in the view.

The EXECUTE privilege for the routine to be executed is required to execute a routine (with the SQL statement CALL or using a function call), but not the privileges which are required to execute the DML statements contained in the routine. In addition, the SELECT privileges for the tables which are addressed in the routine's call parameters by means of subqueries are required.

7.4 Information on routines

Information on routines is provided in the information schemas, see chapter “Information schemas”.

Information schema	View	Information on
INFORMATION_SCHEMA	PARAMETERS	Parameters of routines
INFORMATION_SCHEMA	ROUTINES	Routines
INFORMATION_SCHEMA	ROUTINE_PRIVILEGES	Privileges for routines
INFORMATION_SCHEMA	ROUTINE_TABLE_USAGE	Tables in routines
INFORMATION_SCHEMA	ROUTINE_COLUMN_USAGE	Columns in routines
INFORMATION_SCHEMA	ROUTINE_ROUTINE_USAGE	Routines in other routines
INFORMATION_SCHEMA	VIEW_ROUTINE_USAGE	Routines in views
SYS_INFO_SCHEMA	SYS_PARAMETERS	Parameters of routines
SYS_INFO_SCHEMA	SYS_ROUTINES	Routines
SYS_INFO_SCHEMA	SYS_ROUTINE_PRIVILEGES	Privileges for routines
SYS_INFO_SCHEMA	SYS_ROUTINE_USAGE	Tables and columns in routines
SYS_INFO_SCHEMA	SYS_ROUTINE_ERRORS	Error events in routines
SYS_INFO_SCHEMA	SYS_ROUTINE_ROUTINE_USAGE	Routines in other routines
SYS_INFO_SCHEMA	SYS_VIEW_ROUTINE_USAGE	Routines in views

Table 27: Routines in the information schemas

7.5 Pragmas in routines

The following pragmas are provided specifically for routines:

- DEBUG ROUTINE to output additional information or error information
- DEBUG VALUE to output additional information for the SQL statements SET in routines and RETURN in UDFs
- LOOP LIMIT to limit the number of loop passes

See section "Pragmas and annotations".

The DEBUG ROUTINE and LOOP LIMIT pragmas are only effective ahead of the SQL statement CALL and ahead of the DML statements DECLARE CURSOR, DELETE, INSERT, MERGE, SELECT, and UPDATE. When specified **ahead of** DML statements, these pragmas have an effect on all UDFs and the routines of the DML statement these contain. When placed ahead of SQL statements, these pragmas have no effect **in** a routine.

Other pragmas can also be used in the CALL statement and in routines.

Pragmas EXPLAIN, CHECK, LIMIT ABORT_EXECUTION

These pragmas are effective ahead of the SQL statement CALL and ahead of the DML statements DECLARE CURSOR, DELETE, INSERT, MERGE, SELECT, and UPDATE. When specified ahead of DML statements, they have an effect on all UDFs of the DML statement and all routines contained in these UDFs. When one of these pragmas precedes an SQL statement in a routine, it is ignored.

Pragmas ISOLATION LEVEL, LOCK MODE

When these pragmas precede a CALL statement, they only influence the possibly complex call values of the CALL statement.

These pragmas can also precede SQL statements in routines. They then have the effect described under DECLARE CURSOR, DELETE, INSERT, MERGE, SELECT, and UPDATE.

When these pragmas precede an IF statement, they only influence the conditions of the IF statement. These pragmas can also be specified ahead of the statements contained in the IF statement.

In the case of the SET statement, these pragmas influence the evaluation of the expression on the right-hand side of the assignment.

When these pragmas precede a LOOP, LEAVE, or ITERATE statement, they are ignored.

When these pragmas precede a FOR statement, they only influence the cursor definition of the FOR statement. These pragmas can also be specified ahead of the SQL statements contained in the FOR statement.

When these pragmas precede a WHILE statement, they only influence the condition of the WHILE loop. These pragmas can also be specified ahead of the SQL statements contained in the WHILE statement.

When these pragmas are to influence the UNTIL condition of a REPEAT statement, they must be specified immediately ahead of UNTIL (not ahead of REPEAT). These pragmas can also be specified ahead of the SQL statements contained in the REPEAT statement.

When these pragmas precede a CASE statement, they only influence the expressions outside of the THEN and ELSE statement blocks. These pragmas can also be specified ahead of the SQL statements contained in the CASE statement.

In the case of the RETURN statement, these pragmas have an effect on the evaluation of the RETURN value.

In the case of all other statements in routines, these pragmas have no effect.

Pragmas IGNORE, JOIN, KEEP JOIN ORDER, OPTIMIZATION, SIMPLIFICATION, USE

When one of these optimization pragmas precedes a CALL statement, it only influences the optimization of the possibly complex call values of the CALL statement.

These pragmas can also precede SQL statements of a routine. They then implement the optimization described under DECLARE CURSOR, DELETE, INSERT, MERGE, SELECT, and UPDATE.

When these pragmas precede an IF statement, they only influence the optimization of the IF statement's conditions. These pragmas can also be specified ahead of the statements contained in the IF statement.

In the case of the SET statement, these pragmas influence the optimization of the expression on the right-hand side of the assignment.

When these pragmas precede a LOOP, LEAVE, or ITERATE statement, they are ignored.

When these pragmas precede a FOR statement, they only influence the cursor definition of the FOR statement. These pragmas can also be specified ahead of the SQL statements contained in the FOR statement.

When these pragmas precede a WHILE statement, they only influence the condition of the WHILE loop. These pragmas can also be specified ahead of the SQL statements contained in the WHILE statement.

When these pragmas are set to influence the UNTIL condition of a REPEAT statement, they must be specified immediately ahead of UNTIL (not ahead of REPEAT). These pragmas can also be specified ahead of the SQL statements contained in the REPEAT statement.

When these pragmas precede a CASE statement, they only influence the expressions outside of the THEN and ELSE statement blocks. These pragmas can also be specified ahead of the SQL statements contained in the CASE statement.

In the case of the RETURN statement, these pragmas have an effect on the evaluation of the RETURN value.

In the case of all other statements in routines, these pragmas have no effect.

Pragmas DATA TYPE, PREFETCH, UTILITY MODE

These pragmas are ignored when they precede a CALL statement or an SQL statement of a routine.

7.6 Control statements in routines

Control statements may only be specified in routines. They control execution of a routine,e.g. by means of loops or conditions. They can become extensive and in turn contain sequences of SQL statements themselves.

SQL statement	Function	see
COMPOUND	Executes SQL statements in a common context	"COMPOUND - Execute SQL statements in a common context"
CALL	Call a procedure	"CALL - Execute procedure"
CASE	Executes SQL statements conditionally	"CASE - Execute SQL statements conditionally"
FOR	Executes SQL statements in a loop	"FOR - Execute SQL statements in a loop"
IF	Executes SQL statements conditionally	"IF - Execute SQL statements conditionally"
ITERATE	Switches to the next loop pass	"ITERATE - Switch to the next loop pass"
LEAVE	Terminates loop or COMPOUND statement	"LEAVE - Terminate a loop or COMPOUND statement"
LOOP	Executes SQL statements in a loop	"LOOP - Execute SQL statements in a loop"
REPEAT	Executes SQL statements in a loop	"REPEAT - Execute SQL statements in a loop"
RETURN [1]	Supplies the return value of a User Defined Function (UDF)	"RETURN - Supply the return value of a User Defined Function (UDF)"
SET	Assigns a value	"SET - Assign value"
WHILE	Executes SQL statements in a loop	"WHILE - Execute SQL statements in a loop"

Table 28: Control and diagnostic statements of routines

[1]For UDFs only

In SESAM/SQL V9.0 and higher, nested calls of routines are permitted. The CALL statement is therefore one of the SQL statements permitted in a routine.

7.7 COMPOUND statement in routines

The COMPOUND statement is one of the control statements in routines. It executes further SQL statements in a common context. Common local data, common local cursors, and common exception routines apply for these SQL statements.

A detailed description of the COMPOUND statement is provided on "COMPOUND - Execute SQL statements in a common context".

Local data

Local data comprises variables or exception names which can only be addressed in the COMPOUND statement. The names of the local data must differ from each other.

A data type and, if required, a default value is defined for variables. They have no indicator variable. They can be used in local cursor definitions, local exception routines, and the SQL statements of the COMPOUND statement.

To facilitate understanding, exception names define a name for an exception (without specifying an associated SQLSTATE) or a name for an SQLSTATE. They can be used in local exception routines, see "COMPOUND - Execute SQL statements in a common context".

Local cursors

With the definition of local cursors, cursors are defined which can only be addressed in the COMPOUND statement. The names of the local cursors must differ from each other.

Local cursors can be used in local exception routines and the SQL statements of the COMPOUND statement.

Local cursors are defined without the WITH HOLD clause. The SQL statements STORE and RESTORE may not be applied to local cursors.

Common exception routines

The definition of exception routines determines what response is made when, during processing of an SQL statement in the context of the COMPOUND statement, an SQLSTATE '00000' is reported.

When an SQLSTATE occurs which was specified in an exception routine, the exception routine for the SQLSTATE is executed. For other SQLSTATEs, SESAM/SQL automatically performs exception handling.

The type of exception handling is defined in the exception routines in accordance with the SQLSTATE. When an exception occurs, further SQL statements there decide whether the routine should be continued or terminated. Changes which were made in the context of the COMPOUND statement can be undone.

7.8 Diagnostic information in routines

SESAM/SQL provides diagnostic information in routines. The SQL standard uses the term “diagnostics management” for this.

Diagnostic information is provided in a diagnostics area for an SQL statement executed beforehand. In the case of routines in SESAM/SQL, multiple diagnostics areas can exist at one time (for an SQL statement, for calling an (exception) routine), in particular for nested routines.

> i At the ESQL-Cobol interface, in other words in the application program, the diagnostics area is named “SQLda”.

The following SQL statements, which may only be used in routines, enable a diagnostics area to be accessed in read and/or write mode:

SQL statement	Function	see
GET DIAGNOSTICS	Outputs diagnostic information about a statement	"GET DIAGNOSTICS - Output diagnostic information"
SIGNAL	Reports exception in routine	"SIGNAL - Report exception in routine"
RESIGNAL	Reports exception in local exception routine	"RESIGNAL - Report exception in local exception routine"

Table 29: Control and diagnostic statements of routines

You can improve the programming of routines using these diagnostic statements and the self-defined SQLSTATEs described below. You can analyze exceptions which occur more precisely and respond to these in a differentiated manner.

Success of an SQL statement in a routine

To simplify the description, the success of an SQL statement in a routine is defined as follows in this manual:

- The SQL statement was **successful** if it was terminated with SQLSTATE '00000'.
- The SQL statement was **error-free** if it was terminated with SQLSTATE '00000', an SQLSTATE of the classes '01xxx' (warning) or '02xxx' (no data).
- The SQL statement in a routine was **errored** if it was not terminated error-free.

> i A routine is continued after an error-free SQL statement if no exception routines are defined for the SQLSTATEs of the classes '01000' and '02000'. If, for instance, a warning occurs for an SQL statement in a procedure, the corresponding CALL statement is terminated with SQLSTATE '00000'.

Self-defined SQLSTATEs

SESAM/SQL V9.0 and higher enables you to define SQLSTATEs yourself. The class '46Sxx' (where x is a number or an uppercase letter) is reserved. In this class you can define up to 1296 SQLSTATEs yourself. This class is used neither by the SQL standard nor by SESAM/SQL.

You can specify self-defined SQLSTATEs in the diagnostic statements SIGNAL and RESIGNAL.You can call a specific exception routine on a targeted basis in the SIGNAL diagnostic routine using a self-defined SQLSTATE. In

the exception routine you can use the RESIGNAL diagnostic statements to abort the routine specifically. In both statements you can also enter additional diagnostic information in the diagnostics area.

There are no ready-made SESAM message texts for self-defined SQLSTATEs. When a self-defined SQLSTATE occurs in the application program as an unspecified SQLSTATE, SESAM/SQL generates the message `SEW46xx (&00)` from it. The `MESSAGE_TEXT` from the diagnostics area then appears as insert `(&00)`. This enables you to generate a message text of your own (without an accompanying help text) indirectly in the diagnostic statements SIGNAL and RESIGNAL.

SQLSTATE '45000' (unhandeled SQLSTATE)

With SESAM/SQL you can define a local exception name for an SQLSTATE in a COMPOUND statement, see section “Local data”.

However, you can also define an exception name with no link to an SQLSTATE.

With this exception name you can call a specific exception routine in the SIGNAL diagnostic routine. If this exception routine does not exist or is exited with RESIGNAL (without specifying an SQLSTATE), the routine is terminated with the SQLSTATE '45000'.

SESAM/SQL then generates the following message:

```
SEW4500 UNHANDLED USER DEFINED EXCEPTION (&00). (&01)
```

Insert `(&00)` contains the exception name. If a `MESSAGE_TEXT` was specified for SIGNAL or RESIGNAL, `(&01)` appears as an insert.

When an appropriate exception name and possibly a corresponding `MESSAGE_TEXT` is selected, the user then receives an informative message.

GET DIAGNOSTICS

GET DIAGNOSTICS ascertains information on an SQL statement executed beforehand in a routine and enters this in a procedure parameter (output) or a local variable. The information relates to the statement itself or to the database objects affected by it.

GET DIAGNOSTICS changes neither the content nor the sequence of diagnostics areas. In other words GET DIAGNOSTICS statements which follow each other directly evaluate the same diagnostic information.

A detailed description of the GET DIAGNOSTICS statement is provided on "GET DIAGNOSTICS - Output diagnostic information".

SIGNAL

SIGNAL reports, in a routine, am exception or a self-defined SQLSTATE.

A detailed description of the SIGNAL statement is provided on "SIGNAL - Report exception in routine".

SIGNAL deletes the current diagnostics area and optionally enters the following diagnostic information into the current diagnostics area:

- When an exception name is specified, it is entered as CONDITION_IDENTIFIER. Otherwise a string with the length 0 is assigned.
- The RETURNED_SQLSTATE is supplied:
  - When an SQLSTATE is specified, it is entered as RETURNED_SQLSTATE.

  - When an SQLSTATE is defined for the specified exception name, the defined SQLSTATE is entered for RETURNED_SQLSTATE.
  - Otherwise SQLSTATE '45000' is entered.
- When MESSAGE_TEXT is specified, MESSAGE_TEXT, MESSAGE_LENGTH, and MESSAGE_OCTET_LENGTH are supplied accordingly. Otherwise MESSAGE_TEXT is assigned a string with the length 0.

The routine is continued or terminated with an exception routine:

- When RETURNED_SQLSTATE '45000' and a local exception routine is defined for the RETURNED_SQLSTATE, this exception routine is executed.
- When RETURNED_SQLSTATE = '45000' and a local exception routine is defined for the exception name entered CONDITION_IDENTIFIER, this exception routine is executed.
- Otherwise an unspecified SQLSTATE exists. The routine is terminated with the SQLSTATE entered in RETURNED_SQLSTATE.

Further information:

- Execution of a specific exception routine can be achieved with SIGNAL.
- An SQL statement immediately after the SIGNAL statement is then executed only if the exception routine called by SIGNAL is defined with CONTINUE and was terminated without error.
- If the values (e.g. MESSAGE_TEXT) entered in the diagnostics area for SIGNAL are to be read, GET CURRENT DIAGNOSTICS must be located either immediately after SIGNAL (see preceding note) or it must be used in the exception routine GET STACKED DIAGNOSTICS which is called. This exception routine need not necessarily be part of the current COMPOUND statement. It can also be an exception routine of a higher-ranking routine which has used the routine with the SIGNAL statement. In the latter case, the diagnostics area of the calling statement is then evaluated.
- A routine is continued after an SQL statement which is error-free but not successful. Even if an exception routine was executed with EXIT or UNDO in such a case, the routine terminates with SQLSTATE '00000' unless an SQL statement terminated with an error in the exception routine itself. In such a case, the SIGNAL statement enables the routine to be terminated with a self-defined SQLSTATE.

RESIGNAL

RESIGNAL reports a condition or an SQLSTATE in a local exception routine. In contrast to SIGNAL, the specification of an exception name or SQLSTATE is optional.

A detailed description of the RESIGNAL statement is provided on "RESIGNAL - Report exception in local exception routine".

RESIGNAL uses the diagnostics area of the SQL statement which has activated the exception routine, and if necessary modifies the following diagnostic information:

- If neither an exception name nor SQLSTATE was specified, CONDITION_IDENTIFIER and RETURNED_SQLSTATE remain unchanged. The following applies:
  - RETURNED_SQLSTATE may not contain an SQLSTATE of class '01xxx' or '02xxx'. Otherwise RESIGNAL is terminated with an error.
  - When MESSAGE_TEXT= is specified, RETURNED_SQLSTATE must contain either a self-defined SQLSTATE or the value '45000'. Otherwise RESIGNAL is terminated with an error.
- The current diagnostics area will possibly be modified:

- When an exception name is specified, it is entered as CONDITION_IDENTIFIER. Otherwise a string with the length 0 is assigned.
  - When an SQLSTATE is specified, it is entered as RETURNED_SQLSTATE.
  - When an SQLSTATE is defined for the specified exception name, the defined SQLSTATE is entered for RETURNED_SQLSTATE. Otherwise SQLSTATE '45000' is entered.
- When MESSAGE_TEXT is specified, MESSAGE_TEXT, MESSAGE_LENGTH, and MESSAGE_OCTET_LENGTH are supplied accordingly. Otherwise MESSAGE_TEXT is assigned a string with the length 0.

The routine in which the local exception routine of the RESIGNAL statement was executed is terminated with the SQLSTATE entered in RETURNED_SQLSTATE.

Further information:

- Even after an exception routine defined with EXIT or UNDO has been executed, a routine is terminated with SQLSTATE '00000' unless an SQL statement terminated with an error in the exception routine itself. RESIGNAL enables you to return the SQLSTATE which triggered the exception routine.
- A SIGNAL statement which is called in an exception routine has the same effect as a RESIGNAL statement with explicitly specified exception name or SQLSTATE.

Examples of the use of diagnostic statements

Different situations when querying the SQLSTATE

```
CREATE PROCEDURE proc1() MODIFIES SQL DATA
   BEGIN ATOMIC
      DECLARE state1, state2, state3 CHAR(5);
      DECLARE EXIT HANDLER FOR SQLEXCEPTION
         BEGIN
            DELETE FROM tab1; ------------------------------------------ (3)
            GET STACKED DIAGNOSTICS CONDITION state2 = RETURNED_SQLSTATE;
            GET CURRENT DIAGNOSTICS CONDITION state3 = RETURNED_SQLSTATE;
            ... -------------------------------------------------------- (2)
         END;
      ...
      UPDATE tab2 SET ...;
      GET CURRENT DIAGNOSTICS CONDITION state1 = RETURNED_SQLSTATE; ----- (1)
      ...
   END
```

(1) The local variable `state1` is supplied only when the UPDATE statement has been executed successfully or error-free. It then contains either the SQLSTATE '00000', a warning, or the SQLSTATE '02000' (no data). The exception routine is not executed.

(2) If the UPDATE statement was executed with an error and the DELETE statement was executed without an error, `state2` contains the SQLSTATE of the UPDATE statement which caused the error. `state3` contains the SQLSTATE of the DELETE statement ('00000', a warning, or '02000' (no data)). `state1` is not supplied as the procedure was aborted because of an exception routine (EXIT).

(3) If the DELETE statement of the exception routine was also executed with an error, the procedure is immediately aborted because of the unspecified SQLSTATE. None of the GET DIAGNOSTICS statements is executed.

If the exception routine is defined with CONTINUE (instead of with EXIT) and is executed without error, `state1` is also supplied after an UPDATE statement which was executed with an error. `state1` is then assigned the SQLSTATE of the UPDATE statement which caused the error.

Special handling of the SQLSTATE '02000'

After SQLSTATE '02000' (no data), a routine is normally continued. In the example below, this is accepted in one case and is intended to lead to an error in another.

```
CREATE PROCEDURE proc2(OUT par1 INTEGER, OUT par2 INTEGER) MODIFIES SQL DATA
   BEGIN ATOMIC
      DELETE FROM tab1;
      GET DIAGNOSTICS par1 = ROW_COUNT;
      DELETE FROM tab2;
      GET DIAGNOSTICS par2 = ROW_COUNT;
      IF par2 = 0
         THEN SIGNAL SQLSTATE '46SA1'
              SET MESSAGE_TEXT = 'tab2 must contain at least one record';
      END IF;
   END
```

If the DELETE statement was executed without error, the relevant number of deleted records is entered in the two output parameters. In table `tab1`, the number may also be 0. However, when table `tab2` is empty, the procedure is aborted. Because of the ATOMIC clause, the deletions in table `tab1` are also undone. SESAM/SQL generates the message:

```
SEW46A1 TAB2 MUST CONTAIN AT LEAST ONE RECORD
```

Noting the SQLSTATE which occurred

After an unspecified SQLSTATE, a procedure is aborted and precisely this SQLSTATE is reported. If you also wish to log this event in a table, define, for example, the following exception routine. The RESIGNAL statement returns the SQLSTATE which occurred. Without the RESIGNAL statement, the procedure terminates with SQLSTATE '00000'.

```
CREATE PROCEDURE proc3() MODIFIES SQL DATA
   BEGIN ATOMIC
      DECLARE error CHAR(5);
      DECLARE UNDO HANDLER FOR SQLEXCEPTION
      BEGIN
         GET DIAGNOSTICS CONDITION error = RETURNED_SQLSTATE;
         INSERT INTO logging_tab
           VALUES (CURRENT_TIMESTAMP(3),'SQLSTATE ' || error || ' occurred');
         RESIGNAL;
      END;
   -- procedure body
   ...
   END
```

Search for empty tables

The number of empty tables is to be determined by means of a User Defined Function. If the number of empty tables exceeds the number entered, the search should be aborted with an error.

```
CREATE FUNCTION check_tables(IN max_nbr INTEGER)
                RETURNS INTEGER READS SQL DATA
   BEGIN
      DECLARE "TABLE ERROR" CONDITION;
      DECLARE nbr_empty_tables integer DEFAULT 0;
      DECLARE CONTINUE HANDLER FOR "TABLE ERROR"
         BEGIN
            nbr_empty_tables = nbr_empty_tables + 1;
            IF nbr_empty_tables > max_nbr
               THEN RESIGNAL SET MESSAGE_TEXT = 'TOO MANY EMPTY TABLES';
            END IF;
         END;
   IF (SELECT COUNT(*) FROM tab1) = 0 THEN SIGNAL "TABLE ERROR";
   END IF;
   IF (SELECT COUNT(*) FROM tab2) = 0 THEN SIGNAL "TABLE ERROR";
   END IF;
   IF (SELECT COUNT(*) FROM tab3) = 0 THEN SIGNAL "TABLE ERROR";
   END IF;
   RETURN nbr_empty_tables;
   END
SELECT check_tables(2) INTO :NBR-EMPTY-TABLES FROM TABLE(DEE)
```

If the number of empty tables does not exceed the number entered, the number of empty tables is stored in the user variable `:NBR-EMPTY-TABLES`.

However, if more than two tables exist, the search is terminated with SQLSTATE '45000'. SESAM/SQL then generates the following message:

```
SEW4500 UNHANDLED USER DEFINED EXCEPTION (TABLE ERROR). TOO MANY EMPTY TABLES
```

8 SQL statements

This chapter describes the SQL statements. It is subdivided into two parts:

- Summary of contents
- Alphabetical reference section

8.1 Summary of contents

In this section, the SQL statements are grouped together according to function. This grouping of the statements is oriented to the SQL standard.

SESAM/SQL-specific statements are printed against a gray background.

8.1.1 SQL statements for schema definition and administration

Schema

SQL statement	Function
CREATE SCHEMA	Create a schema
DROP SCHEMA	Delete a schema

Table 30: SQL statements for schemas

Base table

SQL statement	Function
ALTER TABLE	Modify a base table
CREATE TABLE	Create a base table
DROP TABLE	Delete a base table

Table 31: SQL statements for base tables

View

SQL statement	Function
CREATE VIEW	Create a view
DROP VIEW	Delete a view

Table 32: SQL statements for views

Privileges

SQL statement	Function
GRANT	Grant privileges
REVOKE	Revoke privileges

Table 33: SQL statements for privileges

Procedure (Stored Procedure)

SQL statement	Function
CREATE PROCEDURE	create procedure
DROP PROCEDURE	Delete a procedure

Table 34: SQL statements for procedures

User Defined Function (UDF)

SQL statement	Function
CREATE FUNCTION	Create UDF
DROP FUNCTION	Delete UDF

Table 35: SQL statements for User Defined Functions

8.1.2 SQL statements for querying and updating data

Without cursor

SQL statement	Function
DELETE	Delete rows
INSERT	Insert rows in table
MERGE	Insert rows in table or change column values
SELECT...INTO	Read individual rows (static SELECT statement)
SELECT (without INTO)	Read individual rows (dynamic SELECT statement)
UPDATE	Update column values

Table 36: SQL statements for querying and updating data without a cursor

With cursor

The following SQL statements can be used with a static or dynamic cursor.

If an executable statement contains a dynamic cursor, the corresponding cursor description must be prepared before the statement is executed.

In some statements there are certain deviations or restrictions if a dynamic cursor is used. This fact is mentioned in the table.

SQL statement	Function
CLOSE	Close a cursor
DECLARE...CURSOR	Declare a cursor (not executable) Dynamic cursor: the statement identifier for the cursor description is specified instead of the cursor description
DELETE...CURRENT	Delete current row
FETCH	Position cursor and read column value
OPEN	Open a cursor Dynamic cursor: includes USING clause
RESTORE	Restore a cursor
STORE	Save cursor position
UPDATE...CURRENT	Update the current row

Table 37: SQL statements for querying and updating data with cursor

8.1.3 SQL statements for transaction management

SQL statement	Function
SET TRANSACTION	Define the characteristics of an SQL transaction
COMMIT WORK	Commit SQL transaction
ROLLBACK WORK	Roll back SQL transaction.

Table 38: SQL statements for transaction management

8.1.4 SQL statements for session control

SQL statement	Function
SET CATALOG	Set default database name
SET SCHEMA	Set default schema name
SET SESSION AUTHORIZATION	Define authorization identifier
PERMIT	Specify a user identification for SESAM/SQL V1

Table 39: SQL statements for session control

8.1.5 SQL statements for dynamic SQL

Dynamic statement

SQL statement	Function
EXECUTE	Execute a prepared statement
EXECUTE IMMEDIATE	Execute a dynamic statement
PREPARE	Prepare a dynamic statement

Table 40: SQL statements for dynamic statements

Descriptor

SQL statement	Function
ALLOCATE DESCRIPTOR	Request SQL descriptor area
DEALLOCATE DESCRIPTOR	Release SQL descriptor area
DESCRIBE	Query data types of input or output values
GET DESCRIPTOR	Read SQL descriptor area
SET DESCRIPTOR	Modify SQL descriptor area

Table 41: SQL statements for descriptors

8.1.6 WHENEVER statement for ESQL error handling

SQL statement	Function
WHENEVER	Define error handling (not executable)

Table 42: WHENEVER statement for ESQL error handling

8.1.7 SQL statements for managing the storage structure

Storage group

SQL statement	Function
ALTER STOGROUP	Modify a storage group
CREATE STOGROUP	Create a storage group
DROP STOGROUP	Drop a storage group

Table 43: SQL statements for storage groups

Space

SQL statement	Function
ALTER SPACE	Modify space parameter
CREATE SPACE	Create a space
DROP SPACE	Delete a space

Table 44: SQL statements for spaces

Index

SQL statement	Function
CREATE INDEX	Create an index
DROP INDEX	Delete an index
REORG STATISTICS	Re-generate global statistics

Table 45: SQL statements for indexes

8.1.8 SQL statements for managing user entries

SQL statement	Function
CREATE SYSTEM_USER	Create a system entry
CREATE USER	Create an authorization identifier
DROP USER	Delete an authorization identifier
DROP SYSTEM_USER	Delete a system entry

Table 46: SQL statements for managing user entries

8.1.9 Utility statements

Utility statements are statements in SQL syntax for database management.

They are described in the “ SQL Reference Manual Part 2: Utilities”.

8.1.10 Control statements

Routine (Stored Procedure and UDF)

SQL statement[1]	Function
COMPOUND	Statements in the context
CALL	Call a procedure
CASE	Execute statements conditionally
FOR	Execute statements in a loop
IF	Execute statements conditionally
ITERATE	Switch to the next loop pass
LEAVE	Terminate loop or COMPOUND statement
LOOP	Execute statements in a loop
REPEAT	Execute statements in a loop
RETURN	Supply the return value of a User Defined Function (UDF)
SET	Assigns a value
WHILE	Execute statements in a loop

Table 47: SQL statements for procedures

[1]Only in a CREATE PROCEDURE or CREATE FUNCTION statement

8.1.11 Diagnostic statements

Routine (Stored Procedure and UDF)

SQL statement	Function
GET DIAGNOSTICS	Output diagnostic information about a statement
SIGNAL	Report exception in routine
RESIGNAL	Report exception in local exception routine

Table 48: SQL statements for procedures

8.2 Descriptions in alphabetical order

This section describes the syntax and functions of the SQL statements in detail.

8.2.1 Description format

In this section, the SQL statements are described using a uniform syntax. The statements are in alphabetical order. There is one entry per statement, which has the name of the statement as its header.

Structure of an entry

Each entry consists of several parts.
An entry may not include all the parts if some have no meaning for that statement. An entry may also include additional information after the syntax diagram that describes special features or characteristics of the statement involved. The most important parts of each entry are described below.

Statement name - Brief description

A brief description of the function of the statement follows the heading.
This section also describes the prerequisites for successfully executing the statement. In particular, the required access permissions are mentioned.

```
STATEMENT_NAME CLAUSE parameter ...
```

---

parameter

Explanation of the parameter.

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

Examples

This section includes one or more examples illustrating how the statement is used. Most of these are based on the sample database ORDERCUST.

If an example in the manual is accompanied by the symbol on the left, this means that it is present as a component in an instruction file or an ESQL COBOL program in the demonstration database of SESAM/SQL (see the “ Core manual”).

See also

Related statements

8.2.2 SQL statements in routines

In the SQL statements for creating and designing routines below, other SQL statements can also be used:

- CREATE FUNCTION, CREATE PROCEDURE
- CASE, COMPOUND (there also in exception routines), FOR, IF, LOOP, REPEAT, WHILE

Restrictions must be borne in mind for some of these statements.

To make these statements easier to read, the syntax element *routine_sql_statement* is described centrally here for these other SQL statements.

routine_sql_statement ::=

{

case_statement

for_statement

if_statement

iterate_statement

leave_statement

loop_statement

repeat_statement

set_statement

while_statement

return_statement

call_statement

single_row_select_statement

insert_statement

update_searched_statement

delete_searched_statement

merge_statement

open_statement

fetch_statement

update_positioned_statement

delete_positioned_statement

close_statement

get_diagnostics_statement

signal_statement

resignal_statement

}

routine_sql_statement

routine_sql_statement has a maximum length of 32000 characters.

The permitted SQL statements are presented in the following groups:

- Control statements

  case_statement

  CASE statement which conditionally executes further SQL statements, see section “CASE - Execute SQL statements conditionally”.

  for_statement

  FOR statement which executes further SQL statements in a loop, see section “FOR - Execute SQL statements in a loop”.

  if_statement

  IF statement which conditionally executes further SQL statements. see section “IF - Execute SQL statements conditionally”.

  iterate_statement

  ITERATE statement which switches to the next loop pass, see section “ITERATE - Switch to the next loop pass”.

  leave_statement

  LEAVE statement which aborts loops or COMPOUND statements, see section “LEAVE - Terminate a loop or COMPOUND statement”.

  loop_statement

  LOOP statement which executes further SQL statements in a loop, see section “LOOP - Execute SQL statements in a loop”.

  repeat_statement

  REPEAT statement which executes further SQL statements in a loop, see section “REPEAT - Execute SQL statements in a loop”.

  set_statement

  SET statement which assigns a value to a procedure parameter or a local procedure variable, see section “SET - Assign value”.

  while_statement

  WHILE statement which executes further SQL statements in a loop, see section “WHILE - Execute SQL statements in a loop”.

  return_statement

  RETURN statement which returns a return value for the UDF, see section “RETURN - Supply the return value of a User Defined Function (UDF)”. This statement may not be used in procedures.

  call_statement

CALL statement which another procedure calls, see section “CALL - Execute procedure”.

> The DEBUG ROUTINE and LOOP LIMIT pragmas have no effect ahead of a CALL statement in a procedure, see section “CALL - Execute procedure”.
>
> Pragmas for optimization can also be specified in a procedure in the case of a CALL statement. They then have an effect on optimizing the call values.

- SQL statements for querying and updating data without a cursor

  single_row_select_statement

  SELECT statement which reads a single row, see section “SELECT - Read individual rows”.

  insert_statement

  INSERT statement which inserts rows into an existing table, see section “INSERT - Insert rows in table”. This statement may not be used in UDFs.

  update_searched_statement

  UPDATE statement which updates the columns of the rows in a table which satisfy a particular search condition, see section “UPDATE - Update column values”. This statement may not be used in UDFs.

  delete_searched_statement

  DELETE statement which deletes the rows in a table which satisfy a particular search condition, see section “DELETE - Delete rows”. This statement may not be used in UDFs.

  merge_statement

  MERGE statement which, depending on a particular condition, updates rows in a table or inserts rows in a table, see section “MERGE - Insert rows in a table or update column values”. This statement may not be used in UDFs.

- SQL statements for querying and updating data with a cursor:

  These statements are only permitted for a local cursor which is defined in a COMPOUND statement.

  open_statement

  OPEN statement which opens a cursor, see section “OPEN - Open cursor”.

  fetch_statement

  FETCH statement which positions a cursor and possibly reads the current row, see section “FETCH - Position cursor and read row”.

  update_positioned_statement

  UPDATE statement which updates the columns of the row in a table to which the cursor is positioned, see section “UPDATE - Update column values”. This statement may not be used in UDFs.

  delete_positioned_statement

  DELETE statement which deletes the row in a table to which the cursor is positioned, see section “DELETE - Delete rows”. This statement may not be used in UDFs.

  close_statement

  CLOSE statement which closes a cursor. see section “CLOSE - Close cursor” .

- Diagnostic statements

  get_diagnostics_statement

GET DIAGNOSTICS statement for outputting diagnostic information, see section “GET DIAGNOSTICS - Output diagnostic information”.

signal_statement

SIGNAL statement which reports an error in the routine, see section “SIGNAL - Report exception in routine”.

resignal_statement

RESIGNAL statement which reports an error in the exception routine, see section “RESIGNAL - Report exception in local exception routine”.

8.2.3 SQL statement descriptions

- ALLOCATE DESCRIPTOR - Request SQL descriptor area
- ALTER SPACE - Modify space parameters
- ALTER STOGROUP - Modify storage group
- ALTER TABLE - Modify base table
- CALL - Execute procedure
- CASE - Execute SQL statements conditionally
- CLOSE - Close cursor
- COMMIT WORK - Terminate transaction
- COMPOUND - Execute SQL statements in a common context
- CREATE FUNCTION - Create User Defined Function (UDF)
- CREATE INDEX - Create index
- CREATE PROCEDURE - Create procedure
- CREATE SCHEMA - Create schema
- CREATE SPACE - Create space
- CREATE STOGROUP - Create storage group
- CREATE SYSTEM_USER - Create system entry
- CREATE TABLE - Create base table
- CREATE USER - Create authorization identifier
- CREATE VIEW - Create view
- DEALLOCATE DESCRIPTOR - Release SQL descriptor area
- DECLARE CURSOR - Declare cursor
- DELETE - Delete rows
- DESCRIBE - Query data type of input and output values
- DROP FUNCTION - Delete User Defined Function (UDF)
- DROP INDEX - Delete index
- DROP PROCEDURE - Delete procedure
- DROP SCHEMA - Delete schema
- DROP SPACE - Delete space
- DROP STOGROUP - Delete storage group
- DROP SYSTEM_USER - Delete system entry
- DROP TABLE - Delete base table
- DROP USER - Delete authorization identifier
- DROP VIEW - Delete view
- EXECUTE - Execute prepared statement
- EXECUTE IMMEDIATE - Execute dynamic statement
- FETCH - Position cursor and read row
- FOR - Execute SQL statements in a loop

- GET DESCRIPTOR - Read SQL descriptor area
- GET DIAGNOSTICS - Output diagnostic information
- GRANT - Grant privileges
- IF - Execute SQL statements conditionally
- INCLUDE - Insert program text into ESQL programs
- INSERT - Insert rows in table
- ITERATE - Switch to the next loop pass
- LEAVE - Terminate a loop or COMPOUND statement
- LOOP - Execute SQL statements in a loop
- MERGE - Insert rows in a table or update column values
- OPEN - Open cursor
- PERMIT - Specify user identification for SESAM/SQL V1.x
- PREPARE - Prepare dynamic statement
- REORG STATISTICS - Regenerate global statistics
- REPEAT - Execute SQL statements in a loop
- RESIGNAL - Report exception in local exception routine
- RESTORE - Restore cursor
- RETURN - Supply the return value of a User Defined Function (UDF)
- REVOKE - Revoke privileges
- ROLLBACK WORK - Roll back transaction
- SELECT - Read individual rows
- SET - Assign value
- SET CATALOG - Set default database name
- SET DESCRIPTOR - Update SQL descriptor area
- SET SCHEMA - Specify default schema name
- SET SESSION AUTHORIZATION - Set authorization identifier
- SET TRANSACTION - Define transaction attributes
- SIGNAL - Report exception in routine
- STORE - Save cursor position
- UPDATE - Update column values
- WHENEVER - Define error handling
- WHILE - Execute SQL statements in a loop

8.2.3.1 ALLOCATE DESCRIPTOR - Request SQL descriptor area

You use ALLOCATE DESCRIPTOR to create an SQL descriptor area. The descriptor area is used in dynamic statements and cursor descriptions as the interface between the application program and the SQL database.

The structure of an item descriptor and how they are used is described in section “Descriptor area”. ALLOCATE DESCRIPTOR creates the descriptor area but does not define its contents.

ALLOCATE DESCRIPTOR GLOBAL *descriptor* [WITH MAX *number*]

descriptor ::= { *alphanumeric_literal* | :*host_variable* }

number ::= { *integer* | :*host_variable* }

GLOBAL

The descriptor area you create can be used in any compilation unit of the current SQL session.

descriptor

String containing the name of the SQL descriptor area. For *descriptor* you can specify an alphanumeric literal (not in hexadecimal format) or an alphanumeric host variable of the SQL data type CHAR(*n*), where 1 <= *n* <= 18.

The descriptor area name can start and end with one or more blanks. Once leading or trailing blanks have been removed, the remaining string must be an unqualified name (see section “Unqualified names”).

Two descriptor are names are considered identical if, once the blanks have been removed, the remaining unqualified names are identical (see section “Identical unqualified names”).

number

Maximum number of item descriptors in the SQL descriptor area.

For *number* you can specify an integer or a host variable of the SQL data type SMALLINT, where 1 <= *number* <= 1000.

number determines the size of the reserved SQL descriptor area.
If you store longer alphanumeric values in the descriptor area, the space in the descriptor area may be insufficient and an appropriate SQLSTATE is returned. In this case, you must increase the value of *number* (see example).

In UTM applications, the “UTM conversation memory” is used to store SQL descriptor areas. If this memory is insufficient, an error message is issued.

WITH MAX *number* omitted:
20 is the default value for *number*.

Examples

Create an SQL descriptor area for up to 100 item descriptors:

```
ALLOCATE DESCRIPTOR GLOBAL :demo_desc WITH MAX 100
```

Create an SQL descriptor area for 100 item descriptors. The descriptor area should be large enough for the item descriptors to be able to store values of the type CHAR(80).

```
ALLOCATE DESCRIPTOR GLOBAL :demo_desc WITH MAX 200
```

See also

DEALLOCATE DESCRIPTOR, DESCRIBE, GET DESCRIPTOR, SET DESCRIPTOR

8.2.3.2 ALTER SPACE - Modify space parameters

You use ALTER SPACE to modify the parameters of the catalog space or of a user space.

The SPACE view of the INFORMATION_SCHEMA provides you with information on which user spaces have been defined (see chapter “Information schemas”).

The current authorization identifier must own the space. If the storage group is modified, the current authorization identifier must have the special privilege USAGE for the new storage group.

```
ALTER SPACE space

[PCTFREE percent | NO LOG] ...

[USING STOGROUP stogroup]
```

You must specify at least one of the parameters PCTFREE, NO LOG or USING STOGROUP, and each parameter may only be specified once.

space

Name of the space for which parameters are to be modified.

You can qualify the space name with a database name.

The universal user may specify the space name `"CATALOG"` (in double quotes) even if he/she is not the owner of the space. The NO LOG parameter may not be specified here.

PCTFREE *percent*

Free space reservation in the space file expressed as a percentage. *percent* must be an unsigned integer between 0 and 70. The modified free space reservation is not evaluated until the next time the database is reorganized with the REORG utility statement.

PCTFREE *percent* omitted:
The setting for the free space reservation remains unchanged.

NO LOG

Deactivate logging.

Logging is deactivated immediately after the current transaction is terminated with the COMMIT statement.

NO LOG omitted:
The logging setting remains unchanged.

USING STOGROUP *stogroup*

The name of the storage group containing the volumes to be used for the space file. The new storage group is not evaluated until the next time the database is recovered or reorganized with the utility statements RECOVER and REORG respectively.

You can qualify the name of the storage group with a database name. This database name must be the same as the database name of the space.

USING STOGROUP *stogroup* omitted:
The storage group for the space remains unchanged.

Example

This example shows how to modify the free space reservation and the storage group for a space.

ALTER SPACE indexspace PCTFREE 20 USING STOGROUP stogroup3

See also

CREATE SPACE, CREATE STOGROUP

8.2.3.3 ALTER STOGROUP - Modify storage group

You use ALTER STOGROUP to modify the definition of a storage group.

Please note, however, that the definition of a storage group cannot be modified if the storage group is entered in the media table.

The STOGROUPS view of the INFORMATION_SCHEMA provides you with information on which storage groups have been defined (see chapter "Information schemas").

The current authorization identifier must have the special privilege CREATE STOGROUP and must own the storage group.

```
ALTER STOGROUP stogroup { ADD VOLUMES ( volume_name ,...) [ON dev_type ] |
                          DROP VOLUMES ( volume_name ,...) |
                          PUBLIC |
                          TO catid }
```

stogroup

Name of the storage group for which the definition is to be updated. You can qualify the name of the storage group with a database name.

ADD VOLUMES (*volume_name*,...)

Adds new private volumes to the storage group. *volume_name* is an alphanumeric literal indicating the VSN of the volumes. Each VSN can only be specified once for a storage group.

If the storage group previously consisted of private volumes, the new volumes being added must have the same device type.

A storage group can comprise up to 100 volumes.

ON *dev_type*

Alphanumeric literal indicating the device type of the private volumes.
You must specify the device type if the storage group was previously set up on public volumes (PUBLIC).
If the storage group previously consisted of private volumes, you can omit ON *dev_type*. If you do specify ON *dev_type*, you must specify the same device as before.

ON *dev_type* omitted:
The storage group consists of private volumes which all have the same device type as before.

DROP VOLUMES (*volume_name*,...)

Deletes individual private volumes from the definition of the storage group. *volume_name* is an alphanumeric literal indicating the VSN of the volume.

You cannot delete the last volume in a storage group.

PUBLIC

The storage group is set to the default pubset of the BS2000 user ID under which the DBH is running. All private volumes are deleted from the definition of the storage group.

TO *catid*

The new catalog identifier for the volumes is entered in the definition of the storage group. *catid* is an alphanumeric literal indicating the new catalog ID.

In the case of private volumes, the new catalog ID is only used for cataloging the files. The files themselves are still stored on the private volumes. In the case of a pubset, the catalog ID of the pubset on which the storage group is located is changed.

Effect of ALTER STOGROUP

The ALTER STOGROUP statement only modifies the definition of the storage group. It does not affect existing spaces that the volumes in the storage group use.

Volumes deleted from the storage group are not, however, used for new storage space assignments for the spaces. Volumes can be deleted from the storage group explicitly with DROP VOLUME or implicitly by changing from public volumes (PUBLIC) to private volumes or vice versa.
The new definition of the storage group takes effect when files (spaces or backups) are created in the storage group.

Examples

The example below changes the storage group from private volumes to the pubset with the catalog ID O. This is done in two steps.

```
ALTER STOGROUP stogroup4 PUBLIC

ALTER STOGROUP stogroup4 TO 'O'
```

The example below changes the storage group STOGROUP5 from PUBLIC to private volumes. The catalog ID for the files in the storage group remains unchanged.

```
ALTER STOGROUP ordercust.stogroup5

      ADD VOLUMES ('DX017A','DX017B') ON 'D3435'
```

See also

CREATE STOGROUP

8.2.3.4 ALTER TABLE - Modify base table

You use ALTER TABLE to modify an existing base table. You can add columns and their associated indexes, update or delete columns, and add or delete integrity constraints. The value for the reservation of free space which is defined using CREATE SPACE .. PCTFREE is taken into account.

If you are using a CALL DML table, you can only add, update or delete columns and their associated indexes, and update ir delete columns. The restrictions that apply to CALL DML tables are described in the section "Special considerations for CALL DML tables" on "Special considerations for CALL DML tables".

You can also use ALTER TABLE to modify a BLOB table. The restrictions that apply in this case are described in the section "Special considerations for BLOB tables" on "Special considerations for BLOB tables".

You can use the UTILITY MODE pragma to add, change or delete a column in a table (ADD without ADD INDEX, ALTER, DROP). When you activate the pragma (UTILITY MODE ON), the associated statement is performed outside a transaction like a utility statement. This suppresses normal transaction logging for the corresponding statement and thus makes it possible to accelerate performance considerably when modifying large data volumes. However, if an error occurs, it is not possible to roll back the statement. The space containing the base table to be changed is defective and must be repaired (see section "UTILITY MODE pragma").

You cannot use ALTER TABLE to change the table type. You can change the table type by means of the UTILITY statement MIGRATE (see the " SQL Reference Manual Part 2: Utilities").

The BASE_TABLES view in the INFORMATION_SCHEMA provides you with information on which base tables have been defined (see chapter "Information schemas").

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

```
ALTER TABLE table

{

   ADD [COLUMN] column_definition , . . .

       [ADD INDEX index_definition , . . . [USING SPACE space ]] |

   ALTER [COLUMN] column action [ , column action ]  . . .
         [USING FILE exception_file [PASSWORD password ]] |

   DROP [COLUMN] column,... { CASCADE | RESTRICT } |

   ADD [CONSTRAINT integrity_constraint_name ] table_constraint
       [ ,[CONSTRAINT integrity_constraint_name ] table_constraint ] , . . . |

   DROP CONSTRAINT integrity_constraint_name { CASCADE | RESTRICT }

}


action ::=
{

   DROP DEFAULT |

   SET data_type [CALL DML call_dml_default ] |
```

```
    SET default
}

index_definition ::= index ( { column [LENGTH length ] } ,... )

default ::= DEFAULT
{
   alphanumeric_literal
   national_literal
   numeric_literal
   time_literal
   CURRENT_DATE
   CURRENT_TIME(3)
   LOCALTIME(3)
   CURRENT_TIMESTAMP(3)
   LOCALTIMESTAMP(3)
   USER
   SYSTEM_USER
   NULL
   REF( table )
}
```

table

Name of a base table.

ADD [COLUMN] *column_definition*,...

Adds new columns to the base table. The new columns are added after the existing columns. *column_definition* defines the columns, see section "Column definitions".

If c*olumn_definition* contains a default value other than NULL, this default value is inserted into every existing record of the table; this could require some time.

No primary key must be defined in *column_definition*.

An authorization identifier which possesses table privileges for the underlying base table automatically obtains the corresponding privileges for the newly added columns.

If you wish to add a FOR REF column, it does not make sense to use the FOR REF clause for the initial column definition, since this would cause the default value for the REF column to be entered in each row. A more efficient option, particularly with respect to memory requirements, would be to define the column initially with the data type CHAR(237). In this case each row will be assigned the NULL value. The column can then be modified using ALTER COLUMN *column* SET DEFAULT REF(*table*). This does not affect any row entries made up to this point.

ADD INDEX *index_definition*

Definition of one or more indexes for the newly inserted columns.

The rules and referential constraints of the CREATE INDEX statement apply for the index definition, see section “CREATE INDEX - Create index”.

> i The UTILITY MODE ON pragma may not be used together with ADD INDEX.

index

Name of the new index.

column

Name of the column in the base table you want to index. Only columns which are specified in the ADD COLUMN clause may be specified.

LENGTH *length*

Indicates the length up to which the column is to be indexed.

LENGTH *length* omitted:
The column in its entirety in bytes is indexed.

USING SPACE *space*

Name of the space in which the index or indexes is/are to be stored.

The space must already be defined for the database to which the table belongs. The current authorization identifier must own the space.

USING SPACE *space* omitted:
The index is stored in the space for the base table. In the case of a partitioned table, the index is stored in the space for the first partition.

ALTER [COLUMN] *column*

column is the name of the column to be modified.

Modifications of the column are performed in the following order:

- DROP DEFAULT
- SET *data_type*
- SET *default*

You can use one and the same modification type only once for a column.

DROP DEFAULT

Deletes the default (SQL default value) for the column.

The underlying base table must not be a CALL DML table.

SET *data_type*

New data type of the column.

The column whose data type is to be changed must not be column of a primary key. In CALL DML only tables, the column of a primary key can also be specified.

The column may not be used in views, indexes, integrity constraints, and routines.

You can also change the data type of a multiple column. The data type may not be VARCHAR or NVARCHAR. When a data type is changed to a multiple column data type, SESAM/SQL assigns the position number 1 to the first column element. The number of column elements corresponds to the dimension of the new data type.

An atomic column can contain the multiple column data type and vice versa. In this case, SESAM/SQL considers the atomic value to be the same as the value of a multiple column with dimension 1.

The original column data type can only be modified to certain target data types. The table below illustrates which original data types can be combined with which new data types, and which combinations are not, or are only partially, permitted:

Original data type	**New data type**	**New data type**	**New data type**	**New data type**	**New data type**	**New data type**	**New data type**	**New data type**	**New data type**
	INTEGER SMALLINT DECIMAL NUMERIC	REAL DOUBLE PRECISION FLOAT	VARCHAR	CHAR	NVARCHAR	NCHAR	DATE	TIME (3)	TIMESTAMP (3)
INTEGER SMALLINT DECIMAL NUMERIC	yes	yes [1]	no	yes	no	yes [1]	no	no	no
REAL DOUBLE PRECISION FLOAT	yes	yes	no	yes	no	yes	no	no	no
VARCHAR	no	no	yes [2]	no	no	no	no	no	no
CHAR	yes	yes [1]	no	yes	no	yes[4]	yes [1]	yes [1]	yes [1]
NVARCHAR	no	no	no	no	yes [3]	no	no	no	no
NCHAR	yes	yes	no	yes[4]	no	yes	yes	yes	yes
DATE	no	no	no	yes	no	yes	yes	no	no
TIME(3)	no	no	no	yes	no	yes	no	yes	no
TIMESTAMP (3)	no	no	no	yes	no	yes	no	no	yes

Table 49: Permitted and prohibited combinations for data type modifications

[1]A column may be changed to the numeric data types REAL, DOUBLE PRECISION, and FLOAT or to the time data types

DATE, TIME, and TIMESTAMP if the fundamental base table is an SQL table

[2]A column of the data type VARCHAR may only be changed to the new data type with

new_length >= *old_length*. The other data types may not be changed to the data type VARCHAR and vice versa.

3)A column of the data type NVARCHAR may only be changed to the new data type NVARCHAR with

new_length >= *old_length*. The other data types may not be changed to the data type NVARCHAR and vice versa.

4) A code table not equal to _NONE_ must be defined for the database.

SESAM/SQL converts all values in *column* to the new data type row by row. In the case of multiple columns, SESAM/SQL converts the significant values of all variants whose position number is smaller than or equal to the new data type dimension. This means that it is possible that an element's position may change within the multiple column: If the result of converting a column is the NULL value, all following elements whose position number is smaller than or equal to the new data type
dimension are shifted to the left and the NULL value is appended after them.
The same rules apply (except for CHAR <-> NCHAR) when converting a column value as when converting a value by means of the CAST expression (see section "Rules for converting a value to a different data type").
When a column value is converted from CHAR to NCHAR and vice versa, the same rules apply as for the transliteration of a value by the TRANSLATE expression,
CATALOG_DEFAULT being used in the USING clause (see the section
"TRANSLATE() - Transliterate / transcode string").
These rules also apply for the conversion of the column element value of a multiple column.

If a conversion error occurs, an error message or alert is issued.

The rounding of a value does not represent a conversion error.

Example

A column of NUMERIC data type is changed to the data type INTEGER.
SESAM/SQL converts the original column value 450.25 to 450 without issuing an alert.

When conversion errors occur, SESAM/SQL differentiates between truncated strings, truncated column elements and non-convertible values:

- truncated strings
  A column with CHAR or NCHAR data type is to be changed to a new CHAR or NCHAR data type respectively with shorter length. Affected column values which are longer than the new value are truncated to the length of the new data type. If characters which are not spaces are removed, SESAM/SQL issues an alert.

  Example
  The value 'cust_service' in a column which is of alphanumeric data type CHAR(12) or national data type NCHAR(12) is to be converted to data type CHAR(6) or NCHAR(6) respectively. The original column value is replaced by the value 'cust_s'. SESAM/SQL issues an alert.
- truncated column elements
  A multiple column contains at least one column element whose position number is greater than the dimension of the new data type and which contains a significant value not equal to NULL.

Example
A multiple column of alphanumeric data type (7) CHAR (20) or national data type (7) NCHAR (20) is to be converted to the data type (5) CHAR (20) or (5) NCHAR (20) respectively. In some table rows, all 7 elements of the multiple row contain an alphanumeric value.

- Non-convertible values
  For certain column values, a change of data type results in the loss of values with an error message (data exception).

  Examples

  - The value of an original column of numeric data type is too large for the target numeric data type.

    Example
    The value 9999 in an INTEGER column is to be converted to the data type NUMERIC(2,0).

  - A column of alphanumeric data type CHAR or national data type NCHAR is converted to a numeric data type. The original value of the column cannot be represented as numeric value.

    Example
    The value 'Otto' in a column with alphanumeric data type CHAR(4)or national data type NCHAR(4) is to be converted to the data type INTEGER.

  - The length of the value in an originally numeric column or in a column with a time data type is too large for the alphanumeric target data type CHAR or the national target data type NCHAR respectively.

    Example
    The value 9999 in a column of data type INTEGER is to be converted to the alphanumeric data type CHAR(2) or national data type NCHAR(2) respectively.

If the column definition for *column* contains a default, the new data type may not contain a dimensional specification.
If the specified SQL default value is an alphanumeric, national, numeric or time literal, it is converted to the new data type. The conversion must not result in a
conversion error. If the specified SQL default value is a time function, a iteral or the NULL value, it is not changed.
After conversion, the SQL default value for the new data type must conform to the assignment rules for default values (see section “Default values for table columns”).

CALL DML *call_dml_default*

Changes the non-significant value of column in a CALL DML table. May only be specified for CALL DML tables
call_dml_default corresponds to the non-significant attribut value in SESAM/SQLVersion 1.x.
You specify *call_dml_default* as an alphanumeric literal.

CALL DML *call_dml_default* not specified:
If the data type modification applies to the column in a CALL DML/SQL table, *column* retains the non-significant attribute value which was assigned to it during column definition.
If the data type modification applies to a column of a CALL DML only table, i.e. a table with “old“ attribute formats from SESAM versions < V13.1, *column* is assigned the following non-significant attribute value:

- space if the *column* data type is alphanumeric
- digit 0 if the *column* data type is numeric

SET *default*

Defines a new SQL default value for the column.

The underlying base table must not be a CALL DML table.

column cannot be a multiple column.

default must conform to the assignment rules for default values (see section “Default values for table columns”).

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

USING FILE *exception_file* [PASSWORD *password*]

Defines the name of the exception file. *exception_file* must be specified as an alphanumeric literal.
SESAM/SQL creates or uses the exception file only if a column conversion performed using SET *data_type* results in one or more conversion errors (see "ALTER TABLE - Modify base table").

If an exception file is specified, a statement which results in a conversion error is continued. SESAM/SQL issues an alert and replaces the original column values by new values in the affected base table:

- truncated strings are replaced by the corresponding truncated value.
- non-convertible values are replaced by the NULL value.
- column items in a multiple column whose position number is larger than the new data type dimension are truncated.

SESAM/SQL logs the original column values and truncated column elements together with the associated alert or error message in the exception file.

Even when UTILITY MODE is switched ON, a statement which results in a conversion error is not interrupted. The space which contains the base table to be updated remains intact.

For a detailed description of the exception file and its contents, see section “Exception file of SQL statement ALTER TABLE”.

PASSWORD *password*

BS2000 password for the error file. You must specify *password* as an alphanumeric literal.

password can be specified in several different ways:

- 'C'' *string* ''' *string* contains four printable characters.
- 'X'' *hex_string* ''' *hex_string* contains eight hexadecimal characters.
- ' *n* ' *n* is an integer from - 2147483648 through + 2147483647.

USING FILE *exception_file* not specified:
If a column conversion performed using SET *data_type* results in a conversion error, SESAM/SQL does not log the affected column values or column elements in an exception file.
Strings are truncated to the length of the new data type and SESAM/SQL issues an alert.
If conversion errors occur because values cannot be converted or column elements have to be truncated, SESAM/SQL aborts the associated statement and issues an error message.

DROP [COLUMN] *column*,... {CASCADE, RESTRICT}

Deletes one or more columns and associated indices in the base table.
column is the name of the column to be deleted. You can only specify each column name once.

No primary key may be defined for *column*.

You must not specify all columns in the base table.

Deleting a column revokes the column privileges UPDATE and FOREIGN KEY... REFERENCES for this column from the current authorization key. If these privileges have been passed on, then the passed on privileges are also withdrawn.
In addition, deleting the column also deletes all views where *column* was used in the view definition as well as all views whose definitions contain the name of such a “higher level“ view.

The arrangement of the remaining columns in a table can change: if deleting a column results in a gap, all following columns are shifted to the left.

CASCADE

Deletes the specified column(s) and associated indices.
The integrity constraints of other tables or columns which use *column* are also deleted. All routines which reference this column directly or indirectly are deleted.

You cannot use the UTILITY MODE pragma. If you activate the UTILITY MODE, an error message is output and the statement is aborted.

RESTRICT

Deletion of a column is restricted:
The column cannot be deleted if it is used in a view definition or a routine. You may only define an index for the column to be deleted if none of the remaining columns in the base table is named in the affected index definition. The same applies to the integrity constraints.

The UTILITY MODE pragma can be activated when no index is defined for the column.

ADD CONSTRAINT clause

Adds integrity constraints to the base table.

> Adding multiple integrity restraints in an ALTER TABLE statement is more efficient than adding one integrity restraint in each of a correspondingly large number of ALTER TABLE statements.

CONSTRAINT *integrity_constraint_name*

Assigns a name to the integrity constraint. You can qualify the name of the integrity constraint with a database and schema name. The database and schema name must be the same as the database and schema name of the base table.

CONSTRAINT *integrity_constraint_name* omitted:
The integrity constraint is assigned a name according to the following pattern:

UN *integrity_constraint_number*
FK *integrity_constraint_number*
CH *integrity_constraint_number*
where UN stands for UNIQUE, FK for FOREIGN KEY and CH for CHECK.
integrity_constraint_number is a 16-digit number.

table_constraint

Specifies an integrity constraint for the table. *table_constraint* cannot define a primary key constraint.

DROP CONSTRAINT *integrity_constraint_number* {CASCADE, RESTRICT}

Deletes the integrity constraint *integrity_constraint_name*.
integrity_constraint_number may not name a primary key constraint.

CASCADE

If *integrity_constraint_name* is a uniqueness constraint, and if the referential constraint of another table references the column(s) for which *integrity_constraint_name* was defined, the referential constraint of the other table is also implicitly deleted.

RESTRICT

You must not delete a uniqueness constraint on a column if a referential constraint on another table references this column(s).

Special considerations for CALL DML tables

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

- Only the ADD [COLUMN], DROP [COLUMN] and ALTER [COLUMN] clause are permitted with SET *data_type*.
- A newly inserted column must include a CALL DML clause.
- Only the data types CHAR, NUMERIC, DECIMAL, INTEGER and SMALLINT are permitted.
- No integrity constraint or default value (DEFAULT) can be defined for the column.
- The column name must be different to the integrity constraint name of the table constraint since this name is used as the name of the compound primary key.
- A column's data type in a CALL DML table may only be changed to the data type of a CALL DML/SQL table. In particular, a CALL DML table's data type must not be changed to an "old attribute format", i.e. to an attribute format of SESAM version <13.1.
- An "old attribute format" in a CALL DML only table can be changed to the following data types:
  - CHAR with *new_length* >= *old_length*
  - NUMERIC with *old_fraction=new_fraction*
  - DECIMAL with *old_fraction=new_fraction*
  - INTEGER
  - SMALLINT
- You can assign a new non-significant attribute value for columns in a CALL DML table. You may not change the symbolic attribute name.
- If a data type modification results in a value in a CALL DML column receiving the nonsignificant attribute value, the value of the column in question is considered to be nonconvertible. If no exception file was specified, SESAM /SQL issues an error message and aborts the statement. If an exception file is specified, SESAM/SQL reacts as in the case of non-convertible values in an SQL table (see "ALTER TABLE - Modify base table").
- You can neither use the ALTER [COLUMN] clause nor the DROP [COLUMN] clause to change the table type. Even if the columns in a CALL DML only table have been changed or deleted so that none of the columns contains an "old attribute format", the "CALL DML only" table type remains unchanged. You can change the table type by means of the UTILITY statement MIGRATE (see the " SQL Reference Manual Part 2: Utilities").

Converting "old" attributes in a CALL DML only table

The attribute of a CALL DML only table has no explicit type: the type is simply specified by the way the table is saved. The user must interpret the values correctly.

You cannot use ALTER COLUMN to change the type, but only to transfer it to the specified type. When you do this, values of the corresponding type are transferred and those of different types are rejected (SQLSTATE 22SA5). You should therefore only specify the appropriate type. Conversion to another type is only possible if you use a second ALTER COLUMN and specify the new data type.

For example, a binary value can only be changed to INTEGER, SMALLINT. After a second ALTER COLUMN you can also convert it to NUMERIC, DECIMAL and CHAR.

ALTER COLUMN reads each value and prepares it in accordance with its definition in the CALL DML table. Alignment, fill bytes etc. are not taken into account. However, no conversion is performed. After that, a check is performed to determine whether the read value corresponds to the specified format or not.

Since the attributes of the CALL DML only table also contain values of different types, it is advisable to always specify USING FILE *exception_file* for "old" attributes when using ALTER COLUMN. All inappropriate values are then entered in the exception file.

If no exception file is present, ALTER COLUMN aborts when the first inappropriate value is encountered.

Special considerations for BLOB tables

You can also use ALTER TABLE to modify a BLOB table. However, certain types of changes may result in the BLOB table becoming inaccessible to CLI calls. The permitted changes and their effects are described below:

- Inserting a new column in a BLOB table does not affect the execution of CLI calls.
- Additional integrity constraints on BLOB tables can be defined using the ADD CONSTRAINT clause without any negative repercussions.
- If one of the columns OBJ_NR, SLICE_NR, SLICE_VAL or OBJ_REF is deleted or its type is changed, it will no longer be possible to process BLOB values in CLI functions.

Exception file of SQL statement ALTER TABLE

When you modify a column (ALTER COLUMN), you can specify the name of an exception file. If necessary, you can protect the exception file using a BS2000 password. The exception file is used to store column values for which conversion errors resulted in data loss because of a change of data type.
If you have specified an exception file and conversion errors occur during the modification of the data type, SESAM /SQL sets up the exception file as a SAM file under the DBH user ID if this does not yet exist.

If the exception file is not to be stored on the DBH user ID, preparations must have been made, see section “Database files and job variables on foreign user IDs” in the “ Core manual”.

If an exception file is specified, statements which result in a conversion error are not aborted. SESAM/SQL issues an alert and replaces the original column value by a new value in the affected base table. Depending on the error type, the value is replaced by a truncated value or the NULL value.

SESAM/SQL logs the original column values together with the associated error message or alert in the exception file. If an exception file exists, its contents are not overwritten. SESAM/SQL appends the new entries to the existing entries.
The exception file is not subject to transaction logging. It remains intact, even if the transaction which SESAM/SQL uses to write entries to the exception file is implicitly or explicitly rolled back.
You can display the contents of the exception file using the SHOW-FILE command.

Contents of the exception file

The exception file contains an entry for each logged column value. The entry consists of the corresponding SQL status code and the components which identify the column value within the associated base table.

---

entry ::=

row_id
column_name [*posno*]
sqlstate
column_value

row_id ::= { *primary_key* | *row_counter* }

---

row_id

Identifies the table rows which contains the *column_value*.
In tables with primary keys, *row_id* is the primary key value which uniquely identifies the corresponding row. Its representation in the error file corresponds to the representation of *column_value* (see under the appropriate information). The same applies to the compound keys.

In tables without primary key, *row_id* is the counter of the row containing *column_value*. SESAM/SQL numbers all table rows sequentially. The first row in the table contains the value 1 as *row_counter*.
row_counter is an unsigned integer.

column_name

Name of the column containing to the *column_value*. In multiple columns, *column_name* also contains the position number, in unsigned integer format, of the affected column element. The first element of the multiple column has the position number 1.

sqlstate

SQLSTATE of the associated error message or alert.

column_value

Original column value for which the ALTER TABLE statement resulted in a conversion error.
Depending on the data type of the associated, *column_value* is represented in the following ways in the exception file:

Data type of column containing the original value	**Representation of *column_value* in the exception file**
Data type of a CALL DML table column of SESAM up to V13.1	string with a maximum length of 54 characters
CHAR VARCHAR	string with a maximum length of 54 characters
NCHAR NVARCHAR	string with a maximum length of 27 code units
INTEGER, SMALLINT, NUMERIC, DECIMAL,	corresponding numeric literal (integer or fixed-point number)
FLOAT, REAL, DOUBLE PRECISION	corresponding numeric literal (floating point number)
DATE	Date time literal
TIME	Time time literal
TIMESTAMP	Timestamp time literal

Table 50: Representation of *column_value* data types

Strings are represented without surrounding single quotes in the exception file.

If the original value is a string which contains double quotes, these are represented as single quotes in the exception file.

Example

The following example shows an exception file which contains the original column values of the base table SERVICE.

The base table SERVICE has the following structure:

```
SQL CREATE TABLE service
   (service_num  INTEGER CONSTRAINT service_num_primary PRIMARY KEY,
    order_num    INTEGER CONSTRAINT s_order_num_notnull NOT NULL,
    service_date DATE, ...)
```

Its entries are the result of conversion errors which were caused by the following statements:

```
ALTER TABLE service ALTER COLUMN service_price SET NUMERIC(5,2)

USING FILE 'ERR.SERVICE'
```

Excerpt from the exception file ERR.SERVICE:

```
row_id           column_name              sqlstate  column_value

2               SERVICE_PRICE                22SA4          1500
3               SERVICE_PRICE                22SA4          1500
4               SERVICE_PRICE                22SA4          1200
5               SERVICE_PRICE                22SA4          1200
.
.
11              SERVICE_PRICE                22SA4          1200
```

When converting SERVICE_PRICE from NUMERIC (5,0) to NUMERIC(5,2), any rows containing the specified primary key will be ignored.

Examples

The following examples demonstrate how to modify various properties of the CUSTOMERS and ORDERS tables:

Add two new columns, CUST_TEL and CUST_INFO, to the CUSTOMERS table.

```
ALTER TABLE customers

            ADD COLUMN cust_tel CHARACTER(25), cust_info CHARACTER(50)
```

In the CUSTOMERS table, change the data type of the CUST_NUM column.
The original data type was NUMERIC, the new data type is INTEGER.

```
ALTER TABLE customers

            ALTER COLUMN cust_num SET INTEGER
```

Delete the CUST_INFO column from the CUSTOMERS table.
This is possible only if the CUST_INFO column is not used in any view definition. An index or an integrity constraint can then only be defined for the CUST_INFO column if none of the remaining columns of the base table is specified in the definition.

```
ALTER TABLE customers

            DROP COLUMN cust_info RESTRICT
```

Add a uniqueness constraint on the CUST_NUM column of the CUSTOMERS table.

```
ALTER TABLE customers

            ADD CONSTRAINT cust_num_unique UNIQUE(cust_num)
```

Delete the referential constraint between the CUST_NUM column of the ORDERS table and the CUST_NUM column of the CUSTOMERS table. You can look up the names of the integrity constraints used in the TABLE_CONSTRAINTS, REFERENTIAL_CONSTRAINTS and CHECK_CONSTRAINTS views of the INFORMATION-SCHEMA.

```
ALTER TABLE orders

            DROP CONSTRAINT o_cust_num_ref_customers CASCADE
```

See also

CREATE TABLE

8.2.3.5 CALL - Execute procedure

CALL executes a procedure CALL can also be used in a routine to execute another procedure (nested calls of routines).

The CALL statement is a non-atomic SQL statement, as non-atomic statements can be contained in the called procedure.

Procedures and their use in SESAM/SQL are described in detail in chapter “Routines”.

You can ascertain which routines are defined and which routines use each other in the views for routines of the INFORMATION_SCHEMA (see chapter “Information schemas”).

When a procedure expects input parameters, the corresponding values (arguments) must be transferred to the procedure in the CALL statement.

Output values of procedures which are called outside a routine are stored in corresponding host variables or in the SQL descriptor area. Output values of procedures which are called in a higher-level routine are entered in output parameters or in local variables of the higherranking procedure.

The DEBUG ROUTINE, DEBUG VALUE, and LOOP LIMIT pragmas can also be used. See section “Pragmas and annotations”.
They are interpreted only when they are located ahead of a CALL statement which is called externally (in other words from an application), and they then propagate their effect to all directly or indirectly contained CALL statements and User Defined Functions. They have no effect ahead of a CALL statement in a procedure.

Pragmas for optimization can also be specified in a procedure in the case of a CALL statement. They then have an effect on optimizing the call values.

In order to execute a procedure, the current authorization identifier requires the EXECUTE privilege for the procedure to be executed, but not the privileges which are required to execute the DML statements contained in the procedure. In addition, the SELECT privileges for the tables which are addressed in the routine's call parameters by means of subqueries are required.

`CALL` *procedure arguments*

procedure `::=` *routine*

arguments `::=` `([` *expression* `[{ ,` *expression* `}...]])`

---

procedure

Name of the procedure to be executed. You can qualify the procedure name with a database and schema name.

`([` *expression* `[{ ,` *expression* `}...]])`

List of arguments. The number of arguments must be the same as the number of parameters in the procedure definition. The order of the arguments must correspond to that of the parameters. If no parameter is defined for the procedure, the list consists only of the parentheses.

If the nth parameter is of the type IN or INOUT, it is assigned the value of the nth argument before the procedure is executed.

If the nth parameter is of the type OUT or INOUT, the following applies:

- If the CALL statement is static, the nth argument must be a host variable (possibly with indicator variable). The same host variable may not be used as an argument for more than one parameter of the type OUT or INOUT.
- If the CALL statement is dynamic, the nth argument must be a placeholder ("?").

After the procedure has been executed, the values for the parameters of the type OUT or INOUT are transferred to the corresponding host variables or to an SQL descriptor area.

The data type of the nth argument must be compatible with the data type of the nth parameter. For input parameters, see the information in section “Supplying input parameters for routines”. For output parameters, see section “Entering values in a procedure parameter (output) or local variable”.

When, in the case of a static SQL statement, a parameter is specified as a host variable, while pre-assembling (without database contact) SESAM/SQL assumes that a parameter of the type IN or INOUT is concerned and transfers this value to the DBH. Even if a pure output parameter is concerned, the value must therefore either be correctly initialized according to the data type or the host variable will be assigned an indicator variable which must then be supplied with the value -1.

CALL and transaction management

CALL introduces an SQL transaction for procedures which are called outside a routine when no transaction is open. As a procedures contains only DML statements, CALL initiates an SQL transaction for data manipulation.

The procedure statements run at the same isolation level and in the same transaction mode as the CALL statement (see section “SET TRANSACTION - Define transaction attributes”).

When the transaction mode READ ONLY is set, the procedure may not contain any SQL statements for updating data.

CALL and time functions

If the time functions CURRENT_DATE, CURRENT_TIME(3), LOCALTIME(3), CURRENT_TIMESTAMP(3) and LOCALTIMESTAMP(3) are included in a statement multiple times, they are evaluated simultaneously, see section “Time functions”. This information also applies for procedure statements. However, this does not mean that the time functions of all statements of a procedure run are evaluated simultaneously:

- The time functions of the CALL statement are evaluated simultaneously if they occur as a value in input parameters.
- The time functions of each procedure statement are evaluated simultaneously and separately. Different procedure statements consequently generally return different time values.
- The time functions of the COMPOUND statement are evaluated simultaneously when they occur as a default value in variable definitions.
- The time functions of an IF statement are evaluated simultaneously for all search conditions, both in the IF and in the ELSIF branch. However, the time functions of the procedure statements in the THEN and ELSE branches of the IF statement are once again evaluated simultaneously and separately.
- The time functions in cursor descriptions of local cursors are evaluated simultaneously in the OPEN statement for the cursor.

Example

The `GetCurrentYear` procedure (see "CREATE PROCEDURE - Create procedure") is called.

```
CALL ProcSchema.GetCurrentYear (OUT myvar)
```

See also

CREATE PROCEDURE, DROP PROCEDURE

8.2.3.6 CASE - Execute SQL statements conditionally

The CASE statement executes SQL statements depending on specific values (unqualified CASE statement) or conditions (CASE statement with search condition).

The CASE statement may only be specified in a routine, i.e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. Routines and their use in SESAM/SQL are described in detail in chapter “Routines”.

The CASE statement is a non-atomic SQL statement, i.e. further (atomic or non-atomic) SQL statements can occur in it.

If the *search_condition* or an *expression* of a CASE statement corresponds to a table, the authorization identifier which creates the routine using CREATE PROCEDURE or CREATE FUNCTION must have the SELECT privilege for this table.

Execution information

The CASE statement is a non-atomic statement:

- If the CASE statement is part of a COMPOUND statement, the rules described there apply, in particular the exception routines defined there.
- If the CASE statement is **not** part of a COMPOUND statement and one of the SQL statements reports an SQLSTATE, it is possible that only the updates of this SQL statement will be undone. The CASE statement and the routine in which it is contained are aborted. The SQL statement in which the routine was used returns the SQLSTATE concerned.

See also

CREATE PROCEDURE, CREATE FUNCTION

Format of the simple CASE statement

```
CASE expressionx
   WHEN expression1 , ... THEN routine_sql_statement; [ routine_sql_statement; ] ...

   ...

   [ ELSE routine_sql_statement; [ routine_sql_statement; ] ... ]

END CASE
```

expression

Expression that returns an alphanumeric, national, numeric or time value when evaluated.
It cannot be a multiple value with a dimension greater than 1.

expression may not include host variables.
A column may only be specified in a subquery.

The values of *expressionx* and *expression1, ...* must have compatible data types (see section “Compatibility between data types”).

routine_sql_statement

SQL statement which is to be executed in the THEN or ELSE clause depending on the values of *expressionx* and *expression1, ...* .
An SQL statement is concluded with a ";" (semicolon).
Multiple SQL statements can be specified one after the other. They are executed in the order specified.
No privileges are checked before an SQL statement is executed.
An SQL statement in a routine may access the parameters of the routine and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

The syntax and meaning of *routine_sql_statement* are described centrally in section "SQL statements in routines". The SQL statements named there may not be used.

Execution information

expressionx of the CASE statement is calculated.

The WHEN clauses are evaluated from top to bottom.

The expressions *expression1,...* of the WHEN clauses are calculated from left to right.

When a value of an expression calculated in this way corresponds to the value of *expressionx*, the associated THEN branch is executed, and the CASE statement is subsequently terminated.

If none of the calculated values corresponds to *expressionx* but an ELSE branch exists, the ELSE branch of the CASE statement is executed, and the CASE statement is subsequently terminated.

CASE statement is terminated with SQLSTATE '20000'.

Example

Simple CASE statement for calculating the public holiday allowance in wages.

```
CASE MOD(JULIAN_DAY_OF_DATE(CURRENT_DATE),7)
  WHEN 0,1,2,3,4 /* today is a normal workday */
       THEN UPDATE pay_scale SET pay = time_pay;
  WHEN 5 /* today is Saturday, 25% supplement */
       THEN UPDATE pay_scale SET pay = time_pay * 1.25;
  WHEN 6 /* today is Sunday, 50% supplement */
       THEN UPDATE pay_scale SET pay = time_pay * 1.50;
END CASE
```

The CASE statement above could also be replaced by an UPDATE statement with an appropriate *case_expression*.

```
UPDATE pay-scale
   SET pay = time_pay * CASE MOD(JULIAN_DAY_OF_DATE(CURRENT_DATE),7)
                          WHEN 0,1,2,3,4 /* today is a normal workday */
                               THEN 1.00
                          WHEN 5 /* today is Saturday, 25% supplement */
                               THEN 1.25
                          WHEN 6 /* today is Sunday, 50% supplement */
                               THEN 1.50
                          END
```

Format of the CASE statement with search condition

CASE WHEN *search_condition* THEN THEN *routine_sql_statement;* [*routine_sql_statement;*] ...

...

[ELSE *routine_sql_statement;* [*routine_sql_statement;*] ...]

END CASE

search_condition

Search condition that returns a truth value when evaluated
If the result of the search condition is "unknown", no SQL statement is executed in the THEN clause.

routine_sql_statement

See "Format of the simple CASE statement".

Execution information

The WHEN clauses are evaluated from top to bottom.

The *search_condition* of the WHEN clause is evaluated.

When such a calculated search condition returns the truth value TRUE, the associated THEN branch is executed, and the CASE statement is subsequently terminated.

If none of the calculated search conditions returns the truth value TRUE but an ELSE branch exists, the ELSE branch of the CASE statement is executed, and the CASE statement is subsequently terminated.

If none of the calculated search conditions returns the truth value TRUE and no ELSE branch exists, the CASE statement is terminated with SQLSTATE '20000'.

Example

CASE statement with search condition.

```
CASE
   WHEN (EXISTS(select * from T1 where cola = 17))
        THEN update T1 set colb = colb * 1.05;
   WHEN (EXISTS(select * from T2 where colx = 27))
        THEN insert into T2 (pk, coly) values (*, 423);
END CASE
```

8.2.3.7 CLOSE - Close cursor

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

The cursor description is retained. The current cursor position can be saved before closing with STORE (not applicable for local cursors in procedures).

You can close a cursor any number of times and, if desired, open it again with new variable values.

CLOSE *cursor*

---

cursor

Name of the cursor to be closed.

Example

Close the cursor CUR_CONTACTS.

CLOSE cur_contacts

See also

DECLARE CURSOR, FETCH, OPEN, RESTORE, STORE

8.2.3.8 COMMIT WORK - Terminate transaction

You use COMMIT WORK to terminate an SQL transaction and commit the modifications made to the database during that transaction. The updated SQL data is then available to all other transactions.

A new transaction is started by the first SQL statement after COMMIT WORK that initiates an SQL transaction.

```
COMMIT [WORK]
```

SQL transaction

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

Transaction under openUTM

You cannot use the COMMIT WORK statement if you are working with openUTM. In this case, transaction management is performed using only UTM language resources. openUTM ensures the synchronization of SESAM/SQL and UTM transactions. A UTM transaction ends when the next synchronization point is set.

Initiating a transaction

The following SQL statements do not initiate a transaction:

- DECLARE CURSOR (not executable)
- PERMIT
- SET CATALOG
- SET SCHEMA
- SET SESSION AUTHORIZATION
- SET TRANSACTION
- WHENEVER (not executable)
- Utility statements

The statements EXECUTE and EXECUTE IMMEDIATE only initiate an SQL transaction if the dynamic statement to be executed initiates a transaction.

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

Statements within a transaction

The following statements cannot be executed within a transaction:

- SET SESSION AUTHORIZATION
- SET TRANSACTION
- Utility statements

You may not execute or prepare an SQL statement that manipulates data (query, update) in a transaction in which an SQL statement for defining or managing schemas, storage structures or user entries is executed.

CALL DML transaction

The SQL statement COMMIT WORK is not permitted within a CALL DML transaction (see section “SQL statements in CALL DML transactions”).

Effects of COMMIT WORK

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

Effect on subsequent transactions

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

Repercussions on cursors (not applicable for local cursors in procedures)

COMMIT WORK closes all the cursors opened in the transaction. If you want to save the cursor position beyond the end of the transaction, you can save the position with the STORE statement and restore it later with RESTORE.

It is possible to define a cursor using the WITH HOLD clause. A cursor defined in this way will remain open even after COMMIT WORK is executed (successfully). It can then be positioned in a follow-up transaction using FETCH.

Effect on defaults

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

Behavior of SESAM/SQL in the event of an error

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

See also

ROLLBACK WORK, SET TRANSACTION

8.2.3.9 COMPOUND - Execute SQL statements in a common context

The COMPOUND statement executes other SQL statements of a routine in a common context. Common local data (variables and exception names), common local cursors, and common local exception routines apply for these SQL statements.

> i The spelling "COMPOUND" (uppercase) was chosen merely by analogy to the existing notation in the SQL statements for this SQL statement. There is no SQL keyword "COMPOUND".

The COMPOUND statement may only be specified in a routine, i.e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. It is then the only statement in the routine. Routines and their use in SESAM/SQL are described in detail in chapter “Routines”.

[*label* :]

BEGIN [[NOT] ATOMIC]

[*local_data*]

[*local_cursor*]

[*local_exception_handling*]

[*routine_sql_statement;* [*routine_sql_statement;*]...]

END [*label*]

---

label

The label in front of the COMPOUND statement (start label) indicates the start of the COMPOUND statement. It may not be identical to another label in the COMPOUND statement.

The start label need only be specified when the COMPOUND statement is to be terminated by means of a LEAVE statement

The label at the end of the COMPOUND statement (end label) indicates the end of the COMPOUND statement. If the end label is specified, the start label must also be specified. Both labels must be identical.

[NOT] ATOMIC

Determines whether the COMPOUND statement is atomic or non-atomic.This specification influences local exception handling, see "COMPOUND - Execute SQL statements in a common context".If nothing is specified, NOT ATOMIC applies.

local_data

Defines local variables and exception names for the COMPOUND statement, see section “Local data”.

i

The SQL statements of the COMPOUND statement can only access local data which is defined in the COMPOUND statement. They cannot access host variables.

local_cursor

Defines local cursors for the COMPOUND statement, see section “Local cursors”.

i The SQL statements of the COMPOUND statement can only access cursors which are defined in the COMPOUND statement.

local_exception_handling

Defines local exception routines for the COMPOUND statement, see section “Local exception routines”.

SQLSTATEs of classes 40xxx and SQLSTATEs from class '50xxx' cannot be handled in the local exception routines. When such an SQLSTATE occurs, the routine is immediately aborted. In the case of an SQLSTATE of class '40xxx', the entire transaction is also reset.

SQLSTATEs which are not specified explicitly in the exception routines in the form of a class or explicitly ("unspecified SQLSTATEs") are not handled by any exception routine. The same applies when no local exception handling is defined. In these cases SESAM/SQL automatically performs exception handling as follows:

- SQLSTATEs of classes '01xxx’ (warning) or '02xxx’ (no data) are ignored, i.e. the routine is continued as when the SQL statement is executed successfully (SQLSTATE = '00000').
- The following actions are performed for SQLSTATEs which are not in class '01xxx', '02xxx' or '40xxx':
  - Open local cursors are closed.
  - When ATOMIC is specified in the COMPOUND statement, all updates made in the context of the COMPOUND statement are undone.
  - When NOT ATOMIC (default value) is specified in the COMPOUND statement, only updates made in the context of the errored SQL statement are undone.
  - The COMPOUND statement and with it the routine are aborted. The SQL statement in which the routine was used returns the SQLSTATE concerned.

routine_sql_statement

SQL statement which is to be executed in the COMPOUND statement. An SQL statement is concluded with a ";" (semicolon). Multiple SQL statements can be specified one after the other. They are executed in the order specified.No (further) COMPOUND statement may be specified in the COMPOUND statement. In other words, no nested COMPOUND statements are permitted (exception: local exception routines, see "COMPOUND - Execute SQL statements in a common context".) No privileges are checked before an SQL statement is executed. An SQL statement in a routine may access the parameters of the routine and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

The syntax and meaning of *routine_sql_statement* are described centrally in section “SQL statements in routines”. The SQL statements named there may not be used.

Example

You will find examples in chapter “Routines” and in the demonstration database of SESAM/SQL (see the “ Core manual”).

See also

CREATE PROCEDURE, CREATE FUNCTION, CALL, DROP PROCEDURE, DROP FUNCTION, CASE, FOR, IF, ITERATE, LEAVE, LOOP, REPEAT, SET, WHILE, RETURN, GET DIAGNOSTICS, SIGNAL, RESIGNAL, SELECT, INSERT, UPDATE, DELETE, MERGE, OPEN, FETCH, UPDATE, DELETE, CLOSE

Local data

Local data comprises variables or exception names which can only be addressed in the COMPOUND statement.

A data type and, if required, a default value is defined for variables. They have no indicator variable. They can be used in local cursor definitions, local exception routines, and the SQL statements of the COMPOUND statement.

> i *Recommendation* The names of and local variables should differ from column names (e.g. by assigning a prefix such as `par_`).

To facilitate understanding, exception names define a name for an exception (without specifying an associated SQLSTATE) or a name for an SQLSTATE. They can be used in local exception routines, see "COMPOUND - Execute SQL statements in a common context".

---

local_data `::=` `DECLARE` *declaration* `;` `[DECLARE` *declaration* `;]` `...`

declaration `::=`

`{`

local_variable `[,` *local_variable* `],` `...` *data_type* `[` *default* `]` `|`

error_name `CONDITION` `[FOR` *sqlstate* `];`

`}`

sqlstate `::=` `SQLSTATE` `[VALUE]` *alphanumeric_literal*

Multiple variables of the same data type with the same SQL default value can be specified one after another, separated by "," (comma). The definition of a local date is concluded with “;” (semicilon). Multiple definitions can be specified one after the other.

local_variable

Name of the local variable. The names of all local variables must differ from each other, from the local exception names, and from the names of the routine’s parameters.

data_type

Data type of the local variable. Only unqualified local variables exist. *dimension* may not be specified.

default

Specifies the SQL default value for the local variable. The assignment rules for default values apply, see section “Default values for table columns”).

exception_name

Name of an exception or SQLSTATE.

All exception names of all local variables must differ from each other, from the local variables, and from the names of the routine’s parameters.

FOR *sqlstate*

SQLSTATE (alphanumeric literal with the length 5) which is named by *exception_name*. The restrictions for the set of SQLSTATEs must be borne in mind, see “Local exception routines”.

FOR *sqlstate* omitted

Local exception names without FOR clause can be triggered only by a SIGNAL or RESIGNAL statement. They are mapped to the SQLSTATE '45000' (unspecified user exception) and reported to the application program. *exception_name* appears as an insert in the error message.

Example

Definition of local variables.

```
DECLARE a,b,c SMALLINT DEFAULT 0;
```

```
DECLARE mytim TIME(3) DEFAULT CURRENT_TIME;
```

Definition of exception names:

```
DECLARE Tab_not_accessible CONDITION FOR SQLSTATE '42SQK';
DECLARE "CHECK problem" CONDITION FOR SQLSTATE '23SA1';
DECLARE "Unknown problem" CONDITION;
```

Local cursors

With the definition of local cursors, cursors are defined which can only be addressed in the COMPOUND statement. The names of the local cursors must differ from each other.

Local cursors can be used in local exception routines and the SQL statements of the COMPOUND statement.

The SQL statements STORE and RESTORE are not permitted for local cursors.

local_cursor ::= { *declare_cursor_statement* ; } ...

A cursor definition is concluded with a ";" (semicolon).

Multiple cursor definitions can be specified one after the other.

declare_cursor_statement

DECLARE CURSOR statement (see section “DECLARE CURSOR - Declare cursor”) with which the local cursor is defined. The WITH HOLD clause may not be specified.

A local cursor differs from a normal cursor only in its limited area of validity.

Example

See section “DECLARE CURSOR - Declare cursor”.

Local exception routines

local_exception_handling ::= *exception_routine* ; [*exception_routine* ;] . . .

exception_routine ::= DECLARE { CONTINUE | EXIT | UNDO } HANDLER FOR

error_list { *routine_sql_statement* | *compound_statement* }

error_list ::= { *class_list* | *sqlstate_or_error_list* }

class_list ::= { SQLEXCEPTION | SQLWARNING | NOT FOUND }

[{ SQLEXCEPTION | SQLWARNING | NOT FOUND }] ...]

sqlstate_or_error_list ::= { *sqlstate* | *error_name* }

[, { *sqlstate* | *alphanumeric_literal* | *error_name* } , . . .]

sqlstate ::= SQLSTATE [VALUE] *alphanumeric_literal*

The definition of local exception routines determines what response is made when, during processing of an SQL statement in the context of the COMPOUND statement, an SQLSTATE '00000' is reported.

The SQLSTATEs of the classes 0xxxx (with the exception of SQLSTATE = '00000'), '1xxxx', '2xxxx', '3xxxx', and '4xxxx' (with the exception of class '40xxx') can be handled.

Exception routines are concluded with ";" (semicolon). Multiple exception routines can be specified one after the other.

When an SQLSTATE '45000' occurs (defined in the *class_list* or as *sqlstate* or *exception_name*), the exception routine for the specified SQLSTATE is executed.

If the SQLSTATE '45000' (unspecified user condition) occurs as a result of a SIGNAL or RESIGNAL statement, the *exception_name* of the exception information is evaluated, and the corresponding exception routine is executed.

DECLARE

Type of exception handling in accordance with the SQLSTATE. See also the section “Success of an SQL statement in a routine”

CONTINUE

- The updates which were made in the context of the errored SQL statement are undone.
- The exception routine's SQL statement is executed.
- If this SQL statement was terminated without success, the routine is aborted, and this SQLSTATE is returned to the user.
- If this SQL statement was terminated error free, the routine is continued. The SQL statement which reported the SQLSTATE and consequently triggered exception handling is regarded as successful.

EXIT

- The updates which were made in the context of the errored SQL statement are undone.
- The exception routine's SQL statement is executed. Open local cursors are closed.
- If this SQL statement was terminated without success, the routine is aborted, and this SQLSTATE is returned to the user.
- If this SQL statement was terminated error free, the routine is terminated. The COMPOUND statement is regarded as successful (SQLSTATE = '00000' is returned).

UNDO (permitted only when ATOMIC is specified in the COMPOUND statement)

- All updates which were made in the context of the COMPOUND statement are undone.
- The exception routine's SQL statement is executed. Open local cursors are closed.
- If this SQL statement was terminated without success, the routine is aborted, and this SQLSTATE is returned to the user.
- If this SQL statement was terminated error free, the routine is terminated. The COMPOUND statement is regarded as successful (SQLSTATE = '00000' is returned).

class_list

Specification of SQLSTATE sets:

- SQLWARNING indicates the SQLSTATEs of class 01xxx (warning).
- NOT FOUND indicates the SQLSTATEs of class 02xxx (no data).
- SQLEXCEPTION indicates all other SQLSTATEs of classes 0xxxxx through 4xxxx (with the exception of the SQLSTATE '00000' and class 40xxx) which can be handled in the context of an exception routine.

sqlstate

Explicit specification of SQLSTATEs.

Each alphanumeric literal must represent an SQLSTATE in 5 characters (digits or uppercase letters). Only SQLSTATEs which can be handled in the context of an exception routine may be specified.

Each SQLSTATE may only occur once in one of the exception routines of the COMPOUND statement.

The list below shows examples of SQLSTATEs for which separate exception handling can make sense (see the " Messages" manual):

01004	String data was truncated on the right
20000	CASE statement without hits contains no ELSE clause
21000	Derived table contains more than 1 row
22001	String data was truncated on the right
22003	Numeric value too high or too low
22SA1	Decimal places truncated or rounded
23SA0	Referential constraint violated
23SA1	CHECK constraint violated
23SA2	Unique constraint violated
23SA3	NOT-NULL constraint violated
23SA4	NOT-NULL constraint of the primary key violated
23SA5	Unique constraint of the primary key violated
24SA1	Cursor is not closed
24SA2	Cursor is not open
24SA3	Cursor is not positioned on a row

Other SQLSTATEs, e.g. syntax errors, are best handled via one of the previously described sets of SQLSTATEs.

exception_name

Name of an exception or SQLSTATE, see “Local data”.

Only exception names which can be handled in the context of an exception routine may be specified.

> i Each *exception_name* SQLSTATE may only occur once in one of the exception routines of the COMPOUND statement.

routine_sql_statement

SQL statement which is to be executed in the exception routine.

The syntax and meaning of *routine_sql_statement* are described centrally in section “SQL statements in routines”. The SQL statements named there may not be used.

compound_statement

COMPOUND statement which contains multiple SQL statements, see section “COMPOUND - Execute SQL statements in a common context”. Except for the permissible *routine_sql_statements*, a COMPOUND statement specified here may not contain any definitions of local data, cursors, or exception routines.

Example

Definition of unqualified exception handling with two exception routines.

```
DECLARE CONTINUE HANDLER FOR SQLWARNING,NOT FOUND
   SET eot=1;
DECLARE EXIT HANDLER FOR SQLSTATE '23SA0'
   BEGIN END;
```

8.2.3.10 CREATE FUNCTION - Create User Defined Function (UDF)

CREATE FUNCTION creates a UDF and saves its definition in the database.

UDFs and their use in SESAM/SQL are described in detail in chapter “Routines”.

Each routine which is called in the UDF must already exist. Nested calls of routines are thus possible, but rerecursive calls are not.

The current authorization identifier must own the schema to which the UDF belongs. It must also, for all tables and columns which are addressed in the UDF, have the privileges which are required to execute the DML statements contained in the UDF.

The current authorization identifier must have the EXECUTE privilege for the routine called directly in the UDF. It must also, for all tables and columns which are addressed in the UDF, have the privileges which are required to execute the DML statements contained in the routine.

The current authorization identifier automatically obtains the EXECUTE privilege for the UDF created. If it even has authorization to pass on the relevant privileges, it may also pass on the EXECUTE privilege to other authorization identifiers.

The UDF and the objects which are addressed in the UDF must belong to the same database. The names of these objects may possibly be complemented by the UDF's database and schema names.

```
CREATE FUNCTION
    udf ([ udf_parameter_definition [ , udf_parameter_definition ]...])
    RETURNS data_type
    { READS SQL DATA | CONTAINS SQL }
    { routine_sql_statement | compound_statement }


udf ::= routine
udf_parameter_definition ::= [IN] routine_parameter data_type
```

udf

Name of the UDF (maximum length: 31 characters). The unqualified name of the UDF must be different from the other routine names in the schema. You can qualify the table name with a database and schema name. If the CREATE FUNCTION statement is specified in a CREATE SCHEMA statement, the UDF name may be qualified only with the database and schema names from the CREATE SCHEMA statement.

([*udf_parameter_definition* [{ , *udf_parameter_definition* }...]])

List of the UDF call parameters. Any number of UDF parameters is possible. It is limited only by the maximum statement length. If no parameter is defined, the list consists only of the parentheses.

udf_parameter_definition

Definition of a UDF call parameter.
UDF call parameters have no indicator variable.

routine_parameter

Name of the UDF call parameter. The names of the UDF call parameters must differ from each other.

data_type

Data type of the UDF call parameter.
Only unqualified UDF call parameters are permitted.
dimension may not be specified.

RETURNS *data_type*

Data type of the UDF return value.
Only unqualified UDF return values are permitted.
dimension may not be specified.

READS SQL DATA

The UDF can contain SQL statements for reading data, but no SQL statements for updating data. This information is checked. In the event of an error, the statement is rejected with SQLSTATE.

> Called routines of this UDF may **not** contain the MODIFIES SQL DATA specification.

CONTAINS SQL

The UDF contains neither SQL statements for reading data nor for updating data. This information is checked. In the event of an error, the statement is rejected with SQLSTATE.

> UDFs always contain SQL statements, i.e. CONTAINS SQL is always present. The NO SQL case envisaged in the SQL standard does not occur.
> Called routines of this procedure may **not** contain the MODIFIES SQL DATA and READS SQL DATA specifications.

routine_sql_statement

A UDF contains precisely one non-atomic SQL statement or precisely one RETURN statement. The non-atomic SQL statement must contain at least one RETURN statement. The non-atomic SQL statements in SESAM/SQL are COMPOUND (without specification of ATOMIC), CASE, FOR, IF, LOOP, REPEAT, and WHILE. They can contain other (atomic or non-atomic) SQL statements. Atomic SQL statements are the other SQL statements permissible in a routine.

No privileges are checked before an SQL statement is executed.
An SQL statement in a UDF may access the parameters of the UDF and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

The syntax and meaning of *routine_sql_statement* are described centrally in section “SQL statements in routines”. The SQL statements named there, with the exception of the SQL statements for modifying data (INSERT, UPDATE, MERGE, DELETE), may be used.

compound_statement

COMPOUND statement which contains multiple SQL statements and possibly defines common local data, cursors, and exception handling routines for these, see section “COMPOUND - Execute SQL statements in a common context”.

Conditions

SESAM/SQL offers the SQL statements COMPOUND, CASE, FOR, IF, ITERATE, LEAVE, LOOP, REPEAT, SET, and WHILE for controlling routines. These SQL statements are also referred to as control statements.

You obtain diagnostic information in routines with the diagnostic statements GET DIAGNOSTICS, SIGNAL, and RESIGNAL.

In SESAM/SQL, nested calls of routines are permitted. The CALL statement is therefore one of the statements permitted in a routine.

A routine may not contain any SQL statements for transaction management (see "SQL statements for transaction management"). Local cursors can therefore not be accessed on a cross-transaction basis. STORE or RESTORE statements are not statements which are permitted in a routine; their use in a routine makes no sense.

A routine may not contain any dynamic SQL statements or cursor descriptions, see section “Dynamic SQL”.

A routine can be called in a UDF in a dynamic SQL statement. If a procedure contains parameters of the type OUT or INOUT, the corresponding arguments must be specified in a dynamic CALL statement in the form of placeholders.

Example

The `GetCurrentYear` UDF below returns the current year as a number. It contains no SQL statements for reading or updating data.

```
CREATE FUNCTION GetCurrentYear (IN "TIME" TIMESTAMP(3))
   RETURNS DECIMAL(4)
   CONTAINS SQL
   RETURN EXTRACT (YEAR FROM "TIME")
```

You will find further examples in chapter “Routines” and in the demonstration database of SESAM/SQL (see the “ Core manual”).

See also

DROP FUNCTION, COMPOUND, CASE, FOR, IF, ITERATE, LEAVE, LOOP, REPEAT, SET, WHILE, CALL, RETURN, SELECT, INSERT, OPEN, FETCH, CLOSE, GET DIAGNOSTICS, SIGNAL, RESIGNAL

8.2.3.11 CREATE INDEX - Create index

You use CREATE INDEX to generate an index for a base table. SESAM/SQL can use the index to evaluate constraints on one or more columns of the index without accessing the base table or to output the rows in the table in the order of the values in the index column(s).

The restrictions and special considerations that apply to CALL DML tables are described in the section “Special considerations for CALL DML tables”.

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

If you specify the space for the index, the current authorization identifier must own the space.

`CREATE INDEX` *index_definition* `,...ON TABLE` *table* `[USING SPACE` *space* `]`

index_definition `::=` *index* `({` *column* `[LENGTH` *length* `]},...)`

---

index_definition

Definition of one or more indexes

If you create an index for only one column, the column may not be longer than 256 characters. If you create an index involving several columns, the sum of the column lengths plus the total number of columns cannot exceed 256.

index

Name of the new index. The unqualified index name must be unique within the schema. You can qualify the index name with a database and schema name. The database and schema name must be the same as the database and schema name of the base table for which you are creating the index.

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

column

Name of the column in the base table you want to index.

A column cannot occur more than once in an index. You can create an index that applies to several columns (compound index). In this case, the index cannot apply to multiple columns.

LENGTH *length*

Indicates the length up to which the column is to be indexed. *length* must be an unsigned integer between 1 and the length of the column. You can only limit the length if the column is of the following data type: CHAR, VARCHAR, NCHAR and NVARCHAR or data types from SESAM up to V12.

LENGTH *length* omitted:
The column in its entirety in bytes is indexed.

ON TABLE *table*

Name of the base table you are indexing.

If you qualify the table name with a database and schema name, this must be the same as the database and schema name of the index.

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

USING SPACE *space*

Name of the space in which the index is to be stored.

You can qualify the space name with the database name. This database name must be the same as the database name of the base table.

The space must already be defined for the database to which the table belongs. The current authorization identifier must own the space.

USING SPACE *space* omitted:
The index is stored in the space for the base table. In the case of a partitioned table, the index is stored in the space for the first partition.

Special considerations for CALL DML tables

The CREATE INDEX statement for CALL DML tables must take the following restrictions and special considerations into account:

- Every index can only apply to one column.
- Each column can only occur once in an index.
- You can only specify the name of the primary key constraint of a database with a compound key as the column name in the index. This means that the primary key is indexed.

Indexes and integrity constraints

If you define a UNIQUE integrity constraint for a table, the columns specified in the UNIQUE constraint are implicitly indexed. If you explicitly define an index with CREATE INDEX that applies to the same columns, the implicitly defined index is deleted. The explicit index is then also used for the integrity constraint.

Examples

The example below creates a compound index for the columns CUST_NUM and COMPANY in the CUSTOMERS table. The COMPANY column is included in the index to a length of 10 characters. Store the index in the INDEXSPACE space.

```
CREATE INDEX cust_ind (cust_num,company LENGTH 10)

ON TABLE customers USING SPACE indexspace
```

In the CREATE INDEX statement, the index NAT_CUST_IND is defined for the NAT_CUST_NUM and NAT_COMPANY columns of the NAT_CUSTOMERS table. The NAT_COMPANY column has the national data type NCHAR. The first 5 characters of values in the NAT_COMPANY column are included when the index is created (1 character = 2 bytes). The index is to be created on the space with the name NAT_INDEXSPACE.

```
CREATE INDEX nat_cust_ind(nat_cust_num, nat_company LENGTH 10)

ON TABLE nat_customers USING SPACE nat_indexspace
```

See also

DROP INDEX

8.2.3.12 CREATE PROCEDURE - Create procedure

CREATE PROCEDURE creates a procedure and saves its definition in the database.

Procedures and their use in SESAM/SQL are described in detail in chapter "Routines".

Each routine which is called in the procedure must already exist. Nested calls of routines are thus possible, but rerecursive calls are not.

The current authorization identifier must own the schema to which the procedure belongs. It must also, for all tables and columns which are addressed in the procedure, have the privileges which are required to execute the DML statements contained in the procedure.

The current authorization identifier must have the EXECUTE privilege for each routine called in the procedure. It must also, for all tables and columns which are addressed in the procedure, have the privileges which are required to execute the DML statements contained in the procedure.

The current authorization identifier automatically obtains the EXECUTE privilege for the procedure created. If it even has authorization to pass on the relevant privileges, it may also pass on the EXECUTE privilege to other authorization identifiers.

The procedure and the objects which are addressed in the procedure must belong to the same database. The names of these objects may possibly be complemented by the procedure's database and schema names.

CREATE PROCEDURE *procedure* ([*procedure_parameter_definition* [, *procedure_parameter_definition*] ...])

{ MODIFIES SQL DATA | READS SQL DATA | CONTAINS SQL }

{ *routine_sql_statement* | *compound_statement* }

procedure ::= *routine*

procedure_parameter_definition ::= [IN | OUT | INOUT] *routine_parameter data_type*

---

procedure

Name of the procedure (maximum length: 31 characters). The unqualified procedure name must be unique within the routine names of the schema. You can qualify the table name with a database and schema name. If the CREATE PROCEDURE statement is specified in a CREATE SCHEMA statement, the procedure name may be qualified only with the database and schema names from the CREATE SCHEMA statement.

([*procedure_parameter_definition* [{ , *procedure_parameter_definition* } ...]])

List of the procedure parameters. Any number of procedure parameters is possible. It is limited only by the maximum statement length. If no parameter is defined, the list consists only of the parentheses.

procedure_parameter_definition

Definition of a procedure parameter.
Procedure parameters have no indicator variable.

IN: The procedure parameter is an input parameter.

OUT: The procedure parameter is an output parameter.

INOUT: The procedure parameter is an input and output parameter.

routine_parameter

Name of the procedure parameter. The names of the procedure parameters must differ from each other.

data_type

Data type of the procedure parameter.
Only unqualified procedure parameters are permitted.
dimension may not be specified.

MODIFIES SQL DATA

The procedure can contain SQL statements for updating data.

READS SQL DATA

The procedure can contain SQL statements for reading data, but no SQL statements for updating data. This information is checked. In the event of an error, the statement is rejected with SQLSTATE.

> i Called routines of this procedure may **not** contain the MODIFIES SQL DATA specification.

CONTAINS SQL

The procedure contains neither SQL statements for reading data nor for updating data. This information is checked. In the event of an error, the statement is rejected with SQLSTATE.

> i UDFs always contain SQL statements, i.e. CONTAINS SQL is always present. The NO SQL case envisaged in the SQL standard does not occur.
> Called routines of this procedure may **not** contain the MODIFIES SQL DATA and READS SQL DATA specifications.

routine_sql_statement

A procedure contains precisely one atomic or non-atomic SQL statement. The non-atomic SQL statements in SESAM/SQL are COMPOUND (without specification of ATOMIC), CASE, FOR, IF, LOOP, REPEAT, and WHILE. They can contain other (atomic or non-atomic) SQL statements. Atomic SQL statements are the other SQL statements permissible in a routine.

No privileges are checked before an SQL statement is executed.
An SQL statement in a procedure may access the parameters of the procedure and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

The syntax and meaning of *routine_sql_statement* are described centrally in section “SQL statements in routines”. The SQL statements named there may not be used with the exception of RETURN.

compound_statement

COMPOUND statement which contains multiple SQL statements and possibly defines common local data, cursors, and exception handling routines for these, see section “COMPOUND - Execute SQL statements in a common context”.

Conditions

SESAM/SQL offers the SQL statements COMPOUND, CASE, FOR, IF, ITERATE, LEAVE, LOOP, REPEAT, SET, and WHILE for controlling routines. These SQL statements are also referred to as control statements.

You obtain diagnostic information in routines with the diagnostic statements GET DIAGNOSTICS, SIGNAL, and RESIGNAL.

In SESAM/SQL, nested calls of routines are permitted. The CALL statement is therefore one of the statements permitted in a routine.

A routine may not contain any SQL statements for transaction management (see "SQL statements for transaction management"). Local cursors can therefore not be accessed on a cross-transaction basis. STORE or RESTORE statements are not statements which are permitted in a routine; their use in a routine makes no sense.

A routine may not contain any dynamic SQL statements or cursor descriptions, see section “Dynamic SQL”.

A routine can be called in a dynamic SQL statement. If a procedure contains parameters of the type OUT or INOUT, the corresponding arguments must be specified in a dynamic CALL statement in the form of placeholders.

Example

The `GetCurrentYear` procedure below returns the current year as a number. It contains no SQL statements for reading or updating data.

```
CREATE PROCEDURE ProcSchema.GetCurrentYear (OUT current_year INTEGER)
   CONTAINS SQL
   SET current_year = EXTRACT (YEAR FROM CURRENT_DATE)
```

You will find further examples in chapter “Routines” and in the demonstration database of SESAM/SQL (see the “Core manual”).

See also

CALL, DROP PROCEDURE, COMPOUND, CASE, FOR, IF, ITERATE, LEAVE, LOOP, REPEAT, SET, WHILE, SELECT, INSERT, UPDATE, DELETE, MERGE, OPEN, FETCH, UPDATE, DELETE, CLOSE, GET DIAGNOSTICS, SIGNAL, RESIGNAL

8.2.3.13 CREATE SCHEMA - Create schema

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

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

```
CREATE SCHEMA
      { schema [AUTHORIZATION authorization_identifier ] |
        AUTHORIZATION authorization_identifier }
      [ create_table_statement |
        create_view_statement |
        create_function_statement |
        create_procedure_statement |
        grant_statement |
        create_index_statement ] ...
```

schema

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

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

AUTHORIZATION *authorization_identifier*

The authorization identifier owns the schema.

This authorization identifier is used as the name of the schema if you do not specify a schema name.

AUTHORIZATION *authorization_identifier* omitted:
If an authorization identifier has been defined for the compilation unit, it owns the schema. Otherwise, the current authorization identifier becomes the owner.

create/grant_statements

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

create_table_statement

CREATE TABLE statement that creates a base table for the schema.

create_view_statement

CREATE VIEW statement that creates a view for the schema.

create_function_statement

CREATE FUNCTION statement that creates a UDF for the schema.

create_procedure_statement

CREATE PROCEDURE statement that creates a procedure for the schema.

grant_statement

GRANT statement that grants privileges for a base table, a view or a routine of this schema. You cannot grant special privileges with the GRANT statement.

create_index_statement

CREATE INDEX statement that creates and index for the schema.

create/grant_statements not specified:

An empty schema is created.

How CREATE SCHEMA functions

The CREATE TABLE, CREATE VIEW, CREATE FUNCTION, CREATE PROCEDURE, GRANT, and CREATE INDEX statements that are specified in the CREATE SCHEMA statement are executed in precisely the order in which they are specified. You must therefore place statements that reference existing tables, routines or views after the statement that creates these tables, routines or views.

Example

The example below creates the schema ADDONS and the table IMAGES.The privileges for the IMAGES table are assigned to the authorization identifier utiusr1.

```
CREATE SCHEMA addons
CREATE TABLE images OF BLOB
(
 MIME ('image / gif'),
 USAGE ('images for parts.item_cat.image'),
 '<Photographer>Hans Sesamer</Photographer>'
) USING SPACE blobspace
GRANT ALL PRIVILEGES ON images TO utiusr1
```

See also

CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE FUNCTION, CREATE PROCEDURE, GRANT, DROP SCHEMA

8.2.3.14 CREATE SPACE - Create space

You use CREATE SPACE to create a new entry for a new user space in the database metadata and to generate the corresponding file at operating system level.

You can define up to 999 user spaces for a database.
A user space can be up to 4 TB in size on pubsets with "large files".
Otherwise it can be up to 64 GB in size.

The current authorization identifier must have the special privilege USAGE for the storage group used.

If the database catalog space is in a DB user ID, preparations must have been made, see section “Database files and job variables on foreign user IDs” in the “ Core manual”.

If the file of the catalog space was created with a password, you must also specify a password for the user space files. The password must be identical to the BS2000 password for the catalog space file.

```
CREATE SPACE space

   [AUTHORIZATION authorization_identifier ]

   [ PRIMARY allocation |

     SECONDARY allocation |

     PCTFREE percent |

     [NO] SHARE |

     [NO] DESTROY |

     NO LOG ] ...

   [USING STOGROUP stogroup ]
```

space

Name of the space. The first 12 characters of the unqualified space name must be unique within the database. You can qualify the space name with the database name.

AUTHORIZATION *authorization_identifier*

Name of the authorization identifier to be entered as the owner of the space.

AUTHORIZATION *authorization_identifier* omitted:
The current authorization identifier is entered as the owner.

You may only specify each of the following parameters once: PRIMARY, SECONDARY, PCTFREE, [NO] SHARE, [NO] DESTROY, or NO LOG.

PRIMARY *allocation*

Primary allocation of the space file in units of 2K (BS2000 halfpage). *allocation* must be an unsigned integer between 1 and 2 147 483 640.

PRIMARY *allocation* omitted:
PRIMARY 24 is used.

SECONDARY *allocation*

Secondary allocation of the space file in units of 2K (BS2000 halfpage). *allocation* must be an unsigned integer between 1 and 32767.

SECONDARY *allocation* omitted:
SECONDARY 24 is used.

PCTFREE *percent*

Free space reservation in the space file expressed as a percentage. *percent* must be an unsigned integer between 0 and 70.

PCTFREE *percent* omitted:
PCTFREE 20 is used.

[NO] SHARE

SHARE indicates that the space file is sharable, i.e. that the space file can be accessed from more than one BS2000 user ID of the DBH.
NO SHARE indicates that the space file is not sharable.

NO SHARE is recommended for security reasons.

[NO] SHARE omitted:
NO SHARE is used.

[NO] DESTROY

DESTROY indicates that when the space file is deleted the storage space is to be overwritten with binary zeros.
NO DESTROY means that when the space file is deleted, just the storage space is released.

[NO] DESTROY omitted:
DESTROY is used.

NO LOG

No logging.

NO LOG omitted:
The logging setting for the database is used.

USING STOGROUP *stogroup*

Name of the storage group containing the volumes to be used for creating the space file.

If you specify the unqualified name of the storage group, the name is automatically qualified with the database name of the schema. If you qualify the name of the storage group with a database name, this name must be the same as the database name of the space.

USING STOGROUP *stogroup* omitted:
The default storage group D0STOGROUP is used.

Space file at operating system level

The space file is created either under the BS2000 user ID of the DBH or of the database with the following name:

:*catid*:$*bk*.*catalog*.*unqual_space_name*

Only the first 12 characters of the unqualified space name are used for the file name.

Example

Create the space files TABLESPACE and INDEXSPCE with a primary and secondary allocation of 192 2K-entities each. Both files are to have a free space reservation of 10%. They must be sharable and are to be overwritten with binary zeros when deleted.

INDEXSPACE is to be used exclusively to store indexes. Since indexes can be restored from the primary data as part of a media recovery process. Logging is not required and is disabled with NO LOG.

```
CREATE SPACE tablespace PRIMARY 192 SECONDARY 192

PCTFREE 10 SHARE DESTROY USING STOGROUP stogoup1


CREATE SPACE indexspace PRIMARY 192 SECONDARY 192

PCTFREE 10 SHARE DESTROY NO LOG USING STOGOUP stogroup1
```

See also

ALTER SPACE, CREATE STOGROUP

8.2.3.15 CREATE STOGROUP - Create storage group

You use CREATE STOGROUP to create a new storage group. A storage group describes either a pubset or a set of private volumes. The private volumes in a storage group must all have the same device type (see also the " Core manual").

The storage group D0STOGROUP always exists.

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

```
CREATE STOGROUP stogroup { VOLUMES ( volume_name ,...) ON dev_type | PUBLIC }[ON catid ]
```

stogroup

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

The current authorization identifier will own the storage group and is granted the special privilege USAGE for this storage group.

VOLUMES (*volume_name*,...)

The storage group is created on private volumes. *volume_name* is an alphanumeric literal indicating the VSN of the volumes. You can only specify each VSN once. You can specify up to 100 volumes.

All the volumes in a storage group must have the same device type.

ON *dev_type*
Device type of the private volumes. *dev_type* is an alphanumeric literal which can be specified as a string or in hexadecimal format.

PUBLIC

The storage group comprises a pubset.

ON *catid*

Alphanumeric literal indicating the *catid*.

If you specify PUBLIC, this is the catalog ID of the pubset on which the storage group is defined and on which the files are created. In the case of private volumes (VOLUMES), this is the pubset on which the files are catalogged. The files themselves are located on the specified private volumes.

ON *catid* omitted:
The catalog ID assigned to the BS2000 user ID under which the DBH is running is used.

When defining a storage group on a pubset, it is also possible to specify the VOLUMES (*volume_name*,...) ON devicetype parameters instead of the PUBLIC parameter in order to select individual volumes of a pubset. The ID under which the DBH is running has to be authorized to physically allocated on the pubset. This is not checked when the storage group is defined.

Examples

Create the storage group STOGROUP3 on a pubset.

```
CREATE STOGROUP stogoup3 PUBLIC
```

Create a new storage group STOGROUP4 with the specified private volumes. The catalog ID “P” is used to catalog the space files created on the storage group.

```
CREATE STOGROUP stogroup4
   VOLUMES ('DY130A','DY130B','DY130C','DY130D') ON 'D3435'
   ON 'P'
```

See also

DROP STOGROUP

8.2.3.16 CREATE SYSTEM_USER - Create system entry

You use CREATE SYSTEM_USER to define a system entry, i.e. assign authorization identifiers to the system users. You can assign an authorization identifier to more than one user, and a single user may have more than one authorization identifier.

A local UTM system user is identified by the local host name, the local UTM application name and the UTM user ID.

A UTM system user working with SESAM databases via UTM-D is identified by the local host name, the local UTM application name and the local UTM session name (LSES).

A BS2000 (TIAM) system user is identified by the host name and the BS2000 user ID.

Please note that before you move a database to another system, you must first define a valid system entry for the new system. If this is not possible for technical reasons, please contact your service agent.

The current authorization identifier must have the special privilege CREATE USER. If you want to assign a system user an authorization identifier with the special privilege CREATE USER and with GRANT authorization (see section “GRANT - Grant privileges”), the current authorization identifier must also have GRANT authorization.

`CREATE SYSTEM_USER`

    { *utm_user* | *bs2000_user* } `FOR` *authorization_identifier* `AT CATALOG` *catalog*

utm_user `::=` `(`{ *hostname* | * } , { *utm_application_name* | * } , { *utm_userid* | * }`)`

bs2000_user `::=` `(`{ *hostname* | * } , [*] , { *bs2000_userid* | * }`)`

---

utm_user

Defines a system entry for a UTM system user.

bs2000_user

Defines a system entry for a BS2000 system user.

FOR *authorization_identifier*

Name of the previously defined authorization identifier to be assigned to the system user.

AT CATALOG *catalog*

Name of the database for which the assignment of an authorization identifier to a system user is valid.

utm_user

Specification of the UTM user.

hostname

Alphanumeric literal indicating the symbolic host name.

If DCAM is not available on the host, the host is assigned the name “HOMEPROC”.

For UTM-D: Specification of the local host on which the SESAM/SQL database connection was generated.

* All hosts.

utm_application_name

Alphanumeric literal indicating the name of the UTM application.

For UTM-D: Name of the local UTM application.

* All UTM applications

utm_userid

You specify the UTM user ID as an alphanumeric literal defined with KDCSIGN for local UTM system users. For UTM-D, you specify the local UTM session name (LSES).

* All UTM user IDs.

bs2000_user

Specification of the BS2000 user.

hostname

Alphanumeric literal indicating the symbolic host name.

If DCAM is not available on the host, the host is assigned the name “HOMEPROC”.

* All hosts.

bs2000_userid

Alphanumeric literal indicating the BS2000 user ID.

* All BS2000 user IDs.

Example

In this example, two previously defined authorization identifiers are assigned to system users.

```
CREATE SYSTEM_USER (*,*,'PHOTO') FOR utiusr1 AT CATALOG ordercust
```

```
CREATE SYSTEM_USER (*,*,'TEXT') FOR utiusr2 AT CATALOG ordercust
```

This enables the authorization identifier UTIUSR1 to access the ORDERCUST database from the BS2000 user id PHOTO. This enables the authorization identifier UTIANW2 to access the ORDERCUST database from the BS2000 user id TEXT.

See also

DROP SYSTEM_USER, CREATE USER

8.2.3.17 CREATE TABLE - Create base table

You use CREATE TABLE to create a base table in which the data is permanently stored.

SESAM/SQL distinguishes between

- SQL tables that can only be processed with SQL
- BLOB tables that only contain BLOBs
- CALL DML/SQL tables that can be processed with CALL DML and to some extent with SQL
- CALL DML only tables that can only be processed with CALL DML. These CALL DML tables cannot be created with CREATE TABLE. They are created with the MIGRATE statement (see the " SQL Reference Manual Part 2: Utilities").

SQL tables, BLOB tables and CALL-DML/SQL tables can also be created as partitioned tables. A partitioned table is a base table whose data is stored in a number of spaces. The table data contained in a single space is referred to as a partition. In SESAM/SQL the data is distributed row by row to the partitions, and the assignment criterion is the primary key value. See also the section "Special features for partitioned tables". The partitioning can be changed with the utility statement ALTER PARTITIONING FOR TABLE, see the " SQL Reference Manual Part 2: Utilities".

CALL DML only tables and CALL DML/SQL tables are referred to by the term CALL DML tables.

The restrictions that apply when you use CREATE TABLE to create CALL DML tables are described in the section "Special considerations for CALL DML tables".

The structure of BLOB tables is described in the section "Special considerations for BLOB tables".

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

CREATE [CALL DML] TABLE *table*

{ (*declaration* ...) | OF BLOB (*blob-declaration*) }

[USING { SPACE *space* | PARTITION BY RANGE *partition* , ... , *last_partition* }]

declaration ::=
{ *column_definition* | [CONSTRAINT *integrity_constraint_name*] *table_constraint* }

blob-declaration ::=
{ *mime_clause* [, *usage_clause*] [, *alphanumeric_literal*] |
usage_clause [, *alphanumeric_literal*] |
alphanumeric_literal }

mime_clause ::= MIME(*alphanumeric_literal*)

usage_clause ::= USAGE(*alphanumeric_literal*)

partition ::= PARTITION *partno* VALUE {< | <=} (*column_value* ,...) ON SPACE *space*

last_partition ::= PARTITION *partno* [VALUE <=( )] ON SPACE *space*

partno ::= *unsigned_integer*

column_value ::=
{ *alphanumeric_literal* |
national_literal |
numeric_literal |
time_literal }

CALL DML

Creates a CALL DML table.
You can only process CALL DML tables with SESAM CALL DML. The column definitions and integrity conditions must observe certain restrictions (see “Special considerations for CALL DML tables”).

CALL DML omitted:
An SQL or BLOB table is created.
SQL tables can only be processed with SQL. BLOB tables can only be processed with SESAM CLI calls (see chapter “SESAM-CLI”).

TABLE *table*

Name of the new base table. The unqualified table name must be different from all the base table names and view names in the schema. You can qualify the table name with a database and schema name.

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

column_definition

Defines columns for the base table.

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

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

CONSTRAINT *integrity_constraint_name*

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

CONSTRAINT *integrity_constraint_name* omitted:
The integrity constraint is assigned a name according to the following pattern:

UN *integrity_constraint_number*
PK *integrity_constraint_number*
FK *integrity_constraint_number*
CH *integrity_constraint_number*
where UN stands for UNIQUE, PK for PRIMARY KEY, FK for FOREIGN KEY and CH for CHECK.
integrity_constraint_number is a 16-digit number.

table_constraint

Defines an integrity constraint for the base table.

OF BLOB

Creates a BLOB table.

mime_clause

Allows you to define the MIME type. For instance, the MIME type of a Microsoft TM Word document is "application/msword". If the BLOB table is defined without *mime_clause*, the default MIME type "application/octet-stream" is set. You must ensure that only permitted MIME types are specified in *mime_clause*. A list of the most important MIME types can be found under http://www.iana.org/assignments/media-types/index.html.

usage_clause

Allows you to define comments for BLOBs (see example at the end of this section). The default value is a blank.

alphanumeric_literal

In addition to the format described in the appendix, *alphanumeric_literal* must be in XML format (see examples).

USING clause

The USING clause defines whether a non-partitioned (USING SPACE) or a partitioned (USING PARTITION BY RANGE) table is created.

USING SPACE *space*

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

USING PARTITION BY RANGE *partition, ... ,last_partition*

Specifies that a partitioned table is to be created. The table must consist of at least 2 and at most 16 partitions. All partitions in a table must be located in different spaces, and all spaces must already be defined for the database. The table must have a primary key; this can be a single column or a combination of multiple columns.

partition clause

Defines a partition's properties.

partno is a an unsigned integer from 1 ... 16 and is the partition's current number *partno* must be assigned in ascending order for the individual partitions. If less than 16 partitions are defined, the series of numbers can contain gaps, and the first partition need not begin with 1.

(*column_value*,....) is a sequence of column values which defines the upper limit of the primary key interval for the partition concerned. You must always specify at least one column value, but at most as many column values as columns are contained in the primary key. The data type and value of *column_value* must match the data type of the corresponding column of the primary key; the same rules apply as for default values (see the section "Default values for table columns").

The upper limit is either included or excluded by the preceding comparison operator:

<=	Records whose primary key value is *column_value*,... or whose primary key value begins with *column_value*,.. belong to **this** partition.
<	Records whose primary key value is equal to *column_value*,... or whose primary key value begins with *column_value*,.. belong to the **next** partition.

The lexicographical rules apply for the comparison, see the section "Comparison rules".

The upper limits specified must be strictly in ascending order for the individual partitions.

The lower limit for the partition results implicitly from the upper limit of the preceding partition or from the lowest pimary key value in the table (in the first partition). All records from the primary key interval defined in this way belong to this partition.

space specifies the name of the space in which this partition is stored. The space must exist and the space owner must also be the schema owner. The spaces of a partitioned table must be disjunctive, i.e. a space may not be used for two partitions of the same table.

last_partition clause

The same conditions apply for the last partition as for *partition*.
Only the upper limit may not be specified since it is determined here from the highest primary key value. The VALUE clause can therefore also be omitted.

USING omitted:

A non-partitioned table is created in the current schema owner's default space and stored on the storage group D0STOGROUP.
The default space is D0*authorization_identifier* with the first 10 characters of the authorization identifier. If this space does not yet exist, it is created if the current authorization identifier has been granted the special privilege USAGE for the storage group D0STOGROUP.

Special considerations for CALL DML tables

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

- Only the data types CHAR, NUMERIC, DECIMAL, INTEGER and SMALLINT are permitted.
- No default value can be defined for the column with DEFAULT.
- A column that is not a primary key must have a CALL DML clause.
- The table must contain exactly one primary key restraint as the column or table constraint.
- The table constraint defines a compound primary key and must be given a name that corresponds to the name of the compound primary key in SESAM/SQL V1.x.
- The column name must be different from the integrity constraint name of the table constraint since this name is used as the name of the compound primary key.
- The following rules apply for the SAN (symbolic attribute name):
  - precisely 3 characters
  - first character: alphabetic character; second and third characters: alphabetic or numeric characters
  - not allowed: 0, I, O;
    the combinations NAM and END are likewise not allowed.

Special considerations for BLOB tables

In SESAM/SQL, BLOB tables are used as storage locations for BLOBs (**B**inary **L**arge **Ob**jects). BLOB objects are byte chains of variable length, up to a maximum of 2^{31}-1 bytes. With the help of SESAM CLI calls, BLOB values are stored piecemeal in several rows of the BLOB table. The structure of this table will have already been defined using the statement CREATE TABLE *table* OF BLOB. Columns cannot be defined at this point.

A BLOB table consists of the following columns:

- The OBJ_NR column is of data type INTEGER and contains the serial number of the BLOB within the table.
- The SLICE_NR column is of data type INTEGER and contains the serial number of a particular segment.
- The SLICE_VAL column is of type VARCHAR(31000). It contains the individual components of the BLOB value. Beginning with slice number 1, the BLOB value is specified in segments of 31 KB in length. Obviously, the last segment may be shorter than this. The row containing slice number 0 is used to store administrative information on the BLOB. The default settings for this column are defined in the attributes of the OF BLOB clause. In addition to these, they also include the CREATED and UPDATED attributes. This attributes specify the date on which the BLOB was created and last updated.
- The OBJ_REF column is of type CHAR(237). In the row containing slice number 0, it specifies the REF value of the BLOB. Otherwise, the column value is NULL. By default, this column is assigned the REF value for this table’ s class and is defined with the UNIQUE constraint.

The OBJ_NR and SLICE_NR columns together form the primary key of a BLOB table. For this primary key constraint, names generated internally are assigned as normal and must not be used elsewhere in the same schema.

It is possible for the user to append columns using ALTER TABLE. (However, it must be ensured that the default value for these additional columns is the NULL value.)

The *mime_clause*, *usage_clause* and *alphanumeric_literal* in the CREATE TABLE...OF BLOB statement are used to add attributes that describe the BLOB. The total length of all attributes must not exceed 256 bytes.

BLOB values can be incorporated in regular base tables with the help of the REF column (see section “Column definitions”).

Special features for partitioned tables

A partitioned table behaves largely like a non-partitioned table, i.e. the columns, constraints, indexes, and default values relate to all partitions.

As the partition limits are defined with the aid of the primary key, you should observe the following when you create the partitioned table:

- You can change the partition limits of a partitioned table after it has been created using ALTER PARTITIONING FOR TABLE. You can also use the utility statements EXPORT TABLE and IMPORT TABLE to create a table with modified partition limits.
- After a record has been inserted in a partitioned table it is no longer possible to change its primary key value with the UPDATE statement. However, the record can be deleted and reinserted with a new primary key value.

For BLOBs the primary key consists of the OBJ_NR and SLICE_NR columns. The object number is generated in the CLI call SQL_BLOB_OBJ_CREATE or SQL_BLOB_OBJ_CREAT2. These two calls have different characteristics:

- With SQL_BLOB_OBJ_CREATE ("SQL_BLOB_OBJ_CREATE - SQLbocr") the object number is assigned in ascending serial order.
- With SQL_BLOB_OBJ_CREAT2 ("SQL_BLOB_OBJ_CREAT2 - SQLboc2") you specify an object number range. The BLOB's object number is then assigned by SESAM/SQL within this range and also distributed equally within this range. It therefore makes sense to match the partition limits to the object number ranges.

Further information on partitioned tables and usage scenarios is provided in the " Core manual".

Examples

This example shows the CREATE TABLE statement for the non-partitioned table ORDERS of the demonstration database.

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

This example shows a corresponding CREATE TABLE statement for the ORDERS table of the demonstration database as a partitioned table.

```
CREATE TABLE orders
(order_num    INTEGER CONSTRAINT order_num_primary PRIMARY KEY,
cust_num      INTEGER CONSTRAINT o_cust_num_notnull NOT NULL
                 CONSTRAINT o_cust_num_ref_customers
                 REFERENCES customers(cust_num),
contact_num   INTEGER
                 CONSTRAINT contact_num_ref_contacts
                 REFERENCES contacts(contact_num),
order_date    DATE DEFAULT CURRENT_DATE,
order_text    CHARACTER (30),
actual        DATE,
target        DATE,
order_stat    INTEGER DEFAULT 1 CONSTRAINT order_stat_notnull NOT NULL
                                CONSTRAINT order_stat_ref_ordstat
                                           REFERENCES ordstat(ord_stat_num)
)
USING PARTITION BY RANGE
      PARTITION 02 VALUE <= (299) ON SPACE tablespace,
      PARTITION 03 VALUE <= (399) ON SPACE tablesp002,
      PARTITION 09                ON SPACE tablesp003
```

This example shows the CREATE TABLE statement for the partitioned table ADDRESS. The data is split lexicographically into 5 partitions: A through D, E through K, L through O, P through SCH and SCI through Z. The primary key consists of three columns, only the first column being used to determine the partition limits.

```
CREATE TABLE address
(name CHARACTER (40), first_name CHARACTER (40), pers_no INTEGER, ...
 PRIMARY KEY (name, first_name, pers_no))
USING PARTITION BY RANGE
      PARTITION 01 VALUE < ('E')   ON SPACE adr01,
      PARTITION 02 VALUE < ('L')   ON SPACE adr02,
      PARTITION 03 VALUE < ('P')   ON SPACE adr03,
      PARTITION 04 VALUE < ('SCI') ON SPACE adr04,
      PARTITION 05                 ON SPACE adr05
```

This example shows the CREATE TABLE statement for the CALL DML table COMPANY in the COMPANYSCH schema of the CALLCOMPANY database (see the “ CALL-DM Applications” manual).

```
CREATE CALL DML TABLE callcompany.companysch.company
   (pkey         CHARACTER(006) PRIMARY KEY,
    aname         CHARACTER(015) CALL DML ' ' AA8,
    aprice        NUMERIC(05,02) CALL DML -0 AB6,
    astock        NUMERIC(04) CALL DML -0 AC4,
    clastname     CHARACTER(015) CALL DML ' ' AD2,
    cfirstname    CHARACTER(012) CALL DML ' ' AEZ,
    cstreet       CHARACTER(015) CALL DML ' ' AFX,
    czip          CHARACTER(005) CALL DML ' ' AGV,
    ccity         CHARACTER(015) CALL DML ' ' AHT,
    ksince        CHARACTER(006) CALL DML ' ' AJR,
    krabatt      NUMERIC(04,02) CALL DML 0 AKP,
    ...
    psalary(010) NUMERIC(07,02) CALL DML 0 AT5)
   USING SPACE CALLCOMPANY.COMPANY
```

The tables IMAGES and DESCRIPTIONS are defined in the ADDONS schema. Both tables are stored in the space BLOBSPACE. While the BLOB table contains images in gif format, the DESCRIPTIONS table contains texts for these images in the form of Word documents.

```
CREATE TABLE addons.images OF BLOB
      MIME('image / gif'),
      USAGE ('images for parts.item_cat.image'),
            '<Photographer>Hans Sesamer</Photographer>')
    USING SPACE blobspace

CREATE TABLE descriptions OF BLOB
    ( MIME  ('application / msword'),
      USAGE ('word documents for parts.item_cat.desc'),
            '<AUTHOR>Herta Sesamer</AUTHOR>')
    USING SPACE blobspace
```

This example shows the CREATE TABLE statement for the partitioned BLOB table BILL. This table contains bills in the form of Word files. The bills are distributed over the individual partitions according to the quarters of a year.

```
CREATE TABLE bill OF BLOB (MIME  ('application/msword'),
USING PARTITION BY RANGE
      PARTITION 01 VALUE <= (1000000)   ON SPACE quarter01,
      PARTITION 02 VALUE <= (2000000)   ON SPACE quarter02,
      PARTITION 03 VALUE <= (3000000)   ON SPACE quarter03,
      PARTITION 04                      ON SPACE quarter04
```

A bill is generated with the CLI function SQL_BLOB_OBJ_CREAT2. Here the object number range of the bill (*min_no*, *max_no*) is selected in such a way that the bill is stored in the quarter associated with the partition:

```
SQL_BLOB_OBJ_CREAT2(&ref, &catalogId, &minObjNr, &MaxObjNr, &SQLdiag);
```

This example shows the CREATE TABLE statement for the MANUALS table in the sample

```
CREATE TABLE manuals
(ord_num    INTEGER,
 language   NCHAR(20),
 title      NCHAR(30)
)
```

See also

ALTER TABLE, CREATE SCHEMA, CREATE SPACE

8.2.3.18 CREATE USER - Create authorization identifier

You use CREATE USER to create a new authorization identifier.

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

```
CREATE USER authorization_identifier
       AT CATALOG catalog
```

authorization_identifier

Name of the authorization identifier. The first 10 characters of the authorization identifier must be unique within the database.

AT CATALOG *catalog*

Name of the database for which the authorization identifier is to be valid.

Example

Define the authorization identifiers UTIUSR1 and UTIUSR2 for the ORDERCUST database.

```
CREATE USER utiusr1 AT CATALOG ordercust

CREATE USER utiusr2 AT CATALOG ordercust
```

See also

DROP USER, CREATE SYSTEM_USER

8.2.3.19 CREATE VIEW - Create view

You use CREATE VIEW to create a view. A view is a table that is not permanently stored; its rows are derived only when needed.

The current authorization identifier must own the schema for which the view is created. It must have the SELECT privilege for the tables used and the EXECUTE privilege for the UDFs called.

```
CREATE VIEW table
{ [( column ,...)] AS query_expression [WITH CHECK OPTION] |
   ( column ,...) AS VALUES row ,... }

row ::= { ( expression ,...) | expression }
```

table

Name of the new view. The unqualified view name must be unique within the base tables and view names of the schema. You can qualify the view name with a database and schema name.

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

(*column*,...)

Name of the columns of the view. If *query_expression* is specified, you only need to name the view columns if the column names of the tables resulting from query_expression are ambiguous or if there are some derived columns without a name.

(*column*,...) omitted:
The column names of the query_expression are used.

AS *query_expression*

Query expression that describes how the rows of the view are derived from existing base tables and views. The columns in the view have the same data type as the underlying columns in the query expression.

AS VALUES *row ,...*

The specified rows form the new view. All the rows must have the same number of columns, and corresponding columns must have compatible data types (see section “Compatibility between data types”). If several rows are specified, the data type of the view columns results from the rules described in section “Data type of the derived column for UNION”.

expression

Each *expression* in *row* must be atomic. The row consists of the *expression* values in the order specified. A single *expression* therefore returns a row with one column.

Any tables named in *query_expression* and in *row* must belong to the same database as the view. You cannot include host variables and question marks as placeholders for unknown values in the *query_expression* and in *row*. If the columns in the view are named, the number of names must equal the number of columns in the *expression* or *row* table.

WITH CHECK OPTION

All rows that you insert or update via the view must satisfy all conditions of the query expression. The view must be updatable.

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

WITH CHECK OPTION omitted:

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

Privileges for the view

The current authorization identifier is granted the SELECT privilege for the view. This privilege includes GRANT authorization for granting this privilege to other users only if it possesses the SELECT privilege for all the tables used and the GRANT authorization identifier for the EXECUTE privilege for all UDFs called.

If the view is updatable, the current authorization identifier is granted the privileges INSERT, UPDATE, and DELETE on the view if it has been granted these privileges on the underlying base table. Each of these privileges includes the GRANT OPTION if and only if the corresponding privilege on the underlying base table includes the GRANT OPTION.

Updatable view

A view is updatable if *query_expression* is specified and the underlying query expression is updatable (see section “Updatability of query expressions”).

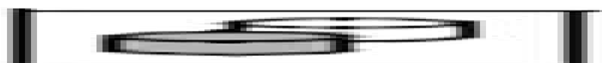
Examples

Define a view which will contain all completed orders of the ORDERS base table.

```
CREATE VIEW completed

AS SELECT * FROM orders WHERE actual IS NOT NULL
```

The example defines the view SUMMARY which will contain the customer names and associated order numbers from the CUSTOMERS and ORDERS tables.



```
CREATE VIEW summary AS SELECT
company, order_nu
```

```
FROM customers, orders WHERE
customers.cust_num=orders.cust_num
```

The example defines the LOCALEDAYNAMES view, which contains the names of the days of the week and allocates each day of the week a number.

```
CREATE VIEW localedaynames (num, name)
       AS VALUES (1 , 'Monday')
           ,(2 , 'Tuesday')
           ,(3 , 'Wednesday')
           ,(4 , 'Thursday')
           ,(5 , 'Friday')
           ,(6 , 'Saturday')
           ,(7 , 'Sunday')
```

You can use this to select the name of the day of the week for a DAY_NUM column.

```
SELECT ..., (SELECT name FROM localedaynames WHERE num = day_num)
```

Compared to the version below this not only is shorter but also has another advantage: If you switch to another language, you only have to change one single view definition instead of several SELECT expressions.

```
SELECT ..., CASE day_num WHEN 1 THEN 'Monday'
                 WHEN 2 THEN 'Tuesday'
                 WHEN 3 THEN 'Wednesday'
                 WHEN 4 THEN 'Thursday'
                 WHEN 5 THEN 'Friday'
                 WHEN 6 THEN 'Saturday'
                 WHEN 7 THEN 'Sunday'
   END
```

This is, of course, also an advantage if LOCALEDAYNAMES were a base table with this content. In that case, however, each use would involve access to persistently stored data in a file. With the view, this type of access is not necessary (just as with the CASE expression).

The view VIEW1 selects from the ORDERS table all order numbers, customer numbers, target completion dates and order status numbers for which the target completion date lies before the specified date.

```
CREATE VIEW view1 AS SELECT order_num,cust_num,target,order_status

FROM orders WHERE target < DATE'2014-05-01'
```

A second view, VIEW2 is defined to reference VIEW1. This contains the order numbers, customer numbers, target completion dates and order status numbers for target completion dates later than the specified date:

```
CREATE VIEW view2 AS SELECT order_num,cust_num,target,order_status

FROM view1 WHERE target > DATE'2013-05-01'
```

VIEW2 produces the following derived table:

order_num	cust_num	target	order_status
210	106	4/1/2014	3
211	106	4/1/2014	4
250	105	3/1/2014	2

A new row is to be added to VIEW2:

```
INSERT INTO view2 (order_num,cust_num,target,order_status)

VALUES (310,100,DATE '2014-06-01',5)
```

The new row is added, but cannot be seen either in VIEW1 or VIEW2. The row complies with the WHERE condition in the definition of VIEW2, but not with the WHERE condition in the definition of VIEW1. If we expand the definition of VIEW1 in VIEW2, we see:

```
CREATE VIEW view2 AS
SELECT view1.order_num, view1.cust_num,view1.target,view1.order_status
FROM
   (SELECT orders.order_num, orders.cust_num,orders. target,orders.
   order_status
   FROM orders
   WHERE orders.target < DATE '2014-05-01') AS view1
WHERE view1.target > DATE '2013-05-01'
```

This makes it clear that the WHERE condition in VIEW1 is “inherited” by the definition of VIEW2, with the result that the row added to the ORDERS table is not visible in VIEW2.

If WITH CHECK OPTION is added to the definition of VIEW2, the INSERT statement is rejected, since only those rows are accepted which fulfill the WHERE condition in VIEW1.

The INSERT statement is, however, also rejected if WITH CHECK OPTION is added to the definition of VIEW2 only. Although the row to be inserted fulfils the WHERE condition in the definition of VIEW2, the INSERT statement is nevertheless rejected since the row fails to fulfil the WHERE condition of VIEW1.

See also

CREATE SCHEMA, DROP VIEW

8.2.3.20 DEALLOCATE DESCRIPTOR - Release SQL descriptor area

You use DEALLOCATE DESCRIPTOR to release an SQL descriptor area.

You must have previously created the descriptor area with ALLOCATE DESCRIPTOR.

```
DEALLOCATE DESCRIPTOR GLOBAL descriptor
```

descriptor

Name of the SQL descriptor area to be released.
You cannot release the descriptor area if there is an open cursor with block mode activated in the same compilation unit (see section “PREFETCH pragma”) and a FETCH NEXT... statement has been executed for this cursor whose INTO clause contains the name of the same SQL descriptor area.

Example

Release SQL descriptor area The descriptor area name is contained in the host variable DEMO_DESC.

```
DEALLOCATE DESCRIPTOR GLOBAL descriptor :demo_desc
```

See also

ALLOCATE DESCRIPTOR, DESCRIBE, GET DESCRIPTOR, SET DESCRIPTOR

8.2.3.21 DECLARE CURSOR - Declare cursor

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

The cursor declaration must physically precede any statement that uses the cursor in the program text. All the statements that use this cursor must be located in the same compilation unit. This does not apply for local cursors (in procedures).

DECLARE CURSOR is not an executable statement.

```
DECLARE cursor [SCROLL | NO SCROLL ] CURSOR
               [WITH HOLD | WITHOUT HOLD]
               FOR { cursor_description | statement_id }


cursor_description ::=
    query_expression
    [ORDER BY sort_expression [ASC | DESC]
             [, sort_expression [ASC | DESC]]...]
    [FETCH FIRST max ROWS ONLY]
    [FOR { READ ONLY | UPDATE [OF column ,...] }]


sort_expression ::= { column | { column ( posno ) | column[posno] } | column_no | expression }
posno ::= unsigned_integer
column_no ::= unsigned_integer
max ::= unsigned_integer
```

cursor

Name of the cursor. You cannot define more than one cursor with the same name within a compilation unit. The scope of validity of the cursor is limited to the compilation unit in which the cursor is defined. This does not apply for local cursors (in procedures).

SCROLL

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

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

If you specify SCROLL, *cursor* cannot be changed. The FOR READ ONLY clause applies implicitly.

NO SCROLL

The derived table can only be read sequentially. The cursor can only be positioned on the next row. In FETCH, only the position specification NEXT is permitted.

WITH HOLDA

A cursor can be defined with WITH HOLD. Keeps such a cursor open at the end of the transaction, even after COMMIT WORK. WITH HOLD cannot be specified for local cursors (in procedures), see section “Cursor”.

Nevertheless, if a cursor defined with WITH HOLD is opened with OPEN or positioned with FETCH within a transaction and the transaction is terminated with ROLLBACK, the cursor will be closed regardless. The cursor will also be closed automatically at the end of the SQL session.

WITHOUT HOLD

Closes any open cursors at the end of the transaction.

cursor_description

Declares a static cursor.

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

statement_id

Declares a dynamic cursor.

statement_id is the name of a dynamic cursor description. You can specify a dynamic cursor description at program runtime. The same clauses can be used as in a static cursor description. You must prepare a dynamic cursor description with a PREPARE statement in which the name *statement_id* is used.

query_expression

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

In *query_expression* the value for host variables, procedure parameters and procedure variables is only determined when the cursor is opened. Special literals and time functions that are used in *query_expression* are not evaluated until the cursor is opened.

ORDER BY

The ORDER BY clause indicates the columns according to which the derived table is to be sorted. The rows are sorted according to the values in the column specified first. If rows occur which have the same values in the first column according to the comparison rules (see section “Comparison of two rows”ff), these will be sorted according to the second column, and so on. In SESAM/SQL, NULL values are considered smaller than all non-NULL values for sorting purposes.

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

You can only specify ORDER BY if no FOR UPDATE clause was declared for the cursor description of *cursor*.

If you specify ORDER BY, *cursor* cannot be changed. The FOR READ ONLY clause applies implicitly.

ORDER BY omitted: The order of the rows in the cursor table is undefined.

column

Name of the column in *query_expression* according to which the table is to be sorted. *column* must be an unqualified column name, excluding the table name. It must belong to the derived table created by *query_expression*.

{*column*(*pos_no*), *column[pos_no]*}

Element of a multiple column according to which the table is to be sorted. *pos_no* is an unsigned integer which indicates the position number of the column element in the multiple column. Otherwise, the column element must belong to the derived table created by *query_expression*.

> i Any square brackets shown here in italics are special characters, and must be specified in the statement.

column_number

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

column_number is an unsigned integer where1 <= *column_number* <= number of derived columns.

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

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

expression

It is also possible to sort a table on the basis of expressions that are not present in the derived table, e.g. UPPER(*column*).

The following conditions must be satisfied:

- *query_expression* must be a simple SELECT expression.
- *expression* may not consist of just one literal.
- *expression* must not contain any subqueries or aggregate functions.

- Columns of tables specified in the FROM clause may be used in *expression*, even if they are not included in the SELECT list.

ASC

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

DESC

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

FETCH FIRST *max* ROWS ONLY

Limits the number of hits returned by a cursor to *max* (unsigned integer > 0) sets of hits. If the cursor position is greater than *max*, an SQLSTATE is returned (no data, class 02xxx). A cursor with this clause is not updatable.

FOR READ ONLY

The FOR READ ONLY clause specifies that *cursor* can only be used to read the records of the derived table (read-only cursor).

If the relevant query expression is not updatable, the FOR READ ONLY clause applies implicitly (see section “Updatability of query expressions”). It also applies if SCROLL, ORDER BY or FETCH FIRST *max* ROWS ONLY was specified in the cursor declaration.

FOR UPDATE

You can only use the FOR UPDATE clause if the relevant query expression is updatable (see section “Updatability of query expressions”) and neither SCROLL nor ORDER BY nor FETCH FIRST *max* ROWS ONLY was specified. You use a FOR UPDATE clause to specify which columns in the underlying table can be updated via the cursor with UPDATE...WHERE CURRENT OF.

If a PREFETCH pragma has been defined for the cursor concerned, the FOR UPDATE clause disables this pragma (see section “PREFETCH pragma”).

FOR UPDATE omitted: If the cursor is updatable (see section “Defining a cursor”) and the FOR READ ONLY clause is not specified, you can update all the columns of the underlying table with UPDATE...WHERE CURRENT.

OF *column*,...

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

Example

In the example below, an updatable cursor cur is declared. The underlying table is TAB. Only column `col` in table TAB can be updated via cursor CUR. To do this, a FOR UPDATE clause with the column name COL is specified in the cursor description.

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

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

OF *column*,... omitted: Each column in the underlying table can be updated with UPDATE...WHERE CURRENT OF.

Examples

The cursor CUR_ORDER selects ORDER_NUM, CUST_NUM, CONTACT_NUM, ORDER_TEXT, TARGET and ORDER_STAT for orders numbered between 300 and 500. The derived table is then sorted on the basis of the order number in ascending order.

```
DECLARE cur_order CURSOR FOR
   SELECT order_num, cust_num, contact_num, order_text, actual, order_stat
   FROM orders
   WHERE order_num BETWEEN 300 AND 500
   ORDER BY order_num ASC
```

The cursor CUR_ORDER1 selects ORDER_NUM, ORDER_DATE, ORDER_TEXT and ORDER_STAT for orders whose customer number is specified in the host variable CUSTOMER_NO.

```
DECLARE cur_order1 CURSOR FOR
   SELECT order_num, order_date, order_text, order_stat
   FROM orders
   WHERE cust_num= :CUSTOMER_NO
```

Use the cursor CUR_VAT to select all services for which no VAT is calculated. It is specified with WITH HOLD so that it remains open even after COMMIT WORK, provided it is open at the end of the transaction.

```
DECLARE CUR_VAT CURSOR WITH HOLD FOR
   SELECT service_num, service_text, vat
   FROM service
   WHERE vat=0.00
   FOR UPDATE
```

Block mode for a static cursor is specified as follows:

```
--%PRAGMA PREFETCH blocking_factor
DECLARE cursor CURSOR FOR cursor_description
```

See also

CLOSE, DELETE, FETCH, INSERT, OPEN, PREPARE, SELECT, UPDATE

8.2.3.22 DELETE - Delete rows

You use DELETE to delete rows from a table.

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

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

DELETE FROM *table* [[AS] *correlation_name*]

[WHERE { *search_condition* | CURRENT OF *cursor* }]

---

table

Name of the table from which rows are to be deleted. The table can be a base table or an updatable view.

correlation_name

Table name used in the *search_condition* as a new name for the table *table*.

The *correlation_name* must be used to qualify the column name in every column specification that references the table *table* if the column name is not unambiguous.

The new name must be unique, i.e. *correlation_name* can only occur once in a table specification of this search condition.

You must give a table a new name if the columns in the table cannot be identified otherwise uniquely.

In addition, you may give a table a new name in order to formulate an expression so that it is more easily understood or to abbreviate long names.

WHERE clause

Indicates the rows to be deleted.

WHERE clause omitted:
All the rows in the table are deleted.

search_condition

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

Table specification in *search_condition* that are outside of subqueries can only reference the specified *table.*

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

CURRENT OF *cursor*

Name of the cursor used to select the rows to be deleted. The cursor must be updatable (see section “Defining a cursor”) and *table* must be the underlying table.

The cursor must be declared in the same compilation unit. It must be open. It must be positioned on a row in the derived table with FETCH before the DELETE statement is issued.

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

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

DELETE is not permitted if block mode is activated for the open cursor *cursor* (see section “PREFETCH pragma”).

If a cursor is defined with the WITH HOLD clause, a DELETE statement may not be issued until a FETCH statement has been executed for this cursor in the same transaction.

DELETE and transaction management

DELETE initiates an SQL transaction outside routines if no transaction is open. If you define an isolation level, you can control what effect this DELETE statement has on concurrent transactions (see section “SET TRANSACTION - Define transaction attributes”).

If an error occurs during the DELETE statement, any deletions already performed are canceled.

Examples

Delete all customers situated in Hanover from the CUSTOMERS table.

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

All customers for whom USA is entered as the country in the CUSTOMERS table are to be deleted from the CONTACTS table. The statement is only executed if the referential constraint CON_REF_CONTACTS in the ORDERS table is not violated.

```
DELETE FROM contacts
WHERE cust_num = (SELECT cust_num FROM customers WHERE country='USA')
```

Use a cursor to delete customers situated in Hanover from the CUSTOMERS table.

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

OPEN cur_customers
```

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

```
FETCH cur_customers INTO :CUSTNUM, :COMPANY, :CITY


DELETE FROM customers WHERE CURRENT OF cur_customers
```

Use a cursor to select all cancelled orders (ORDER_STAT = 5) from the ORDERS table. The entries for these orders are then deleted in the SERVICE and ORDERS tables.

```
DECLARE cur_order1 CURSOR FOR
   SELECT order_num, order_text FROM orders WHERE order_stat = 5
   FOR UPDATE

FETCH cur_order1
   INTO :ORDERS.ORDER_NUM

DELETE FROM orders
   WHERE CURRENT OF cur_order1

DELETE FROM service
   WHERE order_num = :ORDERS.ORDER_NUM
```

See also

INSERT, UPDATE

8.2.3.23 DESCRIBE - Query data type of input and output values

You use DESCRIBE to write the data type descriptions of input/output values of a dynamic statement or cursor description to an SQL descriptor area.

The SQL descriptor area must be created beforehand with ALLOCATE DESCRIPTOR.

You must prepare the dynamic statement or cursor description with PREPARE before the DESCRIBE statement is executed.

```
DESCRIBE [INPUT | OUTPUT] statement_id USING SQL DESCRIPTOR GLOBAL descriptor
```

INPUT

Determines the number of input values of a dynamic statement or cursor description and describes the data type of the input values.

OUTPUT

Determines the number of output values of a dynamic SELECT statement or cursor description and describes the data type of the output values.

statement_id

Dynamic statement or cursor description.

descriptor

Name of the SQL descriptor area into which the type descriptions are to be written (see “Descriptor area field values”).

You can specify the name as an alphanumeric literal or with an alphanumeric host variable.

You cannot use this SQL descriptor area if there is an open cursor with block mode activated (see section “PREFETCH pragma”) and a FETCH NEXT... statement whose INTO clause contains the name of the same SQL descriptor area has been executed for this cursor.

Descriptor area field values

The fields of the SQL descriptor area are supplied with the following values:

The COUNT field contains the number of input values (DESCRIBE INPUT) or the number of output values (DESCRIBE OUTPUT).

In the case of DESCRIBE INPUT, the number is calculated from the number of placeholders in the dynamic statement or cursor description as follows:

Number of placeholders for unqualified values +
Number of aggregate elements of each placeholder for aggregates

For DESCRIBE OUTPUT, the number is calculated from the number of derived columns of the dynamic SELECT statement or cursor description as follows:

Number of unqualified derived columns +
Number of column elements of each multiple derived column

If the number calculated is 0, no other descriptor area fields are set.

If the number is greater than the maximum number of item descriptors specified for ALLOCATE DESCRIPTOR, no other descriptor area fields are set and an appropriate SQLSTATE is set.

Otherwise, the following fields in the SQL descriptor area are supplied with values:

- For each input value for DESCRIBE INPUT:
  - TYPE
  - LENGTH (for alphanumeric data type, national data type and time data type)
  - PRECISION (for numeric data type and for TIME and TIMESTAMP)
  - SCALE (for NUMERIC, DECIMAL, INTEGER and SMALLINT)
  - DATETIME_INTERVAL_CODE (for time data type)
  - OCTET_LENGTH
  - NULLABLE with the value 1
  - REPETITIONS
  - UNNAMED with the value 1
- For each output value for DESCRIBE OUTPUT:
  - TYPE
  - LENGTH (for alphanumeric data type, national data type and time data type)
  - PRECISION (for numeric data type and for TIME and TIMESTAMP)
  - SCALE (for NUMERIC, DECIMAL, INTEGER and SMALLINT)
  - DATETIME_INTERVAL_CODE (for time data type)
  - OCTET_LENGTH
  - NULLABLE
  - REPETITIONS
  - NAME
  - UNNAMED

The values assigned to the above-mentioned fields are described in section “Descriptor area fields”.

All the other fields in the SQL descriptor area are undefined.

Example

```
DESCRIBE OUTPUT cur_description

USING SQL DESCRIPTOR GLOBAL 'DESCR_AREA'
```

See also

ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

8.2.3.24 DROP FUNCTION - Delete User Defined Function (UDF)

DROP FUNCTION deletes a UDF.

UDFs and their use in SESAM/SQL are described in detail in chapter “Routines”.

You can ascertain which routines are defined and which routines use each other in the views for routines of the INFORMATION_SCHEMA (see chapter “Information schemas”).

When a UDF is deleted, the EXECUTE privilege for this UDF is revoked from the current authorization identifier. EXECUTE privileges which have been passed on are also revoked.

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

```
DROP FUNCTION udf { CASCADE | RESTRICT }

udf ::= routine
```

---

udf

Name of the UDF. You can qualify the unqualified UDF name with a database and schema name.

CASCADE

The UDF *udf* and each routine which *udf* calls directly or indirectly are deleted. Views which *udf* uses directly or indirectly are also deleted.

RESTRICT

The UDF *udf* can be deleted only if *udf* is used by no other routine and by no view.

See also

CREATE FUNCTION, CREATE PROCEDURE

8.2.3.25 DROP INDEX - Delete index

You use DROP INDEX to delete an index. The index may have been created explicitly with a CREATE INDEX statement or implicitly by the definition of an integrity constraint (UNIQUE).

The INDEXES view of the INFORMATION_SCHEMA provides you with information on which indexes have been defined (see chapter “Information schemas”).

If an explicitly defined index is also used by an integrity constraint, the index is not deleted but is renamed as an implicit index. The new index name starts with UI and is followed by a 16-digit number.

Indexes created implicitly by an integrity constraint (UNIQUE) are not deleted until the relevant integrity constraint is deleted.

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

```
DROP INDEX index [DEFERRED]
```

index

Name of the index to be deleted.

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

DEFERRED

This clause initiates high-speed deletion in which only the contiguous part of the index is deleted. Any relocations which exist are retained.
The next time the user space is reorganized using the utility statement REORG SPACE all the existing tables and indexes are recovered in the user space. The relocations then also disappear.
Information on the storage structure of indexes is provided in the “ Core manual”.

An implicitly generated index (in the case of a UNIQUE integrity constraint) cannot be deleted explicitly. If necessary, the index must be generated explicitly with CREATE INDEX. Here the “Generate_Type” is merely changed from “implicit” to “explicit” in the metadata. The UNIQUE integration constraint can then be deleted. In this case the index is not deleted and can now be deleted using DROP INDEX ... DEFERRED.

The DEFERRED clause can only be specified in the case of explicit deletion. It cannot be specified when deletion takes place implicitly, e.g. using DROP SPACE CASCADE.

DEFERRED omitted:SESAM/SQL deletes the indexes. This can be time-consuming when indexes are very large and fragmented.

See also

CREATE INDEX

8.2.3.26 DROP PROCEDURE - Delete procedure

DROP PROCEDURE deletes a procedure.

Procedures and their use in SESAM/SQL are described in detail in chapter “Routines”.

You can ascertain which routines are defined and which routines use each other in the views for routines of the INFORMATION_SCHEMA (see chapter “Information schemas”).

When the procedure is deleted, the EXECUTE privilege for this procedure is revoked from the current authorization identifier. EXECUTE privileges which have been passed on are also revoked.

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

`DROP PROCEDURE` *procedure* { `CASCADE` | `RESTRICT` }

procedure `::=` *routine*

---

procedure

Name of the procedure. You can qualify the procedure name with a database and schema name.

CASCADE

The procedure *procedure* and each routine which *procedure* calls directly or indirectly are deleted. Views which *procedure* uses indirectly via a UDF are also deleted.

RESTRICT

The procedure *procedure* can be deleted only if *procedure* is used by no other routine.

See also

CREATE PROCEDURE, CREATE FUNCTION, CALL

8.2.3.27 DROP SCHEMA - Delete schema

You use DROP SCHEMA to delete a database schema.

The SCHEMATA view of the INFORMATION_SCHEMA provides you with information on which schemas have been defined (see chapter “Information schemas”).

The current authorization identifier must own the schema.

```
DROP SCHEMA schema { CASCADE | RESTRICT }
```

schema

Name of the schema.

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

CASCADE

The schema *schema* and all the objects of the schema are deleted. Views, routines, and integrity constraints that reference the base tables, views, and routines in *schema* directly or indirectly are also deleted.

RESTRICT

The schema *schema* can only be deleted when it is empty. All the schema’s base tables, views, and routines must be deleted beforehand.

Example

The example deletes the ADDONS schema, provided that all base tables, views, and routines of the schema have already been deleted. The schema was qualified using the catalog name.

```
DROP SCHEMA ordercust.addons RESTRICT
```

See also

CREATE SCHEMA, DROP TABLE, DROP VIEW, DROP FUNCTION, DROP PROCEDURE

8.2.3.28 DROP SPACE - Delete space

You use DROP SPACE to delete a user space.

The SPACE view of the INFORMATION_SCHEMA provides you with information on which user spaces have been defined (see chapter “Information schemas”).

The current authorization identifier must own the space.

```
DROP SPACE space { CASCADE | RESTRICT } [FORCED]
```

space

name of the user space
You can qualify the name of the space with a database name.

CASCADE

The space *space* is deleted even if it is not empty. The base tables and indexes located in the space are also deleted. This is also the case for the views, routines, and integrity constraints which refer directly or indirectly to these base tables and indexes.

RESTRICT

The space *space* is deleted only if it is empty. All the space’s base tables and indexes must be deleted beforehand.

FORCED

The space *space* is deleted even if it cannot be opened for update processing, e.g. because its BS2000 file does no longer exist. The space is then deleted logically in SESAM/SQL, i.e. removed from the database’s metadata. When CASCADE is also specified, FORCED also applies for spaces which are affected by the deletion of the tables and indexes.

FORCED not specified
The space *space* is deleted only if it can be opened for update processing.

> i SESAM/SQL can open the space for update processing if the space’s BS2000 file can be opened without error, if the space is consistent and if it is not in one of the following states: ”check pending”, “copy pending”, “load running”, “recover pending”, “reorg pending” or “space defect” (see the “ SQL Reference Manual Part 2: Utilities”).
>
> The space file is overwritten with binary zeros if the DESTROY clause was specified when the space was created or updated and SESAM/SQL can access the space’s BS2000 file.

DROP SPACE and transactions

A DROP SPACE statement cannot be followed by a CREATE SPACE statement within the same transaction.

See also

CREATE SPACE, ALTER SPACE

8.2.3.29 DROP STOGROUP - Delete storage group

You use DROP STOGROUP to delete a storage group. You cannot delete a storage group if it is being used for spaces or has been entered in the media table
(see the " Core manual").

The STOGROUPS view of the INFORMATION_SCHEMA provides you with information on which storage groups have been defined (see chapter "Information schemas").

The current authorization identifier must own the storage group.

```
DROP STOGROUP stogroup RESTRICT
```

stogroup

Name of the storage group. The storage group cannot be deleted if it is being used.

You can qualify the name of the storage group with a database name.

See also

CREATE STOGROUP, ALTER STOGROUP

8.2.3.30 DROP SYSTEM_USER - Delete system entry

You use DROP SYSTEM_USER to delete a system entry, i.e. the assignment of an authorization identifier to a system user. You must specify the combination of system user and authorization identifier that was defined for a system entry with CREATE SYSTEM_USER.

You cannot delete a system entry if it is the last assignment of a system user to the authorization identifier of the universal user.

If an SQL transaction belonging to the system user is currently active, his or her system entry is only deleted if another system entry exists for the system user.

The SYSTEM_ENTRIES view of the INFORMATION_SCHEMA provides you with information on which authorization identifiers have been assigned to which system users (see chapter “Information schemas”).

The current authorization identifier must have the special privilege CREATE USER. If the assignment of an authorization identifier with the special privilege CREATE USER and GRANT authorization (see section “GRANT - Grant privileges”) to a system user is to be deleted, the current authorization identifier must also have GRANT authorization.

```
DROP SYSTEM_USER{ utm_user | bs2000_user }
    FOR authorization_identifier
    AT CATALOG catalog

utm_user ::= ({hostname|*},{utm_application_name|*},{utm_userid|*})
bs2000_user ::= ({hostname|*},[*],{bs2000_userid|*})
```

---

utm_user

Delete a system entry of a UTM system user.

bs2000_user

Delete a system entry of a BS2000 system user.

FOR *authorization_identifier*

Name of the authorization identifier assigned to the system user.

AT CATALOG *catalog*

Name of the database for which the assignment of the system user to the authorization identifier is to be deleted.

utm_user

Specification of the UTM user.

The UTM user must be specified precisely as defined with CREATE SYSTEM_USER. * means the system access which was defined with *, not all corresponding system accesses.

hostname

Alphanumeric literal indicating the symbolic host name.

If DCAM is not available on the host, the host is assigned the name 'HOMEPROC'.

* All hosts.

utm_application_name

Alphanumeric literal indicating the name of the UTM application.

* All UTM applications

utm_userid

You specify the UTM user ID as an alphanumeric literal defined with KDCSIGN for local UTM system users. For UTM-D, you specify the local UTM session name (LSES).

* All UTM user IDs.

bs2000_user

Specification of the BS2000 user.

The BS2000 user must be specified precisely as defined with CREATE SYSTEM_USER. * means the system access which was defined with *, not all corresponding system accesses.

hostname

Alphanumeric literal indicating the symbolic host name. If DCAM is not available on the host, the host is assigned the symbolic name 'HOMEPROC'.

* All hosts.

bs2000_userid

Alphanumeric literal indicating the BS2000 user ID.

* All BS2000 user IDs.

Example

In the example below, two system entries are deleted. The system entries must be specified exactly as they were defined with CREATE SYSTEM_USER. The authorization identifiers UTIUSR1 and UTIUSR2 are not deleted.

```
DROP SYSTEM_USER (*,*,'PHOTO') FOR utiusr1 AT CATALOG ordercust

DROP SYSTEM_USER (*,*,'TEXT') FOR utiusr2 AT CATALOG ordercust
```

See also

CREATE SYSTEM_USER, CREATE USER, DROP USER

8.2.3.31 DROP TABLE - Delete base table

You use DROP TABLE to delete a base table and the associated indexes.

When a base table is deleted, all the table and column privileges for this base table are revoked from the current authorization identifier. Table and column privileges that have been passed on are also revoked.

The BASE_TABLES view in the INFORMATION_SCHEMA provides you with information on which base tables have been defined (see chapter “Information schemas”).

You can also use DROP TABLE to delete BLOB tables. In this case all BLOBs contained therein will also be deleted.

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

```
DROP TABLE table [DEFERRED] { CASCADE | RESTRICT }
```

table

Name of the base table to be deleted.

DEFERRED

This clause initiates high-speed deletion of the table in which only the contiguous part of the table and of the associated explicit and implicit indexes are deleted. Any relocations which exist are retained.
The next time the user space is reorganized using the utility statement REORG SPACE all the existing tables and indexes are recovered in the user space.
The relocations which have not been deleted then also disappear.
Information on the storage structure of base tables is provided in the “ Core manual”.

In the case of partitioned tables the DEFERRED clause applies for all partitions; it cannot be restricted to individual partitions.

The DEFERRED clause can only be specified in the case of explicit deletion. In the case of implicit deletion, e. g. with DROP SPACE ... CASCADE, it cannot be specified.

When DEFERRED is to apply only for the table but not for the indexes, the indexes must first be deleted using DROP INDEX (without specifying DEFERRED). The table can then be deleted using DROP TABLE ... DEFERRED.

DEFERRED omitted:
SESAM/SQL deletes the table and all associated indexes. This can be time-consuming when indexes are very large and fragmented.

CASCADE

The base table *table* and all the associated indexes are deleted. All the views, routines,and integrity constraints that reference *table* directly or indirectly are also deleted.

RESTRICT

The deletion of the base table *table* is restricted. The base table *table* cannot be deleted if it is used in a view definition, a routine or an integrity constraint of another base table.

Examples

In this example, the CUSTOMERS table is deleted only if all integrity constraints of other base tables that reference the CUSTOMERS table have been deleted beforehand. In addition, the CUSTOMERS table must not be used in any view definition.

```
ALTER TABLE contacts DROP CONSTRAINT contact_cust_num_ref_customers CASCADE

ALTER TABLE orders DROP CONSTRAINT o_cust_num_ref_customers CASCADE

DROP TABLE customers RESTRICT
```

The example deletes the IMAGES and DESCRIPTIONS tables, together with all indexes, views, and integrity constraints that reference these tables.

```
DROP TABLE images CASCADE

DROP TABLE descriptions CASCADE
```

See also

CREATE TABLE, ALTER TABLE

8.2.3.32 DROP USER - Delete authorization identifier

You use DROP USER to delete an authorization identifier and the associated system entries. You cannot delete an authorization identifier if it is the owner of schemas, spaces or storage groups, if it is the grantor of a privilege, or if an SQL transaction is currently active for the authorization identifier.

You cannot delete the authorization identifier of the universal user.

The USERS view of the INFORMATION_SCHEMA provides you with information on which authorization identifiers have been defined. Information on which authorization identifiers are owners is stored in the SCHEMATA, SPACES and STOGROUPS views. The TABLE_PRIVILEGES, COLUMN_PRIVILEGES, USAGE_PRIVILEGES, CATALOG_PRIVILEGES and ROUTINE_PRIVILEGES views provide you with information on whether the authorization identifier is the grantor of a privilege (see chapter “Information schemas”).

The current authorization identifier must have the special privilege CREATE USER. If you want to delete an authorization identifier that as been granted the special privilege CREATE USER and GRANT authorization (see section “GRANT - Grant privileges”), the current authorization identifier must also have GRANT authorization.

```
DROP USER authorization_identifier AT CATALOG catalog RESTRICT
```

authorization_identifier

Name of the authorization identifier to be deleted.

AT CATALOG *catalog*

Name of the database from which the authorization identifier is to be deleted.

See also

CREATE USER, CREATE SYSTEM_USER, DROP SYSTEM_USER

8.2.3.33 DROP VIEW - Delete view

You use DROP VIEW to delete the definition of a view.

When a view definition is deleted, all the table and column privileges for this view are revoked from the current authorization identifier. Table and column privileges of the view that have been passed on are also revoked.

The VIEWS view of the INFORMATION_SCHEMA provides you with information on which views have been defined. Information on the tables a view uses is provided in the view VIEW_TABLE_USAGE (see chapter "Information schemas").

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

```
DROP VIEW table { CASCADE | RESTRICT }
```

table

Name of the view to be deleted.

CASCADE

The *table* view and all views and routines which refer directly or indirectly to *table* are deleted.

RESTRICT

The deletion of the view *table* is restricted. The view cannot be deleted if it is used in another view definition or in a routine.

See also

CREATE VIEW

8.2.3.34 EXECUTE - Execute prepared statement

You use EXECUTE to execute a statement prepared with PREPARE. Placeholders for input values in the dynamic statement are replaced by specific values.

If the statement is a SELECT statement, the column values of the derived rows are stored in host variables or in an SQL descriptor area.

If the statement is a CALL statement, values of output parameters are stored in host variables or in an SQL descriptor area.

You can use EXECUTE to execute a previously prepared statement any number of times.

A statement can only be executed with EXECUTE in the compilation unit in which it was previously prepared with PREPARE.

```
EXECUTE statement_id
[INTO declaration ]
[USING declaration ]


declaration ::= { variable | SQL DESCRIPTOR GLOBAL descriptor }


variable ::= :host_variable [[INDICATOR] :indicator_variable]
         [,:host_variable [[INDICATOR] :indicator_variable]] ...
```

statement_id

Identifier of the dynamic statement that has been prepared with PREPARE.

If the statement text contains a cursor name, the cursor description for this cursor must be prepared and the cursor opened before the EXECUTE statement is executed.

INTO clause

Indicates where the output values of the dynamic statement specified with *statement_id* are to be stored. The INTO clause must be specified in the following cases:

- The prepared statement is a SELECT statement
- The prepared statement is a CALL statement and the procedure called in it has parameters of the type OUT or INOUT

host_variable

Name of a host variable assigned an output value.

The data type of a host variable must be compatible with the data type of the relevant output value (see section “Reading values into host variables or a descriptor area”).

If an output value is an aggregate with several elements (SELECT statement), the corresponding host variable must be a vector with the same number of elements. The number of specified host variables must be the same as the number of output values in the SELECT statement specified with *statement_id*.

In a procedure call using the CALL statement, the number of host variables specified must match the number of procedure parameters of the type OUT or INOUT in the procedure called.

indicator_variable

Name of the indicator variable for the preceding host variable.

If the host variable is a vector (SELECT statement), the indicator variable must also be a vector with the same number of elements.

The indicator value indicates whether the NULL value was transferred or whether data was lost:

0 The host variable contains the value read. The assignment was error free.

-1 The value to be assigned is the NULL value.

> 0 For alphanumeric and national values:
The host variable was assigned a truncated string. The value of the indicator variable indicates the original length in code units.

descriptor

Name of an SQL descriptor area containing the data type description of the output values and into which the output values (for procedures the procedure parameters of the type OUT or INOUT) are written when the statement specified by *statement_id* is executed.

The SQL descriptor area must be created beforehand and supplied with appropriate values:

- The value of the COUNT field must be the same as the number of output values of the statement specified with *statement_id* (for aggregates one output value for each element, for procedures one output value for each procedure parameter of the type OUT or INOUT) where

  0 <= COUNT <= defined maximum number of item descriptors
- The output values are assigned to the DATA fields of the item descriptors in the order of the items in the descriptor area. The data type description for an item must be compatible with the data type of the corresponding output value (see section “Reading values into host variables or a descriptor area”).

  If the value to be transferred is the NULL value, the appropriate INDICATOR field is set to the value -1. If a string to be assigned is truncated, the corresponding INDICATOR field indicates the original length.

USING clause

Specifies where the input values for the dynamic statement *statement_id* are to be read from. The INTO clause must be specified in the following cases:

- When the SELECT statement contains question marks as placeholders for values

- When the procedure called in the CALL statement has parameters of the type IN or INOUT and the corresponding arguments contain question marks as placeholders for values

host_variable

Name of a host variable containing the value to be assigned to a placeholder in the dynamic statement *statement_id*.

The data type of a host variable must be compatible with the data type of the corresponding placeholder (see section “Values for placeholders”).

If the placeholder represents an aggregate with several elements (SELECT statement), the corresponding host variable must be a vector with the same number of elements.
The number of host variables specified must be the same as the number of placeholders in the SELECT statement.

In a procedure call using the CALL statement, the number of host variables specified must match the number of placeholders for parameters of the data type IN or INOUT.

The user variables are assigned values in the order in which the placeholders are specified in the dynamic statement.

indicator_variable

Name of the indicator variable for the preceding host variable.

If the host variable is a vector (SELECT statement), the indicator variable must also be a vector with the same number of elements.

The value of the indicator variable indicates whether the NULL value is to be transferred:

< 0	The NULL value is to be assigned.
>= 0	The value of the host variable is to be assigned.

descriptor

Name of an SQL descriptor area containing the data types and values for the placeholders in the dynamic statement *statement_id*.

The SQL descriptor area must be created beforehand and supplied with appropriate values:

- The value of the descriptor area field COUNT must be the same as the number of input values required (for aggregates one input value for each element, for procedures one output value for each procedure parameter of the type OUT or INOUT) where

  0 <= COUNT <= defined maximum number of item descriptors
- The values of the DATA fields of the item descriptors (or NULL values if the INDICATOR is negative) are assigned to the placeholders in the dynamic statement in the order of the items in the descriptor area. The data type
  description of an item must be compatible with the data type of the corresponding placeholder (see section “Values for placeholders”).

Example

```
EXECUTE dyn_statement

INTO SQL DESCRIPTOR GLOBAL 'DESCR_AREA'
```

See also

EXECUTE IMMEDIATE, PREPARE, SELECT

8.2.3.35 EXECUTE IMMEDIATE - Execute dynamic statement

You use the EXECUTE IMMEDIATE statement to prepare and execute a dynamic statement in one step. In other words, EXECUTE IMMEDIATE corresponds to a PREPARE statement immediately followed by an EXECUTE statement. The statement does not, however, remain prepared and cannot be executed again with EXECUTE.

Dynamic CALL statements can be executed with EXECUTE IMMEDIATE if the procedure to be called has no procedure parameters or only procedure parameters of the type IN.

`EXECUTE IMMEDIATE` *statement_variable*

statement_variable `::=` `:` *host_variable*

---

statement_variable

Alphanumeric host variable containing the statement text. The host variable can also be of the type CHAR(*n*), where 256 <= *n* <= 32000.

The following conditions must be satisfied:

- The statement text cannot include any host variables or question marks as placeholders for unknown values.
- The statement text cannot contain either SQL comments or comments in the host language. Pragmas (--% PRAGMA) are exceptions.
- The statement text cannot be a SELECT statement or cursor description.
- The RETURN INTO clause cannot be specified in an INSERT statement.
- If the statement text contains a cursor name (DELETE WHERE CURRENT OF, UPDATE WHERE CURRENT OF), the cursor description for this cursor must be prepared and the cursor opened before the EXECUTE IMMEDIATE statement is executed.

Statements for EXECUTE IMMEDIATE

The following statements can be executed with EXECUTE IMMEDIATE:

ALTER SPACE	DROP SCHEMA
ALTER STOGROUP	DROP SPACE
ALTER TABLE	DROP STOGROUP
COMMIT	DROP SYSTEM_USER
CALL (only input parameters; Type IN)	DROP TABLE
CREATE INDEX	DROP USER
CREATE FUNCTION	DROP VIEW
CREATE PROCEDURE	GRANT
CREATE SCHEMA	INSERT (without RETURN INTO clause)
CREATE SPACE	MERGE
CREATE STOGROUP	PERMIT

CREATE SYSTEM_USER
CREATE TABLE
CREATE USER
CREATE VIEW
DELETE
DROP FUNCTION
DROP INDEX
DROP PROCEDURE
REORG STATISTICS
REVOKE
ROLLBACK
SET CATALOG
SET SCHEMA
SET SESSION AUTHORIZATION
SET TRANSACTION
UPDATE

In addition, all utility statements can be executed with EXECUTE IMMEDIATE (see the " SQL Reference Manual Part 2: Utilities").

The following statements cannot be executed with EXECUTE IMMEDIATE:

ALLOCATE DESCRIPTOR
CLOSE
DEALLOCATE DESCRIPTOR
DECLARE CURSOR
DESCRIBE
EXECUTE
EXECUTE IMMEDIATE
FETCH
GET DESCRIPTOR
INCLUDE
OPEN
PREPARE
RESTORE
SELECT
SET DESCRIPTOR
STORE
WHENEVER

Example

An SQL statement is to be compiled and executed at runtime with EXECUTE IMMEDIATE:

The following SQL statement is read into SOURCESTMT as an alphanumeric string:

```
CREATE TABLE ordercust.orderproc.ordstat

(order_stat_num INTEGER, order_stat_text CHAR(15))
```

The statement is compiled and executed with:

```
EXEC SQL EXECUTE IMMEDIATE :SOURCESTMT END-EXEC
```

See also

EXECUTE, PREPARE

8.2.3.36 FETCH - Position cursor and read row

You use FETCH to position a cursor. The new cursor position is either on a row, before the first row or after the last row of the cursor table. If the new cursor position is on a row in the cursor table, this row is the current row and the column values of this row can be read. If no row is read for FETCH because the specified position does not exist, an appropriate SQLSTATE is set, which can be handled with WHENEVER NOT FOUND. If you declare a cursor with SCROLL, the cursor can be positioned with FETCH on any row in the cursor table and in any order. A cursor defined with NO SCROLL can only be positioned on the next row (FETCH NEXT...).

You can transfer the values of the current row to host variables, procedure parameters of the type INOUT or OUT, local variables or an SQL descriptor area.

The cursor declaration with DECLARE CURSOR must be located in the same compilation unit and must physically precede the FETCH statement in the program text.

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

If the cursor is declared with WITH HOLD, the isolation level or consistency level of the transaction must be the same as when the cursor was opened.

If block mode is activated for the cursor (see section “PREFETCH pragma”) and if a FETCH NEXT... statement has already been executed for the cursor, only this FETCH NEXT statement is permitted subsequently for this cursor, i. e. the same statement in a loop or subroutine.

```
FETCH { [NEXT] | PRIOR | FIRST | LAST | RELATIVE n | ABSOLUTE n }
      [FROM] cursor
      { INTO variable ,... | SQL DESCRIPTOR GLOBAL descriptor }


n ::= { integer | :host_variable | routine_parameter | local_variable }


variable ::=
{
   :host_variable [[INDICATOR] :indicator_variable ] |
   routine_parameter |
   local_variable
}
```

NEXT

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

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

PRIOR

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

If the cursor is positioned on the first row of the cursor table, it is positioned before the first row. If it is already positioned in front of the first row, its position remains unchanged.
You can only specify PRIOR if you declared the cursor with SCROLL.

FIRST

Positions the cursor on the first row of the cursor table or before the first row if the cursor table is empty.
You can only specify FIRST if you declared the cursor with SCROLL.

LAST

Positions the cursor on the last row of the cursor table or after the last row if the cursor table is empty.
You can only specify LAST if you declared the cursor with SCROLL.

ABSOLUTE *n*

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

You can specify the following for *n*:

- An integer
- A host variable (if the statement is **not** part of a procedure) of the SQL data type INT or SMALLINT
- A routine parameter or a local variable (if the statement is part of a routine) of the SQL data type INT or SMALLINT

The cursor position is determined by the value of *n* as follows:

> 0 — The cursor is positioned on the *n*th row of the cursor table or after the last row if *n* > number of rows in the cursor table.

0 — The cursor is positioned before the first row of the cursor table.

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

Example

`FETCH ABSOLUTE -1` and `FETCH LAST` are equivalent.

RELATIVE *n*

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

You can specify the following for *n*:

- An integer literal
- A host variable (if the statement is **not** part of a procedure) of the SQL data type INT or SMALLINT
- A routine parameter or a local variable (if the statement is part of a routine) of the SQL data type INT or SMALLINT

The cursor position is determined by the value of *n* as follows:

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

FROM *cursor*

Name of the cursor.

INTO clause

Indicates where the values read are to be stored.

:host_variable, *routine_parameter*, *local_variable*

Name of a host variable (if the statement is **not** part of a procedure) or name of a procedure parameter of the type INOUT or OUT or of a local variable (if the statement is part of a routine). The column value of the derived row is assigned to the specified output destination.

The data type must be compatible with the data type of the relevant output value (see section “Reading values into host variables or a descriptor area”). If an output value is an aggregate with several elements (only in the case of host variables), the corresponding host variable must be a vector with the same number of elements.

The number of specified elements must match the number of columns in the SELECT list of the cursor description. The value of the nth column in the SELECT list is assigned to the nth output destination in the INTO clause.

indicator_variable

Name of the indicator variable for the preceding host variable. If the host variable is a vector, the indicator variable must also be a vector with the same number of elements.

The indicator value indicates whether the NULL value was transferred or whether data was lost:

0	The host variable contains the value read. The assignment was error free.
-1	The value to be assigned is the NULL value.
> 0	For alphanumeric and national values: The host variable was assigned a truncated string. The value of the indicator variable indicates the original length in code units.

descriptor

For a dynamic cursor.

Name of an SQL descriptor area containing the data type description of the output values and into which the output values read with the FETCH statement are written.

The SQL descriptor area must be created beforehand and supplied with appropriate values:

- The value of the COUNT field must be the same as the number of output values, which is calculated as follows: Number of atomic derived columns plus number of column elements of each multiple derived column. The following also applies:

  0 <= COUNT <= defined maximum number of item descriptors
- The output values are assigned to the DATA fields of the item descriptors in the order of the items in the descriptor area. The data type description for an item must be compatible with the data type of the corresponding output value (see section “Reading values into host variables or a descriptor area”).

  If the value to be transferred is the NULL value, the appropriate INDICATOR field is set to the value -1. If a string to be assigned is truncated, the
  corresponding INDICATOR field indicates the original length.

If block mode is activated for the open cursor *cursor*, and if a FETCH NEXT... statement has been executed whose INTO clause contains the name of another SQL descriptor area, you receive an error message.

Behavior of SESAM/SQL in the event of an error

If an error occurs when a value is read (e.g. value is the NULL value, but the indicator variable is not specified; numeric value is too big for the target data type), the cursor is moved to its new position but the assigned values are undefined.

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

Examples

Position the cursor CUR_ORDER on a row in the cursor table and read the column values of the current row in the host variables ORDER_NUM, CUST_NUM, CONTACT_NUM, ORDER_TEXT, TARGET and ORDER_STAT.
Using the indicator variables IND_CONTACT_NUM, IND_ORDER_TEXT and IND_ACTUAL, check whether information has been lost in the transfer of the alphanumeric values and whether any of the columns contain the NULL value.

```
FETCH cur_order
INTO  :ORDER_NUM,
      :CUST_NUM,
      :CONTACT_NUM INDICATOR :IND_CONTACT_NUM,
      :ORDER_TEXT  INDICATOR :IND_ORDER_TEXT,
      :ACTUAL      INDICATOR :IND_ACTUAL,
      :ORDER_STAT
```

Position the cursor CUR_RESULT on a row in the cursor table and read the column values in the descriptor area DESCR_AREA.

```
FETCH cur_result INTO SQL DESCRIPTOR GLOBAL 'DESCR_AREA'
```

See also

CLOSE, DECLARE CURSOR, DELETE, OPEN, STORE, UPDATE

8.2.3.37 FOR - Execute SQL statements in a loop

The FOR-statement executes SQL statements in a loop over all records of an implicitly defined cursor. Cursor operations (e.g. FETCH) are not required here. Nor may they be used for the implicitly defined cursor. The implicitly defined cursor is automatically closed when processing has been concluded.

The ITERATE statement enables you to switch immediately to the next loop pass. The loop can be aborted by means of a LEAVE statement.

The FOR statement may only be specified in a routine, i.e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. Routines and their use in SESAM/SQL are described in detail in chapter “Routines”.

The FOR statement is a non-atomic SQL statement, i.e. further (atomic or non-atomic) SQL statements can occur in it.

If the FOR statement is part of a COMPOUND statement, in the case of corresponding exception handling routines the loop can also be left when a particular SQLSTATE (e.g. no data, class '02xxx') occurs.

[*label* :]

FOR [*forloopname* AS] [*cursor* CURSOR FOR] *query_expression*

  DO *routine_sql_statement;* [*routine_sql_statement;*] . . .

END FOR [*label*]

forloopname ::= *unqual_name*

---

label

The label in front of the FOR statement (start label) indicates the start of the loop. It may not be identical to another label in the loop.

The start label need only be specified when the next loop pass is to be switched to using ITERATE or when the loop is to be left using a LEAVE statement. However, it should always be used to permit SESAM/SQL to check that the routine has the correct structure (e.g. in the case of nested loops).

The label at the end of the FOR statement (end label) indicates the end of the loop. If the end label is specified, the start label must also be specified. Both labels must be identical.

forloopname

Name of the FOR loop. It can be used to qualify the names of the columns of the subsequent cursor description.
forloopname may be up to 31 characters long.

cursor

Optional name for the cursor defined by *query_expression*.
This name must be specified if UPDATE ... WHERE CURRENT OF ... or DELETE ... WHERE CURRENT OF ... is to be used for the cursor or function.

query_expression

Definition of the cursor which is to be processed by the FOR statement.
The cursor must have unambiguously named columns. This can always be achieved by using correlation names.
The data types of the cursor's output values may not be multiple. However, individual occurrences of a multiple field can be used.

routine_sql_statement

SQL statement which is to be executed in the FOR statement.
An SQL statement is concluded with a ";" (semicolon).
Multiple SQL statements can be specified one after the other. They are executed in the order specified.
No privileges are checked before an SQL statement is executed.
An SQL statement in a routine may access the parameters of the routine and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

The syntax and meaning of *routine_sql_statement* are described centrally in section “SQL statements in routines”. The SQL statements named there may not be used.

> i The SQL statements *update_positioned_statement* and *delete_positioned_statement* can also be executed for the corresponding cursor if *cursor* is specified and the *query_expression* is updatable (see section “Rules for updatable query expressions”).

Execution information

The FOR statement is a non-atomic statement:

- If the FOR statement is part of a COMPOUND statement, the rules described there apply, in particular the exception routines defined there.
- If the FOR statement is **not** part of a COMPOUND statement and one of the SQL statements reports an SQLSTATE, it is possible that only the updates of this statement will be undone. The FOR statement and the routine in which it is contained are aborted. The SQL statement in which the routine was used returns the SQLSTATE concerned.

Areas of validity and precedence rules for names

- In the case of an unqualified name (*unqual_name*), first an existing routine parameter or an existing local variable is used with this name. Otherwise the name is searched for in the current statement. If this name also does not exist there, the name (in the case of nested FOR statements) is searched for in the higher-ranking FOR statements “from the inside out”.

- It is recommended that you define a name for the FOR loop (*forloopname*), see below. This makes name references within FOR loops clear. Precedence rules need not then be observed.

Name for a FOR loop:

In the SQL statements of the FOR statement, the current values can be referred to using the column names of the cursor description.

However, it is clearer if a name is defined for the FOR loop (*forloopname*). This name can be used to qualify the columns of the current row:

```
FOR F1 AS SELECT C001, C002 FROM T1 WHERE P < 127
DO
   UPDATE TU
   SET COLX = COLX + F1.C001 WHERE COLY = F1.C002;
END FOR
```

This becomes apparent in a nested FOR statement:

```
FOR F1 AS SELECT C001 FROM T1 WHERE P < 127
DO
    FOR F2 AS SELECT C001, C002 FROM T2 WHERE Q < 875
    DO
        UPDATE TU
               SET COLX = COLX + F1.C001 + F2.C001
               WHERE COLY < F2.C002;
    END FOR;
END FOR
```

See also

CREATE PROCEDURE, CREATE FUNCTION, ITERATE, LEAVE

8.2.3.38 GET DESCRIPTOR - Read SQL descriptor area

You use GET DESCRIPTOR to read the values from the fields in an SQL descriptor area.

You must create the descriptor with ALLOCATE DESCRIPTOR, and it must be supplied with values before you call GET DESCRIPTOR.

```
GET DESCRIPTOR GLOBAL descriptor
    { :host_variable =COUNT |
      VALUE item_number : host_variable = field_id
      [ , host_variable = field_id ] . . . }

item_number ::= { integer | :host_variable }

field_id ::=
{
   REPETITIONS |
   TYPE |
   DATETIME_INTERVAL_CODE |
   PRECISION |
   SCALE |
   LENGTH |
   INDICATOR |
   DATA |
   OCTET_LENGTH |
   NULLABLE |
   NAME |
   UNNAMED
}
```

descriptor

Name of the SQL descriptor area whose item descriptors are to be read.

host_variable=COUNT

Host variable of the type SMALLINT into which the value of the COUNT field is entered.

item_number

Number of the item descriptor in the SQL descriptor area containing the fields to be read. The items in the descriptor area are numbered sequentially starting with 1. You can specify an integer or a host variable for *item_number*, where:
1 <= *item_number* <= defined maximum number of item descriptors

If *item_number* > COUNT, an appropriate SQLSTATE is set, which can be handled with WHENEVER NOT FOUND.

host_variable=field_id

Host variable into which the value of the specified field of the item descriptor *item_number* is entered. The SQL data type of the variable depends on the specified field identifier.

field_id

Field in the item descriptor *item_number* that is to be read. The descriptor area fields are described in the section “Descriptor area fields”. You may specify a *field_id* more than once in a GET DESCRIPTOR statement.

If a value is transferred from a descriptor area field to a host variable, the host variable must be of the type SMALLINT for all of the fields except NAME and DATA.

If the value of the NAME field is to be transferred, the host variable must be of the SQL data type CHAR(*n*) or VARCHAR(*n*), where *n* >= 128.
If the value of the DATA field is to be transferred to a host variable, the host variable must have exactly the same SQL data type indicated by the fields TYPE, DATETIME_INTERVAL_CODE, LENGTH, PRECISION, SCALE of the same item (see section “Transferring values between host variables and a descriptor area”).

Except for the DATA and INDICATOR fields, no vectors can be specified. If DATA and INDICATOR are specified, both must be atomic values or vectors with the same number of elements.

If a vector with several elements is specified, the item numbers for exactly the same number of subsequent items must be <= the defined maximum number of item descriptors. If item numbers > COUNT, an appropriate SQLSTATE is set, which can be handled with WHENEVER NOT FOUND.

GET DESCRIPTOR reads the last value set for the specified field. If the value of the field is undefined, the value returned is also undefined.

The following applies to the DATA field: If the value of the INDICATOR field of the same item < 0, the GET DESCRIPTOR statement must also include the INDICATOR field and only the INDICATOR field is assigned a value.

If vectors are specified, the appropriate number of items are read starting with *item_number*.

Examples

Read the name, data type and length in bytes of the third item descriptor in the SQL descriptor area DEMO_DESC:

```
GET DESCRIPTOR GLOBAL :demo_desc

VALUE 3 :desc_name = NAME :desc_type = TYPE :desc_len = OCTET_LENGTH
```

Query the number of item descriptors in the SQL descriptor area:

```
GET DESCRIPTOR GLOBAL :demo_desc :desc_count = COUNT
```

See also

ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DESCRIBE, SET DESCRIPTOR

8.2.3.39 GET DIAGNOSTICS - Output diagnostic information

GET DIAGNOSTICS ascertains information on an SQL statement executed beforehand in a routine and enters this in a procedure parameter of the type INOUT or OUT or a local variable. The information relates to the statement itself of to the database objects affected by it.

GET DIAGNOSTICS changes neither the content nor the sequence of diagnostics areas. In other words GET DIAGNOSTICS statements which follow each other evaluate the same diagnostic information.

GET DIAGNOSTICS is one of the diagnostic statements, see “Diagnostic information in routines”.

```
GET [CURRENT | STACKED] DIAGNOSTICS
     { statement_info [, statement_info ] ...] |
       CONDITION condition_info [, condition_info ]... }


statement_info ::= name1 = ROW_COUNT


condition_info ::= name2 =
{
   CONDITION_IDENTIFIER |
   RETURNED_SQLSTATE |
   MESSAGE_TEXT |
   MESSAGE_LENGTH |
   MESSAGE_OCTET_LENGTH
}


name1, name2 ::= { local_variable | routine_parameter }
```

CURRENT

The diagnostic information for the SQL statement most recently executed is output.

Normally this statement is used to output the diagnostic information of an SQL statement executed without error.

However, the SQL statement can also have executed an exception routine with exception handling CONTINUE after an SQLSTATE (see “Local exception routines”), and GET DIAGNOSTICS is the next statement in the routine.A local exception routine has its own diagnostics area. CURRENT outputs the diagnostic information of the SQL statement executed most recently in the exception routine. The diagnostic information of the initiating SQL statement is output with STACKED.

STACKED

The diagnostic information of the SQL statement whose SQLSTATE triggered the exception routine is output. STACKED may be specified only in a local exception routine.

name1, *name2*

name1 and *name2* are the names of local variables, or procedure or UDF parameters in which the information written after the equals sign is entered.
The data type of *name1* or *name2* must be compatible with the data type of the information to be entered. The rules in section “Entering values in a procedure parameter (output) or local variable” apply.

name1=ROW_COUNT

name1 is assigned the number of processed rows of the subsequent successfully executed SQL statement: *insert_statement*, *update_searched_statement*,
delete_searched_statement, *merge_statement*. Otherwise the value is undefined.

Data type: DECIMAL(31)

name2=CONDITION_IDENTIFIER

name2 is, if necessary, assigned the name of the condition reported by a SIGNAL or RESIGNAL statement. Otherwise a string with the length 0 is assigned.

Data type: VARCHAR(31)

name2=RETURNED_SQLSTATE

name2 is, if necessary, assigned the value of the reported SQLSTATE. Otherwise a string with the length 0 is assigned.

Data type: VARCHAR(5)

name2=MESSAGE_TEXT

name2 is, if necessary, assigned the message text if MESSAGE_TEXT was specified in the SIGNAL or RESIGNAL statement. Otherwise a string with the length 0 is assigned.

Data type: VARCHAR(120)

name2=MESSAGE_LENGTH

name2 is, if necessary, assigned the length of the message text if MESSAGE_TEXT was specified in the SIGNAL or RESIGNAL statement. Otherwise the value 0 is assigned.

Data type: INTEGER

name2=MESSAGE_OCTET_LENGTH

name2 is, if necessary, assigned the length of the message text in bytes if MESSAGE_TEXT was specified in the SIGNAL or RESIGNAL statement. Otherwise the value 0 is assigned.

Data type: INTEGER

Examples (see also "Diagnostic information in routines" **)**

Outputting diagnostic information of the last SQL statement:

```
GET CURRENT DIAGNOSTICS counter1=ROW_COUNT;
```

Outputting diagnostic information of the SQL statement which triggered the exception routine:

```
GET STACKED DIAGNOSTICS CONDITION
    var1=RETURNED_SQLSTATE,
    var2=MESSAGE_LENGTH, var3=MESSAGE_TEXT;
```

See also

COMPOUND, CREATE FUNCTION, CREATE PROCEDURE, RESIGNAL, SIGNAL

8.2.3.40 GRANT - Grant privileges

GRANT assigns the following privileges:

- Table and column privileges for base tables and views
- Special privileges for databases and storage groups
- EXECUTE privileges for routines

If the GRANT statement is included in a CREATE SCHEMA statement, you cannot grant special privileges with GRANT.

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

- It is the authorization identifier of the universal user.
- It is owner of the table, database, storage group or routine.
- It has GRANT authorization for granting the privileges to other users.

Information on which authorization identifiers are owners is stored in the SCHEMATA, SPACES and STOGROUPS views. The TABLE_PRIVILEGES, COLUMN_PRIVILEGES, USAGE_PRIVILEGES, CATALOG_PRIVILEGES and ROUTINE_PRIVILEGES views provide you with information on whether the authorization identifier has GRANT authorization for a certain privilege (see chapter “Information schemas”).

Only the authorization identifier that granted a privilege (the “grantor”) can revoke that privilege.

The GRANT statement has several formats. Examples are provided under the format concerned.

See also

REVOKE, CREATE SCHEMA

GRANT format for table and column privileges

```
GRANT { ALL PRIVILEGES | table_and_column_privileg ,...}
      ON [TABLE] table
      TO { PUBLIC | authorization_identifier } ,...
      [WITH GRANT OPTION]
```

table_and_column_privilege ::=

```
{
  SELECT |
  DELETE |
  INSERT |
  UPDATE [( column,... )] |
  REFERENCES [( column,... )]
}
```

ALL PRIVILEGES

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

table_and_column_privilege

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

ON [TABLE] t*able*

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

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

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

TO PUBLIC

The privileges are granted to all authorization identifiers. In addition to its own privileges, each authorization identifier also has those which have been granted to PUBLIC. Authorization identifiers added later also have these privileges.

TO *authorization_identifier*

The privileges are granted to *authorization_identifier*. You may specify more than one authorization identifier.

WITH GRANT OPTION

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

WITH GRANT OPTION omitted:

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

table_and_column_privilege

Specification of the individual table and column privileges.

SELECT

Privilege that allows rows in the table to be read.

DELETE

Privilege that allows rows to be deleted from the table.

INSERT

Privilege that allows rows to be inserted into the table.

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

Privilege that allows rows in the table to be updated.

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

(*column*,...) omitted:

All columns in the table may be updated. Columns added later may also be updated.

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

Privilege that allows the definition of referential constraints that reference the table. The reference can be limited to the specified columns. *column* must be the name of a column in the specified table. You can specify more than one column.

(*column*,...) omitted:

All columns in the table may be referenced. Columns added later may also be referenced.

Example

Grant all table privileges for IMAGES to the authorization identifier UTIUSR1, and the table privileges SELECT, DELETE, INSERT, and UPDATE for DESCRIPTIONS to the authorization identifier UTIUSR2. The two authorization identifiers must be created beforehand.

```
GRANT ALL PRIVILEGES ON images TO utiusr1
```

```
GRANT SELECT, DELETE, INSERT, UPDATE ON descriptions TO utiusr2
```

GRANT format for special privileges

```
GRANT { ALL SPECIAL PRIVILEGES | special_privilege , . . .}
      ON CATALOG { catalog | STOGROUP stogroup }
      TO { PUBLIC | authorization_identifier } , . . .
      [WITH GRANT OPTION]

special_privilege ::=
  {
    CREATE USER |
    CREATE SCHEMA |
    CREATE STOGROUP |
    UTILITY |
    USAGE
  }
```

ALL SPECIAL PRIVILEGES

All the special privileges that the current authorization identifier may grant are granted. ALL SPECIAL PRIVILEGES comprises the special privileges.

special_privilege

The special privileges are granted individually. You can specify more than one special privilege.

ON CATALOG *catalog*

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

ON STOGROUP *stogroup*

Name of the storage group for which you want to grant the USAGE privilege. You can qualify the name of the storage group with a database name.

TO

See "GRANT - Grant privileges".

WITH GRANT OPTION

See "GRANT - Grant privileges".

special_privilege

Specification of the individual special privileges.

CREATE USER

Special privilege that allows you to define and delete authorization identifiers. You can only grant the CREATE USER privilege for a database.

CREATE SCHEMA

Special privilege that allows you to define database schemas. You can only grant the CREATE SCHEMA privilege for a database.

CREATE STOGROUP

Special privilege that allows you to define storage groups. You can only grant the CREATE STOGROUP privilege for a database.

UTILITY

Special privilege that allows you to use utility statements. You can only grant the UTILITY privilege for a database.

USAGE

Special privilege that allows you to use a storage group. You can only grant the USAGE privilege for a storage group.

Examples

Grant the special privilege CREATE SCHEMA to the existing authorization identifier UTIUSR.

```
GRANT CREATE SCHEMA ON CATALOG ordercust TO utiusr
```

Grant all special privileges for the database ORDERCUST to the authorization identifier UTIADM.

In addition, grant UTIADM the special privilege which authorizes use of the storage group STOGROUP1.

```
GRANT ALL SPECIAL PRIVILEGES ON CATALOG ordercust TO utiadm
GRANT USAGE ON STOGROUP stogroup1 TO utiadm
```

GRANT format for EXECUTE privileges (procedure)

```
GRANT EXECUTE ON SPECIFIC PROCEDURE procedure
    TO { PUBLIC | authorization_identifier },...
    [WITH GRANT OPTION]

procedure ::= routine
```

EXECUTE ON SPECIFIC PROCEDURE *procedure*

Name of the procedure for which the privilege is to be passed on. You can qualify the procedure name with a database and schema name. If you use the GRANT statement in a CREATE SCHEMA statement, you can only qualify the procedure name with the database and schema name from the CREATE SCHEMA statement.

TO

See "GRANT - Grant privileges".

WITH GRANT OPTION

See "GRANT - Grant privileges".

Example

The privilege of being entitled to execute the `myproc` procedure is granted to all authorization identifiers.

```
GRANT EXECUTE ON SPECIFIC PROCEDURE myproc TO PUBLIC
```

GRANT format for EXECUTE privileges (UDF)

```
GRANT EXECUTE ON SPECIFIC FUNCTION udf
      TO { PUBLIC | authorization_identifier },...
      [WITH GRANT OPTION]

udf ::= routine
```

EXECUTE ON SPECIFIC FUNCTION *udf*

Name of the UDF for which the privilege is to be passed on. You can qualify the unqualified UDF name with a database and schema name. If you use the GRANT statement in a CREATE SCHEMA statement, you may qualify the UDF name only with the database and schema names from the CREATE SCHEMA statement.

TO

See "GRANT - Grant privileges".

WITH GRANT OPTION

See "GRANT - Grant privileges".

Example

The privilege of being entitled to execute the `myproc` UDF is granted to all authorization identifiers.

```
GRANT EXECUTE ON SPECIFIC FUNCTION myudf TO PUBLIC
```

8.2.3.41 IF - Execute SQL statements conditionally

The IF statement executes statements depending on certain conditions. It may may only be specified in a routine, i. e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. Routines and their use in SESAM /SQL are described in detail in chapter "Routines".

The IF statement is a non-atomic SQL statement, i.e. further (atomic or non-atomic) SQL statements can occur in it.

```
IF search_condition
   THEN routine_sql_statement; [ routine_sql_statement; ]...
   [ELSEIF search_condition THEN routine_sql_statement; [ routine_sql_statement; ]...]...
   [ELSE routine_sql_statement; [ routine_sql_statement; ]... ]
END IF
```

search_condition

Search condition that returns a truth value when evaluated
The search condition may contain parameters of routines and (if the statement is part of a COMPOUND statement) local variables, but no host variables.
A column may be specified only if it is part of a subquery

routine_sql_statement

SQLstatement which is to be executed in the THEN or ELSE part of the IF statement. An SQL statement is concluded with a ";" (semicolon).
Multiple SQL statements can be specified one after the other. They are executed in the order specified.
No privileges are checked before an SQL statement is executed.
An SQL statement in a routine may access the parameters of the routine and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

The syntax and meaning of *routine_sql_statement* are described centrally in section "SQL statements in routines". The SQL statements named there may not be used.

Execution information

The IF and ELSEIF clauses are processed from left to right. The associated SQL statements are processed for the first THEN clause whose search condition returns the truth value true. The IF statement is then terminated.

If none of the search conditions returns the truth value true and an ELSE clause exists, the SQL statements of the ELSE clause are processed.

SQL statements are not processed if the associated search condition returns the truth value unknown.

The IF statement is a non-atomic statement:

- If the IF statement is part of a COMPOUND statement, the rules described there apply, in particular the exception routines defined there.

- If the IF statement is **not** part of a COMPOUND statement and one of the SQL statements reports an SQLSTATE, it is possible that only the updates of this SQL statement will be undone. The IF statement and the routine in which it is contained are aborted. The SQL statement in which the routine was used returns the SQLSTATE concerned.

Example

The SQL statements are executed only if the `tab` table is not empty.

```
IF (SELECT COUNT(*) FROM tab) > 0 THEN routine_sql_statement END IF
```

See also

CREATE PROCEDURE, CREATE FUNCTION

8.2.3.42 INCLUDE - Insert program text into ESQL programs

You use INCLUDE to insert program text stored in a PLAM library member into an ESQL program. The program text can contain embedded SQL statements and utility statements, as well as statements in the host language. For example, you could use INCLUDE to insert the communication area between SQL and the host language in an ESQL program, provided that an appropriate member exists in a BS2000 PLAM library.

During precompilation by the ESQL precompiler, the INCLUDE statement is replaced by the text in the specified library member. The INCLUDE statements are processed in the order in which they occur in the program.

`INCLUDE` *library_member*

library_member `::=` { *alphanumeric_literal* | *regular_name* }

---

library_member

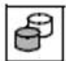
Name of a PLAM library member of the type S. The name must be the valid name of a PLAM library member without a suffix (version specification). If several versions of the specified library member exist in a PLAM library, SESAM/SQL uses the current version.

Allocating PLAM libraries with ESQL precompiler options

Each PLAM library that contains library members must be made known by means of an ESQL precompiler option (see the “ ESQL-COBOL for SESAM/SQL-Server” manual). You use these options to determine the order in which PLAM libraries are searched for library members. If two library members with the same name exist in different PLAM libraries, the ESQL precompiler always uses the first PLAM library encountered that contains this library member.

Example

Insert the library element VARIABLES in an ESQL program. VARIABLES could contain frequently used host variables, for example.



INCLUDE variables

8.2.3.43 INSERT - Insert rows in table

You use INSERT to insert rows into an existing table.

If you want to insert rows into a table, you must either own the table or have the INSERT privilege for the table. Furthermore, the transaction mode of the current transaction must be READ WRITE.

The special literals (see "Special literals") which occur in the INSERT statement (and in preset values) and the time functions CURRENT_DATE, CURRENT_TIME and CURRENT_TIMESTAMP are evaluated once, and the calculated values apply for all inserts.

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

```
INSERT INTO table
        { [ column_list ] [COUNT INTO column ] value-declaration | DEFAULT VALUES }
        [RETURN INTO parameter-declaration ]


value-declaration ::= { query_expression | VALUES { row_2 , row_2 , ... | row_1 } }


parameter-declaration ::=
{
   :host_variable [[INDICATOR] :indicator_variable ] |
   routine_parameter |
   local_variable
}


column_list ::= (
{
   column |
   column[posno] |
   column[min .. max] |
   column ( posno ) |
   column ( min .. max )
},...
)


row_2 ::= { ( insert_expression_2 ,... ) | insert_expression_2 }
```

row_1 ::= { (*insert_expression_1* ,...) | *insert_expression_1* }

insert_expression_2 ::= { *expression* | NULL }

insert_expression_1 ::= { *expression* | NULL | DEFAULT | * | <{ *value* | NULL},...> }

table

Name of the table into which the rows are to be inserted. The table can be a base table or an updatable view.

column_list

Lists the columns, and the occurrence ranges of multiple columns, for which the INSERT statement specifies the values in the rows to be inserted, and stipulates the order for this. The values of the remaining columns in the rows to be inserted are not specified in the INSERT statement; they are DEFAULT or NULL values or values defined by SESAM/SQL.

No *column_list* specified:
The INSERT statement specifies the values in the rows to be inserted for each column of *table* (except for the column specified by COUNT INTO), in the order specified with CREATE TABLE and ALTER TABLE or with CREATE VIEW.

column

Atomic column whose values in the rows to be inserted are specified in the INSERT statement.

column must be a column of the specified table. The order in which you specify the columns does not have to be the same as the order of the columns in the table. You can specify an atomic column only once in the column list.

column(pos_no) /column[pos_no]

Element of a multiple column whose values in the rows to be inserted are specified in the INSERT statement. The multiple column must be part of the table.

If several elements of a multiple column are specified, the range of indexes specified must be contiguous. None of the elements of the multiple column may occur more than once.

pos_no is an unsigned integer >= 1.

column(min..max) / column[min..max]

Elements in a multiple column whose values are indicated in the rows to be inserted in the INSERT statement. The multiple column must be part of the table.

If several elements of a multiple column are specified, the range of indexes specified must be contiguous. None of the elements of the multiple column may occur more than once.

min and *max* are unsigned integers >= 1; *max* must be >= *min*.

> i Any square brackets shown here in italics are special characters, and must be specified in the statement.

COUNT INTO *column*

Atomic column whose values in the rows to be inserted are determined by SESAM/SQL and must not be specified in the INSERT statement (counting column). *column* may not occur in *column_list*.

The column must be of integer or fixed-point number type (SMALLINT, INT, DECIMAL, NUMERIC) and must belong to the primary key. The column may not be contained either in a referential constraint or a check constraint of the table *table*.

SESAM/SQL determines the values of the respective column in all rows to be inserted in such a way that the primary key values are unique within the table.

query_expression

query expression is a query expression whose derived table specifies the required column values of the rows to be inserted. One row is inserted into the table *table* for each row of the result table. If *query expression* returns an empty table, no rows are inserted and an appropriate SQLSTATE is set, which can be handled with WHENEVER NOT FOUND.

VALUES clause

The required column values are specified separately for each row which is to be inserted using *line_2* or *line_1* . The table consisting of all these rows or of this one row plays the same role as the result table of the *query_expression*.

row_2

The number of rows inserted is the number of times *row_2* is specified.

All occurrences of *row_2* must have the same number of columns. The data type of each column of the result table follows from the rules described under “Data type of the derived column for UNION”. If a column in the result table only contains NULL, its data type will be that of the corresponding column of *table*.

insert_expression_2

expression

The *expression* of *insert_expression_2* must be atomic.

NULL

The corresponding column in the rows to be inserted must be atomic. It is set to the NULL value.

row_1

One row is inserted. The result table with the required values for this row to be inserted consists of *row_1*.

insert_expression_1

expression

The *expression* of *insert_expression_1* must be either atomic or a host variable which is a vector with more than one element. If such a host variable or an aggregate is specified, the number of vector or aggregate elements must agree with the number of elements of the respective column in the row to be inserted.

NULL

The respective column in the row to be inserted must be atomic. It is set to the NULL value.

DEFAULT

The respective column in the row to be inserted must be atomic. It is set to the default value. The default value is specified in the definition of the column. If there is no default value defined, the column is set to the NULL value.

*

The corresponding column in the row to be inserted must be atomic, must be of integer or fixed-point number type (SMALLINT, INT, DECIMAL, NUMERIC) and must belong to a primary key. The column may not be contained either in a referential constraint or a check constraint of the table *table*.

* may occur only once in the VALUES clause and must not occur together with COUNT INTO.

The value of the corresponding column in the row to be inserted is determined by SESAM/SQL in such a way that the primary key values within the table are unique.

<{*value*, NULL},...>

Aggregate to be assigned to a multiple column. The number of values must be the same as the number of column elements.

Query_expression, *insert_expression_2* and *insert_expression_1* must not reference a table referring to the underlying base table into which the new rows are inserted. In particular, you may not reference *table*.

The number of columns of *query_expression*, *row_2* and *row_1* must equal the number of column values to be specified for each inserted row, as specified with *column list* and COUNT INTO. The i-th column of the result table contains the values for the i-th column in *column list* (if *column list* is specified), or for the i-th column of *table* (where a column specified with COUNT INTO is skipped).

The assignment rules specified in section "Entering values in table columns" apply to these assignments.

Any remaining columns of the inserted rows are set as follows:

- The column specified by COUNT INTO is set to a value defined by SESAM/SQL.
- Columns with a default value are set to the default value (DEFAULT).
- Columns without a default value are set to the NULL value.

If *table* is a view, the rows will be inserted into the underlying base table; columns of the base table not contained in the view will be set in the same way.

DEFAULT VALUES

Inserts one row into the table; the row consists entirely of the column-specific default values.

Columns with an explicitly defined default value are assigned this default value. Columns without explicitly defined default are assigned the NULL value.

RETURN INTO

The value determined by SESAM/SQL for the column specified with COUNT INTO or for * as *insert_expression_1* is stored in an output destination. If several rows are inserted, the last value determined by SESAM/SQL will be stored.

You can only use the RETURN INTO clause, if either COUNT INTO is specified, or a * is used as *insert_expression_1*.

:host_variable, *routine_parameter*, *local_variable*

Name of a host variable (if the statement is **not** part of a routine) or name of a procedure parameter of the type INOUT or OUT or of a local variable (if the statement is part of a routine). The value of the count column is assigned to the specified output destination.

The output destination must be of numeric data type.

indicator_variable

Name of the indicator variable for the preceding host variable.

Inserting values for multiple columns

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

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

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

> ! **CAUTION!**The position of an element in a multiple column can differ from the position of the corresponding element as specified in the INSERT statement. If an element of a multpIe column is set to the NULL value, all elements with higher position number are shifted “left” by decreasing their position number by one, and the element set to NULL gets the highest position number.

INSERT and integrity constraints

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

INSERT and transaction management

INSERT initiates an SQL transaction outside routines if no transaction is open. If you define an isolation level, you can control how the INSERT statement can be affected by concurrent transactions (see section “SET TRANSACTION - Define transaction attributes”).

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

Examples

Each of the following two statements inserts three rows into the ORDERS table. In the second INSERT statement, the value for the primary key is determined by SESAM/SQL. The last value assigned is stored in the host variable ORDNUMRET.

```
INSERT INTO orders (order_num,cust_num,contact_num,order_text,actual,orderstat)
   VALUES (345, 101, 20, 'Network:installation', DATE'<date>', 1),
          (346, 101, 20, 'Network:installation', DATE'<date>', 1),
          (347, 101, 20, 'Network:installation', DATE'<date>', 1),

INSERT INTO orders (cust_num,contact_num,order_text,actual,orderstat)
   COUNT INTO order_num
   VALUES  (:CUST_NUM,:CONTACT_NUM,:ORDERTEXT1,:ACTUAL1,:ORDERSTAT),
           (:CUST_NUM,:CONTACT_NUM,:ORDERTEXT1,:ACTUAL1,:ORDERSTAT),
           (:CUST_NUM,:CONTACT_NUM,:ORDERTEXT1,:ACTUAL1,:ORDERSTAT),
   RETURN INTO :ORDNUMRET
```

In a table with the name WOMEN, the columns FNAME and LNAME are defined in the same way as in the table CONTACTS. The following INSERT statement adds all female contacts to the WOMEN table:

```
INSERT INTO women (fname, lname)
   SELECT fname, lname
   FROM contacts
   WHERE title IN ('Frau','Fraeulein','Mrs.','Ms.')
```

See also

DELETE, MERGE, UPDATE

8.2.3.44 ITERATE - Switch to the next loop pass

The ITERATE statement switches to the next loop pass.
It may may only be specified in the control statements FOR, LOOP, REPEAT, and WHILE of a routine, i.e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. Routines and their use in SESAM/SQL are described in detail in chapter “Routines”.

`ITERATE` *label*

---

label

Label of the FOR, LOOP, REPEAT, or WHILE statement which contains the ITERATE statement. The current loop pass is terminated. The next loop pass is switched to.

Specifying *label* also enables outer loop passes to be terminated. Inner loop passes are then terminated immediately.

Example

When the value of variable i is divisible by 3, ITERATE causes the next loop pass to be switched to immediately.

```
DECLARE i INTEGER DEFAULT 1;
   ...
   label:
   REPEAT
       SET i = i + 1;
       IF MOD(i,3)=0 THEN ITERATE label;
       END IF;
       ...
       UNTIL i >100
   END REPEAT label;
```

See also

CREATE PROCEDURE, CREATE FUNCTION, FOR, LOOP, REPEAT, WHILE

8.2.3.45 LEAVE - Terminate a loop or COMPOUND statement

The LEAVE statement terminates a loop or COMPOUND statement.
It may may only be specified in the control statements COMPOUND, FOR, LOOP, REPEAT, and WHILE of a routine, i.e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. Routines and their use in SESAM/SQL are described in detail in chapter “Routines”.

`LEAVE` *label*

---

label

Label of the COMPOUND, FOR, LOOP, REPEAT, or WHILE statement which contains the LEAVE statement. The statement identified is terminated.

Example

See the LOOP statement example on "LOOP - Execute SQL statements in a loop".

See also

CREATE PROCEDURE, CREATE FUNCTION, FOR, LOOP, REPEAT, WHILE

8.2.3.46 LOOP - Execute SQL statements in a loop

The LOOP statement executes SQL statements in a loop.

The ITERATE statement enables you to switch immediately to the next loop pass. The loop can be aborted by means of a LEAVE statement.

The LOOP statement may only be specified in a routine, i.e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. Routines and their use in SESAM/SQL are described in detail in chapter “Routines”.

The LOOP statement is a non-atomic SQL statement, i.e. further (atomic or non-atomic) SQL statements can occur in it.

If the LOOP statement is part of a COMPOUND statement, in the case of corresponding exception routines the loop can also be left when a particular SQLSTATE (e.g. no data, class 02xxx) occurs.

`[` *label* `:]`

`LOOP`

  routine_sql_statement; `[` *routine_sql_statement;* `]...`

`END LOOP [` *label* `]`

---

label

The label in front of the LOOP statement (start label) indicates the start of the loop. It may not be identical to another label in the loop.

The start label need only be specified when the next loop pass is to be switched to using ITERATE or when the loop is to be left using a LEAVE statement. However, it should always be used to permit SESAM/SQL to check that the procedure has the correct structure (e.g. in the case of nested loops).

The label at the end of the LOOP statement (end label) indicates the end of the loop. If the end label is specified, the start label must also be specified. Both labels must be identical.

routine_sql_statement

SQL statement which is to be executed in the LOOP statement.
An SQL statement is concluded with a ";" (semicolon).
Multiple SQL statements can be specified one after the other. They are executed in the order specified.
No privileges are checked before an SQL statement is executed.
An SQL statement in a routine may access the parameters of the routine and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

The syntax and meaning of *routine_sql_statement* are described centrally in section “SQL statements in routines”. The SQL statements named there may not be used.

Execution information

The LOOP statement is a non-atomic statement:

- If the LOOP statement is part of a COMPOUND statement, the rules described there apply, in particular the exception routines defined there.
- If the LOOP statement is **not** part of a COMPOUND statement and one of the SQL statements reports an SQLSTATE, it is possible that only the updates of this statement will be undone. The LOOP statement and the routine in which it is contained are aborted. The SQL statement in which the routine was used returns the SQLSTATE concerned.

Example

A loop is canceled after 1000 passes by means of LEAVE.

```
DECLARE i INTEGER DEFAULT 0;
   ...
   label:
   LOOP
       SET i = i+1;
       IF i > 1000 THEN LEAVE label;
       ...
   END LOOP label;
```

See also

CREATE PROCEDURE, CREATE FUNCTION, ITERATE, LEAVE

8.2.3.47 MERGE - Insert rows in a table or update column values

You use MERGE to unite the INSERT and UPDATE functions in one operation. Depending on the result of the constraint in the ON clause, MERGE updates column values of rows which already exist (WHEN MATCHED THEN) or inserts new rows into an existing table (WHEN NOT MATCHED THEN).

This constraint can range from a simple existence query to complex search criteria. Trivial constraint (e.g. `1 <> 1`) are also possible; they lead to it only being possible to update or insert rows.

The special literals (see "Special literals") which occur in the INSERT statement (and in preset values) and the time functions CURRENT_DATE, CURRENT_TIME and CURRENT_TIMESTAMP are evaluated once, and the calculated values apply for all inserts.

If integrity constraints are defined for the table or the columns concerned, these are checked after the insertion or update operation. If an integrity constraint is violated, the inserts and updates are undone and a corresponding SQLSTATE is set.

Specific requirements must be satisfied to execute the MERGE statement:

- To insert or update rows in *table* you must
  - be the owner of *table* or
  - at least have the INSERT privilege when *insert_row* is specified or
  - at least have the UPDATE privilege for all columns which are updated in *update_row* when *update_row* is specified.
- You must also have the SELECT privilege for all tables which are addressed in *table_specification*.
- The transaction mode of the current transaction must be READ WRITE.

```
MERGE INTO table [[AS] correlation_name ]
      USING table_specification
      ON search_condition
      { WHEN MATCHED THEN update_row |
        WHEN NOT MATCHED THEN insert_row }...


update_row ::= UPDATE SET column = column-value [ , column-value ]...


column-value ::= { expression | DEFAULT | NULL}


insert_row ::= INSERT [( column ,...)][COUNT INTO column ] VALUES( value [ ,value ] ... )


value ::= { expression | NULL | DEFAULT | * }
```

---

table

Name of the destination table into which rows are to be inserted or in which rows are to be updated. The destination table can be a base table or an updatable view. It may not contain any multiple columns (see note on "MERGE - Insert rows in a table or update column values").

correlation_name

Table name used in the statement as a new name for the table *table*.

The *correlation_name* must be used to qualify the column name in every column specification that references the table *table* if the column name is not unambiguous.

The new name must be unique, i.e. *correlation_name* can only occur once in a table specification of this statement.

You must give a table a new name if the columns in the table cannot be identified otherwise uniquely.

In addition, you may give a table a new name in order to formulate an expression so that it is more easily understood or to abbreviate long names.

USING *table_specification*

Specifies a source table (different from the destination table *table*) which is to be used to insert rows into the destination table *table* or update rows in the destination table *table*. It may not contain any multiple columns (see note on "MERGE - Insert rows in a table or update column values").
The destination table *table* may also be referenced in the *table_specification*.

ON *search_condition*

Specifies the condition which decides whether the UPDATE clause is to be executed (result: TRUE) or whether the INSERT clause is to be executed (result: FALSE).

More precisely, each row of the source table is checked to see whether there is a row in the destination table so that the *search_condition* is true for the combination of these two rows.
If no such row exists in the destination table, the INSERT clause is executed, i.e. the row in the source table is inserted in the destination table.
If one or more such rows exist in the destination table, the UPDATE clause is executed for each of these rows, i.e. the corresponding rows in the destination table are updated. Two different rows in the source table may not lead to updates in the destination table (multiple update), otherwise the MERGE statement is aborted with SQLSTATE.

WHEN MATCHED THEN *update_row*

A row which is to be updated was found.

UPDATE SET ...

Information for updating the row which is to be updated.
For a description of the *column*, *expression*, `DEFAULT` and `NULL` parameters, see the corresponding descriptions in the SQL statement UPDATE on "UPDATE - Update column values".A row which is to be updated may only be updated once. Any further attempt to update it is rejected with SQLSTATE.
The primary key value in a partitioned table may not be modified.

WHEN NOT MATCHED THEN *insert_row*

No corresponding row was found. The new row is to be inserted.

INSERT ...

Information for inserting the new row.

(column,...)

Lists the columns, for which the INSERT clause of the MERGE statement specifies the values and stipulates the order for this. The values of the remaining columns in the row to be inserted are not specified in the MERGE statement; they are DEFAULT or NULL values or values defined by SESAM/SQL.

For a description of the *column* parameter, see the corresponding description in the SQL statement INSERT on "INSERT - Insert rows in table".

No *column_list* specified:
The MERGE statement specifies the values in the row to be inserted for each column of the target table *table* (except for the column specified by COUNT INTO), in the order specified with CREATE TABLE and ALTER TABLE or with CREATE VIEW.

COUNT INTO *column*

See the corresponding description in the SQL statement INSERT on "INSERT - Insert rows in table".

VALUES (...)

The required column values are specified for the row which is to be inserted.

For a description of *expression*, `NULL`, `DEFAULT` and `*`, see the description of *insert_expression_1* in the SQL statement INSERT on "INSERT - Insert rows in table".

In *expression* you may not specify a table which references the destination table into which the new rows are to be inserted.
In particular you may not reference any column of the destination table.

The number of columns of the VALUES clause must equal the number of column values to be specified for each inserted row, as specified with *(column,...)* and COUNT INTO. The nth column of the destination table contains the values for the nth column specification in *(column,...)* (if *(column,...)* is specified), or for the nth column of *table*, where any column of *table* introduced with COUNT INTO is not counted.

The assignment rules specified in section “Entering values in table columns” apply to these assignments.

Any remaining columns of the inserted rows are set as follows:

- The column specified by COUNT INTO is set to a value defined by SESAM/SQL.
- Columns with a default value are set to the default value (DEFAULT).
- Columns without a default value are set to the NULL value.

If the target table *table* is a view, the rows will be inserted into the underlying base table; columns of the base table not contained in the view will be set in the same way.

Inserting values for multiple columns

Base tables with multiple columns cannot be processed directly in the MERGE statement.

> However, if you specify an (updatable) view with, for example, the query expression SELECT *cloumn_list* FROM *table* defined without multiple columns in *cloumn_list*, the MERGE statement can be executed with this.

MERGE and integrity constraints

By specifying integrity constraints when you define the base table, you can restrict the range of values for the corresponding columns. The value specified in the MERGE statement must satisfy the defined integrity constraint, otherwise the MERGE statement is aborted with SQLSTATE.

MERGE and transaction management

MERGE initiates an SQL transaction outside routines if no transaction is open. If you define an isolation level, you can control how the MERGE statement can be affected by concurrent transactions (see section "SET TRANSACTION - Define transaction attributes").

If an error occurs during the MERGE statement, any updates already performed by the MERGE statement are canceled.

Examples

The example below concerns inventory management when a new delivery arrives. In the case of the existiing articles with the same price the inventory in the base table is updated. New articles in the delivery table are added to the inventory table.

```
MERGE INTO inventory AS b USING delivery AS l
      ON b.article_no = l.article_no AND b.article_price = l.article_price
      WHEN MATCHED THEN
             UPDATE SET article_quant = b.article_quant + l.article_quant
      WHEN NOT MATCHED THEN
                    INSERT (article_no,article_price,article_quant)
                    VALUES (l.article_no,l.article_price,l.article_quant)
```

The complex example below also concerns inventory management when a new delivery arrives. The data for the new delivery is, for example, supplied in the CSV input file DELIVERY.DATA (with a header):

```
Article number,Quantity,New price
1, 4, 18.50
2, 11, 19.90
3, 0, 22.95
4, 3, 84.30
5, 7, 25.90
```

The MERGE statement below updates the *inventory* table for the articles which already exist. New articles in the delivery are added to the inventory table. The header of the CSV file is skipped by means of the WITH ORDINALITY clause in conjunction with WHERE.

```
MERGE INTO  inventory
      USING (SELECT CAST(article number as INT),
                    CAST(quantity as INT),
                    CAST(new price as NUMERIC(10,2))
            FROM TABLE(CSV('DELIVERY.DATA'
                             DELIMITER ',' QUOTE '"' ESCAPE '\',
                             varchar(30), varchar(40), varchar(50)))
                   WITH ORDINALITY
             AS T(article number, quantity, new price, counter)
             WHERE counter > 1)
             AS delivery(article number, quantity, new price)
      ON inventory.article number= delivery.article number
      WHEN MATCHED THEN UPDATE SET
                        quantity = inventory.quantity + delivery.quantity,
                        price = delivery.new price
      WHEN NOT MATCHED THEN INSERT (article number, quantity, price)
                            VALUES(delivery.article number,
                                   delivery.quantity, delivery.new price)
```

See also

DELETE, INSERT, UPDATE

8.2.3.48 OPEN - Open cursor

You use OPEN to open a cursor declared with DECLARE CURSOR .

- The host variables in the cursor description or the values for placeholders in a dynamic cursor description are evaluated.
- The special literals (see "Special literals"), as well as the time functions CURRENT_DATE, CURRENT_TIME and CURRENT_TIMESTAMP are evaluated.

  All the values returned contain the same date and/or time (see section “Time functions”). These values are valid for the cursor table as long as the cursor is open and if the cursor is reopened with RESTORE.

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

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

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

The cursor must be closed.

```
OPEN cursor
     [USING { variable [ , variable ]... | SQL DESCRIPTOR GLOBAL descriptor } ]

variable ::= :host_variable [ [INDICATOR] :indicator_variable ]
```

cursor

Name of the cursor to be opened.

USING clause

For a dynamic cursor.

Specifies where the input values for the dynamic cursor description are to be read from. You must specify the USING clause if the cursor description includes question marks as placeholders for values.

host_variable

Name of a host variable containing the value to be assigned to a placeholder in the dynamic cursor description.

The data type of a host variable must be compatible with the data type of the corresponding placeholder (see section “Values for placeholders”). If the placeholder represents an aggregate with several elements, the corresponding host variable must be a vector with the same number of elements.

The number of host variables specified must be the same as the number of placeholders in the cursor description. The host variables are assigned values in the order in which the placeholders are specified in the dynamic cursor description.

indicator_variable

Name of the indicator variable for the preceding host variable. If the host variable is a vector, the indicator variable must also be a vector with the same number of elements.

The value of the indicator variable indicates whether the NULL value is to be transferred:

< 0	The NULL value is to be assigned.
>= 0	The value of the host variable is to be assigned.

descriptor

Name of an SQL descriptor area containing the data types and values for the placeholders in the dynamic cursor description.

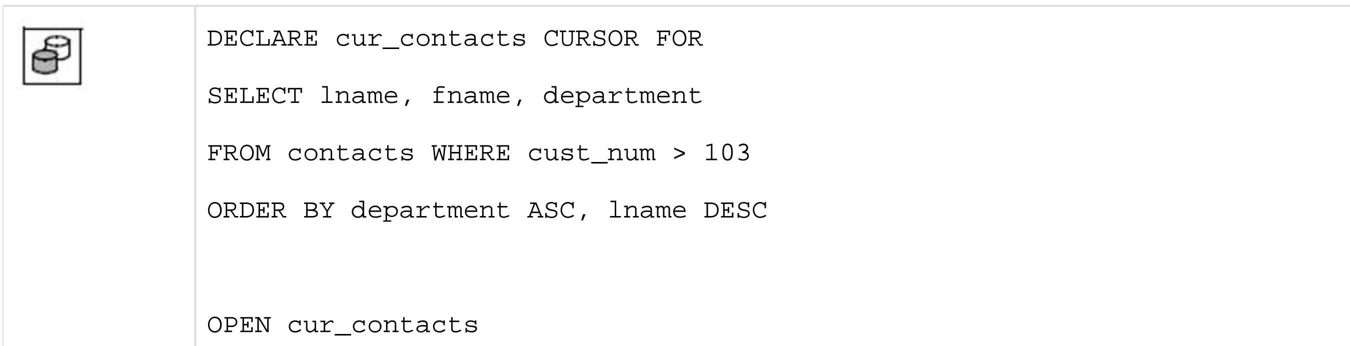
The SQL descriptor area must be created beforehand and supplied with appropriate values:

- The value of the COUNT descriptor area feld must be the same as the number of required input values (for aggregates, one output value for each element) where

  0 <= COUNT <= defined maximum number of item descriptors
- The values of the DATA fields of the item descriptors (or NULL values if the INDICATOR is negative) are assigned to the placeholders in the dynamic statement in the order of the items in the descriptor area. The data type description of an item must be compatible with the data type of the corresponding placeholder (see section “Values for placeholders” "Values for placeholders").

Example

Open a cursor CUR_CONTACTS. The cursor defines a section of the CONTACTS table containing the LNAME, FNAME and DEPARTMENT for all customers with customer numbers greater than 103.



```
DECLARE cur_contacts CURSOR FOR

SELECT lname, fname, department

FROM contacts WHERE cust_num > 103

ORDER BY department ASC, lname DESC


OPEN cur_contacts
```

See also

CLOSE, DECLARE CURSOR, FETCH, PREPARE

8.2.3.49 PERMIT - Specify user identification for SESAM/SQL V1.x

In order to allow programs created with SESAM/SQL V1.x to run without you having to make changes to them, the PERMIT statement is still allowed. Execution of a PERMIT statement does not, however, have any effect. A SESAM /SQL V1.x program can only be executed successfully under the current version of SESAM/SQL if the appropriate privileges have been defined with GRANT.

The PERMIT statement does not initiate a transaction.

See also

GRANT, REVOKE

8.2.3.50 PREPARE - Prepare dynamic statement

You use PREPARE to prepare a dynamic statement or the cursor description of a dynamic cursor for execution at a later time.

You execute a statement prepared with PREPARE with the EXECUTE statement.

The statement identifier used in PREPARE for a cursor description is used to declare a dynamic cursor with DECLARE CURSOR. You open the dynamic cursor with OPEN.

`PREPARE` *statement_id* `FROM` *statement_variable*

statement_variable `::=` `:` *host_variable*

---

statement_id

Name of the dynamic statement or cursor description. You can use this name to reference the statement or cursor description in the compilation unit.

statement_variable

Alphanumeric host variable containing the statement text. The host variable can also be of the type CHAR(*n*), where 256 <= *n* <= 32000.

The following conditions must be satisfied:

- The statement text cannot include any host variables. Question marks are specified as placeholders for unknown values (see “Rules for placeholders”). The placeholders are supplied with values in the USING clause of an EXECUTE or OPEN statement.
- The statement text may not contain comments in the host language. Pragmas (--%PRAGMA) are exceptions.
- A SELECT statement cannot include an INTO clause.
- The RETURN INTO clause cannot be specified in an INSERT statement. The CLI call SQL_DIAG SEQ_GET is available to allow you to use the function you probably know from static INSERT statements. It enables you to simulate RETURN INTO (see "SQL_DIAG_SEQ_GET - SQLdsg").

If *statement_id* is defined for a dynamic cursor, but the statement is not a cursor description, an error is reported. The statement is prepared successfully despite this fact and can be executed with EXECUTE.

Rules for placeholders

A placeholder for an input value in a dynamic statement is represented by a question mark. You can specify a placeholder if the operands and operators associated with the placeholder uniquely define the data type of the placeholder.

Below you will find a summary of the positions permitted or not permitted for placeholders grouped according to whether a monadic or dyadic operator, a range or element query or a pattern comparison is involved, as well as the positions permitted or not permitted for CASE expressions, CAST expressions, numeric functions, string functions, SELECT list, INSERT, UPDATE and MERGE. The data type of a placeholder is also specified for permitted placeholders.

If a placeholder is not permitted at a certain position, this also applies even if the placeholder is enclosed in parentheses.

Example

```
not permitted: (?)+(?)
```

Monadic operators

No placeholders are permitted for monadic operators. The following cases are therefore not permitted:

- The operand of a monadic operator cannot be a placeholder (e.g. -?).
- The operand for IS [NOT] NULL cannot be a placeholder (e.g.? IS NULL).
- The argument of an aggregate function cannot be a placeholder (e.g. AVG(?)).

Dyadic operators

In the case of dyadic operators, only one of the operands can be a placeholder.

Example

```
permitted: ?+1, ?<100, p=?

not permitted: ?=?
```

Data type of the placeholder

If one of the operands for concatenation is a placeholder (?||"..." or "..."||?), the data type of the placeholder is VARCHAR(32000) or NVARCHAR(16000).

For all other dyadic operators, the data type of the placeholder is the same as the data type of the other operand.

Range query

If the first operand in a range query is a placeholder, neither of the other two operands can be a placeholder.

Example

```
permitted: ? BETWEEN 100 AND 500
        50 BETWEEN ? AND ?
not permitted: ? BETWEEN 100 AND ?
```

Data type of the placeholder

The data type of the placeholder is derived from the data types of the values of the other operands which are not placeholders (see "Data type of the placeholder in CASE, BETWEEN and IN").

Element query

In an element query, neither the first operand nor any of the elements in the list may be placeholders.

Example

```
permitted: ? IN ('Frankfurt','Munich','Hamburg')
           x IN (?,'Munich','Hamburg')
           ? IN ('Frankfurt',?,?)
           x IN (?,?,?)
           ? NOT IN (SELECT order_num FROM service WHERE order_text='Training')
not permitted: ? IN (?,?,?)
```

Data type of the placeholder

- If the first operand is a placeholder and the second operand is a subquery, the data type of the placeholder is the same as the data type of the derived column.
- If the first operand is a placeholder and the second operand is a list of expressions, the data type of the placeholder is derived from the data types of the elements in the list that are not placeholders (see "Data type of the placeholder in CASE, BETWEEN and IN").
- If an element in the list is a placeholder and the first element is not a placeholder, the data type of the placeholder is the same as the data type of the first operand.

Pattern comparison

In a pattern comparison, the second and third operand may be placeholders.

Example

```
permitted: x LIKE ? ESCAPE ?

not permitted: ? LIKE y ESCAPE ?
```

Data type of the placeholder

The data type of the placeholder is VARCHAR(32000) or NVARCHAR(16000).

CASE expression

Not all the operands in a CASE expression may be placeholders. If the CASE expression contains one or more THEN or ELSE clauses, not all the operands in these clauses can be placeholders. The following cases are therefore not permitted:

- In a simple CASE expression, the first operand (expression after CASE) is a placeholder and the operand in the WHEN clause is a placeholder or - if there are several WHEN clauses - all the operands in the WHEN clauses are placeholders.
- In a simple CASE expression, all the THEN clauses and the ELSE clause contain placeholders.

  Example

```
permitted: CASE ?
          WHEN 1 THEN 10
          WHEN 2 THEN 20
          WHEN ? THEN 30
          WHEN ? THEN 30
          ELSE 50 END
not permitted: CASE ?
                WHEN ? THEN 10
                WHEN ? THEN 20
                WHEN ? THEN 30
                WHEN ? THEN 30
                ELSE 50 END
not permitted: CASE x
                WHEN 1 THEN ?
                WHEN 2 THEN ?
                ELSE ? END
```

- In a CASE expression with a search condition, all the THEN clauses and the ELSE clause contain placeholders.

  Example

```
permitted: CASE
         WHEN ord_stat_num= 1 THEN ?
         WHEN ord_stat_num= 2 THEN ?
         WHEN ord_stat_num > 2 AND ord_stat_num < 5 THEN ?
         ELSE 50 END
not permitted: CASE
               WHEN ord_stat_num= 1 THEN ?
               WHEN ord_stat_num= 2 THEN ?
               WHEN ord_stat_num > 2 AND ord_stat_num < 5 THEN ?
               ELSE ? END
```

- In a CASE expression with NULLIF, both operands are placeholders (e.g. NULLIF (?,?))
- In a CASE expression with COALESCE, all the operands are placeholders (e.g. COALESCE (?,?,?))

Data type of the placeholder

The data type of the placeholder in a CASE expression depends on the data types of the other operands which are not placeholders.

If an operand of a CASE expression with NULLIF is a placeholder, its data type corresponds to the data type of the other operand.

If several of the other operands are without placeholders, the following rules apply:

- If the first operand of a simple CASE expression is a placeholder and/or if the CASE expression contains one or more placeholders as operands in its WHEN clause or clauses, its data type is derived from the data types of the other operands which are not placeholders and not operands of the THEN or ELSE clause(s).
- If a CASE expression with a search condition or a simple CASE expression contains placeholders in the THEN clause(s) and/or the ELSE clause, its data type is derived from that of the other THEN or ELSE clause operands which are not placeholders.
- If an operand of a CASE expression with COALESCE is a placeholder, its data type is derived from the data types of the other operands which are not placeholders.

The rules described in “Data type of the placeholder in CASE, BETWEEN and IN” apply to the calculation of the placeholder data type.

CAST expression

No restrictions

Data type of the placeholder

The data type of the placeholder in a CAST expression corresponds to the data type of the result value of the CAST expression.

Numeric functions

In the numeric function POSITION, both operands cannot be placeholders (e.g. POSITION (? IN ?)).

Data type of the placeholder

The data type of the placeholders in the numeric functions POSITION, OCTET_LENGTH and CHAR_LENGTH is VARCHAR(32000) or NVARCHAR(16000).

For the numeric function JULIAN_DAY_OF_DATE, the data type of the placeholder is DATE.

String functions

The following are not permitted in string functions:

- In the string functions LOWER and UPPER, the operands cannot be placeholders.
- In the string function TRIM, the first operand (*character*) and/or the second operand (*expression*) cannot be placeholders (e.g. TRIM (TRAILING FROM ?)).
- In the string function SUBSTRING, the first operand cannot be a placeholder (e.g. SUBSTRING ? FROM 1 FOR 5)).

Data type of the placeholder

In the string function SUBSTRING, the data type of the placeholder is NUMERIC(31,0).

Time functions

No restrictions

Data type of the placeholder

For the time function DATE_OF_JULIAN_DAY the data type of the placeholder is INTEGER.

SELECT (list)

In a SELECT expression, an element in the SELECT list may not consist of only one placeholder.

Example

```
permitted: SELECT 3+? FROM ...
```

```
not permitted: SELECT ?,x,p FROM ...
```

INSERT, UPDATE, MERGE

You can specify a placeholder as the column value of an atomic column and for an element in a multiple column.

Example

```
permitted: INSERT INTO tab (x, ...) VALUES (?, ...)
           INSERT INTO t (x) VALUES <..., ?, ...>
           UPDATE tab SET x=?
           UPDATE t SET x=<..., ?, ...>
```

Data type of the placeholder

The data type of the placeholder is the data type of the column. In the case of a multiple column with a dimension > 1, the placeholder is also multiple with the same dimension. Otherwise, the placeholder is atomic.

Data type of the placeholder in CASE, BETWEEN and IN

In CASE expressions, area queries and element queries, the data type of the placeholder is derived in some cases from the data types of the other operands or elements which are not placeholders. In these cases, the following rules apply:

- All the values of the other operands have the data type NCHAR:
  The value of the placeholder has the data type NCHAR with the greatest length.
- At least one value of the other operands has the data type VARCHAR: The value of the placeholder is that with the data type VARCHAR and the greatest or greatest maximum length.
- All the values of the other operands have the data type NCHAR:
  The value of the placeholder has the data type NCHAR with the greatest length.
- At least one value of the other operands has the data type NVARCHAR: The value of the placeholder is that with the data type NVARCHAR and the greatest or greatest maximum length.
- All values of the other operands are an integer or fixed-point type (INT, SMALLINT, NUMERIC, DEC):
  The value of the placeholder has the data type integer or fixed-point number.
  - The number of decimal places is the greatest number of decimal places among the different values of the other operands.
  - The total number of places is the greatest number of places before the decimal point plus the greatest number of decimal places among the different values of the other operands, but not more than 31.
- At least one value of the other operands is of the type floating-point number (REAL, DOUBLE PRECISION, FLOAT); the others have any other numeric data type: The value of the placeholder has the data type DOUBLE PRECISION.
- All the values of the other operands have the time data type:
  The value of the placeholder also has this data type.

Converting the placeholder data type using CAST

The rules for placeholders sometimes result in an undesired data type for a particular placeholder. You can avoid undesired data types by using the CAST expression (see section “CAST expression”).

Example

In the following dynamic UPDATE statement, the placeholder represents a single-digit integer:

```
UPDATE t SET x=?+1
```

If, in the USING clause of the EXECUTE statement, the value 10 is specified for the placeholder, execution is not successful.

An UPDATE statement can be formulated to avoid this data type assignment:

```
UPDATE t SET x=CAST(? AS DEC(5,0))
```

Procedures

A procedure can be called using a dynamic CALL statement. If a procedure contains parameters of the type OUT or INOUT, the corresponding arguments must be specified in a dynamic CALL statement in the form of placeholders.

Assignments for PREPARE

The following SQL statement can be prepared with PREPARE:

ALTER SPACE
ALTER STOGROUP
ALTER TABLE
CALL
COMMIT
CREATE INDEX
CREATE FUNCTION
CREATE PROCEDURE
CREATE SCHEMA
CREATE SPACE
CREATE STOGROUP
CREATE SYSTEM_USER
CREATE TABLE
CREATE USER
CREATE VIEW
DELETE

DROP FUNCTION
DROP INDEX
DROP PROCEDURE
DROP SCHEMA
DROP SPACE
DROP STOGROUP
DROP SYSTEM_USER
DROP TABLE
DROP USER
DROP VIEW
GRANT
INSERT (without RETURN INTO clause)
MERGE
PERMIT
REORG STATISTICS
REVOKE
ROLLBACK
SELECT (without INTO clause)
SET CATALOG
SET SCHEMA
SET SESSION AUTHORIZATION
SET TRANSACTION
UPDATE

Additionally to these SQL statements, dynamic cursor descriptions and all the utility statements can also be prepared with PREPARE (see the " SQL Reference Manual Part 2: Utilities").

The following statements cannot be prepared with PREPARE:

ALLOCATE DESCRIPTOR
CLOSE
DEALLOCATE DESCRIPTOR

DECLARE CURSOR
DESCRIBE
EXECUTE
EXECUTE IMMEDIATE
FETCH
GET DESCRIPTOR
INCLUDE
OPEN
PREPARE
RESTORE
SET DESCRIPTOR
STORE
WHENEVER

Validity period of a prepared statement

An SQL statement prepared with PREPARE remains prepared for execution at least until the end of the current transaction. After the end of the transaction, you should prepare the statement again. If the plan buffer of the DBH still contains the access plan of the SQL statement contained in *statement_variable*, SESAM/SQL uses the existing access plan.

A statement prepared with PREPARE is lost if PREPARE is executed using the same *statement_id* in the same compilation unit and the same SQL session.

The prepared statement is also lost if the statement contains a reference to a dynamic cursor and the prepared cursor description for this cursor is lost.

Example

Prepare a description of the dynamic cursor CUR_SERVICE1 for subsequent execution. The contents of the host variable DESCRIPTION are defined using actions of the ESQL program host language.

```
DECLARE cur_service1 CURSOR FOR cur_description

PREPARE cur_description FROM :DESCRIPTION
```

See also

DECLARE CURSOR, EXECUTE, FETCH, OPEN

8.2.3.51 REORG STATISTICS - Regenerate global statistics

You use REORG STATISTICS to re-generate global statistics on the distribution of values over the columns in an index. These statistics are used to optimize table accesses with search conditions and should be updated whenever extensive changes are made to the data.

The current authorization identifier must either be the owner of the schema to which the index belongs or must have the special privilege UTILITY for the database to which the index belongs.

```
REORG STATISTICS FOR INDEX index
```

index

Name of the index for which the statistics are to be re-generated.

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

See also

CREATE INDEX

8.2.3.52 REPEAT - Execute SQL statements in a loop

The REPEAT statement executes SQL statements in a loop until the specified condition is satisfied. The loop ends with the condition being checked, i.e. it is executed at least once.

The ITERATE statement enables you to switch immediately to the next loop pass. The loop can be aborted by means of a LEAVE statement.

The REPEAT statement may only be specified in a routine, i.e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. Routines and their use in SESAM/SQL are described in detail in chapter "Routines"
.

The REPEAT statement is a non-atomic SQL statement, i.e. further (atomic or non-atomic) SQL statements can occur in it.

If the REPEAT statement is part of a COMPOUND statement, in the case of corresponding exception routines the loop can also be left when a particular SQLSTATE (e.g. no data, class 02xxx) occurs.

[*label* :]

REPEAT *routine_sql_statement;* [*routine_sql_statement;*] . . .

    UNTIL *search_condition*

END REPEAT [*label*]

---

label

The label in front of the REPEAT statement (start label) indicates the start of the loop. It may not be identical to another label in the loop.

The start label need only be specified when the next loop pass is to be switched to using ITERATE or when the loop is to be left using a LEAVE statement. However, it should always be used to permit SESAM/SQL to check that the routine has the correct structure (e.g. in the case of nested loops).

The label at the end of the REPEAT statement (end label) indicates the end of the loop. If the end label is specified, the start label must also be specified. Both labels must be identical.

search_condition

Search condition that returns a truth value when evaluated
The search condition is the stop criterion for the loop.

routine_sql_statement

SQL statement which is to be executed in the REPEAT statement.
An SQL statement is concluded with a ";" (semicolon).
Multiple SQL statements can be specified one after the other. They are executed in the order specified.

No privileges are checked before an SQL statement is executed.
An SQL statement in a routine may access the parameters of the routine and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

The syntax and meaning of *routine_sql_statement* are described centrally in section “SQL statements in routines”. The SQL statements named there may not be used.

Execution information

The REPEAT statement is a non-atomic statement:

- If the REPEAT statement is part of a COMPOUND statement, the rules described there apply, in particular the exception routines defined there.
- If the REPEAT statement is **not** part of a COMPOUND statement and one of the SQL statements reports an SQLSTATE, it is possible that only the updates of this statement will be undone. The REPEAT statement and the routine in which it is contained are aborted. The SQL statement in which the routine was used returns the SQLSTATE concerned.

Example

The loop is executed until the variable i has the value.

```
DECLARE i INTEGER DEFAULT 0;
...
label:
   REPEAT
       SET i= i+2;
       ...
   UNTIL i >1000
   END REPEAT
label;
```

See also

CREATE PROCEDURE, CREATE FUNCTION, ITERATE, LEAVE

8.2.3.53 RESIGNAL - Report exception in local exception routine

RESIGNAL explicitly reports an exception or an SQLSTATE in a local exception routine. In contrast to SIGNAL, the specification of an exception name or SQLSTATE is optional.

RESIGNAL uses the diagnostics area of the SQL statement which has activated the exception routine as the current diagnostics area, and enters corresponding diagnostic information in the current diagnostics area.

RESIGNAL is one of the diagnostic statements. Detailed information on the use and effect of RESIGNAL can be found in section "Diagnostic information in routines".

`RESIGNAL [` *error_name* | *sqlstate* `] [SET` *diagnostic_info* `]`

sqlstate `::= SQLSTATE [VALUE]` *alphanumeric_literal*

diagnostic_info `::= MESSAGE_TEXT=` *message*

message `::={` *alphanumeric_literal* | *local_variable* | *routine_parameter* }

---

exception_name

Name of an exception or SQLSTATE. *exception_name* is defined in the local data of a routine, see "Local data"
.

sqlstate

Explicit specification of a self-defined SQLSTATE (alphanumeric literal with the length 5), see section "Self-defined SQLSTATEs".

exception_name and *sql_state* not specified:

The diagnostic information CONDITION_IDENTIFIER and RETURNED_SQLSTATE remains unchanged.

MESSAGE_TEXT=*alphanumeric_literal*

Any information (maximum length: 120 characters).

MESSAGE_TEXT=*local_variable / routine_parameter*

The value of the local variable or of the specified routine parameter is entered as information text.
The data type of *local_variable / routine_parameter* must be compatible with the data type VARCHAR(120).
The rules in section "Entering values in a procedure parameter (output) or local variable" apply. The text length is entered in MESSAGE_LENGTH and MESSAGE_OCTET_LENGTH.

SET MESSAGE TEXT omitted:

The diagnostic information MESSAGE_TEXT, MESSAGE_LENGTH and MESSAGE_OCTET_LENGTH remains unchanged.

Examples (see also "Diagnostic information in routines" **)**

Reporting a condition with information text:

```
RESIGNAL SET MESSAGE_TEXT='The end is near!';
```

See also

COMPOUND, CREATE FUNCTION, CREATE PROCEDURE, GET DIAGNOSTICS, SIGNAL

8.2.3.54 RESTORE - Restore cursor

You use RESTORE to open a cursor saved with STORE.

The cursor is opened with the same cursor description as for the last OPEN. If host variables have been updated in the meantime, this does not have any effect on the resulting derived table.

If special literals or the time functions CURRENT_DATE, CURRENT_TIME and/or CURRENT_TIMESTAMP are included in the cursor description, they are not reevaluated.

A cursor position saved with STORE can be lost if, in the same or a different transaction, rows starting at the stored position have been deleted in the meantime, or the row on which the cursor was positioned has been updated in such a way that it no longer belongs to the cursor table.

If no cursor position has been saved for the cursor, the cursor is not opened and an appropriate SQLSTATE is set.

Otherwise, the cursor is opened and the cursor position restored. If you want to delete (DELETE ... WHERE CURRENT OF) or update (UPDATE ... WHERE CURRENT OF) a row, the cursor must be positioned on the row with FETCH.

After the RESTORE statement has been executed, all the information on this cursor that has been saved with STORE is deleted. You must save the cursor position again with store before a new RESTORE statement can be executed.

The cursor to be restored must be saved with STORE and must be closed when RESTORE is executed. The transactions containing the STORE and RESTORE statements must have the same isolation level.

For a dynamic cursor, the cursor description must be still be prepared when the RESTORE statement is executed (see also “Validity period of a prepared statement”).

RESTORE must not be used for cursors defined with WITH HOLD.

```
RESTORE cursor
```

cursor

Name of the cursor to be restored.

Processing the cursor after RESTORE

After a RESTORE statement, you must position the cursor on a row with FETCH.

Example

FETCH NEXT positions to the next row in the cursor table.

Only then can the cursor be accessed with an UPDATE or DELETE statement.

See also

DECLARE CURSOR, OPEN, STORE, FETCH, UPDATE, DELETE

8.2.3.55 RETURN - Supply the return value of a User Defined Function (UDF)

RETURN supplies the return value of a UDF. The data type of the return value is defined by the RETURNS clause of the CREATE FUNCTION statement.

The RETURN statement may only be specified in the definition of a UDF with CREATE FUNCTION. UDFs and their use in SESAM/SQL are described in detail in chapter “Routines”.

A RETURN statement terminates the execution of a UDF directly.
If a UDF is not terminated with a RETURN statement, this results in an error in the calling SQL statement.

`RETURN` { *expression* | `NULL` }

---

expression

Expression whose value is assigned to the return value of the UDF.
The expression may contain routine parameters and (if the statement is part of a COMPOUND statement) local variables, but no host variables.
A column may be specified only if it is part of a subquery.
The data type of *expression* must be compatible with the data type of the RETURNS clause from the CREATE FUNCTION statement.
The rules in section “Entering values in a procedure parameter (output) or local variable” apply.

NULL

The return value of the UDF is the NULL value.

See also

CREATE FUNCTION

8.2.3.56 REVOKE - Revoke privileges

REVOKE revokes the following privileges from authorization identifiers:

- Table and column privileges
- Special privileges
- EXECUTE privileges for routines

Only the authorization identifier that granted a privilege (the “grantor”) can revoke that privilege from an authorization identifier (see section “GRANT - Grant privileges”).

The TABLE_PRIVILEGES, COLUMN_PRIVILEGES, USAGE_PRIVILEGES, CATALOG_PRIVILEGES and ROUTINE_PRIVILEGES tables of the INFORMATION_SCHEMA provide you with information on the privileges assigned to the authorization identifiers (see chapter “Information schemas”).

The REVOKE statement has several formats. Examples are provided under the format concerned.

See also

GRANT

REVOKE format for table and column privileges

```
REVOKE { ALL PRIVILEGES | table_and_column_privilege ,...}
       ON [TABLE] table
       FROM { PUBLIC | authorization_identifier } ,...
       { RESTRICT | CASCADE }

table_and_column_privilege ::=
{
   SELECT |
   DELETE |
   INSERT |
   UPDATE [( column ,...)] |
   REFERENCES [( column ,...)]
}
```

ALL PRIVILEGES

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

table_and_column_privilege

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

ON [TABLE] *table*

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

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

FROM PUBLIC

The privileges are revoked from all authorization identifiers. The individual privileges of the individual authorization identifiers are not affected.

FROM *authorization_identifier*

The privileges are revoked from the user with the authorization identifier *authorization_identifier*. You may specify more than one authorization identifier.

CASCADE

An authorization identifier can revoke any privileges it has granted:

- All the specified privileges are revoked.
- If a specified privilege has been forwarded to other authorization identifiers, all privileges forwarded directly or indirectly are deleted.
- Views which were defined either directly or indirectly on the basis of the specified or forwarded privileges are deleted.
- Referential constraints defined on the basis of the specified and forwarded privileges are deleted.
- Routines which were defined either directly or indirectly on the basis of the specified and forwarded privileges are deleted.

RESTRICT

The following restrictions apply to the revoking of privileges:

- A privilege forwarded to other authorization identifiers cannot be revoked for as long as a forwarded privilege like this still exists.

- A privilege on the basis of which a view or referential constraint has been defined cannot be revoked if the view or referential constraint still exists.
- A privilege on the basis of which a routine has been defined cannot be revoked if the routine still exists.

table_and_column_privilege

Specification of the individual table and column privileges.

SELECT

Privilege that allows rows in the table to be read.

DELETE

Privilege that allows rows to be deleted from the table.

INSERT

Privilege that allows rows to be inserted into the table.

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

Privilege that allows rows in the table to be updated.

The revoke operation can be limited to the specified columns.

column must be the name of a column in the specified table. You can specify more than one column.

(*column*,...) omitted: The privilege for updating all the columns in the table is revoked.

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

Privilege that allows the definition of referential constraints that reference the table.

The revoke operation can be limited to the specified columns.

column must be the name of a column in the specified table. You can specify more than one column.

(*column*,...) omitted: The privilege for referencing all the columns in the table is revoked.

REVOKE format for special privileges

```
REVOKE { ALL SPECIAL PRIVILEGES | special_privilege ,...}
       ON CATALOG { catalog | STOGROUP stogroup }
       FROM { PUBLIC | authorization_identifier } ,...
       { RESTRICT | CASCADE }
```

special_privilege ::=

```
{
  CREATE USER |
  CREATE SCHEMA |
  CREATE STOGROUP |
  UTILITY |
  USAGE
}
```

ALL SPECIAL PRIVILEGES

All the special privileges that the current authorization identifier may revoke are revoked. ALL SPECIAL PRIVILEGES revokes the special privileges.

special_privilege

The special privileges are revoked individually. You can specify more than one special privilege.

ON CATALOG *catalog*

Name of the database for which you want to revoke special privileges.

ON STOGROUP *stogroup*

Name of the storage group for which you want to revoke the USAGE privilege. You can qualify the name of the storage group with a database name.

FROM *authorization_identifier*

The privileges are revoked from the user with the authorization identifier *authorization_identifier*. You may specify more than one authorization identifier.

CASCADE

An authorization identifier can revoke any privileges it has granted:

- All the specified privileges are revoked.
- If a specified privilege has been forwarded to other authorization identifiers, all forwarded privileges are deleted implicitly.

RESTRICT

The following restrictions apply to the revoking of privileges:

- A privilege forwarded to other authorization identifiers cannot be revoked for as long as a forwarded privilege like this still exists.

special_privilege

Specification of the individual special privileges.

CREATE USER

Special privilege that allows you to define authorization identifiers. You can only revoke the CREATE USER privilege for a database.

CREATE SCHEMA

Special privilege that allows you to define database schemas. You can only revoke the CREATE SCHEMA privilege for a database.

CREATE STOGROUP

Special privilege that allows you to define storage groups. You can only revoke the CREATE STOGROUP privilege for a database.

UTILITY

Special privilege that allows you to use utility statements. You can only revoke the UTILITY privilege for a database.

USAGE

Special privilege that allows you to use a storage group. You can only revoke the USAGE privilege for a storage group.

Example

Revoke the UPDATE privilege for all columns in the table DESCRIPTIONS from the authorization identifier UTIUSR2.

REVOKE UPDATE ON TABLE descriptions FROM utiusr2 RESTRICT

REVOKE format for EXECUTE privileges (procedure)

```
REVOKE EXECUTE ON SPECIFIC PROCEDURE procedure
        FROM { PUBLIC | authorization_identifier },...
        { RESTRICT | CASCADE }

procedure ::= routine
```

EXECUTE ON SPECIFIC PROCEDURE *procedure*

Name of the procedure for which the privilege is to be revoked. You can qualify the procedure name with a database and schema name.

FROM *authorization_identifier*

The privileges are revoked from the user with the authorization identifier *authorization_identifier*. You may specify more than one authorization identifier.

CASCADE

An authorization identifier can revoke any privileges it has granted:

- All the specified privileges are revoked.
- If a specified privilege has been forwarded to other authorization identifiers, all privileges forwarded directly or indirectly are deleted.
- Views which were defined either directly or indirectly on the basis of the specified or forwarded privileges are deleted.
- Routines which were defined either directly or indirectly on the basis of the specified or forwarded privileges are deleted.

RESTRICT

The following restrictions apply to the revoking of privileges:

- A privilege forwarded to other authorization identifiers cannot be revoked for as long as a forwarded privilege like this still exists.
- A privilege on the basis of which a view has been defined cannot be revoked if the view still exists.
- A privilege on the basis of which a routine has been defined cannot be revoked if the routine still exists.

REVOKE format for EXECUTE privileges (UDF)

```
REVOKE EXECUTE ON SPECIFIC FUNCTION udf
       FROM { PUBLIC | authorization_identifier } ,...
       { RESTRICT | CASCADE }

udf ::= routine
```

EXECUTE ON SPECIFIC FUNCTION *udf*

Name of the UDF for which the privilege is to be revoked. You can qualify the unqualified UDF name with a database and schema name.

FROM

See "REVOKE - Revoke privileges".

CASCADE

An authorization identifier can revoke any privileges it has granted:

- All the specified privileges are revoked.
- If a specified privilege has been forwarded to other authorization identifiers, all forwarded privileges and all routines and views created on the basis of these privileges are deleted in a cascade.
- Views defined on the basis of the specified privilege are deleted in a cascade.
- Routines defined on the basis of this privilege are deleted in a cascade.

RESTRICT

The following restrictions apply to the revoking of privileges:

- A privilege forwarded to other authorization identifiers cannot be revoked for as long as a forwarded privilege like this still exists.
- A privilege on the basis of which a view has been defined cannot be revoked if the view still exists.
- A privilege on the basis of which a routine has been defined cannot be revoked if the routine still exists.

8.2.3.57 ROLLBACK WORK - Roll back transaction

You use ROLLBACK WORK to terminate an SQL transaction and undo all the updates performed since the end of the last SQL transaction. Some statements, such as SET SCHEMA for example, are also rolled back if they were executed before the start of the current transaction but after the end of the last transaction.

ROLLBACK WORK undoes the following updates:

- updated data in SQL schemas
- cursor positions saved with STORE
- database and schema names set with SET CATALOG and SET SCHEMA
- authorization identifiers defined with SET SESSION AUTHORIZATION
- creation (ALLOCATE) and release (DEALLOCATE) of SQL descriptor areas
- values set in SQL descriptor areas

All the cursors opened in the transaction or positioned with FETCH are closed. Dynamic statements and cursor descriptions prepared with PREPARE are lost.

The SET TRANSACTION statement and the utility statements cannot be rolled back.

The first error-free SQL statement that initiates a transaction executed after ROLLBACK WORK starts a new SQL transaction (see section “COMMIT WORK - Terminate transaction”).

```
ROLLBACK [WORK]
```

Implicit execution of ROLLBACK WORK

SESAM/SQL rolls back an SQL transaction by implicitly executing a ROLLBACK WORK statement if one of the following situations occur:

- An unrecoverable error occurs in the current transaction.
- The specified isolation level cannot otherwise be ensured for two or more transactions that access certain SQL data concurrently (see also the “ Core manual”).
- A transaction is interrupted for a long time and is using resources required by other transactions (see also the “ Core manual”).

The effect is the same as if ROLLBACK were called explicitly.

Transactions under openUTM

You cannot use the ROLLBACK WORK statement if you are working with openUTM. In this case, transaction management is performed using only UTM language resources. If a UTM transaction is rolled back, the SQL transaction is also rolled back.

CALL DML transactions

Within a CALL DML transaction, the SQL statement ROLLBACK WORK is not permitted (see section “SQL statements in CALL DML transactions”).

See also

COMMIT

8.2.3.58 SELECT - Read individual rows

You use SELECT to read precisely one row in a table. The column values read are stored in the output destination.

If a derived table contains more than one row, the SELECT statement does not read a row and an appropriate SQLSTATE is set. If you want to read derived tables with more than one row, you must use a cursor.

In order to execute a SELECT statement, you must own the table in which you are querying values, or you must have the SELECT privilege for the table involved.

```
SELECT [ALL | DISTINCT] select_list
    [INTO parameter-declaration [ , parameter-declaration ]...
    FROM table_specification ,...
    [WHERE search_condition ]
    [GROUP BY column ,...]
    [HAVING search_condition ]

parameter-declaration ::=
{
    : host_variable [[INDICATOR] :indicator_variable ] |
    routine_parameter |
    local_variable
}
```

With the exception of the INTO clause, the clauses of the SELECT statement are defined exactly as they are for the SELECT expression and are described in the section “SELECT expression”ff.

INTO

Only for static SELECT statements.

In the case of a static SELECT statement or a SELECT statement in a procedure, you must specify the output destination that is to be assigned the column values of the derived row in the INTO clause.
Indicates where the values read are to be stored.

:host_variable, *routine_parameter*, *local_variable*
Name of a host variable (if the statement is **not** part of a routine) or name of a procedure parameter of the type INOUT or OUT or of a local variable (if the statement is part of a routine). The column value of the derived row is assigned to the specified output destination.

The data type must be compatible with the data type of the corresponding derived column (see section “Reading values into host variables or a descriptor area”). If a derived column is an aggregate with several elements, the corresponding host variable must be a vector with the same number of elements.

The number of specified output destinations must be the same as the number of columns in the SELECT list of the cursor description. The value of the nth column in the SELECT list is assigned to the nth output destination in the INTO clause. If the value to be assigned is the NULL value, the output destination is not set.

If there is no derived row or more than one derived row, no output destination is set.

If there is no derived row, an SQLSTATE is set that can be handled with WHENEVER NOT FOUND. If there is more than one derived row, an SQLSTATE is set that can be handled with WHENEVER SQLERROR.

indicator_variable
Name of the indicator variable for the preceding host variable. If the host variable is a vector, the indicator variable must also be a vector with the same number of elements.

The indicator value indicates whether the NULL value was transferred or whether data was lost:

0	The host variable contains the value read. The assignment was error free.
-1	The value to be assigned is the NULL value.
> 0	For alphanumeric or national values: The host variable was assigned a truncated string. The value of the indicator variable indicates the original length in code units.

indicator_variable omitted:

If the value to be assigned is the NULL value, an appropriate SQLSTATE is set.

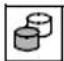
Dynamic SELECT statement

You must not specify an INTO clause in a dynamic SELECT statement. Instead, you specify the INTO clause with the host variables or SQL descriptor area for receiving the derived values in the EXECUTE statement with which you execute the dynamic SELECT statement.

Example

Read the name and VAT rate of the service with the specified service number and store this information in the host variables SERVICE_TEXT and VAT.

The service number is defined by the host variable SERVICE_NUM. Because the service number is unique within the SERVICE table, you can be sure that the query will return only one row.



```
SELECT service_text, vat

INTO :SERVICE_TEXT INDICATOR :IND_SERVICE_TEXT :VAT INDICATOR :IND_VAT

FROM service

WHERE service_num = :SERVICE_NUM
```

8.2.3.59 SET - Assign value

The SET statement assigns a value to a parameter or a local variable of a routine. It may may only be specified in a routine, i.e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. Routines and their use in SESAM/SQL are described in detail in chapter “Routines”.

```
SET { routine_parameter | local_variable } = { expression | NULL }
```

routine_parameter

Procedure parameter of the type INOUT or OUT of the current procedure, see "CREATE PROCEDURE - Create procedure".

local_variable

Local variable of the current COMPOUND statement, see "COMPOUND - Execute SQL statements in a common context".

expression

Expression whose value is assigned to the procedure parameter or local variable. The expression may contain routine parameters and (if the statement is part of a COMPOUND statement) local variables, but no host variables.
A column may be specified only if it is part of a subquery.
The data type of the expression must be compatible with the data type of the procedure parameter or of the local variable. The rules in section “Entering values in a procedure parameter (output) or local variable” apply.

NULL

The procedure parameter or the local variable is assigned the NULL value.

Example

```
SET number_of_reads = (SELECT COUNT (*) FROM mytable)
```

See also

COMPOUND, CREATE FUNCTION, CREATE PROCEDURE

8.2.3.60 SET CATALOG - Set default database name

You use SET CATALOG to define the default database name for unqualified schema names that occur in statements subsequently prepared with PREPARE or EXECUTE IMMEDIATE. The default database name set with the precomiler option continues to be used to qualify unqualified schema names for all other statements. Until the time that the first SET CATALOG (or SET SCHEMA) statement is executed, the database name specified with the precompiler option is used as the default database name for all statements.

The defined default determined by SET CATALOG is revoked when the immediately following transaction - the current UTM transaction in the case of openUTM - is rolled back. This is also true if the transaction immediately following SET CATALOG only contains CALL DML statements. Otherwise the default database name you set with SET CATALOG is valid until a new database name is set with SET CATALOG or SET SCHEMA or until the end of the SQL session. You will find information on the general rules for implicit database and schema names in section "Qualified names".

The SET CATALOG statement does not initiate a transaction.

`SET CATALOG` *default_catalog*

default_catalog `::=` { *alphanumeric_literal* | `:` *host_variable* }

---

default_catalog

Name of the database to act as the default for the current SQL session.

alphanumeric_literal

The database name is specified as an alphanumeric literal (not in the hexadecimal format).

host_variable

The database name is specified as an alphanumeric host variable of the type CHAR or VARCHAR. The host variable cannot be a vector and cannot have an associated indicator variable.

Example

```
SET CATALOG 'ordercust'
```

See also

SET SCHEMA

8.2.3.61 SET DESCRIPTOR - Update SQL descriptor area

You use SET DESCRIPTOR to update an SQL descriptor area. You can update the values for the descriptor area field COUNT or the contents of an item descriptor.

See section "Descriptor area" for information on the structure and use of the descriptor area.

The SQL descriptor area must be created beforehand.

```
SET DESCRIPTOR GLOBAL descriptor

{ COUNT= number |

  VALUE item_number field_id = field_contents [ , field_id = field_contents ] ...  }


number ::= { integer | host_variable }


item_number ::= { integer | host_variable }


field_id ::=

{
   REPETITIONS |
   TYPE |
   DATETIME_INTERVAL_CODE |
   PRECISION |
   SCALE |
   LENGTH |
   INDICATOR |
   DATA
}


field_contents ::= { host_variable | { number | host_variable }
```

---

descriptor

Name of the SQL descriptor area containing the items to be updated.

You cannot update the items in this descriptor area if there is an open cursor with block mode activated (see section "PREFETCH pragma") and a FETCH NEXT... statement whose INTO clause contains the name of the same SQL descriptor area has been executed for this cursor.

COUNT=*number*

The COUNT field is set to the value of *number*.

number

For *number*, specify an integer or a host variable of the SQL data type SMALLINT, where

0 <= *number* <= defined maximum number of item descriptors

The contents of item descriptors with an item number greater than *number* are undefined.

VALUE clause

The specified field of the item descriptor with the item number *item_number* are set to the specified field contents.

If you specify several fields, they are set in the following order regardless of the order in which you specify them in the SET DESCRIPTOR statement:

REPETITIONS

TYPE

DATETIME_INTERVAL_CODE

PRECISION

SCALE

LENGTH

INDICATOR

DATA

item_number

Number of the item descriptor to be updated.

The items in the descriptor area are numbered sequentially starting with 1.

For *item_number* you can specify an integer or a host variable of the type SMALLINT, where

1 <= *item_number*<= COUNT and <= defined maximum number of items

field_id

Field of item descriptor *item_number* to be updated. You can only specify the same field identifier once.

field_contents

New value for the field *field_id*

If *field_id* is DATA, you must specify a host variable for *field_contents.* Otherwise you can specify an integer or a host variable of the SQL data type SMALLINT for *field_contents*. You cannot specify an aggregate or vector for any field except DATA and INDICATOR.

REPETITIONS

The value specified for *field_contents* must be >= 1 and <= 255.

The fields TYPE, DATETIME_INTERVAL_CODE PRECISION, SCALE, and LENGTH are set to the same value for the item descriptors with the item numbers *item_number*, *item_number*+1, ..., *item_number* +REPETITIONS-1,
provided that the item numbers are <= COUNT and <= defined maximum number of item descriptors.

The REPETITIONS field is set to the value of *field_contents* for *item_number*. REPETITIONS is set to 1 for the item descriptors with the item numbers *item_number*+1, ..., *item_number*+REPETITIONS-1.

The other fields for the items involved are set to the value specified or are undefined.

REPETITIONS omitted:
REPETITIONS is set to 1 for *item_number.*

TYPE

Sets the TYPE field. The contents of the DATETIME_INTERVAL_CODE field of the same item descriptor are undefined. The fields PRECISION, SCALE and LENGTH of the same item descriptor are set to default values, depending on the value of the TYPE field:

SQL data type	TYPE	PRECISION	SCALE	LENGTH
NVARCHAR	-42			1
NCHAR	-31			1
CHAR	1			1
NUMERIC	2	1	0	
DECIMAL	3	1	0	
INTEGER	4			
SMALLINT	5			
FLOAT	6	1		
REAL	7			
DOUBLE PRECISION	8			
DATE, TIME, TIMESTAMP	9	0		
VARCHAR	12			1

Table 51: Setting the TYPE field of an item descriptor

Values not specified are undefined.

Except for REPETITIONS, the values of all the other fields for this item descriptor are undefined.

DATETIME_INTERVAL_CODE

Depending on the value of DATETIME_INTERVAL_CODE, the value of the RECISION field is set as follows:

DATETIME_INTERVAL_CODE	PRECISION
1	0
2	0
3	6

Table 52: Setting the DATETIME_INTERVAL_CODE field of an item descriptor

Except for REPETITIONS and TYPE, the values of all the other fields for this item descriptor are undefined.

PRECISION, SCALE, LENGTH

The fields are set in this order. If the TYPE field is already set and PRECISION, SCALE or LENGTH contain default values, these are overwritten.

The value of the DATA field for this item descriptor is undefined.

INDICATOR

If you specify a vector with several elements, a corresponding number of INDICATOR fields for the subsequent item descriptors are set, provided that the item numbers of these items are <= COUNT and <= defined maximum number of item descriptors.

DATA

The data type of the host variable must match the data type indicated by the TYPE, LENGTH, PRECISION, SCALE and DATETIME_INTERVAL_CODE fields of the same item descriptor (see section “Transferring values between host variables and a descriptor area”). If the specified host variable is a vector with several elements, the TYPE, LENGTH, PRECISION, SCALE and DATETIME_INTERVAL_CODE fields of exactly the same number of subsequent item descriptors must specify the same data type, and the item number of these item descriptors must be <= COUNT and <= defined maximum number of item descriptors.

If DATA and INDICATOR are specified, both must be atomic values or vectors with the same number of elements.

The DATA field is set if the associated INDICATOR field is >= 0. Otherwise the contents of the DATA field are undefined.

Examples

The type and number of decimal digits and number of digits after the decimal point in the second item descriptor in the SQL descriptor area :DEMO_DESC are changed:

```
SET DESCRIPTOR GLOBAL :demo_desc

VALUE 2 TYPE = 2, PRECISIONS = 7, SCALE = 2
```

Set the number of item descriptors in the SQL descriptor area DEMO_DESC to zero:

```
SET DESCRIPTOR GLOBAL :demo_desc COUNT = 0
```

See also

ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DESCRIBE, GET DESCRIPTOR

8.2.3.62 SET SCHEMA - Specify default schema name

You use SET SCHEMA to define the default schema name for the unqualified name of integrity constraints, indexes and tables that occur in statements subsequently prepared with PREPARE or EXECUTE IMMEDIATE. The default schema name set with the precompiler option continues to be used to qualify the names of integrity constraints, indexes and tables for all other statements. Until the time that the first SET SCHEMA statement is executed, the schema name set with the precompiler option is used as the default schema name for all statements.

The defined default determined by SET SCHEMA is revoked when the immediately following transaction - the current UTM transaction in the case of openUTM - is rolled back. This is also true if the transaction immediately following SET SCHEMA only contains CALL DML statements.
Otherwise the default schema name you set with SET SCHEMA is valid until a new schema name is set with SET SCHEMA or until the end of the SQL session.

You will find information on the general rules for implicit database and schema names in section “Qualified names”.

The SET SCHEMA statement does not initiate a transaction.

SET SCHEMA *default_schema*

default_schema ::= { *alphanumeric_literal* | :*host_variable* }

---

default_schema

Name of the default schema for the current SQL session. You can qualify the unqualified schema name with a database name.
If you qualify the schema name with a database name, this database name is used as the default database name as if it has been set with SET CATALOG.

alphanumeric_literal

The schema name is specified as an alphanumeric literal (not in the hexadecimal format).

host_variable

The schema name is specified as an alphanumeric host variable of the type CHAR or VARCHAR. The host variable cannot be a vector and cannot have an associated indicator variable.

Examples

Example from the sample database:

SET SCHEMA 'ordercust.orderproc'

Example from the dynamic SQL:

The host variable SOURCESTMT contains the following statement:

```
CREATE TABLE ordstat (order_stat_num INTEGER, order_stat_text CHAR(15))
```

The following statements execute a CREATE TABLE statement for the table ORDSTAT in the schema ORDERPROC of the database ORDERCUST:

```
SET SCHEMA 'ordercust.orderproc'

EXECUTE IMMEDIATE :SOURCESTMT
```

See also

SET CATALOG

8.2.3.63 SET SESSION AUTHORIZATION - Set authorization identifier

You use SET SESSION AUTHORIZATION to define the current authorization identifier for the SQL session.

The current authorization code of an SQL session is defined either with an ESQL precompiler option or with the SET SESSION AUTHORIZATION statement. If no authorization code is defined at either place, the default value `D0USER` is used as the current authorization code for the SQL session.

The setting determined by SET SESSION AUTHORIZATION is revoked when the immediately following transaction - the current UTM transaction in the case of openUTM - is rolled back. This is also true if the immediately following transaction only contains CALL DML statements.
Otherwise the authorization key you set with SET SESSION AUTHORIZATION is valid until a new authorization key is set with SET SESSION AUTHORIZATION or until the end of the SQL session.

The SET SESSION AUTHORIZATION does not initiate a transaction and can therefore only be used outside of an SQL transaction.

`SET SESSION AUTHORIZATION` *new_authorization_identifier*

new_authorization_identifier `::=` { *alphanumeric_literal* | :*host_variable* }

---

new_authorization_identifier

Name of the new authorization identifier that is to be valid for the SQL session. The new authorization identifier is valid until the next SET SESSION AUTHORIZATION statement.

alphanumeric_literal

The new authorization identifier is specified as an alphanumeric literal (not in the hexadecimal format) of the data type CHAR.

host_variable

The new authorization identifier is specified as an alphanumeric host variable of the type CHAR or VARCHAR. The host variable cannot be a vector and cannot have an associated indicator variable.

Examples

Define a new authorization identifier UTIADM for the current SQL session. The current UTM or BS2000 user must have a system entry with this authorization identifier.

SET SESSION AUTHORIZATION 'utiadm'

Specify the authorization identifier for the current SQL session as a host variable.

SET SESSION AUTHORIZATION :USER-NAME

8.2.3.64 SET TRANSACTION - Define transaction attributes

You can use SET TRANSACTION to set the isolation or consistency level and transaction mode for the subsequent SQL transaction.

The isolation or consistency level of a transaction specifies to what degree read operations on rows in the transaction are affected by simultaneous write accesses in a concurrent transaction.

The transaction mode allows you to specify whether table rows can only be read or can also be updated in the subsequent transaction.

> **CAUTION!** If you define an isolation or consistency level, you also influence the degree of concurrency and thus performance: the fewer phenomena you permit, the lesser the degree of concurrency.

The settings made by SET TRANSACTION are only valid for the SQL statements of the immediately following transaction. After the transaction has ended or has been rolled back, the default values remain valid (see “Default values”). The default settings also continue to apply after the end of the transaction when the transaction which follows SET TRANSACTION only contains CALL DML statements, i.e. no SQL statements.

The SET TRANSACTION statement does not initiate a transaction and can only be used outside an SQL transaction.

```
SET TRANSACTION { level [[,] transaction_mode ] | transaction_mode [[,] level ] }


transaction_mode ::= { READ ONLY | READ WRITE }

level ::= { ISOLATION LEVEL isolation-level | CONSISTENCY LEVEL consistency_level }

isolation-level ::=

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

You can omit the comma between the two specifications. If, however, you want your application to be portable, you must include the comma.

ISOLATION LEVEL

Sets the isolation level.

If several transactions work with the same tables simultaneously, the following phenomena can occur in which the read accesses in one transaction are affected by the simultaneous write access of another transaction. By specifying an isolation level, you determine which of these phenomena you want to permit in the subsequent SQL transaction.

The following phenomena are of importance:

- dirty read:
  A transaction updates a row or inserts a new row. A second transaction reads this row before the first transaction has committed the update. If the first transaction is rolled back, the second transaction has read a row that was never committed.
- non-repeatable read:
  A transaction reads a row. Before this transaction is terminated, a second transaction updates or deletes this row and commits the update. If the first transaction then tries to read this row again, either different values will be returned, or an error occurs because the row has been deleted in the meantime. In other words, the result of the second read operation is different to the result of the first.
- phantom:
  A transaction reads rows that satisfy a certain search condition. A second transaction subsequently inserts rows that also satisfy this search condition. If the first transaction repeats the query, the derived table includes the new rows.

READ UNCOMMITTED

Isolation level that offers the least protection against concurrent transactions. All the above-mentioned phenomena are possible. In the subsequent SQL transaction, rows can be read that have not yet been committed and these rows can be updated after they have been read.

You cannot specify READ UNCOMMITTED if, at the same time, you specify the transaction mode READ WRITE.

READ COMMITTED

The phenomena “non-repeatable read” and “phantom” can occur. In the subsequent SQL transaction, rows that have been read can be updated by other transaction after they have been read. No rows are read that have not yet been committed.

REPEATABLE READ

The phenomenon “phantom” can occur. The phenomena “non-repeatable read” and “dirty read” are not possible.

SERIALIZABLE

Complete protection against concurrent transactions is ensured. The phenomena dirty read, non-repeatable read,and phantom cannot occur. The subsequent transaction is unaware of the existence of concurrent transactions.

CONSISTENCY LEVEL

For reasons of upward compatibility with earlier versions, SESAM/SQL provides the clause CONSISTENCY LEVEL as an alternative to isolation level. This means that you define a consistency level which, like the isolation level, determines whether the phenomena “dirty read”, “non-repeatable read” and “phantom” can occur.

consistency_level

Unsigned integer, where 0 <= *consistency_level* <= 4.

Level	Locks set	Rows read
0	Rows read are not locked against updating by other transactions	All rows including those locked against updating by other transactions
1	Rows read are locked against updating by other transactions (until the end of the transaction) unless they are already locked	like 0
2	like 0	Only the rows that other transaction have not locked against updating
3	Rows read are locked against updating by other transactions (until the end of the transaction)	like 2
4	Rows read are locked just as for level 3. The lock against updating by other transactions for nonexistent rows ensures that rows cannot be inserted by other transactions.	like 2

Table 53: Consistency levels

The following table indicates the correlation between isolation and consistency level and which phenomena can occur at the different consistency and isolation levels.

Isolation level	Consistency level	dirty read	nonrepeatable read	phantom
READ UNCOMMITTED	0	x	x	x
-	1	x	x [1]	x
READ COMMITTED	2	-	x	x

REPEATABLE READ	3	-	-	x
SERIALIZABLE	4	-	-	-

Table 54: Correlation between isolation level, consistency level and phenomena

[1]The phenomenon non-repeatable read can only occur for rows which were previously read with dirty read.

READ ONLY

Sets the transaction mode READ ONLY.

Only read database accesses are permitted within the transaction. READ ONLY is the default value for the isolation level READ UNCOMMITTED and the consistency levels 0 and 1.

READ WRITE

Sets the transaction mode READ WRITE.

Only read and write database accesses are possible in the transaction. READ WRITE is the default value for the isolation levels READ COMMITTED, REPEATABLE READ and SERIALIZABLE and for the consistency levels 2, 3 and 4.

You cannot specify READ WRITE if you specify the isolation level READ UNCOMMITED.

Default values

If a connection module entry exists for the isolation or consistency level in the user-specific configuration file (see the " Core manual"), this value is used as the default. If this is not the case, the isolation level SERIALIZABLE, the consistency level 4 and the transaction mode READ WRITE are the default values.

You can use the MAX-ISOLATION-LEVEL operand of the DBH option TRANSACTION-SECURITY to set the isolation level REPEATABLE READ for a DBH. If your SQL statement works with a DBH set in this way, one of the following constraints must be fulfilled:

- the configuration file must contain the connection module parameter ISOL-LEVEL=REPEATABLE-READ (or a lower isolation level)
  or
- you must limit the isolation level to REPEATABLE READ using the SQL statement SET TRANSACTION prior to each transaction.

Scope of validity under openUTM

In a UTM application, the statement SET TRANSACTION is no longer valid once the current UTM transaction terminates. Since only one database transaction can run in a UTM transaction, SET TRANSACTION and the associated SQL transaction must be performed in the same UTM transaction.

8.2.3.65 SIGNAL - Report exception in routine

SIGNAL explicitly reports, in a routine, anexception or a self-defined SQLSTATE.

SIGNAL deletes the current diagnostics area and enters corresponding diagnostic information into the current diagnostics area:

SIGNAL is one of the diagnostic statements. Detailed information on the use and effect of SIGNAL can be found in section "Diagnostic information in routines".

SIGNAL { *error_name* | *sqlstate* } [SET *diagnostic_info*]

sqlstate ::= SQLSTATE [VALUE] *alphanumeric_literal*

diagnostic_info ::= MESSAGE_TEXT= *message*

message ::= { *alphanumeric_literal* | *local_variable* | *routine_parameter* }

---

exception_name

Name of an exception or SQLSTATE. *exception_name* is defined in the local data of a routine, see "Local data"
.

sqlstate

Explicit specification of a self-defined SQLSTATE (alphanumeric literal with the length 5), see section "Self-defined SQLSTATEs".

MESSAGE_TEXT=*alphanumeric_literal*

Any information (maximum length: 120 characters). The text length is entered in MESSAGE_LENGTH and MESSAGE_OCTET_LENGTH.

MESSAGE_TEXT=*local_variable / routine_parameter*

The value of the local variable or of the specified routine parameter is entered as information text.
The data type of *local_variable / routine_parameter* must be compatible with the data type VARCHAR(120).
The rules in section "Entering values in a procedure parameter (output) or local variable" apply. The text length is entered in MESSAGE_LENGTH and MESSAGE_OCTET_LENGTH.

SET MESSAGE TEXT omitted:

The diagnostic information MESSAGE_TEXT, MESSAGE_LENGTH, and MESSAGE_OCTET_LENGTH is supplied with the corresponding NULL values.

Examples (see also "Diagnostic information in routines")

Reporting a self-defined SQLSTATE:

```
SIGNAL SQLSTATE VALUE '46SA5';
```

Reporting a condition with information text:

```
SIGNAL end_job SET MESSAGE_TEXT='The end is near!';
```

See also

COMPOUND, CREATE FUNCTION, CREATE PROCEDURE, GET DIAGNOSTICS, RESIGNAL

8.2.3.66 STORE - Save cursor position

You use STORE to save the current cursor position.

At the end of a transaction, all open cursors are closed. If you want to be able to access the contents of the derived table in the subsequent transaction, you must save the current cursor position with STORE before the end of the transaction. A cursor saved with STORE can be restored with the RESTORE statement.

FETCH cannot be used after STORE.

The cursor must be open.

STORE is not permitted if block mode is activated for the open cursor (see section “PREFETCH pragma”).

STORE must not be used with cursors defined with WITH HOLD.

```
STORE cursor
```

cursor

Name of the cursor whose position is to be stored.

The call overwrites any cursor position for the same cursor previously saved with STORE.

See also

DECLARE CURSOR, OPEN, RESTORE

8.2.3.67 UPDATE - Update column values

You use UPDATE to update the column values of rows in a table.
The primary key value in a partitioned table may not be modified.

The special literals (see "Special literals"), as well as the time functions CURRENT_DATE, CURRENT_TIME and CURRENT_TIMESTAMP in the UPDATE statement (and in default values) are evaluated once, and the values calculated are valid for all updates.

If you want to update a row in the specified table, you must own the table or have the UPDATE privilege for each of the columns to be updated. Furthermore, the transaction mode of the current transaction must be READ WRITE.

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

```
UPDATE table [[AS] correlation_name ]
    SET colspec = column_value [, colspec = column_value ]...
   [WHERE { search_condition | CURRENT OF cursor }]


colspec ::= { column | column ( posno ) | column ( min..max ) }
column_value ::= { expression | <{ value | NULL},...> | DEFAULT | NULL }
```

table

Name of the table containing the rows you want to update. The table can be a base table or an updatable view.

correlation_name

Table name used in the statement as a new name for the table *table*.

The *correlation_name* must be used to qualify the column name in every column specification that references the table *table* if the column name is not unambiguous.

The new name must be unique, i.e. *correlation_name* can only occur once in a table specification of this statement.

You must give a table a new name if the columns in the table cannot be identified otherwise uniquely.

In addition, you may give a table a new name in order to formulate an expression so that it is more easily understood or to abbreviate long names.

column

Name of an atomic column whose contents you want to update. The column must be part of the table. You can only specify a column once in an UPDATE statement.

column(*pos_no*)

Element of a column containing the value you want to update.

The multiple column must be part of the table. If you specify several elements in a multiple column, the range of elements specified must be contiguous. Each element can only be specified once.

pos_no is an unsigned integer >= 1.

column(*min*..*max*)

Range of column elements in a multiple column that are to be assigned values. The multiple column must be part of the table. If you specify several elements in a multiple column, the range of elements specified must be contiguous.
Each element can only be specified once.

min and *max* are unsigned integers >= 1; *max* must be >= *min*.

expression

Expression whose value is to be assigned to the preceding atomic column. The value of the expression must be compatible with the data type of the column (see section "Entering values in table columns").

If *expression* is a host variable, you can also specify a vector. If you do so, the column must be a multiple column and the number of elements in the vector must be the same as the number of column elements.

The following restrictions apply to *expression*:

- Neither the underlying base table for *table* nor a view of this base table can be included in the FROM clause of a subquery in *expression*.
- Aggregate functions (AVG, MAX, MIN, SUM, COUNT) are not permitted.

`<{` *value* `, NULL},...>`

Aggregate to be assigned to a multiple column.

The number of occurrences must be the same as the number of column elements. The data type of *value* must be compatible with the data type of the target column (see section "Entering values in table columns").

DEFAULT

Only for atomic columns.

If a default value is defined for a particular column, this value is entered in that column. Otherwise, the column is assigned the NULL value.

NULL

The preceding column is assigned the NULL value.

WHERE clause

The WHERE clause indicates the rows to be updated.

WHERE clause omitted:
All the rows in the table are updated.

search_condition

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

The following restrictions apply to *search_condition:*

- Column specifications in *search_condition* outside of subqueries can only reference the specified table.
- Neither the underlying base table for *table* nor a view of this base table can be included in the FROM clause of a subquery included in *search_condition*.

If no row satisfies the search condition, no row is updated and an SQLSTATE is set that can be handled with WHENEVER NOT FOUND.

CURRENT OF *cursor*

Name of the cursor used to determine the row to be updated. *table* must be the table specified in the first FROM clause of the cursor description.

The cursor must satisfy the following conditions:

- The cursor must reference *table*.
- The cursor must be updatable.
- When the UPDATE statement is executed, the cursor must be open and positioned on a row in the table with FETCH. In addition, the FETCH statement must be executed in the same transaction as the UPDATE statement.

UPDATE updates the row indicated by *cursor*.

UPDATE is not permitted if block mode is activated for *cursor* (see section “PREFETCH pragma”).

If *cursor* was declared with the FOR UPDATE clause and column specifications, only the columns specified in that clause can be updated.

The UPDATE statement does not influence the position of the cursor. If you want to update the next row in the derived table, you must position the cursor on this row with FETCH.

Updating the values in a multiple column

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

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

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

! **CAUTION!**The position of an element in a multiple column can change. If an element with a low position number is set to the NULL value, all subsequent elements are shifted to the left and the NULL value added to the end.

UPDATE and integrity constraints

By specifying integrity constraints when you define the base table, you can restrict the possible contents of *table*. After UPDATE statement has been executed, the contents of *table* must satisfy the defined integrity constraints.

UPDATE and updatable view

If CHECK OPTION is specified in the definition of an updatable view, only rows that satisfy the query expression in the view definition can be inserted in the view.

UPDATE and transaction management

UPDATE initiates a transaction outside routines if no transaction is open. If you define an isolation level for concurrent transactions, you can control how the UPDATE statement affects these transactions (see section “SET TRANSACTION - Define transaction attributes”).
If an error occurs during an UPDATE statement, any updates that have already been performed are canceled.

Examples

Increase the minimum stock level of all items to 20.

```
UPDATE items
SET    min_stock = 20
WHERE  min_stock < 20
```

Update the minimum stock level using a cursor:

```
DECLARE cur_items CURSOR FOR
   SELECT min_stock FROM items WHERE min_stock < 20
   FOR UPDATE

OPEN cur_items
```

Update the rows involved with a series of FETCH and UPDATE statements.

```
FETCH cur_items INTO :MIN_STOCK

UPDATE items SET min_stock = 20 WHERE CURRENT OF cur_items
```

Use the cursor CUR_VAT to select all services for which no VAT is calculated. Update the rows involved with a series of FETCH and UPDATE statements.

```
DECLARE cur_vat CURSOR WITH HOLD FOR
   SELECT service_num, service_text, vat
   FROM service WHERE vat=0.00
   FOR UPDATE

FETCH NEXT cur_vat
      INTO :SERVICE_NUM, :SERVICE_TEXT INDICATOR :IND_SERVICE_TEXT :VAT INDICATOR :IND_VAT

UPDATE service SET vat=0.15 WHERE CURRENT OF cur_vat
...
```

Update the intensity of the individual color components for the color orange in the COLOR_TAB table. The column RGB for the color intensity is a multiple column:

```
UPDATE color_tab SET rgb(1..3) = <0.8, 0.4, 0> WHERE color_name = 'orange'
```

See also

DELETE, INSERT, MERGE

8.2.3.68 WHENEVER - Define error handling

You use WHENEVER to define the reaction to statements terminated with an SQLSTATE '00000' and '01 *xxx*'.

WHENEVER is not an executable statement.

You can specify the WHENEVER statement more than once in a program. The specifications made in a WHENEVER statement are valid for all subsequent SQL and utility statements in the program text (after all includes have been inserted) until the next WHENEVER statement for the same error class.

WHENEVER...CONTINUE is valid before the first WHENEVER statement.

```
WHENEVER

{ SQLERROR | NOT FOUND }

{ CONTINUE | { GOTO | GO TO }[:] label }
```

SQLERROR

Define handling of:

SQLSTATE "00000", "01*xxx*" and "02*xxx*".

NOT FOUND

Define handling of:

SQLSTATE = "02*xxx*".

CONTINUE

After SQLERROR or NOT FOUND, the program is continued with the next statement. You can use CONTINUE to cancel a previously defined action for the same error class.

If the program section for error handling includes SQL statements, this section should be introduced by a WHENEVER statement with a CONTINUE clause. This avoids endless loops if the error occurs again.

label

Label in an ESQL program.

This clause corresponds to a branch statement in the host language (e.g. GO TO in COBOL).

label must conform to the naming conventions for labels of the host language involved (see the " ESQL-COBOL for SESAM/SQL-Server" manual).

After SQLERROR or NOT FOUND, the program is continued at the location indicated by *label*.

The colon : is only supported for reasons of upward compatibility.

Examples

Continue the program with the paragraph SQLERR following a statement that ends with an SQLSTATE '00000', '01 *xxx*' or '02*xxx*'.

WHENEVER SQLERROR GOTO sqlerr

This example demonstrates how to use the WHENEVER statement when reading a cursor table with FETCH.
Before positioning the cursor, you define a label to cater for situations where the specified cursor position does not exist.
The cursor is positioned on the next row within a loop.
The program works its way through the cursor table reading each row until it reaches the end of the table.
If the specified position does not exist, a corresponding SQLSTATE is set and the program is continued from the label defined in the WHENEVER statement.
The action defined for error handling is cancelled at the label.

```
      WHENEVER NOT FOUND GOTO F-4
F-2.
      FETCH cur_contacts
            INTO :LNAME,
                 :FIRSTNAME INDICATOR :IND_FIRSTNAME,
                 :DEPT      INDICATOR :IND_DEPT

F-3.
```

Output row, go to F-2

```
F-4.

      WHENEVER NOT FOUND CONTINUE
```

8.2.3.69 WHILE - Execute SQL statements in a loop

The WHILE statement executes SQL statements in a loop until the specified search condition is satisfied. The loop begins with a check, i.e. it can already be terminated before the first pass.

The ITERATE statement enables you to switch immediately to the next loop pass. The loop can be aborted by means of a LEAVE statement.

The WHILE statement may only be specified in a routine, i.e. in the context of a CREATE PROCEDURE or CREATE FUNCTION statement. Routines and their use in SESAM/SQL are described in detail in chapter "Routines"
.

The WHILE statement is a non-atomic SQL statement, i.e. further (atomic or non-atomic) SQL statements can occur in it.

If the WHILE statement is part of a COMPOUND statement, in the case of corresponding exception routines the loop can also be left when a particular SQLSTATE (e.g. no data, class 02xxx) occurs.

`[` *label* `:]`

`WHILE` *search_condition*

    `DO` *routine_sql_statement;* `[` *routine_sql_statement;* `]...`

`END WHILE [` *label* `]`

---

label

The label in front of the WHILE statement (start label) indicates the start of the loop. It may not be identical to another label in the loop.

The start label need only be specified when the next loop pass is to be switched to using ITERATE or when the loop is to be left using a LEAVE statement. However, it should always be used to permit SESAM/SQL to check that the routine has the correct structure (e.g. in the case of nested loops).

The label at the end of the WHILE statement (end label) indicates the end of the loop. If the end label is specified, the start label must also be specified. Both labels must be identical.

search_condition

Search condition that returns a truth value when evaluated.

routine_sql_statement

SQL statement which is to be executed in the WHILE statement.
An SQL statement is concluded with a ";" (semicolon).
Multiple SQL statements can be specified one after the other. They are executed in the order specified.
No privileges are checked before an SQL statement is executed.
An SQL statement in a routine may access the parameters of the routine and (if the statement is part of a COMPOUND statement) local variables, but not host variables.

The syntax and meaning of *routine_sql_statement* are described centrally in section “SQL statements in routines”. The SQL statements named there may not be used.

Execution information

The WHILE statement is a non-atomic statement:

- If the WHILE statement is part of a COMPOUND statement, the rules described there apply, in particular the exception routines defined there.
- If the WHILE statement is **not** part of a COMPOUND statement and one of the SQL statements reports an SQLSTATE, it is possible that only the updates of this SQL statement will be undone. The WHILE statement and the routine in which it is contained are aborted. The SQL statement in which the routine was used returns the SQLSTATE concerned.

Example

The loop is executed until the variable i has a value < 100.

```
DECLARE i INTEGER DEFAULT 0;
   label:
   WHILE i < 100
   DO
       SET i = i+1;
       ...
   END WHILE label;
```

See also

CREATE PROCEDURE, CREATE FUNCTION, ITERATE, LEAVE

9 SESAM-CLI

This chapter describes the SESAM CLI (**C**all **L**evel **I**nterface) and is divided into two parts:

- the section “Concept of the SESAM CLI”, which describes the structure of CLI calls, the data types used and the handling of transactions
- the section “SESAM CLI calls”, which describes the function and syntax of calls in alphabetical order

9.1 Concept of the SESAM CLI

The SESAM CLI (**C**all **L**evel **I**nterface) is a procedural interface which is primarily used to access BLOBs (**B**inary **L** arge **Ob**jects). There is also another CLI call at present which you can use to define attribute values for dynamic INSERT statements.

A BLOB is a sequence of bytes of variable length, up to a maximum of 2^{31}-1 bytes. SESAM/SQL allows you to store BLOBs in databases in the form of persistent objects. You can then display and manipulate them outside SESAM/SQL using special programs, such as Microsoft TM Word for example.

The value of a BLOB is referred to simply as a BLOB value. BLOBs can only be stored in special tables, known as BLOB tables. These are created using the SQL statement CREATE TABLE *table* OF BLOB (see "CREATE TABLE - Create base table"). The BLOB values of a BLOB table form the objects of a particular class.

When a BLOB is created, its value and assigned attributes are written to a BLOB table. In addition, a unique REF value is created for referencing the BLOB throughout its lifetime. This REF value can be stored in the REF column of any base table. The BLOB value itself is stored piecemeal in several rows of the BLOB table. This storage method allows for efficient sequential access to BLOB values.

The attributes of a particular BLOB consist of properties defined by the user and properties assigned to the object by SESAM/SQL (see section “CREATE TABLE - Create base table”).

The SESAM CLI allows you to address BLOBs, their classes and attributes, BLOB values and sequences of BLOB values. The individual CLI calls are described in detail in the section “SESAM CLI calls”.

The contents of BLOB values are processed not in SESAM/SQL, but in object-specific programs, such as MS Word in the case of Word documents. When transferring BLOB values from SESAM/SQL to a BS2000 file, for instance, you have two options:

- You can use SQL_BLOB_VAL_GET to read the entire BLOB value from a buffer.
- You can use the command sequence SQL_BLOB_VAL_OPEN, SQL_BLOB_VAL_FETCH and SQL_BLOB_VAL_CLOSE to read out the individual segments of the BLOB value one by one (see the section “Alphabetical reference section”).

> **i** There is a variant with the suffix _STATELESS for each of the functions SQL_BLOB_VAL_OPEN, SQL_BLOB_VAL_FETCH, SQL_BLOB_VAL_STOW and SQL_BLOB_VAL_CLOSE. This set of functions can be used to process BLOBs step-by-step even if UTM dialog step changes occur. The interfaces of these functions are described in the files *sqlblox.h* (for C) and *SQLBLOX* (for COBOL).

The demonstration database of SESAM/SQL (see the “ Core manual”) contains tables for managing BLOBs. There you will also find an ESQL program with C functions for editing these BLOB objects.

9.1.1 Structure of SESAM CLI calls

SESAM CLI calls can be issued from C or COBOL programs.

They all have the following basic structure:

---

cli_call ::= *cli_procedure_name* (*cli_parameters* [, *cli_parameters*]...)

cli_parameters ::= *cli_parameter_name* { IN | IN OUT | OUT } *cli_parameter_data_type*

cli_parameter_data_type ::= { BOOLEAN | INTEGER | CHAR(*max*) | POINTER | SQLda }

---

cli_procedure_name

Name of the calling CLI procedure. Each procedure has a long name and a short name. The long form should be used in C. The short form should be used in COBOL (see alphabetical reference section).

cli_parameter_name

Name of the parameter (see alphabetical reference section).

`IN`, `IN OUT` or `OUT`

Indicates whether the parameter is an input parameter or an output parameter. IN OUT signals an input and output parameter.

cli_parameter_data_type

Name of the parameter data type.

The language-specific C and COBOL syntax of the individual CLI calls can be found in the section "SESAM CLI calls".

Corresponding data types

The table below shows the C and COBOL data types that correspond to SQL data types of CLI routines. The "Type" column indicates whether the parameter is an input parameter or an output parameter. The "Length" column specifies the length of values in bytes.

SQL data type	Type	COBOL data type	C data type	Length
BOOLEAN	IN	PIC S9(p) with the USAGE clause COMP	long int const *	4
	OUT		long int *	
INTEGER	IN	PIC S9(p) with the USAGE clause COMP	long int const *	4
	OUT		long int *	
	IN OUT			

CHAR(n)	IN	PIC X(n)	char const *	n
	OUT		char *	
	IN OUT			
POINTER	IN	PIC X(n) or COBOL group item	char *	4
CHAR(ddd)	OUT	SQLda	SQLda_t *	ddd

Table 55: Corresponding data types

The value “p” in the data type PIC S9(p) must be between 5 and 9.

The SYNCHRONIZED clause need not be specified in COBOL for arguments of CLI calls. The data type POINTER is used to transfer the address of a buffer, the length of which is defined in another parameter.

The COBOL data type SQLda

SQLda is a diagnostics area in SESAM/SQL. It is structured as follows in COBOL:

01	SQLda .				
	05	SQLda01	PIC	S9(4)	BINARY.
		88 SQLda01val		value 910.	
	05	SQLda02	PIC	S9(4)	BINARY.
	05	SQLda03	PIC	S9(4)	BINARY.
	05	SQLda04	PIC	S9(4)	BINARY.
	05	SQLerrline	PIC	S9(4)	BINARY.
	05	SQLerrcol	PIC	S9(4)	BINARY.
	05	SQLda07	PIC	S9(4)	BINARY.
	05	SQLda08	PIC	X(5).	
	05	SQLCLI-SQLSTATE redefines SQLda08 PIC X(5).			
	05	SQLerrm	PIC	X(240).	
	05	SQLda10	PIC	X.	
	05	SQLda21	PIC	S9(9)	BINARY.
	05	SQLda22	PIC	X(4).	
	05	SQLda23	PIC	S9(4)	BINARY.
	05	SQLda24	PIC	X(2).	
	05	SQLrowcount	PIC	S9(9)	BINARY.
	05	SQLda99	PIC	X(634).	

`SQLda`

Diagnostics area for SESAM/SQL.

`SQLda01, SQLda02, ... SQLda99`

Reserved for internal purposes.

`SQLerrline`

In the event of an error, this variable contains the line number of the position in the text of a prepared statement at which the error occurred. If the source of the error cannot be determined or the specified position makes no sense, this is set to 0 (zero).

`SQLerrcol`

In the event of an error, this variable contains the column number of the position in the text of a prepared statement at which the error occurred. If the source of the error cannot be determined or the specified position makes no sense, this is set to 0 (zero).

`SQLerrm`

Following the execution of an SQL statement, this variable contains a message text if the SQL statement returned a value other than 00000 in SQLSTATE. This consists of the error class (W for WARNING or E for ERROR), the message number SQL *nnnn* and the message text itself.

`SQLrowcount`

Following the execution of the corresponding statements, this variable contains the following information:

- in the case of an INSERT statement, the number of rows inserted
- in the case of an UPDATE or DELETE statement with a search condition, the number of rows that satisfied the search condition
- in the case of an UPDATE or DELETE statement without a WHERE clause, the number of rows in the referenced table
- in the case of a MERGE statement, the sum of the number of updated and the number of inserted rows
- in the case of an UNLOAD statement, the number of rows output
- in the case of a LOAD statement, the number of rows newly loaded
- in the case of an EXPORT statement, the number of rows copied to the export file
- in the case of an IMPORT statement, the number of rows copied from the export file

In all other cases, the contents of this variable are not defined.

The C data type SQLda_t

The SQLda_t data type is the C equivalent of COBOL’s SQLda. The diagnostics area in C is structured as follows:

typedef struct {			
	short	SQLda01;	/*length*/
	short	SQLda02;	/*reaction_code*/
	short	SQLda03;	/*error_code*/
	short	SQLda04;	/*errm_significant*/
	short	SQLerrline;	/*sqlrow*/
	short	SQLerrcol;	/*sqlcolumn*/
	short	SQLda07;	/*sqlcode*/
	char	SQLda08[5];	/*sqlstate*/
	char	SQLerrm[240];	/*sqlerrm*/
	signed char	SQLda10;	/*SQLda10*/
	long	SQLda21;	/*check field*/
	char	SQLda22[4];	/*tag*/
	unsigned short	SQLda23;	/*internal*/
	char	SQLda24[2];	/*slack*/
	long	SQLrowcount;	/*row_count*/
	char	SQLda99[634];	/*internal area*/ }
SQLda_t;			

The individual parameters are the same as those in the COBOL data type SQLda.

9.1.2 Statements that initiate transactions in CLI calls

Most CLI calls contain SQL statements that initiate transactions. For instance, with the exception of SQL_BLOB_CLS_REF, all CLI calls contain SQL statements for manipulating data (query, update).

> An SQL transaction must consist either of SQL statements for manipulating data or SQL statements for defining or managing schemas (see “Statements within a transaction”). For this reason, it is not possible to successfully execute CLI calls in an SQL transaction, even if the transaction also consists of SQL statements for defining or managing schemas.

Isolation level

The isolation level can be used to influence the parallel processing of transactions. The individual levels and the phenomena that can occur with concurrent transactions are described in the “SET TRANSACTION - Define transaction attributes”. The following effects may be seen when using BLOBs in CLI functions:

- If a BLOB value is to be read in a transaction with the isolation level SERIALIZABLE or REPEATABLE READ, any attempts on the part of concurrent transactions to update this value will be delayed until the read process is complete. This means that any updates to be carried out by concurrent transactions will either be visible in their entirety or not at all. The “phantom” phenomenon cannot occur here, since REF values are not reused.
- If a BLOB value is to be read in a transaction with the isolation level READ COMMITTED or READ UNCOMMITTED, it may be updated within the transaction by concurrent transactions. As a result, some of values read may be old while others are new. In the course of reading two segments of a BLOB value, it is even possible for the BLOB to be deleted and replaced by another BLOB with the same object number. In such cases, however, you can use the UPDATED attribute to determine when the object was last updated, and thus ensure that you are actually dealing with one and the same object.

Consistency in updates

A BLOB value is stored in several rows of the BLOB table. When replacing a BLOB value, therefore, you generally need more than one DML statement. The updating of a BLOB is not an atomic operation.

For this reason, it may be possible for an update to be only partially successful. For instance, the first BLOB_VAL_STOW call (see "SQL_BLOB_VAL_STOW - SQLbvst") for updating a BLOB value may be successful while the second fails. If this occurs, it is recommended that you reverse all updates using ROLLBACK.

In contrast, the updating of BLOB attributes and the deletion of BLOBs are atomic operations. If they fail, all values will be restored to their original status.

9.2 SESAM CLI calls

- Overview
- Alphabetical reference section
- SQL_BLOB_CLS_ISBTAB - SQLbcis
- SQL_BLOB_CLS_REF - SQLbcre
- SQL_BLOB_OBJ_CLONE - SQLbocl
- SQL_BLOB_OBJ_CREATE - SQLbocr
- SQL_BLOB_OBJ_CREAT2 - SQLboc2
- SQL_BLOB_OBJ_DROP - SQLbodr
- SQL_BLOB_TAG_GET - SQLbtge
- SQL_BLOB_TAG_PUT - SQLbtpu
- SQL_BLOB_VAL_CLOSE - SQLbvcl
- SQL_BLOB_VAL_FETCH - SQLbvfe
- SQL_BLOB_VAL_GET - SQLbvge
- SQL_BLOB_VAL_LEN - SQLbvle
- SQL_BLOB_VAL_OPEN - SQLbvop
- SQL_BLOB_VAL_PUT - SQLbvpu
- SQL_BLOB_VAL_STOW - SQLbvst
- SQL_DIAG_SEQ_GET - SQLdsg

9.2.1 Overview

The following SESAM CLI calls are available to users:

Operations involving BLOB classes

CLI call	Short form	Function
SQL_BLOB_CLS_REF	SQLbcre	Create and output class REF value
SQL_BLOB_CLS_ISBTAB	SQLbcis	Check whether BLOB table exists

Table 56: CLI calls for operations involving BLOB classes

Creating and deleting BLOBs

CLI call	Short form	Function
SQL_BLOB_OBJ_CLONE	SQLbocl	Create a clone of a BLOB
SQL_BLOB_OBJ_CREATE	SQLbocr	Create a BLOB (object number sequential)
SQL_BLOB_OBJ_CREAT2	SQLboc2	Create a BLOB (object number area-specific)
SQL_BLOB_OBJ_DROP	SQLbodr	Delete a BLOB

Table 57: CLI calls for BLOB objects

Reading and setting BLOB attributes

CLI call	Short form	Function
SQL_BLOB_TAG_GET	SQLbtge	Read an attribute value
SQL_BLOB_TAG_PUT	SQLbtpu	Set an attribute value

Table 58: CLI calls for BLOB attributes

Reading and setting BLOB values

CLI call	Short form	Function
SQL_BLOB_VAL_GET	SQLbvge	Output BLOB value
SQL_BLOB_VAL_PUT	SQLbvpu	Set BLOB value
SQL_BLOB_VAL_LEN	SQLbvle	Output the length of a BLOB value

Table 59: CLI calls for BLOB values

Sequential processing of BLOB values

CLI call	Short form	Function
SQL_BLOB_VAL_OPEN	SQLbvop	Open an access handle
SQL_BLOB_VAL_CLOSE	SQLbvcl	Close an access handle
SQL_BLOB_VAL_FETCH	SQLbvfe	Read a BLOB value sequentially

SQL_BLOB_VAL_STOW	SQLbvst	Set a BLOB value sequentially

Table 60: CLI call for individual sequences of BLOB values

Defining attribute values for dynamic INSERT statements

CLI call	Short form	Function
SQL_DIAG_SEQ_GET	SQLdsg	The RETURN INTO function of static INSERT statements is made available for dynamic INSERT statements

Table 61: CLI call for defining attribute values for dynamic INSERT statements

Example

A demonstration program for processing BLOB values by means of SESAM-CLI can be found in the library SIPANY.SESAM-SQL.090.MAN-DB. This is an ESQL-COBOL program from which C functions for executing CLI calls can be launched.

Below, we outline the steps required to create a BLOB object.

1. The REF value *ref_value* is output for the BLOB object class to which the new BLOB object is to belong. The BLOB object is to be located in the table named *table* in the schema named *schema*.

   ```
   SQL_BLOB_CLS_REF(table, schema, ref_value, &SQLDA)
   ```

2. The BLOB object is created by entering the REF value *ref_value* of the class and the name of the database *catalog*. The REF value *ref_value* of the new BLOB object is output.

   ```
   SQL_BLOB_OBJ_CREATE(ref_value, catalog, &SQLDA)
   ```

3. An access handle for writing is opened with ForWriteAccess=1. The REF value *ref_value* of the BLOB object and the database name *catalog* are specified. The access handle is identified in the following by the return value *access_handle*.

   ```
   SQL_BLOB_VAL_OPEN (ref_value, catalog,
                      (long int const *)&ForWriteAccess,
                      access_handle, &SQLDA)
   ```

4. The BLOB value is set sequentially within the access handle. *access_handle* is specified to identify the access handle. This step is repeated until the entire BLOB value has been read from the buffer.

   ```
   SQL_BLOB_VAL_STOW(access_handle, input_buffer,
                     (long int const *)&n, &SQLDA)
   ```

5. The access handle is closed. *access_handle* is specified to identify the access handle.

   ```
   SQL_BLOB_VAL_CLOSE (access_handle, &SQLDA)
   ```

9.2.2 Alphabetical reference section

In this section, the CLI calls are described using a uniform syntax. The calls are in alphabetical order. There is only one entry per call, which has the full name of the call and its short form as its header.

Each entry consists of several parts:

Full call name - short form

The function of the call is described following the heading.

This section also describes the access permissions required to successfully execute the call.

```
Function declaration in C
Function declaration in COBOL
parameter
   Explanation of the parameter.
```

The parameters are described in the order in which they appear in the function declaration.

9.2.3 SQL_BLOB_CLS_ISBTAB - SQLbcis

SQL_BLOB_CLS_ISBTAB checks whether or not a base table is a BLOB table. When the database, table and schema names are entered, the value 1 or 0 is output. If the value 1 is returned, this indicates that the table is a BLOB table. The value 0 signals syntax errors or indicates that the table is not a BLOB table.

This CLI call requires the SELECT privilege for the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_CLS_ISBTAB( char const *TableName
          ,char const *SchemaName
          ,char const *CatalogId
          ,long int *IsBlobTable
          ,struct SQLda_t *SQLda);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbcis IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 TableName PIC X(31).
   01 SchemaName PIC X(31).
   01 CatalogId PIC X(31).
   01 IsBLOBtable PIC S9(9)COMP.
   COPY SQLCA.          *> for group item SQLda.
PROCEDURE DIVISION USING TableName, SchemaName, CatalogId, IsBLOBtable,
          SQLda.
END PROGRAM SQLbcis.
```

`TableName`

Name of a base table. `TableName` must be the unqualified table name without the database and schema names (see section “Unqualified names”). This name is case-sensitive. If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte.

`SchemaName`

Name of the schema in which the base table is located. `SchemaName` must be the unqualified name of the schema excluding the database name (see section “Unqualified names”). This name is case-sensitive. If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte.

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`IsBLOBtable`

Boolean value. If the value 1 is returned, this indicates that the table is a BLOB table. The value 0 signals syntax errors or indicates that the table is not a BLOB table.

`SQLda`

Diagnostics area.

9.2.4 SQL_BLOB_CLS_REF - SQLbcre

SQL_BLOB_CLS_REF returns the class REF value for the objects in a BLOB table. Input is the table and schema name.

This CLI call does not require any privileges.

CLI declaration in C:

```
void SQL_BLOB_CLS_REF( char const *BlobTableName
          ,char const *BlobSchemaName
          ,char *REFvalue
          ,struct SQLda_t *SQLda);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbcre IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 BlobTableName PIC X(31).
   01 BlobSchemaName PIC X(31).
   01 REFvalue PIC X(237).
   COPY SQLCA.       *> for group item SQLda.
PROCEDURE DIVISION USING BlobTableName, BlobSchemaName, REFvalue, SQLda.
END PROGRAM SQLbcre.
```

`BlobTableName`

Name of the BLOB table `BlobTableName` must be the unqualified table name without the database and schema names (see section “Unqualified names”). This name is case-sensitive. If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte.

`BlobSchemaName`

Name of the schema in which the BLOB table is located. `BlobSchemaName` is the unqualified name of the schema excluding the database name (see section “Unqualified names”). This name is case-sensitive. If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte.

`REFvalue`

If the CLI call executes successfully, the class REF value is returned. If this REF value is less than 237 characters in length, it is padded with blanks up to this length. The exact structure of REF values is described on "Column definitions".

`SQLda`

Diagnostics area.

9.2.5 SQL_BLOB_OBJ_CLONE - SQLbocl

The SESAM/SQL CLI call SQL_BLOB_OBJ_CLONE creates a clone of an already existing BLOB object in another database. The clone has the same REF value as the original BLOB object and is created in a BLOB table with the same schema name and table name as the original. The attributes of the clone are set according to the defaults of its BLOB table. The BLOB value has length 0.

This SQL_BLOB_OBJ_CLONE call can be used to replicate a BLOB table in another database. If this is done by repeating the SQL_BLOB_OBJ_CREATE call on the other database. Then the REF values for original and copied objects are different, so that the references in the two databases are different. The SQL_BLOB_OBJ_CLONE call allows the creation of a clone with the same REF value as the original object, so that references in the two databases remain the same.

The SQL_BLOB_OBJ_CLONE call can also be used to recreate a BLOB object that has been deleted erroneously.

CLI declaration in C:

```
#define SQL_BLOB_OBJ_CLONE SQLBOCL
extern void SQL_BLOB_OBJ_CLONE( char const *REFvalue /* in */
          ,char const *CatalogId                     /* in */
          ,SQLda_t *sqlda);                          /* out */
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbocl IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   COPY SQLCA.
PROCEDURE DIVISION USING REFvalue, CatalogId, SQLda.
END PROGRAM SQLbocl.
```

`REFValue`

The entered `REFValue` must be a correct REF value, and must not refer to a class object. The BLOB table referenced by the REF value must exist in the given database. However, it must not contain the BLOB object related by the REF value. If the CLI call executes successfully, the returned REF value references a new BLOB object in the `CatalogId` database. The BLOB value has the length 0. The CREATED and UPDATED attributes contain the same timestamp, and the other attributes are set to the default values of the BLOB table.

`CatalogID`

Unqualified name of the database with the new BLOB table. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

9.2.6 SQL_BLOB_OBJ_CREATE - SQLbocr

SQL_BLOB_OBJ_CREATE creates a new BLOB and outputs the generated REF value. The input parameters include the database name and the REF value of the class to which the new BLOB is to belong. You can also specify the REF value of an existing object of this class. The following values are entered in the BLOB table when a new object is created:

- The BLOB is assigned an object number that is unique within that class. This object number is assigned sequentially.
- The BLOB attributes UPDATED and CREATED are assigned the current time stamp. All other attributes are set in accordance with their default values.

The BLOB value of the newly created BLOB has the length 0.

This CLI call requires the INSERT and SELECT privileges for BLOB tables, as well as the UPDATE privilege for the obj_ref column of the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_OBJ_CREATE( char *REFvalue
          ,char const *CatalogId
          ,struct SQLda_t *SQLda);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbocr IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   COPY SQLCA.      *> for group item SQLda.
PROCEDURE DIVISION USING REFvalue, CatalogId, SQLda.
END PROGRAM SQLbocr.
```

`REFvalue`

REF value of the class or of an existing object from the same class. The exact structure of REF values is described on "Column definitions".

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`SQLda`

Diagnostics area.

9.2.7 SQL_BLOB_OBJ_CREAT2 - SQLboc2

SQL_BLOB_OBJ_CREATE2 creates a new BLOB and outputs the generated REF value. The input parameters include the database name, the REF value of the class to which the new BLOB is to belong, and an interval for the object number. You can also specify the REF value of an existing object of this class. The following values are entered in the BLOB table when a new object is created:

- The BLOB is assigned an object number that is unique within that class. This object number is assigned within the specified interval in accordance with a specific algorithm. This ensures that the object numbers are distributed equally over the specified interval when there are multiple SQL_BLOBOBJ_CREAT2 calls.
- The BLOB attributes UPDATED and CREATED are assigned the current time stamp. All other attributes are set in accordance with their default values.

The BLOB value of the newly created BLOB has the length 0.

This CLI call requires the INSERT and SELECT privileges for BLOB tables, as well as the UPDATE privilege for the obj_ref column of the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_OBJ_CREAT2( char *REFvalue
          ,char const *CatalogId
          ,long int *MinObjectNmbr
          ,long int *MaxObjectNmbr
          ,struct SQLda_t *SQLda);
```

CLI-Deklaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLboc2 IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   01 MinObjectNmbr PIC S9(9) COMP.
   01 MaxObjectNmbr PIC S9(9) COMP.
   COPY SQLCA.     *> for group item SQLda.
PROCEDURE DIVISION USING REFvalue, MinObjectNmbr, MaxObjectNmbr, CatalogId,
          SQLda.
END PROGRAM SQLboc2.
```

`REFvalue`

REF value of the class or of an existing object from the same class. The exact structure of REF values is described on "Column definitions".

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`MinObjectNmbr`

Minimum value for the object number (must be >= 1).

`MaxObjectNmbr`

Maximum value for the object number (must be greater than or equal to the minimum value).

`SQLda`

Diagnostics area.

9.2.8 SQL_BLOB_OBJ_DROP - SQLbodr

SQL_BLOB_OBJ_DROP deletes an existing BLOB, together with its BLOB value and all its attributes. The input parameters include the database name and the REF value of the BLOB. The deletion of a BLOB actually consists of removing one or more rows from the BLOB table. If an error occurs in the process, this is reported back to the caller and the BLOB remains unchanged. (However, the UPDATED attribute can have been changed.) Concurrent transactions are synchronized as normal in SESAM/SQL.

This CLI call requires the DELETE and SELECT privileges for the BLOB table, as well as the UPDATE privilege for the slice_val column of the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_OBJ_DROP( char const *REFvalue
          ,char const *CatalogId
          ,struct SQLda_t *SQLda);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbodr IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   COPY SQLCA.     *> for group item SQLda.
PROCEDURE DIVISION USING REFvalue, CatalogId, SQLda.
END PROGRAM SQLbodr.
```

`REFvalue`

The REF value of the BLOB. The exact structure of REF values is described on "Column definitions".

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`SQLda`

Diagnostics area.

9.2.9 SQL_BLOB_TAG_GET - SQLbtge

SQL_BLOB_TAG_GET outputs the current value of an attribute of an existing BLOB. The input parameters include the REF value of the BLOB, the database name and the name of the attribute (tag). Possible tags include CREATED, UPDATED, MIME and USAGE. In addition, you must define a buffer into which the attribute value will be written, and specify its length. If the BLOB has no attribute with the specified tag, an error message is output.

This CLI call requires the SELECT privilege for the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_TAG_GET( char const *REFvalue
          ,char const *CatalogId
          ,char const *TagName
          ,char *Buffer
          ,long int const *BufferLength
          ,long int *ValueLength
          ,struct SQLda_t *SQLda);
```

CLI-Deklaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbtge IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   01 TagName PIC X(31).
   01 Buffer.     *> of any length
       02 PIC X(1).
   01 BufferLength PIC S9(9) COMP.
   01 ValueLength PIC S9(9) COMP.
   COPY SQLCA.     *> for group item SQLda.
PROCEDURE DIVISION USING REFvalue, CatalogId, TagName, Buffer, BufferLength,
          ValueLength, SQLda.
END PROGRAM SQLbtge.
```

`REFvalue`

The REF value of the BLOB. The exact structure of REF values is described on "Column definitions".

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`TagName`

Name of the attribute (tag). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. `TagName` may not be a blank string.

`Buffer`

Buffer to which the attribute value is to be written.

`BufferLength`

Length of the buffer in bytes. `BufferLength` must be a number >= 0. If the buffer length is less than that of the attribute value with trailing blanks removed, the buffer is filled up as far as its length permits. A message is output in this case.

`ValueLength`

Length of the attribute value read in bytes. If the length of the attribute value is greater than the value specified in `BufferLength`, only part of the attribute value will be transferred to the buffer.

`SQLda`

Diagnostics area.

9.2.10 SQL_BLOB_TAG_PUT - SQLbtpu

SQL_BLOB_TAG_PUT replaces an attribute value of an existing BLOB. The input parameters include the REF value, the database name and the name of the attribute (tag). The new attribute value must be located in a buffer. The address of which must be specified in the input parameters together with the value length.

This CLI call requires the SELECT privilege for the BLOB table, as well as the UPDATE privilege for the slice_val column of the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_TAG_PUT( char const *REFvalue
          ,char const *CatalogId
          ,char const *TagName
          ,char *Buffer
          ,long int const *ValueLength
          ,struct SQLda_t *SQLda);
```

CLI-Deklaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbtpu IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   01 TagName PIC X(31).
   01 Buffer.    *> of any length
       02 PIC X(1).
   01 ValueLength PIC S9(9) COMP.
   COPY SQLCA.     *> for group item SQLda.
PROCEDURE DIVISION USING REFvalue, CatalogId, TagName, Buffer, ValueLength,
          SQLda.
END PROGRAM SQLbtpu.
```

`REFvalue`

The REF value of the BLOB. The exact structure of REF values is described on "Column definitions".

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`TagName`

Name of the attribute (tag). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. `TagName` may not be a blank string.

`Buffer`

Buffer containing the new attribute value.

`ValueLength`

Length of the new attribute value. `ValueLength` must be a number >= 0.

`SQLda`

Diagnostics area as an output parameter.

9.2.11 SQL_BLOB_VAL_CLOSE - SQLbvcl

SQL_BLOB_VAL_CLOSE closes an access handle opened with SQL_BLOB_VAL_OPEN (see "SQL_BLOB_VAL_OPEN - SQLbvop").

An access handle allows you to process BLOB values sequentially. Here the calls SQL_BLOB_VAL_FETCH (see "SQL_BLOB_VAL_FETCH - SQLbvfe") for sequential reading and SQL_BLOB_VAL_STOW (see "SQL_BLOB_VAL_STOW - SQLbvst") for sequential writing are offered.

If you attempt to close an access handle that has already been closed, an error message is output.
This CLI call may require the INSERT privilege for the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_VAL_CLOSE( char *AccessHandle
                        ,struct SQLda_t *SQLda);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbvcl IS PROTOTYPE.
DATA DIVISION.
   01 AccessHandle PIC X(32).
   COPY SQLCA.     *> for group item SQLda.
LINKAGE SECTION.
PROCEDURE DIVISION USING AccessHandle, SQLda.
END PROGRAM SQLbvcl.
```

`AccessHandle`

The value supplied in SQL_BLOB_VAL_OPEN for the access handle to be terminated must be entered here. This value must not be modified by the caller.

`SQLda`

Diagnostics area.

9.2.12 SQL_BLOB_VAL_FETCH - SQLbvfe

SQL_BLOB_VAL_FETCH reads the individual segments of a BLOB value sequentially. Contrast this with the CLI call SQL_BLOB_VAL_GET (see "SQL_BLOB_VAL_GET - SQLbvge"), which reads the entire BLOB value in one go. The advantage of SQL_BLOB_VAL_FETCH over SQL_BLOB_VAL_GET is the fact that it allows the output buffer to be shorter than the BLOB value itself.

To read a BLOB value sequentially using SQL_BLOB_VAL_FETCH, you will need an access handle. This is created using the SQL_BLOB_VAL_OPEN call. With the `ForWriteAccess` parameter of this call you define that you require this access handle for reading (see "SQL_BLOB_VAL_OPEN - SQLbvop"f). Following the SQL_BLOB_VAL_OPEN call, SESAM/SQL returns a unique ID for the access handle.

This ID must be specified each time SQL_BLOB_VAL_FETCH is called. You must also define a buffer into which the BLOB value segments are to be written, and the length of this buffer.

The first time SQL_BLOB_VAL_FETCH is called within an access handle, the buffer is filled with the first segment of the BLOB value. The next time the call is issued with the same access handle, the buffer is filled with the next segment of the BLOB value, and so on. Once the entire BLOB value has been read, a message (SQLSTATE 02000) to this effect is output.

After the BLOB value has been read in its entirety, it will no longer be possible to call SQL_BLOB_VAL_FETCH with this access handle. The access handle must be closed using SQL_BLOB_VAL_CLOSE (see "SQL_BLOB_VAL_CLOSE - SQLbvcl").

The entire sequence of operations (SQL_BLOB_VAL_OPEN, repeated SQL_BLOB_VAL_FETCH calls, SQL_BLOB_VAL_CLOSE) must be executed within a transaction.

This CLI call does not require any privileges.

CLI declaration in C:

```
void SQL_BLOB_VAL_FETCH( char *AccessHandle
          ,char *Buffer
          ,long int const *BufferLength
          ,long int *ValueLength
          ,struct SQLda_t * SQLda);
```

CLI-Deklaration in Cobol:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbvfe IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 AccessHandle PIC X(32).
   01 Buffer.     *> of any length
       02 PIC X(1).
   01 BufferLength PIC S9(9) COMP.
   01 ValueLength PIC S9(9) COMP.
   COPY SQLCA.     *> for group item SQLda.
PROCEDURE DIVISION USING AccessHandle, Buffer, BufferLength, ValueLength,
          SQLda.
END PROGRAM SQLbvfe.
```

`AccessHandle`

ID assigned to the access handle in SQL_BLOB_VAL_OPEN. This value must not be modified by the caller.

`Buffer`

Buffer to which the BLOB value segment is to be written.

`BufferLength`

Length of the buffer in bytes. `BufferLength` must be a number >= 0.

`ValueLength`

Length of the BLOB value segment written to the buffer. If this is less than the value specified in `BufferLength`, this indicates that the BLOB value has been read in its entirety.

`SQLda`

Diagnostics area.

9.2.13 SQL_BLOB_VAL_GET - SQLbvge

SQL_BLOB_VAL_GET reads an entire BLOB value in one go. The input parameters include the REF value of the BLOB, the database name, the buffer to which the value is to be written and the length of this buffer.

This CLI call requires the SELECT privilege for the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_VAL_GET( char const *REFvalue
          ,char const *CatalogId
          ,char *Buffer
          ,long int const *BufferLength
          ,long int *ValueLength
          ,struct SQLda_t *SQLda);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbvge IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   01 Buffer.     *> of any length
       02 PIC X(1).
   01 BufferLength PIC S9(9) COMP.
   01 ValueLength PIC S9(9) COMP.
   COPY SQLCA.     *> for group item SQLda.
PROCEDURE DIVISION USING REFvalue, CatalogId, Buffer, BufferLength,
          ValueLength, SQLda.
END PROGRAM SQLbvge.
```

`REFvalue`

The REF value of the BLOB. The exact structure of REF values is described on "Column definitions".

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`Buffer`

Buffer to which the BLOB value is written.

`BufferLength`

Length of the buffer in bytes. `BufferLength` must be a number >= 0.

`ValueLength`

Length of the BLOB value. If this is greater than the value specified in `BufferLength`, the buffer will contain only the first few bytes of the BLOB value (up to the length `BufferLength`). Otherwise, the first few bytes of the buffer will contain the entire BLOB value.

`SQLda`

Diagnostics area.

9.2.14 SQL_BLOB_VAL_LEN - SQLbvle

When the REF value and database name are entered, SQL_BLOB_VAL_LEN determines the length of a BLOB value and displays this.

The SQL_BLOB_VAL_LEN call requires the SELECT privilege for the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_VAL_LEN( char const *REFvalue
          ,char const *CatalogId
          ,long int *ValueLength
          ,struct SQLda_t *SQLda);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbvle IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   01 ValueLength PIC S9(9) COMP.
   COPY SQLCA.     *> for group item SQLda.
PROCEDURE DIVISION USING REFvalue, CatalogId, ValueLength, SQLda.
END PROGRAM SQLbvle.
```

`REFvalue`

The REF value of the BLOB. The exact structure of REF values is described on "Column definitions".

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`ValueLength`

Length of the BLOB value.

`SQLda`

Diagnostics area.

9.2.15 SQL_BLOB_VAL_OPEN - SQLbvop

SQL_BLOB_VAL_OPEN opens an access handle. Access handles are used in the sequential processing of BLOB values. In SESAM/SQL you can read BLOB values sequentially with SQL_BLOB_VAL_FETCH (see "SQL_BLOB_VAL_FETCH - SQLbvfe") and write them sequentially with SQL_BLOB_VAL_STOW (see "SQL_BLOB_VAL_STOW - SQLbvst").

Once you have finished processing BLOB values sequentially using an access handle, you must close the access handle by calling SQL_BLOB_VAL_CLOSE (see "SQL_BLOB_VAL_CLOSE - SQLbvcl"). An access handle must be opened and closed within the same transaction.

By repeating the CLI calls SQL_BLOB_VAL_FETCH and SQL_BLOB_VAL_STOW, you can read or write BLOB values sequentially. To ensure that this is carried out correctly, you will require an access handle which manages the following information internally:

- the BLOB value to be addressed
- the progress of the read or write operation (which segment is to be read or written next)

The input parameters include the REF value of the BLOB, the database name and the `ForWriteAccess` parameter.
With `ForWriteAccess` you define whether the access handle is to be used for reading or writing purposes.
Each access handle is assigned a unique ID by SESAM/SQL, which must be specified each time SQL_BLOB_VAL_FETCH or SQL_BLOB_VAL_STOW is called.

It is possible to have up to 10 access handles open at any one time, i.e. to have up to 10 sequential processes involving BLOBs running in parallel. If you attempt to initiate an 11th sequential process, this will be rejected with a corresponding message.

If you fail to issue an SQL_BLOB_VAL_CLOSE call once sequential processing is complete, the access handle remains reserved. This makes it impossible to optimize the utilization of the 10 possible access handles.

If you open an access handle for writing purposes and the BLOB to be processed already has a BLOB value, the old BLOB value will be deleted by the SQL_BLOB_VAL_OPEN call, i.e. it will have the length 0. The BLOB itself will be retained.

You should avoid subjecting a particular BLOB to several parallel writing sequences, since these may have conflicting effects on the BLOB value.

If the SQL_BLOB_VAL_OPEN call is issued for reading purposes, you will require the SELECT privilege for the BLOB table. If it is issued for writing purposes, you will require the SELECT and DELETE privileges for the BLOB table, as well as the UPDATE privilege for the slice_val column of the BLOB table.

CLI-Deklaration in C:

```
void SQL_BLOB_VAL_OPEN( char const *REFvalue
            ,char const *CatalogId
            ,long int *ForWriteAccess
            ,char *AccessHandle
            ,struct SQLda_t *SQLda);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbvop IS PROTOTYPE.
DATA DIVISION.
```

```
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   01 ForWriteAccess PIC S9(9) COMP.
   01 AccessHandle PIC X(32).
   COPY SQLCA.     *> for group item SQLda.
PROCEDURE DIVISION USING REFvalue, CatalogId, ForWriteAccess, AccessHandle,
          SQLda.
END PROGRAM SQLbvop.
```

`REFvalue`

The REF value of the BLOB. The exact structure of REF values is described on "Column definitions".

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`ForWriteAccess`

This can be set to the value 1 (=TRUE) or 0 (=FALSE). The values have the following meaning:

0: The access handle is intended for reading purposes (with SQL_BLOB_VAL_FETCH).

1: The access handle is intended for writing purposes (with SQL_BLOB_VAL_STOW).

`AccessHandle`

ID of the access handle. This value must not be modified, as it will be used in all subsequent operations up to the concluding SQL_BLOB_VAL_CLOSE call.

`SQLda`

Diagnostics area.

9.2.16 SQL_BLOB_VAL_PUT - SQLbvpu

SQL_BLOB_VAL_PUT replaces one BLOB value with another contained in a buffer. The input values include the REF value of the BLOB, the database name, the buffer containing the new value and the length of the new value.

This CLI call requires the INSERT, SELECT and DELETE privileges for the BLOB table. You also require the UPDATE privilege for the slice_val column in the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_VAL_PUT( char const *REFvalue
          ,char const *CatalogId
          ,char *Buffer
          ,long int *ValueLength
          ,struct SQLda_t *SQLda);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbvpu IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 REFvalue PIC X(237).
   01 CatalogId PIC X(31).
   01 Buffer.      *> of any length
       02 PIC X(1).
   01 ValueLength PIC S9(9) COMP.
   COPY SQLCA.      *> for group item SQLda.
PROCEDURE DIVISION USING REFvalue, CatalogId, Buffer, ValueLength, SQLda.
END PROGRAM SQLbvge.
```

`REFvalue`

The REF value of the BLOB. The exact structure of REF values is described on "Column definitions".

`CatalogId`

Unqualified name of the database in which the table is located. `CatalogId` is an unqualified name (see section “Unqualified names”). If necessary, this name must be padded with blanks up to a length of 31 characters or terminated with a null byte. If you wish to use the default database name, simply enter a null byte or a string of blanks instead of the database name.

`Buffer`

Buffer containing the new BLOB value.

`ValueLength`

Length of the BLOB value. `ValueLength` must be a number >= 0.

`SQLda`

Diagnostics area.

9.2.17 SQL_BLOB_VAL_STOW - SQLbvst

SQL_BLOB_VAL_STOW writes a new BLOB value sequentially to a BLOB. Contrast this with the CLI call SQL_BLOB_VAL_PUT (see "SQL_BLOB_VAL_PUT - SQLbvpu"), which writes the entire BLOB value in one go. The advantage of SQL_BLOB_VAL_STOW over SQL_BLOB_VAL_PUT is the fact that it allows the buffer to be shorter than the new BLOB value as a whole. The new BLOB value will be transferred in small segments.

To write a BLOB value sequentially using SQL_BLOB_VAL_STOW, you will need an access handle. This is created using the SQL_BLOB_VAL_OPEN call. With the `ForWriteAccess` parameter of this call you define that you require this access handle for writing (see "SQL_BLOB_VAL_OPEN - SQLbvop"f). Following the SQL_BLOB_VAL_OPEN call, SESAM/SQL returns a unique ID for the access handle.

This ID must be specified each time SQL_BLOB_VAL_STOW is called. You must also define the buffer in which the new BLOB value segments are located, and the length of this buffer.

After the BLOB value has been written in its entirety by means of repeated SQL_BLOB_VAL_STOW calls, the access handle must be closed using SQL_BLOB_VAL_CLOSE (see "SQL_BLOB_VAL_CLOSE - SQLbvcl"). Only then will the final segment of the new BLOB value be inserted in the BLOB table.

The entire sequence of operations (SQL_BLOB_VAL_OPEN, repeated SQL_BLOB_VAL_STOW calls, SQL_BLOB_VAL_CLOSE) must be executed within a transaction.

This CLI call requires the INSERT privilege for the BLOB table.

CLI declaration in C:

```
void SQL_BLOB_VAL_STOW( char *AccessHandle
          ,char *Buffer
          ,long int const *ValueLength
          ,struct SQLda_t *SQLda);
```

CLI-Deklaration in Cobol:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLbvst IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   01 AccessHandle PIC X(32).
   01 Buffer.    *> of any length
       02 PIC X(1).
   01 ValueLength PIC S9(9) COMP.
   COPY SQLCA.   *> for group item SQLda.
PROCEDURE DIVISION USING AccessHandle, Buffer, ValueLength, SQLda.
END PROGRAM SQLbvst.
```

`AccessHandle`

ID assigned to the access handle in SQL_BLOB_VAL_OPEN. This value must not be modified by the caller.

`Buffer`

Buffer containing the new value.

`ValueLength`

Length of the value. `ValueLength` must be a number >= 0.

`SQLda`

Diagnostics area.

9.2.18 SQL_DIAG_SEQ_GET - SQLdsg

SQL_DIAG_SEQ_GET can be used to obtain the function of the RETURN INTO clause of static INSERT statements for dynamic INSERT statements.

SQL_DIAG SEQ_GET returns the value determined by SESAM/SQL while executing an INSERT statement with COUNT INTO or with '*' in the VALUES clause.

The function can also be used for static INSERT statements.

It is made available as LLM 'SQLDSG' in the SIPLIB.SESAM-SQL.091.CLI library. If it is to be used in an application program, this LLM must either be explicitly linked to it or the library must be specified as BLSLIBxx when the program is executed. The interfaces for this functions are made available in the library as S type elements *sqldsg.h* (for C) and *sqldsg* (for COBOL).

CLI declaration in C:

```
extern void SQL_DIAG_SEQ_GET( struct SQLda_t *SQLda
          ,char *SequenceValue
          ,signed short *RC);
```

CLI declaration in COBOL:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SQLdsg IS PROTOTYPE.
DATA DIVISION.
LINKAGE SECTION.
   COPY SQLCA .      *> for group item SQLda
   01 SequenceValue PIC X(34).
   01 RC PIC S9(4) BINARY.
PROCEDURE DIVISION USING SQLda, SQL_SequenceValue, RC.
END PROGRAM SQLdsg.
```

`SQLda`

SQL diagnostics area with which the INSERT statement was executed. If the SQLda of a COBOL/ESQL program is to be referenced from a C program, it must be specified in COBOL as EXTERNAL.

`SequenceValue`

Storage area into which the result is to be transferred. The area must be at least 34 Bytes in size.

If the transfer is successful (RC=0), 34 characters have been written into SequenceValue in the following format and with at least one digit:

[<space>...]{+|}[digit...][.][digit...]<space>

The format is selected in such a way:

- that the numerical value suitable for C variables of integer and floating point types can be obtained with the C functions `strtol()`, `atol()`, `srtod()`, `strtof()`, and `atof()`.
- that the numerical value suitable for any numeric COBOL variable can be obtained with the COBOL function NUMVAL.

In the case of RC 0, the contents of `SequenceValue` are not modified.

`RC`

Return value:

0	The desired value has been transferred to the storage area to which SequenceValue refers.
-1	The input parameters were incorrect or the SQLda was not recognized as diagnostics area.
100	It was not possible to determine a value assigned by SESAM/SQL. Possible causes:
	• The INSERT statement contained no COUNT INTO clause and no '*' in the VALUES clause.
	• Execution of the INSERT statement unsuccessful, with an SQLSTATE of the category exception condition.
	• The SQL statement last executed with the same SQLda was not an INSERT statement.

10 Information schemas

This chapter describes the information schemas that provide you with information on the structure of the database.

It describes the views of the INFORMATION_SCHEMA and of the SYS_INFO_SCHEMA.

10.1 Views of the INFORMATION_SCHEMA

In the INFORMATION_SCHEMA, you will find information on database objects. Each authorization identifier only has access to the objects for which it is authorized. The views of the INFORMATION_SCHEMA conform to the SQL standard with regard to objects defined in SESAM/SQL and in the SQL standard. The INFORMATION_SCHEMA includes additional views for SESAM/SQL extensions.

The table below indicates which view of the INFORMATION_SCHEMA contains information on which database object.

The views of the INFORMATION_SCHEMA are described in alphabetical order in the subsequent sections.

Object	View name	Information on
Schema	SCHEMATA	Schemas in the database
Table	TABLES	Tables in the database
	BASE_TABLES	Base tables in the database
	PARTITIONS	Partitions of the base tables
	VIEW_TABLE_USAGE	Tables on which the views are based
	CONSTRAINT_TABLE_USAGE	Tables on which integrity constraints are based
View	VIEWS	Views of the database
Column	COLUMNS	Columns in the database
	BASE_TABLE_COLUMNS	Columns in the base tables
	VIEW_COLUMN_USAGE	Columns on which views are based
	CONSTRAINT_COLUMN_USAGE	Columns on which integrity constraints are based
	INDEX_COLUMN_USAGE	Columns on which indexes are based
	KEY_COLUMN_USAGE	Columns for which a primary key or UNIQUE constraint is defined
Privilege	TABLE_PRIVILEGES	Table privileges
	COLUMN_PRIVILEGES	Column privileges
	CATALOG_PRIVILEGES	Special privileges
	USAGE_PRIVILEGES	USAGE privileges
	ROUTINE_PRIVILEGES	Privileges for routines
Index	INDEXES	Indexes in the database
Integrity constraint	TABLE_CONSTRAINTS	Integrity constraints
	REFERENTIAL_CONSTRAINTS	Referential constraints
	CHECK_CONSTRAINTS	Check constraints

Storage group	STOGROUPS	Storage groups in the database
Volume	STOGROUP_VOLUME_USAGE	Volumes used for storage groups
Space	SPACES	Spaces
Routines	PARAMETERS	Parameters of routines
	ROUTINES	Routines
	ROUTINE_ROUTINE_USAGE	Routines in other routines
	ROUTINE_TABLE_USAGE	Tables in routines
	ROUTINE_COLUMN_USAGE	Columns in routines
	VIEW_ROUTINE_USAGE	Routines in views
User	USERS	Authorization identifier
	SYSTEM_ENTRIES	System entries
DA-LOG-file	DA_LOGS	DA-LOG files
Media table	MEDIA_DESCRIPTIONS MEDIA_RECORDS	Media records of the database specific files
Recovery unit	RECOVERY_UNITS	Recovery units for spaces
Character set	CHARACTER_SETS	Character set
Sort sequence	COLLATIONS	Sort sequence
Transliteration	TRANSLATIONS	Transliterations
Features and Conformance	SQL_FEATURES SQL_IMPL_INFO SQL_LANGUAGES_S SQL_SIZING	Features, subfeatures, implementations, implemented host languages, embedments and implementation-specific maximum values

Table 62: Views of the INFORMATION_SCHEMA (section 2 of 2)

10.1.1 BASE_TABLES

Information on base tables. The current authorization identifier must have at least one table privilege for the base table or the UTILITY privilege for the database.

Column name	Data type	Contents
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table belongs
TABLE_NAME	CHAR (31)	Name of the base table
SPACE_NAME	CHAR (18)	Name of the space in which the base table is stored. If the table is partitioned, "_PARTITIONS_" is output as the name of the space.
TABLE_STYLE	VARCHAR (6)	OLDEST CALL DML only table OLD CALL DML/SQL table NEW SQL table

Table 63: BASE_TABLES view of the INFORMATION_SCHEMA

10.1.2 BASE_TABLE_COLUMNS

Information on base table columns. The current authorization identifier must have at least one column privilege for the column or the UTILITY privilege for the database.

Column name	Data type	Contents
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table belongs
TABLE_NAME	CHAR (31)	Name of the base table
COLUMN_NAME	CHAR (31)	Column name
ORDINAL_POSITION	SMALLINT	Sequence number of the column in the table
COLUMN_DEFAULT	VARCHAR (256)	Default value, as specified in the column definition (e.g. CHAR literal in single quotes) if the current authorization identifier owns the schema TRUNCATED if representation of the default value comprises more than 256 characters and the current authorization identifier owns the schema. The default value cannot be displayed. NULL value in all other cases
IS_NULLABLE	VARCHAR (3)	NO Column cannot accept NULLvalues under any circumstances YES else
DATA_TYPE	VARCHAR (24)	Data type of the column: CHARACTER CHARACTER VARYING NATIONAL CHAR NATIONAL CHAR VARYING REAL DOUBLE PRECISION FLOAT INTEGER SMALLINT NUMERIC DECIMAL

		DATE TIME TIMESTAMP OLDEST
CHARACTER _MAXIMUM_LENGTH	SMALLINT	Max. length of the column in code units if the data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR, NATIONAL CHAR VARYING or OLDEST NULL value in all other cases
NUMERIC_PRECISION	SMALLINT	Total number of significant digits for numeric data types NULL value in all other cases
NUMERIC_PRECISION _RADIX	SMALLINT	Radix for numeric data types NULL value in all other cases
NUMERIC_SCALE	SMALLINT	Number of digits right of the decimal point for exact numeric data types NULL value in all other cases
DATETIME_PRECISION	SMALLINT	Number of digits right of the decimal point for the data types TIME and TIMESTAMP NULL value in all other cases
The columns OLDEST_DESCRIPTOR* are assigned a value if DATA_TYPE is OLDEST:		
OLDEST_DESCRIPTOR1	CHAR (1)	Y left-aligned N not left-aligned NULL value if DATATYPE is not OLDEST
OLDEST_DESCRIPTOR2	CHAR (1)	Y Fill character N No fill character NULL value if DATATYPE is not OLDEST
OLDEST_DESCRIPTOR3	CHAR (1)	Y Null (0) permitted as value

		N Null (0) not permitted NULL value if DATATYPE is not OLDEST
OLDEST_DESCRIPTOR4	CHAR (1)	Y Value has arithmetic result N Value does not have arithmetic result NULL value if DATATYPE is not OLDEST
COLUMN_DESCRIPTOR1	CHAR (1)	Y Column has exactly one single column index and is not included in a compound index N Column has no index or more than one single-column index, or is included in a compound index
COLUMN_DESCRIPTOR2	CHAR (1)	Y Column has exactly one compound index and no single-column index N Column does not have an index, has more than one index, or only one single-column index
COLUMN_DESCRIPTOR3	CHAR (1)	Y Column has more than one index N Column has a maximum of one index
COLUMN_DESCRIPTOR4	CHAR (1)	Y Column has a CALL DML default value N Column does not have a CALL DML default value
COLUMN_DESCRIPTOR5	CHAR (1)	Y Column is a multiple column N Column is an atomic column
PK_DISTANCE	SMALLINT	Distance of the column to the start of the primary key NULL value if the column is not in the primary key
SESAM_SAN	CHAR (3)	Symbolic attribute name of the column

		NULL value if the column is defined in the SQL table
SESAM_DEFAULT	CHAR (2)	CALL DML default (with sign, if necessary, if numeric data type) NULL value if the column is defined in the SQL table
FIRST_OCCURRENCE	SMALLINT	First possible occurrence of a multiple column (= 1) NULL value if the column is not multiple
LAST_OCCURRENCE	SMALLINT	Last possible occurrence of a multiple column NULL value if the column is not multiple

Table 64: BASE_TABLE_COLUMNS view of the INFORMATION_SCHEMA (section 4 of 4)

10.1.3 CATALOG_PRIVILEGES

Information on the database privileges available to the current authorization identifier or which can be granted by the current authorization identifier.

Column name	Data type	Contents
GRANTOR	CHAR (18)	Authorization identifier that granted the privilege or _SYSTEM
GRANTEE	CHAR (18)	Authorization identifier granted the privilege or PUBLIC
CATALOG_NAME	CHAR (18)	Database name
PRIVILEGE_TYPE	CHAR (18)	Privilege type: CREATE USER CREATE SCHEMA CREATE STOGROUP UTILITY
IS_GRANTABLE	VARCHAR (3)	YES The authorization identifier has GRANT authorization for the privilege NO No GRANT authorization

Table 65: CATALOG_PRIVILEGES view of the INFORMATION_SCHEMA

10.1.4 CHARACTER_SETS

Information on character sets available to the current authorization identifier.

Column name	Data type	Contents
CHARACTER_SET_CATALOG	CHAR (18)	Database name
CHARACTER_SET_SCHEMA	CHAR (31)	INFORMATION_SCHEMA
CHARACTER_SET_NAME	CHAR (18)	UTF16, EBCDIC, SQL_TEXT SQL_CHARACTER, SQL_IDENTIFIER
FORM_OF_USE	CHAR (18)	EBCDIC, UTF16
NUMBER_OF_CHARACTERS	INTEGER	With FORM_OF_USE= EBCDIC the number of characters in the character set, NULL value in all other cases
DEFAULT_COLLATE_CATALOG	CHAR (18)	Database name
DEFAULT_COLLATE_SCHEMA	CHAR (31)	INFORMATION_SCHEMA
DEFAULT_COLLATE_NAME	CHAR (18)	EBCDIC_BINARY UTF16_BINARY

Table 66: CHARACTER_SETS view of the INFORMATION_SCHEMA

10.1.5 CHECK_CONSTRAINTS

Information on check constraints belonging to the current authorization identifier, as well as the corresponding check search condition.

Column name	Data type	Contents
CONSTRAINT_CATALOG	CHAR (18)	Database name
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the check constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the check constraint
CHECK_CLAUSE	VARCHAR (32000)	Search condition

Table 67: CHECK_CONSTRAINTS view of the INFORMATION_SCHEMA

10.1.6 COLLATIONS

Information on the sort sequences available to the current authorization identifier.

Column name	Data type	Contents
COLLATION_CATALOG	CHAR (18)	Database name
COLLATION_SCHEMA	CHAR (31)	INFORMATION_SCHEMA
COLLATION_NAME	CHAR (18)	DUCET_NO_VARS DUCET_WITH_VARS
CHARACTER_SET _CATALOG	CHAR (18)	Database name
CHARACTER_SET _SCHEMA	CHAR (31)	INFORMATION_SCHEMA
CHARACTER_SET_NAME	CHAR (18)	UTF16
PAD_ATTRIBUTE	CHAR (9)	NO PAD

Table 68: COLLATIONS view of the INFORMATION_SCHEMA

10.1.7 COLUMNS

Information on all the columns for which the current authorization identifier has privileges.

Column name	Data type	Contents
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table belongs
TABLE_NAME	CHAR (31)	Name of the base table or view
COLUMN_NAME	CHAR (31)	Column name
ORDINAL_POSITION	SMALLINT	Sequence number of the column in the table
COLUMN_DEFAULT	VARCHAR (256)	For base tables only: Default value, as specified in the column definition (e.g. CHAR literal in single quotes) if the current authorization identifier owns the schema. TRUNCATED if representation of the default value comprises more than 256 characters and the current authorization identifier owns the schema. The default value cannot be displayed. NULL value in all other cases
IS_NULLABLE	VARCHAR (3)	NO Column cannot accept NULL values under any circumstances YES else
DATA_TYPE	VARCHAR (24)	Data type of the column: CHARACTER CHARACTER VARYING NATIONAL CHAR NATIONAL CHAR VARYING REAL DOUBLE PRECISION FLOAT INTEGER SMALLINT NUMERIC DECIMAL

		DATE TIME TIMESTAMP OLDEST
CHARACTER _MAXIMUM_LENGTH	SMALLINT	Max. length of the column in code units if the data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR, NATIONAL CHAR VARYING or OLDEST NULL value in all other cases
CHARACTER_OCTET _LENGTH	SMALLINT	Max. length of the column in bytes if the data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR, NATIONAL CHAR VARYING or OLDEST NULL value in all other cases
NUMERIC_PRECISION	SMALLINT	Total number of significant digits for numeric data types NULL value in all other cases
NUMERIC_PRECISION _RADIX	SMALLINT	Radix for numeric data types NULL value in all other cases
NUMERIC_SCALE	SMALLINT	Number of digits right of the decimal point for exact numeric data types NULL value in all other cases
DATETIME _PRECISION	SMALLINT	Number of digits right of the decimal point for the data types TIME and TIMESTAMP NULL value in all other cases
CHARACTER_SET _CATALOG	CHAR (18)	Database name if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
CHARACTER_SET _SCHEMA	CHAR (31)	INFORMATION_SCHEMA

		if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
CHARACTER_SET _NAME	CHAR (18)	EBCDIC if data type is CHARACTER or CHARACTER VARYING UTF16 if data type is NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
COLLATION_ CATALOG	CHAR (18)	Database name if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
COLLATION_SCHEMA	CHAR (31)	INFORMATION_SCHEMA if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
COLLATION_NAME	CHAR (18)	EBCDIC_BINARY if data type is CHARACTER or CHARACTER VARYING UTF16_BINARY if data type is NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
DOMAIN_CATALOG	CHAR (18)	NULL value
DOMAIN_SCHEMA	CHAR (31)	NULL value
DOMAIN_NAME	CHAR (31)	NULL value
FIRST_OCCURRENCE	SMALLINT	First possible occurrence of a multiple column (for base table = 1) NULL value if the column is not multiple
LAST_OCCURRENCE	SMALLINT	Last possible occurrence of a multiple column

		NULL value if the column is not multiple

Table 69: COLUMNS view of the INFORMATION_SCHEMA (section 4 of 4)

10.1.8 COLUMN_PRIVILEGES

Information on all column privileges that the current authorization identifier has or which it has granted.

Column name	Data type	Contents
GRANTOR	CHAR (18)	Authorization identifier that granted the privilege or _SYSTEM
GRANTEE	CHAR (18)	Authorization identifier granted the privilege or PUBLIC
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema containing the column to which the privilege applies
TABLE_NAME	CHAR (31)	Name of the table for whose column the privilege applies
COLUMN_NAME	CHAR (31)	Name of the column to which the privilege was restricted
PRIVILEGE_TYPE	CHAR (18)	Privilege type: SELECT INSERT UPDATE REFERENCES
IS_GRANTABLE	VARCHAR (3)	YES The authorization identifier has GRANT authorization for the privilege NO No GRANT authorization

Table 70: COLUMN_PRIVILEGES view of the INFORMATION_SCHEMA

10.1.9 CONSTRAINT_COLUMN_USAGE

Information on columns that belong to the current authorization identifier and which are used in integrity constraints (except columns that are referenced in referential constraints).

Column name	Data type	Contents
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table belongs
TABLE_NAME	CHAR (31)	Name of the table referenced in the integrity constraint
COLUMN_NAME	CHAR (31)	Column name
CONSTRAINT_CATALOG	CHAR (18)	Database name
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the integrity constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the integrity constraint

Table 71: CONSTRAINT_COLUMN_USAGE view of the INFORMATION_SCHEMA

10.1.10 CONSTRAINT_TABLE_USAGE

Information on tables that belong to the current authorization identifier and which are referenced in check or referential constraints. Only the referenced tables are displayed for referential constraints.

Column name	Data type	Contents
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table belongs
TABLE_NAME	CHAR (31)	Name of the table referenced in the integrity constraint
CONSTRAINT_CATALOG	CHAR (18)	Database name
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the integrity constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the integrity constraint

Table 72: CONSTRAINT_TABLE_USAGE view of the INFORMATION_SCHEMA

10.1.11 DA_LOGS

Information on the DA-LOG files in a database. The current authorization identifier must have the UTILITY privilege for the database or must own at least one user space in the database.

Column name	Data type	Contents
DALOG_CATALOG	CHAR (18)	Database name
DALOG_VERSION	INTEGER	Version number of the DA-LOG file
DALOG_SUBNUMBER	INTEGER	Sequence number of the DA-LOG file within the version
DALOG_INIT	TIMESTAMP (3)	Time of creation

Table 73: DA_LOGS view of the INFORMATION_SCHEMA

10.1.12 INDEXES

Information on the indexes belonging to the current authorization identifier. The current authorization identifier must have the UTILITY privilege for the database or must own the schema in which the index is defined.

Column name	Data type	Contents
INDEX_CATALOG	CHAR (18)	Database name
INDEX_SCHEMA	CHAR (31)	Name of the schema to which the index belongs
INDEX_NAME	CHAR (18)	Name of the index
TABLE_NAME	CHAR (31)	Name of the base table to which the index belongs
SPACE_NAME	CHAR (18)	Name of the space in which the index is stored
LENGTH_I	SMALLINT	Total length of the index
CONSTRAINT_NAME	CHAR (31)	Name of the UNIQUE constraint if the index is used by a UNIQUE constraint. NULL value in all other cases
STATE	VARCHAR (9)	Status: GENERATED DEFECT
GENERATE_TYPE	VARCHAR (8)	as generated: EXPLICIT IMPLICIT
STATISTICS_INFO	VARCHAR (3)	YES Statistics information exists NO Statistics information does not exist
INDEX_TYPE	VARCHAR (8)	Index type: SINGLE COMPOUND

Table 74: INDEXES view of the INFORMATION_SCHEMA

10.1.13 INDEX_COLUMN_USAGE

Information on the columns in the indexes belonging to the current authorization identifier.

Column name	Data type	Contents
INDEX_CATALOG	CHAR (18)	Database name
INDEX_SCHEMA	CHAR (31)	Name of the schema to which the index belongs
INDEX_NAME	CHAR (18)	Name of the index
TABLE_NAME	CHAR (31)	Name of the base table to which the index belongs
COLUMN_NAME	CHAR (31)	Name of the column in the index
ORDINAL_POSITION	SMALLINT	Position of the column in the index
LENGTH_C	SMALLINT	Indicates the length (in bytes) to which the column is included in the index
INDEX_DISTANCE	SMALLINT	Distance of the column to the index start
DATE_TYPE_C	VARCHAR (24)	Data type of the column CHARACTER CHARACTER VARYING NATIONAL CHAR NATIONAL CHAR VARYING REAL DOUBLE PRECISION FLOAT INTEGER SMALLINT NUMERIC DECIMAL DATE TIME TIMESTAMP OLDEST

Table 75: INDEX_COLUMN_USAGE view of the INFORMATION_SCHEMA

10.1.14 KEY_COLUMN_USAGE

Information on primary key and UNIQUE constraints belonging to the current authorization identifier, as well as the name of the corresponding columns.

This view also contains information on referential constraints belonging to the current authorization identifier, as well as the names of the referencing columns.

Column name	Data type	Contents
CONSTRAINT_CATALOG	CHAR (18)	Database name
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the integrity constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the integrity constraint
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table belongs
TABLE_NAME	CHAR (31)	Name of the table to which the integrity constraint belongs
COLUMN_NAME	CHAR (31)	Name of a column in the integrity constraint
ORDINAL_POSITION	SMALLINT	Position of the column in the integrity constraint

Table 76: KEY_COLUMN_USAGE view of the INFORMATION_SCHEMA

10.1.15 MEDIA_DESCRIPTIONS

Information on file attributes for database-specific files. The current authorization identifier must have the UTILITY privilege for the database.

Column name	Data type	Contents
MEDIA_CATALOG	CHAR (18)	Database name
FILE_TYPE	CHAR (6)	File type: DALOG CATLOG PBI CATREC DDLTA
REQUESTS	VARCHAR (3)	YES Volume can be requested at console NO Volume cannot be requested at console
PRIMARY_ALLOC	INTEGER	Primary allocation
SECONDARY_ALLOC	INTEGER	Secondary allocation
SHARABLE	VARCHAR (3)	File sharable: YES NO

Table 77: MEDIA_DESCRIPTIONS view of the INFORMATION_SCHEMA

10.1.16 MEDIA_RECORDS

Information on volume types for database-specific files. The current authorization identifier must have the UTILITY privilege for the database.

Column name	Data type	Contents
MEDIA_CATALOG	CHAR (18)	Database name
FILE_TYPE	CHAR (6)	File type: DALOG CATLOG PBI CATREC DDLTA
DEVICE_DESCRIPTOR	CHAR (18)	Device type or name of the storage group for the file
MEDIUM	CHAR (4)	DISC
ORDINAL_POSITION	SMALLINT	Sequence number of the entry in the media table

Table 78: MEDIA_RECORDS view of the INFORMATION_SCHEMA

10.1.17 PARAMETERS

Information on parameters of routines (procedures and UDFs) for which the current authorization identifier has privileges.

Column name	Data type	Contents
SPECIFIC_CATALOG	CHAR(18)	Database name
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
ORDINAL_POSITION	SMALLINT	Sequence number of the parameter in the routine
PARAMETER_MODE	VARCHAR(5)	IN input parameter OUT output parameter INOUT input and output parameter
IS_RESULT	VARCHAR(3)	NO irrelevant for SESAM/SQL
AS_LOCATOR	VARCHAR(3)	NO irrelevant for SESAM/SQL
PARAMETER_NAME	CHAR(31)	Name of the parameter
DATA_TYPE	VARCHAR(24)	Data type of the parameter CHARACTER CHARACTER VARYING NATIONAL CHAR NATIONAL CHAR VARYING REAL DOUBLE PRECISION FLOAT INTEGER SMALLINT NUMERIC DECIMAL DATE TIME TIMESTAMP
CHARACTER_ MAXIMUM_LENGTH	SMALLINT	Max. length of the parameter in code units if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases

CHARACTER_OCTET_ LENGTH	SMALLINT	Max. length of the parameter in bytes if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
CHARACTER_SET_ CATALOG	CHAR(18)	Database name if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
CHARACTER_SET_ SCHEMA	CHAR(31)	INFORMATION_SCHEMA if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
CHARACTER_SET_ NAME	CHAR(18)	EBCDIC if data type is CHARACTER or CHARACTER VARYING UTF16 if data type is NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
COLLATION_CATALOG	CHAR(18)	Database name if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
COLLATION_SCHEMA	CHAR(31)	INFORMATION_SCHEMA if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
COLLATION_NAME	CHAR(18)	EBCDIC_BINARY,

		if data type is CHARACTER or CHARACTER VARYING UTF16_BINARY, if data type is NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
NUMERIC_PRECISION	SMALLINT	Total number of significant digits for numeric data types NULL value in all other cases
NUMERIC_PRECISION_ RADIX	SMALLINT	Radix for numeric data types NULL value in all other cases
NUMERIC_SCALE	SMALLINT	Number of digits right of the decimal point for exact numeric data types NULL value in all other cases
DATETIME_PRECISION	SMALLINT	Number of digits right of the decimal point for the data types TIME and TIMESTAMP NULL value in all other cases

Table 79: PARAMETERS view of the INFORMATION_SCHEMA (section 3 of 3)

10.1.18 PARTITIONS

Information on table partitions. The current authorization identifier must have the UTILITY privilege for the database or be the owner of the table.

Column name	Data type	Contents
PARTITION_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the partitioned table belongs
TABLE_NAME	CHAR (31)	Name of the partitioned table
SERIAL_NUMBER	SMALLINT	Sequence number of the partition
MAX_KEY_VALUE	VARCHAR (32000)	Comparison for the upper partition boundary as specified in the VALUE clause (external presentation)
SPACE_NAME	CHAR (18)	Name of the space in which the partition is stored

Table 80: PARTITIONS view of the INFORMATION_SCHEMA

10.1.19 RECOVERY_UNITS

Information on recovery units for spaces. The current authorization identifier must have the UTILITY privilege for the database or must own the space.

Column name	Data type	Contents
SPACE_CATALOG	CHAR (18)	Database name
SPACE_NAME	CHAR (18)	Name of the space
RECOVERY_TIMESTAMP	TIMESTAMP (3)	Time of the recovery operation
VERSION	INTEGER	Internal number if RECOVERY_TYPE is COPY NULL value in all other cases
VALIDITY	VARCHAR (3)	YES Recovery unit valid for recovery operations up to next recovery unit NO invalid (may however change to YES after a RECOVER statement) NOT invalid (cannot change)
RECOVERY_UNIT_NAME	VARCHAR (54)	File name of the copy if RECOVERY_TYPE is COPY Internal number if RECOVERY_TYPE is RESTART or REST_TO NULL value in all other cases
SPACE_OWNER	CHAR (18)	Authorization identifier that owns the space
MEDIUM	CHAR (4)	DISC SESAM backup on disk TAPE SESAM backup with ARCHIVE HSMW SESAM backup with HSMS (work file) HSMB SESAM backup with HSMS (additional mirror unit)

<table>
<tr><td></td><td></td><td colspan="2">SRDF SESAM backup with HSMS (SRDF target)
if RECOVERY_TYPE is COPY
NULL value in all other cases</td></tr>
<tr><td>RECOVERY_TYPE</td><td>VARCHAR (7)</td><td colspan="2">Values evaluated by the recovery utility:
COPY
CREATE
RESTART
REST_TO (RESTART TO)
MARK</td></tr>
<tr><td>COPY_TYPE</td><td>VARCHAR (7)</td><td colspan="2">ONLINE or OFFLINE
if RECOVERY_TYPE is COPY
NULL value in all other cases</td></tr>
<tr><td>DALOG_VERSION</td><td>INTEGER</td><td>Version number of the DA-LOG file</td><td rowspan="2">DA-LOG level before the recovery unit is entered</td></tr>
<tr><td>DALOG_SUBNUMBER</td><td>INTEGER</td><td>Sequence number of the DA-LOG file within the version</td></tr>
<tr><td>NEXT_DALOG_VERSION</td><td>INTEGER</td><td>Version number of the DA-LOG file</td><td rowspan="2">DA-LOG level after the recovery unit is entered</td></tr>
<tr><td>NEXT_DALOG
_SUBNUMBER</td><td>INTEGER</td><td>Sequence number of the DA-LOG file within the version</td></tr>
<tr><td>ARCHIVE_DIRECTORY
_NAME</td><td>VARCHAR (54)</td><td colspan="2">Name of the ARCHIVE directory,
if MEDIUM = 'TAPE'
Name of the HSMS archive,
if MEDIUM = 'HSMS', 'HSMW',
'HSMB' or 'SRDF'
NULL value in all other cases</td></tr>
<tr><td>PBI_TIMESTAMP</td><td>TIMESTAMP (3)</td><td colspan="2">Time at which the PBI file was generated</td></tr>
<tr><td>PBI_COUNTER</td><td>INTEGER</td><td colspan="2">undefined</td></tr>
</table>

Table 81: RECOVERY_UNITS view of the INFORMATION_SCHEMA (section 3 of 3)

10.1.20 REFERENTIAL_CONSTRAINTS

Information on referential constraints belonging to the current authorization identifier, as well as the name of the referenced UNIQUE or primary key constraint.

Column name	Data type	Contents
CONSTRAINT_CATALOG	CHAR (18)	Database name
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the referential constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the referential constraint
UNIQUE_CONSTRAINT _CATALOG	CHAR (18)	Database name
UNIQUE_CONSTRAINT _SCHEMA	CHAR (31)	Name of the schema of the referenced table
UNIQUE_CONSTRAINT _NAME	CHAR (31)	Name of the UNIQUE or primary key constraint of the referenced table
MATCH_OPTION	CHAR (7)	NONE
UPDATE_RULE	CHAR (11)	NO ACTION
DELETE_RULE	CHAR (11)	NO ACTION

Table 82: REFERENTIAL_CONSTRAINTS view of the INFORMATION_SCHEMA

10.1.21 ROUTINES

Information on routines (procedures and UDFs) for which the current authorization identifier has privileges.

Column name	Data type	Contents
SPECIFIC_CATALOG	CHAR(18)	Database name
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
ROUTINE_CATALOG	CHAR(18)	Database name
ROUTINE_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
ROUTINE_NAME	CHAR(31)	Name of the routine.
ROUTINE_TYPE	VARCHAR(28)	PROCEDURE if a procedure FUNCTION if a UDF
DATA_TYPE	VARCHAR(24)	Data type of the return value of a UDF CHARACTER CHARACTER VARYING NATIONAL CHAR NATIONAL CHAR VARYING REAL DOUBLE PRECISION FLOAT INTEGER SMALLINT NUMERIC DECIMAL DATE TIME TIMESTAMP NULL value, if a procedure
CHARACTER_ MAXIMUM_LENGTH	SMALLINT	Max. length of the return value in code units if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases

CHARACTER_OCTET_ LENGTH	SMALLINT	Max. length of the return value in bytes if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
CHARACTER_SET_ CATALOG	CHAR(18)	Database name if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
CHARACTER_SET_ SCHEMA	CHAR(31)	INFORMATION_SCHEMA if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
CHARACTER_SET_ NAME	CHAR(18)	EBCDIC if data type is CHARACTER or CHARACTER VARYING UTF16 if data type is NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
COLLATION_CATALOG	CHAR(18)	Database name if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
COLLATION_SCHEMA	CHAR(31)	INFORMATION_SCHEMA if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
COLLATION_NAME	CHAR(18)	EBCDIC_BINARY,

		if data type is CHARACTER or CHARACTER VARYING UTF16_BINARY, if data type is NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
NUMERIC_PRECISION	SMALLINT	Total number of significant digits for numeric data types NULL value in all other cases
NUMERIC_PRECISION_ RADIX	SMALLINT	Radix for numeric data types NULL value in all other cases
NUMERIC_SCALE	SMALLINT	Number of digits right of the decimal point for exact numeric data types NULL value in all other cases
DATETIME_PRECISION	SMALLINT	Number of digits right of the decimal point for the data types TIME and TIMESTAMP NULL value in all other cases
ROUTINE_BODY	VARCHAR(8)	SQL Programming language in which the routine was written
ROUTINE_DEFINITION	VARCHAR(32000)	Text of the routine if the current authorization identifier owns the schema NULL value in all other cases
EXTERNAL_NAME	CHAR(31)	NULL value, irrelevant for SESAM/SQL
EXTERNAL_LANGUAGE	VARCHAR(7)	NULL value, irrelevant for SESAM/SQL
PARAMETER_STYLE	VARCHAR(7)	NULL value, irrelevant for SESAM/SQL
IS_DETERMINISTIC	VARCHAR(3)	NO irrelevant for SESAM/SQL
SQL_DATA_ACCESS	VARCHAR(17)	CONTAINS SQL if CONTAINS SQL was specified in the definition of the routine

		READS SQL DATA if READS SQL DATA was specified in the definition of the routine MODIFIES SQL DATA if MODIFIES SQL DATA was specified in the definition of the routine
IS_NULL_CALL	VARCHAR(3)	NO if a UDF NULL value in all other cases
SQL_PATH	VARCHAR(256)	SQL path In SESAM/SQL, the same as the name of the schema in which the routine is defined
SCHEMA_LEVEL_ ROUTINE	VARCHAR(3)	YES Is part of a schema
MAX_DYNAMIC_ RESULT_SETS	SMALLINT	0 irrelevant for SESAM/SQL
IS_USER_DEFINED_ CAST	VARCHAR(3)	NO if a UDF NULL value in all other cases
IS_IMPLICITLY_ INVOCABLE	VARCHAR(3)	NULL value, irrelevant for SESAM/SQL
SECURITY_TYPE	VARCHAR(22)	NULL value, irrelevant for SESAM/SQL
AS_LOCATOR	VARCHAR(3)	NO if a UDF NULL value in all other cases
NEW_SAVEPOINT_ LEVEL	VARCHAR(3)	NULL value, irrelevant for SESAM/SQL
IS_UDT_DEPENDENT	VARCHAR(3)	NO irrelevant for SESAM/SQL

Table 83: ROUTINES view of the INFORMATION_SCHEMA

10.1.22 ROUTINE_COLUMN_USAGE

Information on the routines (procedures and UDFs) that reference columns belonging to the current authorization identifier, as well as the names of the columns.

Column name	Data type	Contents
SPECIFIC_CATALOG	CHAR(18)	Database name
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
ROUTINE_CATALOG	CHAR(18)	Database name
ROUTINE_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
ROUTINE_NAME	CHAR(31)	Name of the routine.
TABLE_CATALOG	CHAR(18)	Database name
TABLE_SCHEMA	CHAR(31)	Name of the schema to which the table which is addressed in the routine belongs
TABLE_NAME	CHAR(31)	Name of the table used in the routine
COLUMN_NAME	CHAR(31)	Column name

Table 84: ROUTINE_COLUMN_USAGE view of the INFORMATION_SCHEMA

10.1.23 ROUTINE_PRIVILEGES

Information on the privileges for routines (procedures and UDFs) which the current authorization identifier has or which it has granted.

Column name	Data type	Contents
GRANTOR	CHAR(18)	Authorization identifier which granted the privilege or _SYSTEM
GRANTEE	CHAR(18)	Authorization identifier granted the privilege or PUBLIC
SPECIFIC_CATALOG	CHAR(18)	Database name
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
ROUTINE_CATALOG	CHAR(18)	Database name
ROUTINE_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
ROUTINE_NAME	CHAR(31)	Name of the routine.
PRIVILEGE_TYPE	CHAR(18)	EXECUTE
IS_GRANTABLE	VARCHAR(3)	YES The authorization identifier has GRANT authorization for the privilege NO No GRANT authorization

Table 85: ROUTINE_PRIVILEGES view of the INFORMATION_SCHEMA

10.1.24 ROUTINE_ROUTINE_USAGE

Information on the routines (procedures and UDFs) that belong to the current authorization identifier and are called in other routines.

Column name	Data type	Contents
SPECIFIC_CATALOG	CHAR(18)	Database name
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the calling routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the calling routine
ROUTINE_CATALOG	CHAR(18)	Database name
ROUTINE_SCHEMA	CHAR(31)	Name of the schema to which the called routine belongs
ROUTINE_NAME	CHAR(31)	Specific name of the called routine

Table 86: ROUTINE_ROUTINE_USAGE view of the INFORMATION_SCHEMA

10.1.25 ROUTINE_TABLE_USAGE

Information on the tables that belong to the current authorization identifier and which are addressed in routines (procedures and UDFs).

Column name	Data type	Contents
SPECIFIC_CATALOG	CHAR(18)	Database name
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
ROUTINE_CATALOG	CHAR(18)	Database name
ROUTINE_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
ROUTINE_NAME	CHAR(31)	Name of the routine.
TABLE_CATALOG	CHAR(18)	Database name
TABLE_SCHEMA	CHAR(31)	Name of the schema to which the table which is addressed in the routine belongs
TABLE_NAME	CHAR(31)	Name of the table used in the routine

Table 87: View ROUTINE_TABLE_USAGE view of the INFORMATION_SCHEMA

10.1.26 SCHEMATA

Information on all the schemas that belong to the current authorization identifier.

Column name	Data type	Contents
CATALOG_NAME	CHAR (18)	Database name
SCHEMA_NAME	CHAR (31)	Name of the schema
SCHEMA_OWNER	CHAR (18)	Authorization identifier of the owner
DEFAULT_CHARACTER_ SET_CATALOG	CHAR (18)	Database name
DEFAULT_CHARACTER_ SET_SCHEMA	CHAR (31)	INFORMATION_SCHEMA
DEFAULT_CHARACTER_ SET_NAME	CHAR (18)	EBCDIC

Table 88: SCHEMATA view of the INFORMATION_SCHEMA

10.1.27 SPACES

Information on the spaces that belong to the current authorization identifier, and which the current authorization identifier can access via utilities.

Column name	Data type	Contents
SPACE_CATALOG	CHAR (18)	Database name
SPACE_NAME	CHAR (18)	Name of the space
SPACE_OWNER	CHAR (18)	Authorization identifier that owns the space
STOGROUP_NAME	CHAR (18)	Name of the storage group for the space
PCT_FREE	SMALLINT	Free space reservation in percent
LOGGING	VARCHAR (3)	YES Logging activated NO Logging deactivated
The data for STOGROUP_NAME and PCT_FREE can be modified by ALTER SPACE; this data is not actually taken into account until RECOVER or REORG is executed.		

Table 89: SPACES view of the INFORMATION_SCHEMA

10.1.28 SQL_FEATURES

Information on SQL features and their subfeatures in the implemented language environment. All designations are defined in the SQL standard.

Column name	Data type	Contents
FEATURE_ID	VARCHAR (256)	Feature ID (e.g. F111)
FEATURE_NAME	VARCHAR (256)	Feature name (e.g. “Isolation levels other than SERIALIZABLE”)
SUB_FEATURE_ID	VARCHAR (256)	Subfeature ID (e.g. 02)
SUB_FEATURE_NAME	VARCHAR (256)	Subfeature name (e.g. “READ COMITTED isolation level”)
IS_SUPPORTED	VARCHAR (3)	YES Fully supported NO Not supported or only partially supported
IS_VERIFIED_BY	VARCHAR (256)	NULL value
COMMENTS	VARCHAR (256)	Comments

Table 90: SQL_FEATURES view of the INFORMATION_SCHEMA

10.1.29 SQL_IMPL_INFO

Information on the properties of the implementation. All designations are defined in the SQL standard.

Column name	Data type	Contents
IMPL_INFO_ID	VARCHAR (256)	ID of an implementation property
IMPL_INFO_NAME	VARCHAR (256)	Name of an implementation property
INTEGER_VALUE[1]	INTEGER	Numeric value for an implementation property
CHARACTER_VALUE[1]	VARCHAR (256)	Alphanumeric value for an implementation property
COMMENTS	VARCHAR (256)	Comments

Table 91: SQL_IMPL_INFO view of the INFORMATION_SCHEMA

[1]Precisely one of the two columns INTEGER_VALUE or CHARACTER_VALUE has the NULL value depending on whether a numeric or alphanumeric value is available for the implementation attribute.

10.1.30 SQL_LANGUAGES_S

Information on the implemented host languages and embedments. All designations are defined in the SQL standard.

Column name	Data type	Contents
SOURCE	VARCHAR (256)	ISO standard: 'ISO 9075'
SQL_LANGUAGE_YEAR	VARCHAR (256)	Year in which standard appeared: '1989' '1992' '1999' '2008'
CONFORMANCE	VARCHAR (256)	Language scope: '2' 'ENTRY' 'CORE'
INTEGRITY	VARCHAR (256)	'YES' SQL89 “integrity enhancement” implemented NULL value if SQL_LANGUAGE_YEAR '1989'
IMPLEMENTATION	VARCHAR (256)	Specification of an implementationdefined standard, if SOURCE 'ISO 9075'
BINDING_STYLE	VARCHAR (256)	Type of embedment: 'EMBEDDED'
PROGRAMMING_ LANGUAGE	VARCHAR (256)	Supported programming language: 'COBOL'

Table 92: SQL_LANGUAGES_S view of the INFORMATION_SCHEMA

10.1.31 SQL_SIZING

Information on the maximum values of the implementation. All designations are defined in the SQL standard.

Column name	Data type	Contents
SIZING_ID	INTEGER	ID of the maximum value
SIZING_NAME	VARCHAR (256)	Name of the maximum value
SUPPORTED_VALUE	INTEGER	Maximum size of the value: Maximum value 0 — No maximum value or maximum value not known or variable NULL value — Maximum value not important in SESAM/SQL
COMMENTS	VARCHAR (256)	Comments

Table 93: SQL_SIZING view of the INFORMATION_SCHEMA

10.1.32 STOGROUPS

Information on the storage groups that the current authorization identifier can access.

Column name	Data type	Contents
STOGROUP_CATALOG	CHAR (18)	Database name
STOGROUP_NAME	CHAR (18)	Name of the storage group
STOGROUP_OWNER	CHAR (18)	Authorization identifier that owns the storage group
CAT_ID	VARCHAR (4)	BS2000 catalog ID

Table 94: STOGROUPS view of the INFORMATION_SCHEMA

10.1.33 STOGROUP_VOLUME_USAGE

Information on the volumes of the storage groups belonging to the current authorization identifier.

Column name	Data type	Contents
STOGROUP_CATALOG	CHAR (18)	Database name
STOGROUP_NAME	CHAR (18)	Name of the storage group
VOLUME_NAME	CHAR (6)	VSN of the private volumes or PUBLIC
DEVICE_TYPE	VARCHAR (8)	Device type of the private volumes NULL value for PUBLIC
ORDINAL_POSITION	SMALLINT	Sequence number of the private volumes in the storage group (1 for PUBLIC)

Table 95: STOGROUP_VOLUME_USAGE view of the INFORMATION_SCHEMA

10.1.34 SYSTEM_ENTRIES

Information on system entries.

- Current authorization identifier without the CREATE USER privilege:All system entries to the user s own authorization identifier in which the authorization identifier has been entered explicitly.
- Current authorization identifier with CREATE USER privilege but without GRANT authorization:
  All system accesses to authorization identifiers which do not have the CREATE USER privilege or GRANT authorization.
- Current authorization identifier with CREATE USER privilege and with GRANT authorization:
  All system entries

Column name	Data type	Contents
USER_CATALOG	CHAR (18)	Database name
HOST_NAME	CHAR (8)	Host name or *
APPLICATION_NAME	CHAR (8)	Application name or * for UTM system entry Blanks for BS2000 system entry
SYSTEM_USER_NAME	CHAR (8)	BS2000 or UTM user ID
USER_NAME	CHAR (18)	Authorization identifier

Table 96: SYSTEM_ENTRIES view of the INFORMATION_SCHEMA

10.1.35 TABLES

Information on all the tables for which the current authorization identifier has privileges or is the owner.

Column name	Data type	Contents
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table belongs
TABLE_NAME	CHAR (31)	Name of the base table or view
TABLE_TYPE	VARCHAR (18)	BASE TABLE or VIEW

Table 97: TABLES view of the INFORMATION_SCHEMA

10.1.36 TABLE_CONSTRAINTS

Information on integrity constraints on the database schemas that belong to the current authorization identifier.

Column name	Data type	Contents
CONSTRAINT_CATALOG	CHAR (18)	Database name
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the integrity constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the integrity constraint
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table referenced in the integrity constraint belongs
TABLE_NAME	CHAR (31)	Name of the table referenced in the integrity constraint
CONSTRAINT_TYPE	VARCHAR (11)	Type of integrity constraint: FOREIGN KEY UNIQUE PRIMARY KEY CHECK
IS_DEFERRABLE	CHAR (3)	NO
INITIALLY_DEFERRED	CHAR (3)	NO

Table 98: TABLE_CONSTRAINTS view of the INFORMATION_SCHEMA

10.1.37 TABLE_PRIVILEGES

Information on all the table privileges that the current authorization identifier has or which it has granted.

Column name	Data type	Contents
GRANTOR	CHAR (18)	Authorization identifier that granted the privilege or _SYSTEM
GRANTEE	CHAR (18)	Authorization identifier granted the privilege or PUBLIC
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema containing the table to which the privilege applies
TABLE_NAME	CHAR (31)	Name of the table to which the privilege applies
PRIVILEGE_TYPE	CHAR (18)	Privilege type: SELECT INSERT DELETE UPDATE REFERENCES
IS_GRANTABLE	VARCHAR (3)	YES The authorization identifier has GRANT authorization for the privilege NO No GRANT authorization

Table 99: TABLE_PRIVILEGES view of the INFORMATION_SCHEMA

10.1.38 TRANSLATIONS

Information on transliterations which can be executed in the current DBH session.

Column name	Data type	Contents
TRANSLATION_CATALOG	CHAR (18)	Database name
TRANSLATION_SCHEMA	CHAR (31)	INFORMATION_SCHEMA
TRANSLATION_NAME	CHAR (31)	CCS name
SOURCE_CHARACTER_SET_CATALOG	CHAR (18)	Database name
SOURCE_CHARACTER_SET_SCHEMA	CHAR (31)	INFORMATION_SCHEMA
SOURCE_CHARACTER_SET_NAME	CHAR (8)	EBCDIC UTF16
TARGET_CHARACTER_SET_CATALOG	CHAR (18)	Database name
TARGET_CHARACTER_SET_SCHEMA	CHAR (31)	INFORMATION_SCHEMA
TARGET_CHARACTER_SET_NAME	CHAR (8)	EBCDIC UTF16

Table 100: TRANSLATIONS view of the INFORMATION_SCHEMA

10.1.39 USAGE_PRIVILEGES

Information on all the USAGE privileges that the current authorization identifier has or which it has granted.

Column name	Data type	Contents
GRANTOR	CHAR (18)	Authorization identifier that granted the privilege or _SYSTEM
GRANTEE	CHAR (18)	Authorization identifier granted the privilege or PUBLIC
OBJECT_CATALOG	CHAR (18)	Database name
OBJECT_SCHEMA	CHAR (31)	Name of the schema containing the sort sequence or character set to which the privilege applies Blanks for storage group
OBJECT_NAME	CHAR (18)	Name of the storage group, sort sequence or character set to which the privilege applies
OBJECT_TYPE	CHAR (18)	Object to which the privilege applies: STOGROUP CHARACTER SET COLLATION
PRIVILEGE_TYPE	CHAR (18)	USAGE
IS_GRANTABLE	VARCHAR (3)	YES The authorization identifier has GRANT authorization for the privilege NO No GRANT authorization

Table 101: USAGE_PRIVILEGES view of the INFORMATION_SCHEMA

10.1.40 USERS

Information on authorization identifiers.

- Current authorization identifier without the CREATE USER privilege:Own authorization identifier
- Current authorization identifier with CREATE USER privilege but without GRANT authorization:
All authorization identifiers which do not have the CREATE USER privilege or GRANT authorization.
- Current authorization identifier with CREATE USER privilege and with GRANT authorization:
All authorization identifiers

Column name	Data type	Contents
USER_CATALOG	CHAR (18)	Database name
USER_NAME	CHAR (18)	Authorization identifier

Table 102: USERS view of the INFORMATION_SCHEMA

10.1.41 VIEWS

Information on all the views for which the current authorization identifier has privileges.

Column name	Data type	Contents
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the view belongs
TABLE_NAME	CHAR (31)	Name of the view
VIEW_DEFINITION	VARCHAR (32000)	Query expression that defines the view if the current authorization identifier owns the schema NULL value in all other cases
CHECK_OPTION	VARCHAR (8)	NONE No check option set CASCADED Check option set
IS_UPDATABLE	VARCHAR (3)	YES View is updatable NO View is not updatable

Table 103: VIEWS view of the INFORMATION_SCHEMA

10.1.42 VIEW_COLUMN_USAGE

Information on views that reference columns belonging to the current authorization identifier, as well as the names of the corresponding columns.

Column name	Data type	Contents
VIEW_CATALOG	CHAR (18)	Database name
VIEW_SCHEMA	CHAR (31)	Name of the schema to which the view belongs
VIEW_NAME	CHAR (31)	Name of the view
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table that is referenced in the view belongs
TABLE_NAME	CHAR (31)	Name of the table referenced in the view
COLUMN_NAME	CHAR (31)	Column name

Table 104: VIEW_COLUMN_USAGE view of the INFORMATION_SCHEMA

10.1.43 VIEW_ROUTINE_USAGE

Information on the User Defined Functions (UDFs) that belong to the current authorization identifier and are called in views.

Column name	Data type	Contents
TABLE_CATALOG	CHAR(18)	Database name
TABLE_SCHEMA	CHAR(31)	Name of the schema to which the view belongs
TABLE_NAME	CHAR(31)	Name of the view
SPECIFIC_CATALOG	CHAR(18)	Database name
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine

Table 105: VIEW_ROUTINE_USAGE view of the INFORMATION_SCHEMA

10.1.44 VIEW_TABLE_USAGE

Information on the tables that belong to the current authorization identifier and on which view are based.

Column name	Data type	Contents
VIEW_CATALOG	CHAR (18)	Database name
VIEW_SCHEMA	CHAR (31)	Name of the schema to which the view belongs
VIEW_NAME	CHAR (31)	Name of the view
TABLE_CATALOG	CHAR (18)	Database name
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table that is referenced in the view belongs
TABLE_NAME	CHAR (31)	Name of the table referenced in the view

Table 106: VIEW_TABLE_USAGE view of the INFORMATION_SCHEMA

10.2 Views of the SYS_INFO_SCHEMA

The SYS_INFO_SCHEMA contains system-specific data. It provides complete information on all SESAM/SQL objects. The SYS_INFO_SCHEMA may be changed in future versions of SESAM rendering it incompatible.

Only the universal user has access to the views of the SYS_INFO_SCHEMA. The universal user can pass on the SEELCT privilege to other users.
The following table indicates which views of the SYS_INFO_SCHEMA contain information on which database objects.

Object	View name	Information on
Database	SYS_CATALOGS	Database
	SYS_DBC_ENTRIES	all known databases
Schema	SYS_SCHEMATA	Schemas in the database
Table	SYS_TABLES	Tables in the database
	SYS_PARTITIONS	Partitions in the database
	SYS_VIEW_USAGE	Tables on which the views are based
	SYS_CHECK_USAGE	Tables for which check constraints are defined
Column	SYS_COLUMNS	Columns in the database
	SYS_VIEW_USAGE	Columns on which views are based
	SYS_CHECK_USAGE	Columns for which check constraints are defined
Privilege	SYS_PRIVILEGES	Table privileges
	SYS_SPECIAL_PRIVILEGES	Special privileges
	SYS_USAGE_PRIVILEGES	USAGE privileges
	SYS_ROUTINE_PRIVILEGES	Privileges for routines
Index	SYS_INDEXES	Indexes in the database
Integrity constraint	SYS_TABLE_CONSTRAINTS	Integrity constraints
	SYS_REFERENTIAL_CONSTRAINTS	Referential constraints
	SYS_CHECK_CONSTRAINTS	Check constraints
	SYS_UNIQUE_CONSTRAINTS	UNIQUE constraints
Storage group	SYS_STOGROUPS	Storage groups in the database
Space	SYS_SPACES	Spaces
	SYS_SPACE_PROPERTIES	Space properties
Routines	SYS_PARAMETERS	Parameters of routines
	SYS_ROUTINES	Routines

	SYS_ROUTINE_ROUTINE_USAGE	Routines in other routines
	SYS_ROUTINE_USAGE	Tables and columns in routines
	SYS_ROUTINE_ERRORS	Error events in routines
	SYS_VIEW_ROUTINE_USAGE	Routines in views
SQL statements	SYS_DML_RESOURCES	“Costly” DML statements
User	SYS_USERS	Authorization identifier
	SYS_SYSTEM_ENTRIES	System entries
DA-LOG file	SYS_DA_LOGS	DA-LOG files
Media table	SYS_MEDIA_DESCRIPTIONS	Media records of the database-specific files
Recovery unit	SYS_RECOVERY_UNITS	Recovery units for spaces
Locks	SYS_LOCK_CONFLICTS	The lock conflicts which occurred most recently
System environment	SYS_ENVIRONMENT	SESAM/SQL's operating system environment

Table 107: Views of the SYS_INFO_SCHEMA (section 2 of 2)

The views of the SYS_INFO_SCHEMA are described in alphabetical order in the following sections.

10.2.1 SYS_CATALOGS

Information on the database.

Column name	Data type	Contents
CHAR_FORM_OF_USE	CHAR (18)	Name of the coded character set (also: code table) _NONE_ if no coded character set is used.
UNIVERSAL_USER	CHAR (18)	Authorization identifier of the universal user
LOGGING	VARCHAR (3)	Default value for the LOG parameter: YES NO

Table 108: SYS_CATALOGS view of the SYS_INFO_SCHEMA

10.2.2 SYS_CHECK_CONSTRAINTS

Information on check constraints.

Column name	Data type	Contents
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the check constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the check constraint
CHECK_CLAUSE	VARCHAR (32000)	Search condition
CHECK_TYPE_IND	CHAR (1)	Y Check constraint is the NOT NULL constraint N else

Table 109: SYS_CHECK_CONSTRAINTS view of the SYS_INFO_SCHEMA

10.2.3 SYS_CHECK_USAGE

Information on tables and columns of the check constraint.

Column name	Data type	Contents
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the check constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the check constraint
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table or column used by the check constraint belongs
TABLE_NAME	CHAR (31)	Name of the table used in the check constraint or to which the column belongs
COLUMN_NAME	CHAR (31)	Name of the table column used in the check constraint Blanks if information on the table
OBJECT_INDICATOR	CHAR (1)	T Row contains information on table C Row contains information on column
NOT_NULL_COLUMN	CHAR (1)	Y Check constraint forces NOT NULL constraint on the column N else

Table 110: SYS_CHECK_USAGE view of the SYS_INFO_SCHEMA

10.2.4 SYS_COLUMNS

Information on columns in base tables and views of the database.

Column name	Data type	Contents
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table belongs
TABLE_NAME	CHAR (31)	Name of the base table or view
COLUMN_NAME	CHAR (31)	Column name
ORDINAL_POSITION	SMALLINT	Sequence number of the column in the table
COLUMN_DEFAULT	VARCHAR (256)	For base tables only: Default value, as specified in the column definition (e.g. CHAR literal in single quotes) if the current authorization identifier owns the schema TRUNCATED if representation of the default value comprises more than 256 characters and the current authorization identifier owns the schema. The default value cannot be displayed. NULL value in all other cases
IS_NULLABLE	VARCHAR (3)	NO Column cannot accept NULL values under any circumstances YES else
DATA_TYPE	VARCHAR (24)	Data type of the column: CHARACTER CHARACTER VARYING NATIONAL CHAR NATIONAL CHAR VARYING REAL DOUBLE PRECISION FLOAT INTEGER SMALLINT NUMERIC DECIMAL DATE TIME

		TIMESTAMP OLDEST
CHARACTER _MAXIMUM_LENGTH	SMALLINT	Max. length of the column in code units if the data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR, NATIONAL CHAR VARYING or OLDEST NULL value in all other cases
NUMERIC_PRECISION	SMALLINT	Total number of significant digits for numeric data types NULL value in all other cases
NUMERIC_PRECISION _RADIX	SMALLINT	Radix for numeric data types NULL value in all other cases
NUMERIC_SCALE	SMALLINT	Number of digits right of the decimal point for exact numeric data types NULL value in all other cases
DATETIME_PRECISION	SMALLINT	Number of digits right of the decimal point for the data types TIME and TIMESTAMP NULL value in all other cases
The columns OLDEST_DESCRIPTOR* are assigned a value if DATA_TYPE is OLDEST:		
OLDEST_DESCRIPTOR1	CHAR (1)	Y left-aligned N not left-aligned NULL value if DATATYPE is not OLDEST
OLDEST_DESCRIPTOR2	CHAR (1)	Y Fill character N No fill character NULL value if DATATYPE is not OLDEST
OLDEST_DESCRIPTOR3	CHAR (1)	Y Null (0) permitted as value N Null (0) not permitted NULL value if DATATYPE is not OLDEST
OLDEST_DESCRIPTOR4	CHAR (1)	Y Value has arithmetic result N Value does not have arithmetic result

		NULL value if DATATYPE is not OLDEST
COLUMN_DESCRIPTOR1	CHAR (1)	Y Base table column has exactly one single-column index and is not included in a compound index N else
COLUMN_DESCRIPTOR2	CHAR (1)	Y Base table column has exactly one compound index and no single column index N else
COLUMN_DESCRIPTOR3	CHAR (1)	Y Base table column has more than one index N else
COLUMN_DESCRIPTOR4	CHAR (1)	Y Base table column has a CALL DML default value N else
COLUMN_DESCRIPTOR5	CHAR (1)	Y Base table column is a multiple column N else
PK_DISTANCE	SMALLINT	Distance of the column to the start of the primary key NULL value if column is not in the primary key or is not a base table column
SESAM_SAN	CHAR (3)	Symbolic attribute name of the column NULL value if the column is not defined in the base table or SQL table
SESAM_BAN	CHAR (2)	Binary attribute name of the column NULL value if the column is not defined in the base table
SESAM_DEFAULT	CHAR (2)	CALL DML default (with sign if numeric data type) NULL value if the column is not defined in the base table or SQL table
FIRST_OCCURRENCE	SMALLINT	First possible occurrence of a multiple column (for base table = 1) NULL value if the column is not multiple
LAST_OCCURRENCE	SMALLINT	Last possible occurrence of a multiple column NULL value if the column is not multiple

Table 111: SYS_COLUMNS view of the SYS_INFO_SCHEMA

10.2.5 SYS_DA_LOGS

Information on DA-LOG files and/or DA-LOG units in a database.

Column name	Data type	Contents
DALOG_VERSION	INTEGER	Version number of the DA-LOG file
DALOG_SUBNUMBER	INTEGER	Sequence number of the DA-LOG file within the version
DALOG_BLOCKNUMBER	INTEGER	First block in the DA-LOG file for this DA-LOG unit
DALOG_INIT	TIMESTAMP (3)	Time of creation
BLOCK_COUNTER	INTEGER	Last used block in the DA-LOG file
MAX_USER	INTEGER	Max. number of parallel users in the corresponding DBH session
SYSTEM_DATA_BUFFER	INTEGER	Original size of System Data Buffer when writing to DA-LOG file
USER_DATA_BUFFER	INTEGER	Original size of User Data Buffer when writing to DA-LOG file

Table 112: SYS_DA_LOGS view of the SYS_INFO_SCHEMA

10.2.6 SYS_DBC_ENTRIES

Information on all databases which are known to the DBH.

Column name	Data type	Contents
DBC_NUMBER	SMALLINT	DBC ID number
CATALOG_NAME	CHAR (18)	Logical database name
PHYSICAL_NAME	CHAR (18)	Physical database name
USER_ID	CHAR (8)	DB user ID of the database
COPY_NUMBER	CHAR (6)	Version number of the SESAM backup copy of the catalog space, if the database is a SESAM backup copy.
ACCESS_MODE	VARCHAR (5)	Current access mode: READ Permits read access to user data and metadata. WRITE Permits read and write access to user data. Metadata may not be changed. ADMIN Permits read and update access to user data and metadata. REPL A replication is involved. This replication can be accessed in read mode. COPY Permits read access to user data and metadata. The utility statement COPY is permitted.
STATUS	VARCHAR (7)	Database status: ACTIVE The database was opened in the current DBH session. CLOSED The database is closed.

		FREE The database is physically closed and unlocked. LOCKED Because of an SQLSTATE the database is not available in the current DBH session. RECOVER The database is in the state RECOVER. REORG The database is reorganized. REFRESH The database is in the state REFRESH.
STATUS_INFO	VARCHAR(21)	Information on why the database is not available (only when STATUS = LOCKED).
STATUS_TIME	TIMESTAMP(3)	Time the current status was determined

Table 113: SYS_DBC_ENTRIES view of the INFORMATION_SCHEMA

10.2.7 SYS_DML_RESOURCES

Information on “costly” DML statements (in SQL). A DML statement is regarded as costly when the number of logical IOs it triggers and/or its activity time in the DBH is very high compared to other DML statements.

The NUMBER_OF_LOGICAL_IO and ACTIVE_TIME columns in particular contain information relevant to the costs of a statement.

Column name	Data type	Contents
CATALOG_NAME	CHAR (18)	Database name
START_TIME	TIMESTAMP (3)	Start time of the DML statement
END_TIME	TIMESTAMP (3)	End time of the DML statement
HOST_NAME	CHAR (8)	Host name from the identification of the requesting user
APPLICATION_NAME	CHAR (8)	Application name from the identification of the requesting user
CUSTOMER_NAME	CHAR (8)	Name of the requesting user from the identification of the requesting user
CONVERSATION_ID	CHAR (8)	Identification of the requesting user with respect to UTM and SESAM-DBAccess
TAC_NAME	CHAR (8)	Job name of the user ID or name of the program unit which executed the DML statement
MODULE_NAME	CHAR (8)	Name of the compilation unit in which the waiting DML statement was executed
STATEMENT_NAME	VARCHAR (18)	Internal name of the DML statement
STATEMENT_TYPE	VARCHAR (31)	`<type of statement>` (e.g. INSERT)
NUMBER_OF_LOGICAL_IO	INTEGER	Number of logical read and write accesses
NUMBER_OF_PHYSICAL_IO	INTEGER	Number of physical read and write accesses
ELAPSED_TIME	INTEGER	Time which has actually elapsed (milliseconds)
ACTIVE_TIME	INTEGER	Activity time in the DBH (milliseconds)
ACTIVE_TIME_DBH	INTEGER	Activity time in DBH tasks (milliseconds)
ACTIVE_TIME_SVT	INTEGER	Activity time in service tasks (milliseconds)
MEASURE_OF_COSTS	INTEGER	Internal measure of the costs of the application

Table 114: SYS_DML_RESOURCES view of the SYS_INFO_SCHEMA

10.2.8 SYS_ENVIRONMENT

Information on SESAM/SQL's operating system environment Created for maintenance purposes, specifically after a live migration.

Column name	Data type	Contents
INFO_TIMESTAMP	TIMESTAMP (3)	Time of the information (after a live migration, this is the time the live migration took place, otherwise it is a time in the initialization phase of the DBH)
HW_TYPE	CHAR (8)	Hardware type of the current system
OS_VERSION	CHAR (12)	Name and version of the BS2000 operating system
MAIN_MEMORY	INTEGER	Size of the BS2000 main memory in MB
NUMBER_OF_CPU_ MAX	INTEGER	Maximum number of BS2000 CPUs
NUMBER_OF_CPU_ ACTIVE	INTEGER	Number of active BS2000 CPUs
HOST_NAME	CHAR (8)	Host name

Table 115: SYS_ENVIRONMENT view of the SYS_INFO_SCHEMA

10.2.9 SYS_INDEXES

Information on indexes in the database that were created with CREATE INDEX or implicitly with a UNIQUE constraint.

Column name	Data type	Contents
INDEX_SCHEMA	CHAR (31)	Name of the schema to which the index belongs
TABLE_NAME	CHAR (31)	Name of the base table to which the index belongs
COLUMN_NAME	CHAR (31)	Name of the column in the index
INDEX_NAME	CHAR (18)	Name of the index
INDEX_ID	SMALLINT	Identification number of the index
SPACE_NAME	CHAR (18)	Name of the space in which the index is stored
SPACE_ID	SMALLINT	Identification number of the space in which the index is stored
ORDINAL_POSITION	SMALLINT	Position of the column in the index
LENGTH_I	SMALLINT	Total length of the index (in bytes)
LENGTH_C	SMALLINT	Indicates the length (in bytes) to which the column is included in the index
INDEX_DISTANCE	SMALLINT	Distance of the column to the index start
DATA_TYPE_C	VARCHAR (24)	Data type of the column: CHARACTER CHARACTER VARYING NATIONAL CHAR NATIONAL CHAR VARYING REAL DOUBLE PRECISION FLOAT INTEGER SMALLINT NUMERIC DECIMAL DATE TIME TIMESTAMP OLDEST
CONSTRAINT_NAME	CHAR (31)	Name of the UNIQUE constraint if the index is used by a UNIQUE constraint. NULL value in all other cases

STATE	VARCHAR (9)	Status: GENERATED DEFECT
GENERATE_TYPE	VARCHAR (8)	as generated: EXPLICIT IMPLICIT
STATISTICS_INFO	VARCHAR (3)	YES Statistics information exists NO Statistics information does not exist
INDEX_TYPE	VARCHAR (8)	Index type: SINGLE COMPOUND
INDEX_DATE	TIMESTAMP (3)	Time of generation
INDEX_PRIMARY_KEY	CHAR (1)	Y Index is used for the compound key of CALL DML tables N else
TABLE_ID	SMALLINT	Identification number of the base table. When TABLE_ID >= 30720 the table is a partitioned table.

Table 116: SYS_INDEXES view of the SYS_INFO_SCHEMA (section 2 of 2)

10.2.10 SYS_LOCK_CONFLICTS

Information on the lock conflicts which occurred most recently (in chronological order).

Column name	Data type	Contents
TIME_OF_CONFLICT	TIMESTAMP (3)	Time at which the conflict occurred
OBJECT_TYPE	VARCHAR (6)	Type of object to be locked: DBC Database catalog SPACE Space TABLE Base table INDEX Index ROW Row of a base table SI-VAL Value of a secondary index PLAN SQL plan META Metadata area
DBC_NUMBER	SMALLINT	Identification number of the database of the object to be locked (for OBJECT_TYPE not equal to PLAN) NULL value in all other cases
SPACE_ID	SMALLINT	Identification number of the space of the object to be locked (for OBJECT_TYPE= SPACE / TABLE / INDEX / ROW / SI-VAL) NULL value in all other cases
TABLE_ID	SMALLINT	Identification number of the base table of the object to be locked (for OBJECT_TYPE = TABLE / ROW) NULL value in all other cases
INDEX_ID	SMALLINT	Identification number of the index of the object to be locked (for OBJECT_TYPE = INDEX / SI-VAL) NULL value in all other cases
ROW_ID	CHAR (8)	Internal number of the row to be locked (for OBJECT_TYPE = ROW) NULL value in all other cases
SI_VALUE	CHAR (8)	Internal presentation of the key value to be locked (for OBJECT_TYPE = SI-VAL) NULL value in all other cases
PLAN_ID	INTEGER	Internal number of the SQL plan to be locked (for OBJECT_TYPE = PLAN)

		NULL value in all other cases
META_SCHEMA	CHAR (8)	Internal number of the schema in the metadata area which is to be locked (for OBJECT_TYPE = META) NULL value in all other cases
META_TABLE	CHAR (8)	Internal number of the base table in the metadata area which is to be locked (for OBJECT_TYPE = META) NULL value in all other cases
HOST_NAME	CHAR (8)	Host name from the identification of the waiting requesting user
APPLICATION_NAME	CHAR (8)	Application name from the identification of the waiting requesting user
CUSTOMER_NAME	CHAR (8)	Name of the requesting user from the identification of the waiting requesting user
CONVERSATION_ID	CHAR (8)	Identification of the waiting requesting user with respect to UTM and SESAM-DBAccess
TAC_NAME	CHAR (8)	Job name of the user ID or name of the program unit which requested the lock
MODULE_NAME	CHAR (8)	Name of the compilation unit (SQL only) in which the waiting SQL statement was executed NULL value in all other cases
STATEMENT_NAME	VARCHAR (18)	Internal name of the SQL statement which is waiting for the lock NULL value in all other cases
STATEMENT_TYPE	VARCHAR (31)	`<type of statement>` (e.g. INSERT) In the case of SQL statements CALL DML: `<operation code>` In the case of CALL DML statements SYSTEM in the case of system jobs (e.g. administration commands via SEND-MSG)
LOCK_MODE	VARCHAR (31)	Level of the lock request:

		For OBJECT_TYPE = SPACE: NO-UPDATE/SHARED-READ, SHARED-UPDATE/SHARED-READ, EXCLUSIVE-UPDATE/SHARED-READ, EXCLUSIVE-UPDATE/EXCLUSIVE-READ Else: SHARED, EXCLUSIVE
LOCK_TYPE	VARCHAR (8)	Value of the lock request: OBJECT for object lock ADJACENT for environment lock
REQUEST_ANNOUNCED	CHAR (1)	Lock request is to be requested: Y N
LOCKING_OBJECT_TYPE	VARCHAR (6)	Type of object which prevents the lock: DBC Database catalog SPACE Space TABLE Base table INDEX Index ROW Row of a base table SI-VAL Value of a secondary index PLAN SQL plan META Metadata area
LOCKING_HOST_NAME	CHAR (8)	Host name from the identification of the locking requesting user
LOCKING_ APPLICATION_NAME	CHAR (8)	Application name from the identification of the locking requesting user
LOCKING_ CUSTOMER_NAME	CHAR (8)	Name of the requesting user from the identification of the locking requesting user
LOCKING_ CONVERSATION_ID	CHAR (8)	Identification of the locking requesting user with respect to UTM and SESAM-DBAccess
LOCKING_LOCK_MODE	VARCHAR (31)	Level of the object on which the lock failed: NO-UPDATE/SHARED-READ, SHARED-UPDATE/SHARED-READ, EXCLUSIVE-UPDATE/SHARED-READ, EXCLUSIVE-UPDATE/EXCLUSIVE-READ for OBJECT_TYPE = SPACE

		else SHARED, EXCLUSIV

Table 117: SYS_LOCK_CONFLICTS view of the SYS_INFO_SCHEMA

10.2.11 SYS_MEDIA_DESCRIPTIONS

Information on file attributes and media types for database-specific files.

Column name	Data type	Contents
FILE_TYPE	CHAR (6)	File type: DALOG CATLOG PBI CATREC DDLTA
DEVICE_DESCRIPTOR	CHAR (18)	Device type or name of the storage group for the file
MEDIUM	CHAR (4)	DISC
ORDINAL_POSITION	SMALLINT	Sequence number of the entry in the media table
REQUESTS	VARCHAR (3)	YES Volume can be requested at console NO Volume cannot be requested at console
PRIMARY_ALLOC	INTEGER	Primary allocation
SECONDARY_ALLOC	INTEGER	Secondary allocation
SHARABLE	VARCHAR (3)	File sharable: YES NO

Table 118: SYS_MEDIA_DESCRIPTIONS view of the SYS_INFO_SCHEMA

10.2.12 SYS_PARAMETERS

Information on parameters of routines (procedures and UDFs)

Column name	Data type	Contents
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
ORDINAL_POSITION	SMALLINT	Sequence number of the parameter in the routine
PARAMETER_MODE	VARCHAR(5)	IN input parameter OUT output parameter INOUT input and output parameter
PARAMETER_NAME	CHAR(31)	Name of the parameter
DATA_TYPE	VARCHAR(24)	Data type of the column: CHARACTER CHARACTER VARYING NATIONAL CHAR NATIONAL CHAR VARYING REAL DOUBLE PRECISION FLOAT INTEGER SMALLINT NUMERIC DECIMAL DATE TIME TIMESTAMP
CHARACTER_ MAXIMUM_LENGTH	SMALLINT	Max. length of the column in code units if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
NUMERIC_PRECISION	SMALLINT	Total number of significant digits for numeric data types NULL value in all other cases
NUMERIC_PRECISION_ RADIX	SMALLINT	Radix

		for numeric data types NULL value in all other cases
NUMERIC_SCALE	SMALLINT	Number of digits right of the dec. point for exact numeric data types NULL value in all other cases
DATETIME_PRECISION	SMALLINT	Number of digits right of the decimal point for the data types TIME and TIMESTAMP NULL value in all other cases

Table 119: SYS_PARAMETERS view of the SYS_INFO_SCHEMA (section 2 of 2)

10.2.13 SYS_PARTITIONS

Information on table partitions.

Column name	Data type	Contents
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the partitioned table belongs
TABLE_NAME	CHAR (31)	Name of the partitioned table
SERIAL_NUMBER	SMALLINT	Sequence number of the partition
MAX_KEY_VALUE	VARCHAR (32000)	Comparison for the upper partition boundary as specified in the VALUE clause (external presentation)
MAX_NUMBER_ OF_ROWS	INTEGER	Maximum possible number of records in the partition
SPACE_NAME	CHAR (18)	Name of the space in which the partition is stored
SPACE_ID	SMALLINT	Identification number of the space in which the partition is stored
TABLE_ID	SMALLINT	Space-related identification number of the partitioned table
ROW_ID_PREFIX	SMALLINT	Prefix to determine the row number

Table 120: SYS_PARTITIONS view of the SYS_INFO_SCHEMA

10.2.14 SYS_PRIVILEGES

Information on table and column privileges.

Column name	Data type	Contents
GRANTEE	CHAR (18)	Authorization identifier granted the privilege or PUBLIC
TABLE_SCHEMA	CHAR (31)	Name of the schema containing the table or column to which the privilege applies
TABLE_NAME	CHAR (31)	Name of the table to which the privilege applies or for whose column the privilege applies
COLUMN_NAME	CHAR (31)	Name of the column to which the privilege was restricted Blanks if the privilege applies to the whole table
OBJECT_INDICATOR	CHAR (1)	T Row contains information on table C Row contains information on column
PRIVILEGE_TYPE	CHAR (18)	Privilege type: SELECT INSERT DELETE UPDATE REFERENCES
GRANTOR	CHAR (18)	Authorization identifier that granted the privilege or _SYSTEM
IS_GRANTABLE	VARCHAR (3)	YES The authorization identifier has GRANT authorization for the privilege NO No GRANT authorization

Table 121: SYS_PRIVILEGES view of the SYS_INFO_SCHEMA

10.2.15 SYS_RECOVERY_UNITS

Information on recovery units.

Column name	Data type	Contents
SPACE_NAME	CHAR (18)	Name of the space
RECOVERY_TIMESTAMP	TIMESTAMP (3)	Time at which the backup was created
VERSION	INTEGER	Internal number if RECOVERY_TYPE = 'COPY' NULL value in all other cases
VALIDITY	VARCHAR (3)	YES Recovery unit valid for recovery operations up to next recovery unit NO invalid (may however change to YES after a RECOVER statement) NOT invalid (cannot change)
RECOVERY_UNIT_NAME	VARCHAR (54)	File name of the copy if RECOVERY_TYPE = 'COPY' internal number, RECOVERY_TYPE = 'RESTART' or 'REST_TO' NULL value in all other cases
ARCHIVE_COPY_VERSION	VARCHAR (15)	Time of ARCHIVE backup, if MEDIUM = 'TAPE' Time of HSMS backup, if MEDIUM = 'HSMS', 'HSMW' or 'HSMB' NULL value in all other cases
MEDIUM	CHAR (4)	DISC SESAM backup on disk TAPE SESAM backup with ARCHIVE HSMW SESAM backup with HSMS (work file) HSMB SESAM backup with HSMS (additional mirror unit) SRDF SESAM backup with HSMS (SRDF target) if RECOVERY_TYPE = 'COPY'

		NULL value in all other cases	
RECOVERY_TYPE	VARCHAR (7)	Values evaluated by the recovery utility: COPY CREATE RESTART REST_TO (RESTART TO) MARK	
COPY_TYPE	VARCHAR (7)	ONLINE or OFFLINE if RECOVERY_TYPE = 'COPY' NULL value in all other cases	
DALOG_VERSION	INTEGER	Version number of the DA-LOG file	DA-LOG level before the recovery unit is entered
DALOG_SUBNUMBER	INTEGER	Sequence number of the DA-LOG file within the version	
DALOG_BLOCKNUMBER	INTEGER	First block in the DA-LOG file for this DA-LOG unit	
NEXT_DALOG_VERSION	INTEGER	Version number of the DA-LOG file	DA-LOG level after the recovery unit is entered
NEXT_DALOG_SUBNUMBER	INTEGER	Sequence number of the DA-LOG file within the version	
NEXT_DALOG_BLOCKNUMBER	INTEGER	First block in the DA-LOG file for this DA-LOG unit	
LOG_COUNTER	INTEGER	Not currently used	
ARCHIVE_DIRECTORY_NAME	VARCHAR (54)	Name of the ARCHIVE directory, if MEDIUM = 'TAPE' Name of the HSMS archive, if MEDIUM = 'HSMS', 'HSMW', 'HSMB' or 'SRDF' NULL value in all other cases	
ARCHIVE_PBI_VERSION	VARCHAR (15)	Time of ARCHIVE backup of the PBI file, if MEDIUM = 'TAPE' and COPY_TYPE = 'ONLINE' NULL value in all other cases	
PBI_TIMESTAMP	TIMESTAMP (3)	Time at which the PBI file was generated	

PBI_COUNTER	INTEGER	undefined

Table 122: SYS_RECOVERY_UNITS view of the SYS_INFO_SCHEMA (section 3 of 3)

10.2.16 SYS_REFERENTIAL_CONSTRAINTS

Information on referential constraints. The referencing and referenced columns are listed.

Column name	Data type	Contents
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the referential constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the referential constraint
TABLE_NAME	CHAR (31)	Name of the table to which the referential constraint belongs
COLUMN_NAME	CHAR (31)	Name of a referencing column
UNIQUE_CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema of the referenced table
UNIQUE_CONSTRAINT_NAME	CHAR (31)	Name of the UNIQUE or primary key constraint of the referenced table
UNIQUE_CONSTRAINT_TABLE	CHAR (31)	Name of the referenced table
UNIQUE_CONSTRAINT_COLUMN	CHAR (31)	Name of a referenced column
ORDINAL_POSITION	SMALLINT	Position of the column in the referential constraint

Table 123: SYS_REFERENTIAL_CONSTRAINTS view of the SYS_INFO_SCHEMA

10.2.17 SYS_ROUTINES

Information on routines (procedures and UDFs)

Column name	Data type	Contents
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
ROUTINE_TYPE	VARCHAR(28)	PROCEDURE if a procedure FUNCTION if a UDF
DATA_TYPE	VARCHAR(24)	Data type of the return value of a UDF CHARACTER CHARACTER VARYING NATIONAL CHAR NATIONAL CHAR VARYING REAL DOUBLE PRECISION FLOAT INTEGER SMALLINT NUMERIC DECIMAL DATE TIME TIMESTAMP NULL value, if a procedure
CHARACTER_MAXIMUM_LENGTH	SMALLINT	Max. length of the return value in code units if data type is CHARACTER, CHARACTER VARYING, NATIONAL CHAR or NATIONAL CHAR VARYING NULL value in all other cases
NUMERIC_PRECISION	SMALLINT	Total number of significant digits for numeric data types NULL value in all other cases
NUMERIC_PRECISION_RADIX	SMALLINT	Radix for numeric data types

		NULL value in all other cases
NUMERIC_SCALE	SMALLINT	Number of digits right of the decimal point for exact numeric data types NULL value in all other cases
DATETIME_PRECISION	SMALLINT	Number of digits right of the decimal point for the data types TIME and TIMESTAMP NULL value in all other cases
ROUTINE_DEFINITION	VARCHAR(32000)	Text of the routine
SQL_DATA_ACCESS	VARCHAR(17)	CONTAINS SQL if CONTAINS SQL was specified in the definition of the routine READS SQL DATA if READS SQL DATA was specified in the definition of the routine MODIFIES SQL DATA if MODIFIES SQL DATA was specified in the definition of the routine
IS_NULL_CALL	VARCHAR(3)	NO if a UDF NULL value in all other cases
IS_USER_DEFINED_CAST	VARCHAR(3)	NO if a UDF NULL value in all other cases
AS_LOCATOR	VARCHAR(3)	NO if a UDF NULL value in all other cases

Table 124: SYS_ROUTINES view of the SYS_INFO_SCHEMA

10.2.18 SYS_ROUTINE_ERRORS

Information on the the most recent events which were errored or suspected of being errored when routines (procedures and UDFs) were executed. The DEBUG ROUTINE pragma can also result in additional information being logged.

Column name	Data type	Contents
SPECIFIC_CATALOG	CHAR(18)	Name of the database to which the routine belongs
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
START_TIME	TIMESTAMP(3)	Start time of the routine
ERROR_TIME	TIMESTAMP(3)	Time the error event occurred
ERROR_STATE	CHAR(5)	SQLSTATE, if an exception condition occurred Blank, otherwise
ERROR_TEXT	VARCHAR(256)	Message text
LINE_NUMBER	INTEGER	Line number of the errored statement in the text of the routine 0 if the place could not be determined
COLUMN_NUMBER	INTEGER	Column number of the errored statement in the text of the routine 0 if the place could not be determined
HOST_NAME	CHAR(8)	Host name from the identification of the requesting user
APPLICATION_NAME	CHAR(8)	Application name from the identification of the requesting user
CUSTOMER_NAME	CHAR(8)	Name of the requesting user from the identification of the requesting user
CONVERSATION_ID	CHAR(8)	Identification of the requesting user with respect to UTM and SESAM-DBAccess
TAC_NAME	CHAR(8)	Job name of the user ID or name of the program unit which called the routine
MODULE_NAME	CHAR(8)	Name of the compilation unit in which the routine was called
STATEMENT_NAME	VARCHAR(18)	

		Internal name of the SQL statement which called the routine

Table 125: SYS_ROUTINE_ERRORS view of the SYS_INFO_SCHEMA

10.2.19 SYS_ROUTINE_PRIVILEGES

Information on parameters for routines (procedures and UDFs)

Column name	Data type	Contents
GRANTEE	CHAR(18)	Authorization identifier granted the privilege or PUBLIC
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
GRANTOR	CHAR(18)	Authorization identifier which granted the privilege or _SYSTEM
IS_GRANTABLE	VARCHAR(3)	YES The authorization identifier has GRANT authorization for the privilege NO No GRANT authorization

Table 126: SYS_ROUTINE_PRIVILEGES view of the SYS_INFO_SCHEMA

10.2.20 SYS_ROUTINE_ROUTINE_USAGE

Information on the routines (procedures and UDFs) that are called in other routines.

Column name	Data type	Contents
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the calling routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the calling routine
ROUTINE_SCHEMA	CHAR(31)	Name of the schema to which the called routine belongs
ROUTINE_NAME	CHAR(31)	Specific name of the called routine

Table 127: SYS_ROUTINE_ROUTINE_USAGE view of the SYS_INFO_SCHEMA

10.2.21 SYS_ROUTINE_USAGE

Information on the tables and columns which are addressed in routines (procedures and UDFs).

Column name	Data type	Contents
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine
TABLE_SCHEMA	CHAR(31)	Name of the schema to which the table which is addressed in the routine belongs
TABLE_NAME	CHAR(31)	Name of the table used in the routine
COLUMN_NAME	CHAR(31)	Column name used in the routine Blanks if information on a table
OBJECT_INDICATOR	CHAR(1)	T Row contains informations on table C Row contains informations on column

Table 128: SYS_ROUTINE_USAGE view of the SYS_INFO_SCHEMA

10.2.22 SYS_SCHEMATA

Information on the schemas in the database.

Column name	Data type	Contents
SCHEMA_NAME	CHAR (31)	Name of the schema
SCHEMA_OWNER	CHAR (18)	Authorization identifier of the owner

Table 129: SYS_SCHEMATA view of the SYS_INFO_SCHEMA

10.2.23 SYS_SPACES

Information on spaces.

Column name	Data type	Contents
SPACE_NAME	CHAR (18)	Name of the space
SPACE_NAME_SHORT	CHAR (12)	First 12 characters of the space name
SPACE_ID	SMALLINT	Identification number of the space
SPACE_OWNER	CHAR (18)	Authorization identifier that owns the space
STOGROUP_NAME	CHAR (18)	Name of the storage group for the space
PCT_FREE	SMALLINT	Free space reservation in percent
DELTA_STOGROUP	CHAR (1)	Y Space stored on storage group STOGROUP_NAME N Space not yet stored on the storage group STOGROUP_NAME newly assigned with ALTER SPACE
SPACE_DATE	TIMESTAMP (3)	Time of creation or time at which the definition of the tables and indexes on the space were last updated
LOGGING	VARCHAR (3)	YES Logging activated NO Logging deactivated
The data for STOGROUP_NAME and PCT_FREE can be modified with ALTER SPACE; this data is not actually taken into account until RECOVER or REORG is executed.		

Table 130: SYS_SPACES view of the SYS_INFO_SCHEMA

10.2.24 SYS_SPACE_PROPERTIES

Information on space properties. In the case of spaces that are not currently open, only some of the properties will be output.

Column name	Data type	Contents
SPACE_NAME	CHAR (18)	Name of the space
PROPERTY_NAME	CHAR (31)	Name of the space property The following properties are output: • SPACE_ID is the space number determined from the catalog. • SPACE_TIMESTAMP specifies the time at which the space was last modified. • CHECK_TIMESTAMP specifies the time against which the consistency of the space is checked. • MAX_POSSIBLE_PAGE is the highest possible page number of the space 1.073.741.822 (X'3FFFFFFE') for spaces up to 4 TB 16.777.214 (X'00FFFFFE') for spaces up to 64 GB • LAST_USED_PAGE is the last logically occupied 4K-page of the space. • SPACE_LOCK_RECOVER_PENDING specifies whether or not the space could be restored during the repair • SPACE_LOCK_LOAD_RUNNING specifies whether or not the loading of data into a base table of the space using LOAD or IMPORT TABLE has completed. • SPACE_LOCK_IS_COPY specifies whether the space's backup copy is mounted and therefore only read access is permitted. • SPACE_LOCK_IS_REPLICATE specifies whether the space's replicate is mounted and therefore only read access is permitted.

		• SPACE_LOCK_COPY_PENDING specifies whether the space is locked against updates due to a pending COPY cmd. • SPACE_LOCK_CHECK_PENDING specifies whether the integrity constraints have been checked following the loading of data into a base table using LOAD. • SPACE_LOCK_REORG_PENDING specifies that the maximum space size has been reached. Only read accesses and DELETE and REORG SPACE are therefore permitted. • SPACE_FLAG_OPENED specifies whether the space is open. • SPACE_FLAG_MODIFIED specifies whether the space has been modified. • SPACE_FLAG_DEFECT specifies whether the space is defective. • OPEN_TIMESTAMP specifies the time at which the space was last physically opened. The BLOCK_DENSITY_*xx* fields describe the number of blocks whose density factor was found to be greater than (*xx*-10)% and less than or equal to *xx*% since OPEN_TIMESTAMP: • BLOCK_DENSITY_10 • BLOCK_DENSITY_20 • . . . • BLOCK_DENSITY_90 • BLOCK_DENSITY_100
CHARACTER_VALUE	VARCHAR (256)	Value of the space property NULL value If INTEGER_VALUE is a value other than NULL
INTEGER_VALUE	INTEGER	Value of the space property NULL value If CHARACTER_VALUE is a value other than NULL

Table 131: SYS_SPACE_PROPERTIES view of the SYS_INFO_SCHEMA (section 3 of 3)

10.2.25 SYS_SPECIAL_PRIVILEGES

Information on special privileges.

Column name	Data type	Contents
GRANTEE	CHAR (18)	Authorization identifier granted the privilege or PUBLIC
PRIVILEGE_TYPE	CHAR (18)	Privilege type: CREATE USER CREATE SCHEMA CREATE STOGROUP UTILITY
GRANTOR	CHAR (18)	Authorization identifier that granted the privilege or _SYSTEM
IS_GRANTABLE	VARCHAR (3)	YES The authorization identifier has GRANT authorization for the privilege NO No GRANT authorization

Table 132: SYS_SPECIAL_PRIVILEGES view of the SYS_INFO_SCHEMA

10.2.26 SYS_STOGROUPS

Information on the storage groups in the database.

Column name	Data type	Contents
STOGROUP_NAME	CHAR (18)	Name of the storage group
VOLUME_NAME	CHAR (6)	VSN of the private volumes or PUBLIC
STOGROUP_OWNER	CHAR (18)	Authorization identifier that owns the storage group
CAT_ID	VARCHAR (4)	BS2000 catalog ID
DEVICE_TYPE	VARCHAR (8)	Device type of the private volumes NULL value for PUBLIC
ORDINAL_POSITION	SMALLINT	Sequence number of the private volumes in the storage group; 1 for PUBLIC

Table 133: SYS_STOGROUPS view of the SYS_INFO_SCHEMA

10.2.27 SYS_SYSTEM_ENTRIES

Information on the system entries in the database.

Column name	Data type	Contents
HOST_NAME	CHAR (8)	Host name or *
APPLICATION_NAME	CHAR (8)	Application name or * for UTM system entry Blanks for BS2000 system entry
SYSTEM_USER_NAME	CHAR (8)	BS2000 or UTM user ID
USER_NAME	CHAR (18)	Authorization identifier or PUBLIC

Table 134: SYS_SYSTEM_ENTRIES view of the SYS_INFO_SCHEMA

10.2.28 SYS_TABLES

Information on base tables and views in the database.

Column name	Data type	Contents
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table belongs
TABLE_NAME	CHAR (31)	Table name
TABLE_TYPE	VARCHAR (18)	BASE TABLE, VIEW or ABSTRACT TABLE
TABLE_ID	SMALLINT	Identification number of the base table or of the abstract table When TABLE_ID >= 30720 the table is a partitioned table. NULL value for views
SPACE_NAME	CHAR (18)	Name of the space in which the (non-partitioned) table is stored _PARTITIONS_ in the case of a partitioned table NULL value in all other cases
SPACE_ID	SMALLINT	Identification number of the space. In the case of a partitioned table 32767 is output NULL value in all other cases
TABLE_STYLE	VARCHAR (6)	OLDEST CALL DML only table OLD CALL DML/SQL table NEW SQL table NULL value in all other cases
TABLE_DATE	TIMESTAMP (3)	Time of creation or time of last ALTER TABLE
VIEW_DEFINITION	VARCHAR (32000)	Query expression that defines the view for views NULL value in all other cases
TABLE_PRIMARY_KEY	CHAR (1)	S Single primary key

		C Compound primary key NULL value else
CHECK_OPTION	VARCHAR (8)	NONE No check option set CASCADED Check option set NULL value in all other cases
IS_UPDATABLE	VARCHAR (3)	YES View is updatable NO View is not updatable NULL value in all other cases
TEMPORARY_VIEW	VARCHAR (3)	YES View is temporary NO View is permanent NULL value in all other cases

Table 135: SYS_TABLES view of the SYS_INFO_SCHEMA

10.2.29 SYS_TABLE_CONSTRAINTS

Information on all integrity constraints on the tables in the database

Column name	Data type	Contents
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the integrity constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the integrity constraint
CONSTRAINT_TYPE	VARCHAR (11)	Type of integrity constraint: FOREIGN KEY UNIQUE PRIMARY KEY CHECK
TABLE_NAME	CHAR (31)	Name of the table to which the integrity constraint belongs

Table 136: SYS_TABLE_CONSTRAINTS view of the SYS_INFO_SCHEMA

10.2.30 SYS_UNIQUE_CONSTRAINTS

Information on primary key and UNIQUE constraints.

Column name	Data type	Contents
CONSTRAINT_SCHEMA	CHAR (31)	Name of the schema to which the table with the integrity constraint belongs
CONSTRAINT_NAME	CHAR (31)	Name of the integrity constraint
TABLE_NAME	CHAR (31)	Name of the table to which the integrity constraint belongs
COLUMN_NAME	CHAR (31)	Name of a column in the integrity constraint
ORDINAL_POSITION	SMALLINT	Position of the column in the integrity constraint
CONST_TYPE_ID	CHAR (1)	P PRIMARY KEY constraint U UNIQUE constraint
INDEX_NAME	CHAR (18)	Name of the index for the UNIQUE constraint NULL value for primary key constraints

Table 137: SYS_UNIQUE_CONSTRAINTS view of the SYS_INFO_SCHEMA

10.2.31 SYS_USAGE_PRIVILEGES

Information on the USAGE privilege.

Column name	Data type	Contents
GRANTEE	CHAR (18)	Authorization identifier granted the privilege or PUBLIC
OBJECT_SCHEMA	CHAR (31)	Name of the schema containing the sort sequence or character set to which the privilege applies Blanks for storage group
OBJECT_NAME	CHAR (18)	Name of the storage group, sort sequence or character set to which the privilege applies
OBJECT_TYPE	CHAR (18)	Object to which the privilege applies: STOGROUP CHARACTER SET COLLATION
GRANTOR	CHAR (18)	Authorization identifier that granted the privilege or _SYSTEM
IS_GRANTABLE	VARCHAR (3)	YES The authorization identifier has GRANT authorization for the privilege NO No GRANT authorization

Table 138: SYS_USAGE_PRIVILEGES view of the SYS_INFO_SCHEMA

10.2.32 SYS_USERS

Information on all the authorization identifiers in the database.

Column name	Data type	Contents
USER_NAME	CHAR (18)	Authorization identifier
USER_NAME_SHORT	CHAR (10)	First 10 characters of the authorization identifier

Table 139: SYS_USERS view of the SYS_INFO_SCHEMA

10.2.33 SYS_VIEW_USAGE

Information on the tables and columns used by views and temporary views.

Column name	Data type	Contents
VIEW_SCHEMA	CHAR (31)	Name of the schema to which the view belongs
VIEW_NAME	CHAR (31)	Name of the view
TABLE_SCHEMA	CHAR (31)	Name of the schema to which the table or column used by the view belongs
TABLE_NAME	CHAR (31)	Name of the table used in the view or to which the column belongs
COLUMN_NAME	CHAR (31)	Name of the table column used in the view Blanks if information on the table
OBJECT_INDICATOR	CHAR (1)	T Row contains information on table C Row contains information on column
VIEW_COLUMN	CHAR (31)	Name of the view column, if the view is updatable, OBJECT_INDICATOR has the value C and the view column is derived from a table column (in COLUMN_NAME) NULL value in all other cases

Table 140: SYS_VIEW_USAGE view of the SYS_INFO_SCHEMA

10.2.34 SYS_VIEW_ROUTINE_USAGE

Information on the User Defined Functions (UDFs) that are used in views.

Column name	Data type	Contents
TABLE_SCHEMA	CHAR(31)	Name of the schema to which the view belongs
TABLE_NAME	CHAR(31)	Name of the view
SPECIFIC_SCHEMA	CHAR(31)	Name of the schema to which the routine belongs
SPECIFIC_NAME	CHAR(31)	Specific name of the routine

Table 141: SYS_VIEW_ROUTINE_USAGE view of the SYS_INFO_SCHEMA

11 Appendix

This chapter is subdivided into the following parts:

- Overview of the basic syntax elements of SESAM/SQL
- Syntax overview of the CSV file
- SQL keywords

11.1 Syntax elements of SESAM/SQL

The basic syntax elements defined in chapters 3 to 6 of the manual are listed in alphabetical order below.
For these syntax elements, only their name (the name to the left of the definition character "::=") is specified in the syntax of the SQL statements.

> **i** Any square brackets shown here in italics are special characters, and must be specified in the statement.

query_expression ::=
[*query_expression* { UNION [ALL | DISTINCT] | EXCEPT [DISTINCT] }]

{ *select_expression* | TABLE *table* | *join_expression* | (*query_expression*) }

aggregate ::= <{ *value* | NULL },...>

alphanumeric_literal ::=
{ '[*character* ...]'[*separator* ...'[*character* ...]']... |
X'[*hex hex*]...'[*separator* ...'[*hex hex*]...']... }

hex ::= 0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f|A|B|C|D|E|F

annotation ::= /*% *annotation_text* %*/

statement_id ::= *unqual_name*

argumente ::= see *user_defined_function*

expression ::=
{

value |
[*table* .] { *column* | { *column* (*posno*) | *column[posno]* } | { *column* (*min..max*) | *column[min..max]* } } |

function |

subquery |
monadic_op expression |
expression dyadic_op expression |
case_expression |

cast_expression |
(*expression*)

}

column ::= *unqual_name*
posno ::= *unsigned_integer*
min ::= *unsigned_integer*
max ::= *unsigned_integer*

monadic_op ::= { + | - }

dyadic_op ::= { * | / | + | = | || }

authorization_identifier ::= *unqual_name*

letter ::= see *unqual_name*

case_expression ::=

{

CASE

WHEN *search_condition* THEN

...

[ELSE { *expression* | NULL }]
END |

CASE *expressionx*
WHEN *expression1* [, *expression2*] ... THEN { *expression* | NULL }
...

[ELSE { *expression* | NULL }]

END |

NULLIF (*expression1* , *expression2*) |

COALESCE (*expression1* , *expression2,* ... *expressionn*) |

{ MIN | MAX }(*expression1,expression2,* ... , *expressionn*)

}

cast_expression ::= CAST ({ *expression* | NULL } AS *data_type*)

catalog ::= *unqual_name*

data_type ::=

{

[{[*dimension*] | (*dimension*)}] CHAR[ACTER][(*length*)] |

CHAR[ACTER] VARYING(*max*) | VARCHAR(*max*) |

[{[*dimension*] | (*dimension*) }] { NATIONAL CHAR[ACTER] | NCHAR } [(*cu_length* [CODE_UNITS])] |

{ NATIONAL CHAR[ACTER] VARYING | NCHAR VARYING | NVARCHAR } (*cu_max* [CODE_UNITS]) |

[{[*dimension*] | (*dimension*)}]

{

SMALLINT |

INT[EGER] |

NUMERIC [(*precision* [, *scale*])] |DEC[IMAL][(*precision* [, *scale*])] |

REAL |

DOUBLE PRECISION |

FLOAT [(*precision*)] |

DATE |

TIME(3) |

TIMESTAMP(3)

}

}

unqual_base_table_name ::= *unqual_name*

unqual_constraint_name ::= *unqual_name*

unqual_index_name ::= *unqual_name*

unqual_name ::= { *regular_name* | *special_name* }

regular_name ::= *letter* [{ *letter* | *digit* | _ }] ...

special_name ::= " *character...* "

letter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

digit ::= 0|1|2|3|4|5|6|7|8|9

unqual_routine_name ::= *unqual_name*

unqual_schema_name ::= *unqual_name*

unqual_space_name ::= *unqual_name*

unqual_stogroup_name ::= *unqual_name*

error_name ::= *unqual_name*

fixed_pt_number ::= see *numeric_literal*

function ::= { *time_function* | *string_function* | *numeric_function* | *aggregate_function* | *table_function* | *crypto_function* | *user_defined_function* }

integer ::= see *numeric_literal*

floating_pt_number ::= see *numeric_literal*

hex ::= see *alphanumeric_literal*

index ::= see *qualified_name*

integrity_constraint_name ::= see *qualified_name*

join_expression ::=
{

table_specification CROSS JOIN *table_specification* |
table_specification [INNER | { LEFT | RIGHT | FULL } [OUTER]]
JOIN *table_specification* ON *search_condition* |
table_specification UNION JOIN *table_specification* |
(*join_expression*)

}

correlation_name ::= *unqual_name*

crypto_function ::= { ENCRYPT (*expression* , *key*) | DECRYPT (*expression2* , *key* , *data_type*) }

key ::= *expression*

literal ::= { *alphanumeric_literal* | *national_literal* | *special_literal* | *numeric_literal* | *time_literal* }

max ::= *unsigned_integer*

aggregate_function ::= { *operator* ([ALL | DISTINCT] *expression*) | COUNT(*) }

operator ::= {AVG | COUNT | MAX | MIN | SUM }

min ::= *unsigned_integer*

flag ::= see *praedicate*

pattern ::= see *praedicate*

national_literal ::=

{ N'[*character* ...]'[*separator* ...'[*character* ...]'] ... |
NX'[*4hex* ...]'[*separator* ...'[*4hex* ...]'] ...|
U&'[*uc-character* ...]'[*separator...* '[*uc-character* '...] ... [UESCAPE' *esc* '] }

uc-character ::= { *character* | *esc 4hex* | *esc+ 6hex* | *esc esc* }

numeric_function ::=
{

ABS (*expression*) |
CEIL[ING] (*expression*) |
FLOOR (*expression*) |
MOD (*dividend,divisor*) |
SIGN (*expression*) |
TRUNC (*expression*) |
{ CHAR_LENGTH | CHARACTER_LENGTH }

(*expression* [USING { <u>CODE_UNITS</u> | OCTETS }]) |
OCTET_LENGTH (*expression*) |
POSITION (*expression* IN *expression* [USING CODE_UNITS]) |
JULIAN_DAY_OF_DATE (*expression*) |
EXTRACT (*part* FROM *expression*)

}

numeric_literal ::= { *integer* | *fixed_pt_number* | *floating_pt_number* }

integer ::= [{+|-}] *unsigned_integer* [.]

fixed_pt_number ::= [{+|-}] { *unsigned_integer* [. *unsigned_integer*] | *unsigned_integer* . | . *unsigned_integer* }

floating_pt_number ::= *fixed_pt_number* E[{+|-}] *unsigned_integer*

unsigned_integer ::= *digit* ...

operand ::= see *praedicate*

praedicate ::=
{

row comparison_op row |
vector_column comparison_op expression |
row comparison_op { ALL | SOME | ANY } *subquery* |
row [NOT] BETWEEN *row* AND *row* |
vector_column [NOT] BETWEEN *expression* AND *expression* | *expression* IS [NOT] CASTABLE AS *data_type* |

row [NOT] IN { *subquery* | (*row* , ...) } |
vector_column [NOT] IN (*expression* , *expression* , ...) |
operand [NOT] LIKE *pattern* [ESCAPE *character* ...] |
operand [NOT] LIKE_REGEX *regular_expression* [FLAG *flag*] |
expression IS [NOT] NULL |
EXISTS *subquery*

}

row ::= { (*expression* , ...) | *expression* | *subquery* }

vector_column ::= [*table* .]{ *column[min..max]* | *column* (*min..max*) }

comparison_op ::= { = | < | > | <= | >= | <> }

operand ::= *expression*

pattern ::= *expression*

character ::= *expression*

regular_expression ::= *expression*

flag ::= *expression*

pragma ::= --%PRAGMA *pragma_text* , ... *lineend*

qualified_name ::=
{

index |
integrity_constraint_name |

routine |
schema |
space |
stogroup |
table

}

index ::= [[*catalog* .] *unqual_schema_name* .] *unqual_index_name*

integrity_constraint_name ::= [[*catalog* .] *unqual_schema_name* .] *unqual_constraint_name*

routine ::= [[*catalog* .] *unqual_schema_name* .] *unqual_routine_name*

schema ::= [*catalog* .] *unqual_schema_name*

space ::= [*catalog* .] *unqual_space_name*

stogroup ::= [*catalog* .] *unqual_stogroup_name*

table ::=

{

[[*catalog* .] *unqual_schema_name* .] *unqual_base_table_name* |
[[*catalog* .] *unqual_schema_name* .] *unqual_view_name* |
correlation_name

}

regular expression ::= see *praedicate*

regular_name ::= see *unqual_name*

routine ::= see *qualified_name*

routine_parameter ::= *unqual_name*

schema ::= see *qualified_name*

key ::= see *crypto_function*

select_expression ::=

SELECT [ALL | DISTINCT] *select_list*
FROM *table_specification* ,...
[WHERE *search_condition*]
[GROUP BY *column* ,...]
[HAVING *search_condition*]

select_list ::= { * | { *table* .* | *expression* [[AS] *column*] } }

space ::= see *qualified_name*

column ::= see *expression*

col_constraint ::=

{

NOT NULL |

UNIQUE |

PRIMARY KEY |

REFERENCES *table* [(*column*)] |

CHECK (*search_condition*)

}

column_definition ::=
column { *data_type* [*default*] | FOR REF(*table*) }
[[CONSTRAINT *integrity_constraint_name*] *col_constraint*] ...
[*call_dml_clause*]

default ::= DEFAULT
{

alphanumeric_literal |

national_literal |

numeric_literal |

time_literal |

CURRENT_DATE |
CURRENT_TIME(3) |
LOCALTIME(3) |
CURRENT_TIMESTAMP(3) |
LOCALTIMESTAMP(3) |

```
  USER |
  CURRENT_USER |
  SYSTEM_USER |
  NULL |
  REF( tabelle )
}
```

call_dml_clause ::= CALL DML *call_dml_default* [*call_dml_symb_name*]

special_literal ::=

```
{
   CURRENT_CATALOG |
   CURRENT_ISOLATION_LEVEL |
   CURRENT_REFERENCED_CATALOG |
   CURRENT_SCHEMA |
   [CURRENT_]USER |
   SYSTEM_USER
}
```

special_name ::= see *unqual_name*

stogroup ::= see *qualified_name*

search_condition ::= { *praedicate* | *search_condition* { AND | OR } *search_condition* / NOT *search_condition* **|(** *search_condition* **)** }

table ::= see *qualified_name*

table_specification ::=

{

table [[AS] *correlation_name* [(*column* , ...)]] |

subquery [AS] *correlation_name* [(*column* , ...)] |

TABLE([*catalog* .] *table_function*) [WITH ORDINALITY] [[AS] *correlation_name* [(*column* , ...)]] |

join_expression

}

table_constraint ::=
{
 UNIQUE (*column* , . . .) |
 PRIMARY KEY (*column* , . . .) |
 FOREIGN KEY (*column* , . . .) REFERENCES *table* [(*column* , . . .)] |
 CHECK (*search_condition*)
}

table_function ::=
{ CSV ([FILE] *file* DELIMITER *delimiter* [QUOTE *quote*] [ESCAPE *escape*] , *data_type* , . . .) | DEE [()]
}

subquery ::= (*query_expression*)

user_defined_function ::= *unqual_routine_name argumente*

arguments ::= ([*expression* [{ , *expression* } . . .]])

vector_column ::= see *praedicate*

comparison_op ::= see *praedicate*

default ::= see *column_definition*

unsigned_integer ::= see *numeric_literal*

value ::=

{

literal |

: *host_variable* [[INDICATOR] : *indicator_variable*] |

routine_parameter |

local_variable |

```
   ?
}

character ::= see praedicate

string_function ::=
 {
   SUBSTRING ( expression FROM startposition [FOR substring_length ][USING CODE_UNITS]) |
   TRANSLATE ( expression USING [[ catalog .]INFORMATION_SCHEMA.] transname [DEFAULT
   character ] [ ,length ]) |
   TRIM ([[LEADING |TRAILING | BOTH] [ character ] FROM] expression ) |
   LOWER ( expression ) |
   UPPER ( expression ) |
   HEX_OF_VALUE ( expression2 ) |
   VALUE_OF_HEX ( expression3 , data_type ) |
   REP_OF_VALUE ( expression2 ) |
   VALUE_OF_REP ( expression3 , data_type ) |
   COLLATE ( expression USING { DUCET_WITH_VARS | DUCET_NO_VARS } [ ,length ]) |
   NORMALIZE ( expression [ ,NFC | NFD [, length ]])
 }

character ::= expression
length ::= unsigned_integer

time_function ::=
{
   CURRENT_DATE |
   CURRENT_TIME(3) |
   LOCALTIME(3) |
   CURRENT_TIMESTAMP(3) |
   LOCALTIMESTAMP(3) |
   DATE_OF_JULIAN_DAY ( expression )
}

time_literal ::=
{
   DATE ' year-month-day ' |
   TIME ' hour:minute:second '
   TIMESTAMP' jahr-monat-tag hour:minute:second '
}
```

row `::=` see *praedicate*

digit `::=` see *unqual_name*

11.2 Syntax overview of the CSV file

Comments in this syntax presentation are enclosed in double quotes ".

CSV_file_format ::=
[[*CSV_record*] *CSV_record_separator* [[*CSV_record*] *CSV_record_separator*]...] [*CSV_record*]

CSV_record ::=
{ [*CSV_field CSV_delimiter*]... *CSV_non-empty_field* | *CSV_field CSV_delimiter* [*CSV_field CSV_delimiter*
]... }

CSV_record_separator ::=
{ X'04' "(EBCDIC control character NEL)" |
X'0D' "(EBCDIC control character CR)" |
X'15' "(EBCDIC control character LF)" |
X'25' "(EBCDIC control character)" |
"End of record of a BS2000SAM file" }

CSV_field ::= { *CSV non-empty field* | "(leer)" }

CSV_non-empty_field ::= { *CSV plain field* | *CSV quoted field* }

CSV_plain_field ::= *CSV_plain_intro* [*CSV_plain_part*]...

CSV_plain_intro ::= { *CSV_escape_sequence* | "all characters except *CSV_record_separator*, *CSV_delimiter*, *CSV_escape* und *CSV_quote*" }

CSV_plain_part ::= { *CSV_escape_sequence* | "all characters except *CSV_record_separator*, *CSV_delimiter* und *CSV_escape*" }

CSV_quoted_field ::= *CSV_quote* [*CSV_quoted part*]... *CSV_quote*

CSV_quoted_part ::=
{ *CSV_quote CSV_quote* | *CSV_escape_sequence* | "End of record of a BS2000SAM file" | "all characters except *CSV_quote* und *CSV_escape*" }

CSV_escape_sequence ::=
{ *CSV_escape CSV_record_separator* | *CSV_escape CSV_delimiter* | *CSV_escape CSV_quote* | *CSV_escape CSV_escape* }

CSV_delimiter ::= *character*

CSV_quote ::= *character*

CSV_escape ::= *character*

For details of *CSV_delimiter*, *CSV_quote*, and *CSV_escape*, see also the syntax description of the CSV function on "CSV() - Reading a BS2000 file as a table".

11.3 SQL keywords

In SESAM/SQL there are words that are reserved as keywords for SQL and utility statements. These keywords cannot be used as the names of views, tables, columns, etc. in SQL or utility statements or when working with the utility monitor, unless you specify the keyword in the form of a special name.

The synonym processing feature provided by the ESQL precompiler is a convenient way of replacing keywords or of redefining names.

You can use the precompiler option SOURCE-PROPERTIES to set the ESQL-DIALECT parameter to ISO, OLD or ALL-FEATURES. This determines whether the SQL dialect ISO, OLD or FILL has to be used.

The table below lists the reserved keywords and indicates the SQL dialect in which they are valid.

Keyword	ISO	OLD	FULL
ABS	X		X
ABSOLUTE	X		X
ACTION	X		X
ADD	X		X
ALL	X	X	X
ALLOCATE	X		X
ALTER	X		X
AND	X	X	X
ANY	X	X	X
ARE	X		X
AS	X	X	X
ASC	X	X	X
ASSERTION	X		X
AT	X		X
AUTHORIZATION	X	X	X
AVG	X	X	X
BEGIN	X	X	X
BETWEEN	X	X	X
BIT	X		X
BIT_LENGTH	X		X
BLOB	X		X
BOTH	X		X
BY	X	X	X

CALL	X		X
CASCADE	X		X
CASCADED	X		X
CASE	X		X
CAST	X		X
CATALOG	X		X
CEIL	X		X
CEILING	X		X
CHAR	X	X	X
CHARACTER	X	X	X
CHARACTER_LENGTH	X		X
CHAR_LENGTH	X		X
CHECK	X	X	X
CLOSE	X	X	X
COALESCE	X		X
COLLATE	X		X
COLLATION	X		X
COLUMN	X		X
COMMIT	X	X	X
CONNECT	X		X
CONNECTION	X		X
CONSTRAINT	X		X
CONSTRAINTS	X		X
CONTINUE	X	X	X
CONVERT	X		X
COPY			X
CORRESPONDING	X		X
COUNT	X	X	X
CREATE	X	X	X
CROSS	X		X
CURRENT	X	X	X
CURRENT_CATALOG	X		X

CURRENT_DATE	X		X
CURRENT_ISOLATION_LEVEL			X
CURRENT_REFERENCED_CATALOG			X
CURRENT_SCHEMA	X		X
CURRENT_TIME	X		X
CURRENT_TIMESTAMP	X		X
CURRENT_USER	X		X
CURSOR	X	X	X
DATA	X		X
DATE	X		X
DATE_OF_JULIAN_DAY			X
DAY	X		X
DEALLOCATE	X		X
DEC	X	X	X
DECIMAL	X	X	X
DECLARE	X	X	X
DECRYPT			X
DEFAULT	X	X	X
DEFERRABLE	X		X
DEFERRED	X		X
DELETE	X	X	X
DESC	X	X	X
DESCRIBE	X		X
DESCRIPTOR	X		X
DIAGNOSTICS	X		X
DIRECTORY			X
DISCONNECT	X		X
DISTINCT	X	X	X
DOMAIN	X		X
DOUBLE	X	X	X
DROP	X		X

ELSE	X		X
ENCRYPT			X
END	X	X	X
END-EXEC		X	
ESCAPE	X	X	X
EXCEPT	X		X
EXCEPTION	X		X
EXEC	X	X	X
EXECUTE	X	X	X
EXISTS	X	X	X
EXP	X		X
EXPORT			X
EXTERNAL	X		X
EXTRACT	X		X
FALSE	X		X
FETCH	X	X	X
FIRST	X	X	X
FLOAT	X	X	X
FLOOR	X		X
FOR	X	X	X
FORCED			X
FOREIGN	X	X	X
FOUND	X	X	X
FROM	X	X	X
FULL	X		X
GET	X		X
GLOBAL	X		X
GO	X	X	X
GOTO	X	X	X
GRANT	X	X	X
GROUP	X	X	X

HAVING	X	X	X
HEX_OF_VALUE			X
HOLD	X		X
HOUR	X		X
IDENTITY	X		X
IMMEDIATE	X	X	X
IMPORT			X
IN	X	X	X
INDICATOR	X	X	X
INITIALLY	X		X
INNER	X		X
INPUT	X		X
INSERT	X	X	X
INT	X	X	X
INTEGER	X	X	X
INTERSECT	X		X
INTERVAL	X		X
INTO	X	X	X
IS	X	X	X
ISOLATION	X		X
JOIN	X		X
JULIAN_DAY_OF_DATE			X
KEY	X	X	X
LANGUAGE	X	X	X
LAST	X	X	X
LEADING	X		X
LEFT	X		X
LEVEL	X	X	X
LIKE	X	X	X

LIKE_REGEX	X		X
LN	X		X
LOAD			X
LOCAL	X		X
LOCALTIME	X		X
LOCALTIMESTAMP	X		X
LOWER	X		X
MATCH	X		X
MATCHED	X		X
MAX	X	X	X
MERGE	X		X
MIGRATE			X
MIN	X	X	X
MINUTE	X		X
MOD	X		X
MODIFY			X
MODULE	X	X	X
MONTH	X		X
NAMES	X		X
NATIONAL	X		X
NATURAL	X		X
NCHAR	X		X
NEW	X		X
NEXT	X	X	X
NO	X		X
NORMALIZE	X		X
NOT	X	X	X
NULL	X	X	X
NULLIF	X		X
NUMERIC	X	X	X
NVARCHAR			X

OCTET_LENGTH	X		X
OF	X	X	X
OLD	X		X
ON	X	X	X
ONLY	X	X	X
OPEN	X	X	X
OPTION	X	X	X
OR	X	X	X
ORDER	X	X	X
OUTER	X		X
OUTPUT	X		X
OVERLAPS	X		X
PARTIAL	X		X
PERMIT		X	X
POSITION	X		X
POWER	X		X
PRECISION	X	X	X
PREPARE	X	X	X
PRESERVE	X		X
PRIMARY	X	X	X
PRIOR	X	X	X
PRIVILEGES	X	X	X
PROCEDURE	X	X	X
PUBLIC	X	X	X
READ	X	X	X
REAL	X	X	X
RECOVER			X
REF	X		X
REFERENCES	X	X	X
REFRESH			X

RELATIVE	X		X
REORG			X
REP_OF_VALUE			X
RESTORE		X	X
RESTRICT	X		X
RETURN	X	X	X
REVOKE	X		X
RIGHT	X		X
ROLLBACK	X	X	X
ROWS	X		X
SCHEMA	X	X	X
SCOPE	X		X
SCROLL	X	X	X
SECOND	X		X
SECTION	X	X	X
SELECT	X	X	X
SESSION	X		X
SESSION_USER	X		X
SET	X	X	X
SIGN			X
SIZE	X		X
SMALLINT	X	X	X
SOME	X	X	X
SORTED			X
SQL	X	X	X
SQLCODE		X	X
SQLERROR	X	X	X
SQLSTATE	X		X
SQRT	X		X
STORE		X	X
SUBSTRING	X		X
SUM	X	X	X

SYSTEM	X		X
SYSTEM_USER	X		X
TABLE	X	X	X
TEMPORARY	X	X	X
THEN	X		X
TIME	X		X
TIMESTAMP	X		X
TIMEZONE_HOUR	X		X
TIMEZONE_MINUTE	X		X
TO	X	X	X
TRAILING	X		X
TRANSACTION	X	X	X
TRANSLATE	X		X
TRANSLATION	X		X
TRIM	X		X
TRUE	X		X
TRUNC			X
UESCAPE	X		X
UNION	X	X	X
UNIQUE	X	X	X
UNKNOWN	X		X
UNLOAD			X
UPDATE	X	X	X
UPPER	X		X
USAGE	X		X
USER	X	X	X
USING	X	X	X
VALUE	X		X
VALUES	X	X	X
VALUE_OF_HEX			X

VALUE_OF_REP			X
VARCHAR	X		X
VARYING	X		X
VIEW	X	X	X
WHEN	X		X
WHENEVER	X	X	X
WHERE	X	X	X
WITH	X	X	X
WITHOUT	X		X
WORK	X	X	X
WRITE	X	X	X
YEAR	X		X
ZONE	X		X

Table 142: SESAM/SQL keywords

12 Related publications

You will find the manuals on the internet at http://bs2manuals.ts.fujitsu.com. You can order printed versions of manuals which are displayed with the order number.

SESAM/SQL-Server (BS2000)
Core manual
User Guide

SESAM/SQL-Server (BS2000)
SQL Reference Manual Part 2: Utilities
User Guide

SESAM/SQL-Server (BS2000)
CALL-DM Applications
User Guide

SESAM/SQL-Server (BS2000)
Database Operation
User Guide

SESAM/SQL-Server (BS2000)
Utility Monitor
User Guide

SESAM/SQL-Server (BS2000)
Messages
User Guide

SESAM/SQL-Server (BS2000)
Performance
User Guide

ESQL-COBOL (BS2000)
ESQL-COBOL for SESAM/SQL-Server
User Guide

SESAM-DBAccess

Server-Installation, Administration (available on the manual server only)